

# Ch. 1 - Concepts de base

Langage C / C++

R. Absil

Haute École Bruxelles-Brabant  
École supérieure d'Informatique



6 octobre 2021

# Table des matières

1 Introduction

2 Compilation

3 Types de base

4 Constantes

5 Chaînes de caractères

6 Entrées / sorties

7 Expressions

# Table des matières

1 Introduction

2 Compilation

3 Types de base

4 Constantes

5 Chaînes de caractères

6 Entrées / sorties

7 Expressions

# Table des matières

- 1 Introduction
- 2 Compilation
- 3 Types de base
- 4 Constantes
- 5 Chaînes de caractères
- 6 Entrées / sorties
- 7 Expressions

# Table des matières

- 1 Introduction
- 2 Compilation
- 3 Types de base
- 4 Constantes
- 5 Chaînes de caractères
- 6 Entrées / sorties
- 7 Expressions

# Table des matières

- 1 Introduction
- 2 Compilation
- 3 Types de base
- 4 Constantes
- 5 Chaînes de caractères
- 6 Entrées / sorties
- 7 Expressions

# Table des matières

- 1 Introduction
- 2 Compilation
- 3 Types de base
- 4 Constantes
- 5 Chaînes de caractères
- 6 Entrées / sorties
- 7 Expressions

# Table des matières

- 1 Introduction
- 2 Compilation
- 3 Types de base
- 4 Constantes
- 5 Chaînes de caractères
- 6 Entrées / sorties
- 7 Expressions



# Introduction

# Différences majeures entre Java et C / C++

- La sortie du compilateur est un fichier binaire directement exécuté par le processeur
- *Tout* est passé par valeur, en l'absence de spécifications explicites
- Un `new` en C++ (ou son équivalent en C) est significativement plus difficile à gérer
  - Mais de bonnes pratiques permettent d'éviter les ennuis
- Une plus grande liberté de programmation en C++
  - Surcharge d'opérateur
  - Conversions définies par l'utilisateur
  - Une bonne hygiène de programmation est nécessaire
- Le polymorphisme n'est pas « activé » par défaut
- Pas de franches relations d'héritage entre classes aux fonctionnalités « similaires »
- Mécanisme de types génériques implémenté à la compilation

# Différences majeures entre Java et C / C++

- La sortie du compilateur est un fichier binaire directement exécuté par le processeur
- *Tout* est passé par valeur, en l'absence de spécifications explicites
- Un `new` en C++ (ou son équivalent en C) est significativement plus difficile à gérer
  - Mais de bonnes pratiques permettent d'éviter les ennuis
- Une plus grande liberté de programmation en C++
  - Surcharge d'opérateur
  - Conversions définies par l'utilisateur
  - Une bonne hygiène de programmation est nécessaire
- Le polymorphisme n'est pas « activé » par défaut
- Pas de franches relations d'héritage entre classes aux fonctionnalités « similaires »
- Mécanisme de types génériques implémenté à la compilation

# Différences majeures entre Java et C / C++

- La sortie du compilateur est un fichier binaire directement exécuté par le processeur
- *Tout* est passé par valeur, en l'absence de spécifications explicites
- Un `new` en C++ (ou son équivalent en C) est significativement plus difficile à gérer
  - Mais de bonnes pratiques permettent d'éviter les ennuis
- Une plus grande liberté de programmation en C++
  - Surcharge d'opérateur
  - Conversions définies par l'utilisateur
  - Une bonne hygiène de programmation est nécessaire
- Le polymorphisme n'est pas « activé » par défaut
- Pas de franches relations d'héritage entre classes aux fonctionnalités « similaires »
- Mécanisme de types génériques implémenté à la compilation

# Différences majeures entre Java et C / C++

- La sortie du compilateur est un fichier binaire directement exécuté par le processeur
- *Tout* est passé par valeur, en l'absence de spécifications explicites
- Un `new` en C++ (ou son équivalent en C) est significativement plus difficile à gérer
  - Mais de bonnes pratiques permettent d'éviter les ennuis
- Une plus grande liberté de programmation en C++
  - Surcharge d'opérateur
  - Conversions définies par l'utilisateur
  - Une bonne hygiène de programmation est nécessaire
- Le polymorphisme n'est pas « activé » par défaut
- Pas de franches relations d'héritage entre classes aux fonctionnalités « similaires »
- Mécanisme de types génériques implémenté à la compilation

# Différences majeures entre Java et C / C++

- La sortie du compilateur est un fichier binaire directement exécuté par le processeur
- *Tout* est passé par valeur, en l'absence de spécifications explicites
- Un `new` en C++ (ou son équivalent en C) est significativement plus difficile à gérer
  - Mais de bonnes pratiques permettent d'éviter les ennuis
- Une plus grande liberté de programmation en C++
  - Surcharge d'opérateur
  - Conversions définies par l'utilisateur
  - Une bonne hygiène de programmation est nécessaire
- Le polymorphisme n'est pas « activé » par défaut
- Pas de franches relations d'héritage entre classes aux fonctionnalités « similaires »
- Mécanisme de types génériques implémenté à la compilation

# Différences majeures entre Java et C / C++

- La sortie du compilateur est un fichier binaire directement exécuté par le processeur
- *Tout* est passé par valeur, en l'absence de spécifications explicites
- Un `new` en C++ (ou son équivalent en C) est significativement plus difficile à gérer
  - Mais de bonnes pratiques permettent d'éviter les ennuis
- Une plus grande liberté de programmation en C++
  - Surcharge d'opérateur
  - Conversions définies par l'utilisateur
  - Une bonne hygiène de programmation est nécessaire
- Le polymorphisme n'est pas « activé » par défaut
- Pas de franches relations d'héritage entre classes aux fonctionnalités « similaires »
- Mécanisme de types génériques implémenté à la compilation

# Différences majeures entre Java et C / C++

- La sortie du compilateur est un fichier binaire directement exécuté par le processeur
- *Tout* est passé par valeur, en l'absence de spécifications explicites
- Un `new` en C++ (ou son équivalent en C) est significativement plus difficile à gérer
  - Mais de bonnes pratiques permettent d'éviter les ennuis
- Une plus grande liberté de programmation en C++
  - Surcharge d'opérateur
  - Conversions définies par l'utilisateur
  - Une bonne hygiène de programmation est nécessaire
- Le polymorphisme n'est pas « activé » par défaut
- Pas de franches relations d'héritage entre classes aux fonctionnalités « similaires »
- Mécanisme de types génériques implémenté à la compilation



# Différences majeures entre Java et C / C++

- La sortie du compilateur est un fichier binaire directement exécuté par le processeur
- *Tout* est passé par valeur, en l'absence de spécifications explicites
- Un `new` en C++ (ou son équivalent en C) est significativement plus difficile à gérer
  - Mais de bonnes pratiques permettent d'éviter les ennuis
- Une plus grande liberté de programmation en C++
  - Surcharge d'opérateur
  - Conversions définies par l'utilisateur
  - Une bonne hygiène de programmation est nécessaire
- Le polymorphisme n'est pas « activé » par défaut
- Pas de franches relations d'héritage entre classes aux fonctionnalités « similaires »
- Mécanisme de types génériques implémenté à la compilation

# Différences majeures entre Java et C / C++

- La sortie du compilateur est un fichier binaire directement exécuté par le processeur
- *Tout* est passé par valeur, en l'absence de spécifications explicites
- Un `new` en C++ (ou son équivalent en C) est significativement plus difficile à gérer
  - Mais de bonnes pratiques permettent d'éviter les ennuis
- Une plus grande liberté de programmation en C++
  - Surcharge d'opérateur
  - Conversions définies par l'utilisateur
  - Une bonne hygiène de programmation est nécessaire
- Le polymorphisme n'est pas « activé » par défaut
- Pas de franches relations d'héritage entre classes aux fonctionnalités « similaires »
- Mécanisme de types génériques implémenté à la compilation

# Différences majeures entre Java et C / C++

- La sortie du compilateur est un fichier binaire directement exécuté par le processeur
- *Tout* est passé par valeur, en l'absence de spécifications explicites
- Un `new` en C++ (ou son équivalent en C) est significativement plus difficile à gérer
  - Mais de bonnes pratiques permettent d'éviter les ennuis
- Une plus grande liberté de programmation en C++
  - Surcharge d'opérateur
  - Conversions définies par l'utilisateur
  - Une bonne hygiène de programmation est nécessaire
- Le polymorphisme n'est pas « activé » par défaut
- Pas de franches relations d'héritage entre classes aux fonctionnalités « similaires »
- Mécanisme de types génériques implémenté à la compilation

# Différences majeures entre Java et C / C++

- La sortie du compilateur est un fichier binaire directement exécuté par le processeur
- *Tout* est passé par valeur, en l'absence de spécifications explicites
- Un `new` en C++ (ou son équivalent en C) est significativement plus difficile à gérer
  - Mais de bonnes pratiques permettent d'éviter les ennuis
- Une plus grande liberté de programmation en C++
  - Surcharge d'opérateur
  - Conversions définies par l'utilisateur
  - Une bonne hygiène de programmation est nécessaire
- Le polymorphisme n'est pas « activé » par défaut
- Pas de franches relations d'héritage entre classes aux fonctionnalités « similaires »
- Mécanisme de types génériques implémenté à la compilation

# Différences C et C++

- C : Langage procédural uniquement.
  - Pas d'orienté objet et concepts associés.
- Pas de surdéfinition de fonctions
- Pas d'inférence de type.
- Pas de surcharge d'opérateur.
- Pas de références.
- Pas d'exceptions.
- Pas de patrons.
- Pas d'espaces de noms.
- Pas de conversions définies par l'utilisateur.
- Pas de conteneurs standards.
- Pas de `string`

# Différences C et C++

- C : Langage procédural uniquement.
  - Pas d'orienté objet et concepts associés.
- Pas de surdéfinition de fonctions
- Pas d'inférence de type.
- Pas de surcharge d'opérateur.
- Pas de références.
- Pas d'exceptions.
- Pas de patrons.
- Pas d'espaces de noms.
- Pas de conversions définies par l'utilisateur.
- Pas de conteneurs standards.
- Pas de `string`

# Différences C et C++

- C : Langage procédural uniquement.
  - Pas d'orienté objet et concepts associés.
- Pas de surdéfinition de fonctions
- Pas d'inférence de type.
- Pas de surcharge d'opérateur.
- Pas de références.
- Pas d'exceptions.
- Pas de patrons.
- Pas d'espaces de noms.
- Pas de conversions définies par l'utilisateur.
- Pas de conteneurs standards.
- Pas de `string`

# Différences C et C++

- C : Langage procédural uniquement.
  - Pas d'orienté objet et concepts associés.
- Pas de surdéfinition de fonctions
- Pas d'inférence de type.
- Pas de surcharge d'opérateur.
- Pas de références.
- Pas d'exceptions.
- Pas de patrons.
- Pas d'espaces de noms.
- Pas de conversions définies par l'utilisateur.
- Pas de conteneurs standards.
- Pas de `string`



# Différences C et C++

- C : Langage procédural uniquement.
  - Pas d'orienté objet et concepts associés.
- Pas de surdéfinition de fonctions
- Pas d'inférence de type.
- Pas de surcharge d'opérateur.
- Pas de références.
- Pas d'exceptions.
- Pas de patrons.
- Pas d'espaces de noms.
- Pas de conversions définies par l'utilisateur.
- Pas de conteneurs standards.
- Pas de `string`

# Différences C et C++

- C : Langage procédural uniquement.
  - Pas d'orienté objet et concepts associés.
- Pas de surdéfinition de fonctions
- Pas d'inférence de type.
- Pas de surcharge d'opérateur.
- Pas de références.
- Pas d'exceptions.
- Pas de patrons.
- Pas d'espaces de noms.
- Pas de conversions définies par l'utilisateur.
- Pas de conteneurs standards.
- Pas de `string`

# Différences C et C++

- C : Langage procédural uniquement.
  - Pas d'orienté objet et concepts associés.
- Pas de surdéfinition de fonctions
- Pas d'inférence de type.
- Pas de surcharge d'opérateur.
- Pas de références.
- Pas d'exceptions.
- Pas de patrons.
- Pas d'espaces de noms.
- Pas de conversions définies par l'utilisateur.
- Pas de conteneurs standards.
- Pas de `string`

# Différences C et C++

- C : Langage procédural uniquement.
  - Pas d'orienté objet et concepts associés.
- Pas de surdéfinition de fonctions
- Pas d'inférence de type.
- Pas de surcharge d'opérateur.
- Pas de références.
- Pas d'exceptions.
- Pas de patrons.
- Pas d'espaces de noms.
- Pas de conversions définies par l'utilisateur.
- Pas de conteneurs standards.
- Pas de `string`

# Différences C et C++

- C : Langage procédural uniquement.
  - Pas d'orienté objet et concepts associés.
- Pas de surdéfinition de fonctions
- Pas d'inférence de type.
- Pas de surcharge d'opérateur.
- Pas de références.
- Pas d'exceptions.
- Pas de patrons.
- Pas d'espaces de noms.
- Pas de conversions définies par l'utilisateur.
- Pas de conteneurs standards.
- Pas de `string`

# Différences C et C++

- C : Langage procédural uniquement.
  - Pas d'orienté objet et concepts associés.
- Pas de surdéfinition de fonctions
- Pas d'inférence de type.
- Pas de surcharge d'opérateur.
- Pas de références.
- Pas d'exceptions.
- Pas de patrons.
- Pas d'espaces de noms.
- Pas de conversions définies par l'utilisateur.
- Pas de conteneurs standards.
- Pas de `string`

# Différences C et C++

- C : Langage procédural uniquement.
  - Pas d'orienté objet et concepts associés.
- Pas de surdéfinition de fonctions
- Pas d'inférence de type.
- Pas de surcharge d'opérateur.
- Pas de références.
- Pas d'exceptions.
- Pas de patrons.
- Pas d'espaces de noms.
- Pas de conversions définies par l'utilisateur.
- Pas de conteneurs standards.
- Pas de `string`

# Différences C et C++

- C : Langage procédural uniquement.
  - Pas d'orienté objet et concepts associés.
- Pas de surdéfinition de fonctions
- Pas d'inférence de type.
- Pas de surcharge d'opérateur.
- Pas de références.
- Pas d'exceptions.
- Pas de patrons.
- Pas d'espaces de noms.
- Pas de conversions définies par l'utilisateur.
- Pas de conteneurs standards.
- Pas de `string`



# Concepts inexistants en C

<code>try</code>	<code>catch</code>	<code>throw</code>	<code>typeid</code>
<code>const_cast</code>	<code>dynamic_cast</code>	<code>reinterpret_cast</code>	<code>static_cast</code>
<code>namespace</code>	<code>friend</code>	<code>operator</code>	<code>mutable</code>
<code>template</code>	<code>typename</code>	<code>new</code>	<code>delete</code>
<code>protected</code>	<code>public</code>	<code>private</code>	<code>class</code>
<code>this</code>	<code>explicit</code>	<code>using</code>	<code>virtual</code>
<code>export</code>	<code>concept</code>		

# C : le programme minimal

## ■ Fichier `hello-world.c`

```
1  #include <stdio.h>
2
3  int main() //variantes avec arguments en ligne de commande
4  {
5      printf("Hello_World!\n");
6  }
```

- `#include` permet d'importer les fichiers nécessaires
- La fonction `int main()` est l'unique point d'entrée du programme
- `printf` est une fonction permettant d'imprimer en console
  - On peut spécifier des « formats » pour formater l'affichage
- Syntaxe de commentaires comme en Java
- Compilation avec `gcc -o sortie monfichier.c`

# Compilation

# Production d'un exécutable en trois temps

## 1 Précompilation

- Directives précédées de # (préprocesseur)
  - Inclusions des fichiers d'entête
- Immédiats convertis en numériques, etc.
- Produit un programme en C / C++ pur
- Sortie : fichier texte

## 2 Compilation

- Analyse lexicale, syntaxique et sémantique
- Traduction du programme en langage machine
- Produit des fichiers objets

## 3 Édition des liens

- Lie les références et appels entre eux
- Ajoute les bibliothèques externes
- Produit un fichier exécutable ou une bibliothèque directement utilisable

# Production d'un exécutable en trois temps

## 1 Précompilation

- Directives précédées de # (préprocesseur)
  - Inclusions des fichiers d'entête
- Immédiats convertis en numériques, etc.
- Produit un programme en C / C++ pur
- Sortie : fichier texte

## 2 Compilation

- Analyse lexicale, syntaxique et sémantique
- Traduction du programme en langage machine
- Produit des fichiers objets

## 3 Édition des liens

- Lie les références et appels entre eux
- Ajoute les bibliothèques externes
- Produit un fichier exécutable ou une bibliothèque directement utilisable

# Production d'un exécutable en trois temps

## 1 Précompilation

- Directives précédées de # (préprocesseur)
  - Inclusions des fichiers d'entête
- Immédiats convertis en numériques, etc.
- Produit un programme en C / C++ pur
- Sortie : fichier texte

## 2 Compilation

- Analyse lexicale, syntaxique et sémantique
- Traduction du programme en langage machine
- Produit des fichiers objets

## 3 Édition des liens

- Lie les références et appels entre eux
- Ajoute les bibliothèques externes
- Produit un fichier exécutable ou une bibliothèque directement utilisable

# Production d'un exécutable en trois temps

## 1 Précompilation

- Directives précédées de # (préprocesseur)
  - Inclusions des fichiers d'entête
- Immédiats convertis en numériques, etc.
- Produit un programme en C / C++ pur
- Sortie : fichier texte

## 2 Compilation

- Analyse lexicale, syntaxique et sémantique
- Traduction du programme en langage machine
- Produit des fichiers objets

## 3 Édition des liens

- Lie les références et appels entre eux
- Ajoute les bibliothèques externes
- Produit un fichier exécutable ou une bibliothèque directement utilisable

# Production d'un exécutable en trois temps

## 1 Précompilation

- Directives précédées de # (préprocesseur)
  - Inclusions des fichiers d'entête
- Immédiats convertis en numériques, etc.
- Produit un programme en C / C++ pur
- Sortie : fichier texte

## 2 Compilation

- Analyse lexicale, syntaxique et sémantique
- Traduction du programme en langage machine
- Produit des fichiers objets

## 3 Édition des liens

- Lie les références et appels entre eux
- Ajoute les bibliothèques externes
- Produit un fichier exécutable ou une bibliothèque directement utilisable



# Production d'un exécutable en trois temps

## 1 Précompilation

- Directives précédées de # (préprocesseur)
  - Inclusions des fichiers d'entête
- Immédiats convertis en numériques, etc.
- Produit un programme en C / C++ pur
- Sortie : fichier texte

## 2 Compilation

- Analyse lexicale, syntaxique et sémantique
- Traduction du programme en langage machine
- Produit des fichiers objets

## 3 Édition des liens

- Lie les références et appels entre eux
- Ajoute les bibliothèques externes
- Produit un fichier exécutable ou une bibliothèque directement utilisable

# Production d'un exécutable en trois temps

## 1 Précompilation

- Directives précédées de # (préprocesseur)
  - Inclusions des fichiers d'entête
- Immédiats convertis en numériques, etc.
- Produit un programme en C / C++ pur
- Sortie : fichier texte

## 2 Compilation

- Analyse lexicale, syntaxique et sémantique
- Traduction du programme en langage machine
- Produit des fichiers objets

## 3 Édition des liens

- Lie les références et appels entre eux
- Ajoute les bibliothèques externes
- Produit un fichier exécutable ou une bibliothèque directement utilisable

# Production d'un exécutable en trois temps

## 1 Précompilation

- Directives précédées de # (préprocesseur)
  - Inclusions des fichiers d'entête
- Immédiats convertis en numériques, etc.
- Produit un programme en C / C++ pur
- Sortie : fichier texte

## 2 Compilation

- Analyse lexicale, syntaxique et sémantique
- Traduction du programme en langage machine
- Produit des fichiers objets

## 3 Édition des liens

- Lie les références et appels entre eux
- Ajoute les bibliothèques externes
- Produit un fichier exécutable ou une bibliothèque directement utilisable

# Production d'un exécutable en trois temps

## 1 Précompilation

- Directives précédées de # (préprocesseur)
  - Inclusions des fichiers d'entête
- Immédiats convertis en numériques, etc.
- Produit un programme en C / C++ pur
- Sortie : fichier texte

## 2 Compilation

- Analyse lexicale, syntaxique et sémantique
- Traduction du programme en langage machine
- Produit des fichiers objets

## 3 Édition des liens

- Lie les références et appels entre eux
- Ajoute les bibliothèques externes
- Produit un fichier exécutable ou une bibliothèque directement utilisable

# Production d'un exécutable en trois temps

## 1 Précompilation

- Directives précédées de # (préprocesseur)
  - Inclusions des fichiers d'entête
- Immédiats convertis en numériques, etc.
- Produit un programme en C / C++ pur
- Sortie : fichier texte

## 2 Compilation

- Analyse lexicale, syntaxique et sémantique
- Traduction du programme en langage machine
- Produit des fichiers objets

## 3 Édition des liens

- Lie les références et appels entre eux
- Ajoute les bibliothèques externes
- Produit un fichier exécutable ou une bibliothèque directement utilisable

# Production d'un exécutable en trois temps

## 1 Précompilation

- Directives précédées de # (préprocesseur)
  - Inclusions des fichiers d'entête
- Immédiats convertis en numériques, etc.
- Produit un programme en C / C++ pur
- Sortie : fichier texte

## 2 Compilation

- Analyse lexicale, syntaxique et sémantique
- Traduction du programme en langage machine
- Produit des fichiers objets

## 3 Édition des liens

- Lie les références et appels entre eux
- Ajoute les bibliothèques externes
- Produit un fichier exécutable ou une bibliothèque directement utilisable

# Production d'un exécutable en trois temps

## 1 Précompilation

- Directives précédées de # (préprocesseur)
  - Inclusions des fichiers d'entête
- Immédiats convertis en numériques, etc.
- Produit un programme en C / C++ pur
- Sortie : fichier texte

## 2 Compilation

- Analyse lexicale, syntaxique et sémantique
- Traduction du programme en langage machine
- Produit des fichiers objets

## 3 Édition des liens

- Lie les références et appels entre eux
- Ajoute les bibliothèques externes
- Produit un fichier exécutable ou une bibliothèque directement utilisable

# Production d'un exécutable en trois temps

## 1 Précompilation

- Directives précédées de # (préprocesseur)
  - Inclusions des fichiers d'entête
- Immédiats convertis en numériques, etc.
- Produit un programme en C / C++ pur
- Sortie : fichier texte

## 2 Compilation

- Analyse lexicale, syntaxique et sémantique
- Traduction du programme en langage machine
- Produit des fichiers objets

## 3 Édition des liens

- Lie les références et appels entre eux
- Ajoute les bibliothèques externes
- Produit un fichier exécutable ou une bibliothèque directement utilisable



# Production d'un exécutable en trois temps

## 1 Précompilation

- Directives précédées de # (préprocesseur)
  - Inclusions des fichiers d'entête
- Immédiats convertis en numériques, etc.
- Produit un programme en C / C++ pur
- Sortie : fichier texte

## 2 Compilation

- Analyse lexicale, syntaxique et sémantique
- Traduction du programme en langage machine
- Produit des fichiers objets

## 3 Édition des liens

- Lie les références et appels entre eux
- Ajoute les bibliothèques externes
- Produit un fichier exécutable ou une bibliothèque directement utilisable

# Illustration préprocesseur

## ■ Fichier `preproc.c`

```
1  ...
2
3  extern int printf (const char *__restrict __format, ...);
4
5  ...
6
7  int main()
8  {
9      printf("Hello_World!\n");
10 }
```

■ Obtenu avec `gcc -E hello-world.c`

■ Assembleur obtenu avec `-S`

■ Fichier `hello-world.s`

# Compilation séquentielle

- En C / C++, un programme est lu séquentiellement de haut en bas
  - L'ordre des déclarations importe
- Motive l'usage des headers
  - On déclare une fois pour tout le programme
  - On ne veut pas inclure deux fois un header
- Les headers ne sont pas « compilés directement » : ils sont inclus (copier / coller textuel) à l'étape de précompilation
- Peut poser problème en cas de dépendances cycliques
  - Usage d'instructions de compilation conditionnelles
  - Usage de pointeurs pour éviter une taille infinie (cf. Ch. 2)
- Point d'entrée : `main`
- Options de compilations additionnelles
  - `-Wpedantic, -O3, -fno-stack-protector, -l, etc.`

# Compilation séquentielle

- En C / C++, un programme est lu séquentiellement de haut en bas
  - L'ordre des déclarations importe
- Motive l'usage des headers
  - On déclare une fois pour tout le programme
  - On ne veut pas inclure deux fois un header
- Les headers ne sont pas « compilés directement » : ils sont inclus (copier / coller textuel) à l'étape de précompilation
- Peut poser problème en cas de dépendances cycliques
  - Usage d'instructions de compilation conditionnelles
  - Usage de pointeurs pour éviter une taille infinie (cf. Ch. 2)
- Point d'entrée : `main`
- Options de compilations additionnelles
  - `-Wpedantic, -O3, -fno-stack-protector, -l, etc.`

# Compilation séquentielle

- En C / C++, un programme est lu séquentiellement de haut en bas
  - L'ordre des déclarations importe
- Motive l'usage des headers
  - On déclare une fois pour tout le programme
  - On ne veut pas inclure deux fois un header
- Les headers ne sont pas « compilés directement » : ils sont inclus (copier / coller textuel) à l'étape de précompilation
- Peut poser problème en cas de dépendances cycliques
  - Usage d'instructions de compilation conditionnelles
  - Usage de pointeurs pour éviter une taille infinie (cf. Ch. 2)
- Point d'entrée : `main`
- Options de compilations additionnelles
  - `-Wpedantic, -O3, -fno-stack-protector, -l, etc.`

# Compilation séquentielle

- En C / C++, un programme est lu séquentiellement de haut en bas
  - L'ordre des déclarations importe
- Motive l'usage des headers
  - On déclare une fois pour tout le programme
  - On ne veut pas inclure deux fois un header
- Les headers ne sont pas « compilés directement » : ils sont inclus (copier / coller textuel) à l'étape de précompilation
- Peut poser problème en cas de dépendances cycliques
  - Usage d'instructions de compilation conditionnelles
  - Usage de pointeurs pour éviter une taille infinie (cf. Ch. 2)
- Point d'entrée : `main`
- Options de compilations additionnelles
  - `-Wpedantic, -O3, -fno-stack-protector, -l, etc.`

# Compilation séquentielle

- En C / C++, un programme est lu séquentiellement de haut en bas
  - L'ordre des déclarations importe
- Motive l'usage des headers
  - On déclare une fois pour tout le programme
  - On ne veut pas inclure deux fois un header
- Les headers ne sont pas « compilés directement » : ils sont inclus (copier / coller textuel) à l'étape de précompilation
- Peut poser problème en cas de dépendances cycliques
  - Usage d'instructions de compilation conditionnelles
  - Usage de pointeurs pour éviter une taille infinie (cf. Ch. 2)
- Point d'entrée : `main`
- Options de compilations additionnelles
  - `-Wpedantic, -O3, -fno-stack-protector, -l, etc.`

# Compilation séquentielle

- En C / C++, un programme est lu séquentiellement de haut en bas
  - L'ordre des déclarations importe
- Motive l'usage des headers
  - On déclare une fois pour tout le programme
  - On ne veut pas inclure deux fois un header
- Les headers ne sont pas « compilés directement » : ils sont inclus (copier / coller textuel) à l'étape de précompilation
- Peut poser problème en cas de dépendances cycliques
  - Usage d'instructions de compilation conditionnelles
  - Usage de pointeurs pour éviter une taille infinie (cf. Ch. 2)
- Point d'entrée : `main`
- Options de compilations additionnelles
  - `-Wpedantic, -O3, -fno-stack-protector, -l, etc.`



# Compilation séquentielle

- En C / C++, un programme est lu séquentiellement de haut en bas
  - L'ordre des déclarations importe
- Motive l'usage des headers
  - On déclare une fois pour tout le programme
  - On ne veut pas inclure deux fois un header
- Les headers ne sont pas « compilés directement » : ils sont inclus (copier / coller textuel) à l'étape de précompilation
- Peut poser problème en cas de dépendances cycliques
  - Usage d'instructions de compilation conditionnelles
  - Usage de pointeurs pour éviter une taille infinie (cf. Ch. 2)
- Point d'entrée : `main`
- Options de compilations additionnelles
  - `-Wpedantic, -O3, -fno-stack-protector, -l, etc.`

# Compilation séquentielle

- En C / C++, un programme est lu séquentiellement de haut en bas
  - L'ordre des déclarations importe
- Motive l'usage des headers
  - On déclare une fois pour tout le programme
  - On ne veut pas inclure deux fois un header
- Les headers ne sont pas « compilés directement » : ils sont inclus (copier / coller textuel) à l'étape de précompilation
- Peut poser problème en cas de dépendances cycliques
  - Usage d'instructions de compilation conditionnelles
  - Usage de pointeurs pour éviter une taille infinie (cf. Ch. 2)
- Point d'entrée : `main`
- Options de compilations additionnelles
  - `-Wpedantic, -O3, -fno-stack-protector, -l, etc.`

# Compilation séquentielle

- En C / C++, un programme est lu séquentiellement de haut en bas
  - L'ordre des déclarations importe
- Motive l'usage des headers
  - On déclare une fois pour tout le programme
  - On ne veut pas inclure deux fois un header
- Les headers ne sont pas « compilés directement » : ils sont inclus (copier / coller textuel) à l'étape de précompilation
- Peut poser problème en cas de dépendances cycliques
  - Usage d'instructions de compilation conditionnelles
  - Usage de pointeurs pour éviter une taille infinie (cf. Ch. 2)
- Point d'entrée : `main`
- Options de compilations additionnelles
  - `-Wpedantic, -O3, -fno-stack-protector, -l, etc.`

# Compilation séquentielle

- En C / C++, un programme est lu séquentiellement de haut en bas
  - L'ordre des déclarations importe
- Motive l'usage des headers
  - On déclare une fois pour tout le programme
  - On ne veut pas inclure deux fois un header
- Les headers ne sont pas « compilés directement » : ils sont inclus (copier / coller textuel) à l'étape de précompilation
- Peut poser problème en cas de dépendances cycliques
  - Usage d'instructions de compilation conditionnelles
  - Usage de pointeurs pour éviter une taille infinie (cf. Ch. 2)
- Point d'entrée : `main`
- Options de compilations additionnelles
  - `-Wpedantic, -O3, -fno-stack-protector, -l, etc.`

# Compilation séquentielle

- En C / C++, un programme est lu séquentiellement de haut en bas
  - L'ordre des déclarations importe
- Motive l'usage des headers
  - On déclare une fois pour tout le programme
  - On ne veut pas inclure deux fois un header
- Les headers ne sont pas « compilés directement » : ils sont inclus (copier / coller textuel) à l'étape de précompilation
- Peut poser problème en cas de dépendances cycliques
  - Usage d'instructions de compilation conditionnelles
  - Usage de pointeurs pour éviter une taille infinie (cf. Ch. 2)
- Point d'entrée : `main`
- Options de compilations additionnelles
  - `-Wpedantic, -O3, -fno-stack-protector, -l, etc.`

# Compilation séquentielle

- En C / C++, un programme est lu séquentiellement de haut en bas
  - L'ordre des déclarations importe
- Motive l'usage des headers
  - On déclare une fois pour tout le programme
  - On ne veut pas inclure deux fois un header
- Les headers ne sont pas « compilés directement » : ils sont inclus (copier / coller textuel) à l'étape de précompilation
- Peut poser problème en cas de dépendances cycliques
  - Usage d'instructions de compilation conditionnelles
  - Usage de pointeurs pour éviter une taille infinie (cf. Ch. 2)
- Point d'entrée : `main`
- Options de compilations additionnelles
  - `-Wpedantic, -O3, -fno-stack-protector, -l, etc.`

# Exemple : ordre importe

## ■ Fichier `order.c`

```
1 void g();
2
3 int main()
4 {
5     f();
6     g();
7 }
8
9 void f()
10 {
11     printf("Coucou");
12 }
13
14 void g()
15 {
16     printf("Beuh");
17 }
```

# Exemple : inclusion multiple

## ■ Fichier `incl-mult.c`

```
1  #include <stdio.h>
2  #include "incl-mult2.c"
3
4  void g()
5  {
6      printf("Beuh\n");
7  }
8
9  int main()
10 {
11     f();
12 }
```

## ■ Fichier `incl-mult2.c`

```
1  #include <stdio.h>
2  #include "incl-mult.c"
3
4  void f()
5  {
6      printf("Coucou\n");
7      g();
8  }
```

## ■ Solutions : `incl-mult-fixed*`



# Solution

- Séparer les déclarations et leur implémentation
- Utiliser `#ifndef` / `#define`

```
1  #ifndef MULT_1
2  #define MULT_1
3      void g ();
4  #endif
```

```
1  #ifndef MULT_2
2  #define MULT_2
3      void f ();
4  #endif
```

```
1  #include "incl-mult-fixed.h"
2  #include "incl-mult-fixed-2.h"
3
4  int main ()
5  {
6      f ();
7      g ();
8  }
```

- On met les `#include` là où il le faut

# Types de base

# Notion de type (1/2)

- Définit le codage en mémoire
- Définit à quoi correspond un motif

## Exemple

■ Motif binaire 010011001

## Notion de type (1/2)

- Définit le codage en mémoire
- Définit à quoi correspond un motif

### Exemple

■ Motif binaire 010011001

# Notion de type (1/2)

- Définit le codage en mémoire
- Définit à quoi correspond un motif

## Exemple

- Motif binaire 010011001
  - Nombre 77
  - Adresse 0x4D
  - Caractère ASCII 'M'

# Notion de type (1/2)

- Définit le codage en mémoire
- Définit à quoi correspond un motif

## Exemple

- **Motif binaire** 010011001
  - Nombre 77
  - Adresse 0x4D
  - Caractère ASCII 'M'

# Notion de type (1/2)

- Définit le codage en mémoire
- Définit à quoi correspond un motif

## Exemple

- Motif binaire 010011001
  - Nombre 77
  - Adresse 0x4D
  - Caractère ASCII 'M'

# Notion de type (1/2)

- Définit le codage en mémoire
- Définit à quoi correspond un motif

## Exemple

- Motif binaire 010011001
  - Nombre 77
  - Adresse 0x4D
  - Caractère ASCII 'M'



# Notion de type (1/2)

- Définit le codage en mémoire
- Définit à quoi correspond un motif

## Exemple

- Motif binaire 010011001
  - Nombre 77
  - Adresse 0x4D
  - Caractère ASCII 'M'

# Notion de type (2/2)

## ■ Deux sortes de types

- 1 Scalaire : contient une unique valeur à un moment donné
- 2 Structuré : contient plusieurs valeurs
  - Potentiellement de différents types
  - Motif à découper

## Exemple

- Scalaire : `int`, `double`, `char`, `float*`, `enum`
- Structuré : tableaux, classes, structures C, `string` (C++), etc.

# Notion de type (2/2)

## ■ Deux sortes de types

- 1 **Scalaire** : contient une unique valeur à un moment donné
- 2 **Structuré** : contient plusieurs valeurs
  - Potentiellement de différents types
  - Motif à découper

## Exemple

- Scalaire : `int`, `double`, `char`, `float*`, `enum`
- Structuré : tableaux, classes, structures C, `string` (C++), etc.

# Notion de type (2/2)

## ■ Deux sortes de types

- 1 Scalaire : contient une unique valeur à un moment donné
- 2 Structuré : contient plusieurs valeurs
  - Potentiellement de différents types
  - Motif à découper

## Exemple

- Scalaire : `int`, `double`, `char`, `float*`, `enum`
- Structuré : tableaux, classes, structures C, `string` (C++), etc.

# Notion de type (2/2)

## ■ Deux sortes de types

- 1 Scalaire : contient une unique valeur à un moment donné
- 2 Structuré : contient plusieurs valeurs
  - Potentiellement de différents types
  - Motif à découper

## Exemple

- Scalaire : `int`, `double`, `char`, `float*`, `enum`
- Structuré : tableaux, classes, structures C, `string` (C++), etc.

# Notion de type (2/2)

## ■ Deux sortes de types

- 1 Scalaire : contient une unique valeur à un moment donné
- 2 Structuré : contient plusieurs valeurs
  - Potentiellement de différents types
  - Motif à découper

## Exemple

- Scalaire : `int`, `double`, `char`, `float*`, `enum`
- Structuré : tableaux, classes, structures C, `string` (C++), etc.

## Notion de type (2/2)

### ■ Deux sortes de types

- 1 Scalaire : contient une unique valeur à un moment donné
- 2 Structuré : contient plusieurs valeurs
  - Potentiellement de différents types
  - Motif à découper

### Exemple

- Scalaire : `int`, `double`, `char`, `float*`, `enum`
- Structuré : tableaux, classes, structures C, `string` (C++), etc.

# Type incomplet

- Dans une vaste majorité des cas, le compilateur requiert qu'un type  $T$  soit *complet*
  - Si on a besoin de sa taille (pour allouer un  $T$ )
  - Si on a besoin de connaître les opérations que l'on peut réaliser sur un  $T$  (appel de fonction en C++)
- Les types suivants sont *incomplets*
  - `void` et ses versions cv-qualifiées
  - une classe déclarée mais non définie (forward declaration)
  - un tableau de taille inconnue (avec `extern`)
  - un tableau d'éléments de types incomplets
- Parfois, certaines portions de code requièrent d'utiliser temporairement un type incomplet
  - Par ex., dans le cadre d'inclusions multiples



# Type incomplet

- Dans une vaste majorité des cas, le compilateur requiert qu'un type  $T$  soit *complet*
  - Si on a besoin de sa taille (pour allouer un  $T$ )
  - Si on a besoin de connaître les opérations que l'on peut réaliser sur un  $T$  (appel de fonction en C++)
- Les types suivants sont *incomplets*
  - `void` et ses versions cv-qualifiées
  - une classe déclarée mais non définie (forward declaration)
  - un tableau de taille inconnue (avec `extern`)
  - un tableau d'éléments de types incomplets
- Parfois, certaines portions de code requièrent d'utiliser temporairement un type incomplet
  - Par ex., dans le cadre d'inclusions multiples

# Type incomplet

- Dans une vaste majorité des cas, le compilateur requiert qu'un type  $T$  soit *complet*
  - Si on a besoin de sa taille (pour allouer un  $T$ )
  - Si on a besoin de connaître les opérations que l'on peut réaliser sur un  $T$  (appel de fonction en C++)
- Les types suivants sont *incomplets*
  - `void` et ses versions cv-qualifiées
  - une classe déclarée mais non définie (forward declaration)
  - un tableau de taille inconnue (avec `extern`)
  - un tableau d'éléments de types incomplets
- Parfois, certaines portions de code requièrent d'utiliser temporairement un type incomplet
  - Par ex., dans le cadre d'inclusions multiples

# Type incomplet

- Dans une vaste majorité des cas, le compilateur requiert qu'un type  $T$  soit *complet*
  - Si on a besoin de sa taille (pour allouer un  $T$ )
  - Si on a besoin de connaître les opérations que l'on peut réaliser sur un  $T$  (appel de fonction en C++)
- Les types suivants sont *incomplets*
  - `void` et ses versions cv-qualifiées
  - une classe déclarée mais non définie (forward declaration)
  - un tableau de taille inconnue (avec `extern`)
  - un tableau d'éléments de types incomplets
- Parfois, certaines portions de code requièrent d'utiliser temporairement un type incomplet
  - Par ex., dans le cadre d'inclusions multiples

# Type incomplet

- Dans une vaste majorité des cas, le compilateur requiert qu'un type  $T$  soit *complet*
  - Si on a besoin de sa taille (pour allouer un  $T$ )
  - Si on a besoin de connaître les opérations que l'on peut réaliser sur un  $T$  (appel de fonction en C++)
- Les types suivants sont *incomplets*
  - `void` et ses versions cv-qualifiées
  - une classe déclarée mais non définie (forward declaration)
  - un tableau de taille inconnue (avec `extern`)
  - un tableau d'éléments de types incomplets
- Parfois, certaines portions de code requièrent d'utiliser temporairement un type incomplet
  - Par ex., dans le cadre d'inclusions multiples

# Type incomplet

- Dans une vaste majorité des cas, le compilateur requiert qu'un type  $T$  soit *complet*
  - Si on a besoin de sa taille (pour allouer un  $T$ )
  - Si on a besoin de connaître les opérations que l'on peut réaliser sur un  $T$  (appel de fonction en C++)
- Les types suivants sont *incomplets*
  - `void` et ses versions cv-qualifiées
  - une classe déclarée mais non définie (forward declaration)
  - un tableau de taille inconnue (avec `extern`)
  - un tableau d'éléments de types incomplets
- Parfois, certaines portions de code requièrent d'utiliser temporairement un type incomplet
  - Par ex., dans le cadre d'inclusions multiples

# Type incomplet

- Dans une vaste majorité des cas, le compilateur requiert qu'un type  $T$  soit *complet*
  - Si on a besoin de sa taille (pour allouer un  $T$ )
  - Si on a besoin de connaître les opérations que l'on peut réaliser sur un  $T$  (appel de fonction en C++)
- Les types suivants sont *incomplets*
  - `void` et ses versions cv-qualifiées
  - une classe déclarée mais non définie (forward declaration)
  - un tableau de taille inconnue (avec `extern`)
  - un tableau d'éléments de types incomplets
- Parfois, certaines portions de code requièrent d'utiliser temporairement un type incomplet
  - Par ex., dans le cadre d'inclusions multiples

# Type incomplet

- Dans une vaste majorité des cas, le compilateur requiert qu'un type  $T$  soit *complet*
  - Si on a besoin de sa taille (pour allouer un  $T$ )
  - Si on a besoin de connaître les opérations que l'on peut réaliser sur un  $T$  (appel de fonction en C++)
- Les types suivants sont *incomplets*
  - `void` et ses versions cv-qualifiées
  - une classe déclarée mais non définie (forward declaration)
  - un tableau de taille inconnue (avec `extern`)
  - un tableau d'éléments de types incomplets
- Parfois, certaines portions de code requièrent d'utiliser temporairement un type incomplet
  - Par ex., dans le cadre d'inclusions multiples

# Type incomplet

- Dans une vaste majorité des cas, le compilateur requiert qu'un type  $T$  soit *complet*
  - Si on a besoin de sa taille (pour allouer un  $T$ )
  - Si on a besoin de connaître les opérations que l'on peut réaliser sur un  $T$  (appel de fonction en C++)
- Les types suivants sont *incomplets*
  - `void` et ses versions cv-qualifiées
  - une classe déclarée mais non définie (forward declaration)
  - un tableau de taille inconnue (avec `extern`)
  - un tableau d'éléments de types incomplets
- Parfois, certaines portions de code requièrent d'utiliser temporairement un type incomplet
  - Par ex., dans le cadre d'inclusions multiples



# Type incomplet

- Dans une vaste majorité des cas, le compilateur requiert qu'un type  $T$  soit *complet*
  - Si on a besoin de sa taille (pour allouer un  $T$ )
  - Si on a besoin de connaître les opérations que l'on peut réaliser sur un  $T$  (appel de fonction en C++)
- Les types suivants sont *incomplets*
  - `void` et ses versions cv-qualifiées
  - une classe déclarée mais non définie (forward declaration)
  - un tableau de taille inconnue (avec `extern`)
  - un tableau d'éléments de types incomplets
- Parfois, certaines portions de code requièrent d'utiliser temporairement un type incomplet
  - Par ex., dans le cadre d'inclusions multiples

# Les types de base et le standard C / C++

- Le standard C / C++ impose peu de contraintes de taille
  - Un `int` sur une machine n'a peut-être pas la même taille qu'un `int` sur une autre
- Contraintes de bornes pour les types numériques
  - Un `int` est généralement 32 bits, un `double` sur 64 bits
  - Bornes dans `limits.h`
- La représentation des négatifs n'est pas fixée
  - Souvent, complément à deux, little indian
- La représentation des flottants n'est pas fixée
  - Souvent, norme IEEE-754 64 bit
- Il convient de savoir avec quoi on travaille
- Référence complète : [https://en.cppreference.com/w/c/language/arithmetic\\_types](https://en.cppreference.com/w/c/language/arithmetic_types)

# Les types de base et le standard C / C++

- Le standard C / C++ impose peu de contraintes de taille
  - Un `int` sur une machine n'a peut-être pas la même taille qu'un `int` sur une autre
- Contraintes de bornes pour les types numériques
  - Un `int` est généralement 32 bits, un `double` sur 64 bits
  - Bornes dans `limits.h`
- La représentation des négatifs n'est pas fixée
  - Souvent, complément à deux, little indian
- La représentation des flottants n'est pas fixée
  - Souvent, norme IEEE-754 64 bit
- Il convient de savoir avec quoi on travaille
- Référence complète : [https://en.cppreference.com/w/c/language/arithmetic\\_types](https://en.cppreference.com/w/c/language/arithmetic_types)

# Les types de base et le standard C / C++

- Le standard C / C++ impose peu de contraintes de taille
  - Un `int` sur une machine n'a peut-être pas la même taille qu'un `int` sur une autre
- Contraintes de bornes pour les types numériques
  - Un `int` est généralement 32 bits, un `double` sur 64 bits
  - Bornes dans `limits.h`
- La représentation des négatifs n'est pas fixée
  - Souvent, complément à deux, little indian
- La représentation des flottants n'est pas fixée
  - Souvent, norme IEEE-754 64 bit
- Il convient de savoir avec quoi on travaille
- Référence complète : [https://en.cppreference.com/w/c/language/arithmetic\\_types](https://en.cppreference.com/w/c/language/arithmetic_types)

# Les types de base et le standard C / C++

- Le standard C / C++ impose peu de contraintes de taille
  - Un `int` sur une machine n'a peut-être pas la même taille qu'un `int` sur une autre
- Contraintes de bornes pour les types numériques
  - Un `int` est généralement 32 bits, un `double` sur 64 bits
  - Bornes dans `limits.h`
- La représentation des négatifs n'est pas fixée
  - Souvent, complément à deux, little indian
- La représentation des flottants n'est pas fixée
  - Souvent, norme IEEE-754 64 bit
- Il convient de savoir avec quoi on travaille
- Référence complète : [https://en.cppreference.com/w/c/language/arithmetic\\_types](https://en.cppreference.com/w/c/language/arithmetic_types)

# Les types de base et le standard C / C++

- Le standard C / C++ impose peu de contraintes de taille
  - Un `int` sur une machine n'a peut-être pas la même taille qu'un `int` sur une autre
- Contraintes de bornes pour les types numériques
  - Un `int` est généralement 32 bits, un `double` sur 64 bits
  - Bornes dans `limits.h`
- La représentation des négatifs n'est pas fixée
  - Souvent, complément à deux, little indian
- La représentation des flottants n'est pas fixée
  - Souvent, norme IEEE-754 64 bit
- Il convient de savoir avec quoi on travaille
- Référence complète : [https://en.cppreference.com/w/c/language/arithmetic\\_types](https://en.cppreference.com/w/c/language/arithmetic_types)

# Les types de base et le standard C / C++

- Le standard C / C++ impose peu de contraintes de taille
  - Un `int` sur une machine n'a peut-être pas la même taille qu'un `int` sur une autre
- Contraintes de bornes pour les types numériques
  - Un `int` est généralement 32 bits, un `double` sur 64 bits
  - Bornes dans `limits.h`
- La représentation des négatifs n'est pas fixée
  - Souvent, complément à deux, little indian
- La représentation des flottants n'est pas fixée
  - Souvent, norme IEEE-754 64 bit
- Il convient de savoir avec quoi on travaille
- Référence complète : [https://en.cppreference.com/w/c/language/arithmetic\\_types](https://en.cppreference.com/w/c/language/arithmetic_types)

# Les types de base et le standard C / C++

- Le standard C / C++ impose peu de contraintes de taille
  - Un `int` sur une machine n'a peut-être pas la même taille qu'un `int` sur une autre
- Contraintes de bornes pour les types numériques
  - Un `int` est généralement 32 bits, un `double` sur 64 bits
  - Bornes dans `limits.h`
- La représentation des négatifs n'est pas fixée
  - Souvent, complément à deux, little indian
- La représentation des flottants n'est pas fixée
  - Souvent, norme IEEE-754 64 bit
- Il convient de savoir avec quoi on travaille
- Référence complète : [https://en.cppreference.com/w/c/language/arithmetic\\_types](https://en.cppreference.com/w/c/language/arithmetic_types)



# Les types de base et le standard C / C++

- Le standard C / C++ impose peu de contraintes de taille
  - Un `int` sur une machine n'a peut-être pas la même taille qu'un `int` sur une autre
- Contraintes de bornes pour les types numériques
  - Un `int` est généralement 32 bits, un `double` sur 64 bits
  - Bornes dans `limits.h`
- La représentation des négatifs n'est pas fixée
  - Souvent, complément à deux, little indian
- La représentation des flottants n'est pas fixée
  - Souvent, norme IEEE-754 64 bit
- Il convient de savoir avec quoi on travaille
- Référence complète : [https://en.cppreference.com/w/c/language/arithmetic\\_types](https://en.cppreference.com/w/c/language/arithmetic_types)

# Les types de base et le standard C / C++

- Le standard C / C++ impose peu de contraintes de taille
  - Un `int` sur une machine n'a peut-être pas la même taille qu'un `int` sur une autre
- Contraintes de bornes pour les types numériques
  - Un `int` est généralement 32 bits, un `double` sur 64 bits
  - Bornes dans `limits.h`
- La représentation des négatifs n'est pas fixée
  - Souvent, complément à deux, little indian
- La représentation des flottants n'est pas fixée
  - Souvent, norme IEEE-754 64 bit
- Il convient de savoir avec quoi on travaille
- Référence complète : [https://en.cppreference.com/w/c/language/arithmetic\\_types](https://en.cppreference.com/w/c/language/arithmetic_types)

# Les types de base et le standard C / C++

- Le standard C / C++ impose peu de contraintes de taille
  - Un `int` sur une machine n'a peut-être pas la même taille qu'un `int` sur une autre
- Contraintes de bornes pour les types numériques
  - Un `int` est généralement 32 bits, un `double` sur 64 bits
  - Bornes dans `limits.h`
- La représentation des négatifs n'est pas fixée
  - Souvent, complément à deux, little indian
- La représentation des flottants n'est pas fixée
  - Souvent, norme IEEE-754 64 bit
- Il convient de savoir avec quoi on travaille
- Référence complète : [https://en.cppreference.com/w/c/language/arithmetic\\_types](https://en.cppreference.com/w/c/language/arithmetic_types)

# Les types de base et le standard C / C++

- Le standard C / C++ impose peu de contraintes de taille
  - Un `int` sur une machine n'a peut-être pas la même taille qu'un `int` sur une autre
- Contraintes de bornes pour les types numériques
  - Un `int` est généralement 32 bits, un `double` sur 64 bits
  - Bornes dans `limits.h`
- La représentation des négatifs n'est pas fixée
  - Souvent, complément à deux, little indian
- La représentation des flottants n'est pas fixée
  - Souvent, norme IEEE-754 64 bit
- Il convient de savoir avec quoi on travaille
- Référence complète : [https://en.cppreference.com/w/c/language/arithmetic\\_types](https://en.cppreference.com/w/c/language/arithmetic_types)

# Bornes

- En C : fichier `bounds.c`
- On imprime avec `printf` et en spécifiant un format (cf. section suivante)

```
1 #include <stdio.h>
2 #include <limits.h>
3 #include <float.h>
4
5 int main(void)
6 {
7     printf("INT_MIN_====_%d\n", INT_MIN);
8     printf("INT_MAX_====_%d\n", INT_MAX);
9     printf("UINT_MAX_====_%u\n",  UINT_MAX);
10    printf("\n");
11
12    printf("FLT_MIN_====_%e\n", FLT_MIN);
13    printf("FLT_MAX_====_%e\n", FLT_MAX);
14    printf("\n");
15
16    printf("DBL_MIN_====_%e\n", DBL_MIN);
17    printf("DBL_MAX_====_%e\n", DBL_MAX);
18    printf("\n");
19 }
```

# Ce que l'on sait des tailles

- `sizeof`(`type`) permet de connaître la taille d'un type (en bytes)
  - Ne peut être utilisé sur des fonctions ou des types incomplets
- Un byte est constitué de 8 bits ou plus
  - Généralement 8
  - Spécifié par `CHAR_BIT`
- Un `char` est toujours de taille 1 (byte)
- Une adresse (pointeur) est de la taille du bus d'adresse (32 bits en x86, 64 bits en x64)
- La taille d'une classe ou structure est au moins la somme des tailles de ses attributs
  - Mais pas toujours, dépendant des contraintes d'alignement

# Ce que l'on sait des tailles

- `sizeof`(`type`) permet de connaître la taille d'un type (en bytes)
  - Ne peut être utilisé sur des fonctions ou des types incomplets
- Un byte est constitué de 8 bits ou plus
  - Généralement 8
  - Spécifié par `CHAR_BIT`
- Un `char` est toujours de taille 1 (byte)
- Une adresse (pointeur) est de la taille du bus d'adresse (32 bits en x86, 64 bits en x64)
- La taille d'une classe ou structure est au moins la somme des tailles de ses attributs
  - Mais pas toujours, dépendant des contraintes d'alignement

# Ce que l'on sait des tailles

- `sizeof`(`type`) permet de connaître la taille d'un type (en bytes)
  - Ne peut être utilisé sur des fonctions ou des types incomplets
- Un byte est constitué de 8 bits ou plus
  - Généralement 8
  - Spécifié par `CHAR_BIT`
- Un `char` est toujours de taille 1 (byte)
- Une adresse (pointeur) est de la taille du bus d'adresse (32 bits en x86, 64 bits en x64)
- La taille d'une classe ou structure est au moins la somme des tailles de ses attributs
  - Mais pas toujours, dépendant des contraintes d'alignement



# Ce que l'on sait des tailles

- `sizeof`(`type`) permet de connaître la taille d'un type (en bytes)
  - Ne peut être utilisé sur des fonctions ou des types incomplets
- Un byte est constitué de 8 bits ou plus
  - Généralement 8
  - Spécifié par `CHAR_BIT`
- Un `char` est toujours de taille 1 (byte)
- Une adresse (pointeur) est de la taille du bus d'adresse (32 bits en x86, 64 bits en x64)
- La taille d'une classe ou structure est au moins la somme des tailles de ses attributs
  - Mais pas toujours, dépendant des contraintes d'alignement

# Ce que l'on sait des tailles

- `sizeof`(`type`) permet de connaître la taille d'un type (en bytes)
  - Ne peut être utilisé sur des fonctions ou des types incomplets
- Un byte est constitué de 8 bits ou plus
  - Généralement 8
  - Spécifié par `CHAR_BIT`
- Un `char` est toujours de taille 1 (byte)
- Une adresse (pointeur) est de la taille du bus d'adresse (32 bits en x86, 64 bits en x64)
- La taille d'une classe ou structure est au moins la somme des tailles de ses attributs
  - Mais pas toujours, dépendant des contraintes d'alignement

# Ce que l'on sait des tailles

- `sizeof`(`type`) permet de connaître la taille d'un type (en bytes)
  - Ne peut être utilisé sur des fonctions ou des types incomplets
- Un byte est constitué de 8 bits ou plus
  - Généralement 8
  - Spécifié par `CHAR_BIT`
- Un `char` est toujours de taille 1 (byte)
- Une adresse (pointeur) est de la taille du bus d'adresse (32 bits en x86, 64 bits en x64)
- La taille d'une classe ou structure est au moins la somme des tailles de ses attributs
  - Mais pas toujours, dépendant des contraintes d'alignement

# Ce que l'on sait des tailles

- `sizeof(type)` permet de connaître la taille d'un type (en bytes)
  - Ne peut être utilisé sur des fonctions ou des types incomplets
- Un byte est constitué de 8 bits ou plus
  - Généralement 8
  - Spécifié par `CHAR_BIT`
- Un `char` est toujours de taille 1 (byte)
- Une adresse (pointeur) est de la taille du bus d'adresse (32 bits en x86, 64 bits en x64)
- La taille d'une classe ou structure est au moins la somme des tailles de ses attributs
  - Mais pas toujours, dépendant des contraintes d'alignement

# Ce que l'on sait des tailles

- `sizeof(type)` permet de connaître la taille d'un type (en bytes)
  - Ne peut être utilisé sur des fonctions ou des types incomplets
- Un byte est constitué de 8 bits ou plus
  - Généralement 8
  - Spécifié par `CHAR_BIT`
- Un `char` est toujours de taille 1 (byte)
- Une adresse (pointeur) est de la taille du bus d'adresse (32 bits en x86, 64 bits en x64)
- La taille d'une classe ou structure est au moins la somme des tailles de ses attributs
  - Mais pas toujours, dépendant des contraintes d'alignement

# Ce que l'on sait des tailles

- `sizeof(type)` permet de connaître la taille d'un type (en bytes)
  - Ne peut être utilisé sur des fonctions ou des types incomplets
- Un byte est constitué de 8 bits ou plus
  - Généralement 8
  - Spécifié par `CHAR_BIT`
- Un `char` est toujours de taille 1 (byte)
- Une adresse (pointeur) est de la taille du bus d'adresse (32 bits en x86, 64 bits en x64)
- La taille d'une classe ou structure est au moins la somme des tailles de ses attributs
  - Mais pas toujours, dépendant des contraintes d'alignement

# Taille des types de base

- Fichier `sizeof.c`
- En C++ : même principe en imprimant avec `std::cout`

```
1  #include "stdio.h"
2
3  int main()
4  {
5      printf("Size_of_char: %lu_bytes\n", sizeof(char));
6      printf("Size_of_short: %lu_bytes\n", sizeof(short));
7      printf("Size_of_int: %lu_bytes\n", sizeof(int));
8      printf("Size_of_long_int: %lu_bytes\n", sizeof(long int));
9      printf("Size_of_long_long_int: %lu_bytes\n", sizeof(long long int));
10     printf("Size_of_adress: %lu_bytes\n", sizeof(int *));
11     printf("Size_of_float: %lu_bytes\n", sizeof(float));
12     printf("Size_of_double: %lu_bytes\n", sizeof(double));
13     printf("Size_of_long_double: %lu_bytes\n", sizeof(long double));
14 }
```

# Illustration

## ■ Fichier `align.c` (même principe en C++)

```
1  struct A
2  {
3      int i; //size 4
4  };
5
6  struct B
7  {
8      char c; //size 1
9  };
10
11 struct C
12 {
13     int i; //size 4
14     char c; //size 1
15     //3 bytes padding
16 }; //size 8
17
18 int main()
19 {
20     printf("%lu%lu%lu\n", sizeof(struct A), sizeof(struct B), sizeof(struct C));
21 }
```



# Types entiers

## ■ Différents types d'entiers

- `short int` (abrégé en `short`)
- `int`
- `long int` (abrégé en `long`)
- `long long int` (abrégé en `long long`)

## ■ Limites de codage différentes

## ■ Possibilité de travailler en non signé

- `unsigned long int`
- Permet de doubler la valeur maximum du codage
- Mettre un négatif dans un `unsigned` conduit à un comportement indéterminé

## ■ Immédiats codé comme 1, 017, 0x1A, etc.

## ■ Par défaut, les immédiats entiers sont de type `int`

## ■ Type forcé via suffixes `u`, `l`, `ul`

■ `3 + 42ul`

# Types entiers

## ■ Différents types d'entiers

- `short int` (abrégé en `short`)

- `int`

- `long int` (abrégé en `long`)

- `long long int` (abrégé en `long long`)

## ■ Limites de codage différentes

## ■ Possibilité de travailler en non signé

- `unsigned long int`

- Permet de doubler la valeur maximum du codage

- Mettre un négatif dans un `unsigned` conduit à un comportement indéterminé

## ■ Immédiats codé comme 1, 017, 0x1A, etc.

## ■ Par défaut, les immédiats entiers sont de type `int`

## ■ Type forcé via suffixes `u`, `l`, `ul`

- `3 + 42ul`

# Types entiers

## ■ Différents types d'entiers

- `short int` (abrégé en `short`)

- `int`

- `long int` (abrégé en `long`)

- `long long int` (abrégé en `long long`)

## ■ Limites de codage différentes

## ■ Possibilité de travailler en non signé

- `unsigned long int`

- Permet de doubler la valeur maximum du codage

- Mettre un négatif dans un `unsigned` conduit à un comportement indéterminé

## ■ Immédiats codé comme 1, 017, 0x1A, etc.

## ■ Par défaut, les immédiats entiers sont de type `int`

## ■ Type forcé via suffixes `u`, `l`, `ul`

- `3 + 42ul`

# Types entiers

## ■ Différents types d'entiers

- `short int` (abrégé en `short`)

- `int`

- `long int` (abrégé en `long`)

- `long long int` (abrégé en `long long`)

## ■ Limites de codage différentes

## ■ Possibilité de travailler en non signé

- `unsigned long int`

- Permet de doubler la valeur maximum du codage

- Mettre un négatif dans un `unsigned` conduit à un comportement indéterminé

## ■ Immédiats codé comme 1, 017, 0x1A, etc.

## ■ Par défaut, les immédiats entiers sont de type `int`

## ■ Type forcé via suffixes `u`, `l`, `ul`

- `3 + 42ul`

# Types entiers

## ■ Différents types d'entiers

- `short int` (abrégé en `short`)
- `int`
- `long int` (abrégé en `long`)
- `long long int` (abrégé en `long long`)

## ■ Limites de codage différentes

## ■ Possibilité de travailler en non signé

- `unsigned long int`
- Permet de doubler la valeur maximum du codage
- Mettre un négatif dans un `unsigned` conduit à un comportement indéterminé

## ■ Immédiats codé comme 1, 017, 0x1A, etc.

## ■ Par défaut, les immédiats entiers sont de type `int`

## ■ Type forcé via suffixes `u`, `l`, `ul`

■ `3 + 42ul`

# Types entiers

## ■ Différents types d'entiers

- `short int` (abrégé en `short`)
- `int`
- `long int` (abrégé en `long`)
- `long long int` (abrégé en `long long`)

## ■ Limites de codage différentes

### ■ Possibilité de travailler en non signé

- `unsigned long int`
- Permet de doubler la valeur maximum du codage
- Mettre un négatif dans un `unsigned` conduit à un comportement indéterminé

### ■ Immédiats codé comme 1, 017, 0x1A, etc.

### ■ Par défaut, les immédiats entiers sont de type `int`

### ■ Type forcé via suffixes `u`, `l`, `ul`

■ `3 + 42ul`

# Types entiers

## ■ Différents types d'entiers

- `short int` (abrégé en `short`)
- `int`
- `long int` (abrégé en `long`)
- `long long int` (abrégé en `long long`)

## ■ Limites de codage différentes

## ■ Possibilité de travailler en non signé

- `unsigned long int`
- Permet de doubler la valeur maximum du codage
- Mettre un négatif dans un `unsigned` conduit à un comportement indéterminé

## ■ Immédiats codé comme 1, 017, 0x1A, etc.

## ■ Par défaut, les immédiats entiers sont de type `int`

## ■ Type forcé via suffixes `u`, `l`, `ul`

■ `3 + 42ul`

# Types entiers

## ■ Différents types d'entiers

- `short int` (abrégé en `short`)
- `int`
- `long int` (abrégé en `long`)
- `long long int` (abrégé en `long long`)

## ■ Limites de codage différentes

## ■ Possibilité de travailler en non signé

- `unsigned long int`
- Permet de doubler la valeur maximum du codage
- Mettre un négatif dans un `unsigned` conduit à un comportement indéterminé

## ■ Immédiats codé comme 1, 017, 0x1A, etc.

## ■ Par défaut, les immédiats entiers sont de type `int`

## ■ Type forcé via suffixes `u`, `l`, `ul`

■ `3 + 42ul`



# Types entiers

## ■ Différents types d'entiers

- `short int` (abrégé en `short`)
- `int`
- `long int` (abrégé en `long`)
- `long long int` (abrégé en `long long`)

## ■ Limites de codage différentes

## ■ Possibilité de travailler en non signé

- `unsigned long int`
- Permet de doubler la valeur maximum du codage
- Mettre un négatif dans un `unsigned` conduit à un comportement indéterminé

## ■ Immédiats codé comme 1, 017, 0x1A, etc.

## ■ Par défaut, les immédiats entiers sont de type `int`

## ■ Type forcé via suffixes `u`, `l`, `ul`

■ `3 + 42ul`

# Types entiers

## ■ Différents types d'entiers

- `short int` (abrégé en `short`)
- `int`
- `long int` (abrégé en `long`)
- `long long int` (abrégé en `long long`)

## ■ Limites de codage différentes

## ■ Possibilité de travailler en non signé

- `unsigned long int`
- Permet de doubler la valeur maximum du codage
- Mettre un négatif dans un `unsigned` conduit à un comportement indéterminé

## ■ Immédiats codé comme 1, 017, 0x1A, etc.

## ■ Par défaut, les immédiats entiers sont de type `int`

## ■ Type forcé via suffixes `u`, `l`, `ul`

■ `3 + 42ul`

# Types entiers

## ■ Différents types d'entiers

- `short int` (abrégé en `short`)
- `int`
- `long int` (abrégé en `long`)
- `long long int` (abrégé en `long long`)

## ■ Limites de codage différentes

## ■ Possibilité de travailler en non signé

- `unsigned long int`
- Permet de doubler la valeur maximum du codage
- Mettre un négatif dans un `unsigned` conduit à un comportement indéterminé

## ■ Immédiats codé comme 1, 017, 0x1A, etc.

## ■ Par défaut, les immédiats entiers sont de type `int`

## ■ Type forcé via suffixes `u`, `l`, `ul`

■ `3 + 42ul`

# Types entiers

## ■ Différents types d'entiers

- `short int` (abrégé en `short`)
- `int`
- `long int` (abrégé en `long`)
- `long long int` (abrégé en `long long`)

## ■ Limites de codage différentes

## ■ Possibilité de travailler en non signé

- `unsigned long int`
- Permet de doubler la valeur maximum du codage
- Mettre un négatif dans un `unsigned` conduit à un comportement indéterminé

## ■ Immédiats codé comme 1, 017, 0x1A, etc.

## ■ Par défaut, les immédiats entiers sont de type `int`

## ■ Type forcé via suffixes `u`, `l`, `ul`

■ `3 + 42ul`

# Types entiers

## ■ Différents types d'entiers

- `short int` (abrégé en `short`)
- `int`
- `long int` (abrégé en `long`)
- `long long int` (abrégé en `long long`)

## ■ Limites de codage différentes

## ■ Possibilité de travailler en non signé

- `unsigned long int`
- Permet de doubler la valeur maximum du codage
- Mettre un négatif dans un `unsigned` conduit à un comportement indéterminé

## ■ Immédiats codé comme 1, 017, 0x1A, etc.

## ■ Par défaut, les immédiats entiers sont de type `int`

## ■ Type forcé via suffixes `u`, `l`, `ul`

```
■ 3 + 42ul
```

# Types entiers

## ■ Différents types d'entiers

- `short int` (abrégé en `short`)
- `int`
- `long int` (abrégé en `long`)
- `long long int` (abrégé en `long long`)

## ■ Limites de codage différentes

## ■ Possibilité de travailler en non signé

- `unsigned long int`
- Permet de doubler la valeur maximum du codage
- Mettre un négatif dans un `unsigned` conduit à un comportement indéterminé

## ■ Immédiats codé comme 1, 017, 0x1A, etc.

## ■ Par défaut, les immédiats entiers sont de type `int`

## ■ Type forcé via suffixes `u`, `l`, `ul`

- `3 + 42ul`

# Illustration

## ■ Fichier `int.c` (même principe en C++)

```
1  #include <stdio.h>
2  #include <limits.h>
3
4  int main()
5  {
6      unsigned i = -1;
7      if (i == UINT_MAX)
8          printf("Two_complement\n");
9      else
10         printf("Unknown_negative_representation\n");
11
12     unsigned j = 1;
13     char* ptr = (char*)&j;
14     if (*c == 0) //a bit ugly
15         printf("Little_indian\n");
16     else
17         printf("Big_indian\n");
18 }
```

# Types flottants

## ■ Permet de représenter les rationnels

- Pas de réels car codage (et précision) fini

## ■ Souvent, représentés par la norme IEEE-754 64 bit

- $q = M \times B^E$

## ■ `limits` contient aussi des informations relatives aux règles d'arrondis, à l'epsilon machine, etc.

## ■ Immédiats codés comme `0.`, `404.42`, `1.23E256`, etc.

## ■ Type forcé via suffixes `f`, `l`

- `3 + 2lf`



# Types flottants

- Permet de représenter les rationnels
  - Pas de réels car codage (et précision) fini
- Souvent, représentés par la norme IEEE-754 64 bit
  - $q = M \times B^E$
- `limits` contient aussi des informations relatives aux règles d'arrondis, à l'epsilon machine, etc.
- Immédiats codés comme `0.`, `404.42`, `1.23E256`, etc.
- Type forcé via suffixes `f`, `l`
  - `3 + 21f`

# Types flottants

- Permet de représenter les rationnels
  - Pas de réels car codage (et précision) fini
- Souvent, représentés par la norme IEEE-754 64 bit
  - $q = M \times B^E$
- `limits` contient aussi des informations relatives aux règles d'arrondis, à l'epsilon machine, etc.
- Immédiats codés comme `0.`, `404.42`, `1.23E256`, etc.
- Type forcé via suffixes `f`, `l`
  - `3 + 2lf`

# Types flottants

- Permet de représenter les rationnels
  - Pas de réels car codage (et précision) fini
- Souvent, représentés par la norme IEEE-754 64 bit
  - $q = M \times B^E$
- `limits` contient aussi des informations relatives aux règles d'arrondis, à l'epsilon machine, etc.
- Immédiats codés comme `0.`, `404.42`, `1.23E256`, etc.
- Type forcé via suffixes `f`, `l`
  - `3 + 2lf`

# Types flottants

- Permet de représenter les rationnels
  - Pas de réels car codage (et précision) fini
- Souvent, représentés par la norme IEEE-754 64 bit
  - $q = M \times B^E$
- `limits` contient aussi des informations relatives aux règles d'arrondis, à l'epsilon machine, etc.
- Immédiats codés comme `0.`, `404.42`, `1.23E256`, etc.
- Type forcé via suffixes `f`, `l`
  - `3 + 2lf`

# Types flottants

- Permet de représenter les rationnels
  - Pas de réels car codage (et précision) fini
- Souvent, représentés par la norme IEEE-754 64 bit
  - $q = M \times B^E$
- `limits` contient aussi des informations relatives aux règles d'arrondis, à l'epsilon machine, etc.
- Immédiats codés comme `0.`, `404.42`, `1.23E256`, etc.
- Type forcé via suffixes `f`, `l`
  - `3 + 2lf`

# Types flottants

- Permet de représenter les rationnels
  - Pas de réels car codage (et précision) fini
- Souvent, représentés par la norme IEEE-754 64 bit
  - $q = M \times B^E$
- `limits` contient aussi des informations relatives aux règles d'arrondis, à l'epsilon machine, etc.
- Immédiats codés comme `0.`, `404.42`, `1.23E256`, etc.
- Type forcé via suffixes `f`, `l`
  - `3 + 2lf`

# Types flottants

- Permet de représenter les rationnels
  - Pas de réels car codage (et précision) fini
- Souvent, représentés par la norme IEEE-754 64 bit
  - $q = M \times B^E$
- `limits` contient aussi des informations relatives aux règles d'arrondis, à l'epsilon machine, etc.
- Immédiats codés comme `0.`, `404.42`, `1.23E256`, etc.
- Type forcé via suffixes `f`, `l`
  - `3 + 2lf`

# Les variables globales

- Variable déclarée en dehors de tout bloc
- Durée de vie de tout le programme
  - Classe d'allocation statique (cf. Ch. 5)
- Pas de mot-clé spécifique
- Si déclarée avec `static`, a une portée semi-globale
  - Même unité de traduction
  - Portée locale au fichier objet



# Les variables globales

- Variable déclarée en dehors de tout bloc
- Durée de vie de tout le programme
  - Classe d'allocation statique (cf. Ch. 5)
- Pas de mot-clé spécifique
- Si déclarée avec `static`, a une portée semi-globale
  - Même unité de traduction
  - Portée locale au fichier objet

# Les variables globales

- Variable déclarée en dehors de tout bloc
- Durée de vie de tout le programme
  - Classe d'allocation statique (cf. Ch. 5)
- Pas de mot-clé spécifique
- Si déclarée avec `static`, a une portée semi-globale
  - Même unité de traduction
  - Portée locale au fichier objet

# Les variables globales

- Variable déclarée en dehors de tout bloc
- Durée de vie de tout le programme
  - Classe d'allocation statique (cf. Ch. 5)
- Pas de mot-clé spécifique
- Si déclarée avec `static`, a une portée semi-globale
  - Même unité de traduction
  - Portée locale au fichier objet

# Les variables globales

- Variable déclarée en dehors de tout bloc
- Durée de vie de tout le programme
  - Classe d'allocation statique (cf. Ch. 5)
- Pas de mot-clé spécifique
- Si déclarée avec `static`, a une portée semi-globale
  - Même unité de traduction
  - Portée locale au fichier objet

# Les variables globales

- Variable déclarée en dehors de tout bloc
- Durée de vie de tout le programme
  - Classe d'allocation statique (cf. Ch. 5)
- Pas de mot-clé spécifique
- Si déclarée avec `static`, a une portée semi-globale
  - Même unité de traduction
  - Portée locale au fichier objet

# Les variables globales

- Variable déclarée en dehors de tout bloc
- Durée de vie de tout le programme
  - Classe d'allocation statique (cf. Ch. 5)
- Pas de mot-clé spécifique
- Si déclarée avec `static`, a une portée semi-globale
  - Même unité de traduction
  - Portée locale au fichier objet

# Les booléens

## ■ Pré-C11 : inexistant

- Souvent, macros globales

## ■ En C11, il faut inclure `stdbool.h`

## ■ `bool` : types booléens

- Immédiats `true` et `false`

## ■ Conversions implicites des types numériques vers `bool`

- 0 : `false`
- Autre : `true`

## Bonne pratique

- Éviter d'utiliser les conversions implicites vers `bool`
- Déclarer les booléens comme `bool`

# Les booléens

- Pré-C11 : inexistant
  - Souvent, macros globales
- En C11, il faut inclure `stdbool.h`
- `bool` : types booléens
  - Immédiats `true` et `false`
- Conversions implicites des types numériques vers `bool`
  - 0 : `false`
  - Autre : `true`

## Bonne pratique

- Éviter d'utiliser les conversions implicites vers `bool`
- Déclarer les booléens comme `bool`



# Les booléens

- Pré-C11 : inexistant
  - Souvent, macros globales
- En C11, il faut inclure `stdbool.h`
- `bool` : types booléens
  - Immédiats `true` et `false`
- Conversions implicites des types numériques vers `bool`
  - 0 : `false`
  - Autre : `true`

## Bonne pratique

- Éviter d'utiliser les conversions implicites vers `bool`
- Déclarer les booléens comme `bool`

# Les booléens

- Pré-C11 : inexistant
  - Souvent, macros globales
- En C11, il faut inclure `stdbool.h`
- `bool` : types booléens
  - Immédiats `true` et `false`
- Conversions implicites des types numériques vers `bool`
  - 0 : `false`
  - Autre : `true`

## Bonne pratique

- Éviter d'utiliser les conversions implicites vers `bool`
- Déclarer les booléens comme `bool`

# Les booléens

- Pré-C11 : inexistant
  - Souvent, macros globales
- En C11, il faut inclure `stdbool.h`
- `bool` : types booléens
  - Immédiats `true` et `false`
- Conversions implicites des types numériques vers `bool`
  - 0 : `false`
  - Autre : `true`

## Bonne pratique

- Éviter d'utiliser les conversions implicites vers `bool`
- Déclarer les booléens comme `bool`

# Les booléens

- Pré-C11 : inexistant
  - Souvent, macros globales
- En C11, il faut inclure `stdbool.h`
- `bool` : types booléens
  - Immédiats `true` et `false`
- Conversions implicites des types numériques vers `bool`
  - `0` : `false`
  - Autre : `true`

## Bonne pratique

- Éviter d'utiliser les conversions implicites vers `bool`
- Déclarer les booléens comme `bool`

# Les booléens

- Pré-C11 : inexistant
  - Souvent, macros globales
- En C11, il faut inclure `stdbool.h`
- `bool` : types booléens
  - Immédiats `true` et `false`
- Conversions implicites des types numériques vers `bool`
  - 0 : `false`
  - Autre : `true`

## Bonne pratique

- Éviter d'utiliser les conversions implicites vers `bool`
- Déclarer les booléens comme `bool`

# Les booléens

- Pré-C11 : inexistant
  - Souvent, macros globales
- En C11, il faut inclure `stdbool.h`
- `bool` : types booléens
  - Immédiats `true` et `false`
- Conversions implicites des types numériques vers `bool`
  - 0 : `false`
  - Autre : `true`

## Bonne pratique

- Éviter d'utiliser les conversions implicites vers `bool`
- Déclarer les booléens comme `bool`

# Les booléens

- Pré-C11 : inexistant
  - Souvent, macros globales
- En C11, il faut inclure `stdbool.h`
- `bool` : types booléens
  - Immédiats `true` et `false`
- Conversions implicites des types numériques vers `bool`
  - 0 : `false`
  - Autre : `true`

## Bonne pratique

- Éviter d'utiliser les conversions implicites vers `bool`
- Déclarer les booléens comme `bool`

# Les booléens

- Pré-C11 : inexistant
  - Souvent, macros globales
- En C11, il faut inclure `stdbool.h`
- `bool` : types booléens
  - Immédiats `true` et `false`
- Conversions implicites des types numériques vers `bool`
  - 0 : `false`
  - Autre : `true`

## Bonne pratique

- Éviter d'utiliser les conversions implicites vers `bool`
- Déclarer les booléens comme `bool`



# Les booléens

- Pré-C11 : inexistant
  - Souvent, macros globales
- En C11, il faut inclure `stdbool.h`
- `bool` : types booléens
  - Immédiats `true` et `false`
- Conversions implicites des types numériques vers `bool`
  - 0 : `false`
  - Autre : `true`

## Bonne pratique

- Éviter d'utiliser les conversions implicites vers `bool`
- Déclarer les booléens comme `bool`

# Exemple

## ■ Fichier `bool.c` (même principe en C++)

```
1  int main()
2  {
3      bool b = true;
4      if (b)
5          printf("true\n\n");
6      else
7          printf("false\n\n");
8
9      if (-1)
10         printf("-1_true\n\n");
11     else
12         printf("-1_false\n\n");
13
14     if (0)
15         printf("0_true\n\n");
16     else
17         printf("0_false\n\n");
18
19     if (1)
20         printf("1_true\n\n");
21     else
22         printf("1_false\n\n");
23 }
```

# Autres types

## ■ `char` : caractère

- Encodage en fonction du charset
- Non signé :  $[0, 255]$ , signé :  $[-128, 127]$
- Pas de support non ASCII par défaut

## ■ Entiers à taille fixe

- Inclure `stdint.h` ou `cstdint.h` (cpp)
- Exemple : `int8_t`, `int32_t`, etc.

## ■ `void` : « vide » (Type incomplet)

- Utilisé dans les retours de fonction
- Offre une généricité sur les pointeurs (cf. Ch. 2)

## ■ `type*` : pointeur (adresse) vers un élément de type donné (cf. Ch. 2)

## ■ `type&` : référence (C++ seulement) (cf. Ch. 2)

# Autres types

## ■ `char` : caractère

### ■ Encodage en fonction du charset

- Non signé :  $[0, 255]$ , signé :  $[-128, 127]$
- Pas de support non ASCII par défaut

## ■ Entiers à taille fixe

- Inclure `stdint.h` ou `cstdint.h` (cpp)
- Exemple : `int8_t`, `int32_t`, etc.

## ■ `void` : « vide » (Type incomplet)

- Utilisé dans les retours de fonction
- Offre une généricité sur les pointeurs (cf. Ch. 2)

## ■ `type*` : pointeur (adresse) vers un élément de type donné (cf. Ch. 2)

## ■ `type&` : référence (C++ seulement) (cf. Ch. 2)

# Autres types

## ■ `char` : caractère

- Encodage en fonction du charset
- Non signé :  $[0, 255]$ , signé :  $[-128, 127]$
- Pas de support non ASCII par défaut

## ■ Entiers à taille fixe

- Inclure `stdint.h` ou `cstdint.h` (cpp)
- Exemple : `int8_t`, `int32_t`, etc.

## ■ `void` : « vide » (Type incomplet)

- Utilisé dans les retours de fonction
- Offre une généricité sur les pointeurs (cf. Ch. 2)

## ■ `type*` : pointeur (adresse) vers un élément de type donné (cf. Ch. 2)

## ■ `type&` : référence (C++ seulement) (cf. Ch. 2)

# Autres types

## ■ `char` : caractère

- Encodage en fonction du charset
- Non signé :  $[0, 255]$ , signé :  $[-128, 127]$
- Pas de support non ASCII par défaut

## ■ Entiers à taille fixe

- Inclure `stdint.h` ou `cstdint.h` (cpp)
- Exemple : `int8_t`, `int32_t`, etc.

## ■ `void` : « vide » (Type incomplet)

- Utilisé dans les retours de fonction
- Offre une généricité sur les pointeurs (cf. Ch. 2)

## ■ `type*` : pointeur (adresse) vers un élément de type donné (cf. Ch. 2)

## ■ `type&` : référence (C++ seulement) (cf. Ch. 2)

# Autres types

## ■ `char` : caractère

- Encodage en fonction du charset
- Non signé :  $[0, 255]$ , signé :  $[-128, 127]$
- Pas de support non ASCII par défaut

## ■ Entiers à taille fixe

- Inclure `stdint.h` ou `cstdint.h` (cpp)
- Exemple : `int8_t`, `int32_t`, etc.

## ■ `void` : « vide » (Type incomplet)

- Utilisé dans les retours de fonction
- Offre une généricité sur les pointeurs (cf. Ch. 2)

## ■ `type*` : pointeur (adresse) vers un élément de type donné (cf. Ch. 2)

## ■ `type&` : référence (C++ seulement) (cf. Ch. 2)

# Autres types

## ■ `char` : caractère

- Encodage en fonction du charset
- Non signé :  $[0, 255]$ , signé :  $[-128, 127]$
- Pas de support non ASCII par défaut

## ■ Entiers à taille fixe

- Inclure `stdint.h` ou `cstdint.h` (cpp)
- Exemple : `int8_t`, `int32_t`, etc.

## ■ `void` : « vide » (Type incomplet)

- Utilisé dans les retours de fonction
- Offre une généricité sur les pointeurs (cf. Ch. 2)

## ■ `type*` : pointeur (adresse) vers un élément de type donné (cf. Ch. 2)

## ■ `type&` : référence (C++ seulement) (cf. Ch. 2)



# Autres types

## ■ `char` : caractère

- Encodage en fonction du charset
- Non signé :  $[0, 255]$ , signé :  $[-128, 127]$
- Pas de support non ASCII par défaut

## ■ Entiers à taille fixe

- Inclure `stdint.h` ou `cstdint.h` (cpp)
- Exemple : `int8_t`, `int32_t`, etc.

## ■ `void` : « vide » (Type incomplet)

- Utilisé dans les retours de fonction
- Offre une généricité sur les pointeurs (cf. Ch. 2)

## ■ `type*` : pointeur (adresse) vers un élément de type donné (cf. Ch. 2)

## ■ `type&` : référence (C++ seulement) (cf. Ch. 2)

# Autres types

## ■ `char` : caractère

- Encodage en fonction du charset
- Non signé :  $[0, 255]$ , signé :  $[-128, 127]$
- Pas de support non ASCII par défaut

## ■ Entiers à taille fixe

- Inclure `stdint.h` ou `cstdint.h` (cpp)
- Exemple : `int8_t`, `int32_t`, etc.

## ■ `void` : « vide » (Type incomplet)

- Utilisé dans les retours de fonction
- Offre une généricité sur les pointeurs (cf. Ch. 2)

## ■ `type*` : pointeur (adresse) vers un élément de type donné (cf. Ch. 2)

## ■ `type&` : référence (C++ seulement) (cf. Ch. 2)

# Autres types

## ■ `char` : caractère

- Encodage en fonction du charset
- Non signé :  $[0, 255]$ , signé :  $[-128, 127]$
- Pas de support non ASCII par défaut

## ■ Entiers à taille fixe

- Inclure `stdint.h` ou `cstdint.h` (cpp)
- Exemple : `int8_t`, `int32_t`, etc.

## ■ `void` : « vide » (Type incomplet)

- Utilisé dans les retours de fonction
- Offre une généricité sur les pointeurs (cf. Ch. 2)

## ■ `type*` : pointeur (adresse) vers un élément de type donné (cf. Ch. 2)

## ■ `type&` : référence (C++ seulement) (cf. Ch. 2)

# Autres types

## ■ `char` : caractère

- Encodage en fonction du charset
- Non signé :  $[0, 255]$ , signé :  $[-128, 127]$
- Pas de support non ASCII par défaut

## ■ Entiers à taille fixe

- Inclure `stdint.h` ou `cstdint.h` (cpp)
- Exemple : `int8_t`, `int32_t`, etc.

## ■ `void` : « vide » (Type incomplet)

- Utilisé dans les retours de fonction
- Offre une généricité sur les pointeurs (cf. Ch. 2)

## ■ `type*` : pointeur (adresse) vers un élément de type donné (cf. Ch. 2)

## ■ `type&` : référence (C++ seulement) (cf. Ch. 2)

# Autres types

## ■ `char` : caractère

- Encodage en fonction du charset
- Non signé :  $[0, 255]$ , signé :  $[-128, 127]$
- Pas de support non ASCII par défaut

## ■ Entiers à taille fixe

- Inclure `stdint.h` ou `cstdint.h` (cpp)
- Exemple : `int8_t`, `int32_t`, etc.

## ■ `void` : « vide » (Type incomplet)

- Utilisé dans les retours de fonction
- Offre une généricité sur les pointeurs (cf. Ch. 2)

## ■ `type*` : pointeur (adresse) vers un élément de type donné (cf. Ch. 2)

## ■ `type&` : référence (C++ seulement) (cf. Ch. 2)

# Autres types

## ■ `char` : caractère

- Encodage en fonction du charset
- Non signé :  $[0, 255]$ , signé :  $[-128, 127]$
- Pas de support non ASCII par défaut

## ■ Entiers à taille fixe

- Inclure `stdint.h` ou `cstdint.h` (cpp)
- Exemple : `int8_t`, `int32_t`, etc.

## ■ `void` : « vide » (Type incomplet)

- Utilisé dans les retours de fonction
- Offre une généricité sur les pointeurs (cf. Ch. 2)

## ■ `type*` : pointeur (adresse) vers un élément de type donné (cf. Ch. 2)

## ■ `type&` : référence (C++ seulement) (cf. Ch. 2)

# Illustration

## ■ Fichier `other.c` (même principe en C++)

```
1  int main()
2  {
3      char a = 'a';
4      printf("Char_: %c\n", a);
5      char ee = 'é';
6      printf("Char_: %c\n", ee);
7
8      int8_t i8 = 0;
9      int16_t i16 = 0;
10     int32_t i32 = 0;
11     int64_t i64 = 0;
12     printf("%zu_%zu_%zu_%zu\n", sizeof(i8), sizeof(i16), sizeof(i32), sizeof(i64));
13
14     int i = 2;
15     int * addr = &i;
16     printf("i_=%d_stored_at_address_%p\n", i, addr);
17 }
```

# Constantes



# CV-Qualifiers

- Pour chaque type, incluant les types incomplets, il existe trois autres « sous-types »
  - 1 `const` : type constant, accédé en lecture seule
  - 2 `volatile` : type volatile, peut être accédé et modifié par un processus extérieur
  - 3 `const volatile` : les deux en même temps
- Motivation : optimisations compilatoires
  - Un `const` peut parfois être passé par référence
  - Les instructions comprenant des volatiles ne peuvent être réordonnées
- Applications
  - Optimisation de code
  - Multithreading

# CV-Qualifiers

- Pour chaque type, incluant les types incomplets, il existe trois autres « sous-types »
  - 1 `const` : type constant, accédé en lecture seule
  - 2 `volatile` : type volatile, peut être accédé et modifié par un processus extérieur
  - 3 `const volatile` : les deux en même temps
- Motivation : optimisations compilatoires
  - Un `const` peut parfois être passé par référence
  - Les instructions comprenant des volatiles ne peuvent être réordonnées
- Applications
  - Optimisation de code
  - Multithreading

# CV-Qualifiers

- Pour chaque type, incluant les types incomplets, il existe trois autres « sous-types »
  - 1 `const` : type constant, accédé en lecture seule
  - 2 `volatile` : type volatile, peut être accédé et modifié par un processus extérieur
  - 3 `const volatile` : les deux en même temps
- Motivation : optimisations compilatoires
  - Un `const` peut parfois être passé par référence
  - Les instructions comprenant des volatiles ne peuvent être réordonnées
- Applications
  - Optimisation de code
  - Multithreading

# CV-Qualifiers

- Pour chaque type, incluant les types incomplets, il existe trois autres « sous-types »
  - 1 `const` : type constant, accédé en lecture seule
  - 2 `volatile` : type volatile, peut être accédé et modifié par un processus extérieur
  - 3 `const volatile` : les deux en même temps
- Motivation : optimisations compilatoires
  - Un `const` peut parfois être passé par référence
  - Les instructions comprenant des volatiles ne peuvent être réordonnées
- Applications
  - Optimisation de code
  - Multithreading

# CV-Qualifiers

- Pour chaque type, incluant les types incomplets, il existe trois autres « sous-types »
  - 1 `const` : type constant, accédé en lecture seule
  - 2 `volatile` : type volatile, peut être accédé et modifié par un processus extérieur
  - 3 `const volatile` : les deux en même temps
- Motivation : optimisations compilatoires
  - Un `const` peut parfois être passé par référence
  - Les instructions comprenant des volatiles ne peuvent être réordonnées
- Applications
  - Optimisation de code
  - Multithreading

# CV-Qualifiers

- Pour chaque type, incluant les types incomplets, il existe trois autres « sous-types »
  - 1 `const` : type constant, accédé en lecture seule
  - 2 `volatile` : type volatile, peut être accédé et modifié par un processus extérieur
  - 3 `const volatile` : les deux en même temps
- Motivation : optimisations compilatoires
  - Un `const` peut parfois être passé par référence
  - Les instructions comprenant des volatiles ne peuvent être réordonnées
- Applications
  - Optimisation de code
  - Multithreading

# CV-Qualifiers

- Pour chaque type, incluant les types incomplets, il existe trois autres « sous-types »
  - 1 `const` : type constant, accédé en lecture seule
  - 2 `volatile` : type volatile, peut être accédé et modifié par un processus extérieur
  - 3 `const volatile` : les deux en même temps
- Motivation : optimisations compilatoires
  - Un `const` peut parfois être passé par référence
  - Les instructions comprenant des volatiles ne peuvent être réordonnées
- Applications
  - Optimisation de code
  - Multithreading

# CV-Qualifiers

- Pour chaque type, incluant les types incomplets, il existe trois autres « sous-types »
  - 1 `const` : type constant, accédé en lecture seule
  - 2 `volatile` : type volatile, peut être accédé et modifié par un processus extérieur
  - 3 `const volatile` : les deux en même temps
- Motivation : optimisations compilatoires
  - Un `const` peut parfois être passé par référence
  - Les instructions comprenant des volatiles ne peuvent être réordonnées
- Applications
  - Optimisation de code
  - Multithreading



# CV-Qualifiers

- Pour chaque type, incluant les types incomplets, il existe trois autres « sous-types »
  - 1 `const` : type constant, accédé en lecture seule
  - 2 `volatile` : type volatile, peut être accédé et modifié par un processus extérieur
  - 3 `const volatile` : les deux en même temps
- Motivation : optimisations compilatoires
  - Un `const` peut parfois être passé par référence
  - Les instructions comprenant des volatiles ne peuvent être réordonnées
- Applications
  - Optimisation de code
  - Multithreading

# CV-Qualifiers

- Pour chaque type, incluant les types incomplets, il existe trois autres « sous-types »
  - 1 `const` : type constant, accédé en lecture seule
  - 2 `volatile` : type volatile, peut être accédé et modifié par un processus extérieur
  - 3 `const volatile` : les deux en même temps
- Motivation : optimisations compilatoires
  - Un `const` peut parfois être passé par référence
  - Les instructions comprenant des volatiles ne peuvent être réordonnées
- Applications
  - Optimisation de code
  - Multithreading

# Les constantes

## ■ En C, avant C11 : macros

- `#define` `PI` `3.14159265`
- Substitués textuellement
- Éviter dans la mesure du possible

## ■ En C++ et en C11, mot-clé `const`

- Pas `final`

## ■ En lecture seule

## ■ Syntaxe particulière pour les pointeurs (cf. Ch. 2)

- Pointeur constant d'entier, d'entier constant, constant d'entier constant

## ■ En C++, possibilité de créer des fonctions constantes

- Ne modifie pas les attributs de `this`

# Les constantes

## ■ En C, avant C11 : macros

- `#define` `PI 3.14159265`
- Substitués textuellement
- Éviter dans la mesure du possible

## ■ En C++ et en C11, mot-clé `const`

- Pas `final`

## ■ En lecture seule

## ■ Syntaxe particulière pour les pointeurs (cf. Ch. 2)

- Pointeur constant d'entier, d'entier constant, constant d'entier constant

## ■ En C++, possibilité de créer des fonctions constantes

- Ne modifie pas les attributs de `this`

# Les constantes

## ■ En C, avant C11 : macros

- `#define` `PI 3.14159265`
- Substitués textuellement
- Éviter dans la mesure du possible

## ■ En C++ et en C11, mot-clé `const`

- Pas `final`

## ■ En lecture seule

## ■ Syntaxe particulière pour les pointeurs (cf. Ch. 2)

- Pointeur constant d'entier, d'entier constant, constant d'entier constant

## ■ En C++, possibilité de créer des fonctions constantes

- Ne modifie pas les attributs de `this`

# Les constantes

- En C, avant C11 : macros
  - `#define` `PI 3.14159265`
  - Substitués textuellement
  - Éviter dans la mesure du possible
- En C++ et en C11, mot-clé `const`
  - Pas `final`
- En lecture seule
- Syntaxe particulière pour les pointeurs (cf. Ch. 2)
  - Pointeur constant d'entier, d'entier constant, constant d'entier constant
- En C++, possibilité de créer des fonctions constantes
  - Ne modifie pas les attributs de `this`

# Les constantes

## ■ En C, avant C11 : macros

- `#define` `PI 3.14159265`
- Substitués textuellement
- Éviter dans la mesure du possible

## ■ En C++ et en C11, mot-clé `const`

- Pas `final`
- En lecture seule
- Syntaxe particulière pour les pointeurs (cf. Ch. 2)
  - Pointeur constant d'entier, d'entier constant, constant d'entier constant
- En C++, possibilité de créer des fonctions constantes
  - Ne modifie pas les attributs de `this`

# Les constantes

- En C, avant C11 : macros
  - `#define` `PI 3.14159265`
  - Substitués textuellement
  - Éviter dans la mesure du possible
- En C++ et en C11, mot-clé `const`
  - Pas `final`
- En lecture seule
- Syntaxe particulière pour les pointeurs (cf. Ch. 2)
  - Pointeur constant d'entier, d'entier constant, constant d'entier constant
- En C++, possibilité de créer des fonctions constantes
  - Ne modifie pas les attributs de `this`



# Les constantes

- En C, avant C11 : macros
  - `#define` `PI 3.14159265`
  - Substitués textuellement
  - Éviter dans la mesure du possible
- En C++ et en C11, mot-clé `const`
  - Pas `final`
- En lecture seule
- Syntaxe particulière pour les pointeurs (cf. Ch. 2)
  - Pointeur constant d'entier, d'entier constant, constant d'entier constant
- En C++, possibilité de créer des fonctions constantes
  - Ne modifie pas les attributs de `this`

# Les constantes

- En C, avant C11 : macros
  - `#define` `PI 3.14159265`
  - Substitués textuellement
  - Éviter dans la mesure du possible
- En C++ et en C11, mot-clé `const`
  - Pas `final`
- En lecture seule
- Syntaxe particulière pour les pointeurs (cf. Ch. 2)
  - Pointeur constant d'entier, d'entier constant, constant d'entier constant
- En C++, possibilité de créer des fonctions constantes
  - Ne modifie pas les attributs de `this`

# Les constantes

- En C, avant C11 : macros
  - `#define` `PI 3.14159265`
  - Substitués textuellement
  - Éviter dans la mesure du possible
- En C++ et en C11, mot-clé `const`
  - Pas `final`
- En lecture seule
- Syntaxe particulière pour les pointeurs (cf. Ch. 2)
  - Pointeur constant d'entier, d'entier constant, constant d'entier constant
- En C++, possibilité de créer des fonctions constantes
  - Ne modifie pas les attributs de `this`

# Les constantes

- En C, avant C11 : macros
  - `#define` `PI 3.14159265`
  - Substitués textuellement
  - Éviter dans la mesure du possible
- En C++ et en C11, mot-clé `const`
  - Pas `final`
- En lecture seule
- Syntaxe particulière pour les pointeurs (cf. Ch. 2)
  - Pointeur constant d'entier, d'entier constant, constant d'entier constant
- En C++, possibilité de créer des fonctions constantes
  - Ne modifie pas les attributs de `this`

# Les constantes

- En C, avant C11 : macros
  - `#define` `PI 3.14159265`
  - Substitués textuellement
  - Éviter dans la mesure du possible
- En C++ et en C11, mot-clé `const`
  - Pas `final`
- En lecture seule
- Syntaxe particulière pour les pointeurs (cf. Ch. 2)
  - Pointeur constant d'entier, d'entier constant, constant d'entier constant
- En C++, possibilité de créer des fonctions constantes
  - Ne modifie pas les attributs de `this`

# Exemple

## ■ Fichier `const.c`

```
1 int main()  
2 {  
3     int i = 2;  
4     const int ci = 3;  
5     i += 2;  
6     // ci += 2; //ko  
7 }
```

# Chaînes de caractères

# Les immédiats

- Les littéraux (immédiats) chaînes de caractères peuvent être spécifiés comme en Java
  - Exemple : `"Hello_World!"`
- Les littéraux sont de type
  - `char[]` en C
  - `const char[]` en C++
- Les littéraux ont une durée de vie de tout le programme
  - Classe d'allocation statique (cf. Ch. 5)
- Les littéraux sont immuables (même les `char[]`)
  - Les modifier donne un comportement indéterminé
- Les littéraux sont implicitement zéro-terminés
  - Attention aux spécifications de taille
- Même caractères d'échappement qu'en Java



# Les immédiats

- Les littéraux (immédiats) chaînes de caractères peuvent être spécifiés comme en Java
  - Exemple : `"Hello_World!"`
- Les littéraux sont de type
  - `char[]` en C
  - `const char[]` en C++
- Les littéraux ont une durée de vie de tout le programme
  - Classe d'allocation statique (cf. Ch. 5)
- Les littéraux sont immuables (même les `char[]`)
  - Les modifier donne un comportement indéterminé
- Les littéraux sont implicitement zéro-terminés
  - Attention aux spécifications de taille
- Même caractères d'échappement qu'en Java

# Les immédiats

- Les littéraux (immédiats) chaînes de caractères peuvent être spécifiés comme en Java
  - Exemple : `"Hello_World!"`
- Les littéraux sont de type
  - `char[]` en C
  - `const char[]` en C++
- Les littéraux ont une durée de vie de tout le programme
  - Classe d'allocation statique (cf. Ch. 5)
- Les littéraux sont immuables (même les `char[]`)
  - Les modifier donne un comportement indéterminé
- Les littéraux sont implicitement zéro-terminés
  - Attention aux spécifications de taille
- Même caractères d'échappement qu'en Java

# Les immédiats

- Les littéraux (immédiats) chaînes de caractères peuvent être spécifiés comme en Java
  - Exemple : `"Hello_World!"`
- Les littéraux sont de type
  - `char[]` en C
  - `const char[]` en C++
- Les littéraux ont une durée de vie de tout le programme
  - Classe d'allocation statique (cf. Ch. 5)
- Les littéraux sont immuables (même les `char[]`)
  - Les modifier donne un comportement indéterminé
- Les littéraux sont implicitement zéro-terminés
  - Attention aux spécifications de taille
- Même caractères d'échappement qu'en Java

# Les immédiats

- Les littéraux (immédiats) chaînes de caractères peuvent être spécifiés comme en Java
  - Exemple : `"Hello_World!"`
- Les littéraux sont de type
  - `char []` en C
  - `const char []` en C++
- Les littéraux ont une durée de vie de tout le programme
  - Classe d'allocation statique (cf. Ch. 5)
- Les littéraux sont immuables (même les `char []`)
  - Les modifier donne un comportement indéterminé
- Les littéraux sont implicitement zéro-terminés
  - Attention aux spécifications de taille
- Même caractères d'échappement qu'en Java

# Les immédiats

- Les littéraux (immédiats) chaînes de caractères peuvent être spécifiés comme en Java
  - Exemple : `"Hello_World!"`
- Les littéraux sont de type
  - `char[]` en C
  - `const char[]` en C++
- Les littéraux ont une durée de vie de tout le programme
  - Classe d'allocation statique (cf. Ch. 5)
- Les littéraux sont immuables (même les `char[]`)
  - Les modifier donne un comportement indéterminé
- Les littéraux sont implicitement zéro-terminés
  - Attention aux spécifications de taille
- Même caractères d'échappement qu'en Java

# Les immédiats

- Les littéraux (immédiats) chaînes de caractères peuvent être spécifiés comme en Java
  - Exemple : `"Hello_World!"`
- Les littéraux sont de type
  - `char[]` en C
  - `const char[]` en C++
- Les littéraux ont une durée de vie de tout le programme
  - Classe d'allocation statique (cf. Ch. 5)
- Les littéraux sont immuables (même les `char[]`)
  - Les modifier donne un comportement indéterminé
- Les littéraux sont implicitement zéro-terminés
  - Attention aux spécifications de taille
- Même caractères d'échappement qu'en Java

# Les immédiats

- Les littéraux (immédiats) chaînes de caractères peuvent être spécifiés comme en Java
  - Exemple : `"Hello_World!"`
- Les littéraux sont de type
  - `char[]` en C
  - `const char[]` en C++
- Les littéraux ont une durée de vie de tout le programme
  - Classe d'allocation statique (cf. Ch. 5)
- Les littéraux sont immuables (même les `char[]`)
  - Les modifier donne un comportement indéterminé
- Les littéraux sont implicitement zéro-terminés
  - Attention aux spécifications de taille
- Même caractères d'échappement qu'en Java

# Les immédiats

- Les littéraux (immédiats) chaînes de caractères peuvent être spécifiés comme en Java
  - Exemple : `"Hello_World!"`
- Les littéraux sont de type
  - `char[]` en C
  - `const char[]` en C++
- Les littéraux ont une durée de vie de tout le programme
  - Classe d'allocation statique (cf. Ch. 5)
- Les littéraux sont immuables (même les `char[]`)
  - Les modifier donne un comportement indéterminé
- Les littéraux sont implicitement zéro-terminés
  - Attention aux spécifications de taille
- Même caractères d'échappement qu'en Java



# Les immédiats

- Les littéraux (immédiats) chaînes de caractères peuvent être spécifiés comme en Java
  - Exemple : `"Hello_World!"`
- Les littéraux sont de type
  - `char[]` en C
  - `const char[]` en C++
- Les littéraux ont une durée de vie de tout le programme
  - Classe d'allocation statique (cf. Ch. 5)
- Les littéraux sont immuables (même les `char[]`)
  - Les modifier donne un comportement indéterminé
- Les littéraux sont implicitement zéro-terminés
  - Attention aux spécifications de taille
- Même caractères d'échappement qu'en Java

# Les immédiats

- Les littéraux (immédiats) chaînes de caractères peuvent être spécifiés comme en Java
  - Exemple : `"Hello_World!"`
- Les littéraux sont de type
  - `char[]` en C
  - `const char[]` en C++
- Les littéraux ont une durée de vie de tout le programme
  - Classe d'allocation statique (cf. Ch. 5)
- Les littéraux sont immuables (même les `char[]`)
  - Les modifier donne un comportement indéterminé
- Les littéraux sont implicitement zéro-terminés
  - Attention aux spécifications de taille
- Même caractères d'échappement qu'en Java

# Les immédiats

- Les littéraux (immédiats) chaînes de caractères peuvent être spécifiés comme en Java
  - Exemple : `"Hello_World!"`
- Les littéraux sont de type
  - `char[]` en C
  - `const char[]` en C++
- Les littéraux ont une durée de vie de tout le programme
  - Classe d'allocation statique (cf. Ch. 5)
- Les littéraux sont immuables (même les `char[]`)
  - Les modifier donne un comportement indéterminé
- Les littéraux sont implicitement zéro-terminés
  - Attention aux spécifications de taille
- Même caractères d'échappement qu'en Java

# Illustration

## ■ Fichier `str-immediate.c`

```
1  int main()
2  {
3      char * s1 = "Hello"; //length = 5
4      char s2[] = "Hello";
5      printf("%s\n", s1);
6      printf("%s\n", s2);
7
8      char s3[5] = "Hello"; //?!
9      char s4[6] = "Hello";
10     char s5[20] = "Hello";
11
12     printf("%s | \n", s3);
13     printf("%s | \n", s4);
14     printf("%s | \n", s5);
15 }
```

# Fonctionnalités

- En C, il n'existe pas de classe string
  - Car il n'y a pas de classes
- On manipule directement des `char[]` et `const char[]`
- Taille : `strlen`
- Accès : `[]`, `strstr`
- Pas de modification possible si c'est un immédiat
  - Sinon, `strcpy`, `strcat`
- Pas de comparaison lexicographique possible de la forme `s1 < s2`. Utiliser `strcmp`.
- Parsing : `atoi`, `strtol`, `strtof`, `strtod`, `strtok`
  - L'interprétation stoppe sur la première espace
  - En cas d'échec, retourne zéro
- Validation de caractères : `isalnum`, `isalpha`, `isdigit`, `islower`, `isupper`

# Fonctionnalités

- En C, il n'existe pas de classe string
  - Car il n'y a pas de classes
- On manipule directement des `char[]` et `const char[]`
- Taille : `strlen`
- Accès : `[]`, `strstr`
- Pas de modification possible si c'est un immédiat
  - Sinon, `strcpy`, `strcat`
- Pas de comparaison lexicographique possible de la forme `s1 < s2`. Utiliser `strcmp`.
- Parsing : `atoi`, `strtol`, `strtof`, `strtod`, `strtok`
  - L'interprétation stoppe sur la première espace
  - En cas d'échec, retourne zéro
- Validation de caractères : `isalnum`, `isalpha`, `isdigit`, `islower`, `isupper`

# Fonctionnalités

- En C, il n'existe pas de classe string
  - Car il n'y a pas de classes
- On manipule directement des `char[]` et `const char[]`
- Taille : `strlen`
- Accès : `[], strstr`
- Pas de modification possible si c'est un immédiat
  - Sinon, `strcpy, strcat`
- Pas de comparaison lexicographique possible de la forme `s1 < s2`. Utiliser `strcmp`.
- Parsing : `atoi, strtol, strtod, strtok`
  - L'interprétation stoppe sur la première espace
  - En cas d'échec, retourne zéro
- Validation de caractères : `isalnum, isalpha, isdigit, islower, isupper`

# Fonctionnalités

- En C, il n'existe pas de classe string
  - Car il n'y a pas de classes
- On manipule directement des `char[]` et `const char[]`
- Taille : `strlen`
- Accès : `[], strstr`
- Pas de modification possible si c'est un immédiat
  - Sinon, `strcpy, strcat`
- Pas de comparaison lexicographique possible de la forme `s1 < s2`. Utiliser `strcmp`.
- Parsing : `atoi, strtol, strtod, strtok`
  - L'interprétation stoppe sur la première espace
  - En cas d'échec, retourne zéro
- Validation de caractères : `isalnum, isalpha, isdigit, islower, isupper`



# Fonctionnalités

- En C, il n'existe pas de classe string
  - Car il n'y a pas de classes
- On manipule directement des `char[]` et `const char[]`
- Taille : `strlen`
- Accès : `[], strstr`
- Pas de modification possible si c'est un immédiat
  - Sinon, `strcpy, strcat`
- Pas de comparaison lexicographique possible de la forme `s1 < s2`. Utiliser `strcmp`.
- Parsing : `atoi, strtol, strtod, strtok`
  - L'interprétation stoppe sur la première espace
  - En cas d'échec, retourne zéro
- Validation de caractères : `isalnum, isalpha, isdigit, islower, isupper`

# Fonctionnalités

- En C, il n'existe pas de classe string
  - Car il n'y a pas de classes
- On manipule directement des `char[]` et `const char[]`
- Taille : `strlen`
- Accès : `[], strstr`
- Pas de modification possible si c'est un immédiat
  - Sinon, `strcpy, strcat`
- Pas de comparaison lexicographique possible de la forme `s1 < s2`. Utiliser `strcmp`.
- Parsing : `atoi, strtol, strtod, strtok`
  - L'interprétation stoppe sur la première espace
  - En cas d'échec, retourne zéro
- Validation de caractères : `isalnum, isalpha, isdigit, islower, isupper`

# Fonctionnalités

- En C, il n'existe pas de classe string
  - Car il n'y a pas de classes
- On manipule directement des `char[]` et `const char[]`
- Taille : `strlen`
- Accès : `[], strstr`
- Pas de modification possible si c'est un immédiat
  - Sinon, `strcpy, strcat`
- Pas de comparaison lexicographique possible de la forme `s1 < s2`. Utiliser `strcmp`.
- Parsing : `atoi, strtol, strtod, strtok`
  - L'interprétation stoppe sur la première espace
  - En cas d'échec, retourne zéro
- Validation de caractères : `isalnum, isalpha, isdigit, islower, isupper`

# Fonctionnalités

- En C, il n'existe pas de classe string
  - Car il n'y a pas de classes
- On manipule directement des `char[]` et `const char[]`
- Taille : `strlen`
- Accès : `[], strstr`
- Pas de modification possible si c'est un immédiat
  - Sinon, `strcpy, strcat`
- Pas de comparaison lexicographique possible de la forme `s1 < s2`. Utiliser `strcmp`.
- Parsing : `atoi, strtol, strtod, strtok`
  - L'interprétation stoppe sur la première espace
  - En cas d'échec, retourne zéro
- Validation de caractères : `isalnum, isalpha, isdigit, islower, isupper`

# Fonctionnalités

- En C, il n'existe pas de classe string
  - Car il n'y a pas de classes
- On manipule directement des `char[]` et `const char[]`
- Taille : `strlen`
- Accès : `[]`, `strstr`
- Pas de modification possible si c'est un immédiat
  - Sinon, `strcpy`, `strcat`
- Pas de comparaison lexicographique possible de la forme `s1 < s2`. Utiliser `strcmp`.
- Parsing : `atoi`, `strtol`, `strtof`, `strtod`, `strtok`
  - L'interprétation stoppe sur la première espace
  - En cas d'échec, retourne zéro
- Validation de caractères : `isalnum`, `isalpha`, `isdigit`, `islower`, `isupper`

# Fonctionnalités

- En C, il n'existe pas de classe string
  - Car il n'y a pas de classes
- On manipule directement des `char[]` et `const char[]`
- Taille : `strlen`
- Accès : `[]`, `strstr`
- Pas de modification possible si c'est un immédiat
  - Sinon, `strcpy`, `strcat`
- Pas de comparaison lexicographique possible de la forme `s1 < s2`. Utiliser `strcmp`.
- Parsing : `atoi`, `strtol`, `strtof`, `strtod`, `strtok`
  - L'interprétation stoppe sur la première espace
  - En cas d'échec, retourne zéro
- Validation de caractères : `isalnum`, `isalpha`, `isdigit`, `islower`, `isupper`

# Fonctionnalités

- En C, il n'existe pas de classe string
  - Car il n'y a pas de classes
- On manipule directement des `char[]` et `const char[]`
- Taille : `strlen`
- Accès : `[]`, `strstr`
- Pas de modification possible si c'est un immédiat
  - Sinon, `strcpy`, `strcat`
- Pas de comparaison lexicographique possible de la forme `s1 < s2`. Utiliser `strcmp`.
- Parsing : `atoi`, `strtol`, `strtof`, `strtod`, `strtok`
  - L'interprétation stoppe sur la première espace
  - En cas d'échec, retourne zéro
- Validation de caractères : `isalnum`, `isalpha`, `isdigit`, `islower`, `isupper`

# Fonctionnalités

- En C, il n'existe pas de classe string
  - Car il n'y a pas de classes
- On manipule directement des `char[]` et `const char[]`
- Taille : `strlen`
- Accès : `[], strstr`
- Pas de modification possible si c'est un immédiat
  - Sinon, `strcpy, strcat`
- Pas de comparaison lexicographique possible de la forme `s1 < s2`. Utiliser `strcmp`.
- Parsing : `atoi, strtol, strtod, strtok`
  - L'interprétation stoppe sur la première espace
  - En cas d'échec, retourne zéro
- Validation de caractères : `isalnum, isalpha, isdigit, islower, isupper`



# Illustration (1/2)

## ■ Fichier `string-c.c`

```
1 int main()
2 {
3     char s1[] = "Hello_";
4     char s2[] = "World";
5
6     printf("Length=%lu_chars, _size=%lu_bytes\n", strlen(s1), sizeof(s1));
7     printf("Length=%lu_chars, _size=%lu_bytes\n", strlen(s2), sizeof(s2));
8
9     for(long i = 0; i < strlen(s1); i++)
10         printf("%c", s1[i]);
11     for(long i = 0; i < strlen(s2); i++)
12         printf("%c", s2[i]);
13     printf("\n");
14
15     // strcpy(s2, s1); //wrong
16     // strcpy(s1, s2); //also wrong
17     // strcat(s2, s1); // still wrong
18     // strcat(s1, s2); //again, wrong
19 }
```

## Illustration (2/2)

### ■ Fichier `string-c.c`

```
1  int main()
2  {
3      char s1[] = "Hello_";
4      char s2[] = "World";
5
6      char result[20]; //try with other values
7
8      strcpy(result, s2);
9      printf("%s\n", result);
10
11     strcpy(result, s1);
12     printf("%s\n", result);
13
14     char result2[strlen(s1) + strlen(s2) + 1]; //why + 1 ?
15     strcpy(result, s1);
16     strcat(result, s2);
17     printf("%s\n", result);
18 }
```

# Entrées / sorties

# Écriture : `printf`

- Écriture en sortie standard via `printf`
- Diverses options de formatage de type
  - `%d`, `%o`, `%x` : entiers
  - `%f`, `%e`, `%g` : flottants
  - `%s` : chaînes de caractère
  - `%p` : pointeur
- Diverses options de complétion
  - `-` : entrée alignée à gauche (par défaut, à droite)
  - `*` : spécifie la largeur de l'impression comme un paramètre
  - `n`, où  $n \in \mathbb{N}$  : spécifie la largeur de l'impression

# Écriture : `printf`

- Écriture en sortie standard via `printf`
- Diverses options de formatage de type
  - `%d`, `%o`, `%x` : entiers
  - `%f`, `%e`, `%g` : flottants
  - `%s` : chaînes de caractère
  - `%p` : pointeur
- Diverses options de complétion
  - `-` : entrée alignée à gauche (par défaut, à droite)
  - `*` : spécifie la largeur de l'impression comme un paramètre
  - `n`, où  $n \in \mathbb{N}$  : spécifie la largeur de l'impression

# Écriture : `printf`

- Écriture en sortie standard via `printf`
- Diverses options de formatage de type
  - `%d`, `%o`, `%x` : entiers
  - `%f`, `%e`, `%g` : flottants
  - `%s` : chaînes de caractère
  - `%p` : pointeur
- Diverses options de complétion
  - `-` : entrée alignée à gauche (par défaut, à droite)
  - `*` : spécifie la largeur de l'impression comme un paramètre
  - `n`, où  $n \in \mathbb{N}$  : spécifie la largeur de l'impression

# Écriture : `printf`

- Écriture en sortie standard via `printf`
- Diverses options de formatage de type
  - `%d`, `%o`, `%x` : entiers
  - `%f`, `%e`, `%g` : flottants
  - `%s` : chaînes de caractère
  - `%p` : pointeur
- Diverses options de complétion
  - `-` : entrée alignée à gauche (par défaut, à droite)
  - `*` : spécifie la largeur de l'impression comme un paramètre
  - `n`, où  $n \in \mathbb{N}$  : spécifie la largeur de l'impression

# Écriture : `printf`

- Écriture en sortie standard via `printf`
- Diverses options de formatage de type
  - `%d`, `%o`, `%x` : entiers
  - `%f`, `%e`, `%g` : flottants
  - `%s` : chaînes de caractère
  - `%p` : pointeur
- Diverses options de complétion
  - `-` : entrée alignée à gauche (par défaut, à droite)
  - `*` : spécifie la largeur de l'impression comme un paramètre
  - `n`, où  $n \in \mathbb{N}$  : spécifie la largeur de l'impression



# Écriture : `printf`

- Écriture en sortie standard via `printf`
- Diverses options de formatage de type
  - `%d`, `%o`, `%x` : entiers
  - `%f`, `%e`, `%g` : flottants
  - `%s` : chaînes de caractère
  - `%p` : pointeur
- Diverses options de complétion
  - `-` : entrée alignée à gauche (par défaut, à droite)
  - `*` : spécifie la largeur de l'impression comme un paramètre
  - `n`, où  $n \in \mathbb{N}$  : spécifie la largeur de l'impression

# Écriture : `printf`

- Écriture en sortie standard via `printf`
- Diverses options de formatage de type
  - `%d`, `%o`, `%x` : entiers
  - `%f`, `%e`, `%g` : flottants
  - `%s` : chaînes de caractère
  - `%p` : pointeur
- Diverses options de complétion
  - `-` : entrée alignée à gauche (par défaut, à droite)
  - `*` : spécifie la largeur de l'impression comme un paramètre
  - `n`, où  $n \in \mathbb{N}$  : spécifie la largeur de l'impression

# Écriture : `printf`

- Écriture en sortie standard via `printf`
- Diverses options de formatage de type
  - `%d`, `%o`, `%x` : entiers
  - `%f`, `%e`, `%g` : flottants
  - `%s` : chaînes de caractère
  - `%p` : pointeur
- Diverses options de complétion
  - `-` : entrée alignée à gauche (par défaut, à droite)
  - `*` : spécifie la largeur de l'impression comme un paramètre
  - `n`, où  $n \in \mathbb{N}$  : spécifie la largeur de l'impression

# Écriture : `printf`

- Écriture en sortie standard via `printf`
- Diverses options de formatage de type
  - `%d`, `%o`, `%x` : entiers
  - `%f`, `%e`, `%g` : flottants
  - `%s` : chaînes de caractère
  - `%p` : pointeur
- Diverses options de complétion
  - `-` : entrée alignée à gauche (par défaut, à droite)
  - `*` : spécifie la largeur de l'impression comme un paramètre
  - `n`, où  $n \in \mathbb{N}$  : spécifie la largeur de l'impression

# Écriture : `printf`

- Écriture en sortie standard via `printf`
- Diverses options de formatage de type
  - `%d`, `%o`, `%x` : entiers
  - `%f`, `%e`, `%g` : flottants
  - `%s` : chaînes de caractère
  - `%p` : pointeur
- Diverses options de complétion
  - `-` : entrée alignée à gauche (par défaut, à droite)
  - `*` : spécifie la largeur de l'impression comme un paramètre
  - `n`, où  $n \in \mathbb{N}$  : spécifie la largeur de l'impression

## ■ Fichier printf.c

```
1  int main()
2  {
3      printf("Strings :\n");
4      const char* s = "Hello";
5      printf("\t.%10s.\n\t.%-10s.\n\t.%.s.\n", s, s, 10, s);
6
7      printf("Characters :\t%c_%%\n", 65);
8
9      printf("Integers\n");
10     printf("\tDecimal :\t%i_d_%.6i_%.0i_%.+i_%.u\n", 1, 2, 3, 0, 0, 4, -1);
11     printf("\tHexadecimal :\t%x_%.x_%.X_%.#x\n", 5, 10, 10, 6);
12     printf("\tOctal :\t%o_%.#o_%.#o\n", 10, 10, 4);
13
14     printf("Floating_point\n");
15     printf("\tRounding :\t%f_%.0f_%.32f\n", 1.5, 1.5, 1.3);
16     printf("\tPadding :\t%05.2f_%.2f_%.5.2f\n", 1.5, 1.5, 1.5);
17     printf("\tScientific :\t%E_%.e\n", 1.5, 1.5);
18     printf("\tScientific :\t%g_%.g\n", 1.002, 0.000007);
19     printf("\tHexadecimal :\t%a_%.A\n", 1.5, 1.5);
20 }
```

■ Source : [cplusplus.com](http://cplusplus.com)

# Lecture : `scanf`

- `scanf` permet de lire des types de base au clavier
  - Difficile de l'altérer pour d'autres types
  - `setlocale` change le charset
- Fonctionne à l'aide du même système de format que `printf`
  - Si une entrée ne correspond pas au format, le paramètre correspondant est inaltéré
- Retourne le nombre de paramètres affectés avec succès
- Termine une lecture par « Entrée »
- Séparateurs : espace, `\n`, `\t`, `\v` et `\f`

## Exemple

```
int i; scanf("%d", &i);
```

# Lecture : `scanf`

- `scanf` permet de lire des types de base au clavier
  - Difficile de l'altérer pour d'autres types
    - `setlocale` change le charset
- Fonctionne à l'aide du même système de format que `printf`
  - Si une entrée ne correspond pas au format, le paramètre correspondant est inaltéré
- Retourne le nombre de paramètres affectés avec succès
- Termine une lecture par « Entrée »
- Séparateurs : espace, `\n`, `\t`, `\v` et `\f`

## Exemple

```
int i; scanf("%d", &i);
```



# Lecture : `scanf`

- `scanf` permet de lire des types de base au clavier
  - Difficile de l'altérer pour d'autres types
  - `setlocale` change le charset
- Fonctionne à l'aide du même système de format que `printf`
  - Si une entrée ne correspond pas au format, le paramètre correspondant est inaltéré
- Retourne le nombre de paramètres affectés avec succès
- Termine une lecture par « Entrée »
- Séparateurs : espace, `\n`, `\t`, `\v` et `\f`

## Exemple

```
int i; scanf("%d", &i);
```

# Lecture : `scanf`

- `scanf` permet de lire des types de base au clavier
  - Difficile de l'altérer pour d'autres types
  - `setlocale` change le charset
- Fonctionne à l'aide du même système de format que `printf`
  - Si une entrée ne correspond pas au format, le paramètre correspondant est inaltéré
- Retourne le nombre de paramètres affectés avec succès
- Termine une lecture par « Entrée »
- Séparateurs : espace, `\n`, `\t`, `\v` et `\f`

## Exemple

```
int i; scanf("%d", &i);
```

# Lecture : `scanf`

- `scanf` permet de lire des types de base au clavier
  - Difficile de l'altérer pour d'autres types
  - `setlocale` change le charset
- Fonctionne à l'aide du même système de format que `printf`
  - Si une entrée ne correspond pas au format, le paramètre correspondant est inaltéré
- Retourne le nombre de paramètres affectés avec succès
- Termine une lecture par « Entrée »
- Séparateurs : espace, `\n`, `\t`, `\v` et `\f`

## Exemple

```
int i; scanf("%d", &i);
```

## Lecture : `scanf`

- `scanf` permet de lire des types de base au clavier
  - Difficile de l'altérer pour d'autres types
  - `setlocale` change le charset
- Fonctionne à l'aide du même système de format que `printf`
  - Si une entrée ne correspond pas au format, le paramètre correspondant est inaltéré
- Retourne le nombre de paramètres affectés avec succès
- Termine une lecture par « Entrée »
- Séparateurs : espace, `\n`, `\t`, `\v` et `\f`

### Exemple

```
int i; scanf("%d", &i);
```

## Lecture : `scanf`

- `scanf` permet de lire des types de base au clavier
  - Difficile de l'altérer pour d'autres types
  - `setlocale` change le charset
- Fonctionne à l'aide du même système de format que `printf`
  - Si une entrée ne correspond pas au format, le paramètre correspondant est inaltéré
- Retourne le nombre de paramètres affectés avec succès
- Termine une lecture par « Entrée »
- Séparateurs : espace, `\n`, `\t`, `\v` et `\f`

### Exemple

```
int i; scanf("%d", &i);
```

## Lecture : `scanf`

- `scanf` permet de lire des types de base au clavier
  - Difficile de l'altérer pour d'autres types
  - `setlocale` change le charset
- Fonctionne à l'aide du même système de format que `printf`
  - Si une entrée ne correspond pas au format, le paramètre correspondant est inaltéré
- Retourne le nombre de paramètres affectés avec succès
- Termine une lecture par « Entrée »
- Séparateurs : espace, `\n`, `\t`, `\v` et `\f`

### Exemple

```
int i; scanf("%d", &i);
```

## Lecture : `scanf`

- `scanf` permet de lire des types de base au clavier
  - Difficile de l'altérer pour d'autres types
  - `setlocale` change le charset
- Fonctionne à l'aide du même système de format que `printf`
  - Si une entrée ne correspond pas au format, le paramètre correspondant est inaltéré
- Retourne le nombre de paramètres affectés avec succès
- Termine une lecture par « Entrée »
- Séparateurs : espace, `\n`, `\t`, `\v` et `\f`

### Exemple

```
■ int i; scanf("%d", &i);
```

## Lecture : `scanf`

- `scanf` permet de lire des types de base au clavier
  - Difficile de l'altérer pour d'autres types
  - `setlocale` change le charset
- Fonctionne à l'aide du même système de format que `printf`
  - Si une entrée ne correspond pas au format, le paramètre correspondant est inaltéré
- Retourne le nombre de paramètres affectés avec succès
- Termine une lecture par « Entrée »
- Séparateurs : espace, `\n`, `\t`, `\v` et `\f`

### Exemple

```
■ int i; scanf("%d", &i);
```



# Exemple

## ■ Fichier `scanf.c`

```
1  #include <stdio.h>
2  #include <locale.h>
3
4  int main()
5  {
6      int i;
7
8      printf("Tapez_un_entier_(decimal)\n");
9      // scanf("%d", i); //sneaky
10     scanf("%d", &i);
11
12     printf("Vous_avez_tapé%d\n", i);
13
14     printf("Tapez_un_entier_(hexadecimal)\n");
15     scanf("%x", &i);
16
17     printf("Vous_avez_tapé%d\n", i);
18 }
```

# Fonctionnement interne via un buffer

- 1 La première lecture lit une chaîne de caractères validée par « Entrer »
- 2 Cette chaîne est placée dans un buffer en mémoire
- 3 Le buffer est exploré caractère par caractère par `scanf` (ou `'>>'` en C++)
  - Un pointeur désigne le prochain caractère du buffer à lire (incrément automatique)
- 4 La plus longue entrée correspondante est retournée

## Fin de lecture

- Si le contenu du buffer ne suffit pas, `scanf` attend une nouvelle entrée
- Si une partie du buffer n'est pas lue, elle reste pour la prochaine entrée

# Fonctionnement interne via un buffer

- 1 La première lecture lit une chaîne de caractères validée par « Entrer »
- 2 Cette chaîne est placée dans un buffer en mémoire
- 3 Le buffer est exploré caractère par caractère par `scanf` (ou `'>>'` en C++)
  - Un pointeur désigne le prochain caractère du buffer à lire (incréméntation automatique)
- 4 La plus longue entrée correspondante est retournée

## Fin de lecture

- Si le contenu du buffer ne suffit pas, `scanf` attend une nouvelle entrée
- Si une partie du buffer n'est pas lue, elle reste pour la prochaine entrée

# Fonctionnement interne via un buffer

- 1 La première lecture lit une chaîne de caractères validée par « Entrer »
- 2 Cette chaîne est placée dans un buffer en mémoire
- 3 Le buffer est exploré caractère par caractère par `scanf` (ou `'>>'` en C++)
  - Un pointeur désigne le prochain caractère du buffer à lire (incréméntation automatique)
- 4 La plus longue entrée correspondante est retournée

## Fin de lecture

- Si le contenu du buffer ne suffit pas, `scanf` attend une nouvelle entrée
- Si une partie du buffer n'est pas lue, elle reste pour la prochaine entrée

# Fonctionnement interne via un buffer

- 1 La première lecture lit une chaîne de caractères validée par « Entrer »
- 2 Cette chaîne est placée dans un buffer en mémoire
- 3 Le buffer est exploré caractère par caractère par `scanf` (ou `'>>'` en C++)
  - Un pointeur désigne le prochain caractère du buffer à lire (incréméntation automatique)
- 4 La plus longue entrée correspondante est retournée

## Fin de lecture

- Si le contenu du buffer ne suffit pas, `scanf` attend une nouvelle entrée
- Si une partie du buffer n'est pas lue, elle reste pour la prochaine entrée

# Fonctionnement interne via un buffer

- 1 La première lecture lit une chaîne de caractères validée par « Entrer »
- 2 Cette chaîne est placée dans un buffer en mémoire
- 3 Le buffer est exploré caractère par caractère par `scanf` (ou `'>>'` en C++)
  - Un pointeur désigne le prochain caractère du buffer à lire (incréméntation automatique)
- 4 La plus longue entrée correspondante est retournée

## Fin de lecture

- Si le contenu du buffer ne suffit pas, `scanf` attend une nouvelle entrée
- Si une partie du buffer n'est pas lue, elle reste pour la prochaine entrée

# Fonctionnement interne via un buffer

- 1 La première lecture lit une chaîne de caractères validée par « Entrer »
- 2 Cette chaîne est placée dans un buffer en mémoire
- 3 Le buffer est exploré caractère par caractère par `scanf` (ou `'>>'` en C++)
  - Un pointeur désigne le prochain caractère du buffer à lire (incrémenté automatiquement)
- 4 La plus longue entrée correspondante est retournée

## Fin de lecture

- Si le contenu du buffer ne suffit pas, `scanf` attend une nouvelle entrée
- Si une partie du buffer n'est pas lue, elle reste pour la prochaine entrée

## Fonctionnement interne via un buffer

- 1 La première lecture lit une chaîne de caractères validée par « Entrer »
- 2 Cette chaîne est placée dans un buffer en mémoire
- 3 Le buffer est exploré caractère par caractère par `scanf` (ou `'>>'` en C++)
  - Un pointeur désigne le prochain caractère du buffer à lire (incrémentation automatique)
- 4 La plus longue entrée correspondante est retournée

### Fin de lecture

- Si le contenu du buffer ne suffit pas, `scanf` attend une nouvelle entrée
- Si une partie du buffer n'est pas lue, elle reste pour la prochaine entrée



## Fonctionnement interne via un buffer

- 1 La première lecture lit une chaîne de caractères validée par « Entrer »
- 2 Cette chaîne est placée dans un buffer en mémoire
- 3 Le buffer est exploré caractère par caractère par `scanf` (ou `'>>'` en C++)
  - Un pointeur désigne le prochain caractère du buffer à lire (incrémentation automatique)
- 4 La plus longue entrée correspondante est retournée

### Fin de lecture

- Si le contenu du buffer ne suffit pas, `scanf` attend une nouvelle entrée
- Si une partie du buffer n'est pas lue, elle reste pour la prochaine entrée

# Erreurs de lecture

## Extrait de code

```
■ int i; cin >> i; //on entre "12a"
```

- 1 La chaîne "12a" est placée dans le buffer
- 2 Les caractères '1' et '2' sont lus, le pointeur `ptr` est incrémenté
- 3 Le caractère 'a' est lu
- 4 12a n'est pas un nombre, on place donc 12 dans `i`
- 5 'a' reste pour la prochaine lecture, `ptr` n'est pas incrémenté

## Remarque

- Peut provoquer des désynchronisations et boucles infinies

# Erreurs de lecture

## Extrait de code

```
■ int i; cin >> i; //on entre "12a"
```

- 1 La chaîne "12a" est placée dans le buffer
- 2 Les caractères '1' et '2' sont lus, le pointeur `ptr` est incrémenté
- 3 Le caractère 'a' est lu
- 4 12a n'est pas un nombre, on place donc 12 dans `i`
- 5 'a' reste pour la prochaine lecture, `ptr` n'est pas incrémenté

## Remarque

- Peut provoquer des désynchronisations et boucles infinies

# Erreurs de lecture

## Extrait de code

```
■ int i; cin >> i; //on entre "12a"
```

- 1 La chaîne "12a" est placée dans le buffer
- 2 Les caractères '1' et '2' sont lus, le pointeur `ptr` est incrémenté
- 3 Le caractère 'a' est lu
- 4 12a n'est pas un nombre, on place donc 12 dans `i`
- 5 'a' reste pour la prochaine lecture, `ptr` n'est pas incrémenté

## Remarque

- Peut provoquer des désynchronisations et boucles infinies

# Erreurs de lecture

## Extrait de code

```
int i; cin >> i; //on entre "12a"
```

- 1 La chaîne "12a" est placée dans le buffer
- 2 Les caractères '1' et '2' sont lus, le pointeur `ptr` est incrémenté
- 3 Le caractère 'a' est lu
- 4 12a n'est pas un nombre, on place donc 12 dans `i`
- 5 'a' reste pour la prochaine lecture, `ptr` n'est pas incrémenté

## Remarque

- Peut provoquer des désynchronisations et boucles infinies

# Erreurs de lecture

## Extrait de code

```
■ int i; cin >> i; //on entre "12a"
```

- 1 La chaîne "12a" est placée dans le buffer
- 2 Les caractères '1' et '2' sont lus, le pointeur `ptr` est incrémenté
- 3 Le caractère 'a' est lu
- 4 12a n'est pas un nombre, on place donc 12 dans `i`
- 5 'a' reste pour la prochaine lecture, `ptr` n'est pas incrémenté

## Remarque

- Peut provoquer des désynchronisations et boucles infinies

# Erreurs de lecture

## Extrait de code

```
■ int i; cin >> i; //on entre "12a"
```

- 1 La chaîne "12a" est placée dans le buffer
- 2 Les caractères '1' et '2' sont lus, le pointeur `ptr` est incrémenté
- 3 Le caractère 'a' est lu
- 4 12a n'est pas un nombre, on place donc 12 dans `i`
- 5 'a' reste pour la prochaine lecture, `ptr` n'est pas incrémenté

## Remarque

- Peut provoquer des désynchronisations et boucles infinies

# Erreurs de lecture

## Extrait de code

```
■ int i; cin >> i; //on entre "12a"
```

- 1 La chaîne "12a" est placée dans le buffer
- 2 Les caractères '1' et '2' sont lus, le pointeur `ptr` est incrémenté
- 3 Le caractère 'a' est lu
- 4 12a n'est pas un nombre, on place donc 12 dans `i`
- 5 'a' reste pour la prochaine lecture, `ptr` n'est pas incrémenté

## Remarque

- Peut provoquer des désynchronisations et boucles infinies



# Erreurs de lecture

## Extrait de code

```
■ int i; cin >> i; //on entre "12a"
```

- 1 La chaîne "12a" est placée dans le buffer
- 2 Les caractères '1' et '2' sont lus, le pointeur `ptr` est incrémenté
- 3 Le caractère 'a' est lu
- 4 12a n'est pas un nombre, on place donc 12 dans `i`
- 5 'a' reste pour la prochaine lecture, `ptr` n'est pas incrémenté

## Remarque

- Peut provoquer des désynchronisations et boucles infinies

# Désynchronisation : exemple

## ■ Fichier `cin-error.c`

```
1  int n, p;  
2  printf("Donnez_une_valeur_pour_n\n"); // 1 2  
3  scanf("%d", &n);  
4  printf("Merci_pour_%d\n", n);  
5  printf("Donnez_une_valeur_pour_p\n");  
6  scanf("%d", &p);  
7  printf("Merci_pour_%d\n", p);
```

## ■ Même principe en C++ (fichier `cin-error.cpp`)

# Blocage : exemple

## ■ Fichier `cin-error.c`

```
1  int n = 12;
2  char c = 'a';
3  printf("Donnez_un_entier_et_un_caractère\n"); // x 25
4  scanf("%d", &n);
5  printf("Merci_pour_%d_et_%c\n", n, c);
6  printf("Donnez_un_caractère\n");
7  scanf("%c", &c);
8  printf("Merci_pour_%c\n", c);
```

## ■ Même principe en C++ (fichier `cin-error.cpp`)

# Boucle infinie : exemple

## ■ Fichier `cin-error.c`

```
1 //3
2 //à
3 int n;
4 do
5 {
6     printf("Tapez_un_nombre_entier\n");
7     scanf("%d", &n);
8     printf("Son_carré_est_%d\n", (n*n));
9 }
10 while(!n);
```

## ■ Même principe en C++ (fichier `cin-error.cpp`)

# Solutions

## ■ Lire une chaîne de caractère

### ■ Deux choix possibles

- Directement avec `cin / scanf`

- Avec `getline(cin, monstring)` (C++)

### ■ Inclure

- `stdlib.h` (C)

- `string.h` (C++)

### ■ Parser la chaîne

- C : `atoi` et `atof`

- C++ : `stoi(monstring)` et `stod(monstring)`

# Solutions

## ■ Lire une chaîne de caractère

## ■ Deux choix possibles

- 1 Directement avec `cin / scanf`
- 2 Avec `getline(cin, monstring)` (C++)

## ■ Inclure

- `stdlib.h` (C)
- `string.h` (C++)

## ■ Parser la chaîne

- C : `atoi` et `atof`
- C++ : `stoi(monstring)` et `stod(monstring)`

# Solutions

## ■ Lire une chaîne de caractère

## ■ Deux choix possibles

1 Directement avec `cin / scanf`

2 Avec `getline(cin, monstring)` (C++)

## ■ Inclure

■ `stdlib.h` (C)

■ `string.h` (C++)

## ■ Parser la chaîne

■ C : `atoi` et `atof`

■ C++ : `stoi(monstring)` et `stod(monstring)`

# Solutions

- Lire une chaîne de caractère
- Deux choix possibles
  - 1 Directement avec `cin / scanf`
  - 2 Avec `getline(cin, monstring)` (C++)
- Inclure
  - `stdlib.h` (C)
  - `string.h` (C++)
- Parser la chaîne
  - C : `atoi` et `atof`
  - C++ : `stoi(monstring)` et `stod(monstring)`



# Solutions

- Lire une chaîne de caractère
- Deux choix possibles
  - 1 Directement avec `cin / scanf`
  - 2 Avec `getline(cin, monstring)` (C++)
- Inclure
  - `stdlib.h` (C)
  - `string.h` (C++)
- Parser la chaîne
  - C : `atoi` et `atof`
  - C++ : `stoi(monstring)` et `stod(monstring)`

# Solutions

- Lire une chaîne de caractère
- Deux choix possibles
  - 1 Directement avec `cin / scanf`
  - 2 Avec `getline(cin, monstring)` (C++)
- Inclure
  - `stdlib.h` (C)
  - `string.h` (C++)
- Parser la chaîne
  - C : `atoi` et `atof`
  - C++ : `stoi(monstring)` et `stod(monstring)`

# Solutions

- Lire une chaîne de caractère
- Deux choix possibles
  - 1 Directement avec `cin / scanf`
  - 2 Avec `getline(cin, monstring)` (C++)
- Inclure
  - `stdlib.h` (C)
  - `string.h` (C++)
- Parser la chaîne
  - C : `atoi` et `atof`
  - C++ : `stoi(monstring)` et `stod(monstring)`

# Solutions

- Lire une chaîne de caractère
- Deux choix possibles
  - 1 Directement avec `cin / scanf`
  - 2 Avec `getline(cin, monstring)` (C++)
- Inclure
  - `stdlib.h` (C)
  - `string.h` (C++)
- Parser la chaîne
  - C : `atoi` et `atof`
  - C++ : `stoi(monstring)` et `stod(monstring)`

# Solutions

- Lire une chaîne de caractère
- Deux choix possibles
  - 1 Directement avec `cin / scanf`
  - 2 Avec `getline(cin, monstring)` (C++)
- Inclure
  - `stdlib.h` (C)
  - `string.h` (C++)
- Parser la chaîne
  - C : `atoi` et `atof`
  - C++ : `stoi(monstring)` et `stod(monstring)`

# Solutions

- Lire une chaîne de caractère
- Deux choix possibles
  - 1 Directement avec `cin / scanf`
  - 2 Avec `getline(cin, monstring)` (C++)
- Inclure
  - `stdlib.h` (C)
  - `string.h` (C++)
- Parser la chaîne
  - C : `atoi` **et** `atof`
  - C++ : `stoi(monstring)` **et** `stod(monstring)`

# Illustration

## ■ Fichier `cin-error.c`

```
1 printf("Tapez_un_entier_non_nul\n");
2
3 char buffer[20];
4 scanf("%s", buffer);
5 int i = atoi(buffer);
6 if(i != 0)
7     printf("Vous_avez_tapé_%d\n", i);
8 else
9     printf("Pas_un_entier\n");
```

# Illustration

## ■ Fichier `cin-error.cpp`

```
1  cout << "Tapez_un_entier" << endl;  
2  
3  string line;  
4  getline(cin, line);  
5  
6  try  
7  {  
8      int i = stoi(line);  
9      cout << "Vous_avez_tapé_" << i << endl;  
10 }  
11 catch(invalid_argument ex)  
12 {  
13     cout << "Pas_un_entier" << endl;  
14 }
```



# Expressions

# Instruction et expression

- Dans certains langages : concepts disjoints
  - Une expression a une valeur et ne fait pas de travail
  - Une instruction n'a pas de valeur et effectue un travail
- En C / C++, pas toujours...

## Exemple

```
■ cout << "Hello"<< endl;  
■ double y = a*x + b;  
■ int j = ++i;
```

# Instruction et expression

- Dans certains langages : concepts disjoints
  - Une expression a une valeur et ne fait pas de travail
  - Une instruction n'a pas de valeur et effectue un travail
- En C / C++, pas toujours...

## Exemple

```
■ cout << "Hello"<< endl;  
■ double y = a*x + b;  
■ int j = ++i;
```

# Instruction et expression

- Dans certains langages : concepts disjoints
  - Une expression a une valeur et ne fait pas de travail
  - Une instruction n'a pas de valeur et effectue un travail
- En C / C++, pas toujours...

## Exemple

```
■ cout << "Hello"<< endl;  
■ double y = a*x + b;  
■ int j = ++i;
```

# Instruction et expression

- Dans certains langages : concepts disjoints
  - Une expression a une valeur et ne fait pas de travail
  - Une instruction n'a pas de valeur et effectue un travail
- En C / C++, pas toujours...

## Exemple

```
■ cout << "Hello"<< endl;  
■ double y = a*x + b;  
■ int j = ++i;
```

# Instruction et expression

- Dans certains langages : concepts disjoints
  - Une expression a une valeur et ne fait pas de travail
  - Une instruction n'a pas de valeur et effectue un travail
- En C / C++, pas toujours...

## Exemple

```
■ cout << "Hello"<< endl;  
■ double y = a*x + b;  
■ int j = ++i;
```

# Instruction et expression

- Dans certains langages : concepts disjoints
  - Une expression a une valeur et ne fait pas de travail
  - Une instruction n'a pas de valeur et effectue un travail
- En C / C++, pas toujours...

## Exemple

```
■ cout << "Hello"<< endl;  
■ double y = a*x + b;  
■ int j = ++i;
```

# Instruction et expression

- Dans certains langages : concepts disjoints
  - Une expression a une valeur et ne fait pas de travail
  - Une instruction n'a pas de valeur et effectue un travail
- En C / C++, pas toujours...

## Exemple

```
■ cout << "Hello"<< endl;  
■ double y = a*x + b;  
■ int j = ++i;
```



# Instruction et expression

- Dans certains langages : concepts disjoints
  - Une expression a une valeur et ne fait pas de travail
  - Une instruction n'a pas de valeur et effectue un travail
- En C / C++, pas toujours...

## Exemple

- `cout << "Hello"<< endl;`
- `double y = a*x + b;`
- `int j = ++i;`

# Opérateurs

- Tous les opérateurs sont associatifs
  - De gauche à droite ou de droite à gauche

## Entre autres

- $+$ ,  $-$ ,  $*$ ,  $/$ ,  $\%$ ,  $\ll$ ,  $\gg$ ,  $\&\&$ ,  $||$ , etc. : associativité de gauche à droite
- $!$ ,  $*$  (indirection),  $\&$  (prise d'adresse),  $\text{sizeof}$ , etc. : associativité de droite à gauche
- Exemple :  $i + j + k$  est compilé comme  $(i + j) + k$
- Les opérateurs  $+$ ,  $-$ ,  $*$ ,  $/$ ,  $\%$ ,  $?:$ ,  $\&\&$ ,  $||$ ,  $==$ ,  $!$ ,  $!=$  ont la même signification qu'en Java

# Opérateurs

- Tous les opérateurs sont associatifs
  - De gauche à droite ou de droite à gauche

## Entre autres

- $+$ ,  $-$ ,  $*$ ,  $/$ ,  $\%$ ,  $\ll$ ,  $\gg$ ,  $\&\&$ ,  $||$ , etc. : associativité de gauche à droite
- $!$ ,  $*$  (indirection),  $\&$  (prise d'adresse),  $\text{sizeof}$ , etc. : associativité de droite à gauche
- Exemple :  $i + j + k$  est compilé comme  $(i + j) + k$
- Les opérateurs  $+$ ,  $-$ ,  $*$ ,  $/$ ,  $\%$ ,  $?:$ ,  $\&\&$ ,  $||$ ,  $==$ ,  $!$ ,  $!=$  ont la même signification qu'en Java

# Opérateurs

- Tous les opérateurs sont associatifs
  - De gauche à droite ou de droite à gauche

## Entre autres

- $+$ ,  $-$ ,  $*$ ,  $/$ ,  $\%$ ,  $\ll$ ,  $\gg$ ,  $\&\&$ ,  $||$ , etc. : associativité de gauche à droite
- $!$ ,  $*$  (indirection),  $\&$  (prise d'adresse), `sizeof`, etc. : associativité de droite à gauche
- Exemple :  $i + j + k$  est compilé comme  $(i + j) + k$
- Les opérateurs  $+$ ,  $-$ ,  $*$ ,  $/$ ,  $\%$ ,  $?:$ ,  $\&\&$ ,  $||$ ,  $==$ ,  $!$ ,  $!=$  ont la même signification qu'en Java

# Opérateurs

- Tous les opérateurs sont associatifs
  - De gauche à droite ou de droite à gauche

## Entre autres

- $+$ ,  $-$ ,  $*$ ,  $/$ ,  $\%$ ,  $\ll$ ,  $\gg$ ,  $\&\&$ ,  $||$ , **etc.** : associativité de gauche à droite
- $!$ ,  $*$  (indirection),  $\&$  (prise d'adresse), `sizeof`, **etc.** : associativité de droite à gauche
- Exemple : `i + j + k` est compilé comme  $(i + j) + k$
- Les opérateurs  $+$ ,  $-$ ,  $*$ ,  $/$ ,  $\%$ ,  $?:$ ,  $\&\&$ ,  $||$ ,  $==$ ,  $!$ ,  $!=$  ont la même signification qu'en Java

# Opérateurs

- Tous les opérateurs sont associatifs
  - De gauche à droite ou de droite à gauche

## Entre autres

- `+`, `-`, `*`, `/`, `%`, `<<`, `>>`, `&&`, `||`, etc. : associativité de gauche à droite
- `!`, `*` (indirection), `&` (prise d'adresse), `sizeof`, etc. : associativité de droite à gauche
- Exemple : `i + j + k` est compilé comme `(i + j) + k`
- Les opérateurs `+`, `-`, `*`, `/`, `%`, `?:`, `&&`, `||`, `==`, `!`, `!=` ont la même signification qu'en Java

# Opérateurs

- Tous les opérateurs sont associatifs
  - De gauche à droite ou de droite à gauche

## Entre autres

- $+$ ,  $-$ ,  $*$ ,  $/$ ,  $\%$ ,  $\ll$ ,  $\gg$ ,  $\&\&$ ,  $||$ , etc. : associativité de gauche à droite
- $!$ ,  $*$  (indirection),  $\&$  (prise d'adresse), `sizeof`, etc. : associativité de droite à gauche
- Exemple :  $i + j + k$  est compilé comme  $(i + j) + k$
- Les opérateurs  $+$ ,  $-$ ,  $*$ ,  $/$ ,  $\%$ ,  $?:$ ,  $\&\&$ ,  $||$ ,  $==$ ,  $!$ ,  $!=$  ont la même signification qu'en Java

# Opérateurs

- Tous les opérateurs sont associatifs
  - De gauche à droite ou de droite à gauche

## Entre autres

- `+`, `-`, `*`, `/`, `%`, `<<`, `>>`, `&&`, `||`, etc. : associativité de gauche à droite
- `!`, `*` (indirection), `&` (prise d'adresse), `sizeof`, etc. : associativité de droite à gauche
- Exemple : `i + j + k` est compilé comme `(i + j) + k`
- Les opérateurs `+`, `-`, `*`, `/`, `%`, `?:`, `&&`, `||`, `==`, `!`, `!=` ont la même signification qu'en Java



# Évaluation des opérandes

## Attention

- Le standard ne garantit pas l'ordre dans lequel les opérandes sont évalués
- Sauf `&&` et `||` (évaluation paresseuse), `?:` (nécessité)
- $f() + g() + h()$  est compilé comme  $(f() + g()) + h()$  mais  $h()$  peut être évalué avant, après ou entre  $f()$  et  $g()$
- Vrai aussi pour l'ordre d'évaluation des arguments des fonctions

## Hygiène de programmation

- Écrire du code où l'ordre d'évaluation est sans importance

# Évaluation des opérandes

## Attention

- Le standard ne garantit pas l'ordre dans lequel les opérandes sont évalués
- Sauf `&&` et `||` (évaluation paresseuse), `?:` (nécessité)
- $f() + g() + h()$  est compilé comme  $(f() + g()) + h()$  mais  $h()$  peut être évalué avant, après ou entre  $f()$  et  $g()$
- Vrai aussi pour l'ordre d'évaluation des arguments des fonctions

## Hygiène de programmation

- Écrire du code où l'ordre d'évaluation est sans importance

# Évaluation des opérandes

## Attention

- Le standard ne garantit pas l'ordre dans lequel les opérandes sont évalués
  - Sauf `&&` et `||` (évaluation paresseuse), `?:` (nécessité)
- `f() + g() + h()` est compilé comme `(f() + g()) + h()` mais `h()` peut être évalué avant, après ou entre `f()` et `g()`
- Vrai aussi pour l'ordre d'évaluation des arguments des fonctions

## Hygiène de programmation

- Écrire du code où l'ordre d'évaluation est sans importance

# Évaluation des opérandes

## Attention

- Le standard ne garantit pas l'ordre dans lequel les opérandes sont évalués
- Sauf `&&` et `||` (évaluation paresseuse), `?:` (nécessité)
- $f() + g() + h()$  est compilé comme  $(f() + g()) + h()$  mais  $h()$  peut être évalué avant, après ou entre  $f()$  et  $g()$
- Vrai aussi pour l'ordre d'évaluation des arguments des fonctions

## Hygiène de programmation

- Écrire du code où l'ordre d'évaluation est sans importance

# Évaluation des opérandes

## Attention

- Le standard ne garantit pas l'ordre dans lequel les opérandes sont évalués
- Sauf `&&` et `||` (évaluation paresseuse), `?:` (nécessité)
- `f() + g() + h()` est compilé comme `(f() + g()) + h()` mais `h()` peut être évalué avant, après ou entre `f()` et `g()`
- Vrai aussi pour l'ordre d'évaluation des arguments des fonctions

## Hygiène de programmation

- Écrire du code où l'ordre d'évaluation est sans importance

# Évaluation des opérandes

## Attention

- Le standard ne garantit pas l'ordre dans lequel les opérandes sont évalués
- Sauf `&&` et `||` (évaluation paresseuse), `?:` (nécessité)
- $f() + g() + h()$  est compilé comme  $(f() + g()) + h()$  mais  $h()$  peut être évalué avant, après ou entre  $f()$  et  $g()$
- Vrai aussi pour l'ordre d'évaluation des arguments des fonctions

## Hygiène de programmation

- Écrire du code où l'ordre d'évaluation est sans importance

# Évaluation des opérandes

## Attention

- Le standard ne garantit pas l'ordre dans lequel les opérandes sont évalués
- Sauf `&&` et `||` (évaluation paresseuse), `?:` (nécessité)
- $f() + g() + h()$  est compilé comme  $(f() + g()) + h()$  mais  $h()$  peut être évalué avant, après ou entre  $f()$  et  $g()$
- Vrai aussi pour l'ordre d'évaluation des arguments des fonctions

## Hygiène de programmation

- Écrire du code où l'ordre d'évaluation est sans importance

# Exemple

- Aucune des lignes ci-dessous n'a de comportement défini

```
1 i = ++i + i++;  
2 i = i++ + 1;  
3 i = ++i + 1; //well-defined in C++11  
4 ++ ++i; //well-defined in C++11  
5 f(++i, ++i);  
6 f(i = -1, i = -1);  
7 cout << i << i++;  
8 a[i] = i++;
```

- N'écrivez pas ça...



# Erreurs numériques (1/2)

- Certains calculs peuvent empêcher un opérateur de fournir un résultat correct

## Dépassement de capacité

- Résultat de calcul trop grand (en valeur absolue)
- Comportement indéterminé pour les entiers
- $+\text{INF}$  ou  $-\text{INF}$  pour les flottants

## Sous-dépassement de capacité

- Résultat de calcul trop petit (en valeur absolue)
- Conduit à un résultat nul, nombre indistinguishables, ou arrêt brutal

# Erreurs numériques (1/2)

- Certains calculs peuvent empêcher un opérateur de fournir un résultat correct

## Dépassement de capacité

- Résultat de calcul trop grand (en valeur absolue)
- Comportement indéterminé pour les entiers
- $+INF$  ou  $-INF$  pour les flottants

## Sous-dépassement de capacité

- Résultat de calcul trop petit (en valeur absolue)
- Conduit à un résultat nul, nombre indistinguishables, ou arrêt brutal

# Erreurs numériques (1/2)

- Certains calculs peuvent empêcher un opérateur de fournir un résultat correct

## Dépassement de capacité

- Résultat de calcul trop grand (en valeur absolue)
- Comportement indéterminé pour les entiers
- $+INF$  ou  $-INF$  pour les flottants

## Sous-dépassement de capacité

- Résultat de calcul trop petit (en valeur absolue)
- Conduit à un résultat nul, nombre indistinguishables, ou arrêt brutal

# Erreurs numériques (1/2)

- Certains calculs peuvent empêcher un opérateur de fournir un résultat correct

## Dépassement de capacité

- Résultat de calcul trop grand (en valeur absolue)
- Comportement indéterminé pour les entiers
- $+INF$  ou  $-INF$  pour les flottants

## Sous-dépassement de capacité

- Résultat de calcul trop petit (en valeur absolue)
- Conduit à un résultat nul, nombre indistinguishables, ou arrêt brutal

# Erreurs numériques (1/2)

- Certains calculs peuvent empêcher un opérateur de fournir un résultat correct

## Dépassement de capacité

- Résultat de calcul trop grand (en valeur absolue)
- Comportement indéterminé pour les entiers
- $+INF$  ou  $-INF$  pour les flottants

## Sous-dépassement de capacité

- Résultat de calcul trop petit (en valeur absolue)
- Conduit à un résultat nul, nombre indistinguishables, ou arrêt brutal

# Erreurs numériques (1/2)

- Certains calculs peuvent empêcher un opérateur de fournir un résultat correct

## Dépassement de capacité

- Résultat de calcul trop grand (en valeur absolue)
- Comportement indéterminé pour les entiers
- $+INF$  ou  $-INF$  pour les flottants

## Sous-dépassement de capacité

- Résultat de calcul trop petit (en valeur absolue)
- Conduit à un résultat nul, nombre indistinguishables, ou arrêt brutal

# Erreurs numériques (1/2)

- Certains calculs peuvent empêcher un opérateur de fournir un résultat correct

## Dépassement de capacité

- Résultat de calcul trop grand (en valeur absolue)
- Comportement indéterminé pour les entiers
- $+INF$  ou  $-INF$  pour les flottants

## Sous-dépassement de capacité

- Résultat de calcul trop petit (en valeur absolue)
- Conduit à un résultat nul, nombre indistinguishables, ou arrêt brutal

# Erreurs numériques (1/2)

- Certains calculs peuvent empêcher un opérateur de fournir un résultat correct

## Dépassement de capacité

- Résultat de calcul trop grand (en valeur absolue)
- Comportement indéterminé pour les entiers
- $+INF$  ou  $-INF$  pour les flottants

## Sous-dépassement de capacité

- Résultat de calcul trop petit (en valeur absolue)
- Conduit à un résultat nul, nombre indistinguishables, ou arrêt brutal



## Erreurs numériques (2/2)

- Certains calculs peuvent empêcher un opérateur de fournir un résultat correct

### Division par zéro

- Donne le résultat  $+\text{INF}$ ,  $-\text{INF}$  ou NaN
- Peut provoquer un arrêt brutal

### Propagation d'erreur

- Certains calculs sont entachés d'erreurs
- Des opérations successives peuvent grandement faire changer le résultat
- Très complexe à gérer (pas l'objet de ce cours)

## Erreurs numériques (2/2)

- Certains calculs peuvent empêcher un opérateur de fournir un résultat correct

### Division par zéro

- Donne le résultat  $+\text{INF}$ ,  $-\text{INF}$  ou NaN
- Peut provoquer un arrêt brutal

### Propagation d'erreur

- Certains calculs sont entachés d'erreurs
- Des opérations successives peuvent grandement faire changer le résultat
- Très complexe à gérer (pas l'objet de ce cours)

## Erreurs numériques (2/2)

- Certains calculs peuvent empêcher un opérateur de fournir un résultat correct

### Division par zéro

- Donne le résultat  $+\text{INF}$ ,  $-\text{INF}$  ou NaN
- Peut provoquer un arrêt brutal

### Propagation d'erreur

- Certains calculs sont entachés d'erreurs
- Des opérations successives peuvent grandement faire changer le résultat
- Très complexe à gérer (pas l'objet de ce cours)

## Erreurs numériques (2/2)

- Certains calculs peuvent empêcher un opérateur de fournir un résultat correct

### Division par zéro

- Donne le résultat  $+\text{INF}$ ,  $-\text{INF}$  ou NaN
- Peut provoquer un arrêt brutal

### Propagation d'erreur

- Certains calculs sont entachés d'erreurs
- Des opérations successives peuvent grandement faire changer le résultat
- Très complexe à gérer (pas l'objet de ce cours)

## Erreurs numériques (2/2)

- Certains calculs peuvent empêcher un opérateur de fournir un résultat correct

### Division par zéro

- Donne le résultat  $+\text{INF}$ ,  $-\text{INF}$  ou NaN
- Peut provoquer un arrêt brutal

### Propagation d'erreur

- Certains calculs sont entachés d'erreurs
- Des opérations successives peuvent grandement faire changer le résultat
- Très complexe à gérer (pas l'objet de ce cours)

## Erreurs numériques (2/2)

- Certains calculs peuvent empêcher un opérateur de fournir un résultat correct

### Division par zéro

- Donne le résultat  $+\text{INF}$ ,  $-\text{INF}$  ou NaN
- Peut provoquer un arrêt brutal

### Propagation d'erreur

- Certains calculs sont entachés d'erreurs
- Des opérations successives peuvent grandement faire changer le résultat
- Très complexe à gérer (pas l'objet de ce cours)

## Erreurs numériques (2/2)

- Certains calculs peuvent empêcher un opérateur de fournir un résultat correct

### Division par zéro

- Donne le résultat  $+\text{INF}$ ,  $-\text{INF}$  ou NaN
- Peut provoquer un arrêt brutal

### Propagation d'erreur

- Certains calculs sont entachés d'erreurs
- Des opérations successives peuvent grandement faire changer le résultat
- Très complexe à gérer (pas l'objet de ce cours)

## Erreurs numériques (2/2)

- Certains calculs peuvent empêcher un opérateur de fournir un résultat correct

### Division par zéro

- Donne le résultat  $+\text{INF}$ ,  $-\text{INF}$  ou NaN
- Peut provoquer un arrêt brutal

### Propagation d'erreur

- Certains calculs sont entachés d'erreurs
- Des opérations successives peuvent grandement faire changer le résultat
- Très complexe à gérer (pas l'objet de ce cours)



# Exemple

- Fichier `num-error.c`
- Même principe en C++

```
1  int main()  
2  {  
3      int x = INT_MAX;  
4      printf("%d\n", x);  
5      x+=x; // overflow  
6      printf("%d\n\n", x);  
7  
8      float y = 1E30;  
9      printf("%f\n", y);  
10     printf("%f\n\n", y / y / y / y);  
11  
12     double zero = 0;  
13     printf("%f\n", x / zero);  
14 }
```

# Arithmétique flottante

## ■ Le standard ne garantit pas la norme IEEE 754

- Un peu de support pour IEC 559
- Peu de prérequis hardware
- Seule l'associativité est requise

- $a + b == b + a ?$

- $a + b == a + b ?$

## Pourquoi tant de haine ?

- Le C / C++ est proche de la machine, l'arithmétique flottante est *complexe*, et le langage essaye d'être peu restrictif sur le hardware

# Arithmétique flottante

- Le standard ne garantit pas la norme IEEE 754
  - Un peu de support pour IEC 559
- Peu de prérequis hardware
- Seule l'associativité est requise
  - $a + b == b + a$  ?
  - $a + b == a + b$  ?

## Pourquoi tant de haine ?

- Le C / C++ est proche de la machine, l'arithmétique flottante est *complexe*, et le langage essaye d'être peu restrictif sur le hardware

# Arithmétique flottante

- Le standard ne garantit pas la norme IEEE 754
  - Un peu de support pour IEC 559
- Peu de prérequis hardware
- Seule l'associativité est requise
  - $a + b == b + a$  ?
  - $a + b == a + b$  ?

## Pourquoi tant de haine ?

- Le C / C++ est proche de la machine, l'arithmétique flottante est *complexe*, et le langage essaye d'être peu restrictif sur le hardware

# Arithmétique flottante

- Le standard ne garantit pas la norme IEEE 754
  - Un peu de support pour IEC 559
- Peu de prérequis hardware
- Seule l'associativité est requise
  - $a + b == b + a ?$
  - $a + b == a + b ?$

Pourquoi tant de haine ?

- Le C / C++ est proche de la machine, l'arithmétique flottante est *complexe*, et le langage essaye d'être peu restrictif sur le hardware

# Arithmétique flottante

- Le standard ne garantit pas la norme IEEE 754
  - Un peu de support pour IEC 559
- Peu de prérequis hardware
- Seule l'associativité est requise
  - $a + b == b + a ?$
  - $a + b == a + b ?$

Pourquoi tant de haine ?

- Le C / C++ est proche de la machine, l'arithmétique flottante est *complexe*, et le langage essaye d'être peu restrictif sur le hardware

# Arithmétique flottante

- Le standard ne garantit pas la norme IEEE 754
  - Un peu de support pour IEC 559
- Peu de prérequis hardware
- Seule l'associativité est requise
  - $a + b == b + a ?$
  - $a + b == a + b ?$

Pourquoi tant de haine ?

- Le C / C++ est proche de la machine, l'arithmétique flottante est *complexe*, et le langage essaye d'être peu restrictif sur le hardware

# Arithmétique flottante

- Le standard ne garantit pas la norme IEEE 754
  - Un peu de support pour IEC 559
- Peu de prérequis hardware
- Seule l'associativité est requise
  - $a + b == b + a ?$
  - $a + b == a + b ?$

## Pourquoi tant de haine ?

- Le C / C++ est proche de la machine, l'arithmétique flottante est *complexe*, et le langage essaye d'être peu restrictif sur le hardware



# Arithmétique flottante

- Le standard ne garantit pas la norme IEEE 754
  - Un peu de support pour IEC 559
- Peu de prérequis hardware
- Seule l'associativité est requise
  - $a + b == b + a ?$
  - $a + b == a + b ?$

## Pourquoi tant de haine ?

- Le C / C++ est proche de la machine, l'arithmétique flottante est *complexe*, et le langage essaye d'être peu restrictif sur le hardware

# Opérateur d'incrémentation

- Même principe qu'en Java
- En C / C++, une instruction de type `x++` possède une valeur
- Deux types d'utilisation
  - Post-incrémentation `x++` : valeur avant incrémentation
  - Pré-incrémentation `++x` : valeur après incrémentation

## Exemple

```
■ int i = 5; int n = ++i - 5;
```

- Même principe pour la décrémentation

## Hygiène de programmation

- Évitez le code obscur

# Opérateur d'incrémentation

- Même principe qu'en Java
- En C / C++, une instruction de type `x++` possède une valeur
- Deux types d'utilisation
  - Post-incrémentation `x++` : valeur avant incrémentation
  - Pré-incrémentation `++x` : valeur après incrémentation

## Exemple

```
int i = 5; int n = ++i - 5;
```

- Même principe pour la décrémentation

## Hygiène de programmation

- Évitez le code obscur

# Opérateur d'incrémentation

- Même principe qu'en Java
- En C / C++, une instruction de type `x++` possède une valeur
- Deux types d'utilisation
  - 1 Post-incrémentation `x++` : valeur avant incrément
  - 2 Pré-incrémentation `++x` : valeur après incrément

## Exemple

```
int i = 5; int n = ++i - 5;
```

- Même principe pour la décrémentation

## Hygiène de programmation

- Évitez le code obscur

# Opérateur d'incrémentation

- Même principe qu'en Java
- En C / C++, une instruction de type `x++` possède une valeur
- Deux types d'utilisation
  - 1 Post-incrémentation `x++` : valeur avant incrémentation
  - 2 Pré-incrémentation `++x` : valeur après incrémentation

## Exemple

```
int i = 5; int n = ++i - 5;
```

- Même principe pour la décrémentation

## Hygiène de programmation

- Évitez le code obscur

# Opérateur d'incrémentation

- Même principe qu'en Java
- En C / C++, une instruction de type `x++` possède une valeur
- Deux types d'utilisation
  - 1 Post-incrémentation `x++` : valeur avant incrément
  - 2 Pré-incrémentation `++x` : valeur après incrément

## Exemple

```
int i = 5; int n = ++i - 5;
```

- Même principe pour la décrémentation

## Hygiène de programmation

- Évitez le code obscur

# Opérateur d'incrémentation

- Même principe qu'en Java
- En C / C++, une instruction de type `x++` possède une valeur
- Deux types d'utilisation
  - 1 Post-incrémentation `x++` : valeur avant incrémentation
  - 2 Pré-incrémentation `++x` : valeur après incrémentation

## Exemple

```
■ int i = 5; int n = ++i - 5;
```

- Même principe pour la décrémentation

## Hygiène de programmation

- Évitez le code obscur

# Opérateur d'incrémentation

- Même principe qu'en Java
- En C / C++, une instruction de type `x++` possède une valeur
- Deux types d'utilisation
  - 1 Post-incrémentation `x++` : valeur avant incrémentation
  - 2 Pré-incrémentation `++x` : valeur après incrémentation

## Exemple

```
int i = 5; int n = ++i - 5;
```

- Même principe pour la décrémentation

## Hygiène de programmation

- Évitez le code obscur



# Opérateur d'incrémentation

- Même principe qu'en Java
- En C / C++, une instruction de type `x++` possède une valeur
- Deux types d'utilisation
  - 1 Post-incrémentation `x++` : valeur avant incrément
  - 2 Pré-incrémentation `++x` : valeur après incrément

## Exemple

```
int i = 5; int n = ++i - 5;
```

- Même principe pour la décrémentation

## Hygiène de programmation

- Évitez le code obscur

# Opérateur d'incrémentation

- Même principe qu'en Java
- En C / C++, une instruction de type `x++` possède une valeur
- Deux types d'utilisation
  - 1 Post-incrémentation `x++` : valeur avant incrémentation
  - 2 Pré-incrémentation `++x` : valeur après incrémentation

## Exemple

```
int i = 5; int n = ++i - 5;
```

- Même principe pour la décrémentation

## Hygiène de programmation

- Évitez le code obscur

# Opérateur d'incrémentation

- Même principe qu'en Java
- En C / C++, une instruction de type `x++` possède une valeur
- Deux types d'utilisation
  - 1 Post-incrémentation `x++` : valeur avant incrémentation
  - 2 Pré-incrémentation `++x` : valeur après incrémentation

## Exemple

```
int i = 5; int n = ++i - 5;
```

- Même principe pour la décrémentation

## Hygiène de programmation

- Évitez le code obscur

# Exemple

- Fichier `increment.c`
- Même principe en C++

```
1  int main()
2  {
3      int x = 3 + 2;
4      printf("%d\n", x);
5
6      x++;
7      printf("%d\n", x);
8      ++x;
9      printf("%d\n\n", x);
10
11     int y = x++;
12     printf("%d\n", x);
13     printf("%d\n\n", y);
14
15     int z = ++x;
16     printf("%d\n", x);
17     printf("%d\n", z);
18 }
```

# Conversions numériques implicites

- La plupart opérateurs ne sont définis que pour des opérandes de même type (+ entier et + flottant)
- Si les opérandes ne sont pas de même type : conversion, promotion ou tronquage
- Opérateurs arithmétiques et logiques
  - Si l'un des arguments est flottant, l'autre est converti en flottant
  - Sinon, l'argument de codage plus petit est converti dans le codage de l'autre
- Appel de fonction, retour de fonction et affectation : conversion dans la variable réceptrice (même si tronquage)

## Hygiène de programmation

- Convertir explicitement, ou utiliser les mêmes types

# Conversions numériques implicites

- La plupart opérateurs ne sont définis que pour des opérandes de même type (+ entier et + flottant)
- Si les opérandes ne sont pas de même type : conversion, promotion ou tronquage
- Opérateurs arithmétiques et logiques
  - Si l'un des arguments est flottant, l'autre est converti en flottant
  - Sinon, l'argument de codage plus petit est converti dans le codage de l'autre
- Appel de fonction, retour de fonction et affectation : conversion dans la variable réceptrice (même si tronquage)

## Hygiène de programmation

- Convertir explicitement, ou utiliser les mêmes types

# Conversions numériques implicites

- La plupart opérateurs ne sont définis que pour des opérandes de même type (+ entier et + flottant)
- Si les opérandes ne sont pas de même type : conversion, promotion ou tronquage
- Opérateurs arithmétiques et logiques
  - 1 Si l'un des arguments est flottant, l'autre est converti en flottant
    - `long > double > double > float`
  - 2 Sinon, l'argument de codage plus petit est converti
    - Même en cas de `signed` et `unsigned`
- Appel de fonction, retour de fonction et affectation : conversion dans la variable réceptrice (même si tronquage)

## Hygiène de programmation

- Convertir explicitement, ou utiliser les mêmes types

# Conversions numériques implicites

- La plupart opérateurs ne sont définis que pour des opérandes de même type (+ entier et + flottant)
- Si les opérandes ne sont pas de même type : conversion, promotion ou tronquage
- Opérateurs arithmétiques et logiques
  - 1 Si l'un des arguments est flottant, l'autre est converti en flottant
    - `long double > double > float`
  - 2 Sinon, l'argument de codage plus petit est converti
    - Même en cas de `signed` et `unsigned`
- Appel de fonction, retour de fonction et affectation : conversion dans la variable réceptrice (même si tronquage)

## Hygiène de programmation

- Convertir explicitement, ou utiliser les mêmes types



# Conversions numériques implicites

- La plupart opérateurs ne sont définis que pour des opérandes de même type (+ entier et + flottant)
- Si les opérandes ne sont pas de même type : conversion, promotion ou tronquage
- Opérateurs arithmétiques et logiques
  - 1 Si l'un des arguments est flottant, l'autre est converti en flottant
    - `long double > double > float`
  - 2 Sinon, l'argument de codage plus petit est converti
    - Même en cas de `signed` et `unsigned`
- Appel de fonction, retour de fonction et affectation : conversion dans la variable réceptrice (même si tronquage)

## Hygiène de programmation

- Convertir explicitement, ou utiliser les mêmes types

# Conversions numériques implicites

- La plupart opérateurs ne sont définis que pour des opérandes de même type (+ entier et + flottant)
- Si les opérandes ne sont pas de même type : conversion, promotion ou tronquage
- Opérateurs arithmétiques et logiques
  - 1 Si l'un des arguments est flottant, l'autre est converti en flottant
    - `long double > double > float`
  - 2 Sinon, l'argument de codage plus petit est converti
    - Même en cas de `signed` et `unsigned`
- Appel de fonction, retour de fonction et affectation : conversion dans la variable réceptrice (même si tronquage)

## Hygiène de programmation

- Convertir explicitement, ou utiliser les mêmes types

# Conversions numériques implicites

- La plupart opérateurs ne sont définis que pour des opérandes de même type (+ entier et + flottant)
- Si les opérandes ne sont pas de même type : conversion, promotion ou tronquage
- Opérateurs arithmétiques et logiques
  - 1 Si l'un des arguments est flottant, l'autre est converti en flottant
    - `long double > double > float`
  - 2 Sinon, l'argument de codage plus petit est converti
    - Même en cas de `signed` et `unsigned`
- Appel de fonction, retour de fonction et affectation : conversion dans la variable réceptrice (même si tronquage)

## Hygiène de programmation

- Convertir explicitement, ou utiliser les mêmes types

# Conversions numériques implicites

- La plupart opérateurs ne sont définis que pour des opérandes de même type (+ entier et + flottant)
- Si les opérandes ne sont pas de même type : conversion, promotion ou tronquage
- Opérateurs arithmétiques et logiques
  - 1 Si l'un des arguments est flottant, l'autre est converti en flottant
    - `long double > double > float`
  - 2 Sinon, l'argument de codage plus petit est converti
    - Même en cas de `signed` et `unsigned`
- Appel de fonction, retour de fonction et affectation : conversion dans la variable réceptrice (même si tronquage)

## Hygiène de programmation

- Convertir explicitement, ou utiliser les mêmes types

# Conversions numériques implicites

- La plupart opérateurs ne sont définis que pour des opérandes de même type (+ entier et + flottant)
- Si les opérandes ne sont pas de même type : conversion, promotion ou tronquage
- Opérateurs arithmétiques et logiques
  - 1 Si l'un des arguments est flottant, l'autre est converti en flottant
    - `long double > double > float`
  - 2 Sinon, l'argument de codage plus petit est converti
    - Même en cas de `signed` et `unsigned`
- Appel de fonction, retour de fonction et affectation : conversion dans la variable réceptrice (même si tronquage)

## Hygiène de programmation

- Convertir explicitement, ou utiliser les mêmes types

# Conversions numériques implicites

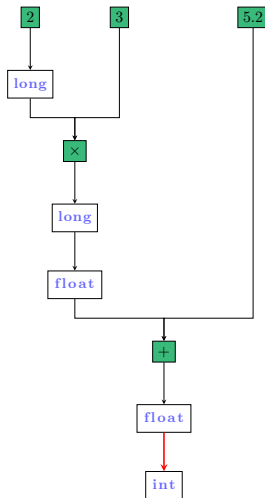
- La plupart opérateurs ne sont définis que pour des opérandes de même type (+ entier et + flottant)
- Si les opérandes ne sont pas de même type : conversion, promotion ou tronquage
- Opérateurs arithmétiques et logiques
  - 1 Si l'un des arguments est flottant, l'autre est converti en flottant
    - `long double > double > float`
  - 2 Sinon, l'argument de codage plus petit est converti
    - Même en cas de `signed` et `unsigned`
- Appel de fonction, retour de fonction et affectation : conversion dans la variable réceptrice (même si tronquage)

## Hygiène de programmation

- Convertir explicitement, ou utiliser les mêmes types

# Exemple

```
int p = 2 * 3L + 5.2F;
```



# Autres conversions

- Opérateurs arithmétiques non définis pour `short`, `bool` et `char`
- Implicitement convertis en `int` si présents
  - Caractères : dépend de l'encodage, de la machine, etc.
  - Booléens : `false` : 0, `true` : autre.
- Conversions non signées autorisées
  - Parfois incohérent : que vaut `-5` en non signé ?
- On peut effectuer des conversions explicites en faisant un cast
  - En C, même syntaxe qu'en Java
  - En C++, mécanisme de conversion dédié (cf. Ch. 8)



# Autres conversions

- Opérateurs arithmétiques non définis pour `short`, `bool` et `char`
- Implicitement convertis en `int` si présents
  - Caractères : dépend de l'encodage, de la machine, etc.
  - Booléens : `false` : 0, `true` : autre.
- Conversions non signées autorisées
  - Parfois incohérent : que vaut `-5` en non signé ?
- On peut effectuer des conversions explicites en faisant un cast
  - En C, même syntaxe qu'en Java
  - En C++, mécanisme de conversion dédié (cf. Ch. 8)

# Autres conversions

- Opérateurs arithmétiques non définis pour `short`, `bool` et `char`
- Implicitement convertis en `int` si présents
  - Caractères : dépend de l'encodage, de la machine, etc.
  - Booléens : `false` : 0, `true` : autre.
- Conversions non signées autorisées
  - Parfois incohérent : que vaut `-5` en non signé ?
- On peut effectuer des conversions explicites en faisant un cast
  - En C, même syntaxe qu'en Java
  - En C++, mécanisme de conversion dédié (cf. Ch. 8)

# Autres conversions

- Opérateurs arithmétiques non définis pour `short`, `bool` et `char`
- Implicitement convertis en `int` si présents
  - Caractères : dépend de l'encodage, de la machine, etc.
  - Booléens : `false` : 0, `true` : autre.
- Conversions non signées autorisées
  - Parfois incohérent : que vaut `-5` en non signé ?
- On peut effectuer des conversions explicites en faisant un cast
  - En C, même syntaxe qu'en Java
  - En C++, mécanisme de conversion dédié (cf. Ch. 8)

# Autres conversions

- Opérateurs arithmétiques non définis pour `short`, `bool` et `char`
- Implicitement convertis en `int` si présents
  - Caractères : dépend de l'encodage, de la machine, etc.
  - Booléens : `false` : 0, `true` : autre.
- Conversions non signées autorisées
  - Parfois incohérent : que vaut `-5` en non signé ?
- On peut effectuer des conversions explicites en faisant un cast
  - En C, même syntaxe qu'en Java
  - En C++, mécanisme de conversion dédié (cf. Ch. 8)

# Autres conversions

- Opérateurs arithmétiques non définis pour `short`, `bool` et `char`
- Implicitement convertis en `int` si présents
  - Caractères : dépend de l'encodage, de la machine, etc.
  - Booléens : `false` : 0, `true` : autre.
- Conversions non signées autorisées
  - Parfois incohérent : que vaut `-5` en non signé ?
- On peut effectuer des conversions explicites en faisant un cast
  - En C, même syntaxe qu'en Java
  - En C++, mécanisme de conversion dédié (cf. Ch. 8)

# Autres conversions

- Opérateurs arithmétiques non définis pour `short`, `bool` et `char`
- Implicitement convertis en `int` si présents
  - Caractères : dépend de l'encodage, de la machine, etc.
  - Booléens : `false` : 0, `true` : autre.
- Conversions non signées autorisées
  - Parfois incohérent : que vaut `-5` en non signé ?
- On peut effectuer des conversions explicites en faisant un cast
  - En C, même syntaxe qu'en Java
  - En C++, mécanisme de conversion dédié (cf. Ch. 8)

# Autres conversions

- Opérateurs arithmétiques non définis pour `short`, `bool` et `char`
- Implicitement convertis en `int` si présents
  - Caractères : dépend de l'encodage, de la machine, etc.
  - Booléens : `false` : 0, `true` : autre.
- Conversions non signées autorisées
  - Parfois incohérent : que vaut `-5` en non signé ?
- On peut effectuer des conversions explicites en faisant un cast
  - En C, même syntaxe qu'en Java
  - En C++, mécanisme de conversion dédié (cf. Ch. 8)

# Autres conversions

- Opérateurs arithmétiques non définis pour `short`, `bool` et `char`
- Implicitement convertis en `int` si présents
  - Caractères : dépend de l'encodage, de la machine, etc.
  - Booléens : `false` : 0, `true` : autre.
- Conversions non signées autorisées
  - Parfois incohérent : que vaut `-5` en non signé ?
- On peut effectuer des conversions explicites en faisant un cast
  - En C, même syntaxe qu'en Java
  - En C++, mécanisme de conversion dédié (cf. Ch. 8)



# Exemple

- Fichier `conv.c`
- Même principe en C++

```
1  int main()
2  {
3      int i1 = 3;
4      int i2 = 2.5;
5      double d1 = i1;
6      double d2 = 2.5f;
7
8      unsigned u1 = i1;
9      unsigned u2 = -1;
10
11     printf("ints: %d %d\n", i1, i2);
12     printf("floating point: %.2f %.2f\n", d1, d2);
13     printf("unsigned: %u %u\n", u1, u2);
14
15     printf("Sizeof_int|float: %lu %lu\n", sizeof(int), sizeof(double));
16
17     printf("Sizeof_2*_3L+_5.0: %lu\n", sizeof(2 * 3L + 5.0));
18
19     printf("%d\n", (0 == true));
20     printf("%d\n", (1 == true));
21     printf("%d\n", (2 == true));
22 }
```

# Les alias de type

- Nom référant un type défini précédemment
- Utile pour abréger certains noms

## Syntaxe en C

```
■ typedef Type Alias;
```

- Pratique pour abréger des noms de types longs

```
■ struct A -> A
```

# Les alias de type

- Nom référant un type défini précédemment
- Utile pour abrégé certains noms

## Syntaxe en C

```
■ typedef Type Alias;
```

- Pratique pour abrégé des noms de types longs

```
■ struct A -> A
```

# Les alias de type

- Nom référant un type défini précédemment
- Utile pour abréger certains noms

## Syntaxe en C

```
■ typedef Type Alias;
```

- Pratique pour abréger des noms de types longs

```
■ struct A -> A
```

# Les alias de type

- Nom référant un type défini précédemment
- Utile pour abréger certains noms

## Syntaxe en C

- `typedef` Type Alias;

- Pratique pour abréger des noms de types longs

- `struct A -> A`

# Les alias de type

- Nom référant un type défini précédemment
- Utile pour abréger certains noms

## Syntaxe en C

- `typedef` Type Alias;

- Pratique pour abréger des noms de types longs

- `struct A -> A`

# Les alias de type

- Nom référant un type défini précédemment
- Utile pour abréger certains noms

## Syntaxe en C

- `typedef` Type Alias;

- Pratique pour abréger des noms de types longs
  - `struct A -> A`

# Exemple

```
1 struct point
2 {
3     double x, y;
4 };
5
6 typedef struct point point;
7
8 point p;
```

- Sans l'alias, on aurait dû écrire `struct point p;`



# Initialisation explicite

## ■ Plusieurs types d'initialisation

- Initialisation scalaire
- Initialisation de tableaux (cf. Ch. 2)
- Initialisation de structures (cf. Ch. 4)

## ■ Toutes les initialisation se font soit

- avec =
- avec = { ... } (tableaux et structures)

## ■ Des conversions, promotions et tronquages implicites sont possibles

## ■ En l'absence d'initialisation explicite, les valeurs par défaut dépendent du type d'allocation

- Cf. Ch. 5

# Initialisation explicite

## ■ Plusieurs types d'initialisation

### ■ Initialisation scalaire

■ Initialisation de tableaux (cf. Ch. 2)

■ Initialisation de structures (cf. Ch. 4)

## ■ Toutes les initialisation se font soit

■ avec =

■ avec = { . . . } (tableaux et structures)

## ■ Des conversions, promotions et tronquages implicites sont possibles

## ■ En l'absence d'initialisation explicite, les valeurs par défaut dépendent du type d'allocation

■ Cf. Ch. 5

# Initialisation explicite

## ■ Plusieurs types d'initialisation

- Initialisation scalaire
- Initialisation de tableaux (cf. Ch. 2)
- Initialisation de structures (cf. Ch. 4)

## ■ Toutes les initialisation se font soit

- avec =
- avec = { . . . } (tableaux et structures)

## ■ Des conversions, promotions et tronquages implicites sont possibles

## ■ En l'absence d'initialisation explicite, les valeurs par défaut dépendent du type d'allocation

- Cf. Ch. 5

# Initialisation explicite

- Plusieurs types d'initialisation
  - Initialisation scalaire
  - Initialisation de tableaux (cf. Ch. 2)
  - Initialisation de structures (cf. Ch. 4)
- Toutes les initialisation se font soit
  - avec =
  - avec = { . . . } (tableaux et structures)
- Des conversions, promotions et tronquages implicites sont possibles
- En l'absence d'initialisation explicite, les valeurs par défaut dépendent du type d'allocation
  - Cf. Ch. 5

# Initialisation explicite

- Plusieurs types d'initialisation
  - Initialisation scalaire
  - Initialisation de tableaux (cf. Ch. 2)
  - Initialisation de structures (cf. Ch. 4)
- Toutes les initialisation se font soit
  - avec =
  - avec = { . . . } (tableaux et structures)
- Des conversions, promotions et tronquages implicites sont possibles
- En l'absence d'initialisation explicite, les valeurs par défaut dépendent du type d'allocation
  - Cf. Ch. 5

# Initialisation explicite

- Plusieurs types d'initialisation
  - Initialisation scalaire
  - Initialisation de tableaux (cf. Ch. 2)
  - Initialisation de structures (cf. Ch. 4)
- Toutes les initialisation se font soit
  - avec =
  - avec = { . . . } (tableaux et structures)
- Des conversions, promotions et tronquages implicites sont possibles
- En l'absence d'initialisation explicite, les valeurs par défaut dépendent du type d'allocation
  - Cf. Ch. 5

# Initialisation explicite

- Plusieurs types d'initialisation
  - Initialisation scalaire
  - Initialisation de tableaux (cf. Ch. 2)
  - Initialisation de structures (cf. Ch. 4)
- Toutes les initialisation se font soit
  - avec =
  - avec = { . . . } (tableaux et structures)
- Des conversions, promotions et tronquages implicites sont possibles
- En l'absence d'initialisation explicite, les valeurs par défaut dépendent du type d'allocation
  - Cf. Ch. 5

# Initialisation explicite

- Plusieurs types d'initialisation
  - Initialisation scalaire
  - Initialisation de tableaux (cf. Ch. 2)
  - Initialisation de structures (cf. Ch. 4)
- Toutes les initialisation se font soit
  - avec =
  - avec = { . . . } (tableaux et structures)
- Des conversions, promotions et tronquages implicites sont possibles
- En l'absence d'initialisation explicite, les valeurs par défaut dépendent du type d'allocation
  - Cf. Ch. 5



# Initialisation explicite

- Plusieurs types d'initialisation
  - Initialisation scalaire
  - Initialisation de tableaux (cf. Ch. 2)
  - Initialisation de structures (cf. Ch. 4)
- Toutes les initialisation se font soit
  - avec =
  - avec = { . . . } (tableaux et structures)
- Des conversions, promotions et tronquages implicites sont possibles
- En l'absence d'initialisation explicite, les valeurs par défaut dépendent du type d'allocation
  - Cf. Ch. 5

# Initialisation explicite

- Plusieurs types d'initialisation
  - Initialisation scalaire
  - Initialisation de tableaux (cf. Ch. 2)
  - Initialisation de structures (cf. Ch. 4)
- Toutes les initialisation se font soit
  - avec =
  - avec = { . . . } (tableaux et structures)
- Des conversions, promotions et tronquages implicites sont possibles
- En l'absence d'initialisation explicite, les valeurs par défaut dépendent du type d'allocation
  - Cf. Ch. 5

# Exemple

## ■ Fichier `init.c`

```
1 int main()
2 {
3     int i; //indeterminate
4     int j = 2;
5
6     //unsigned char * p; //not a great idea
7     //unsigned char * p = 2; // still up to no good
8
9     unsigned char * p; //not a great idea
10    *p = 3;
11    unsigned char * u = 2; // still up to no good
12    *u = 3;
13 }
```