Formation C++ 17 n° 01

Variables, types et notion de programmation impérative

ROSSILLOL–LARUELLE Mattéo

16 novembre 2023

Avant-propos

Avant de commencer, il est important de rappeler que ce cours est réalisé par un étudiant. Par conséquent, il n'a pas la même fiabilité qu'un cours dispensé par un réel enseignant de l'ENSIMAG.

N'utilisez pas ce cours comme un argument d'autorité !

Si un professeur semble, a posteriori, contredire des éléments apportés par ce cours, il a très probablement raison.

Ce document est vivant : je veillerai à corriger les coquilles ou erreurs plus problématiques.

Notion de programmation impérative
Introduction

Notion de programmation impérative

Définition

La programmation impérative est un paradigme de programmation, une façon d'approcher la programmation, qui considère les opérations comme des séquences d'instructions exécutées par l'ordinateur pour modifier l'état du programme.

De manière plus schématique, cela revient (presque) à lire le programme dans l'ordre de lecture usuel.

Notion de *statements*

Définition

Des *statements* sont des fragments d'un programme C++ exécutés en séquence.

En C++, tout *statement* se termine par un point-virgule (;).

Attention Le terme « *statement* » pourrait être traduit par « instruction » en français ; cependant, ce terme est ambigu car il pourrait également référer à des instructions machines : on préférera donc, dans ce cours, parler de *statement*.

Notion de *statements*

Définition

Des *statements* sont des fragments d'un programme C++ exécutés en séquence.

En C++, tout *statement* se termine par un point-virgule (;).
Attention Le terme « *statement* » pourrait être traduit par « instruction » en français ; cependant, ce terme est ambigu car il pourrait également référer à des instructions machines : on préférera donc, dans ce cours, parler de *statement*.

Commentaires

Définition

Un commentaire est un texte documentaire qui n'a aucun impact sur le programme : son contenu est ignoré par le compilateur.

En C++, la syntaxe utilisée est la suivante :

```
1 // Un commentaire sur une ligne.
2
3 /*
4  * Un commentaire sur
5  * une ou plusieurs lignes.
6  */
```

Imprimer sur la console

Pour afficher du texte sur la console, on mettra au début de tous les fichiers les lignes suivantes :

```
1 #include <iostream>
2
3 using namespace std;
```

Pour afficher « txt », on utilisera :

```
1 cout << txt;
```

Si on veut rajouter un retour à la ligne, on préférera :

```
1 cout << txt << '\n';
```

Notion de programmation impérative

La fonction `main`

La fonction `main`

En C++, comme en C, le point de départ de l'exécution du programme est la première instruction de la fonction `main`.

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main(int argc, char* argv[])
6  {
7      cout << "Hello world!" << '\n';
8
9      return 0; // cette ligne est optionnelle
10 }
```

Attention Particularité de cette fonction (et uniquement celle-ci), on peut omettre le `return 0` car c'est la valeur de retour par défaut.

La fonction `main`

```
1 int main(int argc, char* argv[])
```

Cette fonction a pour type de retour un `int` (un entier). Celui-ci notifie si le programme s'est terminé normalement :

- un retour nul signifie qu'il n'y a pas eu d'erreur,
- à l'inverse d'une valeur non nul qui informe, malgré cela, l'erreur rencontrée par sa valeur.

Elle prend deux arguments : un entier et un tableau de chaînes de caractères. Par convention, on a :

- `argc` (*argument count*) qui représente le nombre d'argument passé en ligne de commande,
- `argv` (*argument vector*) qui représente les arguments.

Attention Au moins un argument est passé au programme : il s'agit de son nom (son chemin).

La fonction `main`

```
1 int main(int argc, char* argv[])
```

Cette fonction a pour type de retour un `int` (un entier). Celui-ci notifie si le programme s'est terminé normalement :

- un retour nul signifie qu'il n'y a pas eu d'erreur,
- à l'inverse d'une valeur non nul qui informe, malgré cela, l'erreur rencontrée par sa valeur.

Elle prend deux arguments : un entier et un tableau de chaînes de caractères. Par convention, on a :

- `argc` (*argument count*) qui représente le nombre d'argument passé en ligne de commande,
- `argv` (*argument vector*) qui représente les arguments.

Attention Au moins un argument est passé au programme : il s'agit de son nom (son chemin).

La fonction `main`

```
1 int main(int argc, char* argv[])
```

Cette fonction a pour type de retour un `int` (un entier). Celui-ci notifie si le programme s'est terminé normalement :

- un retour nul signifie qu'il n'y a pas eu d'erreur,
- à l'inverse d'une valeur non nul qui informe, malgré cela, l'erreur rencontrée par sa valeur.

Elle prend deux arguments : un entier et un tableau de chaînes de caractères. Par convention, on a :

- `argc` (*argument count*) qui représente le nombre d'argument passé en ligne de commande,
- `argv` (*argument vector*) qui représente les arguments.

Attention Au moins un argument est passé au programme : il s'agit de son nom (son chemin).

La fonction `main`

```
1 int main(int argc, char* argv[])
```

Cette fonction a pour type de retour un `int` (un entier). Celui-ci notifie si le programme s'est terminé normalement :

- un retour nul signifie qu'il n'y a pas eu d'erreur,
- à l'inverse d'une valeur non nul qui informe, malgré cela, l'erreur rencontrée par sa valeur.

Elle prend deux arguments : un entier et un tableau de chaînes de caractères. Par convention, on a :

- `argc` (*argument count*) qui représente le nombre d'argument passé en ligne de commande,
- `argv` (*argument vector*) qui représente les arguments.

Attention Au moins un argument est passé au programme : il s'agit de son nom (son chemin).

La fonction `main`

```
1 int main(int argc, char* argv[])
```

Cette fonction a pour type de retour un `int` (un entier). Celui-ci notifie si le programme s'est terminé normalement :

- un retour nul signifie qu'il n'y a pas eu d'erreur,
- à l'inverse d'une valeur non nul qui informe, malgré cela, l'erreur rencontrée par sa valeur.

Elle prend deux arguments : un entier et un tableau de chaînes de caractères. Par convention, on a :

- `argc` (*argument count*) qui représente le nombre d'argument passé en ligne de commande,
- `argv` (*argument vector*) qui représente les arguments.

Attention Au moins un argument est passé au programme : il s'agit de son nom (son chemin).

La fonction `main`

```
1 int main(int argc, char* argv[])
```

Cette fonction a pour type de retour un `int` (un entier). Celui-ci notifie si le programme s'est terminé normalement :

- un retour nul signifie qu'il n'y a pas eu d'erreur,
- à l'inverse d'une valeur non nul qui informe, malgré cela, l'erreur rencontrée par sa valeur.

Elle prend deux arguments : un entier et un tableau de chaînes de caractères. Par convention, on a :

- `argc` (*argument count*) qui représente le nombre d'argument passé en ligne de commande,
- `argv` (*argument vector*) qui représente les arguments.

Attention Au moins un argument est passé au programme : il s'agit de son nom (son chemin).

La fonction `main`

```
1 int main(int argc, char* argv[])
```

Cette fonction a pour type de retour un `int` (un entier). Celui-ci notifie si le programme s'est terminé normalement :

- un retour nul signifie qu'il n'y a pas eu d'erreur,
- à l'inverse d'une valeur non nul qui informe, malgré cela, l'erreur rencontrée par sa valeur.

Elle prend deux arguments : un entier et un tableau de chaînes de caractères. Par convention, on a :

- `argc` (*argument count*) qui représente le nombre d'argument passé en ligne de commande,
- `argv` (*argument vector*) qui représente les arguments.

Attention Au moins un argument est passé au programme : il s'agit de son nom (son chemin).

Différentes formes possibles

`argv` est ici un tableau de chaîne de caractères.

```
1 #include <iostream>
2
3 using namespace std;
4
5 int main(int argc, char* argv[])
6 {
7     cout << "Hello world!" << '\n';
8 }
9
```

`argv` est ici un pointeur vers le premier élément d'un tableau de chaînes de caractères.

```
1 #include <iostream>
2
3 using namespace std;
4
5 int main(int argc, char** argv)
6 {
7     cout << "Hello world!" << '\n';
8 }
9
```

Quand on a besoin ni de `argc` ni de `argv`, on peut les omettre.

```
1 #include <iostream>
2
3 using namespace std;
4
5 int main()
6 {
7     cout << "Hello world!" << '\n';
8 }
9
```

Notion de programmation impérative

La fonction `main`

Quelques exemples

Pour bien comprendre la notion de point d'entrée

Exemple

```
1  #include <iostream>
2
3  using namespace std;
4
5  void foo()
6  {
7      cout << "foo" << '\n';
8  }
9
10 void hello()
11 {
12     cout << "Hello world!" << '\n';
13 }
14
15 int main(int argc, char* argv[])
16 {
17     hello();
18 }
19
```

Dans l'exemple ci-dessus, que sera-t-il affiché sur la console ?

Pour bien comprendre la notion de point d'entrée

Exemple

```
1  #include <iostream>
2
3  using namespace std;
4
5  void foo()
6  {
7      cout << "foo" << '\n';
8  }
9
10 void hello()
11 {
12     cout << "Hello world!" << '\n';
13 }
14
15 int main(int argc, char* argv[])
16 {
17     hello();
18 }
19
```

Dans l'exemple ci-dessus, que sera-t-il affiché sur la console ?

Il sera affiché « Hello world ! ».

Pour bien comprendre la notion de point d'entrée

Exemple

```
1  #include <iostream>
2
3  using namespace std;
4
5  void hello()
6  {
7      cout << "Hello world!" << '\n';
8  }
9
10 hello();
11
12 int main(int argc, char* argv[])
13 {
14     hello();
15 }
16
```

Dans l'exemple ci-dessus, que va-t-il se passer et pourquoi ?

Pour bien comprendre la notion de point d'entrée

Exemple

```
1  #include <iostream>
2
3  using namespace std;
4
5  void hello()
6  {
7      cout << "Hello world!" << '\n';
8  }
9
10 hello();
11
12 int main(int argc, char* argv[])
13 {
14     hello();
15 }
16
```

Dans l'exemple ci-dessus, que va-t-il se passer et pourquoi ?

Le programme ne se compilera pas car on appelle une fonction dans la portée globale.

Définition de variables
Introduction

Un langage à typage statique et faible

Définition

Un langage statiquement typé est un langage dont le type des variables est déterminé à la compilation. Il s'oppose à un langage dynamiquement typé.

Définition

Un langage faiblement typé est un langage dont chaque variable, bien qu'elle possède un type, peut en changer en s'appuyant, par exemple, sur des règles de conversion.

Le C++ est un langage à typage statique et faible ; à l'inverse, Python est un langage à typage dynamique et fort.

L'intérêt d'un typage statique

Exemple

Considérons le programme Python ci-dessous :

```python
#! /usr/bin/env python3


def my_sum(a, b):
    """
    Somme deux entiers.
    """

    return a + b

def main():
    my_sum(10, 10)
    my_sum([1], [3]) # fonctionne quand même !
    my_sum(10, "toto") # on a une erreur ici !

if __name__ == "__main__":
    main()
```

L'intérêt d'un typage statique

Exemple

On remarque deux choses :

1. à **la ligne 13**, on a aucune erreur bien que l'on ne veuille que des entiers ;
2. à **la ligne 14**, on a une erreur qui ne se manifeste que lorsque la ligne est atteinte à l'exécution.

En C++, les variables étant statiquement typées, les deux problématiques ci-dessus ne se posent pas.

L'intérêt d'un typage statique

Exemple

On remarque deux choses :

1. à **la ligne 13**, on a aucune erreur bien que l'on ne veuille que des entiers ;
2. à **la ligne 14**, on a une erreur qui ne se manifeste que lorsque la ligne est atteinte à l'exécution.

En C++, les variables étant statiquement typées, les deux problématiques ci-dessus ne se posent pas.

L'intérêt d'un typage statique

Exemple

On remarque deux choses :

1. à **la ligne 13**, on a aucune erreur bien que l'on ne veuille que des entiers ;
2. à **la ligne 14**, on a une erreur qui ne se manifeste que lorsque la ligne est atteinte à l'exécution.

En C++, les variables étant statiquement typées, les deux problématiques ci-dessus ne se posent pas.

L'intérêt d'un typage statique

Exemple

On remarque deux choses :

1. à **la ligne 13**, on a aucune erreur bien que l'on ne veuille que des entiers ;
2. à **la ligne 14**, on a une erreur qui ne se manifeste que lorsque la ligne est atteinte à l'exécution.

En C++, les variables étant statiquement typées, les deux problématiques ci-dessus ne se posent pas.

Notion d'alias

En C++, comme en C, on peut nommer un `type` par un `autre type`.
Il existe deux syntaxes équivalentes :

```
typedef <type> <autre nom>;
```

Figure 1 – définition d'un alias (syntaxe C)

```
using <autre nom> = <type>;
```

Figure 2 – définition d'un alias (syntaxe C++)

Notion d'alias

En C++, comme en C, on peut nommer un `type` par un `autre type`.
Il existe deux syntaxes équivalentes :

```
typedef <type> <autre nom>;
```

Figure 1 – définition d'un alias (syntaxe C)

```
using <autre nom> = <type>;
```

Figure 2 – définition d'un alias (syntaxe C++)

Notion d'alias

En C++, comme en C, on peut nommer un `type` par un `autre type`.
Il existe deux syntaxes équivalentes :

```
typedef <type> <autre nom>;
```

Figure 1 – définition d'un alias (syntaxe C)

```
using <autre nom> = <type>;
```

Figure 2 – définition d'un alias (syntaxe C++)

Notion d'alias

En C++, comme en C, on peut nommer un `type` par un `autre type`.
Il existe deux syntaxes équivalentes :

```
typedef <type> <autre nom>;
```

Figure 1 – définition d'un alias (syntaxe C)

```
using <autre nom> = <type>;
```

Figure 2 – définition d'un alias (syntaxe C++)

Un exemple

Exemple

```
1  #include <iostream>
2
3  using namespace std;
4
5  using mon_super_booleen = int;
6  typedef int mon_super_type;
7
8  int main()
9  {
10     mon_super_entier a = 5;
11     mon_super_type b   = 10;
12
13     cout << a << '\n';
14     cout << b << '\n';
15 }
```

Dans l'exemple ci-dessus que va-t-il s'imprimer sur la console ?

Un exemple

Exemple

```
1  #include <iostream>
2
3  using namespace std;
4
5  using mon_super_booleen = int;
6  typedef int mon_super_type;
7
8  int main()
9  {
10     mon_super_entier a = 5;
11     mon_super_type b   = 10;
12
13     cout << a << '\n';
14     cout << b << '\n';
15 }
```

Dans l'exemple ci-dessus que va-t-il s'imprimer sur la console ?

Il y aura « 5 », puis « 10 ».

Définition de variables
Types fondamentaux

Types fondamentaux

Le C++ est un langage fortement typé : toutes les variables ont un type et qui ne peut pas changer au cours de l'exécution du programme.
Pour l'instant, on se limitera au types fondamentaux (ou fundamental types) à l'exception de `std::nullptr_t` que l'on verra plus tard quand on parlera de pointeur.
Il s'agit de :

- `void` (presque équivalent au `None` de Python),
- `bool` (booléen),
- `char`, `signed char` et `unsigned char` (types caractères ordinaires),
- `char16_t`, `char32_t` et `wchar_t` (*wide character types*),
- `float`, `double` et `long double` (types virgules flottantes),
- `signed char`, `short`, `int`, `long` et `long long` (types entiers signés),
- `unsigned char`, `unsigned short`, `unsigned int`, `unsigned long` et `unsigned long long` (types entiers non signés).

Types fondamentaux

Le C++ est un langage fortement typé : toutes les variables ont un type et qui ne peut pas changer au cours de l'exécution du programme.
Pour l'instant, on se limitera au types fondamentaux (ou fundamental types) à l'exception de `std::nullptr_t` que l'on verra plus tard quand on parlera de pointeur.
Il s'agit de :

- `void` (presque équivalent au `None` de Python),
- `bool` (booléen),
- `char`, `signed char` et `unsigned char` (types caractères ordinaires),
- `char16_t`, `char32_t` et `wchar_t` (*wide character types*),
- `float`, `double` et `long double` (types virgules flottantes),
- `signed char`, `short`, `int`, `long` et `long long` (types entiers signés),
- `unsigned char`, `unsigned short`, `unsigned int`, `unsigned long` et `unsigned long long` (types entiers non signés).

Types fondamentaux

Le C++ est un langage fortement typé : toutes les variables ont un type et qui ne peut pas changer au cours de l'exécution du programme.
Pour l'instant, on se limitera au types fondamentaux (ou fundamental types) à l'exception de `std::nullptr_t` que l'on verra plus tard quand on parlera de pointeur.
Il s'agit de :

- `void` (presque équivalent au `None` de Python),
- `bool` (booléen),
- `char`, `signed char` et `unsigned char` (types caractères ordinaires),
- `char16_t`, `char32_t` et `wchar_t` (*wide character types*),
- `float`, `double` et `long double` (types virgules flottantes),
- `signed char`, `short`, `int`, `long` et `long long` (types entiers signés),
- `unsigned char`, `unsigned short`, `unsigned int`, `unsigned long` et `unsigned long long` (types entiers non signés).

Types fondamentaux

Le C++ est un langage fortement typé : toutes les variables ont un type et qui ne peut pas changer au cours de l'exécution du programme.
Pour l'instant, on se limitera au types fondamentaux (ou fundamental types) à l'exception de `std::nullptr_t` que l'on verra plus tard quand on parlera de pointeur.
Il s'agit de :

- `void` (presque équivalent au `None` de Python),
- `bool` (booléen),
- `char`, `signed char` et `unsigned char` (types caractères ordinaires),
- `char16_t`, `char32_t` et `wchar_t` (*wide character types*),
- `float`, `double` et `long double` (types virgules flottantes),
- `signed char`, `short`, `int`, `long` et `long long` (types entiers signés),
- `unsigned char`, `unsigned short`, `unsigned int`, `unsigned long` et `unsigned long long` (types entiers non signés).

Types fondamentaux

Le C++ est un langage fortement typé : toutes les variables ont un type et qui ne peut pas changer au cours de l'exécution du programme.
Pour l'instant, on se limitera au types fondamentaux (ou fundamental types) à l'exception de `std::nullptr_t` que l'on verra plus tard quand on parlera de pointeur.
Il s'agit de :

- `void` (presque équivalent au `None` de Python),
- `bool` (booléen),
- `char`, `signed char` et `unsigned char` (types caractères ordinaires),
- `char16_t`, `char32_t` et `wchar_t` (*wide character types*),
- `float`, `double` et `long double` (types virgules flottantes),
- `signed char`, `short`, `int`, `long` et `long long` (types entiers signés),
- `unsigned char`, `unsigned short`, `unsigned int`, `unsigned long` et `unsigned long long` (types entiers non signés).

Types fondamentaux

Le C++ est un langage fortement typé : toutes les variables ont un type et qui ne peut pas changer au cours de l'exécution du programme.
Pour l'instant, on se limitera au types fondamentaux (ou fundamental types) à l'exception de `std::nullptr_t` que l'on verra plus tard quand on parlera de pointeur.
Il s'agit de :

- `void` (presque équivalent au `None` de Python),
- `bool` (booléen),
- `char`, `signed char` et `unsigned char` (types caractères ordinaires),
- `char16_t`, `char32_t` et `wchar_t` (*wide character types*),
- `float`, `double` et `long double` (types virgules flottantes),
- `signed char`, `short`, `int`, `long` et `long long` (types entiers signés),
- `unsigned char`, `unsigned short`, `unsigned int`, `unsigned long` et `unsigned long long` (types entiers non signés).

Types fondamentaux

Le C++ est un langage fortement typé : toutes les variables ont un type et qui ne peut pas changer au cours de l'exécution du programme.
Pour l'instant, on se limitera au types fondamentaux (ou fundamental types) à l'exception de `std::nullptr_t` que l'on verra plus tard quand on parlera de pointeur.
Il s'agit de :

- `void` (presque équivalent au `None` de Python),
- `bool` (booléen),
- `char`, `signed char` et `unsigned char` (types caractères ordinaires),
- `char16_t`, `char32_t` et `wchar_t` (*wide character types*),
- `float`, `double` et `long double` (types virgules flottantes),
- `signed char`, `short`, `int`, `long` et `long long` (types entiers signés),
- `unsigned char`, `unsigned short`, `unsigned int`, `unsigned long` et `unsigned long long` (types entiers non signés).

Types fondamentaux

Le C++ est un langage fortement typé : toutes les variables ont un type et qui ne peut pas changer au cours de l'exécution du programme.
Pour l'instant, on se limitera au types fondamentaux (ou fundamental types) à l'exception de `std::nullptr_t` que l'on verra plus tard quand on parlera de pointeur.
Il s'agit de :

- `void` (presque équivalent au `None` de Python),
- `bool` (booléen),
- `char`, `signed char` et `unsigned char` (types caractères ordinaires),
- `char16_t`, `char32_t` et `wchar_t` (*wide character types*),
- `float`, `double` et `long double` (types virgules flottantes),
- `signed char`, `short`, `int`, `long` et `long long` (types entiers signés),
- `unsigned char`, `unsigned short`, `unsigned int`, `unsigned long` et `unsigned long long` (types entiers non signés).

Définition de variables

Types fondamentaux

Les types non entiers

Le mot clef `void`

Définition

Un type incomplet est un type qui manque des informations nécessaires pour connaître sa taille en mémoire.

Une variable de type incomplet est très limitée dans son utilisation.
`void` est un type incomplet qui ne peut pas être complété. Son rôle sera vu, plus en détails, quand on parlera de pointeurs, de fonctions et de `template`.

Exemple

De cette façon, la ligne ci-dessous ne compilera pas :

```
1 void a;
```

Le mot clef `void`

Définition

Un type incomplet est un type qui manque des informations nécessaires pour connaître sa taille en mémoire.

Une variable de type incomplet est très limitée dans son utilisation.
`void` est un type incomplet qui ne peut pas être complété. Son rôle sera vu, plus en détails, quand on parlera de pointeurs, de fonctions et de `template`.

Exemple

De cette façon, la ligne ci-dessous ne compilera pas :

```
1 void a;
```

Les booléens

Un booléen a deux valeurs possibles :

- `false`,
- `true`.

N'importe quel intégral (*integral*) (un caractère ou un entier) ou flottant peut être converti implicitement en booléen. La règle est la suivante :

- si sa valeur est nulle, alors la variable en question vaut `false`,
- sinon, il vaut `true`.

Ceci constitue un des nombreux exemples montrant que le C++ est un langage faiblement typé.

Exemple

```
1 bool a = 255; // a est vrai
2 bool b = 0.; // b est faux
3 bool c = '0'; // b est vrai
4
```

Les booléens

Un booléen a deux valeurs possibles :

- `false`,
- `true`.

N'importe quel intégral (*integral*) (un caractère ou un entier) ou flottant peut être converti implicitement en booléen. La règle est la suivante :

- si sa valeur est nulle, alors la variable en question vaut `false`,
- sinon, il vaut `true`.

Ceci constitue un des nombreux exemples montrant que le C++ est un langage faiblement typé.

Exemple

```
1 bool a = 255; // a est vrai
2 bool b = 0.; // b est faux
3 bool c = '0'; // b est vrai
4
```

Les booléens

Un booléen a deux valeurs possibles :

- `false`,
- `true`.

N'importe quel intégral (*integral*) (un caractère ou un entier) ou flottant peut être converti implicitement en booléen. La règle est la suivante :

- si sa valeur est nulle, alors la variable en question vaut `false`,
- sinon, il vaut `true`.

Ceci constitue un des nombreux exemples montrant que le C++ est un langage faiblement typé.

Exemple

```
1 bool a = 255; // a est vrai
2 bool b = 0.; // b est faux
3 bool c = '0'; // b est vrai
4
```

Les booléens

Un booléen a deux valeurs possibles :

- `false`,
- `true`.

N'importe quel intégral (*integral*) (un caractère ou un entier) ou flottant peut être converti implicitement en booléen. La règle est la suivante :

- si sa valeur est nulle, alors la variable en question vaut `false`,
- sinon, il vaut `true`.

Ceci constitue un des nombreux exemples montrant que le C++ est un langage faiblement typé.

Exemple

```
1 bool a = 255; // a est vrai
2 bool b = 0.; // b est faux
3 bool c = '0'; // b est vrai
4
```

Les booléens

Un booléen a deux valeurs possibles :

- `false`,
- `true`.

N'importe quel intégral (*integral*) (un caractère ou un entier) ou flottant peut être converti implicitement en booléen. La règle est la suivante :

- si sa valeur est nulle, alors la variable en question vaut `false`,
- sinon, il vaut `true`.

Ceci constitue un des nombreux exemples montrant que le C++ est un langage faiblement typé.

Exemple

```
1 bool a = 255; // a est vrai
2 bool b = 0.; // b est faux
3 bool c = '0'; // b est vrai
4
```

Les booléens

Un booléen a deux valeurs possibles :

- `false`,
- `true`.

N'importe quel intégral (*integral*) (un caractère ou un entier) ou flottant peut être converti implicitement en booléen. La règle est la suivante :

- si sa valeur est nulle, alors la variable en question vaut `false`,
- sinon, il vaut `true`.

Ceci constitue un des nombreux exemples montrant que le C++ est un langage faiblement typé.

Exemple

```
1 bool a = 255; // a est vrai
2 bool b = 0.; // b est faux
3 bool c = '0'; // b est vrai
4
```

Les booléens

Un booléen a deux valeurs possibles :

- `false`,
- `true`.

N'importe quel intégral (*integral*) (un caractère ou un entier) ou flottant peut être converti implicitement en booléen. La règle est la suivante :

- si sa valeur est nulle, alors la variable en question vaut `false`,
- sinon, il vaut `true`.

Ceci constitue un des nombreux exemples montrant que le C++ est un langage faiblement typé.

Exemple

```
1 bool a = 255; // a est vrai
2 bool b = 0.; // b est faux
3 bool c = '0'; // b est vrai
4
```

Les booléens

Un booléen a deux valeurs possibles :

- `false`,
- `true`.

N'importe quel intégral (*integral*) (un caractère ou un entier) ou flottant peut être converti implicitement en booléen. La règle est la suivante :

- si sa valeur est nulle, alors la variable en question vaut `false`,
- sinon, il vaut `true`.

Ceci constitue un des nombreux exemples montrant que le C++ est un langage faiblement typé.

Exemple

```
1 bool a = 255; // a est vrai
2 bool b = 0.; // b est faux
3 bool c = '0'; // b est vrai
4
```

Types caractères ordinaires

`char` est soit signé soit non signé mais il diffèrera toujours de `signed char` et `unsigned char`.
`char` est capable de stocker un certain nombre de caractères Unicode dont on peut trouver la liste précise sur
`https://en.cppreference.com/w/cpp/language/charset#Basic_character_set`.
Il a une taille fixe : dans presque toutes les situations, elle est d'un octet.

Wide character types

Définition

La taille des caractères dans certains encodages (par exemple, UTF-8, UTF-16 et UTF-32) est variables. De cette façon, un caractère est divisé en plusieurs unités appelées unités de code.

Les types étudiés ici représentent chacun une unité de code.

On retrouve :

- `char16_t` en UTF-16 (au moins deux octets),
- `char32_t` en UTF-32 (au moins quatre octets),
- `wchar_t` qui est sensé supporté n'importe quelle unité de code qu'importe l'encodage utilisé.

Attention En réalité, `wchar_t` est dysfonctionnel et son usage est à limiter.

Wide character types

Définition

La taille des caractères dans certains encodages (par exemple, UTF-8, UTF-16 et UTF-32) est variables. De cette façon, un caractère est divisé en plusieurs unités appelées unités de code.

Les types étudiés ici représentent chacun une unité de code.
On retrouve :

- `char16_t` en UTF-16 (au moins deux octets),
- `char32_t` en UTF-32 (au moins quatre octets),
- `wchar_t` qui est sensé supporté n'importe quelle unité de code qu'importe l'encodage utilisé.

Attention En réalité, `wchar_t` est dysfonctionnel et son usage est à limiter.

Wide character types

Définition

La taille des caractères dans certains encodages (par exemple, UTF-8, UTF-16 et UTF-32) est variables. De cette façon, un caractère est divisé en plusieurs unités appelées unités de code.

Les types étudiés ici représentent chacun une unité de code.
On retrouve :

- `char16_t` en UTF-16 (au moins deux octets),
- `char32_t` en UTF-32 (au moins quatre octets),
- `wchar_t` qui est sensé supporté n'importe quelle unité de code qu'importe l'encodage utilisé.

Attention En réalité, `wchar_t` est dysfonctionnel et son usage est à limiter.

Wide character types

Définition

La taille des caractères dans certains encodages (par exemple, UTF-8, UTF-16 et UTF-32) est variables. De cette façon, un caractère est divisé en plusieurs unités appelées unités de code.

Les types étudiés ici représentent chacun une unité de code.
On retrouve :

- `char16_t` en UTF-16 (au moins deux octets),
- `char32_t` en UTF-32 (au moins quatre octets),
- `wchar_t` qui est sensé supporté n'importe quelle unité de code qu'importe l'encodage utilisé.

Attention En réalité, `wchar_t` est dysfonctionnel et son usage est à limiter.

Wide character types

Définition

La taille des caractères dans certains encodages (par exemple, UTF-8, UTF-16 et UTF-32) est variables. De cette façon, un caractère est divisé en plusieurs unités appelées unités de code.

Les types étudiés ici représentent chacun une unité de code.
On retrouve :

- `char16_t` en UTF-16 (au moins deux octets),
- `char32_t` en UTF-32 (au moins quatre octets),
- `wchar_t` qui est sensé supporté n'importe quelle unité de code qu'importe l'encodage utilisé.

Attention En réalité, `wchar_t` est dysfonctionnel et son usage est à limiter.

Types virgules flottantes

Il existe trois types différents :

1. `float` est une virgule flottante simple précision,
2. `double` est une virgule flottante double précision,
3. `long double` est, si la machine cible la supporte, une virgule flottante quadruple précision ; sinon si cela est supporté, il s'agit d'une virgule flottante double précision étendue ; sinon si celui-ci existe, un autre format avec une meilleure précision ; sinon, c'est une virgule flottante double précision.

Attention En d'autres termes, les types ci-dessus sont listés par ordre de précision croissante (pas strictement).

Types virgules flottantes

Il existe trois types différents :

1. `float` est une virgule flottante simple précision,
2. `double` est une virgule flottante double précision,
3. `long double` est, si la machine cible la supporte, une virgule flottante quadruple précision ; sinon si cela est supporté, il s'agit d'une virgule flottante double précision étendue ; sinon si celui-ci existe, un autre format avec une meilleure précision ; sinon, c'est une virgule flottante double précision.

Attention En d'autres termes, les types ci-dessus sont listés par ordre de précision croissante (pas strictement).

Types virgules flottantes

Il existe trois types différents :

1. `float` est une virgule flottante simple précision,
2. `double` est une virgule flottante double précision,
3. `long double` est, si la machine cible la supporte, une virgule flottante quadruple précision ; sinon si cela est supporté, il s'agit d'une virgule flottante double précision étendue ; sinon si celui-ci existe, un autre format avec une meilleure précision ; sinon, c'est une virgule flottante double précision.

Attention En d'autres termes, les types ci-dessus sont listés par ordre de précision croissante (pas strictement).

Types virgules flottantes

Il existe trois types différents :

1. `float` est une virgule flottante simple précision,
2. `double` est une virgule flottante double précision,
3. `long double` est, si la machine cible la supporte, une virgule flottante quadruple précision ; sinon si cela est supporté, il s'agit d'une virgule flottante double précision étendue ; sinon si celui-ci existe, un autre format avec une meilleure précision ; sinon, c'est une virgule flottante double précision.

Attention En d'autres termes, les types ci-dessus sont listés par ordre de précision croissante (pas strictement).

Définition de variables

Types fondamentaux

Les entiers

Une histoire de signe

Un entier est dit « signé » s'il est capable de supporter des valeurs négatives ; dans le cas contraire, on dit qu'il est « non signé ».

Pour définir un entier comme signé, on peut utiliser la syntaxe suivante :

```
unsigned <type entier>
```

Figure 3 – forme d'un entier non signé

Par défaut, en C++, les entiers sont signés ; on peut cependant rajouter le mot clef `signed` pour l'expliciter.

Une histoire de signe

Un entier est dit « signé » s'il est capable de supporter des valeurs négatives ; dans le cas contraire, on dit qu'il est « non signé ».
Pour définir un entier comme signé, on peut utiliser la syntaxe suivante :

```
unsigned <type entier>
```

Figure 3 – forme d'un entier non signé

Par défaut, en C++, les entiers sont signés ; on peut cependant rajouter le mot clef `signed` pour l'expliciter.

Une histoire de signe

Un entier est dit « signé » s'il est capable de supporter des valeurs négatives ; dans le cas contraire, on dit qu'il est « non signé ».
Pour définir un entier comme signé, on peut utiliser la syntaxe suivante :

```
unsigned <type entier>
```

Figure 3 – forme d'un entier non signé

Par défaut, en C++, les entiers sont signés ; on peut cependant rajouter le mot clef `signed` pour l'expliciter.

Comment représenter un entier négatif en mémoire ?

On peut proposer trois approches différentes :

1. l'utilisation d'un bit de signe,
2. l'opposé d'un entier est son complément à 1 (bit-à-bit),
3. l'opposé d'un entier sur n bits est son complément à 2^n (abusivement appelé son complément à 2).

Comment représenter un entier négatif en mémoire ?

On peut proposer trois approches différentes :

1. l'utilisation d'un bit de signe,
2. l'opposé d'un entier est son complément à 1 (bit-à-bit),
3. l'opposé d'un entier sur n bits est son complément à 2^n (abusivement appelé son complément à 2).

Comment représenter un entier négatif en mémoire ?

On peut proposer trois approches différentes :

1. l'utilisation d'un bit de signe,
   Problème

$$0001_2 + 1001_2 = 1010_2$$
$$\text{d'où } 1 + (-1) = -2$$

   On devrait donc avoir une interprétation pas très pratique des calculs arithmétiques avec les entiers signés.
2. l'opposé d'un entier est son complément à 1 (bit-à-bit),
3. l'opposé d'un entier sur n bits est son complément à 2^n (abusivement appelé son complément à 2).

Comment représenter un entier négatif en mémoire ?

On peut proposer trois approches différentes :

1. l'utilisation d'un bit de signe,
   Problème

$$0001_2 + 1001_2 = 1010_2$$
$$\text{d'où } 1 + (-1) = -2$$

   On devrait donc avoir une interprétation pas très pratique des calculs arithmétiques avec les entiers signés.
2. l'opposé d'un entier est son complément à 1 (bit-à-bit),
3. l'opposé d'un entier sur n bits est son complément à 2^n (abusivement appelé son complément à 2).

Comment représenter un entier négatif en mémoire ?

On peut proposer trois approches différentes :

1. l'utilisation d'un bit de signe,
2. l'opposé d'un entier est son complément à 1 (bit-à-bit),
   On a dorénavant bien :

$$0001_2 + 1110_2 = 1111_2$$
$$\text{d'où } 1 + (-1) = 0$$

   Problème On a deux représentation de 0 qui sont 0000_2 et 1111_2.
3. l'opposé d'un entier sur n bits est son complément à 2^n (abusivement appelé son complément à 2).

Comment représenter un entier négatif en mémoire ?

On peut proposer trois approches différentes :

1. l'utilisation d'un bit de signe,
2. l'opposé d'un entier est son complément à 1 (bit-à-bit),
   On a dorénavant bien :

$$0001_2 + 1110_2 = 1111_2$$
$$\text{d'où } 1 + (-1) = 0$$

   Problème On a deux représentation de 0 qui sont 0000_2 et 1111_2.

3. l'opposé d'un entier sur n bits est son complément à 2^n (abusivement appelé son complément à 2).

Comment représenter un entier négatif en mémoire ?

On peut proposer trois approches différentes :

1. l'utilisation d'un bit de signe,
2. l'opposé d'un entier est son complément à 1 (bit-à-bit),
   On a dorénavant bien :

$$0001_2 + 1110_2 = 1111_2$$
$$\text{d'où } 1 + (-1) = 0$$

   Problème On a deux représentation de 0 qui sont 0000_2 et 1111_2.
3. l'opposé d'un entier sur n bits est son complément à 2^n (abusivement appelé son complément à 2).

Comment représenter un entier négatif en mémoire ?

On peut proposer trois approches différentes :

1. l'utilisation d'un bit de signe,
2. l'opposé d'un entier est son complément à 1 (bit-à-bit),
3. l'opposé d'un entier sur n bits est son complément à 2^n (abusivement appelé son complément à 2).
   La représentation sur quatre bits de l'opposé de 1 est donc $2^4 - 1 = 15 = 1111_2$.
   On a donc :

$$0001_2 + 1111_2 = \not{1}0000_2 = 0000_2$$
$$\text{d'où } 1 + (-1) = 0$$

0000_2 est aussi la seule représentation de 0.

Comment représenter un entier négatif en mémoire ?

On peut proposer trois approches différentes :

1. l'utilisation d'un bit de signe,
2. l'opposé d'un entier est son complément à 1 (bit-à-bit),
3. l'opposé d'un entier sur n bits est son complément à 2^n (abusivement appelé son complément à 2).

Pour représenter un entier signé, un entier qui peut avoir des valeurs négatives, les concepteurs d'architecture ont tous choisi la troisième méthode.
Un entier signé sur n bits prend donc ses valeurs de -2^{n-1} à $2^{n-1} - 1$.
Le bit de poids fort caractérise le signe de l'entier.
On a, pour un nombre x encodé sur n bits :

$$2^n - x = \neg x + 1$$
$$\text{ou encore } (1 \ll n) - x = \neg x + 1$$

Les entiers et leur taille en mémoire

En C++, le standard est trompeur : la taille spécifiée des entiers n'est pas stricte mais minimale.

Ainsi, on a :

1. `char` au moins un octet,
2. `short` au moins deux octets,
3. `int` au moins deux octets,
4. `long` au moins trois octets,
5. `long long` au moins quatre octets.

Les entiers et leur taille en mémoire

En C++, le standard est trompeur : la taille spécifiée des entiers n'est pas stricte mais minimale.
Ainsi, on a :

1. `char` au moins un octet,
2. `short` au moins deux octets,
3. `int` au moins deux octets,
4. `long` au moins trois octets,
5. `long long` au moins quatre octets.

Les entiers et leur taille en mémoire

En C++, le standard est trompeur : la taille spécifiée des entiers n'est pas stricte mais minimale.
Ainsi, on a :

1. `char` au moins un octet,
2. `short` au moins deux octets,
3. `int` au moins deux octets,
4. `long` au moins trois octets,
5. `long long` au moins quatre octets.

Les entiers et leur taille en mémoire

En C++, le standard est trompeur : la taille spécifiée des entiers n'est pas stricte mais minimale.
Ainsi, on a :

1. `char` au moins un octet,
2. `short` au moins deux octets,
3. `int` au moins deux octets,
4. `long` au moins trois octets,
5. `long long` au moins quatre octets.

Les entiers et leur taille en mémoire

En C++, le standard est trompeur : la taille spécifiée des entiers n'est pas stricte mais minimale.
Ainsi, on a :

1. `char` au moins un octet,
2. `short` au moins deux octets,
3. `int` au moins deux octets,
4. `long` au moins trois octets,
5. `long long` au moins quatre octets.

Les entiers et leur taille en mémoire

En C++, le standard est trompeur : la taille spécifiée des entiers n'est pas stricte mais minimale.
Ainsi, on a :

1. `char` au moins un octet,
2. `short` au moins deux octets,
3. `int` au moins deux octets,
4. `long` au moins trois octets,
5. `long long` au moins quatre octets.

Les modèles de données

La taille exacte de ces types dépend en réalité du modèle de données utilisé et dépend donc de :

- la taille des mots de la machine cible,
- le système d'exploitation de cette même machine.

Les modèles de données

La taille exacte de ces types dépend en réalité du modèle de données utilisé et dépend donc de :

- la taille des mots de la machine cible,
- le système d'exploitation de cette même machine.

Les modèles de données

La taille exacte de ces types dépend en réalité du modèle de données utilisé et dépend donc de :

- la taille des mots de la machine cible,
- le système d'exploitation de cette même machine.

Des alias bien pratiques

Comme les mots clefs précédents rendent la portabilité du code très complexe, on utilise, dans les faits, des alias qui assure que la taille de l'entier soit la bonne :

Type	Taille en mémoire (en octets)	Plage de valeurs	
		Entier non signé	Entier signé
`int8_t`	1	0 à 255	-128 à 127
`int16_t`	2	0 à $65\,535$	$-32\,768$ à $32\,767$
`int32_t`	4	0 à $4{,}29 \times 10^9$	$\pm 2{,}14 \times 10^9$
`int64_t`	8	0 à $1{,}84 \times 10^{19}$	$\pm 9{,}22 \times 10^{18}$

Dernières précisions

La version non signée de `int`n`_t` est `uint`n`_t`.
`int_least`n`_t` assure que l'entier fait au moins n bits.
`int_fast`n`_t` assure que l'entier est le plus rapide et fait au moins n bits.
Pour utiliser ces alias, il faut mettre en début de fichier les lignes :

```
1 #include <cstdint>
2
3 using namespace std; // ne pas mettre plusieurs fois cette ligne
```

Définition de variables
Les littéraux

Les littéraux

Définition

Les littéraux (ou litterals) sont, en C++, des unités syntaxiques qui représente les valeurs constantes intégrées aux langage.

Exemple

Dans l'exemple ci-dessous, les unités syntaxiques à droites des signes = sont des littéraux.

```
1 int a = 5;
2 double b = 5.;
```

Les littéraux

Définition

Les littéraux (ou litterals) sont, en C++, des unités syntaxiques qui représente les valeurs constantes intégrées aux langage.

Exemple

Dans l'exemple ci-dessous, les unités syntaxiques à droites des signes = sont des littéraux.

```
1 int a = 5;
2 double b = 5.;
```

Définition de variables

Les littéraux

Les littéraux entiers

Un entier en décimal

En décimal, les chiffres autorisés sont les chiffres décimaux classiques :

<chiffres>[suffixe]

Figure 4 – entier exprimé en décimal

Exemple

```
1 int a = 5;
```

Un entier en décimal

En décimal, les chiffres autorisés sont les chiffres décimaux classiques :

<chiffres>[suffixe]

Figure 4 – entier exprimé en décimal

Exemple

```
1 int a = 5;
```

Une petite précision syntaxique

`suffixe` détermine le type de l'entier.

- aucun suffixe : il s'agit au moins d'un `int` ;
- `l` ou `L` : il s'agit au moins d'un `long int` ;
- `ll` ou `LL` : il s'agit au moins d'un `long long int`.

En base décimale, les types sont des types signés ; à l'inverse, pour les autres bases, il peut s'agir de types signés ou non signés.
Pour imposer un type non signé, on peut ajouter (potentiellement, en plus de ceux présentés ci-dessus) le suffixe `u` ou `U`.

Une petite précision syntaxique

`suffixe` détermine le type de l'entier.

- aucun suffixe : il s'agit au moins d'un `int` ;
- `l` ou `L` : il s'agit au moins d'un `long int` ;
- `ll` ou `LL` : il s'agit au moins d'un `long long int`.

En base décimale, les types sont des types signés ; à l'inverse, pour les autres bases, il peut s'agir de types signés ou non signés.
Pour imposer un type non signé, on peut ajouter (potentiellement, en plus de ceux présentés ci-dessus) le suffixe `u` ou `U`.

Une petite précision syntaxique

`suffixe` détermine le type de l'entier.

- aucun suffixe : il s'agit au moins d'un `int` ;
- `l` ou `L` : il s'agit au moins d'un `long int` ;
- `ll` ou `LL` : il s'agit au moins d'un `long long int`.

En base décimale, les types sont des types signés ; à l'inverse, pour les autres bases, il peut s'agir de types signés ou non signés.
Pour imposer un type non signé, on peut ajouter (potentiellement, en plus de ceux présentés ci-dessus) le suffixe `u` ou `U`.

Une petite précision syntaxique

`suffixe` détermine le type de l'entier.

- aucun suffixe : il s'agit au moins d'un `int` ;
- `l` ou `L` : il s'agit au moins d'un `long int` ;
- `ll` ou `LL` : il s'agit au moins d'un `long long int`.

En base décimale, les types sont des types signés ; à l'inverse, pour les autres bases, il peut s'agir de types signés ou non signés.
Pour imposer un type non signé, on peut ajouter (potentiellement, en plus de ceux présentés ci-dessus) le suffixe `u` ou `U`.

Une petite précision syntaxique

`suffixe` détermine le type de l'entier.

- aucun suffixe : il s'agit au moins d'un `int` ;
- `l` ou `L` : il s'agit au moins d'un `long int` ;
- `ll` ou `LL` : il s'agit au moins d'un `long long int`.

En base décimale, les types sont des types signés ; à l'inverse, pour les autres bases, il peut s'agir de types signés ou non signés.

Pour imposer un type non signé, on peut ajouter (potentiellement, en plus de ceux présentés ci-dessus) le suffixe `u` ou `U`.

Une petite précision syntaxique

`suffixe` détermine le type de l'entier.

- aucun suffixe : il s'agit au moins d'un `int` ;
- `l` ou `L` : il s'agit au moins d'un `long int` ;
- `ll` ou `LL` : il s'agit au moins d'un `long long int`.

En base décimale, les types sont des types signés ; à l'inverse, pour les autres bases, il peut s'agir de types signés ou non signés.
Pour imposer un type non signé, on peut ajouter (potentiellement, en plus de ceux présentés ci-dessus) le suffixe `u` ou `U`.

Un entier en octal

En octal, les chiffres autorisés sont les chiffres de 0 à 7.

0<chiffres>[suffixe]

Figure 5 – entier exprimé en octal

Attention L'octal n'est plus une base courante.

Exemple

```
1 unsigned int a = 010u; // a vaut 8 en décimal
```

Un entier en octal

En octal, les chiffres autorisés sont les chiffres de 0 à 7.

0<chiffres>[suffixe]

Figure 5 – entier exprimé en octal

Attention L'octal n'est plus une base courante.

Exemple

```
1 unsigned int a = 010u; // a vaut 8 en décimal
```

Un entier en octal

En octal, les chiffres autorisés sont les chiffres de 0 à 7.

0<chiffres>[suffixe]

Figure 5 – entier exprimé en octal

Attention L'octal n'est plus une base courante.

Exemple

```
1 unsigned int a = 010u; // a vaut 8 en décimal
```

En hexadécimal, les chiffres autorisés sont les chiffres de 0 à 9 et de a à f (la syntaxe n'est pas sensible à la casse).

0(x|X)<chiffres>[suffixe]

Figure 6 – entier exprimé en hexadécimal

Exemple

```
1 long long a = 0xaAbCdEfLL; // a vaut 179031535 en décimal
2 unsigned long long b = 0X10ull; // a vaut 16 en décimal
```

En hexadécimal, les chiffres autorisés sont les chiffres de 0 à 9 et de a à f (la syntaxe n'est pas sensible à la casse).

```
0(x|X)<chiffres>[suffixe]
```

Figure 6 – entier exprimé en hexadécimal

Exemple

```
1 long long a = 0xaAbCdEfLL; // a vaut 179031535 en décimal
2 unsigned long long b = 0X10ull; // a vaut 16 en décimal
```

Les entiers en binaire

En binaire, les chiffres autorisés sont les chiffres 0 et 1.

```
0(b|B)<chiffres>[suffixe]
```

Figure 7 – entier exprimé en binaire

Exemple

```
1 unsigned int a = 0B10u; // a vaut 2 en décimal
2 int b = 0b01; // a vaut 1 en décimal
```

Les entiers en binaire

En binaire, les chiffres autorisés sont les chiffres 0 et 1.

0(b|B)<chiffres>[suffixe]

Figure 7 – entier exprimé en binaire

Exemple

```
1 unsigned int a = 0B10u; // a vaut 2 en décimal
2 int b = 0b01; // a vaut 1 en décimal
```

Définition de variables

Les littéraux

Les littéraux virgules flottantes

Les flottants en décimal

```
<chiffres>(e|E)<exposant>[suffixe]
.<chiffres>(e|E)[exposant][suffixe]
[chiffres].<chiffres>(e|E)[exposant][suffixe]
```

Figure 8 – flottant exprimé en décimal

`chiffres` et `exposant` constituent chacun une suite de chiffres en écriture décimale.
`suffixe` détermine le type du flottant.

- `f` ou `F` : il s'agit d'un `float` ;
- aucun suffixe : il s'agit d'un `double` ;
- `l` ou `L` : il s'agit d'un `long double`.

Les flottants en décimal

```
<chiffres>(e|E)<exposant>[suffixe]
.<chiffres>(e|E)[exposant][suffixe]
[chiffres].<chiffres>(e|E)[exposant][suffixe]
```

Figure 8 – flottant exprimé en décimal

`chiffres` et `exposant` constituent chacun une suite de chiffres en écriture décimale.
`suffixe` détermine le type du flottant.

- `f` ou `F` : il s'agit d'un `float` ;
- aucun suffixe : il s'agit d'un `double` ;
- `l` ou `L` : il s'agit d'un `long double`.

Les flottants en décimal

```
<chiffres>(e|E)<exposant>[suffixe]
.<chiffres>(e|E)[exposant][suffixe]
[chiffres].<chiffres>(e|E)[exposant][suffixe]
```

Figure 8 – flottant exprimé en décimal

`chiffres` et `exposant` constituent chacun une suite de chiffres en écriture décimale.
`suffixe` détermine le type du flottant.

- `f` ou `F` : il s'agit d'un `float` ;
- aucun suffixe : il s'agit d'un `double` ;
- `l` ou `L` : il s'agit d'un `long double`.

Les flottants en décimal

```
<chiffres>(e|E)<exposant>[suffixe]
.<chiffres>(e|E)[exposant][suffixe]
[chiffres].<chiffres>(e|E)[exposant][suffixe]
```

Figure 8 – flottant exprimé en décimal

`chiffres` et `exposant` constituent chacun une suite de chiffres en écriture décimale.
`suffixe` détermine le type du flottant.

- `f` ou `F` : il s'agit d'un `float` ;
- aucun suffixe : il s'agit d'un `double` ;
- `l` ou `L` : il s'agit d'un `long double`.

Les flottants en décimal

```
<chiffres>(e|E)<exposant>[suffixe]
.<chiffres>(e|E)[exposant][suffixe]
[chiffres].<chiffres>(e|E)[exposant][suffixe]
```

Figure 8 – flottant exprimé en décimal

`chiffres` et `exposant` constituent chacun une suite de chiffres en écriture décimale.
`suffixe` détermine le type du flottant.

- `f` ou `F` : il s'agit d'un `float` ;
- aucun suffixe : il s'agit d'un `double` ;
- `l` ou `L` : il s'agit d'un `long double`.

Les flottants en décimal

```
<chiffres>(e|E)<exposant>[suffixe]
.<chiffres>(e|E)[exposant][suffixe]
[chiffres].<chiffres>(e|E)[exposant][suffixe]
```

Figure 8 – flottant exprimé en décimal

`chiffres` et `exposant` constituent chacun une suite de chiffres en écriture décimale.
`suffixe` détermine le type du flottant.

- `f` ou `F` : il s'agit d'un `float` ;
- aucun suffixe : il s'agit d'un `double` ;
- `l` ou `L` : il s'agit d'un `long double`.

Un petit exemple

Exemple

```
1 double a = 0.;
2 double b = 1e3;
3 float c = 1.2e2f
4 long double d = .1L
```

Les flottants en hexadécimal

```
0(x|X)<chiffres>(p|P)<exposant>[suffixe]
0(x|X).<chiffres>(p|P)[exposant][suffixe]
0(x|X)[chiffres].<chiffres>(p|P)[exposant][suffixe]
```

Figure 9 – flottant exprimé en hexadécimal

`chiffres` est, cette fois-ci, une séquence de chiffres en écriture hexadécimale.
`exposant` est constitué d'une séquence de chiffres en écriture décimale.
Attention `exposant` n'implique plus une multiplication par une puissance de 10 mais par une puissance de 2.

Les flottants en hexadécimal

```
0(x|X)<chiffres>(p|P)<exposant>[suffixe]
0(x|X).<chiffres>(p|P)[exposant][suffixe]
0(x|X)[chiffres].<chiffres>(p|P)[exposant][suffixe]
```

Figure 9 – flottant exprimé en hexadécimal

`chiffres` est, cette fois-ci, une séquence de chiffres en écriture hexadécimale.
`exposant` est constitué d'une séquence de chiffres en écriture décimale.
Attention `exposant` n'implique plus une multiplication par une puissance de 10 mais par une puissance de 2.

Les flottants en hexadécimal

```
0(x|X)<chiffres>(p|P)<exposant>[suffixe]
0(x|X).<chiffres>(p|P)[exposant][suffixe]
0(x|X)[chiffres].<chiffres>(p|P)[exposant][suffixe]
```

Figure 9 – flottant exprimé en hexadécimal

`chiffres` est, cette fois-ci, une séquence de chiffres en écriture hexadécimale.
`exposant` est constitué d'une séquence de chiffres en écriture décimale.
Attention `exposant` n'implique plus une multiplication par une puissance de 10 mais par une puissance de 2.

Les flottants en hexadécimal

```
0(x|X)<chiffres>(p|P)<exposant>[suffixe]
0(x|X).<chiffres>(p|P)[exposant][suffixe]
0(x|X)[chiffres].<chiffres>(p|P)[exposant][suffixe]
```

Figure 9 – flottant exprimé en hexadécimal

`chiffres` est, cette fois-ci, une séquence de chiffres en écriture hexadécimale.
`exposant` est constitué d'une séquence de chiffres en écriture décimale.
Attention `exposant` n'implique plus une multiplication par une puissance de 10 mais par une puissance de 2.

Les flottants en hexadécimal

```
0(x|X)<chiffres>(p|P)<exposant>[suffixe]
0(x|X).<chiffres>(p|P)[exposant][suffixe]
0(x|X)[chiffres].<chiffres>(p|P)[exposant][suffixe]
```

Figure 9 – flottant exprimé en hexadécimal

`chiffres` est, cette fois-ci, une séquence de chiffres en écriture hexadécimale.
`exposant` est constitué d'une séquence de chiffres en écriture décimale.
Attention `exposant` n'implique plus une multiplication par une puissance de 10 mais par une puissance de 2.

Un petit exemple pour la route

Exemple

```
1 long double a = 0x10P2l; // a vaut 64
2 float b       = 0X.1p4F; // b vaut 1
3 double c      = 0X1.1p4; // c vaut 17
```

Définition de variables

Les littéraux

Les littéraux pour les caractères

Les littéraux pour les caractères

```
[préfixe]'[caractère]'
```

Figure 10 – les littéraux pour les caractères

`préfixe` détermine le type du caractère :

- sans préfixe : il s'agit d'un `char` ;
- `u8` : c'est un `char` (cependant, il s'agit d'un caractère Unicode encodé en UTF-8 sur une seule unité de code) ;
- `u` : il s'agit d'un `char16_t` ;
- `U` : il s'agit d'un `char32_t` ;
- `L` : il s'agit d'un `wchar_t` ;

Les littéraux pour les caractères

```
[préfixe]'[caractère]'
```

Figure 10 – les littéraux pour les caractères

`préfixe` détermine le type du caractère :

- sans préfixe : il s'agit d'un `char` ;
- `u8` : c'est un `char` (cependant, il s'agit d'un caractère Unicode encodé en UTF-8 sur une seule unité de code) ;
- `u` : il s'agit d'un `char16_t` ;
- `U` : il s'agit d'un `char32_t` ;
- `L` : il s'agit d'un `wchar_t` ;

Les littéraux pour les caractères

```
[préfixe]'[caractère]'
```

Figure 10 – les littéraux pour les caractères

`préfixe` détermine le type du caractère :

- sans préfixe : il s'agit d'un `char` ;
- `u8` : c'est un `char` (cependant, il s'agit d'un caractère Unicode encodé en UTF-8 sur une seule unité de code) ;
- `u` : il s'agit d'un `char16_t` ;
- `U` : il s'agit d'un `char32_t` ;
- `L` : il s'agit d'un `wchar_t` ;

Les littéraux pour les caractères

```
[préfixe]'[caractère]'
```

Figure 10 – les littéraux pour les caractères

`préfixe` détermine le type du caractère :

- sans préfixe : il s'agit d'un `char` ;
- `u8` : c'est un `char` (cependant, il s'agit d'un caractère Unicode encodé en UTF-8 sur une seule unité de code) ;
- `u` : il s'agit d'un `char16_t` ;
- `U` : il s'agit d'un `char32_t` ;
- `L` : il s'agit d'un `wchar_t` ;

Les littéraux pour les caractères

`[préfixe]'[caractère]'`

Figure 10 – les littéraux pour les caractères

`préfixe` détermine le type du caractère :

- sans préfixe : il s'agit d'un `char` ;
- `u8` : c'est un `char` (cependant, il s'agit d'un caractère Unicode encodé en UTF-8 sur une seule unité de code) ;
- `u` : il s'agit d'un `char16_t` ;
- `U` : il s'agit d'un `char32_t` ;
- `L` : il s'agit d'un `wchar_t` ;

Les littéraux pour les caractères

```
[préfixe]'[caractère]'
```

Figure 10 – les littéraux pour les caractères

`préfixe` détermine le type du caractère :

- sans préfixe : il s'agit d'un `char` ;
- `u8` : c'est un `char` (cependant, il s'agit d'un caractère Unicode encodé en UTF-8 sur une seule unité de code) ;
- `u` : il s'agit d'un `char16_t` ;
- `U` : il s'agit d'un `char32_t` ;
- `L` : il s'agit d'un `wchar_t` ;

Les littéraux pour les caractères

```
[préfixe]'[caractère]'
```

Figure 10 – les littéraux pour les caractères

`préfixe` détermine le type du caractère :

- sans préfixe : il s'agit d'un `char` ;
- `u8` : c'est un `char` (cependant, il s'agit d'un caractère Unicode encodé en UTF-8 sur une seule unité de code) ;
- `u` : il s'agit d'un `char16_t` ;
- `U` : il s'agit d'un `char32_t` ;
- `L` : il s'agit d'un `wchar_t` ;

Définition de variables
Les tableaux

Définition d'un tableau

Définition

Un tableau est une succession continue d'éléments de même type en mémoire

```
<type> <identifiant>[<taille>];
```

Figure 11 – définition d'un tableau

On crée un tableau de `taille` éléments de type `type` que l'on identifie par le nom `identifiant`.
Attention La taille d'un tableau est statique : elle ne peut pas changer au cours de l'exécution du programme.
La taille d'un tableau caractérise son type.

Exemple

`int[5]` et `int[2]` sont tous deux des tableaux d'entiers mais ce sont deux types distincts.

Définition d'un tableau

Définition

Un tableau est une succession continue d'éléments de même type en mémoire

```
<type> <identifiant>[<taille>];
```

Figure 11 – définition d'un tableau

On crée un tableau de `taille` éléments de type `type` que l'on identifie par le nom `identifiant`.

Attention La taille d'un tableau est statique : elle ne peut pas changer au cours de l'exécution du programme.

La taille d'un tableau caractérise son type.

Exemple

`int[5]` et `int[2]` sont tous deux des tableaux d'entiers mais ce sont deux types distincts.

Définition d'un tableau

Définition

Un tableau est une succession continue d'éléments de même type en mémoire

```
<type> <identifiant>[<taille>];
```

Figure 11 – définition d'un tableau

On crée un tableau de `taille` éléments de type `type` que l'on identifie par le nom `identifiant`.

Attention La taille d'un tableau est statique : elle ne peut pas changer au cours de l'exécution du programme.

La taille d'un tableau caractérise son type.

Exemple

`int[5]` et `int[2]` sont tous deux des tableaux d'entiers mais ce sont deux types distincts.

Définition d'un tableau

Définition

Un tableau est une succession continue d'éléments de même type en mémoire

```
<type> <identifiant>[<taille>];
```

Figure 11 – définition d'un tableau

On crée un tableau de `taille` éléments de type `type` que l'on identifie par le nom `identifiant`.
Attention La taille d'un tableau est statique : elle ne peut pas changer au cours de l'exécution du programme.
La taille d'un tableau caractérise son type.

Exemple

`int[5]` et `int[2]` sont tous deux des tableaux d'entiers mais ce sont deux types distincts.

Définition d'un tableau

Définition

Un tableau est une succession continue d'éléments de même type en mémoire

```
<type> <identifiant>[<taille>];
```

Figure 11 – définition d'un tableau

On crée un tableau de `taille` éléments de type `type` que l'on identifie par le nom `identifiant`.
Attention La taille d'un tableau est statique : elle ne peut pas changer au cours de l'exécution du programme.
La taille d'un tableau caractérise son type.

Exemple

`int[5]` et `int[2]` sont tous deux des tableaux d'entiers mais ce sont deux types distincts.

Accès aux éléments

Si on veut accéder au `n`-ième élément de `identifiant` (les tableaux sont indexés à partir de 0), on utilise la syntaxe suivante :

```
<identifiant>[<n>];
```

Figure 12 – accès aux éléments d'un tableau

Attention Il n'y a aucune vérification de la taille du tableau : on accède à la `n`-ième addresse mémoire suivant celle du premier élément du tableau.

Exemple

On a donc :

```
1 int a[5] = {1, 2, 3, 4, 5};
2 int a[10]; // cette ligne est valide
```

Accès aux éléments

Si on veut accéder au `n`-ième élément de `identifiant` (les tableaux sont indexés à partir de 0), on utilise la syntaxe suivante :

```
<identifiant>[<n>];
```

Figure 12 – accès aux éléments d'un tableau

Attention Il n'y a aucune vérification de la taille du tableau : on accède à la `n`-ième addresse mémoire suivant celle du premier élément du tableau.

Exemple

On a donc :

```
1 int a[5] = {1, 2, 3, 4, 5};
2 int a[10]; // cette ligne est valide
```

Accès aux éléments

Si on veut accéder au `n`-ième élément de `identifiant` (les tableaux sont indexés à partir de 0), on utilise la syntaxe suivante :

```
<identifiant>[<n>];
```

Figure 12 – accès aux éléments d'un tableau

Attention Il n'y a aucune vérification de la taille du tableau : on accède à la `n`-ième addresse mémoire suivant celle du premier élément du tableau.

Exemple

On a donc :

```
1 int a[5] = {1, 2, 3, 4, 5};
2 int a[10]; // cette ligne est valide
```

Une petite astuce

Lorsque l'on initialise un tableau avec un nombre d'éléments données, on n'est pas obligé de spécifié sa `taille` : elle est déterminée par inférence par le compilateur.

Exemple

Les deux lignes suivantes sont valides et équivalentes :

```
1 int tableau[5] = {1, 2, 3, 4, 5};
2 int tableau[] = {1, 2, 3, 4, 5}; // tableau est cependant bien de type int[5]
```

Une petite astuce

Lorsque l'on initialise un tableau avec un nombre d'éléments données, on n'est pas obligé de spécifié sa `taille` : elle est déterminée par inférence par le compilateur.

Exemple

Les deux lignes suivantes sont valides et équivalentes :

```
1 int tableau[5] = {1, 2, 3, 4, 5};
2 int tableau[] = {1, 2, 3, 4, 5}; // tableau est cependant bien de type int[5]
```

Une question de taille

La taille du tableau doit être déterminée, strictement positive et suffisante.

Exemple

Ainsi, les deux lignes suivantes conduisent à une erreur à la compilation.

```
int tableau[]; // quelle est la taille de tableau ?
int tableau[0]; // certains compilateurs parviennent à la traiter mais ce comportement n'est pas standard
```

De plus, `taille` ne doit pas nécessairement être strictement égale au nombre d'éléments lors de l'initialisation.

```
int tableau[5] = {1, 2}; // tout va bien
int tableau[1] = {1, 2}; // c'est le drame
```

Une question de taille

La taille du tableau doit être déterminée, strictement positive et suffisante.

Exemple

Ainsi, les deux lignes suivantes conduisent à une erreur à la compilation.

```
1 int tableau[]; // quelle est la taille de tableau ?
2 int tableau[0]; // certains compilateurs parviennent à la traiter mais ce comportement n'est pas standard
```

De plus, `taille` ne doit pas nécessairement être strictement égale au nombre d'éléments lors de l'initialisation.

```
1 int tableau[5] = {1, 2}; // tout va bien
2 int tableau[1] = {1, 2}; // c'est le drame
```

Définition de variables
Initialisation de variables

Initialisation de variables

L'initialisation d'une variable consiste en l'affectation d'une valeur initiale à celle-ci.
En C++, il existe sept différents types d'initialisation :

- *default initialization*,
- *value initialization*,
- *copy initialization*,
- *list initialization*,
- *direct initialization*,
- *aggregate initialization*,
- *reference initialization*.

Pour l'instant, on n'abordera que les quatre premières notions.

Initialisation de variables

L'initialisation d'une variable consiste en l'affectation d'une valeur initiale à celle-ci.
En C++, il existe sept différents types d'initialisation :

- *default initialization*,
- *value initialization*,
- *copy initialization*,
- *list initialization*,
- *direct initialization*,
- *aggregate initialization*,
- *reference initialization*.

Pour l'instant, on n'abordera que les quatre premières notions.

Initialisation de variables

L'initialisation d'une variable consiste en l'affectation d'une valeur initiale à celle-ci.
En C++, il existe sept différents types d'initialisation :

- *default initialization*,
- *value initialization*,
- *copy initialization*,
- *list initialization*,
- *direct initialization*,
- *aggregate initialization*,
- *reference initialization*.

Pour l'instant, on n'abordera que les quatre premières notions.

Initialisation de variables

L'initialisation d'une variable consiste en l'affectation d'une valeur initiale à celle-ci.
En C++, il existe sept différents types d'initialisation :

- *default initialization*,
- *value initialization*,
- *copy initialization*,
- *list initialization*,
- *direct initialization*,
- *aggregate initialization*,
- *reference initialization*.

Pour l'instant, on n'abordera que les quatre premières notions.

Initialisation de variables

L'initialisation d'une variable consiste en l'affectation d'une valeur initiale à celle-ci.
En C++, il existe sept différents types d'initialisation :

- *default initialization*,
- *value initialization*,
- *copy initialization*,
- *list initialization*,
- *direct initialization*,
- *aggregate initialization*,
- *reference initialization*.

Pour l'instant, on n'abordera que les quatre premières notions.

Initialisation de variables

L'initialisation d'une variable consiste en l'affectation d'une valeur initiale à celle-ci.
En C++, il existe sept différents types d'initialisation :

- *default initialization*,
- *value initialization*,
- *copy initialization*,
- *list initialization*,
- *direct initialization*,
- *aggregate initialization*,
- *reference initialization*.

Pour l'instant, on n'abordera que les quatre premières notions.

Initialisation de variables

L'initialisation d'une variable consiste en l'affectation d'une valeur initiale à celle-ci.
En C++, il existe sept différents types d'initialisation :

- *default initialization*,
- *value initialization*,
- *copy initialization*,
- *list initialization*,
- *direct initialization*,
- *aggregate initialization*,
- *reference initialization*.

Pour l'instant, on n'abordera que les quatre premières notions.

Initialisation de variables

L'initialisation d'une variable consiste en l'affectation d'une valeur initiale à celle-ci.
En C++, il existe sept différents types d'initialisation :

- *default initialization*,
- *value initialization*,
- *copy initialization*,
- *list initialization*,
- *direct initialization*,
- *aggregate initialization*,
- *reference initialization*.

Pour l'instant, on n'abordera que les quatre premières notions.

Initialisation de variables

L'initialisation d'une variable consiste en l'affectation d'une valeur initiale à celle-ci.
En C++, il existe sept différents types d'initialisation :

- *default initialization*,
- *value initialization*,
- *copy initialization*,
- *list initialization*,
- *direct initialization*,
- *aggregate initialization*,
- *reference initialization*.

Pour l'instant, on n'abordera que les quatre premières notions.

Initialisation de variables

L'initialisation d'une variable consiste en l'affectation d'une valeur initiale à celle-ci.
En C++, il existe sept différents types d'initialisation :

- *default initialization*,
- *value initialization*,
- *copy initialization*,
- *list initialization*,
- *direct initialization*,
- *aggregate initialization*,
- *reference initialization*.

Pour l'instant, on n'abordera que les quatre premières notions.

default initialization

La syntaxe utilisée est la suivante :

```
<type> <identifiant>;
```

Figure 13 – default initialization

Quand on initialise une variable avec cette syntaxe, rien de particulier n'est réalisé hormis l'allocation de la zone mémoire correspondante dans la pile d'exécution : elle a donc une valeur quelconque qui dépend de ce qu'il y avait avant à cette adresse mémoire.

Une valeur quelconque ?

Exemple

```cpp
#include <iostream>

using namespace std;

int main()
{
    {
        int tableau[] = {1, 2, 3, 4, 5};
    } // tableau « meurt » ici

    int un_autre_tableau[1];

    cout << un_autre_tableau[2] << '\n';
}
```

Dans l'exemple ci-dessus, que va-t-il se passer ?

Une valeur quelconque ?

Exemple

```cpp
#include <iostream>

using namespace std;

int main()
{
    {
        int tableau[] = {1, 2, 3, 4, 5};
    } // tableau « meurt » ici

    int un_autre_tableau[1];

    cout << un_autre_tableau[2] << '\n';
}
```

Dans l'exemple ci-dessus, que va-t-il se passer ?

Il sera imprimé « 3 » car c'est la valeur qui a été affectée à cette zone mémoire lors de l'initialisation de `tableau`.

value initialization

Les deux syntaxes suivantes sont acceptées :

```
<type>();
<type> <identifiant>{};
```

Figure 14 – value initialization

Quand on initialise une variable avec cette syntaxe, la zone mémoire correspondante à la variable `identifiant` est initialisée à 0.

Attention La première syntaxe retourne une *rvalue*.

value initialization

Les deux syntaxes suivantes sont acceptées :

```
<type>();
<type> <identifiant>{};
```

Figure 14 – value initialization

Quand on initialise une variable avec cette syntaxe, la zone mémoire correspondante à la variable `identifiant` est initialisée à 0.
Attention La première syntaxe retourne une *rvalue*.

copy initialization

La syntaxe acceptée est la suivante :

```
<type> <identifiant> = <autre>;
```

Figure 15 – copy initialization

Le principe de fonctionnement est le même que pour le *direct initialization*.

list initialization

On pose $n \in \mathbb{N}^*$.
Les deux syntaxes suivantes sont acceptées :

```
<type> <identifiant>{<argument1>, …, <argumentn>};
<type> <identifiant> = {<argument1>, …, <argumentn>};
```

Figure 16 – list initialization

Dans le premier cas, on parle de direct list initialization, dans le second, de copy list initialization.
Ces syntaxes ne servent (pour l'instant) que pour l'initialisation de tableaux et elles sont toutes les deux équivalentes.

Pour bien comprendre les différents types d'initialisation

Exemple

```
1 int a;
2 int d = 8;
3 int e = {16};
4 int f = int();
5 int g{};
6 int h[] = {1, 2, 3};
```

Dans l'exemple ci-dessus, quels sont les types d'initialisation utilisée ?

Pour bien comprendre les différents types d'initialisation

Exemple

```
1 int a;
2 int d = 8;
3 int e = {16};
4 int f = int();
5 int g{};
6 int h[] = {1, 2, 3};
```

Dans l'exemple ci-dessus, quels sont les types d'initialisation utilisée ?

1. *default initialization*,

Pour bien comprendre les différents types d'initialisation

Exemple

```
1 int a;
2 int d = 8;
3 int e = {16};
4 int f = int();
5 int g{};
6 int h[] = {1, 2, 3};
```

Dans l'exemple ci-dessus, quels sont les types d'initialisation utilisée ?

1. *default initialization*,
2. *copy initialization*,

Pour bien comprendre les différents types d'initialisation

Exemple

```
1 int a;
2 int d = 8;
3 int e = {16};
4 int f = int();
5 int g{};
6 int h[] = {1, 2, 3};
```

Dans l'exemple ci-dessus, quels sont les types d'initialisation utilisée ?

1. *default initialization*,
2. *copy initialization*,
3. *copy initialization (narrowing interdit)*,

Pour bien comprendre les différents types d'initialisation

Exemple

```
1 int a;
2 int d = 8;
3 int e = {16};
4 int f = int();
5 int g{};
6 int h[] = {1, 2, 3};
```

Dans l'exemple ci-dessus, quels sont les types d'initialisation utilisée ?

1. *default initialization*,
2. *copy initialization*,
3. *copy initialization (narrowing interdit)*,
4. *value initialization*, puis *copy initialization*,

Pour bien comprendre les différents types d'initialisation

Exemple

```
1 int a;
2 int d = 8;
3 int e = {16};
4 int f = int();
5 int g{};
6 int h[] = {1, 2, 3};
```

Dans l'exemple ci-dessus, quels sont les types d'initialisation utilisée ?

1. *default initialization*,
2. *copy initialization*,
3. *copy initialization (narrowing interdit)*,
4. *value initialization*, puis *copy initialization*,
5. *value initialization*.

Pour bien comprendre les différents types d'initialisation

Exemple

```
1 int a;
2 int d = 8;
3 int e = {16};
4 int f = int();
5 int g{};
6 int h[] = {1, 2, 3};
```

Dans l'exemple ci-dessus, quels sont les types d'initialisation utilisée ?

1. *default initialization*,
2. *copy initialization*,
3. *copy initialization (narrowing interdit)*,
4. *value initialization*, puis *copy initialization*,
5. *value initialization*.
6. *copy list initialization*.

Un autre mot clef pour la route

En C++, il est possible d'utiliser le mot clef `auto` lors de l'initialisation d'une variable.

```
auto <identifiant> = <autre>;
```

Figure 17 – Utilisation de `auto`

Le type de `identifiant` est déterminé à partir de `autre` par inférence.

Exemple

```
1 auto a = 0.; // a est un double
2 auto b = 0.f; // b est un float
3 auto c = 5uLL; // c est un unsigned long long
4 auto d = false; // d est un bool
5 auto e; // erreur car il n'est pas possible de déterminer de type
```

Un autre mot clef pour la route

En C++, il est possible d'utiliser le mot clef `auto` lors de l'initialisation d'une variable.

```
auto <identifiant> = <autre>;
```

Figure 17 – Utilisation de `auto`

Le type de `identifiant` est déterminé à partir de `autre` par inférence.

Exemple

```
1 auto a = 0.; // a est un double
2 auto b = 0.f; // b est un float
3 auto c = 5uLL; // c est un unsigned long long
4 auto d = false; // d est un bool
5 auto e; // erreur car il n'est pas possible de déterminer de type
```

Merci pour votre écoute.