# Limitations des types entiers et réels

V. Lepetit, J. Sam, et J.-C. Chappelier

## 1 Limitations des types entiers

Le type `int` n'est pas le seul type qu'on peut utiliser pour déclarer des variables contenant des valeurs positives. Il existe aussi les types : `long int`, qu'on peut écrire aussi simplement `long`, et `short int`, qu'on peut écrire aussi simplement `short`. On peut également ajouter le mot-clé `unsigned` devant chacun de ces 3 types pour obtenir 3 nouveaux types qui servent à déclarer des variables contenant des entiers *positifs* (ou nuls). Par exemple, on peut écrire :

```
int m;
long int n;
long n2;
short int p;
unsigned int q;
unsigned long r;
unsigned short int s;
```

Cependant, tous ces types ne peuvent représenter que des valeurs comprises dans un certain intervalle. En C++, cet intervalle dépend du type, mais aussi du compilateur et de l'ordinateur utilisé. Il n'y a même aucune garantie pour que l'intervalle de valeurs du type `long int` soit plus grand que l'intervalle pour le type `int`, juste qu'il soit au moins aussi grand. Pour donner une idée de l'ordre de grandeur, voici néanmoins les intervalles pour ces types pour un ordinateur « 32 bits » et un compilateur « standard » :

| type | valeur minimale | valeur maximale |
|---|---|---|
| `short int` | -32'678 | +32'767 |
| `int` | -2'147'483'648 | +2'147'483'647 |
| `long int` | environ $-10^{18}$ | $+10^{18}$ |
| `unsigned short int` | 0 | 65535 |
| `unsigned int` | 0 | 4'294'967'295 |
| `unsigned long int` | 0 | environ $+2 \cdot 10^{18}$ |

Plus l'intervalle est grand, plus une variable du type correspondant occupera de place en mémoire.

*Remarques avancées :*

— pour voir les vraies valeurs sur votre ordinateur avec votre compilateur, vous pouvez utiliser les `numeric_limits`, et la bibliothèque `<limits>`, par exemple comme ceci :

```
#include <iostream>
#include <limits>
using namespace std;
int main()
{
  cout << "plus grand unsigned long : "
       << numeric_limits<unsigned long int>::max() << endl;
  cout << "plus petit int : "
       << numeric_limits<int>::min() << endl;
  return 0;
}
```

— pour information, mais ce ne sera pas du tout utilisé dans ce cours, il existe également depuis la norme C++11 des types entiers de taille parfaitement définie : `int8_t`, `uint8_t`, ..., `int64_t`, `uint64_t`.

Pour voir l'impact que peuvent avoir ces limitations en pratique, vous pouvez exécuter le programme suivant [1] :

```
#include <iostream>
using namespace std;

int main()
{
  int n(10);

  cout << n << endl;
  n = n * n;
  cout << n << endl;
  n = n * n;
  cout << n << endl;
  n = n * n;
  cout << n << endl;
  n = n * n;
  cout << n << endl;
  n = n * n;
  cout << n << endl;

  return 0;
}
```

Ce programme initialise la variable `n` à 10, et l'élève au carré plusieurs fois. Les valeurs affichées devraient donc être toutes des puissances de 10, mais en exécutant le programme,

1. Ce programme pourra se réécrire de façon plus compacte dès que vous aurez vu les boucles.

vous verrez que ce n'est pas le cas pour les dernières valeurs, qui sont trop grandes pour être représentées correctement par le type `int`. Vous pouvez également changer le type de `n` de `int` à `long int`, `short int`, `unsigned long int`, ... pour voir l'impact sur les valeurs calculées.

Soyez donc vigilants quand votre programme doit travailler avec de grandes valeurs !

# 2 Limitations des types réels

De la même façon, le type `double` ne peut pas représenter n'importe quel nombre réel, puisqu'il faudrait pour cela une précision infinie. Il existe également le type `float` qui est généralement plus limité que le type `double`, et le type `long double` qui est lui généralement moins limité. Voici les intervalles de valeurs pour ces types pour un ordinateur et un compilateur « standards » :

| type | valeur minimale | valeur maximale |
|---|---|---|
| `float` | $-3.4\ 10^{38}$ | $+3.4\ 10^{38}$ |
| `double` | $-1.8\ 10^{308}$ | $+1.8\ 10^{308}$ |
| `long double` | $-1.2\ 10^{4932}$ | $+1.2\ 10^{4932}$ |

*Remarque avancée :* vous pouvez aussi voir les vraies valeurs sur votre ordinateur et votre compilateur avec `numeric_limits`.

Mais en pratique, la limitation de ces types affecte surtout la *précision* des valeurs représentées : les valeurs réelles, y compris celles entre les intervalles donnés ci-dessus, ne peuvent pas toutes être représentées. Considérons le programme suivant :

```
#include <iostream>
#include <cmath>
using namespace std;

int main()
{
  double a(37.0);
  double racine( sqrt(a) );

  cout << a - racine * racine << endl;

  return 0;
}
```

Si le type `double` pouvait représenter parfaitement les valeurs réelles, ce programme afficherait 0. Or, sur mon ordinateur, j'obtiens environ la valeur $7\ 10^{-15}$, qui est une valeur très petite mais pas nulle. C'est parce que la variable `racine` ne peut stocker exactement la racine de `a`, et donc l'expression `racine * racine` ne vaut pas exactement `a`.

C'est pour ça que les tests d'égalité ou d'inégalité entre `double` (ou `float` ou `long double`) NE DEVRAIENT PAS ÊTRE utilisés [2]. Par exemple, le code suivant :

```
double a(37.0);
double racine( sqrt(a) );

if (a == racine * racine) {
  cout << "ok" << endl;
}
```

n'affiche rien, contrairement à ce qu'on pourrait s'attendre. Si vous devez absolument comparer des valeurs de type `double` vous pouvez utiliser un test tel que celui-ci :

```
\begin{verbatim}
  if (abs(a - racine * racine) < epsilon) {
    cout << "ok" << endl;
  }
```

où `epsilon` est une très petite valeur et `abs` calcule la valeur absolue. Cette valeur devrait être choisie selon la précision du type utilisé, mais comment déterminer cette valeur idéalement sort largement du cadre de ce cours.

2. Nous utilisons parfois de tels tests dans notre cours, mais uniquement par souci de simplicité.