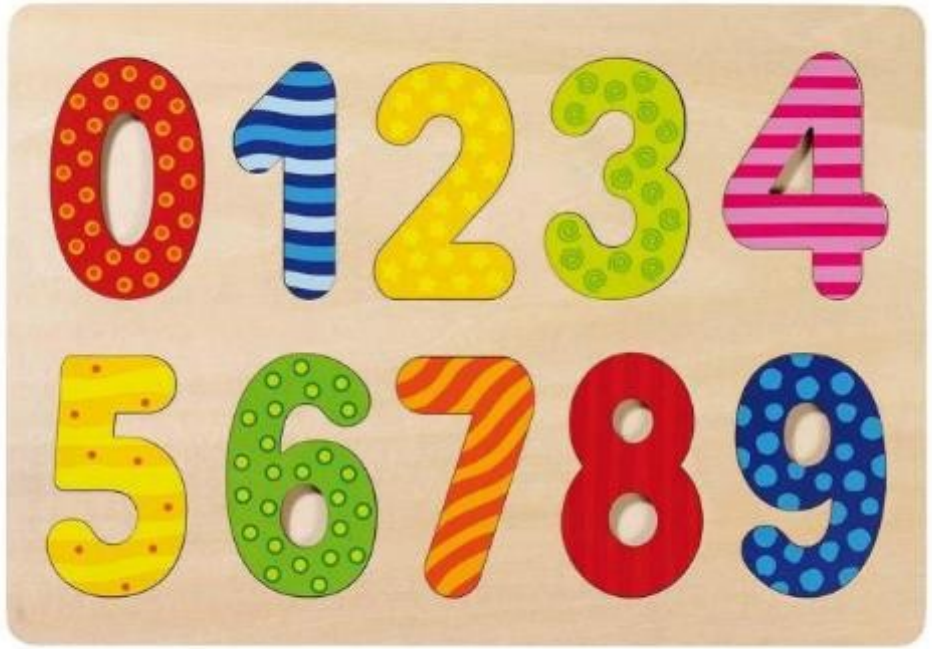


Chapitre 2
Eléments de base

Buts du chapitre

- Être capable de définir, de nommer et d'initialiser des variables et des constantes
- Comprendre les propriétés et les limitations des nombres entiers et en virgule flottante
- Écrire des expressions arithmétiques et des affectations en C++
- Apprécier l'importance des commentaires et d'une bonne présentation du code
- Créer des programmes avec des entrées / sorties utilisateurs de base
- Introduire les rudiments des chaînes de caractères

Plan du chapitre 2

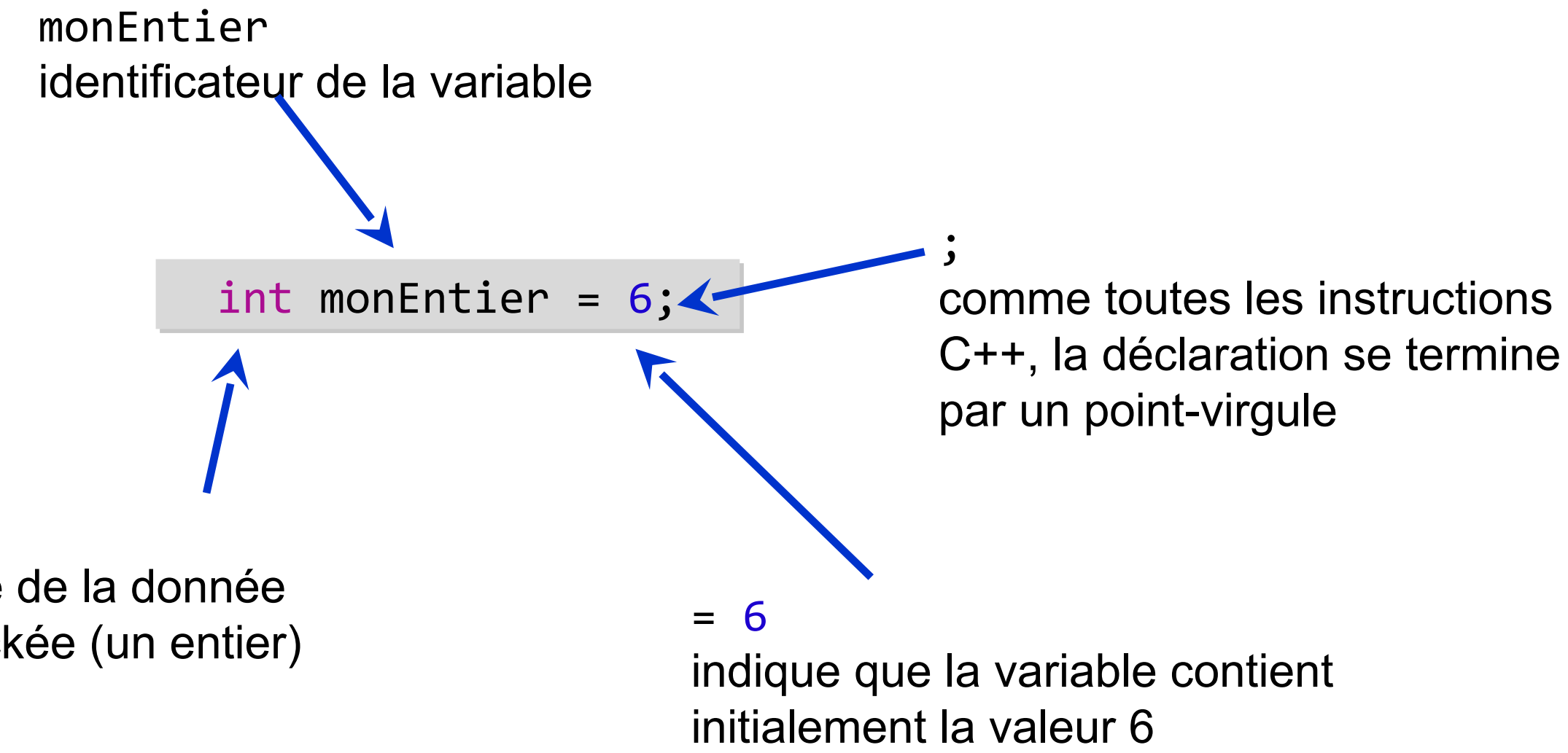
1. Variables et constantes

Variables et constantes

- Une variable sert à stocker une information ou un ensemble d'informations
- Ce qui est stocké dans une variable est appelé contenu de la variable
- Pour déclarer une variable, nous devons indiquer
  - le type de donnée stocké (nombre entier, nombre réel, caractère, …)
  - un identificateur unique et pertinent indiquant son contenu et son utilité
- Une variable peut être initialisée lors de sa création
  - … sans quoi son contenu est indéterminé

```
int monEntier = 6;
```

Exemple de variable



Exemples de déclarations

Déclaration	Commentaire
`int tables = 0;`	Déclare un entier et l'initialise à 0
`int total = tables + chaises;`	La valeur initiale peut être autre chose qu'une constante (ici, une expression). Toutefois, les variables `tables` et `chaises` doivent avoir été déclarées et initialisées préalablement.
`int chaises = "10";`	**Erreur :** on ne peut pas initialiser avec une valeur de type différent. Ici les guillemets indiquent une chaîne de caractères.
`int chaises;`	On peut déclarer une variable sans l'initialiser. Sa valeur reste donc indéterminée, ce qui peut être source d'erreur d'exécution.
`int tables, chaises;`	On peut déclarer ensemble plusieurs variables de même type en les séparant pas des virgules, ici sans initialisation.
`int tables  = 6,` `  chaises;`	Comme précédemment mais avec une initialisation pour `tables`, mais la variable `chaises` reste indéterminée.
`chaises = 1;`	Ce n'est pas une déclaration (le type `int` manque) mais une affectation … que nous verrons plus tard.

Initialisation (variantes)

L'initialisation d'une variable en C++ peut être effectuée de 3 manières différentes, qui sont équivalentes pour les types simples

- initialisation « comme en C »

```
int age = 6;
```

- initialisation « par constructeur »

```
int age(6);
```

- initialisation « uniforme », depuis C++11

```
int age{6};
```

Quand déclarer une variable ?

- Une seule règle : elle doit être déclarée avant son utilisation

- Une question de style
  - Certains vieux programmeurs préfèrent déclarer toutes les variables en début de programme (ou de bloc, ou de fonction)
  - Si le langage le permet, ce qui est le cas du C++, on recommande de déclarer les variables le plus tard possible, typiquement juste avant leur utilisation

Noms de variables

- Il est important de choisir un nom explicite qui indique à quoi sert cette variable

```
c = a * b;
```

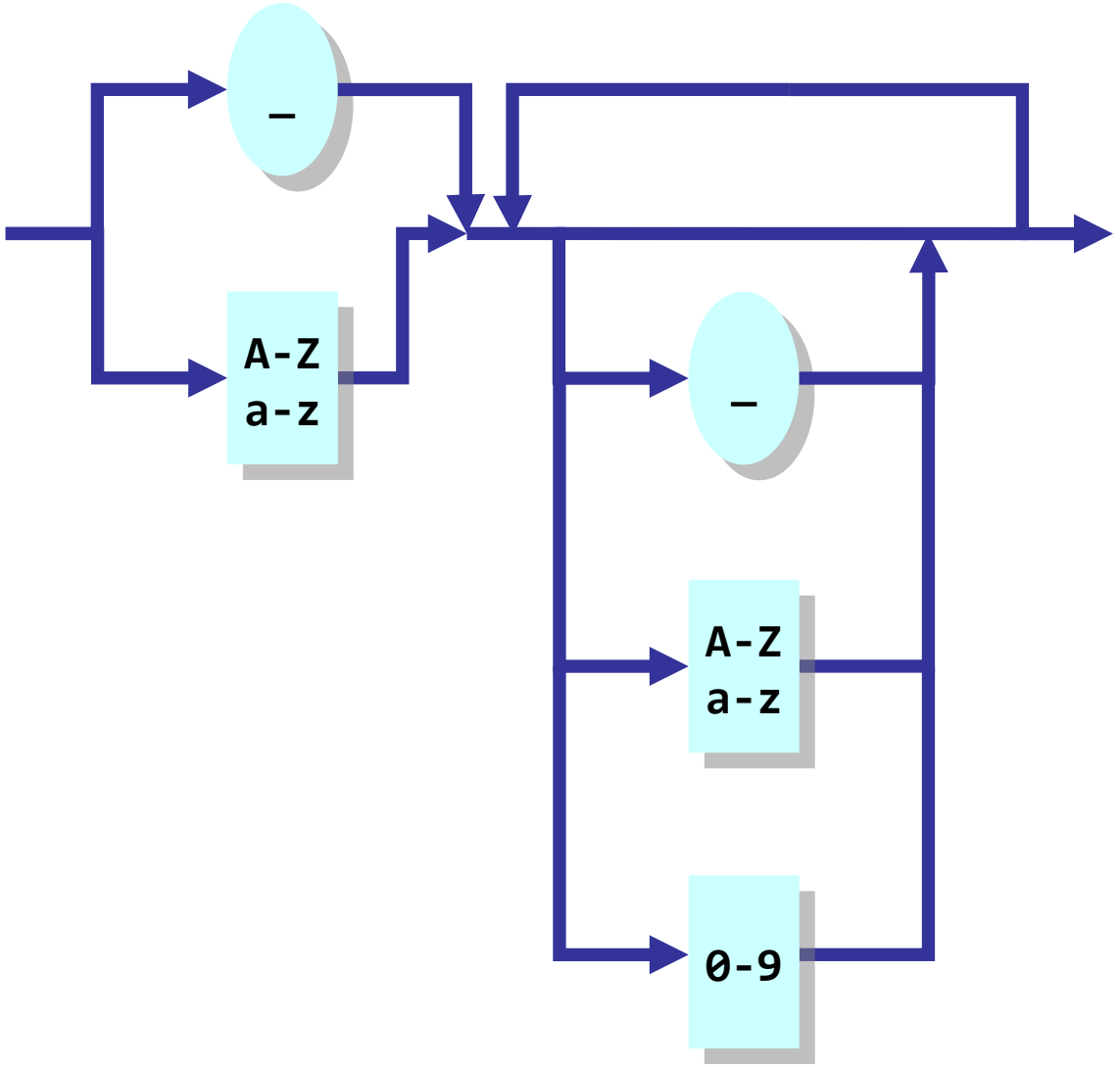
sera plus compréhensible avec des identificateurs parlants :

```
surface = largeur * hauteur;
```

- Les règles syntaxiques C++ pour les noms d'identificateurs doivent être respectées

En C++, un identificateur…

- doit commencer par une lettre ou un souligné « _ » (appelé aussi tiret bas)
- peut contenir lettres, chiffres et soulignés pour les symboles suivants
- est sensible à la casse
  (majuscules et minuscules sont différentes)
- n'a pas de limite de longueur
- ne peut être un des mots réservés C++



Et donc, un identificateur …

- *ne peut pas* commencer par un chiffre
- *ne peut pas* contenir de symbole tels que $ / & % …
- *ne peut pas* contenir d'espaces
  Solutions courantes pour les noms de variables avec plusieurs mots :
  - soit des soulignés `nbre_tables`
  - soit la notation dromadaire `nbreTables`

NB : la première lettre est en minuscule `nbre_tables` `nbreTables`

Attention

- Comme en C++ les identificateurs sont sensibles à la casse

  `volume` `Volume` `VOLUME` sont trois identificateurs différents

- Bonne pratique
  on utilise une convention n'autorisant qu'une seule graphie pour un mot donné (pour éviter de se tromper en écrivant le nom).

  Par exemple, les noms de variables

  - n'utilisent que des minuscules
  - ou utilisent la notation dromadaire

Exemples de noms

```
ABC  somme  TAUX
valMax  numerateur
```

Corrects

```
x  x1  x_1  i  _a
```

Corrects
… mais pas très parlants

```
taux Taux TAUX
```

Corrects
… mais différents

```
12 3x  %change
date-j  numérateur
```

Faux !

Mots réservés en C++

- Le langage réserve un certain nombre d'identificateurs qui ne peuvent pas être utilisés comme noms de variables dans vos déclarations

alignas (since C++11)	default(1)	register(2)
alignof (since C++11)	delete(1)	reinterpret_cast
and	do	requires (since C++20)
and_eq	double	return
asm	dynamic_cast	short
atomic_cancel (TM TS)	else	signed
atomic_commit (TM TS)	enum	sizeof(1)
atomic_noexcept (TM TS)	explicit	static
auto(1)	export(1)(3)	static_assert (since C++11)
bitand	extern(1)	static_cast
bitor	false	struct(1)
bool	float	switch
break	for	synchronized (TM TS)
case	friend	template
catch	goto	this
char	if	thread_local (since C++11)
char8_t (since C++20)	inline(1)	throw
char16_t (since C++11)	int	true
char32_t (since C++11)	long	try
class(1)	mutable(1)	typedef
compl	namespace	typeid
concept (since C++20)	new	typename
const	noexcept (since C++11)	union
consteval (since C++20)	not	unsigned
constexpr (since C++11)	not_eq	using(1)
constinit (since C++20)	nullptr (since C++11)	virtual
const_cast	operator	void
continue	or	volatile
co_await (since C++20)	or_eq	wchar_t
co_return (since C++20)	private	while
co_yield (since C++20)	protected	xor
decltype (since C++11)	public	xor_eq
	reflexpr (reflection TS)	

Sources : https://en.cppreference.com/w/cpp/keyword
(1) meaning changed or new meaning added in C++11
(2) meaning changed in C++17
(3) meaning changed in C++20

L'opérateur d'affectation

- Le contenu d'une variable peut changer au cours de l'exécution

- Son contenu peut être modifié
  - par un opérateur d'affectation (=)
  - par un opérateur d'incrémentation (++) ou de décrémentation (`--`)
  - en y transférant le résultat d'une saisie (`cin >>`)

L'opérateur d'affectation

- Remplace la valeur actuelle de la variable par une nouvelle valeur

```
age = 10;
```

affecte la valeur 10 à la variable age
… ce qui écrase la valeur qui s'y trouvait précédemment

Notez la différence entre la déclaration et l'affectation

```
int table = 6; // déclaration (avec initialisation)
...
table = 8;     // affectation (la variable doit exister)
```

L'opérateur d'affectation

- L'opérateur d'affectation = est une instruction qui copie la valeur de l'expression à droite dans la variable à gauche

- Il n'est pas rare d'évaluer une variable[1] (à droite du =) et de l'affecter (à gauche du =) dans une même instruction

```
compteur = compteur + 2;
```

[1] Évaluer une variable signifie récupérer sa valeur.

Constantes

- Certaines valeurs ne doivent pas changer au cours de l'exécution du programme
- Pour celles-ci, le C++ met à disposition le mot réservé `const` à placer juste avant la déclaration de la variable

```
const double PRIX_CAFE = 3.90;
```

- Son initialisation à la déclaration est obligatoire
- La bonne pratique veut que le nom d'une constante soit écrit en majuscules

Pourquoi utiliser des constantes ?

```
const double VOLUME_BOUTEILLE = 0.75;
const double VOLUME_CANETTE   = 0.33;
...
double litresVin   = nbBouteilles * VOLUME_BOUTEILLE;
double litresBiere = nbCanettes   * VOLUME_CANETTE;
```

- Le code est plus facile à comprendre et à maintenir en cas de changement
- Évite l'apparition d'une valeur magique inexpliquée au milieu du code

constexpr

- `constexpr` pour « constant expression »
  a été introduit avec C++11 et amélioré avec C++14

- L'initialisation d'une variable `constexpr` est obligatoire
  et peut se faire des 3 manières différentes déjà vues
  - Comme en C
  - Initialisation par constructeur
  - Initialisation « uniforme »

```
constexpr int PRIX = 6;
constexpr int PRIX(6);
constexpr int PRIX{6};
```

constexpr

- Au contraire d'une `const`, l'initialisation d'une déclaration `constexpr` doit être réalisable à la compilation
- A l'instar de `const`, `constexpr` fixe définitivement la valeur de l'objet
- `constexpr` s'utilise non seulement pour déclarer une constante mais aussi dans d'autres contextes comme, par exemple, la déclaration d'une fonction (chapitre 4)
- L'usage de `constexpr` permet au compilateur de réaliser certaines optimisations
- L'étude complète de `constexpr` sort du cadre de PRG1
  Nous n'examinerons que le cas des fonctions `constexpr` (chapitre 4)

2. Commentaires

Commentaires

- Ce sont des explications données à la personne qui lit le code

- Ils sont simplement ignorés par le compilateur

```
double volumeCanette = 0.355; // Litres dans une canette de 12 onces
```

BREAKING NEWS: il y a 355 ml dans une canette de 12 onces vendue aux USA

Commentaires – 2 styles

- Commentaire de ligne : après // et jusqu'à la fin de la ligne
  - tout ce qui précède le double slash (//) est du code
  - tout ce qui le suit est un commentaire et se termine à la fin de la ligne

- Commentaire de bloc : entre /* et */
  - tout ce qui suit barre-oblique-étoile est un commentaire jusqu'à ce que l'on rencontre les symboles étoile-barre-oblique
  - peut commencer et s'arrêter en cours de ligne, ou en occuper plusieurs
  - peut englober des commentaires de ligne
  - ne permet pas d'imbriquer d'autres commentaires de bloc

Commentaires - exemples

```
// Commentaire sur une ligne
```

```
double volume = 2; // il peut suivre du code
```

```
// On peut écrire sur plusieurs lignes
// en les commençant toutes par deux fois le symbole /
```

```
/*  ceci est un commentaire sur plusieurs lignes qui commence
    par slash-étoile et fini par étoile-slash */
```

```
/* mais rien n'empêche de n'écrire qu'une ligne */
```

```
double volume /* ou même en milieu de ligne */ = 2;
```

```
/* cette construction imbriquée /* n’est pas valide
 */ le commentaire s’arrête en réalité après les 2
premiers caractères de la ligne qui précède */
```

Qu'est-ce qu'un bon commentaire ?

- Un bon commentaire explique les actions qui vont être réalisées par une fonction, une instruction ou un bloc d'instructions.

- Un mauvais commentaire surcharge le code de formules redondantes et/ou d'explications inutiles.

- Attention

  Le choix des noms de vos variables et la structure de votre code est au moins aussi importante que vos commentaires pour la compréhension

Mettons tout cela ensemble

```
// Ce programme calcule (en litres) le volume d'un pack de NB_CANETTES_PAR_PACK
// canettes de soda plus le volume de NB_BOUTEILLES bouteilles de VOLUME_BOUTEILLE litres.
#include <cstdlib>
#include <iostream>
using namespace std;

int main() {
   const int    NB_CANETTES_PAR_PACK = 6;    // Nb de canettes dans un pack
   const double VOLUME_CANETTE       = 0.355;    // Volume en litres d'une canette
   double totalVolume = NB_CANETTES_PAR_PACK * VOLUME_CANETTE;

   cout << "Un pack de " << NB_CANETTES_PAR_PACK << " canettes contient "
        << totalVolume << " litres." << endl;

   const int    NB_BOUTEILLES    = 1;        // Nb de bouteilles
   const double VOLUME_BOUTEILLE = 2.0;      // Volume en litres d'une bouteille
   totalVolume += NB_BOUTEILLES * VOLUME_BOUTEILLE;

   cout << "Un pack de " << NB_CANETTES_PAR_PACK << " canettes et "
        << NB_BOUTEILLES << " bouteille" << (NB_BOUTEILLES > 1 ? "s" : "")
        << " contiennent " << totalVolume << " litres." << endl;

   return EXIT_SUCCESS;
}
```

3. Types de base

Les types en C++

- Les données en C++ doivent être typées

- Un type définit
  - comment cette donnée est stockée en mémoire (nombre de bits et codage)
  - les opérations possibles avec cette donnée

Les types en C++

- Les types fondamentaux (ou de base) fournis par le langage permettent de stocker des données simples
  - caractères typographiques
  - nombres entiers
  - nombres réels
  - booléens (chap. 3)
  - adresses mémoire (pointeurs)
- Les types composés, définis par les programmeurs ou fournis par des librairies, permettent de stocker des données complexes
  - tableaux (chap. 5)
  - chaînes de caractères (chap. 6)
  - classes définies par le programmeur (chap. 7)

Les types fondamentaux

Caractères

- Représentent un seul caractère tel que `A`, `$`, `1`
- Le type de base est `char`, d'autres types existent pour les caractères étendus

Entiers

- Stockent une valeur entière comme `-7` ou `1024`
- Le type de base est `int`, il peut être qualifié par la taille utilisée pour le stocker et le fait d'être signé ou pas

Réels

- Stockent des nombres réels tels que `3.14` ou `0.01`
- Le type de base est `double`, mais il existe des types permettant une représentation plus ou moins précise

Les types fondamentaux

Booléens

- Stockent une valeur vraie ou fausse (`true` / `false`)
- Le seul type booléen s'appelle `bool`

Pointeurs (voir cours PRG2)

- Stockent une adresse dans la mémoire, typiquement représentée par un nombre de 32 ou 64 bits
- Les noms des types pointeurs sont construits en ajoutant l'étoile `*` au type de base
  - `int*` stocke l'adresse d'une variable de type `int`
  - `double*` stocke l'adresse d'une variable de type `double`
  - `int**` stocke l'adresse d'une variable de type `int*`, …

Les types entiers

- En mémoire, les entiers sont stockés sous forme binaire
  - Le chiffre binaire (0 ou 1) est appelé bit
  - On y accède que par groupe de bits. Le plus petit groupe adressable est appelé byte
  - Normalement, le byte est composé de 8 bits: un octet
- Le nombre de bits utilisés détermine le nombre d'entiers représentables.
  - Avec n bits, on représente 2^n nombres
  - Avec m octets, on représente 2^{8m} nombres
- Les entiers peuvent être signés ou pas. Pour les entiers signés, stocker le signe consomme 1 bit. Avec n bits, on représente donc
  - Non signé: de 0 à 2^n-1, i.e. de 0 à 255 pour un octet
  - Signé: de -2^{n-1} à $2^{n-1}-1$, i.e. de -128 à 127 pour un octet

Les types entiers

- C++ permet de spécifier (partiellement)
  - le nombre de bits (minimal) avec `short`, `long` ou `long long`
  - le fait d'être signé ou pas avec `signed` ou `unsigned`

Signe	Nom du type	Taille
signé	**`signed char`**	= 1 byte, ≥ 8 bits
	`signed` **`short`** *`int`*	≥ char, ≥ 16 bits
	`signed` **`int`**	≥ short, ≥ 16 bits, typiquement 32
	`signed` **`long`** *`int`*	≥ int, ≥ 32 bits
	`signed` **`long long`** *`int`*	≥ long, ≥ 64 bits
non signé	**`unsigned char`**	idem type équivalent signé
	`unsigned short` *`int`*	
	`unsigned` *`int`*	
	`unsigned long` *`int`*	
	`unsigned long long` *`int`*	

Les types entiers

- En pratique, quel est l'intervalle des valeurs d'un type ? Cela dépend…
  - du compilateur (et des options de compilation spécifiant le processeur-cible)
  - du système d'exploitation (OS)
  - du processeur
- Par exemple, le type `long` utilise
  - 4 octets sous IA32-Windows compilé avec Visual C++ ou gcc
  - 4 octets sous IA64-Windows compilé avec Visual C++ ou gcc
  - 8 octets sur IA64-Linux compilé avec gcc
- Exemple avec gcc sur IA64-Mac OS X

```
cout << numeric_limits<long>::lowest() << " -> "
     << numeric_limits<long>::max();
```

```
-9223372036854775808 -> 9223372036854775807
```

Les types entiers

Typiquement, les intervalles des valeurs sont les suivants

Type T	numeric_limits<T>::lowest()	numeric_limits<T>::max()
signed char	-128	127
signed **short** *int*	-32768	32767
signed **int**	-2147483648	2147483647
signed **long** *int*	comme **int** ou **long long**	
signed **long long** *int*	-9223372036854775808	9223372036854775807
unsigned char	0	255
unsigned short *int*	0	65535
unsigned *int*	0	4294967295
unsigned long *int*	comme **unsigned int** ou **unsigned long long**	
unsigned long long *int*	0	18446744073709551615

Si c'est essentiel pour votre application, il est recommandé d'utiliser la bibliothèque **<cstdint>**

<climits> vs. <limits>

- Vous verrez certainement du code qui utilise `SHRT_MIN`, `SHRT_MAX`, `INT_MIN`, `INT_MAX`, `UINT_MIN`, `UINT_MAX`, …
  pour les valeurs limites des types

- Ces macros sont définies dans la librairie `<climits>`. Comme son nom l'indique, cette librairie vient du C

- En C++, il est plus approprié d'utiliser les fonctions fournies par la librairie `<limits>`

  - `numeric_limits<TYPE>::lowest()`
  - `numeric_limits<TYPE>::max()`

  où `TYPE` spécifie le type choisi (`short`, `int`, `unsigned`, …)

Entiers de taille fixe (C++11)

Depuis C++11 (et C99), la librairie **<cstdint>** permet de spécifier le nombre de bits exact souhaité pour un entier grâce aux nouveaux types suivants

Type T	numeric_limits<T>::lowest()	numeric_limits<T>::max()
int8_t	-128	127
int16_t	-32768	32767
int32_t	-2147483648	2147483647
int64_t	-9223372036854775808	9223372036854775807
uint8_t	0	255
uint16_t	0	65535
uint32_t	0	4294967295
uint64_t	0	18446744073709551615

… mais l'existence de ces types n'est pas garantie pour toutes les implémentations.

Entiers de taille fixe (C++11)

`<cstdint>` définit aussi, pour `N = 8, 16, 32` et `64`, les types

`int_fastN_t`
type signé d'au moins N bits sur lequel les calculs sont les plus rapides

`int_leastN_t`
type signé d'au moins N bits utilisant le moins de bits possible

`uint_fastN_t` et `uint_leastN_t`
types non signés équivalents

Tous ces types sont des alias des types de base

Les types dont ils sont synonymes dépendent de l'architecture-cible

En pratique…

- Dans la plupart des cas, nous utiliserons le type `int` qui, typiquement :
  - permet de compter de -2 à 2 milliards
  - pour seulement 4 octets en mémoire
  - avec 4GB de RAM, on peut stocker un milliard de telles variables

- Dans d'autres cas, pour compter la population mondiale par exemple, nous utiliserons un type d'entier plus long

Les types réels

- C++ fournit trois types de nombres réels
  - `float` simple précision
  - `double` double précision
  - `long double` précision étendue
- Le nom `float` provient de leur représentation en virgule flottante, appelée floating-point en anglais
  - Considérons par exemple les nombres 29600, 2.96, et 0.0296.
    On peut les représenter de manière similaire: une séquence de chiffres significatifs: 296 et une indication de la position de la virgule. Quand la valeur est multipliée ou divisée par 10, seule la position de la virgule change. On dit qu'elle « flotte ».
  - Les ordinateurs utilisent la base 2, pas la base 10, mais le principe est le même.

Les types réels

Comment les réels sont-ils codés (en nombre de bits) ?

Type et nombre de bits		Signe	Mantisse	Exposant	Précision
`float`	32	1	23	8	6
`double`	64	1	52	11	15
`long double`	80	1	64	15	17

Valeurs indicatives, dépendent du système

Les types réels

- `float` couvre un intervalle compris entre -10^{38} et 10^{38} environ
- `double` couvre un intervalle compris entre -10^{308} et 10^{308} environ
- Le fait qu'un nombre réel soit représenté en mémoire sur un nombre fini de bits a pour conséquence que (généralement) seule une approximation de ce nombre peut être implémentée
  - Plus le nombre de bits utilisé pour la représentation est grand, meilleure est l'approximation. Ainsi, un nombre réel stocké dans une variable de type *double* (64 bits) sera plus précis que ce même nombre stocké dans une variable de type *float* (32 bits).

Les types réels

- **Exemple**

```
float  f = 0.1f;
double d = 0.1;
```

  - Si on affiche les 20 premières décimales de chacune des variables, on obtient :
    - Pour f : 0.10000000149011611938 **significatifs (7)**[1] **pas significatifs**
    - Pour d : 0.10000000000000000555 **significatifs (16)** **pas significatifs**

[1] pour f, il y a en effet 7 chiffres significatifs car 0.1 devient en notation scientifique : **1.000000…e-01**, soit 1 chiffre significatif pour la partie entière de la mantisse + 6 chiffres significatifs (= précision) pour la partie fractionnaire de la mantisse.

- Le fait qu'un nombre réel ne puisse être représenté que de manière approximative (et non de manière exacte) engendre divers problèmes
  - En particulier, le test d'égalité entre deux réels est problématique (voir chap. 3)
  - La représentation des nombres réels en mémoire et les propriétés qui en résultent (par exemple la précision) seront étudiées dans le cours SYL

<limits> pour les réels

- Pour les nombres réels, il faut distinguer les fonctions
  - `lowest()` : la valeur négative de plus grande valeur absolue
  - `min()` : la valeur strictement positive de plus petite valeur absolue
  - `epsilon()` : la valeur strictement positive de plus petite valeur absolue

```
cout << numeric_limits<float>::lowest(); // -3.40282e+38
cout << numeric_limits<float>::min();    //  1.17549e-38
cout << numeric_limits<float>::max();    //  3.40282e+38
cout << numeric_limits<float>::epsilon();//  1.19209e-07
```

- Pour les nombres entiers, les `min` et `lowest` retournent la même valeur

En pratique…

- Dans la plupart des cas, nous utiliserons par défaut le type `double` qui offre une excellente précision

- Dans certains cas particuliers où la taille mémoire est critique, mais pas la précision, il peut être intéressant de travailler avec `float`

4. Expressions et opérateurs arithmétiques

Expressions

- En C++, tout ce qui correspond à une valeur est une expression
- On dit qu'elle « renvoie » une valeur

```
int valeur = 3;

4         // renvoie la valeur 4
valeur    // renvoie la valeur 3
5 + 2     // renvoie la valeur 7
```

- Nous en avons déjà vu plusieurs sortes
  - Les littéraux constants
  - L'opérateur d'affectation

Littéraux constants

- La plus simple des expressions est une constante exprimée littéralement
  - un caractère : `'A'` entre simples guillemets
  - une chaîne de caractères : `"Hello, World!"` entre doubles guillemets
  - un entier : `42`
  - un réel : `3.14`

Littéraux constants - entiers

- Les entiers peuvent être écrits dans 4 bases (mais toujours stockés en binaire)
  - décimal (base 10) — par défaut
  - binaire (base 2) — avec le préfixe `0b`
  - octal (base 8) — avec le préfixe `0`
  - hexadécimal (base 16) — avec le préfixe `0x`
    les chiffres `A,B,C,D,E` et `F` correspondent aux valeurs de `10` à `15`
- Ainsi, les 4 lignes suivantes sont exactement synonymes

```
int valeur = 42;
int valeur = 0b101010;
int valeur = 052;
int valeur = 0x2A;
```

Littéraux constants - entiers

- Par défaut, les constantes entières décimales sont du *premier* type qui permet de les contenir parmi
  - `int`
  - `long int`
  - `long long int`
- On peut préciser le type en ajoutant des lettres à la fin du nombre
  - un suffixe `U` ou `u` pour `unsigned`
  - un suffixe `L` ou `l` pour `long`
  - ou la combinaison des deux

```
123U     9u
123L     9l
0xCEB0CUL    07lu
```

Littéraux constants - réels

- Les constantes de type réel s'écrivent par leur valeur numérique
- La présence du point (le séparateur décimal en anglais) ou de la notation scientifique (avec E ou e) spécifie qu'il s'agit de réels et pas d'entiers

```
3.14152   22.5E-3   .1000
1.   1.2F   9.5L   10e-3
```

`float` `long double`

- Elles sont de type `double`
  … sauf en présence d'un suffixe `L`, `l`, `F` ou `f`

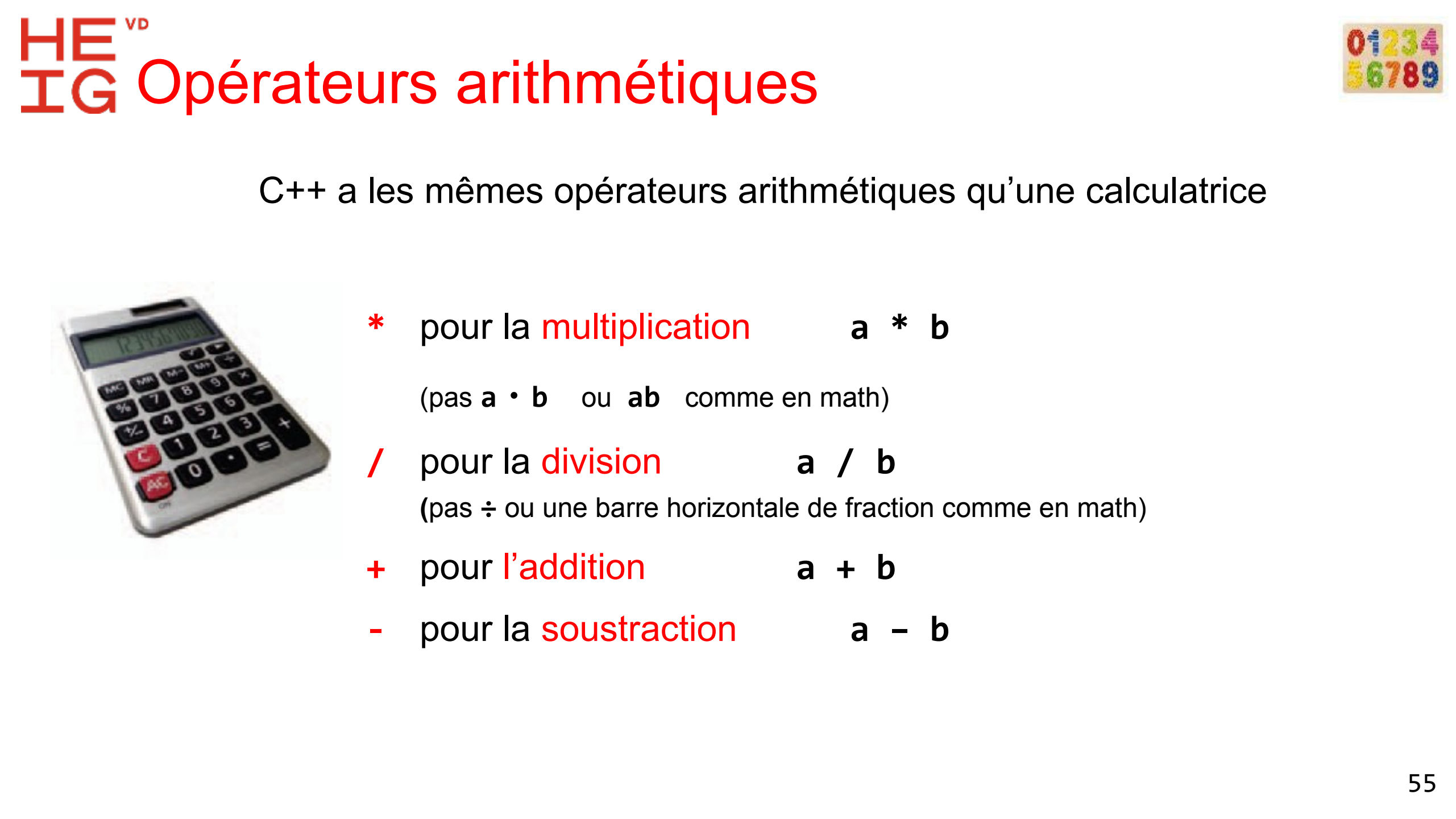
Opérateur d'affectation

- L'opérateur d'affectation est lui aussi une expression qui renvoie une valeur, … c'est ce qui permet d'écrire une instruction telle que

```
z = y = x = 42;
```

- Elle s'effectue en 3 temps
  - `x = 42` affecte la valeur `42` à `x` et renvoie la valeur `42`
  - `y = 42` affecte la valeur `42` à `y` et renvoie la valeur `42`
  - `z = 42` affecte la valeur `42` à `z` et renvoie la valeur `42` qui n'est pas utilisée

Opérateurs arithmétiques



C++ a les mêmes opérateurs arithmétiques qu'une calculatrice

`*` pour la multiplication `a * b`

(pas `a · b` ou `ab` comme en math)

`/` pour la division `a / b`

(pas ÷ ou une barre horizontale de fraction comme en math)

`+` pour l'addition `a + b`

`-` pour la soustraction `a - b`

Arithmétique sur les réels

- Sur les types réels (`float`, `double`, `long double`),
  les opérateurs se comportent comme en mathématiques

```
double resultat;

resultat = 1.0 + 3.0;          // vaut 4.0
resultat = 7.0 - 3.0;          // vaut 4.0
resultat = 3.14 * 2.0;         // vaut 6.28
resultat = 5.0  / 2.0;         // vaut 2.5
```

Arithmétique sur les entiers

- Sur les types entiers (`char`, `short`, `int`, `long` , `long long`),
  les opérateurs `+` , `-` et `*` se comportent comme pour les réels
- L'opérateur `/` effectue une division entière !!
  - `5/2` vaut `2` (entier) et non `2.5` (double)
- L'opérateur `%` calcule le reste d'une division entière, appelé souvent modulo
  - `5%2` vaut `1`
- Pour `a` et `b` entiers, on a toujours l'égalité suivante :

```
(a/b) * b + (a%b)  vaut  a
```

Division entière et modulo

- Pour les entiers négatifs,
  il suffit de se souvenir que a%b est du même signe que a

```
int a = 7, b = 4;    // ou -7, ou -4
cout << "a = " << a << ", ";
cout << "b = " << b << ", ";
cout << "a/b = " << a/b << ", ";
cout << "a%b = " << a%b << endl;
```

```
a =  7, b =  4, a/b =  1, a%b =  3
a = -7, b =  4, a/b = -1, a%b = -3
a =  7, b = -4, a/b = -1, a%b =  3
a = -7, b = -4, a/b =  1, a%b = -3
```

Application

- Combien font 1729 cents (appelés aussi *pennies*) en dollars ?


```
int pennies = 1729;
int dollars = pennies / 100;  // dollars vaut 17
int cents   = pennies % 100;  // cents vaut 29
```

- Réponse : 17 dollars et 29 cents

Opérateurs d'affectation composée

- Il est courant de modifier une variable pour lui ajouter ou soustraire une valeur, la multiplier par une valeur, etc.

Expression	Équivalent avec affectation composée
`nbre = nbre + 2;`	`nbre += 2;`
`total = total - rabais;`	`total -= rabais;`
`bonus = bonus * 2;`	`bonus *= 2;`
`prix  = prix / 2;`	`prix  /= 2;`
`taux  = taux % 100;`	`taux  %= 100;`
`somme = somme *(a + b);`	`somme *= a + b;`

Incrémentation / décrémentation

- Incrémenter et décrémenter de 1 est une opération si fréquente qu'il y a en C++ des opérateurs qui y sont dédiés : `++` et `--`

```
int compteur = 0;

compteur = compteur + 1; // vaut 1
compteur += 1;           // vaut 2
compteur -= 1;           // vaut 1
compteur++;              // vaut 2
compteur--;              // vaut 1
```

- Vous comprenez maintenant pourquoi le C++, évolution du langage C, porte ce nom.

Incrémentation pré et postfixe

- On peut placer l'opérateur unaire ++ (ou --) en position :
  - préfixe, c'est-à-dire avant la variable à incrémenter
  - postfixe, c'est-à-dire après la variable à incrémenter
- La valeur de retour de l'expression est différente
  - si préfixe, retourne la valeur après incrémentation
  - si postfixe, retourne la valeur avant incrémentation

```
int compteur = 42;
compteur++;                 // compteur vaut 43
++compteur;                 // compteur vaut 44
int pre = ++compteur;       // compteur vaut 45
                            // et pre vaut 45
int post = compteur++;      // compteur vaut 46
                            // mais post vaut 45
```

Attention

- Il est possible d'écrire du code au comportement indéfini

```
int i = 2;
int j = ++i * i++;
```

Après ces 2 lignes, `i` vaudra toujours `4`, mais `j` peut valoir `12` ou `9`, car l'ordre dans lequel l'ordre dans lequel les sous-expressions sont évaluées (ici `++i` et `i++`) n'est pas spécifié par le langage

- Ne jamais modifier la même variable plus d'une fois dans la même expression

Expressions mathématiques

- Comment écrire cette formule en C++ ?

$$b + \left(1 + \frac{r}{100}\right)^n$$

- La partie entre parenthèses s'écrit simplement `( 1 + r / 100 )`
- Mais comment écrire *à la puissance n* ?
  - C++ ne propose pas d'opérateur puissance
  - La librairie `<cmath>` propose une fonction **`pow(base, exposant)`**

```
#include <cmath>
using namespace std;
...
double resultat = b + pow(1 + r / 100, n);
```

#include <cmath>

- `<cmath>` fournit les fonctions
  - Trigonométriques — `sin, cos, tan, asin, acos,`
    …
  - Hyperboliques — `cosh, sinh, tanh, asinh,`
    …
  - Exponentielles et logarithmiques — `exp, log, …`
  - Puissances — `pow, sqrt, …`
  - De valeur absolue — `abs, fabs, …`
  - D'arrondi — `round, ceil, floor, …`
  - …
- http://www.cplusplus.com/reference/cmath/

5. Priorités des opérateurs

Opérateurs - priorités

- Comme en mathématiques, les opérateurs multiplicatifs (`*`, `/` et `%`) ont priorité sur les opérateurs additifs (`+` et `-`)
  - `2 * 3 + 4` vaut `10`, et pas `14`
- Si vous voulez changer l'ordre des calculs, il suffit d'ajouter des parenthèses
  - `2 * (3 + 4)` vaut `14`
- Quand deux ou plusieurs opérateurs arithmétiques ont la même priorité, ils sont appliqués de gauche à droite dans l'expression

https://en.cppreference.com/w/cpp/language/operator_precedence

Attention

- Mathématiquement, l'ordre d'application d'additions et soustractions successives ne change rien
- En C++, cela peut avoir un impact (à cause des dépassements ou arrondis)

```
float a = 2e38f;
cout << a + a - a  << " " << a - a + a << endl;
```

```
inf 2e+38
```

```
float a = 2e38f;
float b = 42;
cout << a - a + b << " " << b + a - a << endl;
```

```
42 0
```

Attention

- Les symboles + et – ne représentent pas seulement les opérateurs binaires d'addition et de soustraction
- Ils servent aussi comme opérateurs unaires de signe
- Ces opérateurs unaires sont prioritaires sur les opérateurs arithmétiques binaires, multiplicatifs ou additifs. Par exemple :
  - `2 * -3 + 4` vaut `-2`
  - `2 * -a + 4` vaut `-2` si a vaut 3
  - `2 * -+-+3 + 4` est une expression valide (mais peu lisible) qui vaut `10`

Que valent les expressions suivantes ?

Expressions	Résultats
`2 + 3 * 3`	11
`10 - 10 / 3`	7
`10 % 3 + 1`	2
`4 + 2 * 2 * 5 - 1`	23
`(4 + 2) * 2 * (5 - 1)`	48
`10.0 / 3.0`	3.33
`10.0 - 2.0 % 2.0`	Erreur!
`(3.2 + 0.6) * -2.0`	-7.6

Une question de style

Il est plus simple de lire

```
x1 = (-b + sqrt(b * b – 4 * a * c)) / (2 * a);
```

que

```
x1=(-b+sqrt(b*b-4*a*c))/(2*a);
```

Lesespacessimplifientvraimentlalecture!

Entourez donc toujours d’espaces les opérateurs binaires `+ - * / % =`

Une question de style

Par contre, ne pas mettre d'espace après un `+` ou un `-` unaire tel que `-b`

Cela permet de le distinguer plus facilement du symbole `-` de l'opérateur binaire de soustraction, comme dans `a - b`

Normalement on ne met pas d'espace après une fonction. Pour la racine carrée par exemple, on écrit `sqrt(x)` et pas `sqrt (x)`

6. Dépassement

Dépassement d'entier *(integer overflow)*

- Que se passe-t-il si l'on veut stocker un entier hors de l'intervalle d'un type donné ?

Vérifions-le avec le type `unsigned int`.

```
unsigned vMax = numeric_limits<unsigned>::max();
unsigned vOver = vMax + 1;
cout << vMax << " + 1 = " << vOver << endl;
```

Nous obtenons

```
4294967295 + 1 = 0
```

Vol 501 d'Ariane 5

- Vol inaugural, 4 juin 1996
- Même système de guidage inertiel qu'Ariane 4
- Accélérations 5 fois plus fortes qu'Ariane 4
- Dépassement dans le calcul de la position à partir des accélérations
- L'ordinateur de bord ordonne un virage serré pour corriger la trajectoire
- L'accélération latérale arrache un des boosters latéraux
- La destruction automatique est engagée

Dépassement d'entier non signé

C99 standard (§6.2.5/9) - A computation involving unsigned operands can never overflow, because a result that cannot be represented by the resulting unsigned integer type is reduced modulo the number that is one greater than the largest value that can be represented by the resulting type

Pour un entier non signé stocké sur *n* bits,
toute l'arithmétique s'effectue modulo 2^n

Exemple avec le type `char` stocké sur 8 bits ($2^8 = 256$)

```
unsigned char a = 128;
unsigned char b = 130;
unsigned char c = a + b;   // = 2
unsigned char d = a * b;   // = 0
unsigned char e = a - b;   // = 254
```

Dépassement d'entier signé

C99 standard (§3.4.3/1) - An example of undefined behavior is the behavior on integer overflow

Pour les entiers signés, le comportement est non défini.

- En pratique, il dépend de la représentation utilisée (p.ex. signe et magnitude, complément à 1, ou complément à 2).
- Le compilateur a le droit d'exploiter ce comportement indéfini pour optimiser le code
- Un compilateur qui renverrait toujours 42 en cas de dépassement signé respecterait lui aussi le standard

Optimisation

- Considérons le code suivant

```
int x = std::numeric_limits<int>::max();
cout << x << " < " << (x + 1) << " ? "
     << boolalpha << ( x < x + 1 ) << endl;
```

- Compilé sans optimisation, il affiche

```
2147483647 < -2147483648 ? false
```

- Avec optimisation, il affiche par contre

```
2147483647 < -2147483648 ? true
```

Comment éviter les dépassements ?

- Choisir un format suffisamment grand
  - Pour les données
  - … mais aussi pour les résultats des calculs effectués sur ces données
- Tester avant de calculer
  - `a * b` dépasse le type `int` si
    `std::numeric_limits<int>::max() / a < b`
  - `a + b` dépasse le type `int` si
    `std::numeric_limits<int>::max() - a < b`

Comment éviter les dépassements ?

- En tenir compte quand vous écrivez les formules mathématiques

```
unsigned a = 2000000001;
unsigned b = 3000000001;
 // et vous savez que b >= a

unsigned c = (a + b) / 2;         //  352516353

unsigned d = (a / 2) + (b / 2); // 2500000000

unsigned e = a + (b - a) / 2;    // 2500000001
```

Dépassement - réels

- Que se passe-t-il lorsque la valeur maximale est dépassée à l'exécution ?

```
float monFloat = 1e38f;
cout << "monFloat : " << monFloat << endl;

monFloat = monFloat * 10;
cout << "monFloat : " << monFloat << endl;
```

```
monFloat : 1e+038
monFloat : inf
```

NB: dépend de l'environnement

7. Conversions explicites entre types

Conversions explicites ou implicites

- C++ permet de convertir des valeurs numériques d'un type à l'autre
  - explicitement – le programmeur écrit en toutes lettres le type dans lequel convertir la valeur
  - implicitement – le compilateur décide de la conversion à appliquer lors de l'évaluation d'expression

```
unsigned char a = 30;
 // conversion implicite de l'expression
 // littérale 30 du type int vers unsigned char
```

- Tous les types numériques sont convertibles en tous les autres
- Les conversions peuvent occasionner des pertes de précision ou des débordements

Conversions explicites

- On veut explicitement convertir une valeur dans un autre type
- Il existe deux manières de le faire en C++
  - L'opérateur cast

```
(int)       3.15;
(long int) 2.9834;
(float)     12;
```

  - La forme fonctionnelle

```
int   (3.15);
long  (2.9834);
float (12);
```

la forme fonctionnelle ne peut comporter qu'un seul identificateur

`long int (3.14)` n'est pas possible

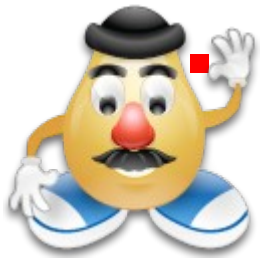
Conversions entières

C++ distingue deux types de conversion entre types entiers

- Promotions numériques
  - Conversions des types plus petits que `int` vers `int` (ou exceptionnellement vers `unsigned int`)
- Ajustements de types
  - Lorsque la conversion ne peut se faire par promotion
  - On distingue les conversions
    - Vers un type non signé
    - Vers un type signé

Promotions numériques entières

- Préservent toujours la valeur convertie
- Sont appliquées implicitement avant l'évaluation de tout opérateur arithmétique, ceux-ci ne prenant pas d'arguments plus petits que `int`
- Les promotions sont
  - `signed char` ou `signed short` → `int`
  - `unsigned char` ou `unsigned short` → `int`
    si le type `int` utilise strictement plus de bits que le type d'origine, ce qui permet une conversion sans perte
  - `unsigned char` ou `unsigned short` → `unsigned int` sinon
  - La norme ne spécifie pas si le type `char` est `signed char` ou `unsigned char`



Autres conversions entières

- Vers un type non signé
  - représenté sur *n* bits, la valeur est convertie modulo 2^n

```
unsigned char a = 300;  // a = 44
unsigned char b = -20;  // b = 236
```

- Vers un type signé
  - Si la valeur convertie est représentable, elle ne change pas
  - Sinon, le résultat est indéfini (dépend de l'implémentation)

Pour une représentation en complément à deux, on aura normalement

```
signed char a =  300;  // a = 44
signed char b = -200;  // b = 56
```

Conversions réelles

- La conversion de `float` en `double` est une promotion numérique
  Cela n'aura pas d'incidence avant le chapitre 4
- Pour les autres conversions entre `float`, `double` et `long double`
  - Si la valeur peut être représentée exactement, elle ne change pas

```
float a = 0.5;
```

  - Si la valeur est comprise entre deux valeurs représentables, le choix entre ces deux valeurs dépend de l'implémentation. *(arithmétique IEEE: la plus proche)*

```
float b = 3.14159265359;
```

  - Sinon, le résultat est indéfini

```
float c = 1e41;
```

Conversions entier ⇆ réel

- La conversion d'entier à réel suit essentiellement les mêmes règles que celles de réel vers réel

- Attention à la taille de la mantisse
  - `float` typiquement sur 23 bits
  - `double` typiquement sur 52 bits

```
float f = 123456789;
cout << fixed << setprecision(0) << f << endl;
```

123456792

Conversion réel ⇒ entier

- Lors de la conversion d'un réel en entier, la partie fractionnaire est tronquée

```
int    a = 2.55;   // a =  2
int    b = -2.55;     // b = -2
double c = 2.55;
int    d = c + 0.5;   // arrondi => 3
```

- Si la valeur n'est pas représentable dans le type entier, le résultat est indéfini

```
unsigned char a = 300;    // a = 44
unsigned char b = 300.;      // b indéterminé
                  // b = 0 (Apple LLVM 8.1)
unsigned char c = int(300.); // c = 44
```

Conversion réel ⇒ entier

- La librairie <`cmath`> fournit les fonctions d'arrondi
  - `trunc` l'entier en tronquant après la virgule
  - `round` l'entier le plus proche
  - `floor` l'entier plus petit ou égal
  - `ceil` l'entier plus grand ou égal

```
value trunc round floor ceil
----- ----- ----- ----- -----
 2.3   2.0   2.0   2.0   3.0
 3.8   3.0   4.0   3.0   4.0
 5.5   5.0   6.0   5.0   6.0
-2.3  -2.0  -2.0  -3.0  -2.0
-3.8  -3.0  -4.0  -4.0  -3.0
-5.5  -5.0  -6.0  -6.0  -5.0
```

Attention aux erreurs d'arrondi

- Qu'affiche ce code ?

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

int main() {
    const double PRIX = 4.35;
    int centimes = 100 * PRIX;
    cout << centimes << endl;
    return EXIT_SUCCESS;
}
```

Attention aux erreurs d'arrondi

- Dans votre ordinateur, les nombres sont stockés en format binaire, pas décimal
- En binaire, il n'y a pas de représentation exacte de `4.35`, tout comme il n'y a pas de représentation exacte de `1/3` en décimal.
- La représentation utilisée par l'ordinateur est un tout petit peu en-dessous de la valeur `4.35`
- Multiplié par `100`, on obtient donc une valeur en double juste inférieure à `435`, en l'occurrence `434.9999999999994316`
- La conversion en `int` par troncature donne `434`

8. Conversions implicites

Conversion implicite

- Il est possible d'écrire des expressions arithmétiques en mélangeant les types.

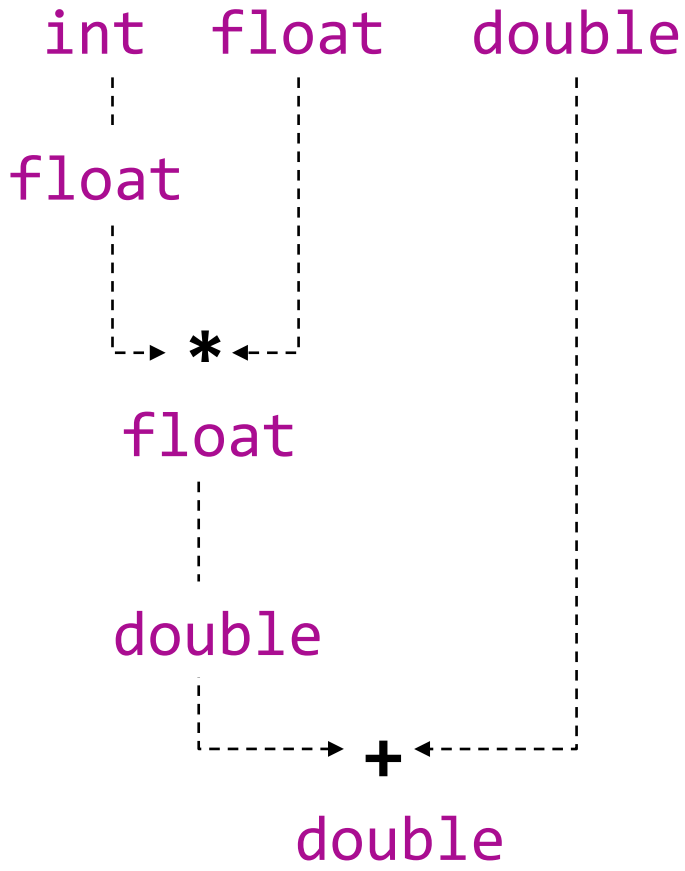
  Par exemple

```
x = 5 * 3.14F + 5.3e-2;
```

- Pourtant, les opérateurs arithmétiques
  - Ne sont définis que pour deux opérandes de même type
  - Ne sont pas définis pour les types entiers plus courts que `int` (`char` et `short`)

Exemple d'ajustement de type

```
x = 5 * 3.14F + 5.3e-2;
```



Conversion arithmétique

- L'évaluation de l'expression se fait selon l'ordre défini par les priorités des opérateurs
  1. Unaire (+,-)
  2. Multiplicatif (*,/,%)
  3. Additif (+,-)
- Pour les opérateurs unaires, on applique si nécessaire la promotion numérique entière
  - Les expressions `+a` et `-a` sont de type `int` si `a` est de type `char` ou `short`, signé ou pas

```
unsigned char  a = 1;
unsigned short b = 1;
unsigned int   c = 1;
cout << -a << " " << -b << " " << -c;
```

```
-1 -1 4294967295
```

Conversion arithmétique

- Pour les opérateurs binaires
  - On applique la promotion numérique entière aux types `char` ou `short`, signés ou pas
  - Si un opérande est de type `long double`, on convertit l'autre en `long double`
  - Sinon, si un opérande est de type `double`, on convertit l'autre en `double`
  - Sinon, si un opérande est de type `float`, on convertit l'autre en `float`
  - Sinon, la conversion est entre deux types entiers. Elle dépend des signes (`signed` → `unsigned`) et des rangs (`int` → `long` → `long long`)

Conversion arithmétique entre entiers

- S'ils sont de même signe, l'opérande de plus petit rang est converti dans le type de plus grand rang

```
int, long                          -> long
int, long long                     -> long long
long, long long                    -> long long
unsigned int, unsigned long        -> unsigned long
unsigned int, unsigned long long   -> unsigned long long
unsigned long, unsigned long long  -> unsigned long long
```

- Sinon, si l'opérande non signé est de rang supérieur ou égal à l'opérande signé, l'opérande signé est converti

```
int, unsigned int                -> unsigned int
int, unsigned long               -> unsigned long
int, unsigned long long          -> unsigned long long
long, unsigned long              -> unsigned long
long, unsigned long long         -> unsigned long long
long long, unsigned long long    -> unsigned long long
```

Conversion arithmétique entre entiers

- Sinon, si le type de l'opérande signé est de rang strictement supérieur à celui du type de l'opérande non signé, soient les 3 paires de types suivantes

```
unsigned int , long
unsigned int , long long
unsigned long, long long
```

- La conversion dépend alors du modèle de donnée, i.e. du nombre de bits utilisés pour représenter chaque type
  - Si l'opérande signé peut représenter toutes les valeurs de l'opérande non signé, l'opérande non signé est converti dans le type signé.
  - Sinon, les deux opérandes sont convertis dans la version non signée du type signé

Conversion arithmétique entre entiers

- En pratique, si `int` utilise 32 bits, `long` 32 et `long long` 64, (Windows)

```
unsigned int , long      -> unsigned long
unsigned int , long long -> long long
unsigned long, long long -> long long
```

- Mais, si `int` utilise 32 bits, `long` 64 et `long long` 64 (Linux ou Mac OS X)

```
unsigned int , long      -> long
unsigned int , long long -> long long
unsigned long, long long -> unsigned long long
```

Conversion arithmétique entre entiers

- Et donc, le code suivant affiche

```
unsigned int  a = 1;
long          b = 2;
unsigned long c = 1;
long long     d = 2;
cout << a - b << " " << c - d << endl;
```

- **-1 18446744073709551615** sous Mac OS X

- **4294967295 -1** sous Windows

Attention !

- Les conversions implicites peuvent entrainer dépassement ou perte de précision. On peut demander au compilateur de nous avertir de ces conversions implicites dangereuses avec l'option `-Wconversion`

```
int unEntier = 3.14;
```

```
warning: conversion to
'int' alters 'double'
constant value
```

- Mélanger des types signés et non signés peut donner des comportements inattendus. On peut demander au compilateur de nous en avertir avec l'option `-Wsign-conversion`

```
int a = 4;
unsigned int b = 5;
cout << a - b; // affiche 4294967295
```

```
warning: conversion to
'unsigned int' from 'int'
may change the sign of the
result
```

9. Saisie et affichage

Entrée et sortie

- Parfois, l'utilisateur doit renseigner la valeur d'une variable
- Le programme doit donc recevoir une entrée (input) de l'utilisateur
  - Avertir l'utilisateur de l'entrée requise avec `cout <<`
  - Lire la réponse de l'utilisateur au clavier avec `cin >>`
- Ces instructions d'entrée / sortie (input / output) sont disponibles via la librairie `<iostream>` que l'on inclut avec `#include <iostream>`

Saisie d'une entrée

- Pour lire une variable de l'entrée standard, on écrit

```
cout << "Entrez le nombre de bouteilles : ";
int nbBouteilles;
cin >> nbBouteilles;
```

Il est indispensable que la variable soit déclarée préalablement.
Il ne sert à rien de l'initialiser, sa valeur étant écrasée par le `cin >>`

- On peut lire plusieurs variables en une fois

```
cout << "Combien de bouteilles et canettes ? ";
int nbBouteilles, nbCanettes;
cin >> nbBouteilles >> nbCanettes;
```

Saisie de plusieurs entrées

- L'utilisateur peut répondre sur une ligne, les entrées séparées par des blancs

```
Combien de bouteilles et canettes ? 2 6
```

- Ou sur plusieurs, les entrées séparées en pressant la touche « Entrée »

```
Combien de bouteilles et canettes ? 2
6
```

```
cout << "Combien de bouteilles et canettes ? ";
int nbBouteilles, nbCanettes;
cin >> nbBouteilles >> nbCanettes;
```

Sortie formatée

- Vous désirez afficher sur la sortie standard les prix de 3 achats, le prix total, et les TVA correspondantes

```
Prix HT    TVA
-------------
  10.24   0.79
 117.20   9.02
   6.99   0.54
-------------
 134.43  10.35
```

- Vous disposez de ces 3 prix en `double`, et de la TVA en `const double`

- Comment réaliser cet affichage ?

```
const double TVA = 0.077;

double prix1 = 10.2372;
double prix2 = 117.2;
double prix3 = 6.9923435;
```

Vous savez déjà comment

- Calculer la valeur de la TVA avec l'opérateur `*`
- Calculer le prix total avec l'opérateur `+`
- Calculer la TVA sur le total en les combinant avec des parenthèses
- Afficher le résultat avec `cout <<`

```
Prix HT    TVA
--------------
10.2372  0.788264
117.2  9.0244
6.99234  0.53841
--------------
134.43  10.3511
```

```
cout << "Prix HT    TVA" << endl;
cout << "--------------" << endl;
cout << prix1 << "  " << prix1 * TVA << endl;
cout << prix2 << "  " << prix2 * TVA << endl;
cout << prix3 << "  " << prix3 * TVA << endl;
cout << "--------------" << endl;
cout << prix1 + prix2 + prix3 << "  "
     << (prix1 + prix2 + prix3) * TVA << endl;
```

- Mais le résultat n'est pas satisfaisant

<iomanip>

- Comment améliorer l'affichage ?
  - Choisir le format d'affichage des nombres réels,
    par exemple pour qu'il affiche exactement 2 chiffres après la virgule
  - Contrôler le nombre de caractères utilisé pour l'affichage d'une variable
    par exemple pour pouvoir aligner les données selon des colonnes
- Ces fonctionnalités sont fournies par les librairies `<iomanip>` et `<ios>`
- `<ios>` est déjà inclus par `<iostream>`
- Ajouter cette ligne à votre code

```
#include <iomanip>
```

fixed, scientific

```
double a = 3.1415926534;
double b = 2006.0;
double c = 1.0e-10;

cout << "default:";
cout << endl << a << endl << b
     << endl << c << endl << endl;

cout << "scientific:" << scientific;
cout << endl << a << endl << b
     << endl << c << endl << endl;

cout << "fixed:" << fixed;
cout << endl << a << endl << b
     << endl << c << endl << endl;
```

```
default:
3.14159
2006
1e-10

scientific:
3.141593e+00
2.006000e+03
1.000000e-10

fixed:
3.141593
2006.000000
0.000000
```

setprecision()

- Définit le nombre de chiffres significatifs. Par défaut, 6.
- Pour avoir n chiffres significatifs, on ajoute la ligne suivante au début du programme précédent

```
cout << setprecision(n);
```

- Son effet dépend du format d'affichage
  - Par défaut au plus n chiffres en tout
  - `Fixed` exactement n chiffres après la virgule
  - `Scientific` exactement n chiffres après la virgule

Effet de setprecision()

```
cout << setprecision(6);
```

```
default:
3.14159
2006
1e-10

scientific:
3.141593e+00
2.006000e+03
1.000000e-10

fixed:
3.141593
2006.000000
0.000000
```

```
cout << setprecision(3);
```

```
default:
3.14
2.01e+03
1e-10

scientific:
3.142e+00
2.006e+03
1.000e-10

fixed:
3.142
2006.000
0.000
```

setw( )

- Définit le nombre de caractères utilisés pour l'affichage de l'élément suivant uniquement !!
- Ajoute des espaces en tête si nécessaire ou un autre caractère défini par `setfill()`

```
#include <iostream>      // cout, endl
#include <iomanip>       // setw
using namespace std;

int main() {
   cout << setw(10);
   cout << 77 << endl;
   return 0;
}
```

- Sortie:

```
        77
```

Revenons au problème original

```
const int w = 7;
cout << setprecision(2) << fixed;

cout << setw(w) << "Prix HT"
     << setw(w) << "TVA" << endl;
cout << "--------------" << endl;
cout << setw(w) << prix1
     << setw(w) << prix1 * TVA << endl;
cout << setw(w) << prix2
     << setw(w) << prix2 * TVA << endl;
cout << setw(w) << prix3
     << setw(w) << prix3 * TVA << endl;
cout << "--------------" << endl;
cout << setw(w) << prix1 + prix2 + prix3
     << setw(w) << (prix1 + prix2 + prix3) * TVA
     << endl;
```

```
Prix HT    TVA
--------------
  10.24   0.79
 117.20   9.02
   6.99   0.54
--------------
 134.43  10.35
```

10. Caractères et chaînes de caractères (notions de base)

Le type char

- Le type `char` permet de stocker des entiers, signés ou pas, stockés sur 8 bits
- Il permet surtout de stocker des caractères via un codage qui fait correspondre caractères et valeurs numériques
- Le code ASCII original utilise 7 bits
  - de 0x00 à 0x1F et 0x7F pour les codes de contrôle
  - de 0x20 à 0x7E pour les caractères imprimables
  - des extensions au code ASCII utilisent 0x80 à 0xFF pour coder les caractères spéciaux

Code ASCII

- Le code ASCII 8 bits comporte deux parties :
  - Une partie fixe (caract. 0 à 127) : identique partout dans le monde
  - Une partie variable (caract. 128 à 255) : dépend de la région considérée
- La table ci-contre correspond au code **ASCII ISO Latin-1** (Europe de l'Ouest)
- Plus de détails sur : https://fr.wikipedia.org/wiki/American_Standard_Code_for_Information_Interchange

v·d·m	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1x	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2x	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4x	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5x	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6x	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7x	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL
8x	PAD	HOP	BPH	NBH	IND	NEL	SSA	ESA	HTS	HTJ	VTS	PLD	PLU	RI	SS2	SS3
9x	DCS	PU1	PU2	STS	CCH	MW	SPA	EPA	SOS	SGCI	SCI	CSI	ST	OSC	PM	APC
Ax	NBSP	¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬	- SHY	®	¯
Bx	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
Cx	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
Dx	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
Ex	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
Fx	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

Le type `char`

- Le type `char` peut être précédé d'un modificateur
  - `signed char` pour des entiers entre -128 et 127
  - `unsigned char` pour des entiers entre 0 et 255
  - Sans modificateur, le type peut être signé ou pas, cela dépend du compilateur
  - Le modificateur de signe n'affecte pas le caractère codé par les 8 bits du `char`
- Les constantes littérales de type `char` s'écrivent entourées d'apostrophes
  - `'a'` pour le `a` minuscule
  - `'3'` pour le caractère `3`

Affichage

- Le type détermine si on affiche le caractère ou la valeur numérique correspondante
- Attention aux promotions de type !

```
cout << 'A' << " " << (int)'A'     << endl;

cout << (char)0x41 << " " << 0x41 << endl;

cout << 'D' + 'a' - 'A' << " "
     << char('D' + 'a' - 'A')      << endl;
```

```
A 65
A 65
100 d
```

string - initialisation

- On peut initialiser les variables de type `string` des trois manières habituelles

```
string hello = "Hello, World!";
string hello("Hello, World!");
string hello{"Hello, World!"};
```

- Contrairement aux types simples, une variable non initialisée n'est pas de valeur indéfinie

```
string hello; // contient la chaine vide ""
```

Affectation – opérateur =

- L'opérateur = permet d'affecter une nouvelle valeur à une `string`
- Il convertit implicitement des expressions de type `char` ou des chaînes, y compris les constantes littérales

```
string str1, str2, str3;
str1 = "Test string: "; // chaine C littérale
str2 = 'x';             // caractère
str3 = str1;            // string
```

Concaténation – l'opérateur +

- L'opérateur + permet de concaténer deux chaines

```
string hello("Hello, ");
string world("World!");
string hw1 = hello + world;
  // hw1 contient "Hello, World!"
```

- il peut être utilisé avec une constante littérale

```
string hw2 = "Hello, " + world;
string hw3 = hello + "World!";
  // hw2 et hw3 contiennent "Hello, World!"
```

- Mais pas avec deux

```
string hw4 = "Hello, " + "World!";
  // erreur de compilation
```

Concaténation – l'opérateur +

- Il peut aussi être utilisé pour raccrocher un caractère en début ou en fin de chaine

```
string hello("Hello, ");
string hw5 = hello + 'W';    // hw5 contient "Hello, W"
string hw6 = 'W' + hello;    // hw6 contient "WHello, "
```

- Par contre, il n'est pas possible de concaténer une `string` avec un entier

Concaténation – l'opérateur +=

- Comme pour les opérateurs sur les entiers et les réels, il y a un opérateur auto-affecté correspondant, qui accepte les `char`, les `string` et les chaines littérales

```
string str("Hello");

str += ',';
// a le même effet que str = str + ',';
// str contient "Hello,"

str += " World!";
// a le même effet que str = str + " World!";
// str contient maintenant "Hello, World!"
```

Accès aux caractères – opérateur [ ]

Pour une chaine `str` et un entier `i`, l'expression `str[i]` permet d'accéder en lecture comme en écriture au `i`ième caractère – en numérotant depuis 0.

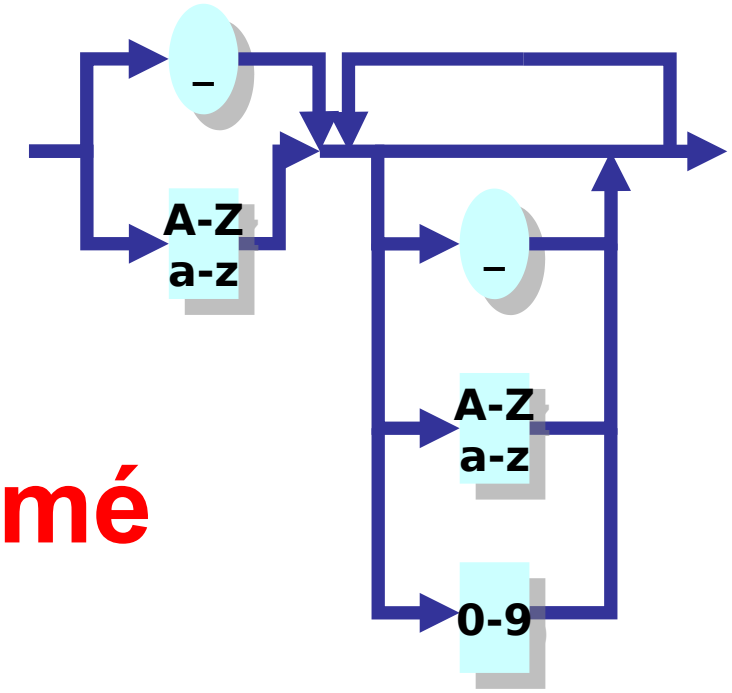
```
string hello("Hello, World!");
char fifth = hello[4];
hello[4] = ' ';

cout << hello << endl;
cout << fifth << " remplacé par un blanc \n”;
```

```
Hell , World!
o remplacé par un blanc
```

Au-delà…

- Le type `string` offre de très nombreuses autres possibilités via l'appel de « méthodes » de la classe

- Ce sera le sujet du chapitre 6



13. Résumé

```
Prix HT     TVA
--------------
  10.24    0.79
 117.20    9.02
   6.99    0.54
--------------
 134.43   10.35
```

Résumé

- Les variables permettent de stocker des données.
  Elles sont caractérisées par
  - Un type, qui définit comment elles sont stockées et quelles opérations ont peut leur appliquer
  - Un identificateur unique, leur nom

- Les identificateurs
  - Commencent par une lettre ou un « _ »
  - Continuent par des lettres, chiffres ou « _ »
  - Sont sensibles à la casse
  - Ne peuvent être un mot réservé du langage

Résumé

- On peut donner une valeur à une variable
  - En l'initialisant
  - Avec l'opérateur d'affectation
  - Avec `cin >>`
- Si une valeur ne change jamais au cours du programme, on peut définir la variable comme étant constante
  - Les constantes permettent d'éviter le problème des nombres magiques
- On documente le code
  - En choisissant bien les identificateurs
  - En y ajoutant des commentaires

Résumé

- Pour les nombres entiers
  - On utilise typiquement le type `int`
  - Il existe d'autres types plus ou moins longs (de 8 à 64 bits), signés ou pas
  - Il faut faire attention à l'intervalle des valeurs représentables et éviter d'en déborder
- Pour les nombres réels
  - On utilise typiquement le type `double`
  - Il existe d'autres types (`float`, `long double`) plus ou moins longs, et donc plus ou moins précis
  - Il faut faire attention à l'intervalle des valeurs, mais aussi à la précision

Résumé

- Une expression est un bout de code C++ qui retourne une valeur. Cela inclut
  - Les constantes littérales
  - Les opérateurs d'affectation, arithmétiques, d'incrémentation, …
- Pour calculer, on dispose
  - Des opérateurs `+`, `-`, `*`, `/` sur les réels
  - Des opérateurs `+`, `-`, `*`, `/`, `%` sur les entiers.
  - Des fonctions définies par la librairie <cmath> pour les opérations plus complexes (puissances, racines, trigonométrie, arrondis, …)
  - Attention à ne pas confondre la division entière de celle sur les réels, qui utilisent le même symbole `/`

Résumé

- On peut convertir les valeurs d'un type à l'autre.
  - Explicitement ou implicitement
  - Attention à rester dans l'intervalle des valeurs représentables
  - Attention à la perte éventuelle de précision
  - Attention aux arrondis
- On peut lire et écrire le contenu de variables avec `cin >>` et `cout <<`
  - Pour les réels, on peut contrôler le type d'affichage et sa précision
  - Pour tous les types, on peut contrôler la largeur de champs de l'affichage

Résumé

- Les types `char` et `string` permettent de stocker des caractères seuls ou du texte.
  - `char` se comporte comme un type entier pour tous les opérateurs autres que ceux d'affichage `>>` et `<<`
  - `string` possède les opérateurs d'affectation (`=`), de concaténation (`+` et `+=`) et d'accès à une lettre via sa position (`[]`)