Chapitre 2
Eléments de base

Structure du chapitre

- Variables
  - Définition, initialisation
  - Identificateurs
  - Constantes
- Types de base
  - `int, double, bool, char, string`
- Opérateurs
- Références et Pointeurs
- Priorité des opérateurs
- Commentaires

1. Variables

Variables

- Les variables servent à stocker des valeurs.
- Une variable doit être déclarée avant son utilisation.
- Pour déclarer une variable, il faut
  - le type de donnée stocké (nombre entier, nombre réel, caractère, …)
  - un nom (identificateur)
- Bonne pratique : initialiser les variables lors de la déclaration (création).
  - … sans quoi son contenu est indéterminé

```
int mon_entier;
```

Déclaration, contenu indéterminé ☹

```
int mon_entier = 6;
```

Déclaration + initialisation ☺

Exemple de déclaration des variables

Nom

```
int mon_entier = 6;
```

Type (un entier)

Contenu

```
int nb_mois = 12;
```

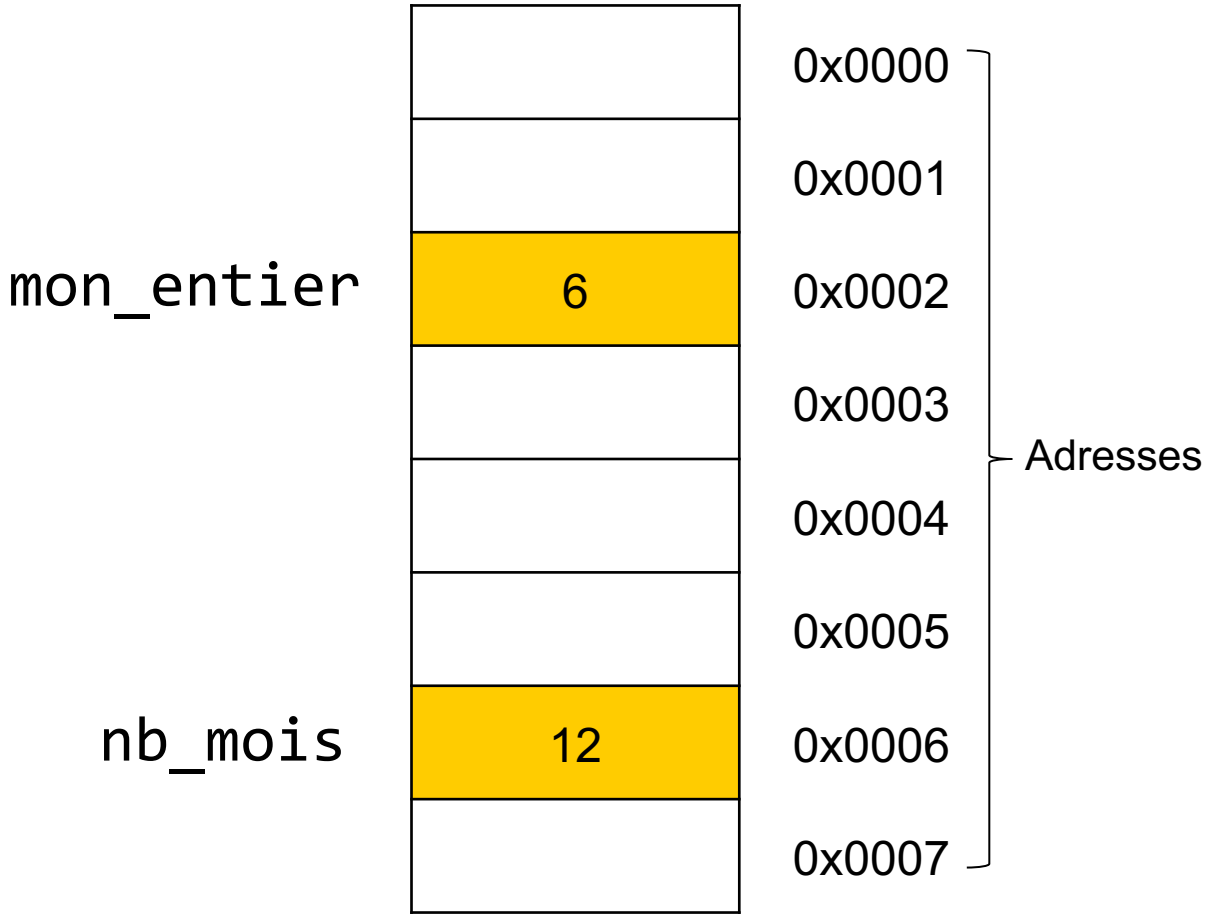


Illustration de la mémoire

Mais pourquoi utiliser des variables

- Faciliter la modification, la lecture et la réutilisabilité du programme.
- Stocker une entrée utilisateur, un résultat d'une opération, etc.

```
1 cout << "Nb bouteilles dans un pack : " << 6;
2 cout << "Volume d'un pack(l) = " << 0.33 * 6;
3 cout << "Poids d'un pack(g)  = " << 13   * 6;
4 cout << "Entrer le Nb de pack à expédier :";
5 cout << "Le poids de votre colis(g) :" << 13 * 6 * nb_pack;
```

Dans le programme ci-dessus :

- Modifier le nombre (`6`) de bouteilles par pack change les lignes 1,2,3, et 5
- Le calcul du poids d'un pack (`13.2 * 6`) est répété aux lignes 3 et 5.
- Il faut une variable `nb_pack` pour stocker l'entrée utilisateur, ligne 4.

Mais pourquoi utiliser des variables

- En utilisant des variables, l'exemple précédent devient

```
int nb_bouteilles = 6;
cout << "Nb bouteilles dans un pack : " << nb_bouteilles;
int vol_bouteille = 33;
cout << "Volume d'un pack(cl) = " << vol_bouteille * nb_bouteilles;
int poids_bouteille = 13;
int poids_pack = poids_bouteille * nb_bouteilles;
cout << "Poids d'un pack(g) = " << poids_pack;
cout << "Entrer le Nb de pack à expédier :";
int nb_pack;
cin >> nb_pack;
cout << "Le poids de votre colis(g) :" << poids_pack * nb_pack;
```

- Plus simple à modifier et surtout bien plus lisible ..

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 ?
  - elle doit être déclarée avant son utilisation
  - Bonne pratique : en C++, on recommande de déclarer les variables aussi près que possible de l'endroit où elles sont utilisées.

Noms de variables

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

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

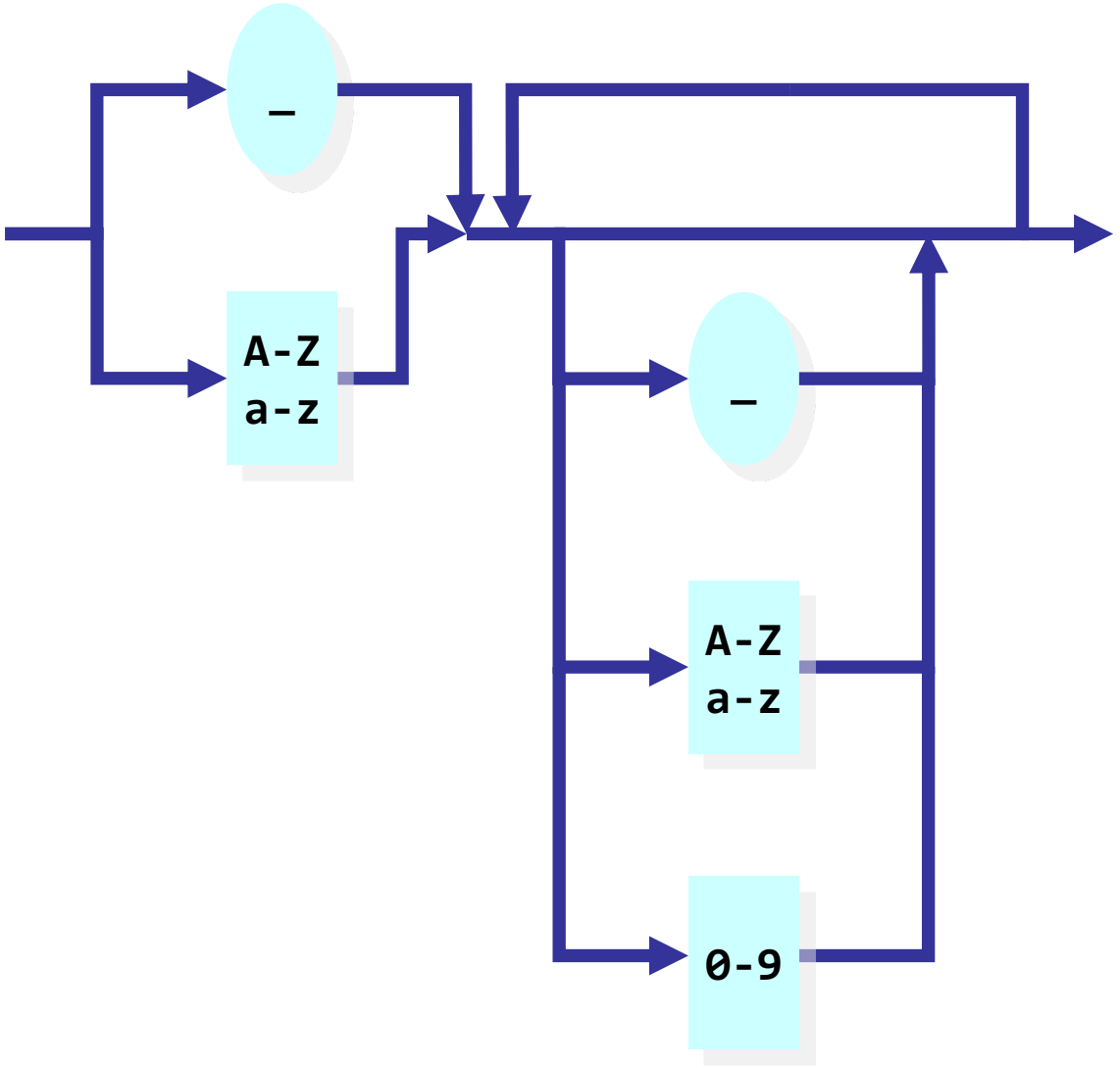
sera plus compréhensible avec des identificateurs parlants :

```
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
- ne peut pas contenir d'espaces ou de caractères spéciaux.
- est sensible à la casse
  (majuscules et minuscules sont différentes)
- ne peut être un des mots réservés C++



Cpp core guidelines - identificateurs

- NL5 : Le nom d'une variable ne doit pas mentionner son type (i.e. n'utilisez pas la notation hongroise)
- NL7 : La longueur d'un nom de variable doit être ± proportionnelle à sa portée (distance entre ses utilisations)
- NL8 : Utilisez un style de nommage consistant : `snake_case`, `camelCase`, `PascalCase`, …
- NL9 : N'utilisez pas TOUT_EN_MAJUSCULE pour les identificateurs autres que les macros (vues en PRG2)
- NL10 : Préférez le style `snake_case`, c'est le style utilisé par la Standard Template Library. Utilisez éventuellement `Majuscule_initiale` pour les types que vous définissez vous-même, comme Bjarne Stroustrup.

Constantes

- Certaines variables ne doivent pas changer de valeurs après leurs initialisations dans le programme. On dit qu'elles sont immutables.
- Pour celles-ci, le C++ met à disposition le mot réservé `const` à placer juste avant la déclaration de la variable

```
const int vol_bouteille = 33;
```

- Son initialisation à la déclaration est obligatoire

Cpp core guideline CON.1 : Par défaut, définissez toutes vos variables `const`. Les variables mutables (non `const`) devraient être l'exception

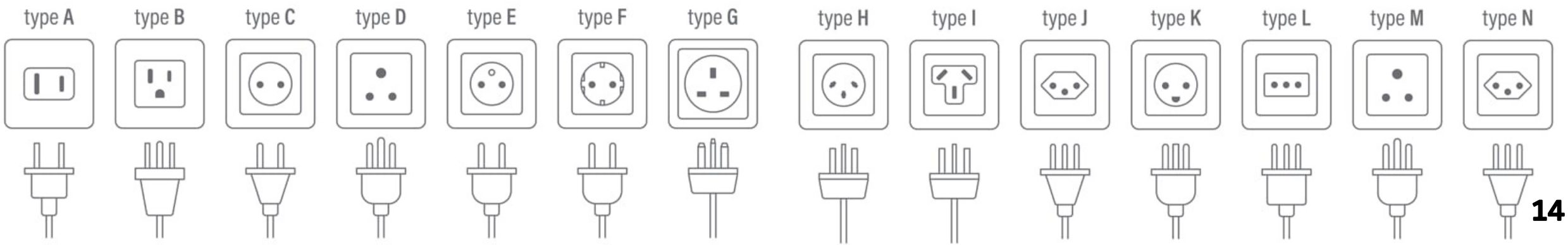
Pourquoi utiliser des constantes ?

- Reprenons notre exemple de pack de bouteilles. Le code est encore plus facile à comprendre et à maintenir en cas de changement

```
const int nb_bouteilles   =  6;
const int vol_bouteille   = 33;
const int poids_bouteille = 13;
cout << "volume d'un pack(cl)       = " << vol_bouteille * nb_bouteilles;
cout << "Nb bouteilles dans un pack = " << nb_bouteilles;
const int poids_pack = poids_bouteille * nb_bouteilles;
cout << "poids d'un pack(g)         = " << poids_pack;
cout << "entrer le nb de pack à expédier :";
int nb_pack;  // seule variable mutable
cin >> nb_pack;
cout << "le poids de votre colis(g) :" << poids_pack * nb_pack;
```

- On évite aussi l'apparition d'un « magic number » inexpliqué au milieu du code

2. Types de base



Les types de base

- Les données en C++ sont typées
- Un type définit
  - comment cette donnée est stockée en mémoire (nombre de bits et codage)
  - les opérations possibles avec cette donnée

- Les types fondamentaux (ou de base) fournis par le langage permettent de stocker des données simples
  - caractères typographiques
  - nombres entiers
  - nombres réels
  - booléens

Les types de base

```
int nb_packs = 10;
```

- Stockent une valeur entière comme `-7` ou `1024`
- Il peut être qualifié par la taille utilisée pour le stocker et le fait d'être signé ou pas *(voir chapitre 6)*

```
float volume_canette = 0.33f;
double pi = 3.141592;
```

- Stockent des nombres réels tels que `3.14` ou `0.01`
- `Le type double permet plus de précision que float.`

Les types de base

```
char c = 'a';
```

- Le type `char` permet de stocker un caractère.
- En réalité il stocke un entier, sa taille est de 1 byte.
- Le code ASCII fait correspondre caractères et valeurs numériques.
- Pour définir une valeur littérale de type `char` on utilise les simples guillemets.

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

```
char c1 = 'a'; // 97
char c2 = 'Z'; // 90
char c2 = '5'; // 53

char c1 = 97; // 'a'
char c2 = 90; // 'Z'
char c2 = 53; // '5'
```

0	NUL (null)	32	(space)	64	@	96	`
1	SOH (start of header)	33	!	65	A	97	a
2	STX (start of text)	34	”	66	B	98	b
3	ETX (end of text)	35	#	67	C	99	c
4	EOT (end of transmission)	36	$	68	D	100	d
5	ENQ (enquiry)	37	%	69	E	101	e
6	ACK (acknowledge)	38	&	70	F	102	f
7	BEL (bell)	39	’	71	G	103	g
8	BS (backspace)	40	(	72	H	104	h
9	HT (horizontal tab)	41	)	73	I	105	i
10	LF (line feed/new line)	42	*	74	J	106	j
11	VT (vertical tab)	43	+	75	K	107	k
12	FF (form feed / new page)	44	,	76	L	108	l
13	CR (carriage return)	45	-	77	M	109	m
14	SO (shift out)	46	.	78	N	110	n
15	SI (shift in)	47	/	79	O	111	o
16	DLE (data link escape)	48	0	80	P	112	p
17	DC1 (data control 1)	49	1	81	Q	113	q
18	DC2 (data control 2)	50	2	82	R	114	r
19	DC3 (data control 3)	51	3	83	S	115	s
20	DC4 (data control 4)	52	4	84	T	116	t
21	NAK (negative acknowledge)	53	5	85	U	117	u
22	SYN (synchronous idle)	54	6	86	V	118	v
23	ETB (end of transmission block)	55	7	87	W	119	w
24	CAN (cancel)	56	8	88	X	120	x
25	EM (end of medium)	57	9	89	Y	121	y
26	SUB (substitute)	58	:	90	Z	122	z
27	ESC (escape)	59	;	91	[	123	{
28	FS (file separator)	60	<	92	\	124	|
29	GS (group separator)	61	=	93	]	125	}
30	RS (record separator)	62	>	94	^	126	~
31	US (unit separator)	63	?	95	_	127	DEL (delete)

Les types de base

```
string nom    = "Bob";
string prenom = "Alice";
string titre  = "Programmation en c++";
string ville;
```

- Le type `string` permet de stocker une chaîne de caractères.
- Contrairement aux types simples une variable non initialisée est définie comme une chaîne vide.
- Pour définir une variable de type `string` on utilise les doubles guillemets.

Les types de base

```
bool vrai = true;
bool faux = false;
```

- Stockent une valeur booléenne : vraie ou fausse (`true` / `false`)
- C'est en fait un type numérique qui peut prendre les valeurs
  - 0 pour `false`
  - 1 pour `true`
- Toute valeur numérique non nulle correspond à `true`.

```
bool vrai = 42;
bool faux = 0;
```

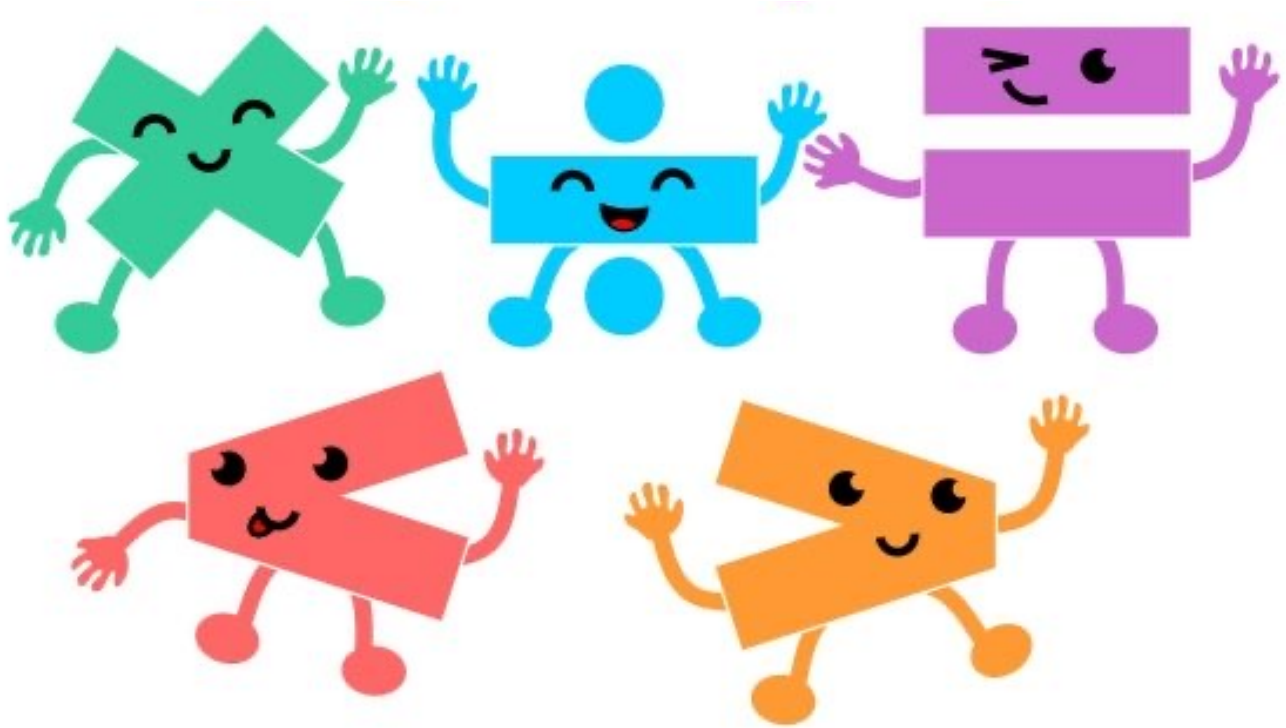
Les types de base

```
auto x = 1;      // int
auto d = 1.5;    // double
auto c = 'c’;    // char
auto b = true;   // bool
```

- Le mot clé `auto` indique au compilateur de déduire le type de la variable déclarée comme étant celui de son expression d’initialisation.

à utiliser lorsque le type n’est pas connu

3. Opérations



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
```

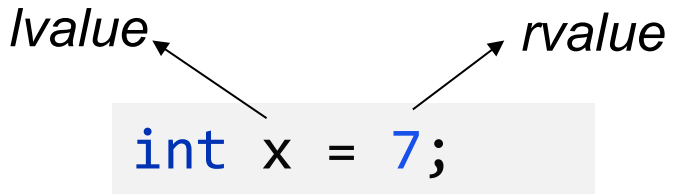
- Nous en avons déjà vu plusieurs sortes
  - Les littéraux constants
  - L'opérateur d'affectation

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 : `59`
  - un réel : `3.1416`

lvalue vs rvalue

- lvalue (valeur de localisation) :
  - est une expression qui fait référence à une adresse mémoire.
  - une lvalue a une adresse qui peut être accessible par le programme.
  - Exemple : les noms de variables
- rvalue (valeur de résultat) :
  - est une expression qui ne fait pas référence à une adresse mémoire (n'est pas un lvalue).
  - Exemple : les littéraux constants, les résultats d'expressions qui ne se résolvent pas en lvalues



Opérateur d'affectation =

- L'opérateur d'affectation = copie la valeur de l'expression à droite dans la variable à gauche
- Exemple: `age = 10;` affecte la valeur 10 à la variable age
- Elle remplace la valeur actuelle de la variable par une nouvelle valeur ce qui écrase la valeur qui s'y trouvait précédemment

- Seule une `lvalue` peut prendre place à gauche de l'opérateur. Il faut avoir `lvalue = rvalue;`

```
10 = age;
```

Opérateur d'affectation =

```
int mon_entier = 6;
mon_entier = 10;
```

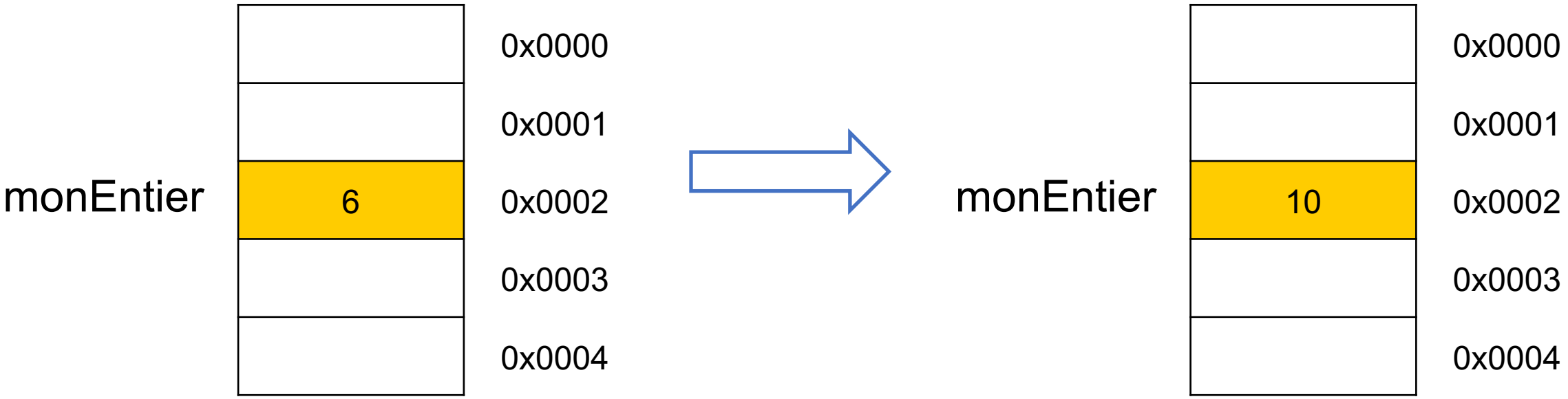
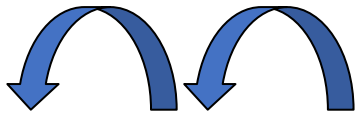


Illustration de la mémoire

Opérateur d'affectation =

- L'opérateur d'affectation est lui-même une expression qui renvoie la lvalue dans laquelle elle a écrit. Cela permet d'enchainer les affectations, i.e. d'écrire



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

- L'expression s'évalue de droite à gauche, i.e. d'abord `y = 42` , qui
  - affecte la valeur `42` à y
  - retourne y dont la valeur est `42`
- Puis il évalue `x = (y = 42)` , qui
  - affecte la valeur `42` à x
  - retourne x dont la valeur `42` n'est pas utilisée

Arithmétique sur les réels

- Les types réels (`float`, `double`, …) disposent des opérateurs `+`, `-`, `*`, et `/` qui se comportent comme en mathématiques

```
double somme = 1.0 + 3.0;          // 4.0

double difference = 7.0 - 3.0;     // 4.0

double produit = 3.14 * 2.0;       // 6.28

double quotient = 5.0 / 2.0;       // 2.5
```

Arithmétique sur les entiers

- Les types entiers (`int`, …) disposent des opérateurs `+`, `-`, `*`, `/`, et `%`
- Les opérateurs `+`, `-`, et `*` se comportent comme pour les réels
- L'opérateur `/` effectue une division entière et donne donc un résultat entier
- L'opérateur `%` calcule le reste d'une division entière, appelé modulo

```
int somme = 1 + 3;
// 4

int difference = 7 - 3;
// 4

int produit = 3 * 2;
// 6

int quotient_entier = 5 / 2;
// 2, pas 2.5

int modulo = 8 % 3;
// 2, le reste de la division
// entière de 8 par 3
```

Division entière et modulo

- Pour a et b entiers, on a toujours l'égalité suivante :

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

- Pour a et/ou b 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 centimes en francs ?

```
int total = 1729; // centimes
int francs = total / 100;
int centimes = total % 100;

cout << total << " centimes valent " << francs
     << " francs et " << centimes << " centimes";
```

```
1729 centimes valent 17 francs et 29 centimes
```

Opérateurs d'affectation composée

- Il est courant de modifier une variable pour lui ajouter ou soustraire une valeur, la multiplier par une valeur, etc.
- Les opérateurs `+=`, `-=`, `*=`, `/=`, et `%=` permettent de combiner opération arithmétique et affectation

Expression	Affectation composée équivalente
`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;`

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
```

Incrémentation pré et postfixe

- La valeur de retour des opérateurs d'incrémentation unaire ++ (ou --) varie selon leur position :
  - préfixe - avant la variable à incrémenter – ils retournent la variable incrémentée, et donc sa valeur après incrémentation
  - postfixe - après la variable à incrémenter – ils retournent une copie de 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
```

Opérateurs unaires -, +

- 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 :
  - `-3 + 4` vaut 1
  - `-a + 4` vaut `-2` si a vaut 6

Opérateurs de comparaison

- C++ fournit 6 opérateurs de comparaison

`<`	plus petit que
`>`	plus grand que
`<=`	plus petit ou égal
`>=`	plus grand ou égal
`==`	égal (ne pas confondre avec l'affectation =)
`!=`	différent de

- Ils permettent de comparer des
  - entiers
  - réels
  - chaînes de caractères (string)
- Ils renvoient des valeurs de type `bool`

Opérateurs de comparaison (exemple)

- Tests d'égalité

```
cout << (3 == 5) << (3 == 3);
cout << (3 != 5) << (3 != 3);
```

```
01
10
```

- Inégalités strictes

```
cout << (3 < 5)  << (3 < 3)  << (5 < 3);
cout << (3 > 5)  << (3 > 3)  << (5 > 3);
```

```
100
001
```

- Inégalités larges

```
cout << (3 <= 5) << (3 <= 3) << (5 <= 3);
cout << (3 >= 5) << (3 >= 3) << (5 >= 3);
```

```
110
011
```

Comparer des chaines de caractères

- La comparaison de chaines (string) fonctionne comme l'ordre lexicographique, i.e. celui d'un dictionnaire
  - On compare le premier caractère des 2 chaines
  - S'il est identique, on compare le second
  - S'il est identique, on compare le troisième
  - …
  - Jusqu'à ce qu'un caractère diffère ou qu'on arrive au bout d'au moins une des chaines.
- Une fois le premier caractère différent atteint, on compare leurs codes ASCII, et donc
  - `' ' < '0' < '9' < 'A' < 'Z' < 'a' < 'z'`
- Si au contraire on atteint la fin d'une des chaines, la plus courte est plus petite que la plus longue

Opérateur de comparaison trilatérale <=>

- Autrement appelé l'opérateur spaceship (vaisseau spatial).
- A <=> B : détermine si A < B, A == B ou A > B en une seule opération. Il retourne une valeur qui n'est comparable qu'avec le littéral zéro.

A < B → (A <=> B) < 0

A > B → (A <=> B) > 0

A == B → (A <=> B) == 0

```
const int x = 5, y = 3;

cout << ((x <=> y) < 0);      // affiche 0 (false) car x > y
cout << ((x <=> y) > 0);      // affiche 1 (true) car x > y
cout << ((x <=> y) == 0);     // affiche 0 (false) car x != y
```

Opérateurs logiques

- C++ dispose des opérateurs logiques suivants
  - not
  - or
  - and

not « ! »	
0	1
1	0

or « || »		
0	0	0
0	1	1
1	0	1
1	1	1

and « && »		
0	0	0
0	1	0
1	0	0
1	1	1

```
cout << ( (5 == 5) &&  (3 > 6) ) << endl;  // 0 (false)
cout << ( (5 == 5) ||  (3 > 6) ) << endl;  // 1 (true)
cout << ( (5 == 5) and (3 < 6) ) << endl;  // 1 (true)
cout << ( (5 != 5) or  (3 > 6) ) << endl;  // 0 (false)
cout << !(5 == 5)    << endl;              // 0 (false)
cout << not(5 != 5) << endl;               // 1 (true)
```

Evaluation court-circuit

- L'évaluation des expressions `and` (`&&`) et `or` (`||`) se fait de gauche à droite et s'arrête dès que possible

  `(a && b)` évalue d'abord `a`, n'évalue pas `b` si `a` est faux

  `(a || b)` évalue d'abord `a`, n'évalue pas `b` si `a` est vrai

- Cela permet d'écrire sans risque de division par zéro une expression telle que

```
((n != 0) and (p < m / n))
```

- Mais attention aux effets de bord de la non évaluation du terme de droite.

```
(i or ++j)  // j pas incrémenté si i est true
```

Lois de De Morgan

- La négation de la conjonction de deux propositions est équivalente à la disjonction des négations des deux propositions

`!(A and B)` est équivalent à `(!A or !B)`

- La négation de la disjonction de deux propositions est équivalente à la conjonction des négations des deux propositions

`!(A or B)` est équivalent à `(!A and !B)`

Lois de De Morgan : Exemple

- Le code suivant calcule des frais d'expédition hors des USA continentaux

```
fraisExpedition = 10.0;
if (not (pays == "USA" and etat != "AK" and etat != "HI")) {
                                  // Alaska            Hawaï
   fraisExpedition = 20.0;
}
```

- On peut le réécrire plus simplement

```
fraisExpedition = 10.0;
if (pays != "USA" or etat == "AK" or etat == "HI") {
                             // Alaska          Hawaï
   fraisExpedition = 20.0;
}
```

Opérations sur le type char

- Comme le type char stock un entier correspondant au code ASCII du caractère, nous pouvons l'utiliser avec les opérations arithmétiques.

```
char x = 'B'; // 66
cout << x; // affiche B

x += 1;
cout << x; // affiche C

x += 32;
cout << x; // affiche c

x = x - 1;
cout << x; // affiche b
```

string - Concaténation – 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 pour deux chaine de caractères

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

`string` - 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

`string` - 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!"
```

`string` - 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`ème 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";
```

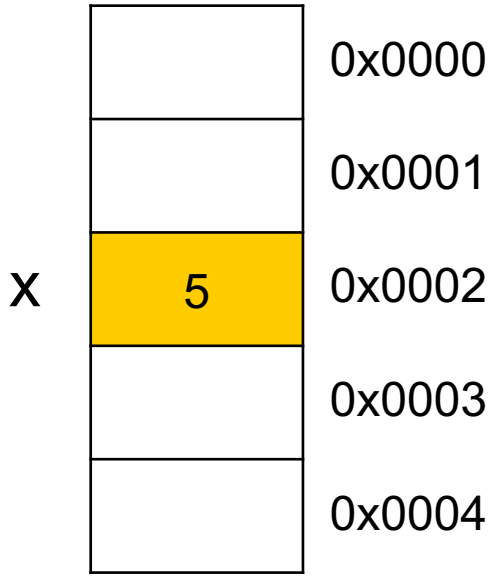
```
Hell , World!
o remplacé par un blanc
```

4. Pointeurs et références

Pointeurs

- Opérateur d'adresse & :
  - Permets de récupérer l'adresse d'une variable.

```
int x = 5;
cout << &x; // 0x0002
```



- Opérateur de déréférencement * :
  - Renvoie la valeur stockée à une adresse mémoire donnée sous forme de lvalue.

```
int x = 5;
cout << &x;     // 0x0002
cout << *(&x); // 5
*(&x) = 10;
cout << x;      // 10
```

Pointeurs

- Un pointeur est une variable qui contient une adresse mémoire (généralement celle d'une autre variable) comme valeur.
- Les types pointeurs sont déclarés en ajoutant un astérisque * après le type pointé
- On peut changer l'adresse stockée dans un pointeur. Ainsi on change la variable à laquelle il pointe.
- On peut changer la valeur stockée à l'adresse pointée en utilisant l'opérateur de déréférencement *

Pointeurs

```
int x = 5;
int* ptr = &x;

cout << x;      // 5
cout << ptr;    // 0x0002
cout << *ptr;   // 5
cout << &ptr;   // 0x0004

*ptr = 10;
cout << x;      // 10

int y = 15;
ptr = &y;

cout << y;      // 15
cout << ptr;    // 0x0001
cout << *ptr;   // 15
cout << &ptr;   // 0x0004
```

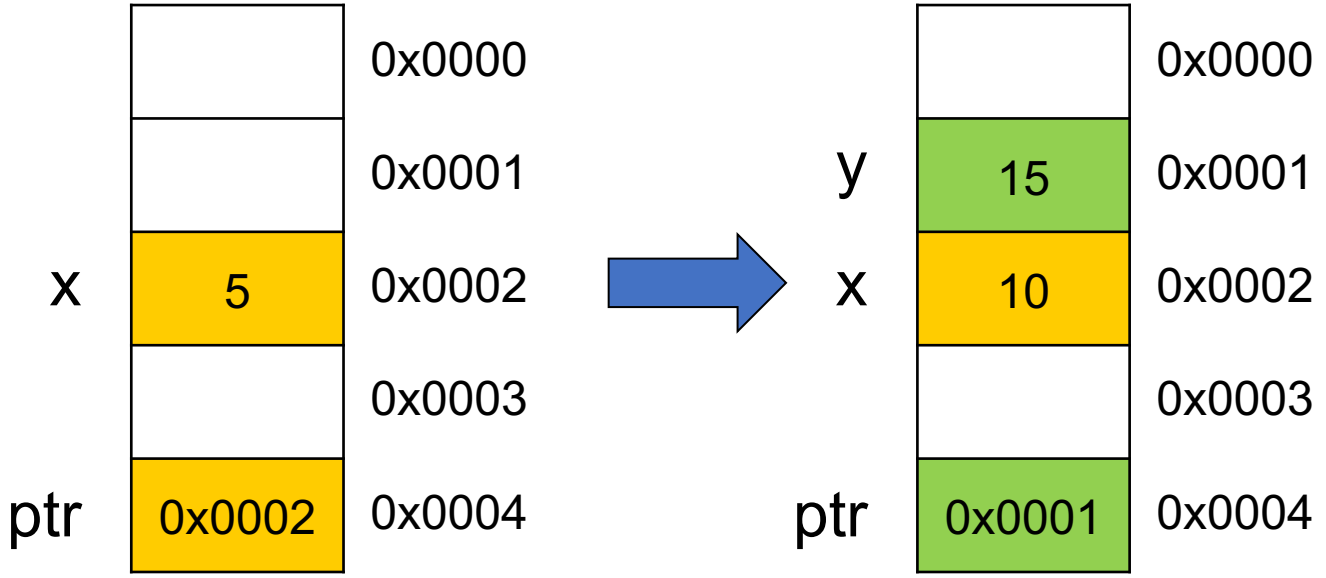


Illustration de la mémoire

Pointeurs avec const

- Avec des pointeurs, nous pouvons modifier :
  - ce vers quoi le pointeur pointe (en attribuant au pointeur une nouvelle adresse à conserver)
  - la valeur à l'adresse détenue (en attribuant une nouvelle valeur au pointeur déréférencé)
- Comme pour les références, pour pointer vers une lvalue non modifiable (const), nous avons besoin d'un pointeur vers const.

```
const int x = 5;   // x est une lvalue non modifiable (const)
int* ptr = &x;                    // erreur
const int* ptr_vers_const = &x; // ok
```

  - `ptr_vers_const` est un pointeur (non-const) vers une valeur const.
  - `ptr_vers_const` peut alors changer de contenu et pointer une autre valeur.

Pointeurs avec const

- Un pointeur vers const :
  - est nécessaire pour pointer vers un lvalue const (non modifiable)
  - peut pointer vers un lvalue modifiable (non const)
  - ne permet pas de modifier la valeur vers laquelle il pointe.

```
const int x = 5;
const int* ptr_vers_const = &x;
cout << *ptr_vers_const; // affiche 5

const int y = 10;
ptr_vers_const = &y;     // ok - ptr_vers_const est non const
cout << *ptr_vers_const; // affiche 10

int z = 15;
ptr_vers_const = &z;     // ok - ptr_vers_const  est non const
cout << *ptr_vers_const; // affiche 15
*ptr_vers_const = 20;    // erreur – ptr_vers_const  pointe const int
```

Pointeurs avec const

- Les pointeurs déclarés const :
  - Le pointeur lui-même peut être déclaré comme non modifiable (const)
  - Les pointeurs const ne peuvent pas changer l'adresse vers laquelle ils pointe.

```
int x = 5;
int y = 10;
int* const cptr = &x; // ok – const pointer vers non const lvalue
cptr = &y; // erreur – const pointer ne peut pas changer son contenu

*cptr = 7; // ok – x est modifiable
cout << *cptr << x; // affiche 77
```

Références

- L'utilisation de pointeurs constants vers une lvalue (e.g. vers une variable) est typique du langage C. En C++ on préfère la syntaxe des références.

```
int a = 0;
int *const ptr_a = &a;
*ptr_a = 42;
ptr_a = 0;
  // ne compile pas à cause du const
cout << a;
  // affiche 42
```

```
int a = 0;
int& ref_a = a;
ref_a = 42;
  // change la valeur de a,
  // pas la variable référencée
cout << a;
  // affiche 42
```

- Déclarer une référence utilise le symbole & après le type. Elle doit être initialisée. La référence se comporte comme un synonyme de la lvalue référencée. Il n'y a pas de copie effectuée

- Quelle lvalue est référencée ne peut être modifié. La valeur de cette variable peut l'être. On ne peut dès lors pas créer une référence vers une constante.

Référence constante

- L'utilisation de pointeurs constants vers une constante est typique du langage C. En C++ on préfère la syntaxe des références constantes.

```
int a = 42;
const int *const ptr_a = &a;
cout << *ptr_a;
  // affiche 42
*ptr_a = 0;
  // ne compile pas à cause du 1er const
ptr_a = 0;
  // ne compile pas à cause du 2ème const
```

```
int a = 42;
const int& cref_a = a;
cout << cref_a;
  // affiche 42
cref_a = 0;
  // ne compile pas
```

- Déclarer une référence constante utilise le symbole & après le type et le mot clé `const`. Elle doit être initialisée.

- On ne peut changer ni l'emplacement mémoire référencé ni la valeur qui y est stockée.

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 ou de droite à gauche dans l'expression selon leurs règles d'association.

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

Priorités relatives de ces opérateurs

Priorité	Opérateurs	Description	Associativité
1	`a++, a--, a[], …`	unaires postfixes	gauche à droite →
2	`++a, --a, +a, -a, !a, *a, &a, …`	unaires préfixes	droite à gauche ←
3	`a*b, a/b, a%b`	multiplicatifs	gauche à droite →
4	`a+b, a-b`	additifs	
5	`<<, >>`	Lecture et affichage	
6	`<=>`	spaceship	
7	`<, <=, >, >=`	comparaison	
8	`==, !=`	(in)égalité	
9	`&&, and`	ET logique	
10	`||, or`	OU logique	
11	`=, +=, -=, *=, /=, %=`	affectations	droite à gauche ←

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
`a = 1; 9 / 3 + ++a`	5
`(3.2 + 0.6) * -2.0`	-7.6
`5 == 5 && 2 > 5 || (3 <=> 3 == 0)`	true

6. Commentaires

```
// dernière section du chapitre
// il ne reste que deux slides
// accrochez-vous …
```

Commentaires

- Permettent d'inclure des explications en langage courant, données à la personne qui lit le code
- Ils sont ignorés par le compilateur

```
double volumeCanette = 0.33; // en Litre
```

-  Éviter de surcharge le code de commentaires redondantes et/ou d'explications inutiles. Le choix des noms de vos variables et fonctions et la structure de votre code est au moins aussi importante que vos commentaires pour sa compréhension

Commentaires – 2 styles

- après // et jusqu'à la fin de la ligne

```
// 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 /
```

- entre /* et */. Il peut commencer et s'arrêter en cours de ligne, ou en occuper plusieurs

```
/*  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 /* n'importe */ volume /* où */ = 2;
```