Les bases de l'Informatique

Thierry Vaira

BTS SN

v1.0 - 13 août 2017

Traiter des données I

- Un processeur ne sait que **traiter des données**.
  - ➠ Le traitement est assuré par l'exécution d'instructions simples codées en binaire.
  - ➠ Les données seront n'importe quelle information codée en binaire (une « valeur numérique »).
- On rassemble tout cela à l'intérieur d'un **programme** et finalement :
  - ➠ un programme ne fera que **traiter des données** ...
  - ➠ un ordinateur ne servira qu'à **traiter des données** ...
  - ➠ l'informatique ce n'est que ... **traiter des données** !

Notion de variable

- Pour manipuler des données, les langages de programmation nous offrent le concept de **variable**.
- Une variable est un **espace de stockage pour un résultat**.
- Une variable est associé à un **symbole** (habituellement un **nom** qui sert d'identifiant) qui renvoie à une **position de la mémoire** (une **adresse**) dont le contenu peut prendre successivement différentes valeurs pendant l'exécution d'un programme.
- De manière générale, les variables ont :
  - un **type** : c'est la convention d'interprétation de la séquence de bits qui constitue la variable. Le type de la variable spécifie aussi sa **taille** (la longueur de cette séquence) soit habituellement 8 bits, 32 bits, 64 bits, ... Cela implique qu'il y a un **domaine de valeurs** (ensemble des valeurs possibles).
  - une **valeur** : c'est la séquence de bits elle même. Cette séquence peut être codée de différentes façons suivant son type.

Bonnes pratiques

Rob Pike (ancien chercheur des Laboratoires Bell et maintenant ingénieur chez Google) :

> *« Règle n°4 : Les algorithmes élégants comportent plus d'erreurs que ceux qui sont plus simples, et ils sont plus difficiles à appliquer. Utilisez des algorithmes simples ainsi que des structures de données simples. »*

➠ Cette règle n°4 est une des instances de la philosophie de conception KISS (*Keep it Simple, Stupid* dans le sens de « Ne complique pas les choses »).

> *« Règle n°5 : Les données prévalent sur le code. Si vous avez conçu la structure des données appropriée et bien organisé le tout, les algorithmes viendront d'eux-mêmes. La structure des données est le coeur de la programmation, et non pas les algorithmes. »*

➠ Cette règle n°5 est souvent résumée par « Écrivez du code stupide qui utilise des données futées ! ».

Des données numériques

Le caractère '2' :

Caractère '2'

0	0	1	1	0	0	1	0

50 ou 0x32

Code ASCII

L'entier 2 :

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0

signe

... 2^3 2^2 2^1 2^0

Le réel 2,0 :

0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

signe **exposant** **mantisse**

La chaîne de caractères "2" :

Caractère '2'

0	0	1	1	0	0	1	0

50 ou 0x32

Code ASCII

Caractère NUL '\0'

0	0	0	0	0	0	0	0

0 ou 0x00

Fin de chaîne

Les types scalaires

- Une **variable scalaire** est destinée par son **type** à contenir une **valeur atomique**.
- Les valeurs atomiques sont :
  - les **booléens**
  - les **nombres entiers**
  - les **nombres à virgule flottante**

Remarque : les **caractères** sont en fait des nombres entiers ! En effet, on est obligé de convertir les caractères sous une forme binaire qui constitue un code. Par exemple, le code ASCII du caractère '`A`' est 65 (`0x41` en hexadécimal).

Les types composés

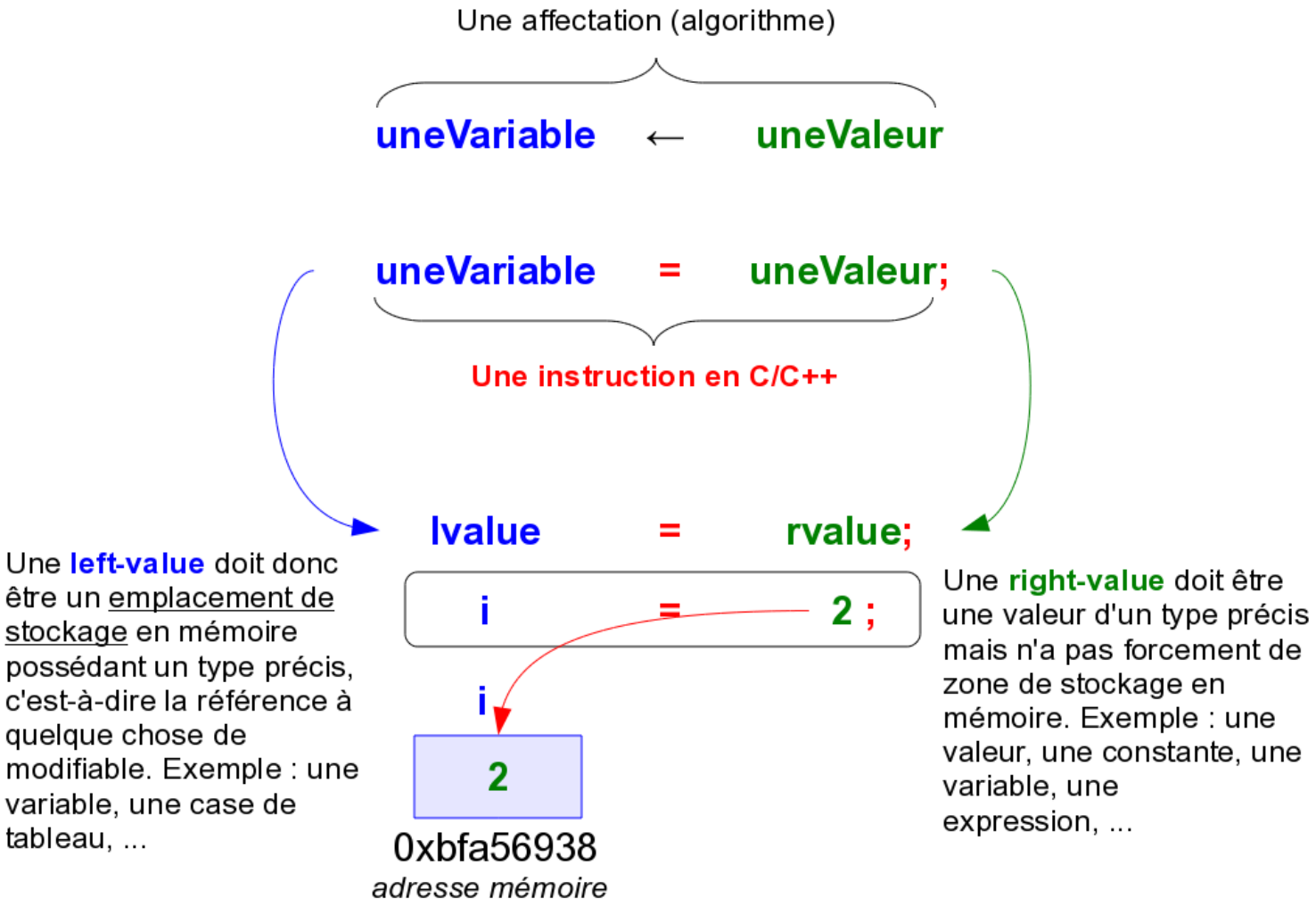
- Il est possible de regrouper des types scalaires différents dans des **structures de données**.
- Il est possible de regrouper des types scalaires identiques dans des **tableaux**.
- La **chaîne de caractères** peut être considérée comme un type composé.
- On verra ensuite des **collections** de données (arbre, liste, pile, file, ...).

Traiter des données II

- Pour traiter des données, il faut pouvoir les manipuler et faire des opérations sur celles-ci.
- Les langages de programmation nous offrent le concept d'**instructions** dans lesquelles on pourra utiliser des **opérateurs**.
- On rappelle que les instructions que l'ordinateur peut comprendre ne sont pas celles du langage humain. Le processeur sait juste exécuter un nombre limité d'instructions bien définies (son jeu d'instructions).
- Des instructions typiques comprises par un ordinateur sont par exemple :
  - copier le contenu de la case 123 et le placer dans la case 456
  - ajouter le contenu de la case 321 à celui de la case 654
  - placer le résultat dans la case 777
  - si le contenu de la case 999 vaut 0, mettre 0 dans la case 345

L'affectation

- La première des instructions est la possibilité d'**affecter une valeur à une variable**. La plupart des langages proposent l'**opérateur d'affectation =** :



Les variables dans les algorithmes

- Pour créer une variable, il faut la **déclarer** en lui donnant **un nom** (éloquent et précis) et **un type** :

```
// Déclaration d'une variable de type entier
Variable indice : Entier

// Déclaration d'une variable de type caractère
Variable lettre : Caractère

// Déclaration d'une constante de type réel
Constante PI : Réel = 3,14

// On affecte une valeur à une variable
indice <- 0
```

Les variables dans les langages de programmation

- Dans les langages compilés, les variables doivent être déclarées en précisant leur type. Il est conseillé de toujours les initialiser :

```
// En C/C++ :

// une variable entière
int i = 0;
// une variable réelle
float j = 2.5;
// interdit
i = 2.5; // 2.5 n'est pas de
     type entier
```

- Dans les langages interprétés, seules les valeurs ont un type ce qui n'oblige pas toujours à déclarer les variables :

```
// En PHP :

// une valeur entière
$i = 0;
// une valeur réelle
$j = 2.5;
// autorisée
$i = 2.5; // la variable $i
    n'a pas de type déclaré
```

Caractéristiques

- De manière générale, on distinguera les caractéristiques suivantes :
  - son **nom** : c'est-à-dire sous quel **nom** est déclaré la variable ;
  - son **type** : c'est la convention d'interprétation de la séquence de bits qui constitue la variable. Le type de la variable spécifie aussi sa **taille** (la longueur de cette séquence) soit habituellement 8 bits, 32 bits, 64 bits, ... ;
  - sa **valeur** : c'est la séquence de bit elle même. Cette séquence peut être codée de différrentes façons suivant son type. Certaines variables sont déclarées **constantes** (ou en lecture seule) et sont donc protégées contre l'écriture. Le mot clé `const` permet de déclarer des variables constantes ; son adresse : c'est l'endroit dans la mémoire ou elle est stockée ;
  - sa **durée de vie** : c'est la portion de code dans laquelle la variable existe. Généralement, on parle de variable **locale** (à un bloc {}) ou **globale** (accessible de partout dans le code du programme) ;
  - sa **visibilité** : c'est un ensemble de règles qui fixe qui peut utiliser la variable. En C++, les variables (membres) **private** ne sont visibles qu'à l'intérieur de la classe par opposition aux variables (membres) **public** qui elles sont visibles aussi à l'extérieur de la classe. En C, il est possible de "cacher" certaines variables ;

Créer une variable

- Pour créer une variable en C/C++, il faut bien évidemment lui **choisir son nom** (parlant et précis) mais aussi obligatoirement lui **choisir son type** :
  - pour les variables **booléenes** : `bool` (seulement en C++)
  - pour les **nombres entiers** : `char`, `short`, `int`, `long` et `long long`
  - pour les **nombres à virgule flottante** : `float`, `double` et `long double`
- Le type `char` permet aussi de stocker le code ASCII d'une lettre (**caractère**).
- Il est recommandé d'**initialiser la variable** (attention à ne pas l'oublier !) avec une valeur du type correspondant.
- *Remarque :* pour les variables de type entier, il est possible de préciser à la déclaration si celle-ci aura un **signe** (`signed`) ou non (`unsigned`).

Exemple 1a : des variables de différents types en C

```c
#include <stdio.h> /* fichier en-tête pour accéder à la déclaration de la fonction printf */

int main() /* la fonction principale appelée automatiquement au lancement de l'exécutable */
{
   int nombreDOeufs = 3; // 3 est une valeur entière
   unsigned long int jeSuisUnLong = 12345678UL; // U pour unsigned et L pour long
   float quantiteDeFarine = 350.0; // ".0" rend la valeur réelle
   char lettre = 'g'; // ne pas oublier les quotes : ' '

   printf("La variable nombreDOeufs a pour valeur %d\n", nombreDOeufs);
   printf("La variable nombreDOeufs occupe %d octet(s)\n", sizeof(int)); // l'opérateur sizeof retourne la
         taille en octets de la variable
   printf("La variable jeSuisUnLong a pour valeur %lu\n", jeSuisUnLong);
   printf("La variable jeSuisUnLong occupe %d octet(s)\n", sizeof(unsigned long int));
   printf("La variable quantiteDeFarine a pour valeur %f\n", quantiteDeFarine);
   printf("La variable quantiteDeFarine occupe %d octet(s)\n", sizeof(float));
   printf("La variable lettre a pour valeur %c (%d ou 0x%02X)\n", lettre, lettre, lettre);
   printf("La variable lettre occupe %d octet(s)\n", sizeof(char));
   printf("Recette : il faut %d oeufs et %.1f %c de farine\n", nombreDOeufs, quantiteDeFarine, lettre);

   return 0; /* fin normale du programme */
}
```

Exemple 1a : fabrication de l'exécutable (sous Linux)

```
$ gcc <fichier.c>
```

- *Attention :* Le binaire, le décimal et l'hexadécimal ne sont qu'une représentation numérique d'une même valeur (cf. la variable `lettre`).

Exemple 1a : exécution

```
$ ./a.out
La variable nombreDOeufs a pour valeur 3
La variable nombreDOeufs occupe 4 octet(s)
La variable jeSuisUnLong a pour valeur 12345678
La variable jeSuisUnLong occupe 4 octet(s)
La variable quantiteDeFarine a pour valeur 350.000000
La variable quantiteDeFarine occupe 4 octet(s)
La variable lettre a pour valeur g (103 ou 0x67)
La variable lettre occupe 1 octet(s)
Recette : il faut 3 oeufs et 350.0 g de farine
```

Exemple 1b : des variables de différents types en C++

```
#include <iostream> /* pour cout */
using namespace std;

int main() /* la fonction principale appelée automatiquement au lancement de l'exécutable */
{
   bool reussie = true; // true est une valeur booléenne
   int nombreDOeufs = 3; // 3 est une valeur entière
   unsigned long int jeSuisUnLong = 12345678UL; // U pour unsigned et L pour long
   float quantiteDeFarine = 350.0f; // ".0" rend la valeur réelle et f pour float
   char unite = 'g'; // ne pas oublier les quotes : ' '

   cout << "La variable nombreDOeufs a pour valeur " << nombreDOeufs << endl;
   cout << "La variable jeSuisUnLong a pour valeur " << jeSuisUnLong << endl;
   cout << "La variable quantiteDeFarine a pour valeur " << quantiteDeFarine << endl;
   cout << "La variable unite a pour valeur " << unite << endl;
   cout << "Recette " << reussie << " : il faut " << nombreDOeufs << " oeufs et " << quantiteDeFarine << "
         " << unite << " de farine" << endl;

   return 0; /* fin normale du programme */
}
```

Exemple 1b : fabrication de l'exécutable (sous Linux)

```
$ g++ <fichier.cpp>
```

Exemple 1b : exécution

```
La variable nombreDOeufs a pour valeur 3
La variable jeSuisUnLong a pour valeur 12345678
La variable quantiteDeFarine a pour valeur 350
La variable unite a pour valeur g
Recette 1 : il faut 3 oeufs et 350 g de farine
```

Remarque : `std` représente un **espace de nom**. `cin` et `cout` existent dans cet espace de nom mais pourraient exister dans d'autres espaces de noms. Le nom complet pour y accéder est donc `std::cin` et `std::cout`. L'opérateur :: permet la résolution de portée en C++ (un peu comme le / dans un chemin !).

Pour éviter de donner systématiquement le nom complet, on peut écrire le code ci-dessous. Comme on utilise quasiment tout le temps des fonctions de la bibliothèque standard, on utilise presque tout le temps " `using namespace std;` " pour se simplifier la vie !

Exemple 1c : utilisation de l'espace de nom `std`

```
#include <iostream>

using namespace std; // si C++ ne connaît pas un symbole (cout, endl ici), il le cherchera dans std

int main()
{
   cout << "Hello world !" << endl;
   return 0;
}
```

Les types entiers

- `bool` : false ou true → booléen (seulement en C++)
- `unsigned char` : 0 à 255 ($2^8 - 1$) → entier très court (1 octet ou 8 bits)
- `[signed] char` : -128 (-2^7) à 127 ($2^7 - 1$) → idem mais en entier relatif
- `unsigned short [int]` : 0 à 65535 ($2^{16} - 1$) → entier court (2 octets ou 16 bits)
- `[signed] short [int]` : -32768 (-2^{15}) à +32767 ($2^{15} - 1$) → idem mais en entier relatif
- `unsigned int` : 0 à 4.295e9 ($2^{32} - 1$) → entier sur 4 octets ; **taille "normale" actuelle**
- `[signed] int` : -2.147e9 (-2^{31}) à +2.147e9 ($2^{31} - 1$) → idem mais en entier relatif
- `unsigned long [int]` : 0 à 4.295e9 → entier sur 4 octets ou plus ; sur PC identique à "int" (hélas...)
- `[signed] long [int]` : -2.147e9 à -2.147e9 → idem mais en entier relatif
- `unsigned long long [int]` : 0 à 18.4e18 ($2^{64} - 1$) → entier (très gros !) sur 8 octets sur PC
- `[signed] long long [int]` : -9.2e18 (-2^{63}) à -9.2e18 ($2^{63} - 1$) → idem mais en entier relatif

Les types à virgule flottante

- `float` : Environ 6 chiffres de précision et un exposant qui va jusqu'à $\pm 10^{\pm 38}$ $\rightarrow$ Codage IEEE754 sur 4 octets
- `double` : Environ 10 chiffres de précision et un exposant qui va jusqu'à $\pm 10^{\pm 308}$ $\rightarrow$ Codage IEEE754 sur 8 octets
- `long double` $\rightarrow$ Codé sur 10 octets

Complément à 2

- Les `signed char/int` sont implantés en **complément à 2** pour intégrer **un bit de signe** pour représenter les entiers relatifs.
- Tout d'abord, on utilise le bit de poids fort (bit le plus à gauche) du nombre pour représenter son signe (égal à 1 et il est négatif et sinon il est positif).
- Il reste donc pour représenter le nombre : taille en bits - 1.
- Les nombres positifs sont représentés normalement, en revanche les nombres négatifs sont obtenus de la manière suivante :
  - On inverse les bits de l'écriture binaire de sa valeur absolue (opération binaire NON), on fait ce qu'on appelle le complément à un
  - On ajoute 1 au résultat (les dépassements sont ignorés).

Complément à 2 : Exemple

- Pour coder le nombre (-4), on fera :
  - On prend le nombre positif 4 : `00000100`
  - On inverse les bits : `11111011`
  - On ajoute 1 pour obtenir le nombre -4 : `11111100`
- On peut remarquer que le bit de signe s'est positionné automatiquement à 1 pour indiquer un nombre négatif.
- La représentation en complément à 2 a été réténue car elle permet de d'effectuer les opérations arithmétiques usuelles naturellement. On peut vérifier que l'opération **3 + (-4)** se fait sans problème : `00000011 + 11111100 = 11111111`
- Le complément à deux de `11111111` est `00000001` soit 1 en décimal, donc avec le bit de signe à 1, ont obtient **(-1)** en décimal.

Le standard IEEE 754

- Le **standard IEEE 754 définit les formats de représentation des nombres à virgule flottante** de type `float/double` (signe, mantisse, exposant, nombres dénormalisés) et valeurs spéciales (infinis et NaN), un ensemble d'opérations sur les nombres flottants et quatre modes d'arrondi et cinq exceptions (cf. fr.wikipedia.org/wiki/IEEE_754).
- La version 1985 de la norme IEEE 754 définit **4 formats** pour représenter des nombres à virgule flottante :
  - simple précision (32 bits : 1 bit de signe, 8 bits d'exposant (-126 à 127), 23 bits de mantisse, avec bit 1 implicite),
  - simple précision étendue (>= 43 bits, obsolète, implémenté en pratique par la double précision),
  - double précision (64 bits : 1 bit de signe, 11 bits d'exposant (-1022 à 1023), 52 bits de mantisse, avec bit 1 implicite),
  - double précision étendue (>= 79 bits, souvent implémenté avec 80 bits : 1 bit de signe, 15 bits d'exposant (-16382 à 16383), 64 bits de mantisse, sans bit 1 implicite).

Mantisse et partie significative

- Le compilateur `gcc` (pour les architectures compatible Intel 32 bits) utilise le format **simple précision** pour les variables de type `float`, **double précision** pour les variables de type `double`, et la double précision ou la double précision étendue (suivant le système d'exploitation) pour les variables de type `long double`.
- La **mantisse est la partie décimale** de la partie significative, **comprise entre 0 et 1**.
- Donc **1+mantisse** représente la **partie significative**.
- Elle se code (attention on est à droite de la virgule) :
  - pour le premier bit avec le poids 2^{-1} soit `0,5`,
  - puis 2^{-2} donc `0,25`,
  - 2^{-3} donnera `0,125`, ainsi de suite ...
- Par exemple : `,010` donnera `,25`.

Les limites des nombres

- Le fichier `limits.h` contient, sous forme de constantes ou de macros, les limites concernant le codage des entiers et le fichier `float.h` contient celles pour les "floattants".

Exemple : les limites des nombres en C

```c
#include <stdio.h>
#include <limits.h>
#include <float.h>

int main() {
   printf("*** Limites pour les entiers ***\n");
   printf("Nombre de bits dans un char : %d bits\n", CHAR_BIT);
   printf("%d <= char <= %d\n", CHAR_MIN, CHAR_MAX);
   printf("0 <= unsigned char <= %u\n", UCHAR_MAX);
   printf("%d <= int <= %d\n", INT_MIN, INT_MAX);
   printf("0 <= unsigned int <= %u\n", UINT_MAX);
   printf("%ld <= long <= %ld\n", LONG_MIN, LONG_MAX);
   printf("0 <= unsigned long <= %lu\n", ULONG_MAX);
   printf("\n*** Limites pour les réels ***\n");
   printf("%e <= float <= %e\n", FLT_MIN, FLT_MAX);
   printf("%e <= double <= %e\n", DBL_MIN, DBL_MAX);
   printf("%Le <= long double <= %Le\n", LDBL_MIN, LDBL_MAX);
   return 0;
}
```

Exemple : exécution

```
*** Limites pour les entiers ***
Nombre de bits dans un char : 8 bits
-128 <= char <= 127
0 <= unsigned char <= 255
-2147483648 <= int <= 2147483647
0 <= unsigned int <= 4294967295
-2147483648 <= long <= 2147483647
0 <= unsigned long <= 4294967295

*** Limites pour les réels ***
1.175494e-38 <= float <= 3.402823e+38
2.225074e-308 <= double <= 1.797693e+308
3.362103e-4932 <= long double <= 1.189731e+4932
```

- Première loi : Le type d'une variable spécifie sa taille (de sa représentation en mémoire) et ses limites. Attention, elle peut donc **déborder** (*overflow*).

Les constantes

- Celles définies **pour le préprocesseur** : c'est simplement une **substitution syntaxique pure** (une sorte de copier/coller). Il n'y a aucun typage de la constante.

```
#define PI 3.1415 /* en C traditionnel */
```

  L'utilisation de `#define` améliore surtout la lisibilité du code source. La convention usuelle est d'utiliser des MAJUSCULES (pour les distinguer des variables).

- Celles définies **pour le compilateur** : c'est une **valeur typée**, ce qui permet des contrôles lors de la compilation.

```
const double pi = 3.1415; // en C++ et en C ISO
```

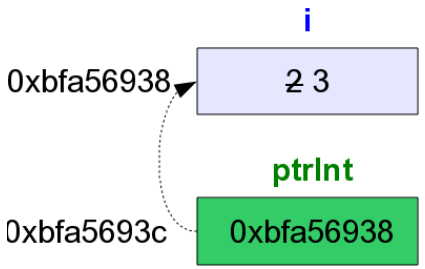
  A utiliser pour des valeurs constantes.

Les pointeurs

- Les pointeurs sont des variables spéciales permettant de **stocker une adresse** (pour la manipuler ensuite).
- L'adresse représente généralement l'**emplacement mémoire d'une variable** (ou d'une autre adresse).
- Comme la variable a un type, le pointeur qui stockera son adresse doit être du même type pour la manipuler convenablement.
- Le type `void*` représentera un **type générique** de pointeur : en fait cela permet d'indiquer sagement que l'on ne sait pas encore sur quel type il pointe.

Utilisation des pointeurs

- On utilise l'étoile (*) pour **déclarer un pointeur**.
  déclaration d'un pointeur (*) sur un entier (int) : `int *ptrInt;`
- On utilise le & devant une variable pour **initialiser ou affecter un pointeur avec une adresse**.
  déclaration d'un entier i qui a pour valeur 2 : `int i = 2;`
  affectation avec l'adresse de la variable i (&i) : `ptrInt = &i;`
- On utilise l'étoile devant le pointeur (*) pour **accéder à l'adresse stockée**.
  indirection ("pointe sur le contenu de i") : `*ptrInt = 3;`



Maintenant la variable i contient 3

Particularités des pointeurs

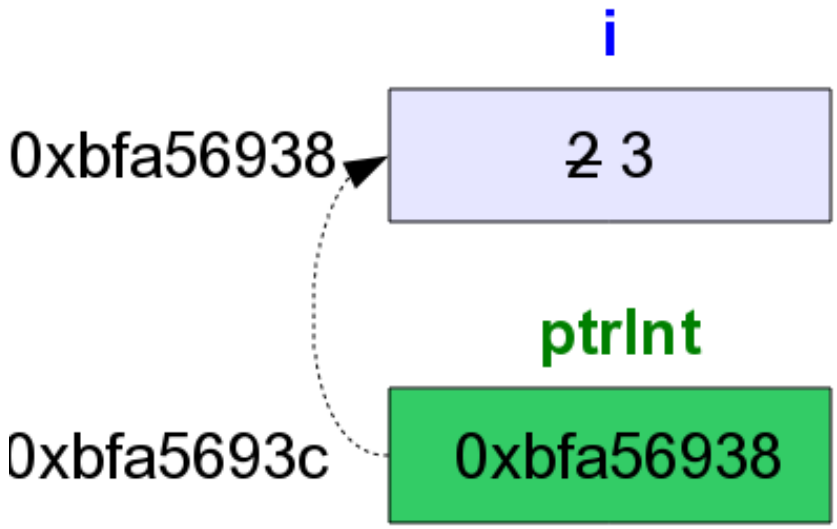
- Avec `printf`, il est possible d'afficher une adresse mémoire en utilisant le formateur %p.
- Pour connaître la taille qu'occupe une variable de type pointeur, on utilisera `sizeof()`. Sur une architecture donnée, tous les pointeurs ont la même taille (probablement 32 bits). Sur 32 bits, un pointeur pourra contenir une adresse dans un espace de 4 GO (2^{32}).
- Les pointeurs peuvent être incrémentés, décrémentés, additionnés ou soustraits. Dans ce cas, leur nouvelle valeur dépend du type sur lequel ils pointent.
- Exemple : incrémenter un pointeur de `char` ajoute 1 à sa valeur (un caractère est représenté sur 1 octet). Incrémenter un pointeur sur un `int` ajoute 4 à sa valeur (cela dépend de l'architecture, un entier pouvant être représenté sur 4 octets).

Exemple : manipulons un pointeur

```c
#include <stdio.h>

int main()
{
   int i = 2; // déclaration d'un entier i qui a pour valeur 2
   int *ptrInt; // déclaration d'un pointeur (*) sur un entier (int)

   printf("La variable i a pour valeur %d et pour adresse %p\n", i, &i);
   ptrInt = &i; // affectation du pointeur avec l'adresse de la variable i
   printf("La variable ptrInt contient l'adresse %p et pointe donc sur i qui contient %d\n", ptrInt, *
         ptrInt);

   *ptrInt = 3; // j'accède à l'adresse contenue dans le pointeur et je pointe dessus (*ptrInt) et je
         modifie son contenu

   printf("La variable i a maintenant pour valeur %d\n", i);

   return 0;
}
```

Exemple 2 : exécution

```
La variable i a pour valeur 2 et pour adresse 0xbfa56938

La variable ptrInt contient l'adresse 0xbfa56938 et pointe donc sur i qui
     contient 2

La variable i a maintenant pour valeur 3
```



Les références en C++

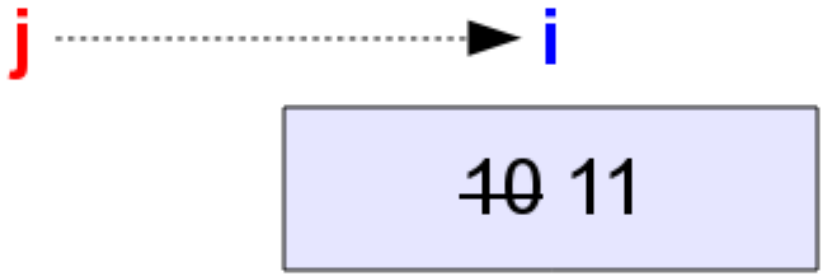
- En C++, il est possible de déclarer une référence `j` sur une variable `i` : cela permet de créer un **nouveau nom `j` qui devient synonyme de `i`** (**un alias**).
- On pourra donc modifier le contenu de la variable en utilisant une référence.
- La déclaration d'une référence se fait en précisant le type de l'objet référencé, puis le symbole &, et le nom de la variable référence qu'on crée.
- Une référence ne peut être initialisée qu'une seule fois : à la déclaration.
- Une référence ne peut donc référencer qu'une seule variable tout au long de sa durée de vie.

Exemple : les références

```
#include <iostream>

int main (int argc, char **argv)
{
   int i = 10; // i est un entier valant 10
   int &j = i; // j est une référence sur un entier, cet entier est i.
   //int &k = 44; // ligne7 : illégal

   std::cout << "i = " << i << std::endl;
   std::cout << "j = " << j << std::endl;

   // A partir d'ici j est synonyme de i, ainsi :
   j = j + 1; // est équivalent à i = i + 1 !

   std::cout << "i = " << i << std::endl;
   std::cout << "j = " << j << std::endl;

   return 0;
}
```

Exemple : exécution

```
i = 10
j = 10
i = 11
j = 11
```



Si on dé-commente la ligne 7, on obtient cette erreur à la compilation :

```
ligne 7: erreur: invalid initialization of non-const référence of type 'int&'
     from a temporary of type 'int'
```

Intérêt des références

Attention : le = dans la déclaration de la référence n'est pas réellement une affectation puisqu'on ne copie pas la valeur de `i`. En fait, on affirme plutôt le lien entre `i` et `j`. En conséquence, la ligne 7 est donc parfaitement illégal ce que signale le compilateur.

- Comme une référence établit un lien entre deux noms, leur utilisation est efficace dans le cas de variable de grosse taille car cela évitera toute copie.
- Les références sont (systématiquement) utilisées dans le passage des paramètres d'une fonction (ou d'une méthode) dès que le coût d'une recopie par valeur est trop important ("gros" objet).
- Exemple :

```
void truc(const grosObjet& rgo);
```

Variables globale et locale

- une **variable globale** est une **variable déclarée à l'extérieur du corps de toute fonction ou classe**, et pouvant donc être utilisée n'importe où dans le programme. On parle également de variable de portée globale.
- une **variable locale** est une **variable qui ne peut être utilisée que dans la fonction ou le bloc où elle est définie**. On parle également de variable de portée locale.

Durée de vie

Exemple : Variable globale et locale

```c
#include <stdio.h>
int g = 1; // je suis une variable globale

int main()
{
   int x = 5; // je suis une variable locale, ma durée de vie est celle du main

   printf("La variable globale g a pour valeur %d\n", g);
   printf("La variable locale x a pour valeur %d\n", x);

   return 0;
}
```

Exemple : exécution

```
La variable globale g a pour valeur 1
La variable locale x a pour valeur 5
```

Visibilité

Exemple : "Attention, une variable peut en cacher une autre"

```c
#include <stdio.h>

int i = 1; // je suis une variable globale de nom i

int main()
{
   int i = 2; // je suis une variable locale de nom i ! aie !

   printf("La variable i a pour valeur %d\n", i);

   return 0;
}
```

- Conclusion : La visibilité s'applique à la plus "proche" déclaration.

Exemple : exécution

```
La variable i a pour valeur 2
```

typdedef

- Le mot réservé typedef (signifie littéralement « définition de type ») permet simplement la définition de **synonyme de type** qui peut ensuite être utilisé à la place d'un nom de type :

Exemple d'utilisation de typedef

```
typedef int                entier;
typedef float              reel;

entier a; // a de type entier donc de type int
reel   x; // x de type réel donc de type float
```

Conclusion : Le mot-clé typedef permet donc au programmeur de créer de nouveaux noms de types.

Intérêt de typdedef

`typedef` peut être utilisé à la fois pour :

- donner plus de clarté au code source
- rendre plus facile les modifications de ses propres types de données
- permettre la portabilité du code source (sur une autre ou future plateforme)

Notez que beaucoup de langages fournissent l'alias de types ou la possibilité de déclarer plusieurs noms pour le même type.

enum

- enum permet de déclarer un **type énuméré** constitué d'un ensemble de constantes appelées **énumérateurs**.
- Une variable de type énuméré peut recevoir n'importe quel énumérateur (lié à ce type énuméré) comme valeur.
- Le premier énumérateur vaut zéro (par défaut), tandis que tous les suivants correspondent à leur précédent incrémenté de un.

Exemple d'utilisation de enum

```
enum couleur_carte
{
    TREFLE = 1, /* un énumérateur */
    CARREAU, /* 1+1 donc CARREAU = 2 */
    COEUR  = 4, /* en C, les énumérateurs sont équivalents à des entiers (int)
         */
    PIQUE  = 8 /* il est possible de choisir explicitement les valeurs (ou de
        certaines d'entre elles). */
};
```

Exemple : utilisation des typedef et enum précédents

```
typedef int              entier;
typedef float            reel;
typedef enum{FALSE,TRUE} booleen;

void main() {
   entier e = 1, reel r = 2.5, booleen fini = FALSE;
   enum couleur_carte carte = CARREAU;
   printf("Le nouveau type entier possède une taille de %d octets (ou %d bits)\n", sizeof(entier), sizeof(
         entier)*8);
   printf("La variable e a pour valeur %d et occupe %d octets\n", e, sizeof(e));
   printf("La variable r a pour valeur %.1f et occupe %d octets\n", r, sizeof(r));
   printf("La variable fini a pour valeur %d et occupe %d octets\n", fini, sizeof(fini));
   printf("La variable carte a pour valeur %d et occupe %d octets\n", carte, sizeof(carte));
}
```

Exemple : exécution

```
Le nouveau type entier possède une taille de 4 octets (ou 32 bits)
La variable e a pour valeur 1 et occupe 4 octets
La variable r a pour valeur 2.5 et occupe 4 octets
La variable fini a pour valeur 0 et occupe 4 octets
La variable carte a pour valeur 2 et occupe 4 octets
```

lvalue

- Une *Left-value* (valeur gauche) est un élément de syntaxe C/C++ pouvant **être écrit à gauche d'un opérateur d'affectation (=)**.
- *Exemple :* une variable, une case de tableau, ...
- Modèle d'une affectation :
  `lvalue = rvalue;` soit `left-value` ← `right-value`
- Une **left-value** doit donc être **un emplacement de stockage en mémoire possédant un type précis**, c'est-à-dire la référence à quelque chose de modifiable.

Affectation d'une lvalue

```c
#include <stdio.h>

int main()
{
   int x;

   x = 5; // affectation de la valeur entière 5 à la variable x (x est
       une lvalue)

   2 = x + 1; // problème car 2 n'est pas une lvalue (2 n'est pas "
       modifiable") !

   return 0;
}
```

- Le compilateur détecte l'erreur et le message est très clair : "error: lvalue required as left operand of assignment"

rvalue

- Une *Right-value* (valeur droite) est un élément de syntaxe C/C++ pouvant **être écrit à droite d'un opérateur d'affectation (=)**.
- *Exemple :* une valeur, une constante, une variable, une expression, ...
- Modèle d'une affectation :
  `lvalue = rvalue;` soit `left-value` ← `right-value`
- Une **right-value** doit être une valeur d'un type précis mais n'a pas forcement de zone de stockage en mémoire.
- Une affectation est encore une **right-value** ... ce qui donne le droit d'écrire : `i = j = 10;`
- C'est équivalent à `i=(j=10);` c'est-à-dire : `j=10; i=j;`

Utilisation d'une affectation comme rvalue

```c
#include <stdio.h>

int main()
{
   int i = 0;
   int j = 0;

   i = j = 10;

   printf("i = %d\n", i); // Pas de surprise : i = 10
   printf("j = %d\n", j); // Pas de surprise : j = 10

   return 0;
}
```

Conversion "forcée" par la lvalue

- Les opérateurs d'affectation (=, -=, += ...), appliqués à des valeurs de type numérique, provoquent la conversion de leur opérande de droite dans le type de leur opérande de gauche. Cette conversion "forcée" peut être "dégradante" (avec perte).

```
#include <stdio.h>

int main() {
   int x = 5; float y = 1.5;
   int res;

   res = (x + y); // cela revient à faire (int)(x + y) car res est de
        type int
   printf("res = %d\n", res); // Affiche : res = 6

   return 0;
}
```

Conversion automatique

- Soit l'opération suivante : `short int a = 2; a + 2;`
- L'opération `a + 2` revient à faire l'addition entre un `short int` (a) et un `int` (2). Cela est impossible car on ne peut réaliser que des opérations entre **type identique**.
- Une conversion implicite (automatique) sera faite (pouvant donner lieu à un *warning* de la part du compilateur).
- Les conversions d'ajustement de type automatique réalisées suivant la hiérarchie ci-dessous sont réalisées **sans perte** :
  1. `char` → `short int` → `int` → `long` → `float` → `double` → `long double`
  2. `unsigned int` → `unsigned long` → `float` → `double` → `long double`

Conversion forcée ou cast

- Une conversion de type (ou de promotion de type) peut être implicite (automatique) ou **explicite** (c'est-à-dire forcée par le programmeur).
- Lorsqu'elle est explicite, on utilise l'**opérateur de *cast*** : `(float)a` permet de forcer le `short int a` en `float`.
- Les conversions forcées peuvent être des **conversions dégradantes (avec perte)**. Par exemple : `int b = 2.5;`
- En effet, le cast `(int)b` donnera 2 : perte de la partie décimale. Cela peut être dangereux (source d'erreur).

Les opérateurs arithmétiques et logiques

- Les **opérateurs arithmétiques** (`+`, `-`, `*`, `/` et `%`) et les **opérateurs relationnels** (`<`, `<=`, `>`, `>=`, `==` et `!=`) ne sont définis que pour des **opérandes d'un même type** parmi : `int`, `long int` (et leurs variantes non signées), `float`, `double` et `long double`.
- Mais on peut constituer des expressions mixtes (opérandes de types différents) ou contenant des opérandes d'autres types (`bool`, `char` et `short`), grâce aux **conversions implicites et explicites**.
- Les **opérateurs logiques** `&&` (et), `||` (ou) et `!` (non) acceptent n'importe quel opérande numérique (entier ou flottant) ou pointeur, en considérant que tout opérande de valeur non nulle correspond à "faux". Les deux opérateurs `&&` et `||` sont **"à court-circuit"** : le second opérande n'est évalué que si la connaissance de sa valeur est indispensable.
- *Remarque :* l'opérateur **modulo (%)** permet d'obtenir le reste d'une division euclidienne. C'est un opérateur très utilisé en programmation.

Opérateurs logiques à court-circuit

Attention : le second opérande n'est évalué que si la connaissance de sa valeur est indispensable.

```
int a = 0;
// Danger : si le premier opérande suffit à déterminer l'évaluation
    du résultat logique
if( 0 < 1 || a++ != 5 ) // Attention : a++ != 5 n'est pas évalué donc
     (a n'est pas incrémenté) car 0 < 1 et donc toujous VRAI dans un
    OU
    printf("VRAI !\n"); // Affiche toujours : VRAI !
else printf("FAUX !\n");
if( 1 < 0 && a++ != 5 ) // Attention : a++ != 5 n'est pas évalué donc
     (a n'est pas incrémenté) car 1 < 0 et et donc toujous FAUX dans
    un ET
    printf("VRAI !\n");
else printf("FAUX !\n"); // Affiche toujours : FAUX !
printf("a = %d\n", a); // Affichera toujours : a = 0 !!!
```

Opérateurs logique et bit à bit (1/2)

Ne pas confondre les opérateurs logiques avec les opérateurs bit à bit

```
unsigned char a = 1; unsigned char b = 0;
unsigned char aa = 20; /* non nul donc VRAI en logique */
unsigned char bb = 0xAA;

// Ne pas confondre !
/* ! : inverseur logique */
/* ~ : inverseur bit à bit */
printf("a = %u - !a = %u - ~a = %u (0x%hhX)\n", a, !a, ~a, ~a);
printf("b = %u - !b = %u - ~b = %u (0x%hhX)\n", b, !b, ~b, ~b);

printf("aa = %u (0x%hhX) - !aa = %u - ~aa = %u (0x%hhX)\n", aa, aa, !
    aa, ~aa, ~aa);
printf("bb = %u (0x%hhX) - !bb = %u - ~bb = %u (0x%hhX)\n", bb, bb, !
    bb, ~bb, ~bb);
```

Opérateurs logique et bit à bit (2/2)

- Pour les opérateurs bit à bit, il est conseillé d'utiliser la représentation en hexadécimale :

Exemple

```
a = 1 - !a = 0 - ~a = 4294967294 (0xFE)

b = 0 - !b = 1 - ~b = 4294967295 (0xFF)

aa = 20 (0x14) - !aa = 0 - ~aa = 4294967275 (0xEB)

bb = 170 (0xAA) - !bb = 0 - ~bb = 4294967125 (0x55)
```

Les opérateurs d'affectation

- Les **opérateurs d'affectation** (=, +=, -=, *=, /=, %=, &=, ^=, |=, «= et »=) nécessitent une **lvalue** pour l'opérande de gauche.
- L'affectation à la déclaration d'une variable est appelée "**déclaration avec initialisation**", par exemple : `int a = 5;` déclare a et l'initialise avec la valeur entière 5.
- **Affectation avec opérateur** : Il existe toute une gamme d'opérateurs permettant d'effectuer une opération avec le contenu d'une variable et de mettre le résultat dans cette même variable.
  - Une opération du type : `a operateur= expression;` équivaut à : `a = a operateur (expression);`
  - Par exemple : L'opérateur "+=" signifie "affecter en additionnant à" : `a += 2;` est équivalent `a = a + 2;`

Opération sur les pointeurs

- Les opérations arithmétiques sur les pointeurs sont bien évidemment réalisées sur les adresses contenues dans les variables pointeurs.
- Le type du pointeur a une influence importante sur l'opération.
- Supposons un tableau `t` de 10 entiers (`int`) initialisés avec des valeurs croissantes de 0 à 9. Si on crée un pointeur `ptr` sur un entier (`int`) et qu'on l'initialise avec l'adresse d'une case de ce tableau, on pourra alors se déplacer avec ce pointeur sur les cases de ce tableau. Comme `ptr` pointe sur des entiers (c'est son type), son adresse s'ajustera d'un décalage du nombre d'octets représentant la taille d'un entier (`int`). Par exemple, une incrémentation de l'adresse du pointeur correspondra à une opération +4 (octets) si la taille d'un `int` est de 4 octets !

Opération de déplacement d'un pointeur

```
int t[10] = { 0,1,2,3,4,5,6,7,8,9 };
int *ptr; // un pointeur pour manipuler des int

ptr = &t[5]; // l'adresse d'une case du tableau
printf("Je pointe sur la case : %d (%p)\n", *ptr, ptr); // Je pointe
    sur la case : 5 (0xbf8d87b8)

ptr++; // l'adresse est incrémentée de 4 octets pour pointer sur l'
    int suivant
printf("Maintenant, je pointe sur la case : %d (%p)\n", *ptr, ptr);
    // Maintenant, je pointe sur la case : 6 (0xbf8d87bc)

ptr -= 4; // en fait je recule de 4 int soit 4*4 octets pour la
    valeur de l'adresse
printf("Maintenant, je pointe sur la case : %d (%p)\n", *ptr, ptr);
    // Maintenant, je pointe sur la case : 2 (0xbf8d87ac)
```

Opérateurs d'incrémentation et de décrémentation

- Les **opérateurs unaires** (à une seule opérande) d'incrémentation (++) et de décrémentation (--) agissent sur la valeur de leur unique opérande (qui doit être une **lvalue**) et fournissent la valeur après modification lorsqu'ils sont placés à gauche (comme dans `++n`) ou avant modification lorsqu'ils sont placés à droite (comme dans `n--`).
- `i++;` est (à première vue) équivalent à `i = i + 1;`
- Mais `i++` est une *right-value* donc ...

```
int i = 10;
int j = i++; // équivalent à int j=i; i=i+1;

printf("i = %d\n", i); // Affiche : i = 11
printf("j = %d\n", j); // Affiche : j = 10
```

Attention c'est une post-incrémentation : on augmente i après avoir affecté j.

Opérateur ternaire : ?

- L'**opérateur ternaire** ? ressemble au `if(...) {...} else {...}` mais joue un rôle de *right-value* et pas de simple instruction.
- La syntaxe est la suivante : `(A?B:C)` prend la valeur de l'expression `B` si l'expression `A` est vraie, sinon la valeur de l'expression `C`.

```
int age = 1; int parite; /* un booléen */

printf("J'ai %d an%c\n", age, age > 1 ? 's' : ''); // J'ai 1 an
printf("Je suis %s\n", age >= 18 ? "majeur" : "mineur"); // Je suis
    mineur
parite = (age%2 == 0 ? 1 : 0 );
printf("Parité = %d\n\n", parite); // Parité = 0

age = 20;
printf("J'ai %d an%c\n", age, (age > 1) ? 's' : ''); // J'ai 20 ans
printf("Je suis %s\n", (age >= 18) ? "majeur" : "mineur"); // Je suis
     majeur
parite = (age%2 == 0 ? 1 : 0 );
printf("Parité = %d\n", parite); // Parité = 1
```

Priorité des opérateurs (1/2)

Liste des opérateurs du plus prioritaire au moins prioritaire :

- :: (opérateur de résolution de portée en C++)
- `.` `->` `[]` (référence et sélection) `()` (appel de fonction) `()` (parenthèses) `sizeof()`
- `++` `~` (inverseur bit à bit) `!` (inverseur logique) `-` (unaire) `&` (prise d'adresse) `*` (indirection) `new` `delete` `delete[]` (opérateurs de gestion mémoire en C++)
- `()` (conversion de type)
- `*` `/` `%` (multiplication, division, modulo)
- `+` `-` (addition et soustraction)
- `«` `»` (décalages et envoi sur flots)
- `<` `<=` `>` `>=` (comparaisons)
- `==` `!=` (comparaisons)
- `&` (ET bit à bit)
- `^` (OU-Exclusif bit à bit)
- `|` (OU-Inclusif bit à bit)
- `&&` (ET logique)
- `||` (OU logique)
- `(?` `:` `)` (expression conditionnelle ou opérateur ternaire)
- `=` `*=` `/=` `%=` `+=` `=` `«=` `»=` `&=` `|=` `~=`
- `,` (mise en séquence d'expressions)

Priorité des opérateurs (2/2)

Quelques exemples :

1. `y = (x+5)` est équivalent à : `y = x+5` car l'opérateur + est prioritaire sur l'opérateur d'affectation =.
2. `(i++) * (n+p)` est équivalent à : `i++ * (n+p)` car l'opérateur ++ est prioritaire sur *. En revanche, * est prioritaire sur +, de sorte qu'on ne peut éliminer les dernières parenthèses.
3. `moyenne = 5 + 10 + 15 / 3` donnera 20 (/ est plus prioritaire que le +) alors que mathématiquement le résultat est 10 ! Il faut alors imposer l'ordre en l'indiquant avec des parenthèses : `moyenne = (5 + 10 + 15) / 3`
4. *Important :* Si deux opérateurs possèdent la même priorité, C exécutera les opérations de la gauche vers la droite (sauf pour les opérateurs suivants où l'ordre est de la droite vers la gauche : ++ ~ (inverseur bit à bit) ! (inverseur logique) - (unaire) & (prise d'adresse) * (indirection) `new delete delete[] (? :)` et = *= /= %= += = «= »= &= |= ~=).
5. L'ordre des opérateurs n'est donc pas innocent. En effet : 3/6*6 donnera 0 alors que 3*6/6 donnera 3 !

Conclusion : comme il est difficile de se rappeler de l'ensemble des priorités des opérateurs, le programmeur préfère coder explicitement en utilisant des parenthèses afin de s'éviter des surprises. Cela améliore aussi la lisibilité.