Polytech'Paris Sud - S2 « Programmation Impérative » - 2018/2019

NOTES DE COURS - Partie I -

Frédéric VOISIN (frederic.voisin@u-psud.fr)

Compétences associées au module :

- Dans la continuité du module « Introduction à l'informatique » du S1, savoir concevoir des algorithmes simples, les mettre en œuvre dans un langage comme C++ et les tester.
- Savoir utiliser les types d'un langage de programmation pour représenter des informations complexes et structurées.
- À l'aide d'un modèle simple, comprendre la dynamique d'exécution d'un programme afin d'éviter certaines erreurs de programmation.
- Utiliser les « pointeurs » pour représenter des informations complexes (structures récursives ou avec partage, comme les graphes) ou dont la taille n'est pas connue à l'avance.

Contenu du module :

- Rappels et compléments en C++ (types, expressions et structures de données)
- Environnement et état mémoire – Portée des identificateurs - Durée de vie des objets
- Représentation de l'exécution des appels de fonctions. Récursivité des fonctions Notion de « référence » et passage « par référence ». Pointeurs.
- Allocation dynamique de mémoire et structures de données dynamiques.

Organisation Pratique :

16h Cours – 16h TD – 16h TP (sous Windows et l'environnement CodeBlocks)

Modalités de contrôle des connaissances :

Controle1 : 25% ; Controle2 : 30% ; mini-projet : 25% ; I.E. de cours/TD-TP noté : 20%

Aucun document autorisé pendant les contrôles.
Les projets et TP notés sont systématiquement soumis à des comparateurs de code anti-plagiat.

Dates prévisionnelles des contrôles : jeudi 21 février et lundi 20 mai 2019

Page Web du cours: www.lri.fr/~fv/teaching.html . Cette page contiendra des liens vers des tutoriels ou des FAQ, une version PDF du polycopié, des énoncés de TD/TP et quelques corrigés.

Ces notes de cours ne constituent ni un manuel de référence du langage C++, ni un cours d'introduction à la programmation C++ ou à l'algorithmique. Il existe divers ouvrages consultables en bibliothèque ou sur Internet pour cela. Elles ne dispensent pas de suivre le cours et de noter les compléments d'explications et exemples traités en amphi et qui n'apparaissent ***pas*** *dans ces notes de cours. Revoyez aussi vos supports de cours de S1 pour une présentation de certaines notions.*

Il peut subsister dans ce polycopié des coquilles ou des erreurs (notamment dans les fragments de code qui sont édités manuellement pour en améliorer la présentation). Merci de les signaler !

Chapitre I - Rappels sur le langage C++

Dans les pages suivantes, un certain nombre d'informations proviennent de la **norme ANSI/ISO** pour le langage C++. De même que les autres normes utilisées dans la vie courante, elle décrit les exigences que doit satisfaire un compilateur pour être conforme à cette norme : la norme décrit le comportement attendu pour chaque construction du langage (**sémantique** du langage), ce qui est permis et ce qui ne l'est pas, ce qui est vérifié par le compilateur et ce qui ne l'est pas et qui devra être garanti par le programmeur. Une implémentation (voir ci-dessous) peut offrir des extensions par rapport aux minima imposés. Un programme qui utilise ces extensions risque de ne pas compiler sur une autre implémentation, voire de fournir des résultats différents. Votre compilateur sur votre poste de travail pourra donc avoir parfois un comportement plus permissif que ce qui est strictement requis, vous exposant potentiellement à des problèmes de portabilité si vous voulez utiliser votre programme dans un autre environnement[1].

Pour compliquer le tout, il existe des versions successives de la norme, chacune ayant enrichi le langage par des extensions ou précisé certains comportements. Dans les exemples nous utiliserons les possibilités syntaxiques de la norme dite C++11, implémentée dans le compilateur *gcc* utilisé sous CodeBlocks. Il existe une norme C++14 et même C++17 depuis décembre 2017. Pour le sous-ensemble du langage que nous utiliserons, ces différences ne poseront pas problème. Par contre, si votre compilateur est réglé sur une version précédente, certains exemples pourraient ne pas compiler.

« **Implémentation** » : pour une machine, il peut exister plusieurs compilateurs issus de fournisseurs différents (gcc, Visual Studio, ...) qui peuvent proposer leurs propres extensions de la norme. De même, un compilateur est parfois adapté pour des architectures machines différentes, notamment en fonction de la taille d'un « mot » (l'unité facilement adressable sur une machine particulière) exprimée en nombre d'octets. Une « implémentation » est le couple formé par un compilateur particulier, sur un type de machine particulier.

Dans cette première partie, nous présentons rapidement des notions dont la plupart ont été vues en S1. Nous présentons plus les difficultés d'utilisation que les principes de ces constructions, principalement via des exemples. On rappelle qu'on ne voit ici que les bases du langage C++, celles qu'on retrouve d'une façon ou d'une autre dans tout langage de programmation « classique ». Il restera bien d'autres points à apprendre en C++, pour ceux/celles qui l'utiliseront comme langage pour leurs travaux.

Comme le langage C++ est issu du langage C, auquel il apporte un certain nombre d'améliorations, nous signalerons au passage les différences les plus marquantes entre ces deux langages.

1 Pour éviter ce problème les compilateurs vous donnent en général la possibilité de préciser via une option que vous voulez une compilation respectant la norme, donc interdisant certaines extensions. Par exemple avec le compilateur `gcc` il est recommandé d'utiliser les options `-g -Wall -std=c++11` et de paramétrer `CodeBlocks` (voir l'onglet `Settings` et l'entrée du menu `Compiler`) pour que ces options soient systématiquement utilisées.

I - Types de base et constructeurs de types simples :

1. les types numériques et leurs variations
2. les booléens (`bool`)
3. les caractères
4. le type « chaîne de caractères » (`string`)
5. les « vecteurs » (`vector`)
6. les tableaux « simples »
7. les enregistrements (`struct`)

II - Expressions et instructions

III - Autres constructeurs de types :

1. la directive `typedef`
2. les énumérations (`enum`)
3. les types avec variantes (`union`)

I - Types de base et constructeurs de types simples

1. Les types numériques et leurs variations :

Les principes de représentation en binaire des nombres ont été vus en S1. Revoir les supports de cours associés, ainsi que les exemples en TD, pour le codage binaire des entiers (entiers signés ou non signés, principe du « complément à 2 », intervalle de valeurs représentables avec un nombre de bits donnés) ainsi que la représentation des « réels » par des nombres « à virgule flottante » (mantisse, exposant, impossibilité de représenter certaines valeurs de façon exacte, etc).

Rappel des types numériques définis dans le standard :

```
char      les « caractères » ou « entiers (très) courts »
short     les « entiers courts »
int       les « entiers standards »
long      les « entiers longs »
```

en versions **`signed`** et **`unsigned`** (pas de représentation du signe donc plage de valeurs plus importante). Par cohérence, les diverses combinaisons sont autorisées même si cela donne lieu à l'existence de types à la signification « mystérieuse »: qu'est-ce qu'un « caractère signé »?

```
float          /* même principe pour les flottants */
double         17.05, 4., -0.11, .87, -.73, 3.5e12, -7e-2
long double
```

Notons un sous-type intéressant, **`size_t`**, sous-type d'entier **non signé** utile pour tout ce qui est indice, taille associée à un type etc. Selon la norme, cela correspond à un type « pour représenter la taille mémoire (exprimée en octets) d'un objet quelconque dans l'implémentation considérée ». Utile par exemple pour représenter la taille d'une chaîne de caractères ou un indice de tableau, puisque aucun de ces objets ne pourra avoir plus d'éléments que ce qui est représentable par `size_t`. Si on veut définir un indice pour un tableau sans savoir exactement la plage de valeurs à considérer, on peut la déclarer avec le type `size_t` sans craindre de mauvaise surprise quand on passera d'une machine à une autre. *Chaque implémentation définit ce type comme synonyme d'un des types « entiers non-signés » : le mieux adapté à l'architecture de la machine.*

Représentation des nombres et plages de valeurs associées

Les entiers sont représentés sous forme binaire, donc l'intervalle des valeurs représentables dépend du nombre de bits dont on dispose pour une variable de la classe d'entiers considérée. Si on veut pouvoir représenter à la fois des nombres positifs et négatifs (intervalle centré sur 0), le domaine de valeurs représentables sera bien sûr plus petit que si on ne représente que des entiers naturels, puisque qu'il faut mémoriser (sur un bit) le signe du nombre : on diminue de moitié la valeur absolue de la plus grande valeur représentable. Historiquement, plusieurs représentations ont cohabité (signe + valeur absolue, « complément à 1 », « complément à 2 »), ce qui fait que la norme a été conçue pour être compatible avec les différents choix. Même si certains n'ont plus qu'un intérêt historique, cela diminue les domaines des valeurs représentables dans la norme.

Les différentes classes d'entiers correspondent à des nombres de bits différents, donc des plages de valeurs différentes aussi.

Plages de valeurs <u>minimales</u> pour chaque type numérique selon la norme

Une implémentation est libre d'associer à chaque type une plage de valeurs élargie par rapport au standard (bases du standard: `char` = un octet, `short` = 2 octets, `int` = 2 octets, `long` = 4 octets; `float` = 4 octets, `double` = 8 octets).

Dans de très nombreuses implémentations actuelles, un `int` occupe 4 octets et non pas 2, donc offre une plage de valeurs plus étendue que celle indiquée ci-dessous ! La raison est que le type entier le plus utilisé (`int`) correspond à l'unité mémoire la plus efficacement adressable qui, dans les machines et systèmes modernes, n'est plus deux octets mais quatre octets.

Ces plages de valeurs minimales sont définies dans le fichier `<climits>`, ainsi que des constantes `INT_MIN`, `INT_MAX`, `SHRT_MIN`, `SHRT_MAX`, etc., qui permettent de mentionner ces valeurs sous forme symbolique. On peut connaître le nombre d'octets utilisés dans une implémentation donnée grâce à l'opérateur `sizeof` qui donne un résultat exprimé en octets (en fait dans le type `size_t`).

```
long double ld; double d; float f;
long l; int i; short s; char c;

cout << "Long: " << sizeof(l) << ", int: " << sizeof(i)
     << ", short: " << sizeof(s) << ", char: " << sizeof(c) << endl;
cout << "Long double: " << sizeof(ld) << ", double: " << sizeof(d)
     << ", float: " << sizeof(f) << endl;
cout << "SHRT_MIN: " << SHRT_MIN << ", SHRT_MAX: " << SHRT_MAX
     << ", USHRT_MAX: " << USHRT_MAX << endl;
cout << "INT_MIN: " << INT_MIN << ", INT_MAX: " << INT_MAX << endl;
cout << "size_t:" << sizeof(size_t) << endl;
```

Résultats sur une implémentation particulière :

```
Long: 8, int: 4, short: 2, char: 1
Long double: 16, double: 8, float: 4
SHRT_MIN: -32768, SHORT_MAX: 32767, USHORT_MAX: 65535
INT_MIN: -2147483648, INT_MAX: 2147483647
size_t: 4
```

Types entier	Plage de valeurs **minimales** (d'après la norme)
`unsigned char`	`0 .. 255` (`0 .. 2`^{8}`-1`)
`signed char`	`-127 .. +127` (`-2`^{7}`+1 .. 2`^{7}`-1`) [2]
`unsigned short int`	`0 .. 65535`
`short int`	`-32 767 .. +32 767` (`-2`^{15}`+1 .. 2`^{15}`-1`)
`unsigned int`	`0 .. 65 535`
`int`	`-32 767 .. +32 767`
`unsigned long int`	`0 .. 4 294 967 295`
`long int`	`-2 147 483 647 .. +2 147 483 647` (`-2`^{31}`+1 .. 2`^{31}`-1`)

Dans cette implémentation, les plages de valeurs de `int` d'une part et `long double` d'autre part vont au delà de ce qui apparaît dans ce tableau. Il en est de même pour la valeur de `SHRT_MIN`. Le compilateur respecte la norme puisque celle-ci ne prescrit que des plages minimales.

Pour les entiers, la manière de traiter un dépassement de capacité diffère selon que la variable a été déclarée `unsigned` (comportement cyclique garanti) ou non (comportement dépendant du compilateur). Nous n'en dirons pas plus : évitons simplement les dépassements de capacité !

D'autres langages de programmation ont des approches différentes : en JAVA la taille et donc la plage des valeurs d'un type d'entier est fixée par le langage et ne dépend pas d'une implémentation. Un `int` varie toujours dans l'intervalle -2^{15}*.. 2*^{15}*-1 et il n'existe pas d'entiers « non signés ». Dans le langage OCAML, un bit de chaque mot doit être réservé à un usage interne, divisant ainsi par deux la plage de valeurs disponibles. Un `int` varie alors dans l'intervalle -2*$^{30.}$*..2*^{30}*-1 sur une machine 32 bits et est toujours « signé ». La notion « d'entier » peut donc ne pas correspondre dans les différents langages.*

Pour les **nombres « en virgule flottante »** nous ne rappelons pas les détails de la représentation, vus en S1. Ces nombres sont représentés sous la forme :

signe mantisse * 2^{exposant}

avec une partie « exposant » qui délimite **l'amplitude** des valeurs représentables et une partie « mantisse » qui fixe la **précision** de la représentation (nombre de chiffres significatifs après la virgule). Pour des nombres flottants en « simple précision » stocké sur 32 bits, on aura par exemple 1 bit pour le signe, 8 bits pour l'exposant (de -126 à 127 : il y a un décalage par rapport à la plage standard de valeurs vue pour les entiers de manière à simplifier la comparaison entre nombres flottants) et 23 bits pour la mantisse. Il y a de nombreux détails importants de représentation (par exemple pour représenter la notion « d'infini » (+ ∞ ou - ∞) et des valeurs « indéfinies » (*NaN* : « Not a Number »). Pour un nombre en double précision, le format standard est 1 bit de signe, 11 bits d'exposants (de -1022 à 1023) et 52 bits pour la mantisse, soit 64 bits au total. En double précision, on privilégie la partie mantisse (le nombre de chiffres significatifs) plutôt que la partie exposant (l'amplitude des valeurs représentables).

En pratique, la plupart des compilateurs utilisent les mêmes plages de valeurs pour les mêmes types, et ces plages dépendent plus de l'architecture de la machine (taille d'un « mot » en nombres d'octets) que du fournisseur du compilateur.

2 On remarque qu'on a perdu une valeur dans l'affaire ! Dans de nombreuses implémentations, la plage de valeurs est en fait `-128 .. + 127`.

Conversions entre entiers et flottants :

Il existe une « hiérarchie » entre les types numériques et une notion de « promotion » d'un type simple vers un type plus élaboré (en termes de plages de valeurs) pour les calculs :

```
char, short → int  → unsigned long → double
float → double → long double
```

- Pour une **expression** arithmétique **combinant des opérandes de classes de nombres distinctes,** l'évaluation de chaque opérateur donne un résultat dont le type est **celui de l'opérande de type le plus élevé** dans la hiérarchie. Ainsi, une addition d'un `char` avec un `int` se fera en tant que `int` .Il en est de même quand deux quantités sont comparées via les opérateurs classiques : `<`, `<=`, `>`, `>=`, `==`, `!=`.

- Dans une **affectation**, le type du résultat de l'évaluation de la partie droite est **ramené** vers le type de la partie gauche de l'affectation (extension ou au contraire troncation vers 0 des flottants vers les entiers). **Il peut donc y avoir perte d'information**.

Attention aux dépassements de plages de valeurs et aux arrondis pour les flottants ! La précision finale (le dépassement de capacité) dépend du maillon faible de la chaîne de calculs.

2. Les booléens

C'est le type des « valeurs de vérité », c'est-à-dire « *vrai* » et « *faux* ». En C++ cela donne donc le type `bool` et ses deux constantes `true` et `false`.

Un lourd héritage issu des temps obscurs de l'informatique et du langage C fait cependant que toute valeur entière peut être interprétée comme « valeur de vérité », en assimilant `0` à `false` et toute entier non nul à `true`. L'impression par l'opérateur `<<` d'une valeur booléenne s'affiche d'ailleurs sous la forme de `1` ou `0` selon la valeur. Dans la suite de ces notes de cours, on utilise les booléens explicitement, pour faciliter la lecture des exemples.

En C, on peut utiliser le type `bool` (via l'inclusion de `stdbool.h`) à partir de la norme ISO C99.

3. Les caractères :

Les caractères sont vus comme un sous-type d'entiers : on interprète les « valeurs » des caractères comme leur rang dans un **alphabet implicitement ordonné**. Il existe de très nombreux alphabets dont certains utilisent plus d'un octet par symbole et ne correspondent pas au type `char` de C++. Parmi les plus connus, citons ASCII, EBDIC, iso8859-1, dit aussi iso-latin-1, windows-1252 et les plus récents UTF-8 et autres variantes d'Unicode. Le but de ces alphabets récents est de permettre de représenter les nombreux jeux de caractères qui existent sur la planète (ainsi que certains symboles de nature plus iconique) et pas seulement la grosse centaine de lettres présentes dans les alphabets des langues latines. Unicode normalise ainsi la représentation de plus de 110 000 symboles.

En C++ il existe aussi le type `wchar_t` (pour « *wide char* », sur 4 octets) qui permet de représenter un alphabet plus étendu que l'ASCII. Nous ne nous en servirons pas dans ce cours.

Les constantes littérales pour les caractères s'écrivent comme dans **`'a', 'Z'`** ou **`'3'`**, c.à.d. **un** caractère isolé entre deux signes « apostrophes ».[3]

Dans la suite nous utilisons le **code ASCII,** originellement de 128 caractères, étendu ultérieurement à 256 caractères pour y inclure des caractères dits « semi-graphiques » ; l'encodage n'est cependant pas standardisé pour les codes au-dessus de 127. En ASCII, les minuscules sont classées consécutivement par ordre alphabétique, de même que les majuscules. Les symboles qui représentent les chiffres sont aussi classés consécutivement selon l'ordre naturel (de `'0'` à `'9'`). On peut passer d'un de ces caractères à un autre par un décalage d'un nombre entier de positions `'a' + 3` → `'d'` ou calculer la distance entre deux caractères `'5' – '1'` → `4`. En ASCII, les chiffres viennent avant les majuscules qui viennent avant les minuscules, **mais** ces trois plages de valeurs ne sont **pas** contiguës. On se sert rarement de la « valeur » d'un caractère[4], mais plutôt de **leur ordre relatif dans l'alphabet**, comme dans les exemples ci-dessus. Il existe aussi des fonctions de bibliothèque (comme `toupper, tolower, isalpha, isalnum, isspace`) qui permettent de s'affranchir partiellement de ces problèmes d'alphabet. Pour utiliser ces fonctions, il faut inclure le fichier d'« en-têtes » `<cctype>` pour que leurs en-têtes soient connus du compilateur.

1. Écrire une fonction `atoi` qui prend en entrée une chaîne de caractères contenant la représentation en base 10 d'un nombre et qui renvoie la valeur de ce nombre. On suppose que la chaîne n'est pas vide, ne contient pas d'autres symboles que des chiffres (pas de signe !), que l'alphabet est l'ASCII et que la valeur du nombre tient dans un `int`. L'en-tête de la fonction est : `int atoi(string s);`

```
atoi("128") → 128
atoi("8") → 8
```

2. Écrire une fonction `char lower(char lettre)` : si le paramètre est une majuscule, la fonction renvoie la minuscule correspondante, sinon elle renvoie le paramètre lui-même. On suppose que l'alphabet est l'ASCII.

```
lower('C') → 'c',
lower('a') → 'a'
lower('+') → '+'
```

Ce n'est pas parce que ça compile que c'est correct !

```
'c' * 2 → ?
'C' + 'A' → ?
```

Puisque `char` est un sous-type de `int`, les expressions ci-dessus sont correctes (la multiplication et l'addition sont permises !), le compilateur n'a pas de raison de protester. Ça ne donne pas pour autant un sens à ces expressions si on les interprète comme des « caractères » et non pas comme des « entiers de petite valeur ». Ici, sachant qu'en ASCII `'c'` vaut `99`, `'c' * 2` est un caractère semi-graphique (code au-delà de 127) et en tant que « petit entier signé » il représente un nombre négatif !

3 On fera attention à distinguer un entier comme `3`, du symbole qui la désigne dans l'alphabet (*exercice* : cherchez sa valeur en ASCII !). On différenciera aussi le caractère `'3'` de `"3"`, la chaîne de caractères réduite à cet unique caractère, de même qu'on ne confond pas un élément avec un ensemble réduit à cet élément. !

4 Pourquoi écrire le code de `'A'` (65 en ASCII) quand on peut écrire `'A'`? Laissons le compilateur gérer l'encodage !

Notations spéciales : Certains caractères bénéficient d'une représentation spéciale: `'\n'`, `'\r'`, `'\t'`, … Il existe aussi deux notations utiles : `'\''`(apostrophe) et `'\\'`(« anti-slash »). Pour le caractère « guillemet » on peut utiliser deux notations au choix : `'\"'` ou `'"'`. Enfin, le caractère `'\0'` (de code ASCII 0, à ne pas confondre avec `'0'`) sert de « marqueur de fin » dans les chaînes de caractères dans le langage C et est utile à connaître pour utiliser certaines librairies écrites en C.

4. les chaînes de caractères (`string`[5])

Contrairement à son prédécesseur le langage C, le langage C++ définit (via l'en-tête `<string>`) un type « chaîne de caractères ». C'est en fait un « conteneur » de caractères et il dispose donc des opérations sur les conteneurs, dont les plus utiles sont :

- `s.size()` (ou `s.length()`) : la taille de la chaîne `s`, de type `size_t`, donc non signé
- `s.at(i)` : accès au caractère d'indice `i` de la chaîne `s`, en vérifiant la validité de `i`.
- `s[i]` : similaire à `s.at(i)` **sauf** que la validité de l'indice n'est **pas** contrôlée. Si la valeur de l'indice n'est pas correcte, le comportement est indéfini[6].
- `s.push_back(c)` : ajoute la caractère `c` en queue de `s`
- `s.pop_back()`: supprime le dernier caractère de `s`, réduisant donc sa taille de 1
- `s1 + s2` : renvoie une **nouvelle** chaîne qui est la concaténation des deux chaînes `s1` et `s2`. Aucune des deux chaînes `s1` et `s2` n'est modifiée.
- `s1.append(s2)` : ajoute les éléments de `s2` à la fin de `s1`, dans l'ordre. Modifie donc `s1`.

Les indices commencent à `0`. Une boucle de parcours d'une chaîne `s` a en général la forme suivante :

```
for(size_t i = 0; i < s.size(); i = i+1) { … }
```[7]

On peut aussi procéder à des affectations entre chaînes de caractères, tester leur égalité, tester deux chaînes selon l'ordre lexicographique (ordre sur les caractères étendu à un ordre sur les mots, comme dans le dictionnaire). Il existe d'autres possibilités, notamment pour supprimer ou remplacer des portions de chaînes ailleurs qu'aux extrémités mais elles sont d'usage un peu plus complexe en passant par une notion « d'itérateur ». Nous les verrons via des exemples, si besoin.

Exemples de déclarations et d'initialisations:

```
string s;                    // initialisée à la chaine vide
string s2 = "Hello world"; // ou aussi string s2("Hello world");
string s4(s2, 2, 5);         // sous-chaîne s2[2]..s2[6]. Ici "llo w"
string s5(10, 'x');          // 10 exemplaires de 'x': "xxxxxxxxxx"
string s5 = { 'a', 'b', 'c' }; // "abc"
```

Certaines de ces initialisations ne sont valables qu'en C++11, pas dans les versions plus anciennes. Certaines syntaxes peu intuitives sont données ici principalement parce qu'on les retrouvera aussi pour d'autres conteneurs.

---

5 Son « vrai » nom est `std::string`, sauf si on a invoqué la phrase magique `using namespace std;`

6 Dans un premier temps, nous privilégierons la notation `s.at(i)` pour que d'éventuelles erreurs d'indices nous soient explicitement signalées et provoquent une erreur à l'exécution. Cela aidera à avoir les bons réflexes pour raisonner sur les indices et évitera que des erreurs subsistent silencieusement et induisent une erreur lors d'une démo !

7 Attention au parcours à rebours. *Exercice* : indiquez pourquoi la boucle ci-dessous est incorrecte (le compilateur indiquant d'ailleurs un « warning») : `for(size_t i = s.size()-1; i >= 0; i=i-1) { … }`

Notation à utiliser : appel de fonction, de méthode ou d'opérateur ?

En C++, `string` correspond à ce qu'on appelle une « classe ». Une classe définit un nom de type mais aussi des « **méthodes** », c'est à dire des opérations qu'on peut appliquer aux éléments de la classe. La syntaxe pour appeler une méthode a la forme suivante : *élémernt.nomMethode(arguments)*. L'élément sur lequel on veut appliquer l'opération est distingué et est mis « devant » le nom de la méthode et les autres arguments. Par exemple :

```
string s = "Hello world";
char c = s.at(0);                /* trois appels de méthodes */
size_t j = s.size();
s.push_back('a');
```

En C++ on peut aussi définir des **fonctions** en donnant le nom de la fonction, le nom et le type de ses arguments, ainsi le type de valeur retournée en résultat. La syntaxe d'appel est alors de la forme suivante : *nomFonction(arguments).*

```
void copieDans(string s1, string s2) {
  for(size_t i = 0; i < s2.size(); i = i+1) {
      s1.push_back(s2.at(i));
  }
}

string s = "Hello world";
string s2 = "Hi everybody";
copieDans(s1, s2);               /* syntaxe d'appel de fonction */
int i = s1.size();               /* syntaxe d'appel de méthode */
```

Enfin, certaines opérations ont une forme d'appel encore différente puisqu'on leur nom est un symbole d'opérateur (arithmétique, de comparaison, ou autre comme `<<`) avec une syntaxe dite « infixe » : l'opérateur est placé entre ses opérandes. Il en est ainsi pour l'opérateur `+` de `string` qui réalise la concaténation de chaînes de caractères, où pour les opérateurs d'entrée-sortie `>>` et `<<` disponibles pour imprimer différents types de valeur avec les « stream ».

```
void imprime (string s) {
   cout << "la chaine vaut: " << s << endl;
}
```

Dans ce cours, vous n'aurez à définir vous-mêmes que des fonctions, jamais des méthodes ou des opérateurs. Par contre, vous aurez souvent à écrire des appels de méthodes ou d'opérateurs pour effectuer une opération sur un élément d'une classe. Il convient de comprendre les différences entre ces formes d'appel, pour utiliser la syntaxe d'appel qui convient selon le statut de l'opération que vous voulez appeler.

5. les « vecteurs » (la classe `std::vector` pour être précis !)

Ils font partie des « conteneurs de données » définies dans la librairie associée au langage C++. Ils peuvent contenir un nombre variable d'éléments, **tous de même type**. On précise le type des éléments stockés dans la déclaration du vecteur. La taille du vecteur peut varier au cours de l'exécution du programme. C'est leur intérêt principal par rapport au type « tableau simple » que nous allons voir dans la prochaine section.

```
#include <vector>

vector<int> myIntVector(10)[8]; // vecteur de int initialisé avec 10 occurrences de 0
vector<int> v2(4, 0);         // vecteur initialisé avec 4 occurrences de la valeur 0
vector<double> myDoubleVector(10, 1.5); // vecteur de double
vector<int> v1;               // vecteur de taille 0
vector<int> v3 = { 1, 3, 5, 7 };
vector<vector<int>> vv  = { v1, v2, v3 };
```

Pour les vecteurs, on dispose du même type d'opérations que celles décrites sur `string`, à l'exception de `length`, `append`, de l'opérateur `'+'` et des opérateurs de relation d'ordre (l'opérateur `<` et ses collègues) : comme le type des éléments du vecteur n'est pas forcément muni d'une relation d'ordre, on ne peut pas disposer au niveau des vecteurs un ordre lexicographique basé sur celle-ci. De même, on ne dispose pas de l'opérateur `<<` pour imprimer le contenu d'un vecteur et il faut définir une fonction explicite d'impression.

6. les tableaux « simples »

Ils correspondent à des conteneurs **homogènes** et **de taille fixe** d'éléments qu'on référence via un indice. Les indices vont de 0 à la dimension du tableau - 1. La dimension d'un tableau doit être précisée par une expression **statique** (c.à.d. la valeur doit être explicite dans le texte du programme et connue du compilateur). La différence avec les vecteurs est donc que **leur taille est fixe** et doit être connue à l'avance. Il ne s'agit pas d'une classe et on ne dispose pas de méthodes comme `at` et `size` ni d'opérations comme l'affectation ou le test d'égalité entre tableaux. Pour l'exemple ci-dessous, puisque la taille est fixe, pas la peine de passer par un vecteur.

```
int jours[12] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
jours[1] = 29;  /* en 2020 */
```

Quelques exemples pour illustrer la notion « d'expression statique » pour dimensionner un tableau:

```
void f(int n) {
  int t[n];                 /* KO ! La dimension de t n'est pas « statique » */
  for(size_t i = 0; i < n; i=i+1)  { t[i] = 0; }
}

void g(int n) {
  const int max = n;        /* max est bien une constante, mais pas statique*/
  int t[max];               /* KO ! La dimension de t n'est pas « statique » */
  for(size_t i = 0; i < n; i=i+1)  { t[i] = 0; }
}
```

8 Attention, cela suppose qu'il existe une valeur par défaut pour le type des éléments. Si ce n'est pas le cas, on obtient des messages d'erreur «bizarres» à la compilation. La déclaration suivante fournit explicitement la valeur à utiliser.

```
#define MAX 10
void h() {
   int t2[MAX*2];            /* OK ! MAX*2 est une expression « statique » */
   for(i = 0; i < MAX*2; i=i+1)  { t2[i] = 0; }
}
```

À la différence des vecteurs, on ne dispose pas d'une fonction pour récupérer à l'exécution la dimension du tableau à partir de son nom. Il faut soit passer cette information via un paramètre d'une fonction ou une variable, soit se reposer sur une convention (une valeur donnée dans le tableau sert de mrqueur de fin) pour retrouver à l'exécution de cette information. Pour un tableau déclaré **en dehors** d'une fonction, on obtient sa taille grâce à l'opérateur `sizeof`. Le résultat est **incorrect** pour un **paramètre** de type « tableau », parce qu'un tableau passé en argument n'est pas copié : on passe juste son adresse de début; c'est ce mécanisme qui permet de modifier le contenu d'un tableau passé en argument alors qu'on ne le peut pas pour des paramètres d'autres types. À noter que dans ce cas, le compilateur `gcc` vous prévient par un warning.

```
int tab[10] = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };

void incorrect(int t[]) {      /* t: paramètre de type tableau */
  cout << "sizeof(t) dans incorrect: " << sizeof(t) << endl;
  for(size_t i = 0; i < sizeof(t); i=i+1) { t[i] = 10; }
}

/*  Ici, on passe la dimension du tableau via un paramètre supplémentaire. Ce  paramètre
 *  représente la dimension du tableau ou une sous-partie d'un tableau plus grand
 */
void correct(int t[], size_t dim) {
  for(size_t i = 0; i < dim; i=i+1) { t[i] = 20; }
}

void imprime(int t[], size_t dim) {
  for(size_t i = 0; i < dim; i=i+1) { cout << t[i] << " "; }
  cout << endl;
}

int main() {
  cout << "sizeof(tab) dans main: " << sizeof(tab) << endl;
  incorrect(tab); imprime(t, 10);
  correct(tab, 10); imprime(t, 10);
  return 0;
}
```

donne les impressions suivantes dans notre implémentation:

```
sizeof(tab) dans main: 40      /* 4 * 10 i.e. (sizeof(int) * 10) */
sizeof(t) comme parametre: 8   /* 8 = taille d'un « pointeur » */
10 10 10 10 10 10 10 10 0 0    /* deux éléments non initialisés  */
20 20 20 20 20 20 20 20 20 20
```

Question : que se serait-il passé àl'exécution de `incorrect`, si `tab` avait été de dimension 2 ?

Dimensionnement à partir de la valeur initiale : un tableau déclaré **en dehors** d'une fonction peut être initialisé dans sa déclaration et sa dimension déduite de cette valeur initiale (ce qui évite de se tromper) **si** la dimension n'a pas été fournie explicitement. La plupart des compilateurs autorisent aussi de telles initialisations pour des tableaux définis dans des fonctions même si cette facilité n'est pas dans la norme. Ci-dessous, la dimension du tableau est déduite par le compilateur :

```
int jours[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
```

Cependant, si on donne une valeur initiale **incomplète** pour un tableau dont la dimension a été fournie explicitement, le tableau ne sera que partiellement initialisé :

```
int t[10] = { 0 };
```

n'initialise que `t[0]` à `0` et laisse les autres cases du tableau non initialisées.

Ne pas confondre « **initialisation** » (le contexte est celui d'une **déclaration**) et « **affectation** » (qui est une **instruction**), malgré la syntaxe proche (utilisation du symbole `=` dans les deux cas). Ainsi,

```
int t[] = { -1, 3, 5 };    /* déclaration avec initialisation */
```

est parfaitement correct alors que la séquence suivante ne l'est pas :

```
int t[3];
t = { -1, 3, 5 };          /* affectation globale sur t: incorrect ! */
```

On ne peut **pas** faire d'affectation ni de comparaison globale sur un tableau puisque le compilateur ne mémorise pas l'information sur la taille du tableau. On doit procéder à ces opérations case par case... Autant de raisons pour préférer les « vecteurs » aux tableaux, sauf si on est sûr de ce qu'on fait ou qu'on a de vrais problèmes d'efficacité (les vecteurs sont plus coûteux en termes de performances).

Valeur incorrecte d'indice: avec la notation `t[i]`, où `t` est un vecteur, une string ou un tableau simple, il est « **illégal** » pour `i` d'avoir une valeur en dehors de la dimension de `t`, mais en C++ **aucun contrôle de validité des indices n'est effectué** (contrairement à d'autres langages où ces accès illégaux sont détectés et signalés à l'exécution, ou bien à la fonction `at` définie pour les chaînes et les vecteurs). Le comportement dans un tel cas est **non spécifié** : cette erreur de programmation peut entraîner une erreur immédiate, ou accéder à une zone mémoire sans effet désastreux visible, ou avoir un effet (écrasement de la valeur d'une autre variable) qui ne sera détecté qu'à un instant dans le temps arbitrairement éloigné. De même, ce comportement dépend de l'implémentation, donc cette erreur peut se révéler différemment selon le compilateur, ou le système d'exploitation, la machine... **Aucune garantie** de comportement n'est assurée pour un accès illégal. C'est une faiblesse du langage C++ par rapport à des langages récents. **C'est à vous de garantir par votre technique de programmation que ce cas n'arrive jamais !**

Tableaux multidimensionnels : on les représente par des tableaux de tableaux:

```
int mat[10][20];           /* accès: mat[i][j], pas mat[i,j] */
```

Ci-dessus `mat[0]` est un tableau de 200 entiers stockés dans `mat[0][0]`, …, `mat[9][19]`. Attention à ne pas écrire

```
int t[10, 20];             /* piège !*/
```

qui est syntaxiquement correct mais ne ne représente pas ce qu'on croit (voir l'opérateur « virgule » plus loin dans ces notes de cours).Terminons avec un exemple d'initialisation d'un tableau de tableaux:

```
int tab[2][3] = { { 1, 2, 3 }, { 4, 5, 6 } };
```

7. les enregistrements (« `struct` »)

Ils correspondent à des « produits cartésiens » dans lesquels on donne un nom à chaque composant. Les composants peuvent avoir des types différents, par opposition aux tableaux dont toutes les cases ont des valeurs de même type. On accède à un composant en précisant le nom du champ (ou « attribut ») de l'objet sous la forme `objet.nom`. Typiquement, pour représenter une date on préférera un enregistrement à 3 champs entiers, auxquels on pourra associer des noms, plutôt qu'un tableau de 3 entiers qui obligerait à se souvenir de quel indice correspond à quelle partie de la date.

```
struct Date {
  unsigned char jour;
  unsigned char mois;
  short annee;
};

Date today = { 15, 02, 2018 };
Date tomorrow;
tomorrow = today;              // affectation entre structures
tomorrow.jour = today.jour+1; // ne gère pas les fins de mois et d'année !
```

On peut faire des **affectations globales** entre **instances de structures** (le compilateur décompose une telle affectation en autant d'affectations champ-à-champ qu'il est nécessaire), mais **on ne peut pas comparer** globalement (via `==` ou `!=` ou `<=`, etc.) de telles instances. Les structures peuvent être passées en argument à des fonctions (par valeur ou par référence) ou renvoyer en résultat.

8. Le type `void`:

Il s'agit du type « sans valeurs ». On ne peut pas déclarer de variables de ce type. Il sert principalement[9] à indiquer qu'une fonction ne renvoie pas de résultat (ce qu'on appellerait une procédure dans d'autres langages) ou ne prend pas de paramètre en entrée. La plupart des compilateurs autorise qu'on écrive une déclaration d'une fonction sans paramètre (renvoyant par exemple un `int`) sous la forme `int f()` là où formellement on devrait écrire `int f(void)`.

Si une fonction a été définie comme ayant un type de retour `void`, elle ne doit **jamais** renvoyer de valeur (puisque le code qui l'appelle suppose qu'elle ne retourne pas de valeur). Si au contraire une fonction a été déclarée comme retournant autre chose que `void`, elle doit **dans tous les cas** renvoyer une valeur, résultat que le code qui l'a appelée s'attend à recevoir. Si le compilateur détecte un cas pour lequel votre fonction pourrait ne pas renvoyer de valeur, il vous le signale par un « warning ».

9 On reverra aussi ce type ultérieurement, en liaison avec les « pointeurs ».

II - Expressions et Instructions

*Une **expression** correspond à un calcul et a une notion de **résultat** alors qu'une **instruction** n'a pas de notion de résultat mais travaille par« effet de bord » (modification de l'état du système: changement de la valeur d'une variable, impression d'un message, etc). En C++ comme en C, une expression suivie d'un `';'` a le statut d'une instruction : elle est évaluée et son résultat est simplement ignoré. C'est notamment le cas pour un appel de fonction dont on souhaite ignorer le résultat retourné (ou qui a été déclarée comme retournant `void`). Il est important de comprendre la distinction entre ces deux notions, car certaines constructions du langage C++ requièrent une **expression**, alors que d'autres requièrent une **instruction**. Cela vous évitera d'essayer de contourner des messages d'erreur du compilateur par ajout ou suppression, au petit bonheur la chance, de `';'` qui peuvent dans certains cas changer la structure, et souvent le sens, de votre programme.*

Opérateurs arithmétiques:

sur les entiers : `+`, `-`, `*`, `/` (division entière), `%` (modulo)
sur les flottants: `+`, `-`, `*`, `/` (division flottante)

Les opérateurs + et – existent aussi sous forme unaire préfixe, comme dans `– x`. On verra ultérieurement que c'est aussi le cas du symbole `*`, mais avec un sens très différent de la multiplication (déréférencement de pointeurs). Il n'existe **pas** en C++ d'opérateur « élévation à la puissance », contrairement à ce que certains étudiants font semblant de croire lors des examens !

Il existe deux opérateurs particuliers: la **pré/post-incrémentation** et **pré/post-décrémentation**. Nous illustrons ci-dessous le fonctionnement de l'incrémentation, la décrémentation étant similaire, en remplaçant `++` par `--` ! Soient `x`, `y` deux variables entières, `y = x++` consiste à prendre la valeur courante de `x`, la mémoriser temporairement pour la suite du calcul (ici l'affectation à `y`) **puis** incrémenter `x`. A l'inverse, `y = ++x` consiste à incrémenter `x` **avant** d'en prendre la nouvelle valeur pour la suite du calcul. Il y a « **post**-incrémentation » dans le premier cas, « **pré**-incrémentation » dans le second. Bien sûr, on peut préférer écrire à la place les fragments de code plus classiques `y = x; x = x+1` dans le premier cas, et `x = x+1; y = x` dans le second cas. Ces opérateurs donnent simplement une écriture plus compacte, utile notamment quand une construction englobante requiert syntaxiquement une expression (la valeur résultat de l'opérateur ++ peut être utilisée pour la suite du calcul) ou n'autorise qu'une unique instruction.

***Attention**:* on quitte ici le champ des expressions au sens mathématique, puisque ces deux opérateurs introduisent des « effets de bord ». Deux évaluations successives de la même expression (par exemple `x++`) donnent des résultats différents, ce qui complique le raisonnement sur de tels programmes: par exemple la formule `x++ == x++` s'évalue à « faux », alors que pour toute expression `e` on s'attendrait à ce que l'égalité `e == e` vaut « vrai ». On fera aussi attention à ce que `x++` n'est **pas** un « raccourci » pour `x+1` puisque dans le premier cas la valeur de `x` est modifiée ![10]

***Opérateurs de comparaison* :** `==`, `!=`, `<`, `<=`, `>`, `>=`

Ces opérateurs renvoient `false` ou `true`. Ils peuvent se combiner mais pas avec le sens habituel : la notation classique `x <= y <= z` compile mais correspond à `(x <= y) <= z`. Par exemple, si `z`

10 Piège : que fait l'instruction `x = x++;` trouvée dans certaines copies d'examen ?

vaut `1`, cette expression vaut toujours « vrai »[11] ! On a le même phénomène avec des expressions telles que `x == y == z`. Ne pas utiliser vos réflexes mathématiques habituels et écrire plutôt `x == y and y == z` par exemple.
Avec l'option `-Wall` le compilateur `gcc` vous met cependant en garde contre ces erreurs classiques :

warning: suggest parentheses around comparison in operand of '=='

Opérateur d'affectation **:** On rappelle que, hélas, le symbole `=` représente l'affectation et non pas l'égalité qui est notée `==` en C++. À ce titre, l'opérateur `=` n'est donc pas « symétrique » : `x = y` est fondamentalement différent de `y = x`. Pour éviter des déconvenues, habituez-vous à prononcer l'opérateur `=` comme « reçoit » plutôt que comme « égal » : on ne s'attend pas à ce que « reçoit » soit symétrique !

L'affectation est une expression, pas une instruction[12] : l'affectation produit un résultat (la valeur de l'opérande droit). En général on ignore ce résultat mais il est parfois utile, soit « en cascade » pour initialiser un ensemble de variables avec la même valeur, soit pour simplifier l'écriture de boucles :

```
int a, b, c;
a = b = c = 12;        /* se lit : a = (b = (c = 12)) */
```

Exemple : copie du contenu d'un tableau d'entiers `t1` dans un tableau `t2` jusqu'au premier `0` (inclus). On suppose que `t1` contient bien un `0` et que `t2` est de taille suffisante.

```
i = 0; while ((t2[i] = t1[i]) != 0) { i = i+1; }
```

La boucle s'arrête après avoir copié le `0` de `t1`. En fin de boucle, `i` est l'indice de la case qui contient le `0`. Ici, on peut même écrire :

```
i = 0; while (t2[i] = t1[i]) { i = i+1; }
```
[13]

On peut préférer à ce programme le programme plus classique ci-dessous qui n'utilise pas la valeur renvoyée par l'affectation :

```
i = 0;
while (t1[i] != 0) { t2[i] = t1[i]; i = i+1;[14] }
t2[i] = 0;      /* ne pas oublier l'affectation du 0 final dans cette version  */
```

ou encore le fragment de programme suivant :

```
size_t i = 0; int c;
do { c = t1[i]; t2[i] = c; i = i+1; } while(c != 0);
```

qui est moins naturel et qui laisse l'indice `i` **après** le dernier entier copié (le `0`).

---

11 Pourquoi ?

12 Mais `x = y;` est bien une **instruction** à cause du `;` qui suit l'expression d'affectation.

13 Bienvenue au club des programmeurs qui compliquent inutilement la vie de leurs relecteurs et qui prennent des risques inutiles. Si on ne fait pas attention, ce code semble indiquer un comportement **très** différent:-(

14 Et non pas `while(t1[i] != 0) { t2[i] = t1[i++]; }` dont le comportement est indéfini puisque la norme ne précise pas quelle sera la valeur de `i` utilisée pour accéder à `t2` : tout dépend si on calcule l'adresse de la case mémoire associée à `t1[i++]`avant ou après celle correspondant à `t2[i]`.

Attention à ne pas confondre **égalité (==)** et **affectation (=)**, comme dans l'exemple suivant dans lequel on a confondu `=` et `==` dans le test de l'instruction conditionnelle. Considérons un appel tel que `errone(3)` : l'affectation à de `0` à `i` a lieu et c'est la valeur renvoyée par l'affectation, donc `0`, qui est utilisée pour le test. Celui-ci est donc interprété comme « faux » et on passe dans la branche `else`, tout en ayant mis `i` à `0` au passage ! On obtient deux messages apparemment « contradictoires ».

```
void errone(int i) {
   if (i = 0) {                          /* erreur: '=' au lieu de '==' */
      cout << "i est nul\n";
   } else { cout << "i n'est pas nul\n"; }
   cout << "Valeur de i: " << i;                /* va imprimer 0 */
}
```

Si vous faites l'effort de lire les messages du compilateur, vous êtes explicitement mis en garde :

```
warning: suggest parentheses around assignment used as truth value
```

Pour la plupart des opérateurs arithmétiques, il existe une variante qui combine cet opérateur et l'affectation. Par exemple `x += 3;` est équivalent à `x = x + 3;`[15]

Opérateurs logiques: and (&&), or (||) et not (!)

Les opérateurs logiques `and` et `or` sont dits « séquentiels » ou « paresseux » : ils n'évaluent leur second opérande que si le résultat de l'évaluation du premier ne suffit pas à avoir la valeur finale, selon les tables de vérité classiques. Ces deux opérateurs ne sont donc **pas** commutatifs contrairement à leurs homologues de la logique classique ($\wedge$ et $\vee$). Les opérateurs `&&`, `||` et `!` sont les équivalents sous forme de symboles des opérateurs `and`, `or` et `not`. Préférez les versions nommées pour minimiser le risque d'erreur. En langage C, seules les versions sous forme de symboles existent.

Exemple (déterminer si une année est bissextile): une année bissextile est divisible par 4 et non divisible par 100, sauf si dans ce cas elle est divisible par 400

```
bool bissextile (int annee) {
   return (annee % 4 == 0)
           and ((annee % 100 != 0) or (annee % 400 == 0));
}
```

Cette fonction est équivalente à la fonction ci-dessous dans laquelle on a remplacé les opérateurs logiques par des combinaisons d'instructions conditionnelles:

```
bool bissextileBis (int annee) {
  if (annee % 4 == 0) {
    if (annee % 100 != 0) {
       return true;
    } else { return (annee % 400 == 0); }
  } else { return false; }
}
```

L'aspect « séquentiel » de l'opérateur `and` est surtout utile quand deux conditions sont à tester mais que la seconde ne doit l'être que si la première est vérifiée, par exemple pour parcourir un tableau :

15 L'intérêt n'est pas juste une notation plus concise mais d'imposer au compilateur de ne calculer qu'une fois l'adresse de la cible de l'affectation. Considérez par exemple une expression telle que `t[x*2][y*3] += 3`.

`/*` renvoie l'indice de la première occurrence de `val` dans `v.` Si `val` n'apparaît pas dans `v`, renvoie la taille de `v` (qui n'est donc pas un indice correct).`*/`

```
size_t indiceDans(vector<int> v, int val) {
   size_t i = 0;
   while (i < v.size() and v.at(i) != val) { i=i+1; }
   return i;
}
```

Intervertir les deux opérandes du `and` rendrait le programme illégal : si la valeur recherchée est absente on accèderait à `v.at(v.size())` ce qui est illégal.

Opérateurs bit-à-bit : ils attendront le cours de S4. Attention cependant à ne pas confondre ces opérateurs `|` et & avec `||` et && et à ne pas les utiliser par erreur. À chaque fois le premier élément est l'opérateur bit-à-bit alors que le second est l'opérateur logique. Pour ne pas vous tromper, utilisez donc `or` et `and` pour les opérateurs logiques.

Opérateur conditionnel . La syntaxe de cet opérateur ternaire est :

condition `?` *cas-vrai* `:` *cas-faux*

Cet opérateur s'utilise à la place de **l'instruction** conditionnelle (`if`) quand on a besoin d'une **expression**, donc d'une notion de valeur :

```
int valeurAbsolue(int v) { return (v >= 0 ? v : -v);[16] }
```

Cette fonction est équivalente à

```
int valeurAbsolueBis(int v) {
  if (v >= 0) { return v; } else { return -v; }
}
```

Cette forme de conditionnelle n'est utile que comme opérande d'une expression plus compliquée.

Appels de fonctions : les appels à des fonctions dont le type de retour n'est pas `void` peuvent bien sûr apparaître comme opérandes d'une expression plus complexe. En particulier, ils peuvent être des arguments dans un autre appel de fonction:

```
(s1+s2).size() * 2,
(-b + sqrt(pow(b, 2) – 4 * a *c)) / (2 * a)
```

Le compilateur doit connaître les prototypes (ou « en-têtes ») des fonctions avant de les appeler. Soit parce qu'on les a explicitement déclarées auparavant, soit via l'inclusion d'un fichier d'en-tête qui contient ces prototypes de fonctions.

Nous reviendrons plus en détail sur la mécanique des appels de fonctions par la suite.

16 Saurez-vous trouver l'erreur dans cette fonction (et sa version alternative) ?

Opérateur « , » : à n'utiliser que dans de rares cas ! L'expression `e1, e2` évalue d'abord `e1`, ignore son résultat puis évalue `e2` et renvoie son résultat[17]. La construction n'a d'intérêt que si on évalue `e1` pour autre chose que son résultat, c.à.d. si `e1` a un « effet de bord », par exemple parce que `e1` est une affectation. L'utilisation la plus classique est celui d'une boucle manipulant plusieurs compteurs à initialiser en début de boucle (et à mettre à jour en fin de boucle) alors que la syntaxe du `for` n'autorise qu'**une** expression d'initialisation et **une** expression de mise à jour.

Dans cette première version, on utilise deux indices: l'un parcourt la chaîne de gauche à droite, l'autre de droite à gauche. On peut arrêter les comparaisons dès que les indices se croisent (ou sont égaux dans le cas d'une chaîne de longueur paire) ou que les caractères « en miroir » sont différents.

```
bool estPalindrome(string s) {
  int i, j;[18]
  for(i = 0, j = s.size()-1; i < j; i = i+1, j = j-1) {
     if (s.at(i) != s.at(j)) { return false; }
  }
  return true;
}
```

L'opérateur `','` permet de considérer syntaxiquement `i=i+1, j=j-1` comme une unique expression dont on utilise les « effets de bord ». De même pour `i=0, j=s.size()-1`.

Rien n'empêche d'utiliser une version avec une boucle `while` :

```
bool estPalindrome2(string s) {
  int i = 0, j = s.size()-1;
  while(i < j)  {
     if (s.at(i) != s.at(j)) { return false; }
     else { i = i+1; j = j-1; }
  }
  return true;
}
```

Les fonctions précédentes traitent leurs indices de façon symétrique et sont plus simples à comprendre que la version avec un seul indice. Le code ci-après est d'ailleurs erroné : trouvez pourquoi et donnez en une version correcte:

```
bool estPalindromeErronee(string s) {  /* Incorrect ! */
  int i = 0, imax = s.size()-1;
  int imilieu = imax / 2;
  for(i = 0; i <= imilieu; i = i+1) {      /* pourquoi <= ? */
     if (s.at(i) != s.at(imax-i)) { return false; }
  }
  return true;
}
```

Attention : ne pas utiliser l'opérateur `','` pour la partie condition d'une boucle : il ne fait ni un « et », ni un « ou » puisqu'il ignore complètement la valeur de l'expression de gauche.

17 Ce qui explique pourquoi `t[e1,e2]` compile mais ne fait pas ce qu'on croit et ne correspond pas à `t[e1][e2]`.

18 On passe par `int` et non pas par `size_t` car `j` deviendrait« négatif » pour la chaîne vide, ce qui est gênant pour un type « `unsigned` » comme `size_t`. On pourrait aussi traiter à part la chaîne vide.

Priorité, associativité des opérateurs en C et problèmes associés:

les opérateurs sont listés par **priorités décroissantes**. L'associativité des opérateurs binaires apparaît dans la colonne de droite.

Opérateurs « primaires » : `() [] -> .` *gauche à droite*
`.` *est relatif aux « structures »,* `->` *aux structures pointées*

Opérateurs unaires : `! - + ++ -- sizeof * &`
Ce sont ici les versions ***unaires*** *de* `+` *et* `-`
Le `*` *unaire est le déréférencement de pointeur,*
`&` *l'opérateur « adresse de... »*
`not` *(ou* `!`*) est la négation logique,*

Opérateurs **binaires** multiplicatifs: `* / %` *gauche à droite*

Opérateurs **binaires** additifs: `+ -` *gauche à droite*

Opérateurs de comparaison: `< <= > >=` *gauche à droite*

Opérateurs d'égalité : `== !=` *gauche à droite*

« Et » logique : `and` (&&) *gauche à droite*[19]

« Ou » Logique : `or` (||) *gauche à droite*

Opérateur ternaire conditionnel : `_?_:_`

Les opérateurs d'affectation : `= += *=` ... *droite à gauche*

Opérateur virgule : `,` *gauche à droite*

Exemples de mise en œuvre des priorités sur des fragments de programmes :

```
4*5-6-3                          → 11
4*5-(6-3)                        → 17
4*4 % 9/3   → (16%9)/3 → 7/3     → 2
4*4 % (9/3) → 16 % 3             → 1
12 + 3 * 5 % 2 + 3               →
b = x + y == z + t and t <= w    →
```

Les indications de priorité et associativité servent à indiquer comment est **structurée** une expression : quel opérateur s'applique à quels opérandes. Elles ne disent rien sur **l'ordre d'évaluation** des opérandes d'une expression binaire à l'exécution, ordre qui n'est **pas** défini dans le langage (i.e. ça dépend de l'implémentation) **sauf** pour les opérateurs logiques `and` et `or`. On fera attention à ce que le résultat d'une évaluation ne dépende pas d'un ordre particulier, ce qui peut être le cas si une expression a des effets de bord : nous le montrons sur deux exemples ci-dessous. Les exemples sont caricaturaux pour être concis, mais illustrent bien le problème. Dans les deux cas `x` vaut `5` à la fin mais on n'a aucune garantie sur la valeur de `x` utilisée pour le numérateur :-(

```
int x = 4;
x / ++x   peut valoir soit 1 (on évalue le dénominateur, puis le numérateur: 5/5 → 1)
                    soit 0 (on évalue le numérateur, puis le dénominateur: 4/5 → 0)
```

19 L'opérateur `and` est donc prioritaire sur l'opérateur `or`. Cependant pour faciliter la lecture des programmes et minimiser le risque d'erreur il est vivement conseillé de parenthéser les expressions comportant plusieurs `and` ou `or`.

Pour les appels de fonctions, **l'ordre d'évaluation des arguments est non précisé** par le langage:

```
int x = 4;
```
`f(x, x++)` peut valoir soit `f(5, 4)` si les arguments sont évalués de droite à gauche,
soit `f(4, 4)` s'ils sont évalués de gauche à droite.

Dans les deux cas `x` vaudra `5` à la fin.

Des programmes qui dépendent d'un ordre particulier d'évaluation sont **illégaux** car non portables (ils dépendent du choix fait dans le compilateur). On **évitera absolument** d'écrire de tels programmes. Si votre résultat peut dépendre d'un ordre particulier d'évaluation, la solution consiste à ordonnancer les évaluations en passant par des affectations à des variables auxiliaires, pas à deviner comment tel compilateur traite ce problème : un autre compilateur pourrait faire le choix inverse, ou une optimisation pourrait changer l'ordre des calculs.

Le problème de contrôler l'ordre des évaluations ne se pose pas uniquement avec des opérateurs comme `++`, mais généralement avec toute expression susceptible de provoquer un « effet de bord », comme par exemple avec `f(a)+f(b)` avec une fonction `f` qui ne fait pas que renvoyer un résultat mais qui imprime aussi un message. Le message imprimé va dépendre de l'ordre dans lequel on évalué les deux opérandes de l'addition ;

Le problème se pose aussi pour maîtriser d'éventuels débordements des plages de valeurs numériques. Les numériciens savent qu'en informatique `a+(b+c)` n'est pas toujours égal à`(a+b)+c` puisque ces deux expressions peuvent se comporter différemment en termes de dépassement de capacité ou de perte de précision (pour les flottants). Il ne s'agit pas là d'un simple problème de programmation mais d'un problème fondamental de maîtrise des erreurs d'arrondis et de dépassement de capacité en calcul scientifique, qui dépassent très largement l'objet de ces « rappels » de C++.

Les instructions simples ou composées

Rappel : toute **expression** suivie de `';'` a la valeur syntaxique d'une instruction : on évalue l'expression et on oublie son résultat !

le passage en séquence : exécution dans l'ordre de chacune des instructions de la séquence.

le bloc :
```
{ déclarations éventuelles de variables locales au bloc
  liste d'instructions du bloc
}
```
Un bloc sert à grouper une liste d'instructions pour en faire une construction qui, syntaxiquement, a la valeur d'une instruction unique qui peut apparaître dans un corps de boucle ou dans une conditionnelle. Un bloc peut aussi être vide comme dans `{ }` pour compléter explicitement un corps de boucle ou une partie `else`.

Un bloc peut introduire des **variables locales** qui ne sont visibles que dans les instructions du bloc. Les variables locales à un bloc masquent les variables homonymes déclarées dans les constructions englobant le bloc; Elles ne sont plus visibles une fois qu'on a quitté ce bloc. L'intérêt de déclarer des variables dans un bloc plutôt qu'au tout début du corps d'une fonction, est d'une part de faciliter la compréhension du programme en ne forçant pas le lecteur à prendre en compte des variables qui n'ont d'intérêt que dans une partie restreinte de la fonction, d'autre part d'avoir plus de chance de savoir les

initialiser avec une valeur pertinente, plutôt qu'au tout début de la fonction.Le langage C++ n'oblige pas à regrouper les déclarations de variables en tête du bloc. Dispersez les déclarations au milieu des instructions ne simplifie cependant pas la lecture du programme. Plutôt à éviter, sauf bonnes raisons !

L'instruction **« conditionnelle »** :

`if (`*condition*`)` *Instruction*

ou `if (`*condition*`)`*Instruction* `else` *Instruction*

Rappel : on a le droit à une **unique** instruction mais qui peut être un « bloc » au sens ci-dessus, c'est-à-dire une liste d'instructions entre accolades. On rappelle aussi que la partie `else` est facultative et se rapporte toujours au dernier `if` incomplet.

Exemple avec indentation trompeuse:

```
if (cond1)
   if (cond2) instr1
else instr2
```

La partie `else` se rapporte à `cond2` et non pas à `cond1` malgré l'indentation trompeuse. Pour éviter toute ambiguïté, il est **vivement** conseillé de toujours mettre tant la partie « `then` » que la partie `else` entre accolades et d'utiliser une indentation qui facilite la lecture. Ci-dessous, le second `if` est inclus dans un bloc; la partie `else` étant en dehors de ce bloc, il n'y a pas d'ambiguïté possible.

```
if (cond1) {
   if (cond2) {
       instr1
   }
} else { instr2 }
```

Facilitez la compréhension de votre code par une personne qui devra éventuellement le relire (pour le mettre au point, l'adapter, l'étendre, voire le noter !), peu importe si cela prend quelques caractères de plus à taper. Dans la «vraie vie», un programme est plus souvent lu qu'il n'est écrit. Il n'y a que pendant ses études qu'on se désintéresse d'un programme dès qu'il est au point :-)

Les boucles:

`for(`*initialisation*`;` *condition d'itération*`;` *incrément*`)` *une_instruction*

`while(`*condition d'itération*`)` *une_instruction*

`do` *une_instruction* `while` (*condition d'itération*`);`

On prendra garde au fait qu'on a le droit à **une unique** instruction dans le corps des boucles, donc on mettra en général un bloc pour préciser la partie de programme qui doit être répétée (corps de la boucle). Attention aux `';'` qui séparent les trois parties de l'en-tête d'une boucle `for`. La boucle `for` la plus courte s'écrit `for(;;);` ou `for(;;){}` et ne fait rien un nombre infini de fois. Ce programme boucle donc. Son seul « intérêt » est de montrer que chacune des parties est facultative et que les trois composants de l'en-tête de boucle sont des expressions.

Rappelons que dans une boucle `for`, on peut déclarer l'indice de parcours localement à la boucle mais celui-ci ne sera plus visible après la boucle. Ci-dessous, on ne saurait donc pas pour quelle raison on est sorti de la boucle :

```
for(size_t i = 0; i < s.size() and s.at(i) != v; i = i+1) { ... }
```

Les trois types de boucles ont des comportements différents quant à la mise à jour des compteurs, ou la propriété de toujours passer au moins une fois dans le corps de boucle ou pas. On prendra garde à bien choisir le type de boucle qui convient le mieux à chaque cas. La boucle `do_while` est souvent utilisée à tort à la place d'une boucle `while` dans des cas où il faudrait pouvoir de faire aucun passage dans le corps de boucle (parcourt d'une chaîne vide, d'un tableau vide).

À l'intérieur d'une boucle, on a le droit aux instructions de « rupture du flot de contrôle » :

`continue` : passe directement à l'itération suivante de la boucle courante, en effectuant au préalable la mise à jour des compteurs de boucle dans le cas d'une boucle `for`.

`break` : quitte la boucle courante (et uniquement celle-là).

L'instruction **`return;`** ou **`return`** *expression*`;` permet de quitter immédiatement la **fonction** courante, en renvoyant le résultat de l'évaluation de l'expression dans le second cas. C'est une erreur qu'une fonction déclarant un type de retour autre que `void` atteigne la fin de son corps sans avoir exécuté un `return` avec une valeur (le compilateur émet un avertissement)`.` La valeur de retour est alors indéfinie et si le résultat est utilisé dans une expression (par exemple `3 + f(5)` où `f` est supposée renvoyer un entier) le résultat global est arbitraire.

L'usage de ces trois dernières constructions est parfois controversé, certaines méthodologies de programmation recommandant qu'une fonction n'ait toujours qu'un seul `return`, situé à la fin du bloc principal de la fonction, et que toute boucle soit toujours quittée uniquement via sa condition d'itération, donc sans `break` ou `continue`, quitte à introduire des variables booléennes pour contrôler l'enchaînement des instructions. Tous les langages modernes ont maintenant de telles instructions de sortie « raisonnée ». La seule règle qu'on se donnera ici consistera à se demander si le code est plus clair (plus simple à comprendre, à prouver correct) à l'aide de ces trois instructions, ou au contraire en introduisant des variables booléennes de contrôle. Il n'y a pas de réponse unique.

L'instruction **`switch`**

```
switch (expression) {
   case cste1 : liste d'instructions
   case cste2 : liste d'instructions
   ...
   case csteN : liste d'instructions
  /* la partie ci-dessous est facultative */
   default :  liste d'instructions
}
```

Les *cste* associées aux `case` sont des constantes **statiques** de type `int` (ou assimilées). On ne peut pas mettre d'intervalles de valeurs :-(Les listes d'instructions peuvent être vides, ce qui permet d'associer un même traitement à une liste de cas consécutifs. Le fonctionnement est le suivant:

- On évalue (une seule fois) l'expression de contrôle du `switch`, ce qui fournit une valeur
- Si un cas a pour étiquette la valeur obtenue, on exécute la liste d'instructions associée à ce cas. **Si** sa liste d'instructions finit par un `break` on passe à l'instruction qui suit le `switch`, **sinon** on continue en séquence avec la liste d'instruction du cas suivant ! On continue donc jusqu'à tomber sur un `break` ou la fin du `switch`.

- Lorsque la valeur de l'expression ne correspond à aucun cas : si la clause `default` est présente, elle sert d'étiquette pour tous les cas qui ne sont pas listés par ailleurs et se comporte comme eux. Si elle est absente, on quitte le `switch`. Cette partie `default`, si elle est présente, est normalement listée en dernier car ses instructions ne sont pas supposés s'appliquer à un autre cas. Le programme est aussi plus simple à comprendre et moins sujet à erreur si on étatit plus tard amener à ajouter des cas.

Les passages par les instructions associées aux différents cas s'enchaînent donc tant que leurs listes d'instructions ne contiennent pas une instruction `break`. En l'absence de `break`, l'ordre des différents cas importe et on n'exécute pas seulement les instructions associées au cas correspondant à la valeur de l'expression du `switch`.

```
void compteEspaces(string s) {
   int nbEspaces=0, nbLignes = 0, nbCarac=0;
   for(size_t i = 0; i < s.size(); i=i+1) {
      switch(s[i]) {
         /* pour '\n'  on incrémente a la fois nbLignes et nbEspaces
          * pour ' '  ou '\t'  on incrémente seulement nbEspaces
          * Dans tous les cas, on compte un caractere
          */
         case '\n' : nbLignes = nbLignes+1;
         case ' '  :
         case '\t' : nbEspaces = nbEspaces+1;
         default   : nbCarac =  nbCarac+1 ;
      }
   }
   cout << "Nombre d'espaces: " << nbEspaces << " dont "
        << nbLignes << " fins de ligne."
        << " Nombre de caracteres: " << nbCarac << endl;
}
```

L'ajout d'un `break;` après `nbEspaces = nbEspaces+1;` ferait que `nbCarac` ne compterait que les caractères distincts des « espaces ».

Attention à ne pas confondre le rôle de `break` à l'intérieur d'un `switch` et le rôle de `break` à l'intérieur d'une boucle ! L'instruction `break` associée ci-dessus au cas `'\r'` permet de prendre en compte ce caractère pour l'ignorer mais ne quitte pas la boucle.

Un dernier exemple : le squelette d'une fonction pour gérer un menu.

```
void menu() {
  bool encore = true;
  do {
     char c;
     cout << "Entrez l'action désirée:";
     cin >> c;
     switch (c) {
        case ' ':
        case '\n':
        case '\t': continue; /* itération suivante du do while */
        case 'q':
        case 'Q':  encore = false; /* on va devoir quitter la boucle */
                   break;      /* break du switch, pas du do while ! */
        case 'r':
        case 'R':  /* actions associées à 'r' et 'R' à mettre ici ... */
                   break;
        case 'w':
        case 'W':  /* actions associées à 'w' et 'W' à mettre ici ... */
                   break;
        default: /* pour toute autre valeur de c ... */
          cerr << "Choix errone: " << c << "\nReessayer!\n";
     } /* fin du switch */
  } while (encore);
}
```

Ici aussi il ne faut pas se tromper sur le rôle du **`break`** associé aux cas `'q'` et `'Q'`: l'usage de la variable booléenne (`encore`) de contrôle de la boucle est indispensable. Le sens de chaque instruction `break` est relative à la construction englobante (boucle ou `switch`) la plus proche.

III – Autres constructions de types

1. La directive `typedef` :

`typedef` *définition-de-type-existant nom*;

L'identificateur *nom* peut alors être utilisé dans des déclarations de variables, de paramètres, etc, comme s'il s'agissait d'un type de base.

```
typedef int Table[10];
Table t;                /* t est un tableau de 10 int */
```

Notamment utile en liaison avec les constructions qui vont suivre pour s'abstraire (partiellement) des types de base du langage et être plus proche du vocabulaire de l'application, en nommant les objets comme ils doivent l'être. D'autres exemples apparaissent ci-dessous.

2. Les énumérations :

Elles servent à pouvoir définir un type avec des valeurs nommées dans une liste finie de constantes.

```
enum Couleur { Bleu, Blanc, Rouge };
Couleur c1 = Blanc, c2 = Rouge;
```

De façon interne ces valeurs sont représentées par des entiers. On peut même donner des valeurs précises aux constantes déclarées dans l'énumération, mais nous ne nous servirons pas en général de cette possibilité. À noter qu'en C++, contrairement au langage C, le compilateur n'accepte pas de confusion entre le type énuméré et le type entier qui sert à le représenter. On ne peut mettre dans une variable `Couleur` qu'une valeur de l'énumération.

```
c1 = 2;                 // KO. Ne compile pas :-)
c1 = c1 + c2;           // KO, ne compile pas : + renvoie un entier, pas une couleur
int i = c1 + c2;        // Compile.
```

On voit que la cloison entre `Couleur` et entiers n'est pas entièrement étanche. Si vous imprimez une valeur de l'énumération, vous obtenez d'ailleurs sa représentation sous forme d'entier.

3. Les unions (types avec variantes)

Nous présentons ci-dessous un exemple de représentation (simple) de figures géométriques, telles qu'elles pourraient être manipulées dans un éditeur de dessins. Les figures sont en 2D et chaque sorte de figure a ses propres caractéristiques. On suppose que toute figure a un point de référence: pour un cercle ça pourrait être son centre, pour un rectangle (disposé parallèlement aux axes) le coin inférieur gauche. Pour un cercle on aura besoin de son rayon, pour un rectangle de sa longueur et de sa largeur... Il faudra être capable d'ajouter ultérieurement de nouveaux types de figures (segment, triangle, losange, polygone arbitraire ?)

Nous commençons par représenter la notion de « point » (dans un espace à deux dimensions)

```
typedef struct { double x, y; } Point2D;
Point2D origine = { 0.0, 0.0 };
```

Puis une énumération avec les types de figures actuellement définis :

```
typedef enum { Cercle, Rectangle } TypeFigure;
```

Enfin, nous donnons la représentation des figures :

```
typedef struct {
  TypeFigure type;          /* pour mémoriser le type de figure actuellement stocké ! */
  Point2D origine;          /* commun a toutes les figures */
  double rayon;             /* pertinent pour les cercles */
  double largeur, longueur; /* pertinent pour les rectangles */
} Figure;
```

Dans cette première version très basique les attributs correspondent **à la réunion** de tous les attributs nécessaires pour au moins un des types de figures. Facile à programmer mais coûteux et peu sûr : pour un objet (par exemple un cercle) on conserve des attributs qui n'ont aucun sens pour lui (par ex. la longueur et la largeur). Pour savoir si une variable stocke actuellement les caractéristiques d'un cercle ou d'un rectangle, on a prévu le champ `type` : selon sa valeur, on sait qu'on doit s'intéresser soit au champ `rayon`, soit aux champs `largeur` et `longueur`. D'autres défauts sont incontournables :

- pas de cohérence entre le type apparent de la figure et l'usage qu'on en fait : pour la même figure on peut s'intéresser à la fois à l'attribut `rayon` et à l'attribut `largeur` puisque ceux-ci sont tous présents bien qu'il soient logiquement incompatibles !
- même si on construit correctement les objets, n'importe qui peut modifier l'attribut `type` indépendamment des autres attributs;

Bref, une version sans aucune sûreté de programmation et coûteux en place mémoire :-(

```
double perimetre(Figure fig) { /* fonction dont le calcul dépend du type de figure */
  switch(fig.type) {
    case Cercle :    return 2 * 3.14 * fig.rayon;
    case Rectangle:  return 2*(fig.largeur + fig.longueur);
    default: cerr << "Type de figure inconnu: " << fig.type << endl;
             return -1.0; /* valeur ne pouvant correspondre à une longueur */
  }
}

void imprimePoint (Point2D p) {
  cout << "Coordonnées du point de reference: ("
       << p.x << ", " << p.y << ")\n";
}

void imprime(Figure fig) {
  imprimePoint(fig.origine);
  switch (fig.type) {
    case Cercle:  cout << " Cercle - Rayon : " << fig.rayon << endl;
                  break;
    case Rectangle: cout << "Rectangle - Largeur: " << fig.largeur
                         << ", longueur: " << fig.longueur << endl;
                    break;
    default: cerr << "Type de figure inconnu: " << fig.type << endl;
  }
}
```

```
int main() {              /* un programme de démonstration */
  Point2D p = { 1.0, 1.0 }; Figure monCercle, monRect;

  monCercle.origine = p; monCercle.type = Cercle;
  monCercle.rayon = 3.0;

  monRect.origine = p; monRect.type = Rectangle;
  monRect.largeur = 1.0; monRect.longueur = 2.0;

  cout << "Perimetre du cercle: " << perimetre(monCercle) << endl;
  cout << "Perimetre du rectangle: " << perimetre(monRect) << endl;
  imprime(monCercle); imprime(monRect);
  return 0;
}
```

***Exercice**: ajouter un nouveau type de figure: le segment, caractérisé par ses deux points extrêmes.*

Les types `union` correspondent au besoin précédent où on voudrait avoir le choix entre plusieurs possibilités: les informations à représenter diffèrent selon ce qu'on doit représenter. Nous utilisons directement cette construction en liaison avec un `typedef` :

```
typedef union { /*  union avec trois possibilités * /
   double d;
   char c;
   int i;
} MonType;

MonType v;      /*  variable de ce type union */
```

L'utilisation d'une `union` pour `v` permet de stocker un `double` **ou** un `char` **ou** un `int` (à la différence d'une `struct` dans laquelle ces « **ou** » sont des « **et** »). Le compilateur s'arrange pour réserver un espace mémoire qui convient à n'importe laquelle de ces possibilités. **À chaque usage** de `v` on doit préciser comment interpréter l'objet, en suffixant le nom de la variable par le nom de la variante à utiliser: `v.d` interprète la valeur dans `v` comme un `double`, `v.c` comme un `char` et `v.i` comme un `int`. Le type `union` ne permet pas à lui seul de savoir, à un moment donné de l'exécution, quelle est la « bonne » interprétation pour un objet de ce type. Cette information doit venir par ailleurs, par exemple parce que cette `union` fait partie d'une structure dans laquelle un champ mémorise la variante qui est actuellement représentée dans l'objet.

Dans l'exemple des figures, on ne garde qu'un ensemble cohérent d'attributs : un rayon **ou** une largeur et une longueur. Dans la représentation d'une figure, une partie fixe regroupe les informations pertinentes pour toutes les figures et une partie correspond aux informations pertinentes pour chaque type de figure pris séparément. Comme précédemment le champ `type` permet de mémoriser le type d'objet actuellement représenté:

```
typedef struct { /* sans changement */
  double x, y;
} Point2D;

typedef enum { Cercle, Rectangle } TypeFigure;
```

```
typedef struct {              /* attributs propres aux cercles */
  double rayon;
} AttCercle;

typedef struct {              /* attributs propres aux rectangles */
  double largeur, longueur;
} AttRectangle;

/* la partie « variante » : on est soit de type AttRectangle, soit de type AttCercle */
typedef union {
  AttRectangle rect;
  AttCercle cercle;
} AttSpecifiques;

typedef struct figures {
  Point2D origine;       /* commun a toutes les figures */
  TypeFigure type;       /* memorise le type de figure actuellement stocke */
  AttSpecifiques att;    /* isole la partie specifique */
} Figure;
```

***Avantages* :**

- chaque objet ne prend que la place nécessaire pour le **plus complexe** des objets à représenter (ici cela correspond aux rectangles qui ont deux attributs de type `double` pour la largeur et la longueur, alors qu'un cercle n'a besoin que d'un `double` pour son rayon).
- Si on déclare accéder à une figure comme étant un cercle, on n'a plus accès à des attributs pertinents pour les rectangles, et vice-versa. Donc une meilleure protection de programmation.

***Inconvénients*:**

- La notation est plus lourde syntaxiquement pour accéder aux champs spécifiques : sélectionner le champ qui correspond à l'union, puis préciser la variante considérée, puis sélectionner le champ qui nous intéresse vraiment : **`f.att.cercle.rayon`**
- Aucune garantie du compilateur quant à un usage cohérent des attributs : pas de cohérence entre la valeur de l'attribut `type` et l'usage fait des champs.

Cependant, rien n'empêche d'en faire un usage raisonnable comme ci-dessous:

```
double perimetre(Figure f) {
  switch(f.type) {
     case Cercle :     return (2*3.14*f.att.cercle.rayon);
     case Rectangle:
       return(2*(f.att.rect.largeur + f.att.rect.longueur));
     default: cerr << "Figure inconnue !\n"; return -1.0;
  }
}
```

***Exercice**: ajouter un nouveau type de figure: le segment, caractérisé par ses deux points extrêmes.*