




Dans ce chapitre nous discutons de l'une des caractéristiques les plus puissantes et le plus difficiles à maîtriser du langage C++ : le *pointeur*. La manipulation de pointeurs est ce qui distingue principalement le C++ des autres langages tels que *Java* ou *C#*. En effet, ces langages ne supportent pas la manipulation de pointeurs, caractéristique essentielle au développement d'applications embarqués (tels que la programmation d'appareils électroménagers, de véhicules motorisés ou d'appareils médicaux). C'est pourquoi le langage C++ est si populaire dans l'industrie du développement d'applications industrielles et commerciales.




Objectifs

- Utiliser les pointeurs
 - Pour passer des arguments à des fonctions en *appel par référence via pointeurs*
- Comprendre les relations entre les pointeurs, les tableaux et les chaînes
 - Déclarer et utiliser des tableaux pour chaînes de caractères

Marco Lavoie 14278 ORD - Langage C++

Comme nous l'avons vu au chapitre 3, nous pouvons utiliser des *références* (opérateur `&`) pour effectuer des appels par référence. Les pointeurs permettent aux programmes de simuler ce genre d'appels et de créer et manipuler des structures de données dynamiques ou, plus précisément, des structures de données capables de grossir et de rétrécir comme les listes chaînées, les queue, les piles et les arborescences (techniques de programmation que vous étudierez aux étapes ultérieures de TGI).



Aperçu

- Déclaration de variables pointeurs
- Opérateurs de pointeurs
- Appels de fonctions par référence via pointeurs
- Arithmétique des pointeurs
- Relation entre les pointeurs et les tableaux
 - Tableaux de pointeurs
- Introduction au traitement de caractères et de chaînes
 - Principes de base
 - Bibliothèque de manipulation de chaînes de caractères

Marco Lavoie 14278 ORD - Langage C++

Dans ce chapitre, nous expliquons et analysons les concepts de base des pointeurs ainsi que les relations intimes entre les tableaux, les pointeurs et les chaînes de caractères.

Nous verrons l'emploi des pointeurs avec les structures au chapitre 6, tandis qu'aux chapitres 9 et 10 nous apprendrons que la programmation orientée objets s'effectue au moyen de pointeurs et de références.



Introduction

- Les **pointeurs** : une caractéristique
 - parmi les plus puissantes en programmation
 - parmi les plus difficiles à maîtriser
- Pas unique au C++
 - C, Pascal, assembleur, etc.
- Plusieurs cours de TGI exploitent les pointeurs
 - Classes et structures de données I et II
 - Programmation assembleur
 - Systèmes embarqués I et II
 - Protocoles de communication
 - Algorithmes d'analyse syntaxique

Marco Lavoie

14728 ORD - Langage C++

Comme mentionné précédemment, les pointeurs sont l'une des caractéristiques les plus puissantes et les plus difficiles à maîtriser du langage C++. Plusieurs langages tels que *Java* et *C#* exploitent intrinsèquement les pointeurs mais de façon cachée au programmeur, principalement car la gestion des pointeurs dans un programme est particulièrement difficile et occasionne souvent des bogues dans ceux-ci. Conséquemment ces langages de programmation « cachent » le concept de pointeurs aux programmeurs afin de lui éviter ces difficultés. Par contre, le langage C++ dévoile les pointeurs aux programmeurs, laissant à ces derniers le loisir de les manipuler dans leurs programmes. À cette liberté est cependant rattaché un niveau de difficulté accru dans la programmation d'applications complexes.

Hérité du langage C (l'ancêtre du C++), les pointeurs seront exploités intensivement dans d'autres cours de votre programme d'études, aux étapes 3 à 6.



déréférencement

- Les pointeurs implantent le principe de déréférencement
 - Une variable contient une valeur
 - Un pointeur est une variable contenant l'adresse d'une variable contenant une valeur

```
int v = 18;
int *pv = &v;
cout << v << ", " << pv;
```

- pv contient l'adresse de la variable v

v : 18
pv : 18, 0012FF60

18, 0012FF60

Marco Lavoie

14728 ORD - Langage C++

Les *variables pointeurs* contiennent des adresses mémoire. En règle générale, une variable conventionnelle contient directement une valeur spécifique (p.ex. la variable **v** contient la valeur **18**), tandis qu'une variable pointeur contient plutôt l'adresse d'une variable qui, elle, contient une valeur spécifique (p.ex. **pv** contient l'adresse de **v** qui elle contient **18**). Ainsi, un nom de variable référence directement une valeur, alors qu'un pointeur référence indirectement une valeur. Référencer une valeur via un pointeur s'appelle la *déréférencement*.

L'opérateur d'adresse & retourne l'adresse de la variable spécifiée (p.ex. **&v** retourne l'adresse du premier octet de mémoire alloué à **v**). Dans l'exemple ci-contre, cette adresse (i.e. adresse mémoire de **v** obtenue par **&v**) est stockée dans la variable pointeur **pv**. On dit que le pointeur **pv** « pointe » à **v**, tel qu'illustré par la flèche ci-contre.



Opérateur *

- Permet de
 - Déclarer une variable pointeur
 - Déréférencer au contenu de la variable pointée

```
int v = 18;
int *pv = &v;
cout << v << ", " << pv << ", " << *pv << endl;
*pv = 33;
cout << v << ", " << pv << ", " << *pv << endl;
```

18, 0012FF60, 18
33, 0012FF60, 33

Marco Lavoie

14728 ORD - Langage C++

L'*opérateur de déréférencement* * a deux fonctions :

1. Il sert à déclarer des variables pointeurs.
2. Il est à *déréférencer* un pointeur afin d'accéder à l'espace mémoire pointé.

Dans l'exemple ci-contre, l'opérateur * est utilisé une première fois pour déclarer la variable pointeur **pv** : **pv** est une variable pointeur (**int *pv;**) destinée à stocker l'adresse d'une valeur entière (**int *pv;**).

La seconde utilisation de l'opérateur * exploite la déréférence pour modifier le contenu du bloc mémoire alloué à la variable **v** via l'adresse de ce bloc stockée dans **pv**. Ainsi, les deux affectations **v = 33** et ***pv = 33** sont équivalentes puisqu'elles font référence au même bloc mémoire.



Relation entre * et &

7

- Les opérateurs * et & sont l'inverse l'un de l'autre

```
int v = 18, *pv = &v;
cout << "L'adresse de v est " << &v << endl;
<< "La valeur de pv est " << *pv << endl;
cout << "La valeur de v est " << v << endl;
<< "La valeur de *pv est " << *pv << endl;
cout << "La valeur de &*pv est " << &*pv << endl;
<< "La valeur de *pv est " << *pv << endl;
```

```
L'adresse de v est 0012FF60
La valeur de pv est 0012FF60
La valeur de v est 18
La valeur de *pv est 18
La valeur de &*pv est 0012FF60
La valeur de *pv est 0012FF60
```

Marco Lavoie

14728 ORD - Langage C++

Il y a une relation intrinsèque entre l'opérateur d'adresse & et celui de déréférencement *: l'opérateur d'adresse sert à obtenir l'adresse d'un bloc mémoire (tel que celui alloué par le compilateur à la variable **v**) et l'opérateur de déréférencement sert à référencer le contenu d'un bloc mémoire via son adresse (tel que l'accès au contenu de la variable **v** via son adresse).

Observez l'affichage produit par les deux dernières instructions de l'exemple ci-contre : les opérateurs & et * sont l'inverse l'un de l'autre puisque le même résultat (l'adresse stockée dans **pv**) apparaît lorsqu'on les applique tous deux consécutivement à **pv**.



Priorité des opérateurs

8

- Incluant tous les opérateurs vus à date, en ordre décroissant de priorité

Opérateurs	Associativité
() []	De gauche à droite
static_cast<type>() ++ -- (versions suffixe)	De droite à gauche
** -- + - (versions préfixe) ! & *	De gauche à droite
/ % *	De gauche à droite
+ -	De gauche à droite
<< >>	De gauche à droite
< <= > >=	De gauche à droite
== !=	De gauche à droite
&&	De gauche à droite
	De gauche à droite
?:	De droite à gauche
= += -= *= /= %=	De droite à gauche

Marco Lavoie

14728 ORD - Langage C++

Le tableau ci-contre illustre le niveau de priorité et l'associativité des opérateurs vus à date. Attention à ne pas confondre l'opérateur de déréférencement * avec l'opérateur arithmétique de multiplication *, qui exploitent le même symbole (l'étoile) mais ont un niveau de priorité distinct.

Il est important de noter que l'opérateur de déréférencement dispose d'un niveau de priorité plus élevé que celui des opérateurs arithmétiques. Ainsi, l'expression ***pv + 1** est équivalente à **(*pv) + 1**, et non **(pv + 1)**. Cette distinction sera importante lorsque nous verrons plus loin l'arithmétique des pointeurs.



Relation entre * et & (suite)

9

- Ne pas confondre la *déréférencement par pointeur* avec la *déréférencement par référence*

```
int r = 1, &pr = r;
int v = 2, *pv = &v;

pr = pr * 10;
*pv = *pv * 10;

cout << r << ", " << &r << ", " << pr << ", " << &pr << endl;
cout << v << ", " << &v << ", " << pv << ", " << &pv << endl;
```

```
10, 0012FF60, 10, 0012FF60
20, 0012FF48, 0012FF48, 012FF3C
```

Marco Lavoie

14728 ORD - Langage C++

Les programmeurs débutants en C++ confondent parfois les *variables alias* (déclarés avec l'opérateur d'adresse &) et les *variables pointeurs* (déclarées avec l'opérateur de déréférencement *), puisque toutes deux servent généralement à référencer le contenu d'une autre variable.

Dans l'exemple ci-contre, la variable alias **pr** et la variable pointeur **pv** référencent toutes deux le contenu d'une autre variable (**pr** référence le contenu de **r** et **pv** référence le contenu de **v**). Cependant la variable alias **pr** ne nécessite aucun autre opérateur pour accéder au contenu de **r**, alors que la variable pointeur **pv** requiert l'opérateur de déréférencement * pour accéder au contenu de **v**.

La variable alias **pr** ne requiert aucun autre opérateur pour accéder au contenu de **r** car, en fait, **pr** et **r** font référence au même bloc mémoire (adresse **0012FF60**) alors que **v** et **pv** sont deux variables distinctes, telle qu'illustrée dans l'affichage ci-contre.



Paramètres pointeurs

10

- Alternative aux paramètres de référence
 - Permet à une fonction de modifier le contenu des arguments d'invocation
- Rappel** : paramètres de référence

```
// Transmutation des arguments
void swap( int &x, int &y ) {
    int temp = x;
    x = y;
    y = temp;
}

int main() {
    int a = 10, b = 20;

    std::cout << setw( 3 );

    std::cout << a << b << std::endl;
    swap( a, b );
    std::cout << a << b << std::endl;

    return 0;
}
```

Marco Lavoie

14728 ORD - Langage C++

En C++, il existe trois façons de passer des arguments vers une fonction lors de l'invocation de celle-ci :

1. l'appel par valeur,
2. l'appel par référence via référence, et
3. l'appel par référence via pointeur.

Dans le chapitre trois nous avons présenté les deux premières façons.

L'exemple ci-contre illustre le passage d'arguments par appel par référence via référence : l'opérateur d'adresse **&** est utilisé pour déclarer les paramètres de la fonction **swap** comme étant des alias des arguments fournis lors de l'invocation de **swap** dans **main**. Ainsi, lorsque **swap** manipule le contenu des paramètres **x** et **y**, elle manipule en réalité le contenu des variables **a** et **b** de **main**.



Paramètres pointeurs (suite)

11

- Paramètre pointeur : reçoit l'adresse de l'argument correspondant
 - Permet ainsi de modifier le contenu de cet argument

```
// Transmutation des arguments
void swap( int *x, int *y ) {
    int temp = *x;
    *x = *y;
    *y = temp;
}

int main() {
    int a = 10, b = 20;

    std::cout << setw( 3 );

    std::cout << a << b << std::endl;
    swap( &a, &b );
    std::cout << a << b << std::endl;

    return 0;
}
```

Marco Lavoie

14728 ORD - Langage C++

L'appel par référence via pointeur exploite l'opérateur d'adresse **&** conjointement avec l'opérateur de déréférencement ***** pour faire en sorte que la fonction manipule le contenu des arguments. La version de la fonction **swap** présentée ci-contre déclare ses paramètres comme étant des pointeurs, c.à.d. des variables destinées à contenir l'adresse de d'autres variables. Conséquemment, les instructions du corps de la fonction **swap** exploitent l'opérateur de déréférencement ***** pour accéder au contenu des arguments **a** et **b** par déréférencement via leurs adresses respectives stockées dans **x** et **y**. De même, l'invocation dans **main** transmet à **swap** les adresses des variables **a** et **b**, qui seront respectivement affectées aux paramètres **x** et **y**.

La page suivante de la présentation illustre bien cette relation entre les paramètres pointeurs de **swap** et les arguments fournis par **main** lors de l'invocation.



Paramètres pointeurs (suite)

12

- Exécution pas à pas de l'exemple précédent

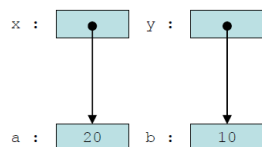
```
// Transmutation des arguments
void swap( int *x, int *y ) {
    int temp = *x;
    *x = *y;
    *y = temp;
}

int main() {
    int a = 10, b = 20;

    std::cout << setw( 3 );

    std::cout << a << b << std::endl;
    swap( &a, &b );
    std::cout << a << b << std::endl;

    return 0;
}
```



– Les paramètres **x** et **y** pointent vers les arguments correspondants **a** et **b**

Puisque le paramètre pointeur **x** contient l'adresse de l'argument **a**, et le paramètre pointeur **y** contient l'adresse de l'argument **b**, les instructions de **swap** manipulent les valeurs stockées dans **a** et **b** via ses paramètres pointeurs et l'opérateur de déréférencement *****. Ainsi, l'instruction ***x = *y;** dans **swap** extrait indirectement la valeur de **b** (**y** contenant l'adresse de **b**, ***y** faisant référence au bloc de mémoire alloué à **b**) pour l'attribuer à la variable **a** (**x** contient l'adresse de **a**, donc ***x** fait référence au bloc de mémoire alloué à **a**).

L'opérateur de déréférencement ***** peut être vu comme signifiant « suivre la flèche » dans l'illustration ci-contre. Ainsi, dans l'instruction suivante :

```
cout << x << *x;
```

La première valeur affichée est le contenu de **x** (soit l'adresse de **a**), alors que la seconde valeur affichée est 20, soit le contenu de la variable **a**, accédée indirectement via son adresse stockée dans **x**.



Paramètres pointeurs vs références

13

- Même fonctionnalité, mais attention aux distinctions dans le code source

Paramètres pointeurs

```
// Transmutation des arguments
void swap( int *x, int *y ) {
    int temp = *x;
    *x = *y;
    *y = temp;
}

int main() {
    int a = 10, b = 20;

    std::cout << setw( 3 );

    std::cout << a << b << std::endl;
    swap( &a, &b );
    std::cout << a << b << std::endl;

    return 0;
}
```

Paramètres références

```
// Transmutation des arguments
void swap( int &x, int &y ) {
    int temp = x;
    x = y;
    y = temp;
}

int main() {
    int a = 10, b = 20;

    std::cout << setw( 3 );

    std::cout << a << b << std::endl;
    swap( a, b );
    std::cout << a << b << std::endl;

    return 0;
}
```

Marco Lavoie

14728 ORD - Langage C++

Les deux exemples ci-contre sont deux versions d'un même programme. Le premier programme (celui de gauche) exploite l'appel par référence via pointeurs pour inverser le contenu des arguments `a` et `b` de `main`. La seconde version du programme (celui de droite) exploite l'appel par référence via références pour effectuer la même tâche.

Il est important de noter les distinctions dans ces deux versions du programme :

- Les paramètres pointeurs requièrent l'opérateur de déréférencement (`*`) dans le corps de la fonction pour manipuler le contenu des arguments `a` et `b`, et l'invocation doit fournir les adresses de ses arguments à la fonction (d'où l'opérateur d'adresse `&` accompagnant les arguments).
- Les paramètres références ne requièrent que l'utilisation de l'opérateur d'adresse (`&`) dans l'en-tête de la fonction afin de déclarer les paramètres comme des alias des arguments.



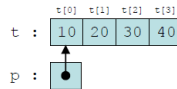
Arithmétique des pointeurs

14

- Une variable pointeur contient une adresse mémoire (d'une autre variable?)
 - On peut manipuler cette adresse comme si c'était une valeur entière

```
int t[] = { 10, 20, 30, 40 };
int *p = &t[ 0 ];

for ( int i = 0; i < 4; i++ ) {
    cout << *p << " ";
    p++;
}
```



- Puisque `p` est de type `int *`, `p++` incrémente l'adresse dans `p` de la taille d'un `int`

Marco Lavoie

14728 ORD - Langage C++

Les pointeurs étant des adresses mémoire, ce sont des opérandes valides dans des expressions arithmétiques, d'affectation et de comparaison. Toutefois, pas tous ces opérateurs ne sont applicables aux pointeurs.

Les opérateurs arithmétiques applicables aux pointeurs sont limités : on peut les incrémenter (`++`) ou les décrémenter (`--`), leur additionner (`+` ou `+=`) ou leur soustraire (`-` ou `-=`) un entier et leur soustraire un autre pointeur.

Supposons dans l'exemple ci-contre que le bloc mémoire associé au tableau `t` débute à l'adresse 3000. On aura donc sur une architecture 16 bits `&t[0]` = 3000, `&t[1]` = 3002, `&t[2]` = 3004, etc. Comme `p` est initialisé à 3000, l'expression `p++` incrémente l'adresse dans `p` non pas de un octet, mais de 2 octets puisque `p` est déclaré comme pointeur de type `int` (un entier étant stocké dans 2 octets dans une architecture 16 bits). Donc la boucle incrémente successivement `p` à 3002, 3003, 3006, etc.



Arithmétique des pointeurs (suite)

15

- Autre exemple :

```
const int N = 7;
int t[ N ] = { 10, 20, 30, 40, 50, 60, 70 };

for ( int *p = &t[ N-1 ]; p > &t[ 0 ]; p -= 2 )
    cout << *p << " ";
```

- Qu'affiche le code ci-dessus?

```
70 50 30
```

- `p` est décrémentée de 2 après chaque itération
- Aucune itération lorsque `p == &t[0]`

Marco Lavoie

14728 ORD - Langage C++

Dans l'exemple ci-contre, le pointeur `p` est initialisé à l'adresse du dernier élément du tableau `t`, et à chaque itération de la boucle `for` l'adresse dans `p` est décrémentée de 4 octets (puisque un entier est stocké sur deux octets et que `p` est un pointeur de type `int`, `p -= 2` correspondant à une décrément de 4 octets de l'adresse contenue dans `p`).



Restrictions de type

16

- Le type d'une variable pointeur doit correspondre au type de la variable pointée

```
int    v[10];
int    *p1 = &v[ 0 ]; ✓
double *p2 = &v[ 0 ]; ✗
```

- Cette restriction est imposée par l'arithmétique des pointeurs
 - Dans l'exemple ci-dessus, incrémenter p2 ne fait pas pointer au prochain élément de v, ce qui est le cas avec p1
- Cette restriction peut être contournée avec le transtypage explicite

```
double *p2 = reinterpret_cast< double * >( &v[ 0 ] );
```

Auteur : Marco Lavoie | Adaptation : Sébastien Bois

25908 IFM - Langage C++

Comme mentionné précédemment, une variable pointeur doit avoir un type qui correspond au type de valeurs pointées par celle-ci. Dans l'exemple ci-contre l'affectation de p2 à l'adresse du début du tableau v est invalide car le type de la variable pointeur (double) ne correspond pas au type de la valeur pointée (int).

Malgré le fait que le transtypage explicite permet de contourner cette restriction, il est très dangereux d'effectuer un tel transtypage dans un programme, au risque de causer une défaillance du programme à l'exécution. Généralement on évite de faire de tels transtypages en programmation, sauf dans des circonstances bien particulières et hors sujet de ce cours.



Relation entre pointeurs et tableaux

17

- Les tableaux et pointeurs sont intimement liés
 - Leur utilisation est presque interchangeable
- En effet, un tableau est considéré par le compilateur comme une variable pointeur
 - int t[] devient const int *t
 - L'opérateur [] est converti en arithmétique des pointeurs équivalente

Marco Lavoie

14728 ORD - Langage C++

En C++, l'utilisation des tableaux et des pointeurs est *presque* interchangeable. On peut se représenter le nom d'un tableau comme étant un pointeur constant initialisé à l'adresse du premier élément du tableau. En fait, on peut remplacer l'opérateur d'indexation [] dans une expression par l'arithmétique de pointeurs correspondant appliqué au nom du tableau, qui agit comme pointeur.



Pointeurs vs tableaux (suite)

18

- Exemples
 - Supposons les déclarations suivantes
- Les expressions suivantes sont équivalentes

```
int v[ 10 ], u;
int *p;

p = &v[ 0 ];      ➡ p = v;
p = &v[ 3 ];      ➡ p = v + 3;
u = v[ 2 ];       ➡ u = *( v + 2 );

v[ 5 ] = 17;      ➡ p = v;
                    p[5] = 17;

void f( int [] ); ➡ void f( int * );
```

Marco Lavoie

14728 ORD - Langage C++

Comme le démontre les exemples ci-contre, on peut traiter un tableau comme un pointeur et l'utiliser dans l'arithmétique des pointeurs. Par exemple, l'expression $\mathbf{*(v + 2)}$ renvoie à l'élément de tableau $\mathbf{v[2]}$. En général, toute expression d'un tableau comportant des indices peut s'écrire avec un pointeur et un *décalage* du contenu de ce dernier.

Réciproquement, les pointeurs peuvent être indexés de la même façon qu'un tableau. L'avant dernier exemple ci-contre fait référence au sixième élément de v via $\mathbf{p[5]}$, et ce même si p est non pas un tableau mais un pointeur contenant l'adresse du tableau v.

Comme le nom d'un tableau est essentiellement un pointeur constant, il est interdit de changer l'adresse correspondant au début du tableau. Ainsi l'expression $\mathbf{v += 2}$ serait interdite dans l'exemple ci-contre car elle tente de modifier l'adresse contenue dans v.

Pointeurs vs tableaux (suite)

19

- Exemples (suite)
 - Les programmeurs d'expérience interchangent souvent tableaux et pointeurs

```
const int N = 4;
int v[ N ] = { 10, 20, 30, 40 };

for ( int i = 0; i < N; i++ )
    v[ i ] = v[ i ] + 10;

for ( int *p = v; p < v + N; p++ )
    *p = *p + 10;
```

équivalents

Marco Lavoie

14728 ORD - Langage C++

Les deux boucles ci-contre sont équivalentes. La seconde boucle exploite un pointeur initialisé à l'adresse du tableau **v** pour parcourir les éléments de ce dernier. L'arithmétique des pointeurs permet de modifier l'adresse contenue dans **p** en l'incrémentant successivement, à chaque itération, afin de faire passer l'adresse qu'elle contient d'un élément de **v** au suivant.

Notez qu'on aurait aussi pu formuler la deuxième boucle **for** comme suit :

```
for ( int *p = v, int i = 0; i < N; i++)
    p[i] = p[i] + 10;
```

Comme mentionné précédemment, l'opérateur d'indexation (**[]**) et celui de déréférencement (*****, avec l'arithmétique des pointeurs) sont interchangeables dans la manipulation de tableaux comme des pointeurs ou vice-versa.

Exercice #5.1

20

- Sans faire appel à l'opérateur **[]** ni à une variable d'itération de type **int**, complétez le programme suivant de sorte qu'il affiche le nombre de valeurs paires trouvées dans le tableau
- Soumettez votre projet selon les indications de l'instructeur

```
int main() {
    const int N = 20;
    int valeurs[ N ];
    int compte = 0;

    // Remplir de valeurs aléatoires
    srand( time( 0 ) );
    for ( int i = 0; i < N; i++ ) {
        valeurs[ i ] = rand() % 100;
        std::cout << valeurs[ i ] << " ";
    }
    std::cout << std::endl;

    // INSÉREZ VOTRE CODE ICI

    std::cout << "# de valeurs paires: "
                << compte << std::endl;

    return 0;
}
```

Marco Lavoie

14728 ORD - Langage C++

Pour solutionner cet exercice, vous devez modifier la boucle **for** de telle sorte qu'elle manipule le tableau **valeurs** comme un pointeur, exploitant l'arithmétique des pointeurs et l'opérateur de déréférencement pour remplacer toutes références à l'opérateur d'indexation dans la boucle. L'abandon de l'opérateur d'indexation rendra conséquemment caduc l'emploi de la variable d'itération **i**.

Inspirez-vous de l'exemple mentionné à la page précédente pour convertir la boucle **for**.

Tableaux et chaînes de caractères

21

- Une chaîne de caractères est stockée dans un tableau

```
char nom[] = "Yvan Larue";
char *ville = "Verdun";
```

- Équivalentes à

```
char nom[] = { 'Y', 'v', 'a', 'n', ' ', 'L', 'a', 'r', 'u', 'e', '\0' };
char *ville = { 'V', 'e', 'r', 'd', 'u', 'n', '\0' };
```

- Les flux d'entrée/sortie peuvent manipuler des chaînes de caractères

```
char nom[30];
cin >> nom;
cout << nom;
```

Marco Lavoie

14728 ORD - Langage C++

Comme mentionné au chapitre 4, en C++ une chaîne est un tableau de caractères se terminant par le caractère nul (**'\0'**). On y accède par le biais du nom du tableau qui peut être considéré comme un pointeur constant vers le premier caractère de la chaîne. En ce sens, une chaîne est identique à un tableau puisqu'un nom de tableau est également un pointeur constant vers son premier élément.

On peut affecter directement une chaîne en provenance du clavier à un tableau en utilisant l'extraction de flux avec **cin**. Il est important de préciser que le tableau doit être assez grand pour stocker la chaîne entrée et son caractère nul de terminaison. Dans l'exemple ci-contre, la taille maximale de chaîne que peut fournir l'utilisateur pour **nom** est donc 29 caractères.



Affichage de chaînes de caractères

22

- Le flux de sortie `cout` affiche le contenu d'un tableau de `char` jusqu'au premier caractère nul ('`\0`') rencontré

```
char s[] = { 'a', 'b', 'c', '\0', 'd', 'e', 'f', '\0' };
cout << s;    // affiche abc
```

- Il faut donc s'assurer qu'un '`\0`' sera trouvé dans la chaîne à afficher

```
char s[] = { 'a', 'b', 'c' };
cout << s;    // affiche abc suivi de divers caractères
              // jusqu'à ce que 0 soit rencontré en
              // mémoire
```

Marco Lavoie

14728 ORD - Langage C++

Similairement au flux d'entrée `cin` qui peut lire une chaîne directement dans un tableau de caractères, le flux de sortie `cout` considère un tableau de caractères comme contenant une chaîne terminée par le caractère nul, et affiche donc les caractères contenus dans le tableau jusqu'au premier caractère nul rencontré dans celui-ci.

Il est très important que tout tableau de caractères passé à `cout` contiennent au moins un caractère nul, sinon le flux de sortie affichera le contenu du tableau ainsi que le contenu de la mémoire vive suivant le bloc mémoire alloué au tableau, et ce jusqu'à ce qu'un caractère nul soit rencontré en mémoire. Il arrive même parfois qu'une telle situation cause l'arrêt de l'exécution du programme.



Lecture de chaînes de caractères

23

- Le flux d'entrée `cin` peut lire une chaîne
 - Il ajoute le '`\0`' à la fin

```
char s[ 10 ];
cout << "Chaîne? ";
cin >> s;
cout << s << endl;
```

```
C:\>test.exe
Chaîne? abc
abc
C:\>
```

- Attention au dépassement de capacité

```
C:\>test.exe
Chaîne? abcdefghij
```

- Erreur d'exécution : le '`\0`' est écrit dans la mémoire invalide

- Solution :


```
char s[ 10 ];
cout << "Chaîne? ";
cin >> setw( 10 ) >> s;
cout << s << endl;
```

Va lire au maximum 9 caractères

Marco Lavoie

14728 ORD - Langage C++

Comme mentionné précédemment, le tableau fourni à `cin` doit être assez grand pour stocker la chaîne entrée et son caractère nul de terminaison. On peut utiliser le manipulateur de flux `setw` pour assurer que la chaîne lue dans le tableau n'excède pas la taille de celui-ci. Notez que `setw` réserve une position du tableau pour le caractère nul de terminaison de chaîne.



Lecture de chaînes de caractères (suite)

24

- Notez que `cin` considère l'espace (et le retour de chariot) comme séparateur

```
char s[ 10 ];
cout << "Chaîne? ";
cin >> s;
cout << s << endl;
```

```
C:\>test.exe
Chaîne? abc def
abc
C:\>
```

- Pour lire une chaîne avec ses espaces

```
char s[ 10 ];
cout << "Chaîne? ";
cin.getline( s, 10, '\n' );
cout << s << endl;
```

```
C:\>test.exe
Chaîne? abc def
abc def
C:\>
```

Séparateur à considérer
Taille du tableau fourni
Tableau où stocker la chaîne

Marco Lavoie

14728 ORD - Langage C++

Par défaut, `cin` considère le caractère blanc (i.e. l'espace) et le retour de chariot comme séparateurs de valeurs dans le flux d'entrée. Dans le premier exemple ci-contre, seuls les trois premiers caractères fournis par l'utilisateur (abc) sont lus dans le tableau `s` car le caractère blanc suivant `c` est considéré comme séparateur. Le reste des caractères fournis (def) demeurent dans le flux d'entrée pour la prochaine opération de lecture (c.à.d. pour le prochain `cin` exécuté).

La fonction membre `getline` de `cin` permet de spécifier le séparateur à considérer lors de la prochaine lecture. Cette fonction est donc utile pour lire des chaînes contenant des blancs, tels qu'un nom complet ou une adresse postale.

Nous verrons au chapitre 6 ce qu'est une fonction membre.



Bibliothèque de fonctions de manipulation de chaînes de caractères

25

- Inclure le fichier d'en-tête **<cstring>**

```
#include <cstring>
```

Prototype	Description
<code>char * strcpy(char *s1, const char *s2)</code>	Copie la chaîne s2 dans le tableau de caractères s1 . La valeur de s1 est retournée.
<code>char * strncpy(char *s1, const char *s2, size_t n)</code>	Copie au maximum n caractères de la chaîne s2 dans le tableau de caractères s1 . La valeur de s1 est retournée.
<code>char * strcat(char *s1, const char *s2)</code>	Ajoute la chaîne s2 à la fin de la chaîne dans s1 . Le premier caractère de s2 supprime le <code>'\0'</code> à la fin de la chaîne dans s1 . La valeur de s1 est retournée.
<code>char * strncat(char *s1, const char *s2, size_t n)</code>	Ajoute au maximum n caractères de la chaîne s2 à la fin de la chaîne dans s1 . Le premier caractère de s2 supprime le <code>'\0'</code> à la fin de la chaîne dans s1 . La valeur de s1 est retournée.

Marco Lavoie

14728 ORD - Langage C++

La bibliothèque de manipulation de chaînes offre de nombreuses fonctions utiles pour manipuler, comparer et rechercher des chaînes, les séparer en jetons (ou pièces logiques) et déterminer leur longueur.

Le tableau ci-contre n'énumère que les fonctions les plus couramment utilisées par les programmeurs. Plusieurs autres fonctions non présentées ici sont disponibles.



Bibliothèque de fonctions (suite)

26

- Dans le fichier d'en-tête **<cstring>**

Prototype	Description
<code>int strcmp(const char *s1, const char *s2)</code>	Compare la chaîne dans s1 à celle dans s2 . La fonction retourne 0 si elles sont égales, -1 si s1 est alphabétiquement inférieure à s2 , ou +1 si s1 est alphabétiquement supérieure à s2 .
<code>int strncmp(const char *s1, const char *s2, size_t n)</code>	Compare jusqu'à n caractères de la chaîne dans s1 à celle dans s2 . La fonction retourne 0 si elles sont égales, -1 si s1 est alphabétiquement inférieure à s2 , ou +1 si s1 est alphabétiquement supérieure à s2 .
<code>size_t strlen(const char *s)</code>	Détermine la longueur de la chaîne s . Le nombre de caractères précédant le premier caractère nul (<code>'\0'</code>) rencontré est retournée.

Marco Lavoie

14728 ORD - Langage C++

Plusieurs des fonctions de la bibliothèque, telles que certaines présentées ci-contre, contiennent des paramètres ayant le type de données **size_t**. Ce type est défini comme étant un type d'entier **unsigned int** ou **unsigned long**, dépendant du compilateur et du système d'exploitation. Les variables de type **size_t** peuvent donc être manipulées comme si elles étaient de type entier non signé.



Fonctions `strcpy()` et `strncpy()`

27

- Copier une chaîne dans une autre

```
#include <iostream>
#include <cstring>

using std::cout;
using std::endl;

int main() {
    char x[] = "Joyeux Noël";
    char y[ 25 ], z[ 15 ];

    cout << "Contenu de x: " << x << endl;

    strcpy( y, x );
    cout << "Contenu de y: " << y << endl;

    strncpy( z, x, 6 ); // Ne copie pas le '\0'
    z[ 6 ] = '\0';

    cout << "Contenu de z: " << z << endl;

    return 0;
}
```

```
C:\>copier.exe
Contenu de x: Joyeux Noël
Contenu de y: Joyeux Noël
Contenu de z: Joyeux
C:\>
```

Marco Lavoie

14728 ORD - Langage C++

La fonction **strcpy** copie son second argument, une chaîne, dans son premier argument, un tableau de caractères qui doit être assez grand pour stocker la chaîne et son caractère nul de terminaison.

La fonction **strncpy** équivaut à **strcpy**, à la seule différence qu'elle spécifie le nombre de caractères maximal à copier de la chaîne au tableau. Cette fonction est pratique lorsqu'on ne connaît pas à priori la longueur de la chaîne qui sera fournie en second argument mais qu'on connaît la taille du tableau fourni en premier argument.

Sachez que la fonction **strncpy** ne copie pas nécessairement le caractère nul de terminaison de son deuxième argument, car ce caractère n'est copié que si le nombre de caractères à copier dépasse la longueur de la chaîne d'au moins un caractère.



Fonctions `strcat()` et `strncat()`

28

- Ajoute une chaîne à la fin d'une autre

```
#include <iostream>
#include <cstring>

using std::cout;
using std::endl;

int main() {
    char s1[ 20 ] = "Bonne ";
    char s2[] = "année ";
    char s3[ 40 ] = { 0 };

    cout << "s1: " << s1 << endl;
    cout << "s2: " << s2 << endl;

    strcat( s1, s2 );
    cout << "strcat( s1, s2 ): " << s1 << endl;
    strncat( s3, s1, 6 );
    cout << "strncat( s3, s1, 6 ): " << s3 << endl;
    strcat( s3, s1 );
    cout << "strcat( s3, s1 ): " << s3 << endl;

    return 0;
}
```

```
C:\>ajouter.exe
s1: Bonne
s2: année
strcat( s1, s2 ): Bonne année
strncat( s3, s1, 6 ): Bonne
strcat( s3, s1 ): Bonne Bonne année
C:\>
```

Marco Lavoie

14728 ORD - Langage C++

La fonction `strcat` ajoute son second argument, une chaîne, à la fin du contenu de son premier argument, un tableau de caractères contenant une chaîne. Le premier caractère du second paramètre remplace le caractère nul (`'\0'`) qui termine la chaîne contenue dans le premier argument. Le programmeur doit s'assurer que le tableau servant à stocker la première chaîne est assez grand pour stocker la combinaison des deux chaînes et du caractère nul de terminaison (lequel est copié de la deuxième chaîne).

Quant à la fonction `strncat`, elle ajoute un nombre maximum donné de caractères de la deuxième à la première chaîne, puis un caractère nul de terminaison est ajouté au résultat.



Fonctions `strcmp()` et `strncmp()`

29

- Compare alphabétiquement deux chaînes

```
#include <iostream>
#include <cstring>

using std::cout;
using std::endl;

int main() {
    char s1[] = "Bonne année";
    char s2[] = "Bonne année";
    char s3[] = "Bonnes vacances";

    cout << "strcmp( s1, s2 ): " << strcmp( s1, s2 ) << endl;
    cout << "strcmp( s1, s3 ): " << strcmp( s1, s3 ) << endl;
    cout << "strcmp( s3, s1 ): " << strcmp( s3, s1 ) << endl;

    cout << "strncmp( s1, s3, 5 ): " << strncmp( s1, s3, 5 ) << endl;
    cout << "strncmp( s1, s3, 7 ): " << strncmp( s1, s3, 7 ) << endl;
    cout << "strncmp( s3, s1, 7 ): " << strncmp( s3, s1, 7 ) << endl;

    return 0;
}
```

```
C:\>comparer.exe
strcmp( s1, s2 ): 0
strcmp( s1, s3 ): -1
strcmp( s3, s1 ): 1
strncmp( s1, s3, 5 ): 0
strncmp( s1, s3, 7 ): -1
strncmp( s3, s1, 7 ): 1
C:\>
```

Marco Lavoie

14728 ORD - Langage C++

La fonction `strcmp` compare la première chaîne à la deuxième, caractère par caractère. Elle retourne 0 si les chaînes sont identiques en contenu et en longueur, une valeur négative si la première est inférieure à la deuxième, ou une valeur positive si la première chaîne est supérieure à la deuxième.

Pour essayer de bien comprendre ce que signifie qu'une chaîne est « supérieure » ou « inférieure » à une autre, il faut se référer à la *table des codes ASCII*. Comme nous l'avons vu au chapitre 2, chaque caractère en C++ est représenté par une valeur entière dictée par la table ASCII. La fonction `strcmp` compare les deux chaînes caractère par caractère selon leurs codes ASCII respectifs. Dès que la fonction rencontre deux caractères ayant des codes ASCII différents, elle retourne une valeur numérique déterminée selon ces deux codes.



Fonctions `strlen()`

30

- Retourne la longueur d'une chaîne
 - C'est-à-dire le nombre de caractères avant le premier caractère nul (`'\0'`) rencontré dans la chaîne

```
#include <iostream>
#include <cstring>

using std::cout;
using std::endl;

int main() {
    char s1[] = "abcdefghijklmnopqrstuvwxy";
    char s2[] = "XXX";

    cout << "Longueur de s1: " << strlen( s1 ) << endl;
    cout << "Longueur de s2: " << strlen( s2 ) << endl;

    strcpy( s1 + 3, s2, strlen( s2 ) );
    cout << "Chaîne s1: " << s1 << endl;

    return 0;
}
```

```
C:\>longueur.exe
Longueur de s1: 26
Longueur de s2: 3
Chaîne s1: abcXXXghijklmnopqrstuvwxy
C:\>
```

Marco Lavoie

14728 ORD - Langage C++

La fonction `strlen` prend une chaîne comme argument et renvoie le nombre de caractères inclus dans cette chaîne. Le caractère nul de terminaison n'est toutefois pas inclus dans la longueur.

En fait, la fonction `strlen` retourne l'indice de l'élément contenant le caractère nul dans la chaîne. Par exemples, dans un tableau `t` contenant une chaîne de trois caractères, le caractère nul sera contenu dans le 4^{ème} élément du tableau, soit dans `t[3]`.



Exercice #5.2

31

- Sans faire appel aux fonctions du fichier d'entête `<cstring>`, définissez les deux fonctions suivantes
 - `int maStrlen(const char *)` retourne la longueur de la chaîne en paramètre
 - `char * maStrReverse(char *)` inverse le contenu de la chaîne en paramètre et retourne celle-ci comme valeur de retour
- N'oubliez pas les conventions d'écriture
- Soumettez votre projet selon les indications de l'instructeur

Marco Lavoie

14728 ORD - Langage C++

Cet exercice vous demande de proposer du code source correspondant à la fonction `strlen` de la bibliothèque standard. Comme mentionné à la page précédente, la longueur d'une chaîne dans un tableau de caractères correspond à l'indice de l'élément du tableau contenant le caractère nul de terminaison de chaîne.

En ce qui concerne l'implantation de la fonction `maStrReverse`, vous pouvez exploiter une boucle `for` avec deux variables d'itération : la variable `lo` est initialisée à 0 et incrémentée à chaque itération, alors que la variable `hi` est initialisée à la position précédant le caractère nul dans la chaîne et décrémentée à chaque itération. Tant que les deux indices (`lo` et `hi`) ne se sont pas rejoint, le code dans la boucle doit échanger les éléments `s[lo]` et `s[hi]`.

L'entête d'une telle boucle peut être comme suit :

```
for (int lo=0, hi=strlen(s)-1; lo < hi; lo++, hi--)
```



Tableaux de pointeurs

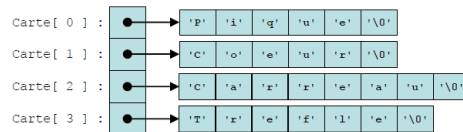
32

- Puisqu'un pointeur est une variable, on peut définir un tableau de pointeurs

```
int t[5] = { 0 };
int *p[5] = { 0 }; // Contient les adresses 0
```

- Exemple concret

```
char *carte[] = { "Pique", "Coeur", "Carreau", "Trefle" };
```



Marco Lavoie

14728 ORD - Langage C++

Les tableaux peuvent également contenir des pointeurs. Une utilisation courante d'une telle structure de données consiste à former un *tableau de chaînes*, communément appelé *tableau de chaînes de caractères*. Chaque élément dans le tableau représente une chaîne de caractères, bien qu'en C++ une chaîne soit en fait un pointeur constant vers le premier caractère de celle-ci.

Dans l'exemple ci-contre, même si les chaînes semblent stockées dans le tableau `carte`, en fait seuls les pointeurs y sont stockés, chaque pointeur ciblant le premier caractère de sa chaîne correspondante (c.à.d. l'élément du tableau contient l'adresse de ce caractère).

Même si le tableau de pointeur est de taille fixe (c.à.d. qu'il ne peut contenir qu'un nombre limité de pointeurs), chacun de ses pointeurs peut pointer vers une chaîne de n'importe quelle longueur. Cette souplesse est un exemple des puissantes possibilités de structuration de données du C++.



Les tableaux ne sont pas des objets

33

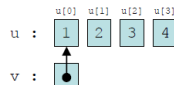
- Donc ils n'ont aucune fonctionnalité intrinsèque (i.e. aucune fonction membre)

- Ne peuvent être affectés l'un à l'autre


```
int u[4] = { 1, 2, 3, 4 };
int v[4];
v = u; // Erreur de compilation
```

- Et attention à l'interprétation du code ci-dessous

```
int u[4] = { 1, 2, 3, 4 };
int *v;
v = u; // Compilation OK mais...
```



- Ne fonctionne pas aussi avec les chaînes

```
char u[10] = "Allô";
char v[10];
v = u; // Erreur
```

→ strcpy(v, u); ✓

Marco Lavoie

14728 ORD - Langage C++

Contrairement à d'autres langages comme *Java* et *C#*, les tableaux en C++ ne sont pas des objets (tels que les collections en *Java*). Conséquemment, plusieurs opérateurs communément exploités pour manipuler des tableaux dans ces langages ne fonctionnent pas de la même façon en C++. Il est par exemple impossible de copier une chaîne de caractères d'un tableau à un autre simplement en affectant le premier au second. Il faut obligatoirement exploiter la fonction `strcpy` ou `strncpy` pour réaliser un tel transfert.

Nous verrons au chapitre 16 la classe `string` qui offre des fonctionnalités de manipulation de chaînes comparables à celles retrouvées dans *Java*.



Erreurs de programmation

34

- Déréférencer un pointeur non initialisé ou initialisé à 0 cause une erreur d'exécution
 - Exemple :

```
int *v;
cout << *v; // v pointe vers quoi?
```
- Initialiser un pointeur à une constante plutôt qu'à une adresse
 - Exemple :

```
int v = 8;
int *p1, *p2;
p1 = v;
p2 = &v;
```
- Confondre le contenu du pointeur et son contenu déréférencé (i.e. oublier l'opérateur *)
 - ```
cout << *p2 << p2; // affiche 8 et l'adresse de v
```

Marco Lavoie

14728 ORD - Langage C++

Les pointeurs sont sources de multiples erreurs de programmation dues à la complexité de leur utilisation en C++. La cause d'erreur la plus fréquente est d'appliquer l'opérateur de déréférencement (\*) à une variable pointeur contenant une adresse invalide, soit l'adresse *nulle* (c.à.d. 0), soit une adresse n'étant associée à aucun bloc mémoire alloué au programme. Une telle situation est une erreur de logique, c'est-à-dire que le programme est compilé avec succès, mais il ne fonctionnera pas correctement à l'exécution.



## Erreurs de programmation (suite)

35

- Exploiter l'arithmétique des pointeurs sur un pointeur ne correspondant pas à un tableau est généralement une erreur de logique
- Déborder la capacité d'un tableau via l'arithmétique des pointeurs
- Tenter de changer l'adresse contenue dans un tableau
  - Exemple : 

```
int t[10], u[10];
int *v;
v = t; // OK
u = t; // Erreur: int u[10] équivalent à const int *u
```

Marco Lavoie

14728 ORD - Langage C++

Une autre erreur logique fréquemment occasionnée par l'arithmétique de pointeurs contenant l'adresse d'un tableau est le débordement d'indice. La boucle suivante est un exemple d'une telle erreur :

```
int v[] = { 10, 20, 30, 40 };
int *pv = &v;
for (int i = 0; i <= 4; i++)
 cout << *(pv + i) << endl;
```

Lors de l'itération pour  $i = 4$ ,  $*(pv + i)$  donne l'adresse de l'entier succédant à 40 dans le tableau (supposant une architecture 32 bits et  $\&v = 3000$ ) :

|      |      |      |      |      |      |      |
|------|------|------|------|------|------|------|
| 10   | 20   | 30   | 40   | ?    | ?    | ?    |
| 3000 | 3004 | 3008 | 3012 | 3016 | 3020 | 3024 |

Pour  $i = 4$ ,  $(pv + i) = (3000 + 4 * 4) = 3016$ , donc  $*(pv + i)$  retourne la valeur entière stockée au bloc de 4 octets débutant à l'adresse 3016 (c.à.d. ?).



## Erreurs de programmation (suite)

36

- Ne pas allouer assez d'espace pour le caractère nul ('`\0`') dans un tableau de caractères
  - Et utiliser une chaîne dans un tableau de caractères n'étant pas terminée par '`\0`'
- Exploiter les fonction `strncpy()` et `strncat()` sans s'assurer que la chaîne résultante sera terminée par '`\0`'
- Lire une chaîne avec `cin` en oubliant que l'espace est considéré comme un séparateur
  - Utiliser plutôt `cin.getline()` lorsque requis

Marco Lavoie

14728 ORD - Langage C++

La manipulation de chaînes dans des tableaux de caractères occasionne souvent des erreurs de logique. C'est le cas, entre autres, si la chaîne n'est pas terminée par le caractère nul dans le tableau, ce qui peut être le résultat d'un appel mal planifié à `strncpy`.

Toute manipulation de chaînes de caractères dans un programme doit susciter chez le programmeur une attention particulière afin d'éviter de telles erreurs de logiques, qui ne sont pas détectées à la compilation mais requièrent l'utilisation du débogueur pour être identifiées lors de l'exécution du programme.



## Bonnes pratiques de programmation

37

- Privilégier les paramètres de référence plutôt que les paramètres pointeurs

```
// Transmutation des arguments
void swap(int &x, int &y) {
 int temp = x;
 x = y;
 y = temp;
}

// Transmutation des arguments
void swap(int *x, int *y) {
 int temp = *x;
 *x = *y;
 *y = temp;
}
```

- Toujours s'assurer qu'un tableau de caractères est assez grand pour contenir la chaîne et le caractère nul ( '\0' )
- Éviter l'arithmétique des pointeurs autant que possible
  - Privilégier l'opérateur [ ] à l'opérateur \*

Marco Lavoie

14728 ORD - Langage C++

Comme mentionné précédemment, la manipulation de pointeurs dans un programme C++ est souvent la cause d'erreurs de logique qui occasionnent la défaillance du programme lors de son exécution. Conséquemment le programmeur doit porter une attention particulière aux instructions d'un programme manipulant des pointeurs. Lors bonnes pratiques de programmation dans de telles circonstances évitent bien des heures de débogages au programmeur averti.



## Devoir #4

38

- Solutionnez le problème distribué par l'instructeur
  - Créer un nouveau projet console *Visual Studio*
  - Solutionner le problème tel que décrit, avec les spécifications supplémentaires suivantes
    - Le programme doit aussi transmettre de façon sonore le message
    - Dans la console, afficher une barre oblique (/) entre chaque mot
    - Les caractères autres que lettres et chiffres doivent être affichés dans la console sous forme d'espace et de façon sonore par une pause équivalente à celle entre deux mots
    - Le programme doit afficher le nombre de caractères invalides (i.e. n'ayant pu être transmis)
  - Respectez l'échéance imposée par l'instructeur
  - Soumettez votre projet selon les indications de l'instructeur
    - **Attention** : respectez à la lettre les instructions de l'instructeur sur la façon de soumettre vos travaux, sinon la note EC sera attribuée à ceux-ci

Marco Lavoie

14728 ORD - Langage C++

Votre programme doit lire des caractères du clavier (à l'aide de `cin`) et afficher ceux-ci en caractères Morse correspondants. Vous pouvez utiliser à cette fin une structure de sélection `switch` imbriquée dans une boucle parcourant le tableau de caractères lus à l'aide de l'instruction `cin.getline`.



## Devoir #4 (suite)

39

- Voici quelques notes supplémentaires sur le devoir
  - L'instructeur a distribué un exécutable de la solution. Inspirez-vous en!
  - Le fichier d'en-tête `<windows.h>` contient les deux fonctions suivantes qui vous seront utiles
    - `Beep( int frequence, int duree )`
      - Active le haut-parleur de l'ordinateur à une fréquence donnée pour une durée donnée en millisecondes
      - Exemple: `Beep( 500, 50 );` // ton de 500 Hz pour 50 ms
    - `Sleep( int duree )`
      - Suspend temporairement l'exécution pour une durée donnée en millisecondes
      - Exemple: `Sleep( 100 );` // Arrêt d'exécution 100 ms

Marco Lavoie

14728 ORD - Langage C++

Alternativement au `switch`, vous pouvez définir un tableau de 39 rangées (une pour chaque caractère traduisible) et de 6 colonnes (maximum de ti et ta dans un caractère traduisible). Ce tableau étant initialisé à la déclaration à l'aides des constantes `TI` et `TA`, vous pouvez y faire référence dans le code afin de convertir les caractères en code Morse sonore :

```
const int TI = 50; // durée d'un ti en ms
const int TA = 3 * TI; // durée d'un ta en ms
```

```
// ti et ta pour chaque caractère valide
const int codeMorse[][6] = {
 { TI, TA }, // A
 { TA, TI, TI, TI }, // B
 { TA, TI, TA, TI }, // C
 { TA, TI, TI }, // D
 { TI }, // E
 ...
}
```



## Devoir #4 (suite)

40

- Notes supplémentaires (suite)
  - Pour avoir plus de détails sur la transmission en code Morse, consultez *Wikipédia* ([http://fr.wikipedia.org/wiki/Alphabet\\_morse](http://fr.wikipedia.org/wiki/Alphabet_morse))
    - Surtout la section intitulée « **Représentation et cadence** »

Marco Lavoie

14728 ORD - Langage C++

N'hésitez pas à consulter la référence ci-contre pour bien comprendre les principes de base du code Morse.

La solution de ce devoir exigera de votre part plusieurs heures de travail et de débogage. Il est donc important que vous commenciez à y travailler le plus tôt possible afin d'avoir suffisamment le temps de demander de l'aide à l'instructeur au besoin.



## Pour la semaine prochaine

41

- Vous devez relire le contenu de la présentation du chapitre 5
  - Il y aura un **examen** sur les chapitres 1 à 5 au prochain cours
    - À livres et ordinateurs ouverts
    - Accès au compilateur interdit
  - Profitez-en pour réviser le contenu de ces chapitres

Marco Lavoie

14728 ORD - Langage C++

En début de classe la semaine prochaine vous aurez à répondre à des questions sur le C++ sans consultation du matériel pédagogique. Vous êtes donc fortement encouragé à relire les notes de cours du chapitre 5.

Profitez-en pour réviser le contenu des chapitres précédents.