Introduction à la programmation
Travaux pratiques: séance 7
INFO0201-1

B. Baert & F. Ludewig
Bruno.Baert@ulg.ac.be - F.Ludewig@ulg.ac.be

Programme de la séance

- Les pointeurs
  $\rightarrow$ Variables qui contiennent l'adresse d'autres variables (= leur localisation en mémoire)
  - Ce qu'est réellement un tableau (statique)
  - Manipulation des pointeurs (passage par adresse et par référence)

Programme de la séance

- Les pointeurs
  $\rightarrow$ Variables qui contiennent l'adresse d'autres variables (= leur localisation en mémoire)
  - Ce qu'est réellement un tableau (statique)
  - Manipulation des pointeurs (passage par adresse et par référence)

- Fonctions : passage de paramètres par adresse et par référence
  $\rightarrow$ Travailler sur les variables initiales, et pas des copies.

Programme de la séance

- Les pointeurs
  $\rightarrow$ Variables qui contiennent l'adresse d'autres variables
  (= leur localisation en mémoire)
  - Ce qu'est réellement un tableau (statique)
  - Manipulation des pointeurs (passage par adresse et par référence)

- Fonctions : passage de paramètres par adresse et par référence
  $\rightarrow$ Travailler sur les variables initiales, et pas des copies.

- Les tableaux dynamiques
  $\rightarrow$ Gestion de tableaux de taille variable avec l'objet *vector*

Introduction : l'adressage de la mémoire (1/2)

La mémoire d'un ordinateur peut être considérée comme un très grand **tableau d'octets** (= ensemble de 8 bits).
Ainsi une mémoire RAM de 1 Go contient $2^{30} = 1\ 073\ 741\ 824$ octets. Ces éléments de tableau sont donc indexés de 0 à 1 073 741 823. L'**index** de chaque élément représente son **adresse mémoire**.

Introduction : l'adressage de la mémoire (1/2)

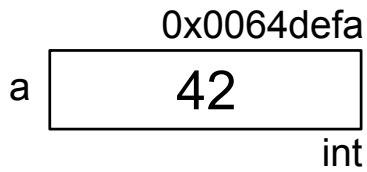
La mémoire d'un ordinateur peut être considérée comme un très grand **tableau d'octets** (= ensemble de 8 bits).
Ainsi une mémoire RAM de 1 Go contient $2^{30} = 1\ 073\ 741\ 824$ octets. Ces éléments de tableau sont donc indexés de 0 à $1\ 073\ 741\ 823$. L'**index** de chaque élément représente son **adresse mémoire**.
Sur un ordinateur 32 bits, les adresses mémoire sont représentées par des entiers dont la valeur maximum est $2^{32} = 4\ 294\ 967\ 296$. Il n'est donc pas possible d'**indexer** des tailles de mémoire supérieures à $4 \times 2^{30} = 4$ Go sur ces ordinateurs. Il faut alors recourir à un système 64 bits, qui peut adresser des tailles de mémoire allant jusqu'à 16 Eo (exaoctet), soit 1 000 000 To ou 10^9 Go.

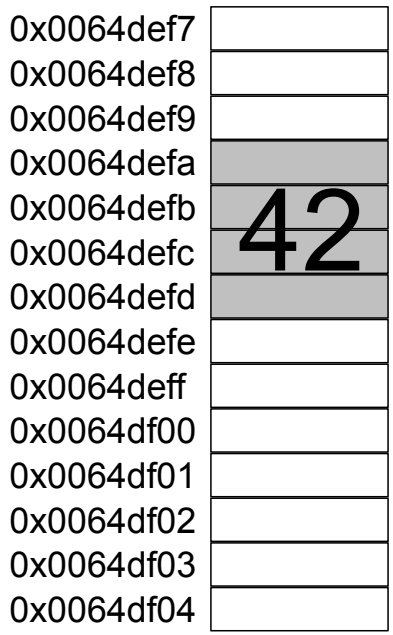
Introduction : l'adressage de la mémoire (2/2)

Soit une variable

```
1 int a = 42;
```

Cette déclaration associe le nom de la variable ("a"), le type **int** et une adresse de la mémoire où la valeur sera stockée. En supposant que cette adresse soit 0x0064defa, on peut représenter la variable selon le schéma suivant :





Une variable de type **int** occupe 4 octets en mémoire, il faudra donc un bloc de 4 octets (0x0064defa, 0x0064defb, 0x0064defc et 0x0064defd) pour stocker la variable. L'adresse de la variable est l'**adresse du premier octet** du bloc de mémoire occupé par la variable.

L'opérateur de référencement

Le langage C++ permet de connaître l'adresse mémoire d'une variable grâce à l'**opérateur de référencement &**.

```
1 int a = 42;
2 cout << "a = " << a << endl; // 42
3 cout << "&a = " << &a << endl; // 0x0064defa
```

L'opérateur de référencement

Le langage C++ permet de connaître l'adresse mémoire d'une variable grâce à l'**opérateur de référencement &**.

```
1 int a = 42;
2 cout << "a = " << a << endl; // 42
3 cout << "&a = " << &a << endl; // 0x0064defa
```

L'affichage de l'adresse mémoire d'une variable n'est pas très utile. Par contre, **stocker l'adresse mémoire** d'une variable dans une autre variable peut être très utile. Cela permettra ainsi de modifier le contenu de la variable dont on aura retenu l'adresse.

$\rightarrow$ On utilise un type de variable spécifique, **le pointeur**.

L'opérateur de référencement

Le langage C++ permet de connaître l'adresse mémoire d'une variable grâce à l'**opérateur de référencement &**.

```
1 int a = 42;
2 cout << "a = " << a << endl; // 42
3 cout << "&a = " << &a << endl; // 0x0064defa
```

L'affichage de l'adresse mémoire d'une variable n'est pas très utile. Par contre, **stocker l'adresse mémoire** d'une variable dans une autre variable peut être très utile. Cela permettra ainsi de modifier le contenu de la variable dont on aura retenu l'adresse.

$\rightarrow$ On utilise un type de variable spécifique, **le pointeur**.

- Manipuler les adresses est plus léger et plus rapide que de travailler avec les variables ou les tableaux.

L'opérateur de référencement

Le langage C++ permet de connaître l'adresse mémoire d'une variable grâce à l'**opérateur de référencement &**.

```
1 int a = 42;
2 cout << "a = " << a << endl; // 42
3 cout << "&a = " << &a << endl; // 0x0064defa
```

L'affichage de l'adresse mémoire d'une variable n'est pas très utile. Par contre, **stocker l'adresse mémoire** d'une variable dans une autre variable peut être très utile. Cela permettra ainsi de modifier le contenu de la variable dont on aura retenu l'adresse.

→ On utilise un type de variable spécifique, **le pointeur**.

- Manipuler les adresses est plus léger et plus rapide que de travailler avec les variables ou les tableaux.
- **Attention** : manipuler des adresses, c'est jouer directement avec la mémoire. Les erreurs peuvent être graves et difficiles à détecter.

Les pointeurs

Un **pointeur** est donc une variable spéciale, dont le contenu est l'adresse d'une autre variable. Comment déclarer un pointeur :

type* nom_pointeur ;

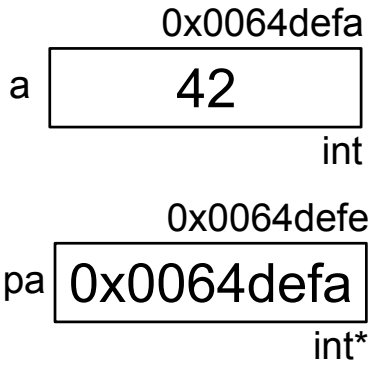
- L'étoile "*****" indique que la variable est un pointeur ;
- **type** est le type de variable auquel le pointeur fait référence.

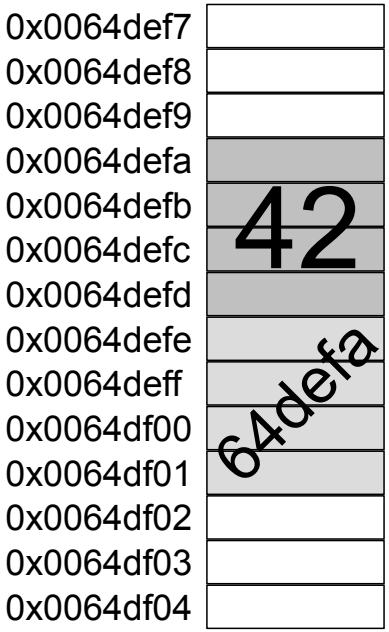
```
1 int a = 42; // un entier normal
2 int* pa = &a; // un pointeur avec l'adresse de 'a'
3 cout << "a = " << a << endl; // 42
4 cout << "&a = " << &a << endl; // 0x0064defa
5 cout << "pa = " << pa << endl; // 0x0064defa
6 cout << "&pa = " << &pa << endl; // 0x0064defe
```

ATTENTION : Un pointeur doit être initialisé avant d'être utilisé. Sans cela, il pointe vers une zone mémoire indéterminée, qui ne peut pas être utilisée !

Les pointeurs

Dans la mémoire on a la représentation suivante :



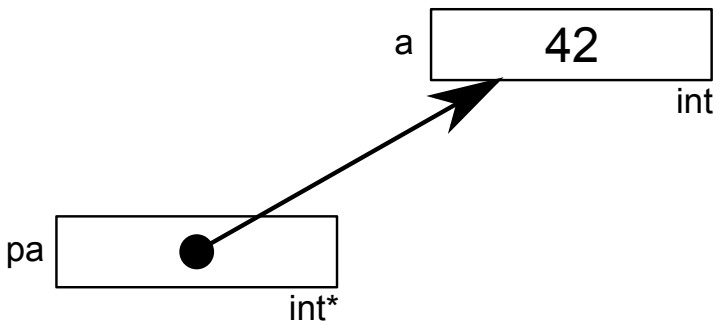


La valeur du pointeur 'pa' est égale à l'adresse mémoire de la variable 'a' : 0x0064defa. Cependant, 'pa' est une variable différente de 'a', et possède donc sa propre adresse : 0x0064defe.

L'opérateur de déréférencement

Lorsqu'un pointeur 'pa' contient l'adresse d'une autre variable 'a', on peut vouloir accéder à la valeur de cette variable 'a' à partir du pointeur 'pa'.
On utilise pour cela l'**opérateur de déréférencement *** qui permet d'accéder à la variable vers laquelle pointe une variable de type pointeur.

```
1 int a = 42; // un entier normal
2 int* pa = &a; // un pointeur vers 'a'
3 cout << "*pa = " << *pa << endl; // 42
4 *pa = 60;
5 cout << "a = " << a << endl; // 60
```



Manipulation des pointeurs : opérations

On ne peut pas faire d'opérations arithmétiques avec des pointeurs (à quelques exceptions près - voir séance 8), mais on peut évidemment en faire sur les variables vers lesquelles ils pointent :

```
1     int x, y, z;
2     int *a = &x;
3     int *b = &y;
4     int *c = &z;
5     *a = 0;
6     *b = 5;
7     *c = 7;
8     a = b+c // Faux
9     *a = *b+*c // Vrai --> place la valeur 12 dans la
      zone pointée par a
```

Exemple

Déterminer l'affichage de ce programme

```
1  int main()
2  {
3      int a,b;
4      int *x,*y;
5      a=50;
6      b=80;
7      x=&a;
8      y=&b;
9      *x=*y;
10     cout << "a vaut : " << a << endl;
11     x=y;
12     *x=1000;
13     cout << "b vaut : " << b << endl;
14     return 0;
15 }
```

Exemple

Le programme affichera :

a vaut : 80
b vaut : 1000

Exemple

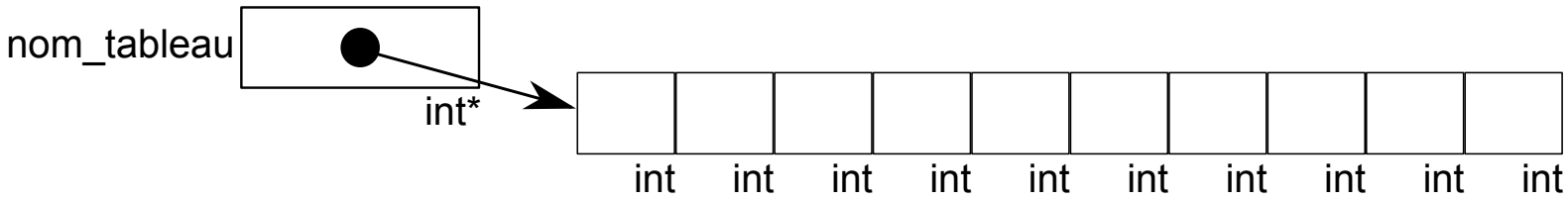
Déroulement étape par étape

```
1  int main()
2  {
3    int a,b; int *x,*y; a=50; b=80;
4    x=&a; // x pointe vers a
5    y=&b; // y pointe vers b
6    *x=*y; // la valeur de la variable pointée par x
       prend la valeur de la variable pointée par y
7    // --> a = b; // 80
8    cout << "a vaut : " << a << endl; // a = 80
9    x=y;// x pointe vers la variable pointée par y -> b
10   *x=1000; // la variable pointée par x (b) prend la
       valeur 1000
11   cout << "b vaut : " << b << endl; // 1000
12   return 0;
13 }
```

Les pointeurs et les tableaux

Un nom de tableau classique (statique) n'est en fait qu'un pointeur qui est initialisé à la compilation du programme et ne peut être modifié lors de l'exécution.

Soit un tableau **int** `nom_tableau[10]` : nom_tableau est un pointeur vers le premier élément du tableau et tous les éléments du tableau se suivent dans la mémoire.



Si les éléments du tableau ont une taille de 4 octets (type int), les adresses des éléments du tableau sont :

- Premier élément : l'adresse du tableau
- Deuxième élément : l'adresse du tableau + 4
- Troisième élément : l'adresse du tableau + 8
- Quatrième élément : l'adresse du tableau + 12
- ...

Les pointeurs et les tableaux : l'opérateur d'indice

L'opérateur d'indice **[]** permet d'accéder à un élément particulier d'un tableau. Avec la notion de pointeur, on peut maintenant comprendre comment il fonctionne :

- L'opérateur [] décale d'abord le pointeur initial du tableau (qui pointe vers le premier élément) d'une quantité égale au produit de l'indice et de la taille d'un élément.
- L'opérateur [] déréférence ensuite ce pointeur, ce qui permet d'accéder à la **valeur** de l'élément vers lequel il pointe.

nom_tableau est un pointeur et nom_tableau[i] est une variable.

```
1 double t[10];
2 // t = 0x0068be42 = 6 864 450
3 *t = 0; // --> t[0] = 0;
4 double* t5 = &t[5];
5 // t5 = 6 864 450 + 5x4 = 6 864 470
6 *t5 = 0; // --> t[5] = 0;
```

Passage d'un tableau en paramètre d'une fonction

- On comprend maintenant pourquoi, lorsqu'un tableau est passé en paramètre à une fonction, les valeurs du tableau initial sont modifiées lorsque ses valeurs sont modifiées dans la fonction.
- En effet, le nom de tableau passé en paramètre est un **pointeur**. Le tableau auquel on accède dans la fonction est donc **celui** qui a servi lors de l'appel de la fonction, et pas une copie.
- Pour la même raison, on ne peut donc absolument pas utiliser un tableau déclaré dans une fonction comme valeur de retour. En effet,
  - les variables locales déclarées dans une fonction n'existent que pendant l'exécution de cette fonction (ceci est valable pour tous les blocs d'instruction)
  - on retournerait alors un pointeur vers un tableau qui cesserait d'exister après l'exécution de la fonction.

Fonction : paramètres par adresse et par référence

Nous avons déjà vu la notion de fonction et de passage de paramètre par valeur. Il existe au total 3 moyens de passer des arguments à une fonction :

Fonction : paramètres par adresse et par référence

Nous avons déjà vu la notion de fonction et de passage de paramètre par valeur. Il existe au total 3 moyens de passer des arguments à une fonction :

- **Passage par valeur** : la valeur de la variable passée en paramètre est copiée dans une nouvelle variable qui n'existe que dans la fonction. Les modifications appliquées à cette copie ne changent pas la valeur de la variable initiale.

Fonction : paramètres par adresse et par référence

Nous avons déjà vu la notion de fonction et de passage de paramètre par valeur. Il existe au total 3 moyens de passer des arguments à une fonction :

- **Passage par valeur** : la valeur de la variable passée en paramètre est copiée dans une nouvelle variable qui n'existe que dans la fonction. Les modifications appliquées à cette copie ne changent pas la valeur de la variable initiale.
- **Passage par adresse** : c'est un pointeur vers la variable utile qui est passé en paramètre. Il permet d'accéder à la valeur de cette variable mais aussi de modifier la valeur de la variable initiale.

  Remarque un tableau est donc toujours passé par adresse

Fonction : paramètres par adresse et par référence

Nous avons déjà vu la notion de fonction et de passage de paramètre par valeur. Il existe au total 3 moyens de passer des arguments à une fonction :

- **Passage par valeur** : la valeur de la variable passée en paramètre est copiée dans une nouvelle variable qui n'existe que dans la fonction. Les modifications appliquées à cette copie ne changent pas la valeur de la variable initiale.
- **Passage par adresse** : c'est un pointeur vers la variable utile qui est passé en paramètre. Il permet d'accéder à la valeur de cette variable mais aussi de modifier la valeur de la variable initiale.

  Remarque un tableau est donc toujours passé par adresse
- **Passage par référence** : On transmet indirectement un pointeur. Le passage par référence stipule que la variable ne doit pas être copiée, mais que les modifications qui lui sont appliquées dans la fonction doivent l'être à la variable initiale. Il suffit pour cela de précéder le nom de la variable du caractère "&" : int fonction(int &a)

Fonction : paramètres par adresse et par référence

Nous avons déjà vu la notion de fonction et de passage de paramètre par valeur. Il existe au total 3 moyens de passer des arguments à une fonction :

- **Passage par valeur** : la valeur de la variable passée en paramètre est copiée dans une nouvelle variable qui n'existe que dans la fonction. Les modifications appliquées à cette copie ne changent pas la valeur de la variable initiale.
- **Passage par adresse** : c'est un pointeur vers la variable utile qui est passé en paramètre. Il permet d'accéder à la valeur de cette variable mais aussi de modifier la valeur de la variable initiale.

  Remarque un tableau est donc toujours passé par adresse
- **Passage par référence** : On transmet indirectement un pointeur. Le passage par référence stipule que la variable ne doit pas être copiée, mais que les modifications qui lui sont appliquées dans la fonction doivent l'être à la variable initiale. Il suffit pour cela de précéder le nom de la variable du caractère "&" : int fonction(int &a)

Avec ces deux dernières méthodes, seule l'adresse est passée en argument à la fonction. Il n'existe alors qu'une seule zone mémoire partagée entre la fonction et la partie du programme qui l'appelle. Toutes les modifications réalisées dans la fonction seront appliquées aux variables originales.

Fonction : passage par adresse

```
1  void minmax(int i, int j, int* min, int* max)
2  {
3      if(i<j)
4      { *min=i; *max=j; }
5      else
6      { *min=j; *max=i; }
7  }
8  int main()
9  {
10     int a, b, w, x;
11     cout << "Tapez la valeur de a : "; cin >> a;
12     cout << "Tapez la valeur de b : "; cin >> b;
13     minmax(a, b, &w, &x);
14     cout << "Le plus petit vaut : " << w << endl;
15     cout << "Le plus grand vaut : " << x << endl;
16     return 0;
17 }
```

Fonction : passage par adresse

Explication de l'exemple :

- Dans cet exemple, on a une fonction minmax() qui prend comme paramètres 2 entiers 'i' et 'j' et 2 pointeurs vers des entiers 'min' et 'max'. Cette fonction trouve le plus petit de 'i' et de 'j' et place sa valeur dans l'entier pointé par 'min'. Elle trouve le plus grand des 2 entiers et le copie dans l'entier pointé par 'max'.
- Dans la fonction main(), on déclare 4 entiers 'a', 'b', 'w', et 'x'. On demande à l'utilisateur de saisir au clavier les entiers 'a' et 'b'. Lors de l'appel de fonction minmax(a,b,&w,&x), on copie la valeur de 'a' dans 'i', la valeur de 'b' dans 'j'. On copie la valeur de l'adresse '&w' (un pointeur vers 'w') dans 'min' et on copie '&x' (un pointeur vers 'x') dans 'max' : 'min' pointe donc vers 'w' et 'max' vers 'x'. Lors de l'appel, on va donc récupérer dans 'w' le plus petit des entiers et dans 'x' le plus grand des 2 entiers.

Fonction : passage par référence

```
1  void minmax(int i, int j, int& min, int& max)
2  {
3      if(i<j)
4      { min=i; max=j; }
5      else
6      { min=j; max=i; }
7  }
8  int main()
9  {
10     int a, b, w, x;
11     cout << "Tapez la valeur de a : "; cin >> a;
12     cout << "Tapez la valeur de b : "; cin >> b;
13     minmax(a, b, w, x);
14     cout << "Le plus petit vaut : " << w << endl;
15     cout << "Le plus grand vaut : " << x << endl;
16     return 0;
17 }
```

Fonction : passage par référence

Explication de l'exemple :

- Au lieu d'utiliser un passage de paramètres par pointeur comme dans l'exemple précédent, on peut utiliser un passage de paramètres par référence.
- Dans cet exemple, la fonction minmax() possède 4 paramètres : 2 entiers 'i' et 'j' passés par valeur et 2 entiers 'min' et 'max' passés par référence. 'i' et 'j' sont les paramètres en entrée de la fonction minmax(). 'min' et 'max' sont les paramètres en sortie de cette fonction. Le passage par référence permet donc d'avoir plus d'une variable de retour dans une fonction.
- Lors de l'écriture de la fonction minmax(), on remarquera le symbole & placé après le type qui indique que le paramètre est passé par référence.
- Lors de l'appel de minmax(), on remarquera aussi qu'il s'écrit minmax(a,b,w,x) ; sans symbole particulier. 'a' et 'b' sont passés par valeur et 'w' et 'x' sont passés par référence.

L'exécution de ce programme affichera :

```
Tapez la valeur de a : 25
Tapez la valeur de b : 12
Le plus petit vaut : 12
Le plus grand vaut : 25
```

L'objet *vector*

- Pour gérer de manière pratique des tableaux dynamiques, c'est-à-dire des tableaux dont la taille peut changer pendant l'exécution du programme, il existe en C++ un objet spécifique : l'objet **vector**

L'objet *vector*

- Pour gérer de manière pratique des tableaux dynamiques, c'est-à-dire des tableaux dont la taille peut changer pendant l'exécution du programme, il existe en C++ un objet spécifique : l'objet **vector**
- Les avantages principaux de cet objet sont les suivants :
  - Gestion de l'allocation et de la libération de mémoire sous-jacente
  - Tableaux redimensionnables lors de l'exécution
  - Copie de tableaux complets facilitée

L'objet *vector*

- Pour gérer de manière pratique des tableaux dynamiques, c'est-à-dire des tableaux dont la taille peut changer pendant l'exécution du programme, il existe en C++ un objet spécifique : l'objet **vector**
- Les avantages principaux de cet objet sont les suivants :
  - Gestion de l'allocation et de la libération de mémoire sous-jacente
  - Tableaux redimensionnables lors de l'exécution
  - Copie de tableaux complets facilitée
- Pour pouvoir l'utiliser, il faut inclure la bibliothèque "vector" :

```
#include <vector>
```

L'objet *vector* : utilisation

- Les objets `vector` font appel à une notion, les *templates*, pour spécifier le type des variables stockées dans un tableau dynamique. La notation de la déclaration d'un objet `vector` est la suivante : `vector<type> mon_vector;`

```
1 vector<int> mon_tableau_dynamique;
```

L'objet *vector* : utilisation

- Les objets `vector` font appel à une notion, les *templates*, pour spécifier le type des variables stockées dans un tableau dynamique. La notation de la déclaration d'un objet `vector` est la suivante : `vector<type> mon_vector;`

```
1 vector<int> mon_tableau_dynamique;
```

- On peut également déclarer un objet `vector` d'une certaine taille (qui peut désormais être une variable qui ne prend une valeur connue qu'à l'exécution), et préciser avec quelle valeur remplir tous les éléments du tableau par défaut : `vector<type> mon_vector(int size, type value);`

```
1 int n, m;
2 cin >> n >> m;
3 vector<int> mon_tableau_dynamique(n, m);
4 // tableau dynamique de n éléments égaux à m
```

L'objet *vector* : utilisation

- On peut accéder aux éléments d'un objet `vector` comme à ceux d'un tableau normal. On utilise pour cela l'opérateur [ ] :

```
1 vector<double> mon_tableau(100, 0);
2 for(int i=0; i < 100; i++)
3 {
4   mon_tableau[i] = cos(i/100.);
5 }
```

ATTENTION : La valeur de l'index utilisé avec l'opérateur [ ] doit être inférieure à la taille de l'objet `vector`, comme pour les tableaux statiques. Dans le cas contraire, le résultat n'est pas défini, mais mènera certainement à une erreur !

Propriétés et fonctions membres de l'objet *vector*

L'objet *vector* contient également toute une série de fonctions permettant d'obtenir des informations ou d'effectuer des manipulations sur le tableau de données qu'il représente.

Toute la documentation concernant les fonctions liées à l'objet *vector* est disponible à l'adresse
`http://www.cplusplus.com/reference/vector/vector/`

Remarque : Dans certaines de ces fonctions, il est fait appel au concept d'*itérateur*. Cette notion ne fait pas partie du cadre de ce cours, mais est expliquée en détails sur le site de référence, avec de nombreux exemples explicites.

Propriétés et fonctions membres de l'objet *vector*

Parmi les fonctions principales, on trouve ainsi , avec `v` un objet de type vecteur :

- **int** `v.size()` : Renvoit la taille du tableau de données contenu dans le vector `v`

Propriétés et fonctions membres de l'objet *vector*

Parmi les fonctions principales, on trouve ainsi , avec `v` un objet de type vecteur :

- **int** `v.size()` : Renvoit la taille du tableau de données contenu dans le vector `v`
- **void** `v.resize(`**int** `n,` **type** `val)` : Redimensionne le vecteur à la taille *n*
  - Si *n* est plus grand que la taille précédente : le `vector` est agrandit à la taille *n* et les nouveaux éléments prennent la valeur *val* ;
  - Si *n* est plus petit que la taille précédente : le `vector` est réduit à la taille *n* et les éléments excédentaires sont supprimés (le paramètre *val* n'est dans ce cas pas nécessaire)

Propriétés et fonctions membres de l'objet *vector*

Parmi les fonctions principales, on trouve ainsi , avec `v` un objet de type vecteur :

- **int** `v.size()` : Renvoit la taille du tableau de données contenu dans le vector `v`
- **void** `v.resize(`**int** `n, `**type** `val)` : Redimensionne le vecteur à la taille *n*
  - Si *n* est plus grand que la taille précédente : le `vector` est agrandit à la taille *n* et les nouveaux éléments prennent la valeur *val* ;
  - Si *n* est plus petit que la taille précédente : le `vector` est réduit à la taille *n* et les éléments excédentaires sont supprimés (le paramètre *val* n'est dans ce cas pas nécessaire)
- **bool** `v.empty()` : Teste si le vector `v` est vide (taille = 0)

Propriétés et fonctions membres de l'objet *vector*

- **void** `v.push_back(`**type** `val)` : Ajoute un élément à la fin du `vector` avec la valeur *val*

Propriétés et fonctions membres de l'objet *vector*

- **void** `v.push_back(`**type** `val)` : Ajoute un élément à la fin du `vector` avec la valeur *val*
- **void** `v.pop_back()` : Supprime le dernier élément du `vector`

Propriétés et fonctions membres de l'objet *vector*

- **void** `v.push_back(`**type** `val)` : Ajoute un élément à la fin du `vector` avec la valeur *val*
- **void** `v.pop_back()` : Supprime le dernier élément du `vector`
- **type&** `v.front()` : Renvoit une *référence* vers le premier élément du `vector`, *i.e.* `v[0]`

Propriétés et fonctions membres de l'objet *vector*

- **void** `v.push_back(`**type** `val)` : Ajoute un élément à la fin du `vector` avec la valeur *val*
- **void** `v.pop_back()` : Supprime le dernier élément du `vector`
- **type&** `v.front()` : Renvoit une *référence* vers le premier élément du `vector`, *i.e.* `v[0]`
- **type&** `v.back()` : Renvoit une *référence* vers le dernier élément du `vector`, *i.e.* `v[v.size() - 1]`

Propriétés et fonctions membres de l'objet *vector*

- **void** `v.push_back(`**type** `val)` : Ajoute un élément à la fin du `vector` avec la valeur *val*
- **void** `v.pop_back()` : Supprime le dernier élément du `vector`
- **type&** `v.front()` : Renvoit une *référence* vers le premier élément du `vector`, *i.e.* `v[0]`
- **type&** `v.back()` : Renvoit une *référence* vers le dernier élément du `vector`, *i.e.* `v[v.size() - 1]`
- **void** `v.clear()` : Supprime tous les éléments du `vector`, ramenant ainsi sa taille à 0

Propriétés et fonctions membres de l'objet *vector*

- **void** `v.push_back(`**type** `val)` : Ajoute un élément à la fin du `vector` avec la valeur *val*
- **void** `v.pop_back()` : Supprime le dernier élément du `vector`
- **type&** `v.front()` : Renvoit une *référence* vers le premier élément du `vector`, *i.e.* `v[0]`
- **type&** `v.back()` : Renvoit une *référence* vers le dernier élément du `vector`, *i.e.* `v[v.size() - 1]`
- **void** `v.clear()` : Supprime tous les éléments du `vector`, ramenant ainsi sa taille à 0
- **type*** `v.data()` : Renvoit un pointeur direct vers la zone mémoire contenant le tableau de données du `vector`

Opérateurs utilisables avec l'objet *vector*

Certains opérateurs peuvent être utilisés avec les objets de type `vector` :

- L'opérateur d'affectation "=" :
  Cet opérateur permet, contrairement à ce qui se produirait avec deux tableaux classiques, de **copier** les valeurs d'un `vector` dans un autre.

Opérateurs utilisables avec l'objet *vector*

Certains opérateurs peuvent être utilisés avec les objets de type `vector` :

- L'opérateur d'affectation "=" :
  Cet opérateur permet, contrairement à ce qui se produirait avec deux tableaux classiques, de **copier** les valeurs d'un `vector` dans un autre.

- L'opérateur de comparaison "==" :
  Cet opérateur vérifie d'abord que les deux objets `vector` ont la même taille (`.size()`) et si c'est le cas, les éléments respectifs des deux `vector` sont comparés deux à deux.
  La comparaison est donc **vraie** si le contenu des deux `vector` est strictement égal, et **faux** dans le cas contraire.

Exercices

1. Opérations via des pointeurs :
   Déclarer une variable entière a, et trois pointeurs vers des entiers $*b$, $*c$ et $*d$ qui pointent tous vers a. Prendre comme valeur initiale $a = 2$, multiplier successivement les valeurs de la variable pointée par chacun des pointeurs $*b$, $*c$ et $*d$ en affichant la valeur de a à chaque étape.

2. Manipulations de pointeurs :
   Ecrire un programme qui affiche les adresses de chacun des éléments d'un tableau de 10 entiers. Vérifier de combien d'octets les éléments sont séparés, et comparer entre des tableaux de type **double**, **int** et **char**.
   Remarque : pour afficher l'adresse au format décimal, il est utile de convertir le pointeur en un nombre entier (**int**) au préalable.

Exercices

3. Calcul sur un nombre arbitraire de données :
   Ecrire un programme grâce auquel l'utilisateur peut saisir deux séries (de même longueur) de données expérimentales. Ecrire une fonction qui prend en paramètre deux `vector` et retourne un nouvel objet `vector` dont chaque élément est le plus grand des deux éléments correspondants des deux `vectorx` passés en paramètre :

   $$R_i = \max(A_i, B_i), \text{ pour } i = 0 \rightarrow n - 1,$$

   où A et B sont les paramètres et R est le tableau de résultat.
   Appliquer cette fonction aux données saisies par l'utilisateur.
   Le nombre de données expérimentales n n'est pas connu à priori : l'utilisateur le choisit au début de l'exécution du programme.

4. Passage par référence et par adresse :
   Ecrire une fonction qui prend en paramètre un nombre réel x et modifie sa valeur en $\sqrt[4]{x}$. La fonction retourne un entier qui vaut 0 si la fonction n'a pas rencontré de problème et 1 si la valeur de x était négative (dans ce cas, la valeur finale de x restera inchangée). Ecrire une version de la fonction qui utilise un passage par adresse et une version qui utilise un passage par référence et les utiliser pour calculer $\sqrt[4]{65\ 536} = \sqrt[4]{2^{16}}$.

Exercices

5. Traduction en code morse

   On souhaite écrire un programme grâce auquel on peut saisir quelques mots au clavier, et voir leur transcription en alphabet morse affichée à l'écran et enregistrée dans un fichier. Pour ce faire :

   - Charger les données du fichier "alphabet_morse.txt", qui contient des lignes du type "F ..-.", donnant la correspondance de toutes les lettres de l'alphabet, dans deux `vector`, l'un contenant l'alphabet latin, l'autre l'alphabet morse.
   - Faire saisir des mots à l'utilisateur et les placer dans un 3^{e} `vector`
     - Jusqu'à ce que le mot "STOP" soit saisi
     - Vérifier que le mot ne contient que des lettres de A à Z
     - Si le mot contient des minuscules, les transformer en majuscules
   - A l'aide des correspondances entre les deux `vector` précédemment remplis, traduire les mots un par un et lettre par lettre en alphabet morse.
     - Afficher les mots en alphabet morse à l'écran
     - Enregistrer les mots traduits en morse dans un fichier "code_morse.txt"
   - BONUS : Tester le programme avec le fichier "alphabet_militaire.txt" au lieu de l'alphabet morse. Si le programme précédent a été correctement écrit, la traduction en alphabet militaire devrait fonctionner sans aucune modification.