Introduction à l'informatique
Travaux pratiques: séance 5
INFO0205-1

X. Baumans
(xavier.baumans@ulg.ac.be)

Programme de la séance

- Tableaux statiques (à 1 et 2 dimensions)
  $\rightarrow$ stockage d'une série d'éléments d'un même type

Programme de la séance

- Tableaux statiques (à 1 et 2 dimensions)
  $\rightarrow$ stockage d'une série d'éléments d'un même type

- Rappels sur les nombres aléatoires

Programme de la séance

- Tableaux statiques (à 1 et 2 dimensions)
  $\rightarrow$ stockage d'une série d'éléments d'un même type

- Rappels sur les nombres aléatoires

Compléments :

Programme de la séance

- Tableaux statiques (à 1 et 2 dimensions)
  $\rightarrow$ stockage d'une série d'éléments d'un même type

- Rappels sur les nombres aléatoires

Compléments :

- Librairies mathématiques
  $\rightarrow$ utilisation de fonctions mathématiques supplémentaires *(à lire)*

Programme de la séance

- Tableaux statiques (à 1 et 2 dimensions)
  $\rightarrow$ stockage d'une série d'éléments d'un même type

- Rappels sur les nombres aléatoires

Compléments :

- Librairies mathématiques
  $\rightarrow$ utilisation de fonctions mathématiques supplémentaires *(à lire)*
- Opérateurs d'affectation et unaires (+=, -=, ++, −−, ...)
  $\rightarrow$ expressions condensées, code plus concis *(à lire)*

Programme de la séance

- Tableaux statiques (à 1 et 2 dimensions)
  $\rightarrow$ stockage d'une série d'éléments d'un même type

- Rappels sur les nombres aléatoires

Compléments :

- Librairies mathématiques
  $\rightarrow$ utilisation de fonctions mathématiques supplémentaires *(à lire)*
- Opérateurs d'affectation et unaires (+=, -=, ++, −−, ...)
  $\rightarrow$ expressions condensées, code plus concis *(à lire)*
- Le type **char** et le code ASCII
  (caractères alphanumériques) *(à lire)*

Les tableaux statiques

Lorsque nous devons utiliser une suite ou série de données, il peut être plus commode d'utiliser un tableau. Celui-ci permet de centraliser les données sous un même nom de variable et l'accès aux différents éléments est réalisé par l'utilisation d'un indice.

Un tableau permet de mémoriser une suite ou série de valeurs d'un même type. La taille d'un tableau statique est fixée pour toute l'exécution du programme.

Exemples :

Les tableaux statiques

Lorsque nous devons utiliser une suite ou série de données, il peut être plus commode d'utiliser un tableau. Celui-ci permet de centraliser les données sous un même nom de variable et l'accès aux différents éléments est réalisé par l'utilisation d'un indice.

Un tableau permet de mémoriser une suite ou série de valeurs d'un même type. La taille d'un tableau statique est fixée pour toute l'exécution du programme.

Exemples :

- 1D : un vecteur, une liste, un ensemble de données, ...

Les tableaux statiques

Lorsque nous devons utiliser une suite ou série de données, il peut être plus commode d'utiliser un tableau. Celui-ci permet de centraliser les données sous un même nom de variable et l'accès aux différents éléments est réalisé par l'utilisation d'un indice.

Un tableau permet de mémoriser une suite ou série de valeurs d'un même type. La taille d'un tableau statique est fixée pour toute l'exécution du programme.

Exemples :

- 1D : un vecteur, une liste, un ensemble de données, ...
- 2D : une matrice, ...

Les tableaux statiques : déclaration

Comment déclarer un tableau 1D :

type nom_tableau[taille_tableau] ;

Un tableau 2D :

type nom_tableau[taille_1][taille_2] ;

Exemple :

```
1 int v[3]; // tableau de 3 éléments entiers
2 double tableau_2D[10][20]; // tableau de 10x20 double
```

Les tableaux statiques : déclaration

Comment déclarer un tableau 1D :

type nom_tableau[taille_tableau] ;

Un tableau 2D :

type nom_tableau[taille_1][taille_2] ;

Exemple :

```
1 int v[3]; // tableau de 3 éléments entiers
2 double tableau_2D[10][20]; // tableau de 10x20 double
```

- **Type** de variable : **int**, **float**, **double**, **char**, ...

Les tableaux statiques : déclaration

Comment déclarer un tableau 1D :

type nom_tableau[taille_tableau] ;

Un tableau 2D :

type nom_tableau[taille_1][taille_2] ;

Exemple :

```
1 int v[3]; // tableau de 3 éléments entiers
2 double tableau_2D[10][20]; // tableau de 10x20 double
```

- **Type** de variable : **int**, **float**, **double**, **char**, ...
- **Nom** du tableau : le plus **représentatif** et **concis** possible

Les tableaux statiques : déclaration

Comment déclarer un tableau 1D :

type nom_tableau[taille_tableau] ;

Un tableau 2D :

type nom_tableau[taille_1][taille_2] ;

Exemple :

```
1 int v[3]; // tableau de 3 éléments entiers
2 double tableau_2D[10][20]; // tableau de 10x20 double
```

- **Type** de variable : **int**, **float**, **double**, **char**, ...
- **Nom** du tableau : le plus **représentatif** et **concis** possible
- **[...]** : autant de paires de crochets que de dimensions

Les tableaux statiques : déclaration

Comment déclarer un tableau 1D :

type nom_tableau[taille_tableau] ;

Un tableau 2D :

type nom_tableau[taille_1][taille_2] ;

Exemple :

```
1 int v[3]; // tableau de 3 éléments entiers
2 double tableau_2D[10][20]; // tableau de 10x20 double
```

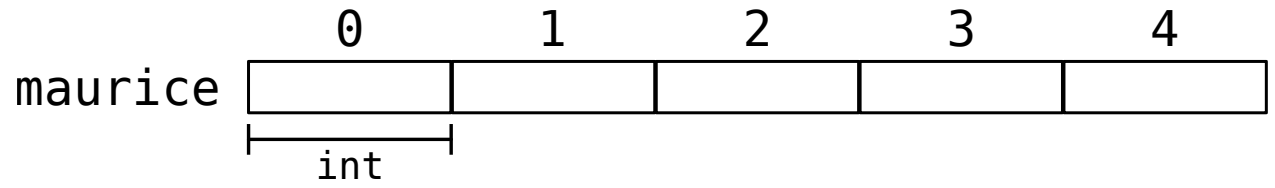
- **Type** de variable : **int**, **float**, **double**, **char**, ...
- **Nom** du tableau : le plus **représentatif** et **concis** possible
- **[...]** : autant de paires de crochets que de dimensions
- **Taille** : entre chaque crochet préciser la taille de la dimension correspondante du tableau

Les tableaux statiques : déclaration et initialisation

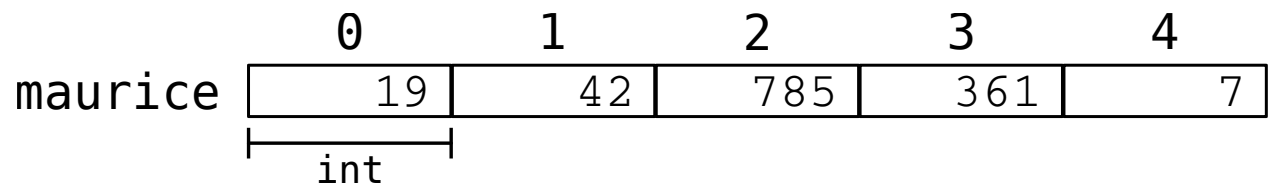
Déclaration d'un tableau = **réservation** de la mémoire

Les éléments (cases) d'un tableau sont contigus dans la mémoire

```
1 int maurice[5]; // déclaration d'un tableau de 5 int
```




```
1 // déclaration et initialisation d'un tableau de 5 int
2 int maurice[5] = {19, 42, 785, 361, 7};
```

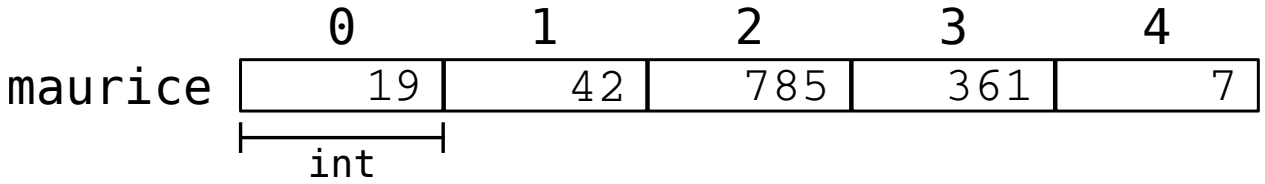


Si on veut initialiser tout le tableau à zéro :

```
1 int peter[5] = {0}; // équivalent à {0, 0, 0, 0, 0}
```

Attention au volume de mémoire !

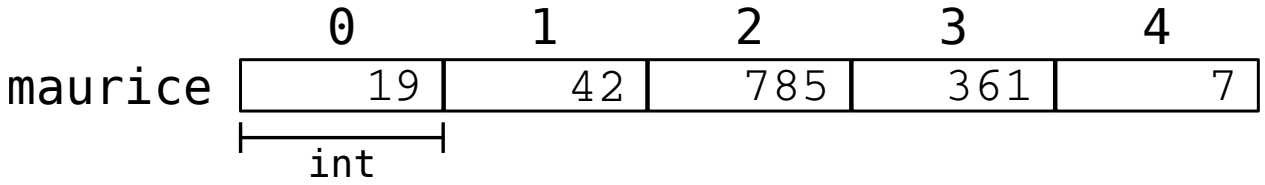
Les tableaux statiques : accès aux élements



Les éléments du tableau sont numérotés par des **indices** de 0 à $N - 1$, où N est la taille du tableau.

Ils représentent l'***offset*** (décalage) de l'élément à partir du début du tableau : c'est pour cela que le premier élément possède l'indice 0.

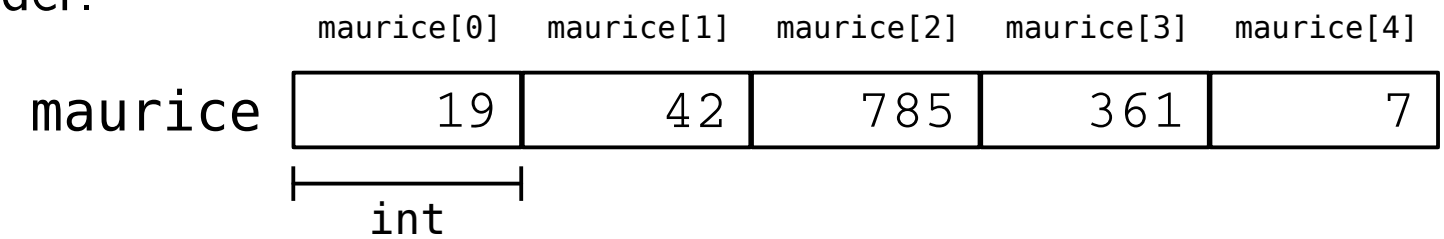
Les tableaux statiques : accès aux élements



Les éléments du tableau sont numérotés par des **indices** de 0 à $N - 1$, où N est la taille du tableau.

Ils représentent l'***offset*** (décalage) de l'élément à partir du début du tableau : c'est pour cela que le premier élément possède l'indice 0.

L'accès aux élements du tableau s'effectue au moyen de crochets [...] entre lesquels on place l'indice de l'élément auquel on veut accéder.



L'indice doit toujours être un **entier** !
Il peut être une valeur **constante** ou la valeur d'une **variable**.

Les tableaux statiques : accès aux élements

Attention à la numérotation des indices !

```
1 int v[3] = {4, 6, 8};
2 cout << "Element 1 = " << v[0] << endl; // affiche 4
3 cout << "Element 2 = " << v[1] << endl; // affiche 6
4 cout << "Element 3 = " << v[2] << endl; // affiche 8
5 v[1] = 12;
6 cout << "Element 2 = " << v[1] << endl; // affiche 12
```

Tout comme pour les tableaux 1D, chaque dimension d'un tableau à D dimensions est numérotée de 0 à $N_D - 1$ où N_D est la taille de la D-ième dimension du tableau.

```
1 int v[3][4] = { {1, 2, 3,  4},
2                 {5, 6, 7,  8},
3                 {8, 9, 10, 11} };
4 cout << v[1][2] << endl; // affiche 7
5 cout << v[2][0] << endl; // affiche 8
```

Les tableaux statiques : accès aux élements

- Erreur d'accès à un tableau
  L'utilisation d'un indice non compris entre 0 et $N - 1$ ne sera pas détectée par le compilateur (C/C++).
  Le programme tentera donc d'accéder à une partie de la mémoire qui n'a pas été réservée pour le tableau. Dans ce cas :

Les tableaux statiques : accès aux élements

- Erreur d'accès à un tableau
  L'utilisation d'un indice non compris entre 0 et $N-1$ ne sera pas détectée par le compilateur (C/C++).
  Le programme tentera donc d'accéder à une partie de la mémoire qui n'a pas été réservée pour le tableau. Dans ce cas :
  - Le programme peut s'arrêter avec une erreur du type "segmentation fault" (accès à une zone mémoire interdite par le système)

Les tableaux statiques : accès aux élements

- Erreur d'accès à un tableau
  L'utilisation d'un indice non compris entre 0 et $N-1$ ne sera pas détectée par le compilateur (C/C++).
  Le programme tentera donc d'accéder à une partie de la mémoire qui n'a pas été réservée pour le tableau. Dans ce cas :
  - Le programme peut s'arrêter avec une erreur du type "segmentation fault" (accès à une zone mémoire interdite par le système)
  - Parfois l'exécution du programme ne s'arrêtera pas. On accède alors à de la mémoire non allouée, qui peut correspondre à d'autres variables et être modifiée pendant l'exécution. Seules des erreurs de valeur et donc de fonctionnement du programme apparaitront dans ce cas : ce sont les plus difficiles à détecter.

Exemple en pratique

La plupart du temps, on utilise des boucles pour initialiser, traiter et afficher des tableaux. Dans ce cas, une variable entière est utilisée pour accéder aux éléments du tableaux.

```
1  int main()
2  {
3     int v[1000] = {0};
4     // initialiser avec des multiples de 2
5     for(int i = 0; i < 1000; i++)
6     {
7        v[i] = i*2;
8     }
9     return 0;
10 }
```

Nombres aléatoires

Il est fréquent en programmation d'avoir besoin de générer des **nombres aléatoires**. Pour ce faire, il existe une fonction

rand() ;

Cette fonction retourne un entier positif pseudo-aléatoire compris dans l'intervalle [0 :RAND_MAX].

Pour l'utiliser, il faut inclure la bibliothèque **cstdlib** :

```
1 #include <cstdlib>
```

Nombres aléatoires : manipulations

- Pour définir les limites de l'intervalle des nombres générés
  $\rightarrow$ Utilisation de la fonction modulo

```
1 int a = rand()%100; // nombre entre 0 et 99
```

Nombres aléatoires : manipulations

- Pour définir les limites de l'intervalle des nombres générés
  $\rightarrow$ Utilisation de la fonction modulo

```
1 int a = rand()%100; // nombre entre 0 et 99
```

- Obtenir un nombre réel
  $\rightarrow$ Convertir en réel et diviser

```
1 // nombre entre 0 et 1
2 double a = (double)rand()/RAND_MAX;
3
4 // nombre entre 10 et 20
5 double b = 10. + 10.*(double)rand()/RAND_MAX;
```

Nombres aléatoires : initialisation du générateur

Le générateur produira toujours la même suite de nombres (pseudo-aléatoires). Pour obtenir des nombres aléatoires qui ne soient pas identiques d'une exécution à l'autre, on utilise la fonction **srand(x)** qui permet de commencer la série à un endroit déterminé.

Cette valeur de départ x est appelée ***seed*** (graine) et doit être différente à chaque exécution.
On utilise généralement pour cette valeur le nombre de secondes écoulées depuis le 1^{er} Janvier 1970 (qui est donc différent à chaque exécution). Ce nombre est obtenu par la fonction **time(NULL)**.

```
1 // initialisation:
2 srand(time(NULL));
3 // nombre aléatoire différent à chaque exécution:
4 int a = rand();
```

La fonction **time** nécessite d'inclure la libraire <**ctime**>

Exercices (1/2)

1. Recherche dans un tableau
   - Créer un tableau de 50 éléments de type **double** ;
   - Le remplir de nombres aléatoires ;
   - Afficher le tableau ;
   - Déterminer l'indice du plus petit élément du tableau ;
   - Afficher la valeur de cet élément et l'indice correspondant.

2. Tri de tableau :
   Modifier l'exercice précédent pour trier les éléments du tableau du plus petit au plus grand.
   - Afficher le tableau non trié ;
   - Parcourir le tableau à la recherche du plus petit élément ;
   - Permuter cet élément avec le premier du tableau ;
   - Chercher le plus petit élément parmi ceux restants ;
   - Permuter cet élément avec le second du tableau ;
   - Poursuivre jusqu'à ce que le tableau soit trié ;
   - Afficher le tableau trié ;

Exercices (2/2)

3. Petit modèle météorologique
   - Écrire un petit programme qui va afficher les prévisions météo sur plusieurs jours (nombre au choix). Pour ce faire, procéder dans le cadre suivant :
     - Considérer un modèle binaire (seulement deux types de temps, par exemple : beau temps et pluie) ;
     - Se baser sur l'affirmation suivante donnée jadis par grand-père : "Demain verra paraître le même temps qu'aujourd'hui dans 60% des cas !" ;
     - S'aider évidemment des nombres aléatoires pour modéliser cette probabilité ;
     - A la fin des prévisions, afficher le nombre total de jours de beau temps, idem pour le nombre total de jours de pluie.

COMPLÉMENTS

Opérateurs d'affectation condensés

Il existe une série d'opérateurs qui permettent de réaliser une **opération** avec une variable et de remplacer immédiatement sa **valeur** par le **résultat** de l'opération.

Opérateurs d'affectation condensés

Il existe une série d'opérateurs qui permettent de réaliser une **opération** avec une variable et de remplacer immédiatement sa **valeur** par le **résultat** de l'opération.
Exemple avec l'addition :

```
1 int a = 2;
2 int b = 5;
3 a += b; // --> a = a + b
4 cout << "a = " << a << endl; // affiche 7
```

Opérateurs d'affectation condensés

Il existe une série d'opérateurs qui permettent de réaliser une **opération** avec une variable et de remplacer immédiatement sa **valeur** par le **résultat** de l'opération.
Exemple avec l'addition :

```
1 int a = 2;
2 int b = 5;
3 a += b; // --> a = a + b
4 cout << "a = " << a << endl; // affiche 7
```

De la même manière, on peut utiliser les opérateurs suivants :

+=	a+=b	$a = a + b$
-=	a-=b	$a = a - b$
*=	a*=b	$a = a * b$
/=	a/=b	$a = a/b$
%=	a%=b	$a = a\%b$

Opérateurs unaires

Il existe en C++ des **opérateurs** qui ne nécessitent qu'un seul **opérande** : ce sont les **opérateurs unaires** $++$ et $--$.

Ceux-ci servent respectivement à **incrémenter** et **décrémenter** la valeur d'une variable. Ceux-ci ont également un comportement différent selon qu'ils sont utilisés sous leur forme **préfixée** ou **suffixée**.

Opérateurs unaires

Il existe en C++ des **opérateurs** qui ne nécessitent qu'un seul **opérande** : ce sont les **opérateurs unaires** $++$ et $--$.

Ceux-ci servent respectivement à **incrémenter** et **décrémenter** la valeur d'une variable. Ceux-ci ont également un comportement différent selon qu'ils sont utilisés sous leur forme **préfixée** ou **suffixée**.

```
1 int a = 2, b = 0;
2 a++; // --> a = 3
3 b = a++; // --> b = 3, a = 4
4 b = ++a; // --> b = 5, a = 5
```

Opérateurs unaires

Il existe en C++ des **opérateurs** qui ne nécessitent qu'un seul **opérande** : ce sont les **opérateurs unaires** $++$ et $--$.

Ceux-ci servent respectivement à **incrémenter** et **décrémenter** la valeur d'une variable. Ceux-ci ont également un comportement différent selon qu'ils sont utilisés sous leur forme **préfixée** ou **suffixée**.

```
1 int a = 2, b = 0;
2 a++; // --> a = 3
3 b = a++; // --> b = 3, a = 4
4 b = ++a; // --> b = 5, a = 5
```

Lorsque l'opérateur est **préfixé**, l'opération est d'abord effectuée sur l'opérande et le résultat est utilisé dans l'expression correspondante.

Lorsque l'opérateur est **suffixé**, la valeur actuelle de l'opérande est utilisée dans l'expression puis l'opération est appliquée à l'opérande.

La bibliothèque standard du C++

Le langage C++ définit une série de fonctions et classes standardisées qui constituent la **bibliothèque standard du C++**. Ces fonctions sont supposées être toujours fournies avec le compilateur et peuvent donc être utilisées dans tous les programmes.

La bibliothèque standard du C++

Le langage C++ définit une série de fonctions et classes standardisées qui constituent la **bibliothèque standard du C++**.
Ces fonctions sont supposées être toujours fournies avec le compilateur et peuvent donc être utilisées dans tous les programmes.
Elles sont contenues dans plusieurs fichiers et doivent être inclues au début du programme grâce à l'instruction pré-processeur

```
1 #include <librairie>
```

La bibliothèque standard du C++

Le langage C++ définit une série de fonctions et classes standardisées qui constituent la **bibliothèque standard du C++**.
Ces fonctions sont supposées être toujours fournies avec le compilateur et peuvent donc être utilisées dans tous les programmes.
Elles sont contenues dans plusieurs fichiers et doivent être inclues au début du programme grâce à l'instruction pré-processeur

```
1 #include <librairie>
```

On trouve des fonctions et classes pour :

- Gérer l'affichage et la saisie au clavier (**cout**, **cin**)
  $\rightarrow$ contenu dans **iostream**
- Utiliser des fonctions mathématiques avancées
  $\rightarrow$ contenu dans **cmath**
- Traiter du texte et des chaînes de caractères
  $\rightarrow$ contenu dans **string**

La bibliothèque standard du C++ : espace de nom 'std'

Toutes les fonctions de la librairie standard sont contenues dans un **espace de nom** appelé "**std**".

Il est donc nécessaire de précéder toutes les fonctions qu'elle contient par "**std : :**"

```
1 std::cout << "texte" << std::endl;
```

La bibliothèque standard du C++ : espace de nom 'std'

Toutes les fonctions de la librairie standard sont contenues dans un **espace de nom** appelé "**std**".

Il est donc nécessaire de précéder toutes les fonctions qu'elle contient par "**std : :**"

```
1 std::cout << "texte" << std::endl;
```

Ou d'utiliser au préalable l'instruction

```
1 using namespace std;
```

qui implique que l'espace de nom **std** est utilisé par défaut et permet ainsi de se dispenser de la notation "**std : :**".

La bibliothèque “cmath” (1/2)

La bibliothèque **cmath** contient des fonctions mathématiques avancées. Parmi celles-ci, on trouve

La bibliothèque “cmath” (1/2)

La bibliothèque **cmath** contient des fonctions mathématiques avancées. Parmi celles-ci, on trouve

- les fonctions trigonométriques : **cos, sin, tan, acos,...**

La bibliothèque “cmath” (1/2)

La bibliothèque **cmath** contient des fonctions mathématiques avancées. Parmi celles-ci, on trouve

- les fonctions trigonométriques : **cos, sin, tan, acos,...**
- les fonctions exponentielles et logarithmiques : **exp, log,...**

La bibliothèque “cmath” (1/2)

La bibliothèque **cmath** contient des fonctions mathématiques avancées. Parmi celles-ci, on trouve

- les fonctions trigonométriques : **cos, sin, tan, acos,...**
- les fonctions exponentielles et logarithmiques : **exp, log,...**
- les puissances : **pow, sqrt,...**

La bibliothèque “cmath” (1/2)

La bibliothèque **cmath** contient des fonctions mathématiques avancées. Parmi celles-ci, on trouve

- les fonctions trigonométriques : **cos, sin, tan, acos,...**
- les fonctions exponentielles et logarithmiques : **exp, log,...**
- les puissances : **pow, sqrt,...**

```
1  #include <cmath>
2  using namespace std;
3
4  int main()
5  {
6      double a = cos(2.*3.1415926535); // --> a = 1
7      double b = sqrt(16); // --> b = 4
8      double c = pow(2, b); // --> c = 2 ^ b = 16
9      return 0;
10 }
```

La bibliothèque “cmath” (2/2)

Quelques exemples :

- **pow(x,y)** : calcule x à la puissance y (x^y)

La bibliothèque “cmath” (2/2)

Quelques exemples :

- **pow(x,y)** : calcule x à la puissance y (x^y)
- **sqrt(x)** : calcule la racine carrée de x ($\sqrt{x}$)

La bibliothèque “cmath” (2/2)

Quelques exemples :

- **pow(x,y)** : calcule x à la puissance y (x^y)
- **sqrt(x)** : calcule la racine carrée de x ($\sqrt{x}$)
- **log(x)** : calcule le logarithme naturel de x ($\ln(x)$)

La bibliothèque “cmath” (2/2)

Quelques exemples :

- **pow(x,y)** : calcule x à la puissance y (x^y)
- **sqrt(x)** : calcule la racine carrée de x ($\sqrt{x}$)
- **log(x)** : calcule le logarithme naturel de x ($\ln(x)$)
- **log10(x)** : calcule le logarithme en base 10 de x ($\log(x)$)

La bibliothèque “cmath” (2/2)

Quelques exemples :

- **pow(x,y)** : calcule x à la puissance y (x^y)
- **sqrt(x)** : calcule la racine carrée de x ($\sqrt{x}$)
- **log(x)** : calcule le logarithme naturel de x ($\ln(x)$)
- **log10(x)** : calcule le logarithme en base 10 de x ($\log(x)$)
- **fabs(x)** : calcule la valeur absolue de x ($|x|$)

La bibliothèque “cmath” (2/2)

Quelques exemples :

- **pow(x,y)** : calcule x à la puissance y (x^y)
- **sqrt(x)** : calcule la racine carrée de x ($\sqrt{x}$)
- **log(x)** : calcule le logarithme naturel de x ($\ln(x)$)
- **log10(x)** : calcule le logarithme en base 10 de x ($\log(x)$)
- **fabs(x)** : calcule la valeur absolue de x ($|x|$)
- **floor(x)** : calcule la valeur entière directement inférieure à x

La bibliothèque “cmath” (2/2)

Quelques exemples :

- **pow(x,y)** : calcule x à la puissance y (x^y)
- **sqrt(x)** : calcule la racine carrée de x ($\sqrt{x}$)
- **log(x)** : calcule le logarithme naturel de x ($\ln(x)$)
- **log10(x)** : calcule le logarithme en base 10 de x ($\log(x)$)
- **fabs(x)** : calcule la valeur absolue de x ($|x|$)
- **floor(x)** : calcule la valeur entière directement inférieure à x

Des informations très complètes concernant l'utilisation de ces fonctions et de nombreuses autres peuvent être obtenues à l'adresse http://www.cplusplus.com/reference/

Le type char

Le type **char** permet de stocker un **caractère** dans une variable.

En interne, il s'agit en fait d'un petit entier, permettant de stocker des valeurs comprises entre 0 et +127. Les valeurs numériques correspondent au code ASCII du caractère et peuvent ainsi représenter les différents caractères.
Exemple : le caractère "A" possède le code ASCII 64.

```
1 char caractere = 'A'; // caractère A == 64
2 char c = 97; // caractère a == 97
```

ATTENTION : L'affectation à un type **char** s'effectue avec des guillemets simples ('), contrairement au texte (chaînes de caractères) qui s'écrit avec des guillemets doubles (").

```
1 char g = "B"; // ERREUR, cela ne compilera pas !
```

Le code ASCII

Dec = Decimal; Hex = Hexadecimal; Char = Character

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(	72	48	H	104	68	h
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	□