Planning POO

Lundi 13/11	TP7 (2ème TP C++)
Lundi 20/11	TP8 (3ème TP C++)
Lundi 27/11	TP9 (1ère évaluation C++ - 30% de la note finale)
...	
Mardi 05/12	Dernier cours C++
Lundi 11/12	TP11 (6ème TP C++)
Lundi 18/12	TP12 (TP noté C++ - 30% de la note finale)

- Le TP noté python du 16/10 compte pour 40% de la note finale (les notes vous seront communiquées cette semaine)

Tableaux & pointeurs

Exemple

```cpp
#include <iostream>

using namespace std;

int main()
{
  int t[10];

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

  for(int i=0; i<10; i++)
    cout << "t[" << i << "]: " << t[i]
         << endl;

  return 0;
}
```

Listing 1: code14.cpp

- La déclaration *int t[10];* réserve en mémoire l'emplacement pour 10 éléments de type entier.
- Dans la première boucle, on initialise chaque élément du tableau, le premier étant conventionnellement numéroté 0.
- Dans la deuxième boucle, on parcourt le tableau pour afficher chaque élément.
- On notera qu'on utilise la notation [] pour l'accès à un élément du tableau.

Quelques règles

Il ne faut pas confondre les éléments d'un tableau avec le tableau lui-même.
Ainsi, par exemple, *t[2] = 3; t[0]++;* sont des écritures valides.

Mais écrire t1 = t2, si t1 et t2 sont des tableaux, n'est pas possible!

Il n'existe pas, en C++, de mécanisme d'affectation globale pour les tableaux.

Quelques règles

Les indices peuvent prendre la forme de n'importe quelle expression arithmétique d'un type entier.

Par exemple, si *n*, *p*, *k* et *j* sont de type *int*, il est valide d'écrire:

```
t[n-3], t[3*p-2*k+j%4]
```

Il n'existe pas de mécanisme de contrôle des indices! Il revient au programmeur de ne pas écrire dans des zones mémoire qu'il n'a pas allouées.
Source de nombreux bugs ...

Quelques règles

En C ANSI et en iso C++, la dimension d'un tableau (=son nombre d'éléments) ne peut être qu'une constante, ou une expression constante. Certains compilateurs acceptent néanmoins le contraire en tant qu'extension du langage.

```
const int N = 50;
int t[N]; // Valide quels que soient la norme et le
    compilateur
int n = 50;
int t[n]; // n'est pas valide systématiquement -
    utilisation déconseillée!
```

Tableaux à plusieurs indices

On peut écrire:

```
int t[5][3];
```

pour réserver un tableau de 15 éléments (5*3) de type entier.
On accède alors à un élément en jouant sur les deux indices du tableau.

Le nombre d'indices peut être quelconque en C++. On prendra néanmoins en compte les limitations de la machine, comme la quantité de mémoire à disposition.

Initialisation d'un tableau

```
#include <iostream>
using namespace std;

int main()
{
  int t[6] = {0, 3, 4, 7, 9, 13};

  for (int i=0; i<6; i++)
    cout << t[i] << ";" ;
  cout << endl;

  return 0;
}
```

Listing 2: code16.cpp

Nous avons déjà initialisé des tableaux grâce à des boucles.

On peut en fait les initialiser "en dur" lors de leur déclaration.

On utilisera alors la notation {} comme dans l'exemple ci-contre.

Pointeurs

Un pointeur est une adresse mémoire. C'est un nombre hexadécimal et il se définit à partir d'un type (*int*, *float*, ...).

Exemples de déclarations de pointeurs:

```
int *v; // pointeur sur int
float* q; // pointeur sur float
double* p; // pointeur sur double
```

Pour écrire la déclaration d'un seul pointeur, on peut placer '*' comme on veut: le coller au type, au nom de la variable ou laisser un espace entre les deux.
Mais pour déclarer simultanément deux pointeurs sur double, p1 et p2, on écrira:

```
double *p1, *p2; // p1 et p2 pointeurs sur double
double* q1, q2; //q1 pointeur sur double, q2 double!
```

Pointeurs - Les opérateurs * et &

```
#include <iostream>
using namespace std;

int main()
{
  int *ptr;
  int i = 42;

  ptr = &i;
  cout << "ptr = " << ptr << endl;
  cout << "*ptr = " << *ptr
       << endl;
  return 0;
}
```

Listing 3: code17.cpp

On commence par déclarer une variable *ptr* de type *int **: un pointeur sur entier.
Puis une variable *i* de type entier.

On assigne à *ptr* **l'adresse** en mémoire de la variable *i*, grâce à l'opérateur &.

On affiche ensuite *ptr*: une **adresse**, une valeur qui sera affichée en hexadécimal.

Puis on affiche la valeur pointée par *ptr* (la même que la valeur de *i*). On dit que l'on a **déréférencé** le pointeur *ptr*.

```
$ ./a.out
ptr = 0x7ffde3edb19c
*ptr = 42
```

Pointeurs - Les opérateurs * et &

```cpp
#include <iostream>
using namespace std;

int main()
{
  int *ptr;
  int i = 42;

  ptr = &i;
  cout << "ptr =  " << ptr << endl;
  cout << "*ptr = " << *ptr << endl;
  i += 3;
  cout << "i = " << i << ", *ptr = " << *ptr
       << endl;
  *ptr = 122;
  cout << "*ptr = " << *ptr << ", i = " << i
       << endl;
  return 0;
}
```

Listing 4: code17V2.cpp

```
$ ./a.out
ptr = 0x7ffde3edb19c
*ptr = 42
i = 45, *ptr = 45
*ptr = 122, i = 122
```

Pointeurs - Les opérateurs * et &

	adresses	valeurs	variables
mémoire	0x0		
	⋮		
	0x42	0xffffcbf4	ptr
	0x43		
	⋮		
	0xffffcbf4	42	i
	0xffffcbf5		
	0xffffcbf6		
	0xffffcbf7		
	⋮	⋮	⋮

Voici une représentation schématisée de l'exemple précédent.
On voit bien que la valeur de la variable *ptr* est l'adresse en mémoire de la variable *i*.
Le type du pointeur est important: il permet de connaître la taille en mémoire de la valeur pointée!

Pour un type entier, il s'agira des 4 octets suivant l'adresse *0xffffcbf4*.
La taille du pointeur lui-même varie en fonction du nombre de bits du système : 16, 32, ou pour les machines actuelles: 64 bits.

Relation tableaux et pointeurs

En C++, l'identificateur d'un tableau (càd son nom, sans indice à sa suite) est considéré comme un pointeur.

Par exemple, lorsqu'on déclare le tableau de 10 entiers

```
int t[10];
```

La notation t est équivalente à &t[0], c'est-à-dire à l'adresse de son premier élément.
La notation &t est possible aussi (c'est équivalent à t).

Pointeurs - Arithmétique des pointeurs

Une adresse est une valeur entière. Il paraît donc raisonnable de pouvoir lui additionner ou lui soustraire un entier, en suivant toutefois des règles particulières.

Que signifie ajouter 1 à un pointeur sur entier ? Est-ce la même chose que pour un pointeur sur char par exemple?

Non

Ajouter 1 à un pointeur a pour effet de le décaler en mémoire du nombre d'octets correspondant à la taille du type pointé.
En ajoutant (soustrayant) 1 à un pointeur sur *int* (*float*, *double*, *char* ...), on le décale en mémoire de la taille d'un *int* (resp. *float*, *double*, *char* ...).

On appelle ce mécanisme l'arithmétique des pointeurs.

Relation tableaux et pointeurs

On sait maintenant qu'un tableau peut être considéré comme un pointeur.
Plus précisément, il s'agit d'un pointeur constant.
Pour accéder aux éléments d'un tableau, on a donc deux possibilités :

- La notation indicielle : t[5]
- La notation pointeur : *(t+5)

Attention:

- La priorité des opérateurs est importante : *(t+5) $\neq$ *t + 5
- Un nom de tableau est un pointeur constant! On ne peut pas écrire *tab*+=1 ou *tab*=*tab*+1 ou encore *tab*++ pour parcourir les éléments d'un tableau.

Pointeurs particuliers

- **Le pointeur nul** (noté NULL), dont la valeur vaut 0.
  Il est utile car il permet de désigner un pointeur ne pointant sur rien.
  Evidemment déréférencer le pointeur nul conduit irrémédiablement à une erreur de segmentation.
- **Le pointeur générique void ***.
  Un pointeur est caractérisé par deux informations: la valeur de l'adresse pointée et la taille du type pointé.
  *void ** ne contient que l'adresse. Il permet donc de manipuler n'importe quelle valeur sans souci de la taille du type. C'était un type très utile en C, notamment pour écrire des fonctions génériques valables quel que soit le type des données.

Allocation statique et dynamique

<table>
<tr><td rowspan="6">MÉMOIRE</td><td rowspan="3">PILE
(stack)</td><td>main</td><td>variables de la fonction main</td></tr>
<tr><td>fct_1</td><td>variables et arguments de la fonction fct_1
appelée dans main</td></tr>
<tr><td>fct_2</td><td>variables et arguments de la fonction fct_2
appelée dans fct_1</td></tr>
<tr><td colspan="3">La pile peut grandir en occupant la mémoire libre</td></tr>
<tr><td colspan="3">mémoire libre</td></tr>
<tr><td colspan="3">Le tas peut grandir en occupant la mémoire libre</td></tr>
<tr><td></td><td>TAS
(heap)</td><td colspan="2">Le tas offre un espace de mémoire dite d'allocation dynamique. C'est un espace mémoire qui est géré par le programmeur, en faisant appel aux opérateurs d'allocation new pour allouer un espace et delete pour libérer cet espace.</td></tr>
</table>

Ceci est une représentation schématisée de la mémoire occupée par un processus au cours de son exécution.

On connaît déjà la **pile** (ou stack en anglais) qui contient les variables et les tableaux que l'on a déclarés jusqu'à présent.

Le **tas** (ou heap) est une zone de la mémoire qui peut grandir au fil de l'exécution et dont le contenu est géré par le programmeur. Mais,
"Un grand pouvoir implique de grandes responsabilités!"

Les opérateurs new et delete

- *new* est un opérateur unaire prenant comme argument un type.
- La syntaxe est: *new montype* où *montype* représente un type quelconque.
- Il renvoie un pointeur de type *montype** dont la valeur est l'adresse de la zone mémoire allouée pour notre donnée de type *montype*. Par exemple:

```
int *ptr = new int;
```

- On peut maintenant utiliser notre pointeur pour accéder à un entier dont on a alloué l'espace mémoire.
- **(Important)** Une autre syntaxe permet d'allouer un espace mémoire contigü pour plusieurs données à la fois. Le pointeur renvoyé, toujours de type *montype**, pointe vers la première valeur allouée.

```
int* ptr2 = new int[10];
```

$\Rightarrow$ On peut donc utiliser *new* pour manipuler des tableaux.

Les opérateurs new et delete

- Attention car on ne peut évidemment pas allouer indéfiniment de la mémoire, celle-ci étant finie. Un programme trop gourmand risque de mettre le système entier en danger et bien souvent celui-ci préfèrera le terminer de manière brutale.
- *delete* est l'opérateur permettant de faire le ménage dans la mémoire en libérant l'espace qui ne sert plus.
- Lorsqu'un pointeur *ptr* a été alloué par *new*, on écrira alors *delete ptr* pour le libérer.
- S'il s'agit d'un tableau, on écrira *delete [] ptr;*

Les opérateurs new et delete

Remarques:

- Des précautions doivent être prises lors de l'utilisation de *delete*.
- *delete* ne doit pas être utilisé pour des pointeurs déjà détruits.
- *delete* ne doit pas être utilisé pour des pointeurs obtenus autrement que par l'utilisation de *new*.
- Une fois un pointeur détruit, on doit évidemment arrêter de l'utiliser.

Exemple

```
#include <iostream>
using namespace std;
double sum(double *val, int n)
{
  double res = 0;
  for (int i=0; i<n; i++)
    res += val[i];
  return res;
}

int main()
{
  int n;
  double *val;
  cout << "nombres de valeurs: ";
  cin >> n;
  val = new double[n];
  for (int i = 0; i<n; i++) {
    cout << i << ": " ;
    cin >> val[i];
  }
  cout << "Moyenne de ces valeurs: " <<
          sum(val,n)/n << endl;
  delete [] val;
  return 0;
}
```

Listing 5: code18.cpp

Ce programme calcule la moyenne des valeurs que l'utilisateur entre au clavier durant l'exécution. Mais le nombre de ces valeurs varie aussi! Si nous avions alloué un tableau statiquement, sur la pile, comme jusqu'à présent, nous aurions dû entrer une valeur constante pour sa taille qui aurait pu être soit trop grande, soit trop petite.
On notera

- l'utilisation de *delete* qui permet de libérer notre pointeur proprement à la fin de notre programme.
- L'utilisation du pointeur *val* avec la notation indicielle [], au lieu d'utiliser l'arithmétique des pointeurs.

Initialisation avec new

On peut initialiser en une seule ligne un tableau alloué avec *new* de la manière suivante:

```
...
double * v;
v = new double[5] {-1.1, 0., 0.2, 0.4, 12.1};
....
delete [] v;
....
```

Pointeurs sur fonctions

Lorsqu'un exécutable est chargé en mémoire, ses fonctions le sont évidemment aussi. Par voie de conséquence, elles ont donc une adresse, que C++ permet de pointer.

Si nous avons une fonction dont le prototype est le suivant :

```
int fct(double, double);
```

Un pointeur sur cette fonction sera déclaré de la façon suivante :

```
int (* fct_ptr)(double, double); // le pointeur s'appellera fct_ptr
```

On notera l'utilisation des parenthèses.

En effet, écrire

```
int * fct(double, double) //même chose que int* fct(double, double)
```

ne signifie pas du tout la même chose.

Pointeurs sur fonctions

```
#include <iostream>
using namespace std;

double fct1(double x){
  return x*x;
}
double fct2(double x){
  return 2*x;
}
void apply(double *val, int n, double (*fct
     )(double)){
  for (int i=0; i<n; i++)
    val[i]= (*fct)(val[i]);
}

void aff_tab(double *val, int n){
  for (int i=0; i<n; i++)
    cout << i << ": " << val[i] << endl;
}
int main()
{
  double t[10] = {1,2,3,4,5,6,7,8,9,10};
  aff_tab(t, 10);
  apply(t, 10, fct1);
  cout << "Apres apply (elevation au carre)
       : " << endl;
  aff_tab(t, 10);
}
```

Listing 6: code19.cpp

On définit deux fonctions ayant même valeur de retour et même type d'argument, *fct1* et *fct2*.
La fonction *aff_tab* n'est là que pour aider et affiche un tableau de doubles donné en paramètre.
La nouveauté se situe dans la fonction *apply* qui va appliquer la fonction passée en paramètre sur chaque élément d'un tableau de n éléments, à l'aide d'un pointeur.
On notera que, pour appeler la fonction pointée, il faut la déréférencer, toujours à l'aide de l'opérateur *.
Cela permet d'écrire des fonctions génériques puissantes et se passer de l'écriture de code redondants!

En effet, on aurait pu appliquer des fonctions de la librairie *cmath* comme *cos* ou *sqrt*, sans réécrire pour chaque cas une boucle pour l'appliquer à chaque élément de ce tableau.

Chaînes de caractères en C++

```cpp
#include <iostream>

using namespace std;

int main()
{
  char *str = "Hello World"; // char*
  char str2[10] = {'C','o','u','c','o','u','\0'};
        // char[10]
  string str3 = "Bonjour"; // string

  cout << str << endl;
  cout << str2 << endl;
  cout << str3 << endl;

  return 0;
}
```

Listing 7: code20.cpp

```
 ./code20
Hello World
Coucou
Bonjour
```

Les chaînes de caractères, telles qu'elles sont représentées en C, sont toujours valables en C++.

Il existe trois manières de manipuler les chaînes de caractères en C++, comme on peut le voir dans l'exemple de gauche.

- En utilisant *char**, comme en C (attention, ne nécessite pas *new*!)
- Avec un tableau de char, d'une taille donnée
- En utilisant le type *string* (plus simple)

Chaînes de caractères en C++

```cpp
#include <iostream>

using namespace std;

int main()
{
  char *str = "Hello World"; // char*
  char str2[10] = {'C','o','u','c','o','u','\0'};
        // char[10]
  string str3 = "Bonjour"; // string

  cout << str << endl;
  cout << str2 << endl;
  cout << str3 << endl;

  return 0;
}
```

Listing 7: code20.cpp

```
 ./code20
Hello World
Coucou
Bonjour
```

En C, les chaînes de caractères peuvent êtres vues comme des tableaux de *char*.

Il y a néanmoins une convention supplémentaire.

Une chaîne de caractères est donc un ensemble d'octets contigus se terminant par le caractère nul noté \0, ceci afin de donner une fin à la chaîne.

La notation *"Hello World"* définit donc un pointeur sur caractères vers une zone de la mémoire où est définie la constante *"Hello World"*. On récupère cette adresse dans le pointeur *str*.

Chaînes de caractères en C++

```cpp
#include <iostream>

using namespace std;

int main()
{
  char *str = "Hello World"; // char*
  char str2[10] = {'C','o','u','c','o','u','\0'};
        // char[10]
  string str3 = "Bonjour"; // string

  cout << str << endl;
  cout << str2 << endl;
  cout << str3 << endl;

  return 0;
}
```

Listing 7: code20.cpp

```
 ./code20
Hello World
Coucou
Bonjour
```

On peut tout aussi bien définir une chaîne en déclarant un tableau et en l'initialisant avec la notation $\{\}$ comme dans l'exemple.

Pour les afficher, on utilise *cout*.

Les méthodes ne sont pas tout à fait équivalentes. En effet on peut modifier *str2[0]* mais pas *str[0]*. Et il est possible (mais compliqué) de modifier le premier caractère de *str3*.

Chaînes de caractères en C++

```
#include <iostream>
using namespace std;

int main(int nb, char *args[])
{
  cout << "Mon programme possede "
       << nb << " arguments" << endl;
  for (int i=0; i<nb; i++)
    cout << args[i] << endl;
  return 0;
}
```

Listing 8: code21.cpp

```
$ ./a.out salut tout le monde
Mon programme possède 5 arguments
./a.out
salut
tout
le
monde
```

On peut passer à un programme des valeurs, lorsqu'on l'appelle sur la ligne de commande, dans le terminal.
Le programme les reçoit comme des arguments de la fonction *main* (dont nous n'avons pas parlé jusqu'ici).

- Le premier argument de la fonction est conventionnellement un entier qui correspond au nombre d'arguments fournis au programme.
- Le deuxième paramètre est un peu plus complexe. Il s'agit d'un tableau de pointeurs sur *char* de taille non définie.

Chaque élément de ce tableau est donc un pointeur vers une chaîne de caractères qui existe quelque part en mémoire. On peut donc le balayer, comme dans la boucle de l'exemple. Chaque élément *args[i]* est donc de type *char** et peut être considéré comme une chaîne de caractères.

Conventionnellement, le premier élément *args[0]* est le nom du programme exécuté.

Structures

Les structures en C++

- Jusqu'à présent, nous avons rencontré les tableaux qui étaient un regroupement de données de même type.
- Il peut être utile de grouper des données de types différents et de les regrouper dans une même entité.
- En C, il existait déjà un tel mécanisme connu sous le nom de structure.

  C++ conserve ce mécanisme, tout en lui ajoutant de nombreuses possibilités.

  Ainsi, nous allons créer de nouveaux types de données, plus complexes, à partir des types que nous connaissons déjà.

Déclaration d'une structure

```cpp
#include <iostream>
#include "struct.hh"

using namespace std;

int main()
{
  Personne p1;

  p1.age = 35;
  p1.poids = 55.7;
  p1.taille = 160.5;
  cout << p1.age << " "
       << p1.poids << " "
       << p1.taille
       << endl;
  return 0;
}
```

Listing 9: code22.cpp

```cpp
#ifndef __STRUCT_HH__
#define __STRUCT_HH__

struct Personne
{
  int age;
  double poids;
  double taille;
};

#endif
```

Listing 10: struct.hh

Dans cet exemple nous commençons par déclarer la structure *Personne* dans un fichier *struct.hh*.

Ce n'est pas obligatoire, nous aurions pu déclarer cette structure dans le fichier contenant la fonction *main*, avant de l'utiliser, mais c'est une bonne habitude qui deviendra de plus en plus importante au fur et à mesure que nous avancerons dans ce cours.

Déclaration d'une structure (directives dans le header)

```cpp
#include <iostream>
#include "struct.hh"

using namespace std;

int main()
{
  Personne p1;

  p1.age = 35;
  p1.poids = 55.7;
  p1.taille = 160.5;
  cout << p1.age << " "
       << p1.poids << " "
       << p1.taille
       << endl;
  return 0;
}
```

Listing 9: code22.cpp

```cpp
#ifndef __STRUCT_HH__
#define __STRUCT_HH__

struct Personne
{
  int age;
  double poids;
  double taille;
};

#endif
```

Listing 10: struct.hh

On aurait pu se passer des 3 lignes commençant par *#ifndef*, *#define* et *#endif* dans *struct.hh* car le code est très simple ici. Mais dans le cas général, il est fortement conseillé de les inclure!

Si on les inclut, les 3 lignes sont indispensables; si l'une manque, ça donne une erreur de compilation!

En général, le nom qui suit *#ifndef* est celui du fichier d'entête en majuscules, précédé et terminé par "__" et dans lequel on remplace "." par "_".

Les noms après *#ifndef* et *#define* doivent être identiques.

Déclaration d'une structure

```cpp
#include <iostream>
#include "struct.hh"

using namespace std;

int main()
{
  Personne p1;

  p1.age = 35;
  p1.poids = 55.7;
  p1.taille = 160.5;
  cout << p1.age << " "
       << p1.poids << " "
       << p1.taille
       << endl;
  return 0;
}
```

Listing 9: code22.cpp

```cpp
#ifndef __STRUCT_HH__
#define __STRUCT_HH__

struct Personne
{
  int age;
  double poids;
  double taille;
};

#endif
```

Listing 10: struct.hh

Une structure se déclare grâce au mot-clé *struct* suivi du nom de la structure.

La structure Personne devient alors un type de données.
Ce type est le regroupement d'un entier et de deux doubles.
Dans la fonction *main*, après avoir inclus le fichier header au début du code, nous pouvons déclarer une variable de type *Personne*.

Cette variable s'appellera *p1*.

Déclaration d'une structure

```cpp
#include <iostream>
#include "struct.hh"

using namespace std;

int main()
{
  Personne p1;

  p1.age = 35;
  p1.poids = 55.7;
  p1.taille = 160.5;
  cout << p1.age << " "
       << p1.poids << " "
       << p1.taille
       << endl;
  return 0;
}
```

Listing 9: code22.cpp

```cpp
#ifndef __STRUCT_HH__
#define __STRUCT_HH__

struct Personne
{
  int age;
  double poids;
  double taille;
};

#endif
```

Listing 10: struct.hh

Les valeurs des différents types contenus dans la structure sont appelées des champs.

Pour accéder aux champs d'une variable dont le type est une structure, on utilisera l'opérateur '.' suivi du nom du champ. Ainsi *p1.age* est de type entier et on peut lui associer une valeur pour l'afficher ensuite.

Initialisation

```
#include <iostream>
using namespace std;

struct Personne
{
  int age;
  double poids;
  double taille;
};

int main()
{
  Personne toto = {35, 78, 168.5};

  cout << toto.age << " "
       << toto.poids << " "
       << toto.taille << endl;
  return 0;
}
```

Listing 11: code23.cpp

Dans l'exemple précédent, nous initialisions la structure en attribuant une valeur à chacun de ses champs.

Dans certains cas, cela peut s'avérer long et peu pratique.

Une autre façon est d'initialiser les champs de la structure au moment de son instanciation à la manière d'un tableau, grâce à l'opérateur {}.

Structures contenant des tableaux

```
#include <iostream>

using namespace std;

struct NamedPoint
{
  double x;
  double y;
  char nom[10];
};

int main()
{
  NamedPoint pt = {0,0, "Origine"};

  cout << " nom " << pt.nom
       << " x : " << pt.x
       << " y : " << pt.y << endl;
}
```

Listing 12: code24.cpp

Une structure peut contenir un tableau. La taille de celui-ci sera réservée en mémoire quand une variable issue de cette structure sera créée.

On notera l'initialisation de la structure dans l'exemple ci-contre.

Tableaux de structures

```
#include <iostream>
using namespace std;
struct NamedPoint
{
  double x;
  double y;
  char nom[10];
};
int main()
{
  NamedPoint pt[3] = {{0,0, "Origine"},
                      {1,0, "X"},
                      {0,1, "Y"}};
  for (int i=0; i<3; i++)
    cout << " nom : " << pt[i].nom
         << ", x : " << pt[i].x
         << ", y : " << pt[i].y
         << endl;
}
```

Listing 13: code25.cpp

On peut également créer des tableaux contenant des instances d'une même structure.
Dans l'exemple, on déclare un tableau de 3 points, que l'on initialise. Chaque élément du tableau est initialisé avec la notation $\{\}$ et lui-même est initialisé comme cela.
Puis on parcourt le tableau avec une boucle for pour en afficher chaque champ.
On fera attention au type de chaque élément :

- *pt* est un tableau de 3 *NamedPoint*,
- *pt[0]* est de type *NamedPoint*,
- *pt[0].nom* est un tableau de 10 *char*.

Structures imbriquées

```
#include <iostream>
using namespace std;

struct Date
{
  int jour;
  int mois;
  int annee;
};
struct Valeur
{
  double x;
  Date date;
};
int main()
{
  Valeur v = {5.5, {2,4,2017}};
  cout << "A la date: " << v.date.jour
       << "/" << v.date.mois << "/"
       << v.date.annee << endl
       << "Valeur: " << v.x << endl;
}
```

Listing 14: code26.cpp

Créer une structure revient à créer un nouveau type. On peut donc utiliser ce nouveau type comme champ d'une autre structure comme dans l'exemple ci-contre.

Ici encore, on notera le type des objets sur lesquels on travaille:

- *v* est de type *Valeur*
- *v.x* est un *double*
- *v.date* est de type *Date*
- *v.date.jour* est un entier

Structures et fonctions

```
#include <iostream>
#include <cmath> //pour utiliser pow
using namespace std;

struct Point
{
  double x;
  double y;
};
double myNorme(Point p, Point q)
{
  return sqrt(pow(p.x - q.x, 2)
         + pow(p.y - q.y, 2));
}
int main()
{
  Point a = {2, 2};
  Point b = {3, 3};
  cout << "norme: " << myNorme(a,b)
       << endl;
}
```

Listing 15: code27.cpp

On crée une fonction *myNorme* calculant la norme euclidienne de deux points de $\mathbb{R}^2$ définissant un vecteur.

On remarque au passage l'emploi de fonctions de la librairie *cmath*.

Les deux *Point* a et b sont passés à la fonction par copie.
myNorme reçoit donc une copie des points et non les points eux-mêmes.

C'est le passage par valeur. Si on modifie les valeurs des champs dans myNorme, ceux-ci ne sont pas modifiés à l'extérieur de la fonction.

Structures et fonctions

```
#include <iostream>
#include <cmath> //pour utiliser pow
using namespace std;

struct Point
{
  double x;
  double y;
};
double myNorme(Point & p, Point & q)
{
  return sqrt(pow(p.x - q.x, 2) +
              pow(p.y - q.y, 2));
}
int main()
{
  Point a = {2, 2};
  Point b = {3, 3};
  cout << "norme: " << myNorme(a,b)
       << endl;
}
```

Listing 16: code27_V2.cpp

Cette fois-ci le passage se fait par référence. Rien ne change à part l'entête de la fonction.

Ici, les valeurs des champs des paramètres ne changent pas (on aurait pu (dû!) les déclarer *const*).

Structures et fonctions

```
#include <iostream>
#include <cmath> //pour utiliser pow
using namespace std;
struct Point
{
  double x;
  double y;
};
double myNorme(Point* p, Point* q)
{
  return sqrt(pow(p->x - q->x, 2) +
              pow(p->y - q->y, 2));
}

int main()
{
  Point a = {2, 2};
  Point b = {3, 3};
  cout << "norme: "
       << myNorme(&a, &b) << endl;
}
```

Listing 17: code27_V3.cpp

On passe maintenant des pointeurs sur *Point* à notre fonction. On voit que l'appel de la fonction s'en trouve modifié: on ne passe plus les *Point* eux-mêmes mais leur adresse (obtenue grâce à l'opérateur &).

Le corps de la fonction aussi a changé.

Utiliser l'opérateur '.' n'a plus de sens si on travaille sur un pointeur. Un pointeur n'est pas du même type qu'une structure! C'est une adresse!

Structures et fonctions

```
#include <iostream>
#include <cmath> //pour utiliser pow
using namespace std;
struct Point
{
  double x;
  double y;
};
double myNorme(Point* p, Point* q)
{
  return sqrt(pow(p->x - q->x, 2) +
              pow(p->y - q->y, 2));
}

int main()
{
  Point a = {2, 2};
  Point b = {3, 3};
  cout << "norme: "
       << myNorme(&a, &b) << endl;
}
```

Listing 17: code27_V3.cpp

Si nous avions voulu utiliser l'opérateur '.' quand même, nous aurions pu le faire, au prix d'une écriture un peu lourde.

En effet, si on déréférence le pointeur sur *Point*, on obtient un objet de type *Point*, et ainsi le traiter comme tel ...

C++ introduit une facilité syntaxique pour éviter cela: l'opérateur flèche '->'

Ainsi,
$(*p).x$ est équivalent à $p\text{->}x$

Fonctions membres

```
#include <iostream>
using namespace std;

struct Point{
  double x; double y;
  void initialise(double, double);
  void deplace(double dx, double dy);
  void affiche();
};
void Point::initialise(double abs,
    double ord){ x = abs; y = ord;}
void Point::deplace(double dx, double
    dy){ x+= dx; y += dy;}
void Point::affiche(){
  cout << "x : " << x << " y : "
       << y << endl;}

int main(){
  Point p;
  p.initialise(1,4); p.deplace(0.1, 2);
  p.affiche(); }
```

Listing 18: code28.cpp

On ne se contente plus de données dans la structure. On ajoute aussi des fonctions:

- Une fonction *initialise* qui prend deux paramètres qui seront destinés à initialiser les coordonnées de notre *Point*.
- Une fonction *deplace*, qui prend deux paramètres et qui modifiera les coordonnées en fonction.
- Une fonction *affiche* qui génère un affichage de notre point.

Fonctions membres

```
#include <iostream>
using namespace std;

struct Point{
  double x; double y;
  void initialise(double, double);
  void deplace(double dx, double dy);
  void affiche();
};
void Point::initialise(double abs,
    double ord){ x = abs; y = ord;}
void Point::deplace(double dx, double
    dy){ x+= dx; y += dy;}
void Point::affiche(){
  cout << "x : " << x << " y : "
       << y << endl;}

int main(){
  Point p;
  p.initialise(1,4); p.deplace(0.1, 2);
  p.affiche(); }
```

Listing 18: code28.cpp

Les fonctions *initialise*, *deplace* et *affiche* sont des fonctions membres de la structure *Point*.

On les déclare dans la structure.

On les définit à l'extérieur de celle-ci MAIS le nom de la structure doit apparaître, suivi de :: appelé **opérateur de résolution de portée**.

En effet, comment connaître x et y dans la fonction si le compilateur ne sait pas qu'elles appartiennent à *Point*?

Un code ordonné

```
#ifndef __POINT_HH__
#define __POINT_HH__

#include <iostream>
using namespace std;

struct Point{
  double x;
  double y;
  void initialise(double,
       double);
  void deplace(double dx,
       double dy);
  void affiche();
};

#endif
```

Listing 19: Point.hh

```
#include "Point.hh"
void Point::initialise(double abs,
     double ord){
  x = abs;
  y = ord;
}
void Point::deplace(double dx,
     double dy){
  x += dx;
  y += dy;
}
void Point::affiche(){
  cout << "x : " << x
       << " y : " << y << endl;
}
```

Listing 20: Point.cpp

On sépare la déclaration de la structure, de sa définition. On crée un fichier *NomStructure.hh* (par ex.) destiné à la déclaration. Et le fichier *NomStructure.cpp* contiendra le code définissant les fonctions membres.

Un code ordonné

```
#include <iostream>
#include "Point.hh"

int main(){
  Point x;

  x.initialise(1,1);
  x.deplace(0.1, -0.1);
  x.affiche();
}
```

Listing 21: ex_struct_fct_membres.cpp

Voilà à quoi va ressembler notre fichier contenant la fonction *main* désormais.
On inclut le fichier header *Point.hh*.
Ainsi le compilateur connaîtra le type *Point*.
Pour la compilation, on compile en même temps les deux fichiers .cpp pour créer notre exécutable.
Il y a d'autres méthodes de compilation, à l'aide de **make** par exemple.

```
g++ ex_struct_fct_membres.cpp Point.cpp
  ./a.out
  x : 1.1 y : 0.9
```

Classes et Objets

Introduction

Nous entrons donc maintenant dans la partie **Programmation Orientée Objet** de ce cours.

Nous avons vu les structures: ce sont des types définis par l'utilisateur qui contiennent à la fois des données, mais aussi des fonctions membres.

En fait, en C++, ces structures sont un cas particulier d'un mécanisme plus général, les Classes.

Introduction

Pourquoi ne pas simplement travailler sur des structures ?

Les structures existent en C++ par souci de compatibilité avec C. Les Classes peuvent être considérées comme les structures mais leur comportement par défaut vis-à-vis de l'encapsulation est différent. De plus, le terme 'structure' ne renvoie pas à la terminologie Objet aussi bien que le terme 'classe', plus commun.

L'encapsulation fait partie intégrante de la POO. Elle permet de masquer certains membres et fonctions membres au monde extérieur.

Dans l'idéal, on ne devrait jamais accéder aux données d'une classe, mais agir sur ses méthodes (fonctions membres).

Déclaration

```
#ifndef __POINT2D_V1_HH__
#define __POINT2D_V1_HH__

class Point2D
{
private:
  float _x;
  float _y;

public:
  void initialise(float, float);
  void affiche();
};

#endif
```

Listing 22: Point2D_V1.hh

Une classe se déclare comme une structure.
Les étiquettes *private* et *public* sont utiles pour définir le niveau de visibilité des membres à l'extérieur de la classe.

Les membres déclarés *private* ne sont pas accessibles par des objets d'un autre type.
Les membres dits publics sont accessibles partout.
Ici, on ne pourra donc plus écrire, dans la fonction *main*:

```
Point2D p;
p._x = 5; //erreur-membre privé!
```

Par convention, les noms des membres privés commencent par '_' (tiret du bas).

private/public

```
#ifndef __POINT2D_V1_HH__
#define __POINT2D_V1_HH__

class Point2D
{
private:
  float _x;
  float _y;

public:
  void initialise(float, float);
  void affiche();
};

#endif
```

Listing 22: Point2D_V1.hh

Par défaut, si on ne spécifie rien, les membres d'une classe ont le statut privé. C'est le contraire pour une structure.

Dans une structure, on peut utiliser le mot-clé *private* pour rendre privés des attributs ou des fonctions.

Constructeurs

```
#ifndef __POINT2D_V2_HH__
#define __POINT2D_V2_HH__

class Point2D
{
private:
  float _x;
  float _y;

public:
  Point2D(float,float);
  void affiche();
};

#endif
```

Listing 23: Point2D_V2.hh

Au lieu d'utiliser une fonction *initialise*, il existe un autre mécanisme en C++: les **constructeurs**.
Les fonctions comme *initialise* ont, en effet, des inconvénients:

- On ne peut pas forcer l'utilisateur de la classe à l'appeler.
- Elle n'est pas appelée au moment de la création de l'objet ce qui peut être ennuyeux si des opérations doivent être effectuées dès le début de la vie de notre instance.

```
#include<iostream>
#include "Point2D_V2.hh"
using namespace::std;

Point2D::Point2D(float abs, float ord){
  _x = abs, _y = ord;
}
void Point2D::affiche(){
  cout << _x << " : " << _y << endl; }
```

Listing 24: Point2D_V2.cpp

Constructeurs

```
#ifndef __POINT2D_V2_HH__
#define __POINT2D_V2_HH__

class Point2D
{
private:
  float _x;
  float _y;

public:
  Point2D(float,float);
  void affiche();
};

#endif
```

Listing 23: Point2D_V2.hh

Un constructeur est une fonction membre ne renvoyant rien, qui porte le même nom que la classe.
Elle est appelée lors de la déclaration d'un objet.
Si nous voulions déclarer maintenant un objet de type *Point2D*, nous serions obligés de fournir les deux coordonnées.

Point2D p(3, 4); par exemple, déclare l'objet p de type *Point2D* et appelle dans la foulée le constructeur avec les paramètres 3 et 4.

```
#include<iostream>
#include "Point2D_V2.hh"
using namespace::std;

Point2D::Point2D(float abs, float ord){
  _x = abs, _y = ord;
}
void Point2D::affiche(){
  cout << _x << " : " << _y << endl; }
```

Listing 24: Point2D_V2.cpp

Constructeurs

```
#ifndef __POINT2D_V2_HH__
#define __POINT2D_V2_HH__

class Point2D
{
private:
  float _x;
  float _y;

public:
  Point2D(float,float);
  void affiche();
};

#endif
```

Listing 23: Point2D_V2.hh

Il peut y avoir autant de constructeurs que l'on veut, en fonction des besoins.

Il faudra les distinguer les uns des autres par des paramètres distincts.
On peut également, si cela est nécessaire, placer un constructeur dans la partie *private* de notre classe. Ainsi, son utilisation sera bloquée à l'extérieur de la classe.

```
#include<iostream>
#include "Point2D_V2.hh"
using namespace::std;

Point2D::Point2D(float abs, float ord){
  _x = abs, _y = ord;
}
void Point2D::affiche(){
  cout << _x << " : " << _y << endl; }
```

Listing 24: Point2D_V2.cpp

Destructeur

```
#ifndef __POINT2D_V3_HH__
#define __POINT2D_V3_HH__

class Point2D
{
private:
  float _x, _y;

public:
  Point2D(float,float);
  ~Point2D();
};

#endif
```

Listing 25: Point2D_V3.hh

La classe est déclarée dans un fichier header .hh

On voit les deux données membres déclarées *private*, ainsi qu'un constructeur de la classe ayant deux paramètres de type float.

Le destructeur de la classe porte le même nom que celle-ci précédé du caractère ~ (tilde).

Il est appelé **lors de la destruction de l'instance courante**. Son objectif est de permettre au programmeur de libérer de l'espace mémoire si nécessaire.
De même si des fichiers ont été ouverts ou des connexions (vers des bases de données par exemple) ont été initiées durant la vie de l'instance.