Programmation Orientée Objet — Episode #1

Le contexte

C vs C++98

Caractéristiques

- Typage plus fort
- Impératif **et** Orienté Objet
- Réutilisation de code
- Performance privilégiée

De nombreuses fonctionnalités ajoutées par C++... dont certaines reprises par le C.

Compilation — quasi-identique

`g++` — `gcc` en mode C++
std=c++98 (ou gnu++98 pour certaines extensions)

Fichiers en-tête
Convention : `.hpp`

Utilisation du C

```c
int main() { // () signifie (void) en C++
  puts("Hello world!");
  return 0;
}
```

À la compilation

```
error: 'puts' was not declared in this scope
```

Utilisation du C

```c
int main() { // () signifie (void) en C++
  puts("Hello world!");
  return 0;
}
```

```c
extern int puts( char const * );
```

À la compilation

```
undefined reference to 'puts( char const * )'
```

Utilisation du C

```
1  int main() { // () signifie (void) en C++
2    puts("Hello world!");
3    return 0;
4  }
```

```
1  extern "C" {
2    int puts( char const * );
3  }
```

En-tête standard
`#include<stdio.h>`

Le premier programme

```cpp
#include <iostream>

int main() { // () signifie (void) en C++

  std::cout << "Hello world!";

  return 0;

}
```

La sortie

```
Hello world!$
```

Le premier programme

```cpp
#include <iostream>

int main() { // () signifie (void) en C++

  std::cout << "Hello world!" << std::endl;

  return 0;

}
```

std ?

Les nouveautés — Les changements

Espace de noms

Espace de noms

Objectif

Limiter la collision des noms (et le besoin de noms différents)

Déclaration dans un `namespace`

```
1  namespace un_espace {
2    int trois() {
3      return 3;
4    }
5  }
```

Espace de noms

Objectif

Limiter la collision des noms (et le besoin de noms différents)

Un `namespace` peut être rouvert et complété.

Déclaration dans un `namespace`

```
1  namespace un_espace {
2    int quatre();
3  }
```

Espace de noms

Objectif

Limiter la collision des noms (et le besoin de noms différents)

Un `namespace` peut être rouvert et complété.

Déclaration dans un `namespace`

```
1  namespace un_espace {
2    int quatre();
3  }
```

La fonction peut être définie à l'extérieur (*e.g.* dans un `.cpp`).

```
1  int un_espace::quatre() {
2    return 4;
3  }
```

Espace de noms — Utilisation

Utilisation indiquant le `namespace`

```
1  int n = un_espace::trois();
```

Utilisation en ajoutant aux défauts

```
1  using namespace un_espace; // tout le namespace
2  using un_espace::trois(); // seulement un_espace::trois() et ses
       surcharges
3  int m = trois();
```

n'importe où, actif localement (et fermé avec le bloc).

Espace de noms — Les principaux

- Le **namespace global** : celui de `main`, des fonctions `C` et de tout ce qui n'est pas déclaré dans un `namespace`.
- Le **namespace std** : celui de toutes les bibliothèques standard :
  `using namespace std;` (en général en début de fichier (`.cpp`) après les `include`.

Espace de noms anonyme

Joue le rôle du `static` de `C` (restreint la visibilité au fichier).

```
1  namespace un_espace {
2    namespace anonyme {
3      int vingt() {
4        return 20;
5      }
6    }
7    int cent() {
8      return 5 * vingt();
9    }
10 }
```

Non ouvrable par `using` et actif uniquement dans le fichier.

Espace de noms — Exemples

```
1  #include <iostream>
2
3  namespace un_espace_de_noms {
4    int maximum(int a, int b) {
5      return a < b ? b : a;
6    }
7  }
8
9  int main() {
10
11   std::cout << maximum(5,2) << std::endl;
12
13   return 0;
14
15 }
```

Espace de noms — Exemples

```
1  #include <iostream>
2
3  namespace un_espace_de_noms {
4    int maximum(int a, int b) {
5      return a < b ? b : a;
6    }
7  }
8
9  int main() {
10
11   std::cout << un_espace_de_noms::maximum(5,2) << std::endl;
12
13   return 0;
14
15 }
```

Espace de noms — Exemples

```
1  #include <iostream>
2
3  namespace un_espace_de_noms {
4    int maximum(int a, int b) {
5      return a < b ? b : a;
6    }
7  }
8
9  int main() {
10
11   using namespace un_espace_de_noms;
12   std::cout << maximum(5,2) << std::endl;
13
14   return 0;
15
16 }
```

Les entrées/sorties — TBC

Les nouveautés — 1/?

Entrées/sorties

```
1  #include <iostream>
2
3  int main() {
4
5    using namespace std;
6
7    cout << "Saisir une valeur : ";
8    int a;
9    cin >> a;
10   cout << "Valeur saisie : " << a << endl;
11   cerr << "Message sur la sortie d'erreur standard." << endl;
12
13   return 0;
14
15 }
```

Utilisation de << et >> : décalages binaires ?

Allocation dynamique

Les nouveautés — 2/?

Allocation dynamique – -TBC

```
1  int *i = new int;
2  int *tableau = new int[10];
```

Désallocation dynamique — TBC

```
1  delete i;
2  delete[] tableau;
```

Les références — TBC

Les nouveautés — 3/?

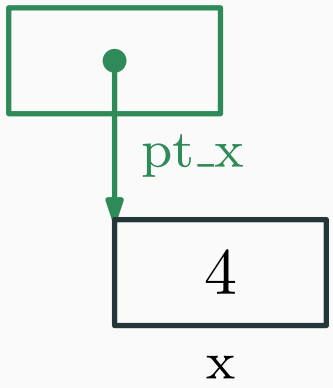
Les références

```
1  #include <iostream>
2
3  int main() {
4
5    using namespace std;
6    int x = 3;
7    int *pt_x = &x;
8    *pt_x = 4;
9
10   cout << x << endl;
11
12   return 0;
13
14 }
```

Les nouveautés — 3/?

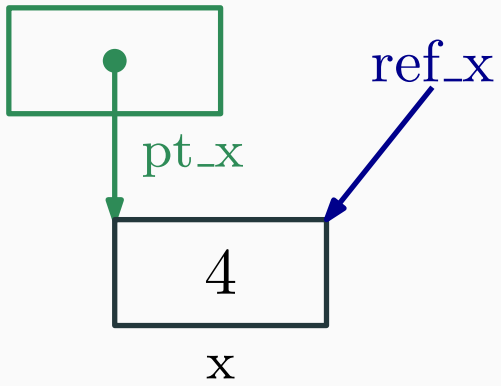
Les références

```
1  #include <iostream>
2
3  int main() {
4
5    using namespace std;
6    int x = 3;
7    int *pt_x = &x;
8    *pt_x = 4;
9
10   cout << x << endl;
11
12   return 0;
13
14 }
```



Les références

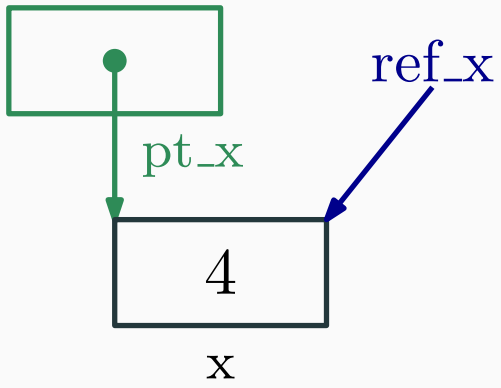
```
1  #include <iostream>
2
3  int main() {
4
5    using namespace std;
6    int x = 3;
7    int &ref_x = x;
8    ref_x = 5;
9
10   cout << x << endl; // ?
11
12   return 0;
13
14 }
```



Les nouveautés — 3/?

Les références

```
1  #include <iostream>
2
3  int main() {
4
5    using namespace std;
6    int x = 3;
7    int &ref_x = x;
8    ref_x = 5;
9
10   cout << x << endl; // ?
11
12   return 0;
13
14 }
```



Nouveau concept

Gain en lisibilité, écriture et sûreté !

Les nouveautés — 3/?

`pt_x++` ?

`pt_x` pointe sur une autre adresse

`ref_x++` ?

`x` est augmenté de 1

Les nouveautés — 3/?

```
1  #include <iostream>
2
3  int main() {
4
5    using namespace std;
6    int x = 3;
7    int &ref_x = x;
8    int *pt_x = &x;
9
10   cout << pt_x++ << endl; // une nouvelle adresse
11   cout << ++ref_x << endl; // 4
12
13   return 0;
14
15 }
```

Les nouveautés — 3/?

Que vaut `ref_x` ?

```
1  #include <iostream>
2
3  int main() {
4
5    int x = 3;
6    int &ref_x = x;
7
8    std::cout << &ref_x << std::endl; // adresse de x
9    return 0;
10
11 }
```

Références et passage par variable

Les nouveautés — 4/?

Passage par variable

```
1  #include <iostream>
2
3  void changer( int &x ) {
4    x = 2;
5  }
6
7  int main() {
8
9    using namespace std;
10
11   int x = 3;
12   changer(x);
13   cout << x << endl; // ?
14
15   return 0;
16
17 }
```

Les nouveautés — 4/?

Différence avec le C

La **signature** de la fonction détermine si passage par valeur **ou** par variable.

Surcharge de fonctions

Les nouveautés — 5/?

Permet d'avoir deux fonctions du même nom.

Différenciées par le type des arguments.

```
1  #include <iostream>
2
3  int maximum(int a, int b) {
4    return a < b ? b : a;
5  }
6  double maximum(double a, double b) {
7    return a < b ? b : a;
8  }
9
10 int main() {
11
12   std::cout << maximum(2,3) << " " << maximum(2.0, 3.0) << std::
         endl;
13   return 0;
14
15 }
```

Les nouveautés — 5/?

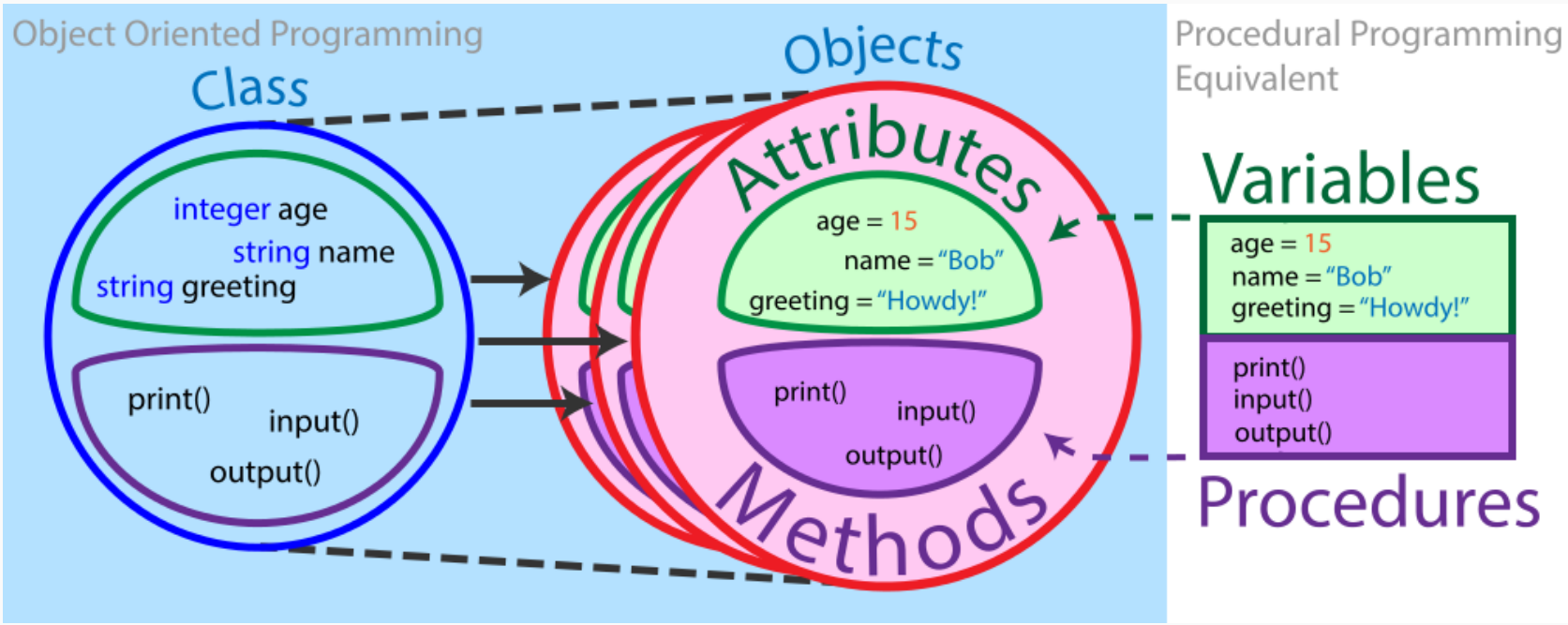
Permet d'avoir deux fonctions du même nom.

Différenciées par le type des arguments.

```
1  #include <iostream>
2
3  int maximum(int a, int b) {
4    return a < b ? b : a;
5  }
6  double maximum(double a, double b) {
7    return a < b ? b : a;
8  }
9
10 int main() {
11   using namespace std;
12   cout << maximum(2,3) << " " << maximum(2.0, 3.0) << endl;
13   cout << maximum(2, 3.0) << endl; // ?
14   return 0;
15
16 }
```

Les classes — Introduction

Schématiquement



Les classes — Un exemple basique

Point.hpp

```
1  #ifndef POINT_HPP
2  #define POINT_HPP
3  #include <iostream>
4
5  class Point {
6
7    private: // par défaut --- TBC
8      int x;
9      int y;
10   public:
11     Point(); // aucun type de retour
12     Point(int, int); // Surcharge du constructeur
13     void afficher();
14
15     int get_x() { return x; }
16     int get_y() { return y; }
17
18 };
19 #endif
```

Point.cpp

```cpp
#include "Point.hpp"

using namespace std;

Point::Point() {
  cout << "Constructeur par défaut." << endl;
  (*this).x = 0;
  (*this).y = 0;
}

Point::Point(int x, int y) {
  (*this).x = x;
  (*this).y = y;
}

void Point::afficher() {
  cout << "[" << (*this).x << "," << (*this).y << "]" << endl;
}
```

main_point.cpp

```cpp
#include "Point.hpp"

int main() {

  Point *P1 = new Point();
  Point P2(2,3);

  (*P1).afficher(); // la méthode afficher est appelée sur l'
    objet P1
  P2.afficher(); // la méthode afficher est appelée sur l'objet
    P2

  return 0;

}
```

Et la mémoire ?

```
1  #include "Point.hpp"
2
3  int main() {
4
5    Point *P1 = new Point();
6    Point P2(2,3);
7
8    (*P1).afficher(); // la méthode afficher est appelée sur l'
       objet P1
9    P2.afficher(); // la méthode afficher est appelée sur l'objet
       P2
10
11   // et au niveau de la memoire ?
12   delete P1; // OK
13   delete P2; // error: cannot delete expression of type 'Point'
14
15   return 0;
16
17 }
```

Que fait `delete` ?

- Classique sur les types standards
- Mais sur les classes ?

Le destructeur
Doit être déclaré avec la syntaxe `~Classe()`

Comportement par défaut non défini.

Mémoire et constructeurs — TBC

```
1  #ifndef POINT_CONST_HPP
2  #define POINT_CONST_HPP
3  #include <iostream>
4
5  class Point_const {
6
7    private: // par défaut --- TBC
8      int x;
9      int y;
10   public:
11     Point_const(); // aucun type de retour
12     Point_const(const Point_const &p);
13     Point_const(int, int); // Surcharge du constructeur
14
15     // Doivent être const... Suffisant ?
16     int get_x() const { return x; }
17     int get_y() const { return y; }
18
19 };
20 #endif
```

Mémoire et constructeurs — TBC

```cpp
#include "Point_const.hpp"

using namespace std;

Point_const::Point_const() {
  cout << "Constructeur par défaut." << endl;
  (*this).x = 0;
  (*this).y = 0;
}

Point_const::Point_const(const Point_const &p) {
  cout << "Constructeur par recopie." << endl;
  (*this).x = p.get_x();
  (*this).y = p.get_y();
}

Point_const::Point_const(int x, int y) {
  (*this).x = x;
  (*this).y = y;
}
```

Mémoire et constructeurs — TBC

```
1  #include <iostream>
2  #include "Point_const.hpp"
3
4  static Point_const comparer(Point_const p, Point_const p1) {
5    return p.get_x() < p1.get_x() ? p : p1;
6  }
7  int main() {
8    Point_const p;
9    Point_const p1;
10   comparer(p, p1);
11
12   return 0;
13 }
```