ECE PARIS
ÉCOLE D'INGÉNIEURS



Robin FERCOQ


# Conception et Programmation Orientée Objet C++

# POO - C++

## Sommaire général du semestre

| **COURS** | Semaine suivante → | **TPs** |
|---|---|---|
| 1. *Intro, concepts, 1 exemple* | | 1. *Organisation objet des données* |
| 2. *Modélisation objet / UML* | | 2. *Diagrammes de classe UML* |
| 3. *C++ pratique 1* | | 3. *C++ pratique, E/S, string, vector* |
| **4. *C++ pratique 2*** | | 4. *C++ pratique, type &, surcharge* |
| 5. *Classes & C++ : bases* | | 5. *Date : une classe simple en C++* |
| 6. *Classes & C++ : compléments* | | 6. *UML et C++, associations* |
| 7. *Conteneurs & C++ : la STL* | | 7. *Gestion de collections complexes* |
| 8. *Héritage / polymorphisme* | | 8. *Collections polymorphes* |
| 9. *Modèles objets avancés* | | 9. *Modèle composite et graphismes* |
| 10. *Exceptions, flots, fichiers ...* | | 10. *Persistance / fichiers / except.* |
| 11. *Templates côté développeur* | | 11. *Développement de templates* |
| 12. *Gestion mémoire / smarts ptrs* | | 12. *Soutenance de **projet** ...* |

# C++ *pratique 2*

# COURS 4

**A) Structs simples**
**B) Pointeurs * et Références &**
**C) La qualification const**
**D) Valeurs paramètres par défaut**
**E) Surcharge de fonctions**
**F) Surcharge d’opérateurs**
**G) Allocation dynamique new/delete**

# COURS 4

A) Structs simples
B) Pointeurs * et Références &
C) La qualification const
D) Valeurs paramètres par défaut
E) Surcharge de fonctions
F) Surcharge d'opérateurs
G) Allocation dynamique new/delete

# Structs simples

# Structs simples

- *En C++ comme en C on retrouve la notion de* **`struct`**
- *En C++ il n'est pas utile d'utiliser* `typedef`

**coords.h** — **C++**

```
#ifndef COORDS_H_INCLUDED
#define COORDS_H_INCLUDED

struct Coords
{
    double x, y;
};

#endif // COORDS_H_INCLUDED
```

**coords.h** — **C**

```
#ifndef COORDS_H_INCLUDED
#define COORDS_H_INCLUDED

typedef struct coords
{
    double x, y;
}
t_coords;

#endif // COORDS_H_INCLUDED
```

# Structs simples

- *En C++ comme en C on retrouve la notion de* **`struct`**
- *En C++ il n'est pas utile d'utiliser* `typedef`

**coords.h** **C++**

```
struct Coords
{
    double x, y;
};
```

**coords.h** **C**

```
typedef struct coords
{
    double x, y;
}
t_coords;
```

# Structs simples

- *En C++ comme en C on retrouve la notion de* **`struct`**
- *En C++ il n'est pas utile d'utiliser* `typedef`

**coords.h** — **C++**

**Majuscule**
**Convention universelle**

```
struct Coords
{
    double x, y;
};
```

**Attention au ; à la fin !**
**L'oublier conduit à des erreurs méchantes dans le(s) fichier(s) où le .h est inclus**

**coords.h** — **C**

```
typedef struct coords
{
    double x, y;
}
t_coords;
```

# Structs simples

- *En C++ comme en C la struct est **gérée par valeur***
- *En C++ on peut affecter une valeur littérale sans caster*

**main.cpp** — **C++**

```
a : 3.1 2.5
a : 4.5 6.6
b : 3.1 2.5
```

```cpp
#include "coords.h"

int main()
{
    Coords a = {3.1, 2.5};
    std::cout << "a : " << a.x << " " << a.y << std::endl;
    Coords b = a;
    a = {4.5, 6.6};
    std::cout << "a : " << a.x << " " << a.y << std::endl;
    std::cout << "b : " << b.x << " " << b.y << std::endl;
```

**main.c** — **C**

```c
#include "coords.h"

int main()
{
    t_coords a = {3.1, 2.5};
    printf("a : %.1f %.1f\n", a.x, a.y);
    t_coords b = a;
    a = (t_coords){4.5, 6.6};
    printf("a : %.1f %.1f\n", a.x, a.y);
    printf("b : %.1f %.1f\n", b.x, b.y);
```

# Structs simples

- *En C++ comme en C la struct est **gérée par valeur***
- *En C++ on peut affecter une valeur littérale sans caster*

**main.cpp**

```
#include "coords.h"

int main()
{
    Coords a = {3.1, 2.5};

    Coords b = a;

    a = {4.5, 6.6};
```

```
a : 3.1 2.5
a : 4.5 6.6
b : 3.1 2.5
```

**C++**

**main.c**

```
#include "coords.h"

int main()
{
    t_coords a = {3.1, 2.5};

    t_coords b = a;

    a = (t_coords){4.5, 6.6};
```

**Caster**

**C**

# Structs simples

- *En C++ la struct se comporte « comme un scalaire »*
- *En C++ la struct est en fait une classe !*

**main.cpp**

```cpp
#include "coords.h"

int main()
{
    Coords a = {3.1, 2.5};

    Coords b = a;

    a = {4.5, 6.6};
```

```
a : 3.1 2.5
a : 4.5 6.6
b : 3.1 2.5
```

**C++**

- *La struct est une classe mais une classe avec des attributs publiques*
- *En général en orienté objet on se méfie des attributs publiques : ça rompt le principe d'encapsulation...*

# Structs simples

- *Envisageables pour grouper peu d'infos élémentaires*
- *Pas pour des types d'objets complexes*

**coords.h** **C++**

```
struct Coords
{
    double x, y;
};
```

- *Par exemple la bibliothèque standard propose la struct* `std::pair< >` *avec les attributs first et second*
- *Nous verrons lors du cours sur les classes les conséquences possibles du non respect du principe d'encapsulation*

# COURS 4

A) Structs simples
B) Pointeurs * et Références &
C) La qualification const
D) Valeurs paramètres par défaut
E) Surcharge de fonctions
F) Surcharge d’opérateurs
G) Allocation dynamique new/delete

# Pointeurs * et Références &

# Pointeurs * et Références &

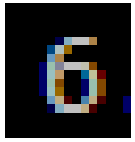
- *Les tableaux ont une sémantique par référence implicitement c'est l'adresse des données qui est transmise aux sous-programmes, données pas copiées*
- *Les types scalaires et les classes (`class` ou `struct`) ont une sémantique par valeur, par défaut c'est une copie des données qui est transmise aux sous-progs.*
- *Ceci a une conséquence sur les performances : copier des données coûte plus cher qu'une adresse*
- *Une autre conséquence est que si un sous-programme doit **modifier** des données de l'appelant il faut soit :*
  - *Retourner la nouvelle valeur*
  - *Utiliser explicitement un passage par adresse*

# Pointeurs * et Références &

- *Modification donnée appelant **par valeur retour***
- *1ère copie de donnée à l'appel, 2ème copie au retour*
- *Performance ok pour quelques octets*

C++

```
int doubler(int x)
{
    return 2*x;
}

int main()
{
    int val = 3;

    val = doubler( val );

    std::cout << val << std::endl;
```



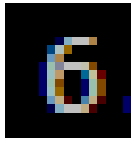
# Pointeurs * et Références &

- *Modification donnée appelant **par valeur retour***
- *1ère copie de donnée à l'appel, 2ème copie au retour*
- *Performance ok pour quelques octets*

C++

```
int doubler(int x)
{
    return 2*x;
}

int main()
{
    int val = 3;

    val = doubler( val );

    std::cout << val << std::endl;
```

6 3

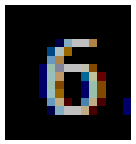


# Pointeurs * et Références &

- *Modification donnée appelant **par adresse***
- *Copie de l'adresse des données à l'appel*
- *L'appelé accède directement aux données appelant*

C++

```
void doubler(int *px)
{
    *px = 2 * *px;
}

int main()
{
    int val = 3;

    doubler( &val );

    std::cout << val << std::endl;
```



# Pointeurs * et Références &

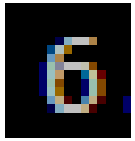
- *Modification donnée appelant **par adresse***
- *Copie de l'adresse des données à l'appel*
- *L'appelé accède directement aux données appelant*

**C++**

```
void doubler(int *px)
{
    *px = 2 * *px;
}

int main()
{
    int val = 3;

    doubler( &val );

    std::cout << val << std::endl;
```

modifier val

adresse de val



# Pointeurs * et Références &

- *Modification donnée appelant **par adresse***
- *Copie de l'adresse des données à l'appel*
- *L'appelé accède directement aux données appelant*

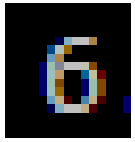
C++

```
void doubler(int *px)
{
    *px = 2 * *px;
}

int main()
{
    int val = 3;
    doubler( &val );
    std::cout << val << std::endl;
```

**Déclarer un passage par adresse**

**Déréférencement : « valeur à cette adresse »**

**Indirection : « adresse de cette variable »**

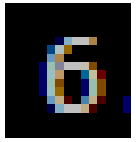


# Pointeurs * et Références &

- *Modification donnée appelant **par référence***
- *Copie de l'adresse des données à l'appel*
- *L'appelé accède directement aux données appelant*

**C++**

```
void doubler(int& x)
{
    x = 2 * x;
}

int main()
{
    int val = 3;

    doubler( val );

    std::cout << val << std::endl;
```



# Pointeurs * et Références &

- *Modification donnée appelant **par référence***
- *Copie de l'adresse des données à l'appel*
- *L'appelé accède directement aux données appelant*

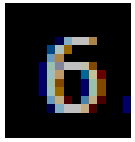
C++

```
void doubler(int& x)
{
    x = 2 * x;
}

int main()
{
    int val = 3;

    doubler( val );

    std::cout << val << std::endl;
```

**Déclarer un passage par référence**

**Lors de cet appel x est un alias pour val**

**Rien à préciser au niveau de l'appelant**



# Pointeurs * et Références &

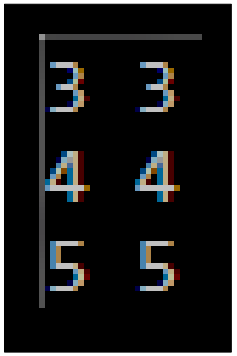
- *En déclarant (ou pas) un paramètre **par référence***
  - *Il suffit de mettre (ou pas) & après le type*
  - *Le code appelant reste le même (pas de &var)*
  - *Le code appelé reste le même (pas de *param)*
- *On peut aussi déclarer une référence à une variable, c'est moins utile que pour les paramètres...*
- *Techniquement une référence type& est gérée par l'exécutable comme un pointeur type* mais la référence a des règles différentes :*
  - *Elle doit être initialisée a sa déclaration*
  - *Elle ne pourra pas référencer une autre donnée*
  - *Elle ne peut pas référencer « rien » (pas de NULL)*

# Pointeurs * et Références &

- *Une référence est donc une liaison irrévocable avec les données référencées jusqu'à ce que la ~~mort~~ fin du scope les sépare*

C++

```
int  a = 3;
int& b = a;

std::cout << a << " " << b << std::endl;

int c = 4;
b = c;
std::cout << a << " " << b << std::endl;

a = 5;
std::cout << a << " " << b << std::endl;
```



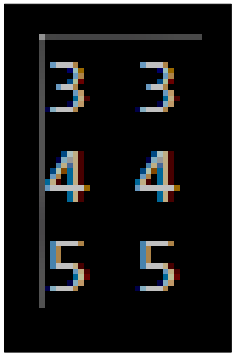
# Pointeurs * et Références &

- *Une référence est donc une liaison irrévocable avec les données référencées jusqu'à ce que la ~~mort~~ fin du scope les sépare*

C++

```
int  a = 3;
int& b = a;
```
**La référence b est définitivement un alias de a**

```
std::cout << a << " " << b << std::endl;

int c = 4;
b = c;
std::cout << a << " " << b << std::endl;

a = 5;
std::cout << a << " " << b << std::endl;
```
**Ceci est équivalent à a = c;**



# Pointeurs * et Références &

- *Une référence est donc une liaison irrévocable avec les données référencées jusqu'à ce que la ~~mort~~ fin du scope les sépare*

C++

```
char mat[6][7];
... remplissage de la matrice ...

for (int lig=5; lig>0; --lig)
    for (int col=0; col<7; ++col)
    {
        char& caseIci  = mat[lig][col];
        char& caseHaut = mat[lig-1][col];

        if ( caseIci==' ' && caseHaut!=' ' )
        {
            caseIci = caseHaut;
            caseHaut = ' ';
        }
    }
```

*Ce code fait « tomber » les caractères dans la matrice (code pour jeu de Puissance 4)*

# Pointeurs * et Références &

- *Equivalent au code précédent sans références : le code est plus court mais moins explicite*

C++

```
char mat[6][7];
... remplissage de la matrice ...

for (int lig=5; lig>0; --lig)
    for (int col=0; col<7; ++col)
    {
        if ( mat[lig][col]==' ' && mat[lig-1][col]!=' ' )
        {
            mat[lig][col] = mat[lig-1][col];
            mat[lig-1][col] = ' ';
        }
    }
```

*Ce code fait « tomber » les caractères dans la matrice (code pour jeu de Puissance 4)*

# Pointeurs * et Références &

- *Comme pour les pointeurs avec * la position du & pour déclarer les références n'a pas d'importance*
- *En terme de « style » de codage*
  - *on peut coller le & à la variable ou paramètre*
    **`float &x`**
  - *ou coller le & au type :* ***style majoritaire dans les docs***
    **`float& x`**

⇨ Google C++ Style Guide :Pointer and Reference Expressions

When declaring a pointer variable or argument, you may place the asterisk adjacent to either the type or to the variable name:

```
// These are fine, space preceding.
char *c;
const string &str;

// These are fine, space following.
char* c;
const string& str;
```

You should do this consistently within a single file, so, when modifying an existing file, use the style in that file.

# Pointeurs * et Références &

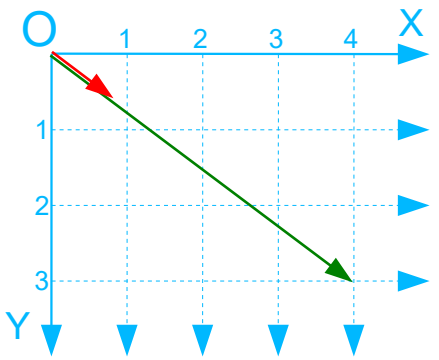
- *C'est dans la communication des objets aux sous-programmes que les références auront le plus d'utilité*
- *Sans référence, modifier un objet **par valeur retour** ( ici c'est acceptable : l'objet est léger )*

**C++**

```
Coords normaliser(Coords vecteur)
{
    double norme = sqrt( pow(vecteur.x, 2) + pow(vecteur.y, 2) );
    Coords resultat = {vecteur.x/norme, vecteur.y/norme};
    return resultat;
}

int main()
{
    Coords v{4.0, 3.0};
    std::cout << "v : " << v.x << " " << v.y << std::endl;

    v = normaliser(v);
    std::cout << "v : " << v.x << " " << v.y << std::endl;
```

**Attention risque division par 0**

```
v : 4.0 3.0
v : 0.8 0.6
```



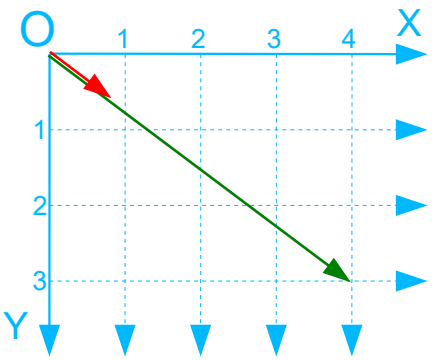
# Pointeurs * et Références &

- *C'est dans la communication des objets aux sous-programmes que les références auront le plus d'utilité*
- *Sans référence, modifier un objet **par valeur retour** Version courte avec objet-valeur retourné directement*

**C++**

```
Coords normaliser(Coords vecteur)
{
    double norme = sqrt( pow(vecteur.x, 2) + pow(vecteur.y, 2) );
    return norme ? {vecteur.x/norme, vecteur.y/norme} : vecteur;
}


int main()
{
    Coords v{4.0, 3.0};
    std::cout << "v : " << v.x << " " << v.y << std::endl;

    v = normaliser(v);
    std::cout << "v : " << v.x << " " << v.y << std::endl;
```

```
v : 4.0 3.0
v : 0.8 0.6
```



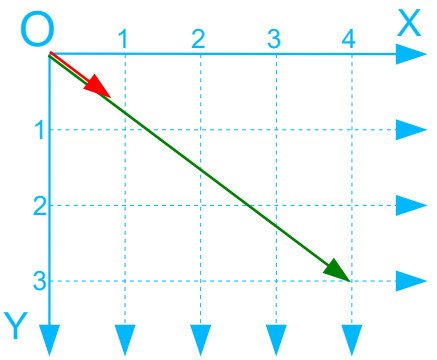
# Pointeurs * et Références &

- *C'est dans la communication des objets aux sous-programmes que les références auront le plus d'utilité*
- *Sans référence, modifier un objet **par adresse***
  *Noter **&** dans l'appel et **->** à la place de **.** dans l'appelé*

**C++**

```
void normaliser(Coords* vecteur)
{
    double norme = sqrt( pow(vecteur->x, 2) + pow(vecteur->y, 2) );
    if ( norme!=0 )
    {  vecteur->x /= norme;  vecteur->y /= norme;  }
}

int main()
{
    Coords v{4.0, 3.0};
    std::cout << "v : " << v.x << " " << v.y << std::endl;

    normaliser(&v);
    std::cout << "v : " << v.x << " " << v.y << std::endl;
```

```
v : 4.0 3.0
v : 0.8 0.6
```



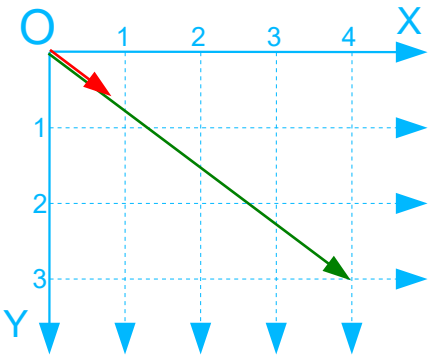
# Pointeurs * et Références &

- *C'est dans la communication des objets aux sous-programmes que les références auront le plus d'utilité*
- *Sans référence, modifier un objet **par adresse***
  *Noter **&** dans l'appel et **->** à la place de **.** dans l'appelé*

**C++**

```
void normaliser(Coords* vecteur)
{
    double norme = sqrt( pow(vecteur->x, 2) + pow(vecteur->y, 2) );
    if ( norme!=0 )
    {  vecteur->x /= norme;  vecteur->y /= norme;  }
}

int main()
{
    Coords v{4.0, 3.0};
    std::cout << "v : " << v.x << " " << v.y << std::endl;

    normaliser(&v);
    std::cout << "v : " << v.x << " " << v.y << std::endl;
```

```
v : 4.0 3.0
v : 0.8 0.6
```



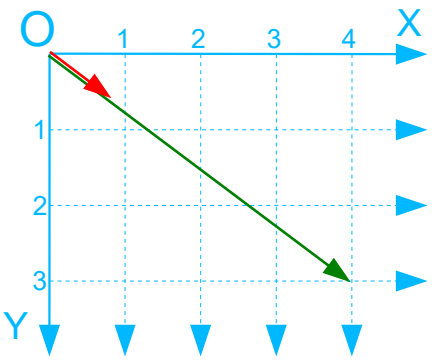
# Pointeurs * et Références &

- *C'est dans la communication des objets aux sous-programmes que les références auront le plus d'utilité*
- *Avec référence, modifier un objet **par référence***
  *Noter pas de **&** dans l'appel et **.** dans l'appelé*

**C++**

```
void normaliser(Coords& vecteur)
{
    double norme = sqrt( pow(vecteur.x, 2) + pow(vecteur.y, 2) );
    if ( norme!=0 )
    {  vecteur.x /= norme;  vecteur.y /= norme;  }
}

int main()
{
    Coords v{4.0, 3.0};
    std::cout << "v : " << v.x << " " << v.y << std::endl;

    normaliser(v);
    std::cout << "v : " << v.x << " " << v.y << std::endl;
```

```
v : 4.0 3.0
v : 0.8 0.6
```



# Pointeurs * et Références &

- *C'est dans la communication des objets aux sous-programmes que les références auront le plus d'utilité*
- *Sauf raison particulière de vouloir demander un pointeur à l'appelant,* ***on préfère donc le passage par référence au passage par adresse...***
- *En revanche stocker des références (vecteurs...) sera difficile: on utilisera des pointeurs pour les collections...*
- *Attention, le terme « référence » est utilisé dans des situations différentes et peut vouloir dire 2 choses*
  - *La référence* **&** *technique spécifique du C++*
  - *La référence en conception orientée objet (sur des diagrammes d'objets...) qui pourra se traduire en C++ parfois par* **&** *parfois par* *****

# Pointeurs * et Références &

***Attention à ne pas tout mélanger***

| contexte \ symbole | * | & |
|---|---|---|
| déclaration | pointeur sur « type à gauche » | référence à « type à gauche » |
| utilisation | déréférencement : valeur pointée par | indirection : adresse de |

# Pointeurs * et Références &

- *Le passage par référence n'est pas utilisé que pour permettre à un sous-programme de modifier des données de l'appelant, on l'utilise aussi pour* ***éviter des copies inutiles*** ***(pour plus de quelques octets)***

C++

```
void afficherCoords(Coords& c)
{
    std::cout << c.x << " " << c.y << std::endl;
}

void afficherListeCoords(std::vector<Coords>& lst)
{
    for (size_t i=0; i<lst.size(); ++i)
        afficherCoords(lst[i]);
}

int main()
{
    std::vector<Coords> quad = { {1,1}, {3,0}, {4,2}, {2,3} };

    afficherListeCoords(quad);
```

```
1.0 1.0
3.0 0.0
4.0 2.0
2.0 3.0
```

# Pointeurs * et Références &

- *Le passage par référence n'est pas utilisé que pour permettre à un sous-programme de modifier des données de l'appelant, on l'utilise aussi pour* ***éviter des copies inutiles*** ***(pour plus de quelques octets)***

C++

```
void afficherCoords(Coords& c) 1 Coords pèse 16 octet, la copie serait acceptable
{
    std::cout << c.x << " " << c.y << std::endl;
}
                    Une collection de Coords peut peser lourd → référence
void afficherListeCoords(std::vector<Coords>& lst)
{
    for (size_t i=0; i<lst.size(); ++i)
        afficherCoords(lst[i]);
}

int main()
{
    std::vector<Coords> quad = { {1,1}, {3,0}, {4,2}, {2,3} };

    afficherListeCoords(quad);
```

```
1.0 1.0
3.0 0.0
4.0 2.0
2.0 3.0
```

# COURS 4

**A) Structs simples**
**B) Pointeurs * et Références &**
**C) La qualification const**
**D) Valeurs paramètres par défaut**
**E) Surcharge de fonctions**
**F) Surcharge d'opérateurs**
**G) Allocation dynamique new/delete**

# La qualification const

# La qualification const

- *Le passage par référence n'est pas utilisé que pour permettre à un sous-programme de modifier des données de l'appelant, on l'utilise aussi pour* ***éviter des copies inutiles*** *(pour plus de quelques octets)*

Référence

C++

```
void afficherCoords(Coords& c)
{
    std::cout << c.x << " " << c.y << std::endl;
}

void afficherListeCoords(std::vector<Coords>& lst)
{
    for (size_t i=0; i<lst.size(); ++i)
        afficherCoords(lst[i]);
}

int main()
{
    std::vector<Coords> quad = { {1,1}, {3,0}, {4,2}, {2,3} };

    afficherListeCoords(quad);
```

Référence

```
1.0 1.0
3.0 0.0
4.0 2.0
2.0 3.0
```

# La qualification const

- *Ça implique qu'on confie des « données originales » à des sous-programmes destinés à ne pas les modifier Ils **pourraient** les modifier mais ne le **feront pas**...*

C++

```
void afficherCoords(Coords& c)
{
    std::cout << c.x << " " << c.y << std::endl;
}

int main()
{
    std::vector<Coords> quad = { {1,1}, {3,0}, {4,2}, {2,3} };

    afficherListeCoords(quad);
```

Affichage

Lecture seule faite

Lecture possible
Ecriture possible

```
1.0 1.0
3.0 0.0
4.0 2.0
2.0 3.0
```

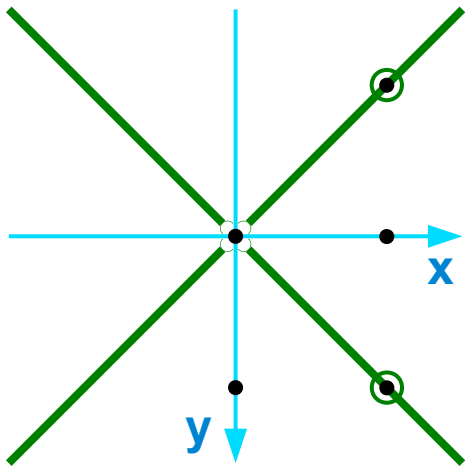
# La qualification const

- *Ça implique qu'on confie des « données originales » à des sous-programmes destinés à ne pas les modifier*
  *Ils **pourraient** les modifier mais ne le **feront pas**...*
  *Exemple : on affiche une info spécifique pour les coords qui sont **sur les diagonales (origine exclue)***

C++

```
void afficherCoords(Coords& c)
{
    std::cout << c.x << " " << c.y;
    if ( c.x!=0 && (c.x=-c.y || c.x==c.y) )
        std::cout << " Diagonale";
    std::cout << std::endl;
}
```

```
0 0
0 1
1 1 Diagonale
1 0
1 -1 Diagonale
```




```
int main()
{
    std::vector<Coords> quad = { {0,0}, {0,1}, {1,1}, {1, 0}, {1, -1} };
    afficherListeCoords(quad);
```

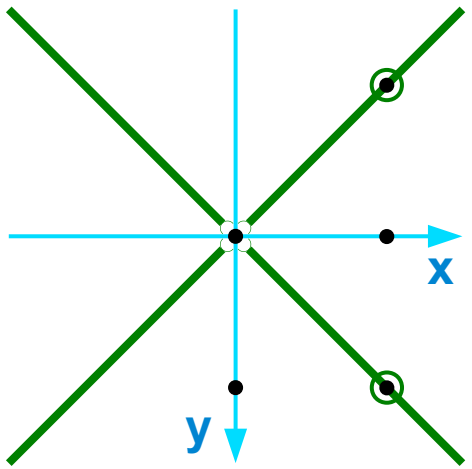
# La qualification const

- *Ça implique qu'on confie des « données originales » à des sous-programmes destinés à ne pas les modifier Ils **pourraient** les modifier mais ne le **feront pas**...*
- ***Sauf par accident !***

C++

```
void afficherCoords(Coords& c)
{
    std::cout << c.x << " " << c.y;
    if ( c.x!=0 && (c.x=-c.y || c.x==c.y) )
        std::cout << " Diagonale";
    std::cout << std::endl;
}
```

```
0 0
0 1
1 1 Diagonale
1 0
1 -1 Diagonale
```




```
int main()
{
    std::vector<Coords> quad = { {0,0}, {0,1}, {1,1}, {1, 0}, {1, -1} };
    afficherListeCoords(quad);
```

# La qualification const

- *Ça implique qu'on confie des « données originales » à des sous-programmes destinés à ne pas les modifier Ils **pourraient** les modifier mais ne le **feront pas**...*
- ***Sauf par accident !***

C++

```
void afficherCoords(Coords& c)
{
    std::cout << c.x << " " << c.y;
    if ( c.x!=0 && (c.x=-c.y || c.x==c.y) )
        std::cout << " Diagonale";
    std::cout << std::endl;
}
```

1er appel

```
0 0
0 1
1 1 Diagonale
1 0
1 -1 Diagonale
```

2ème appel

```
0 0
0 1
1 1 Diagonale
0 0
1 -1 Diagonale
```

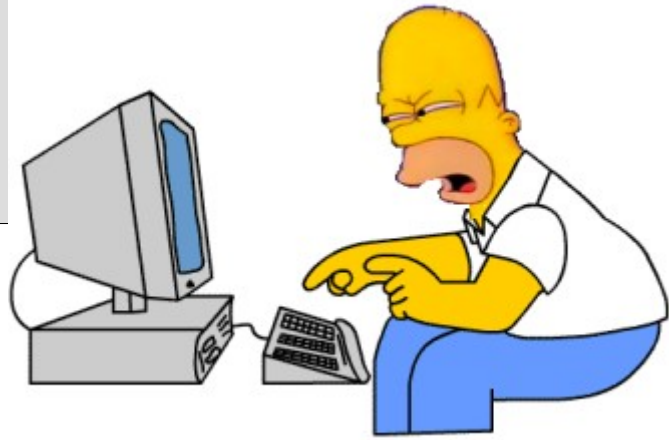
?

```
int main()
{
    std::vector<Coords> quad = { {0,0}, {0,1}, {1,1}, {1, 0}, {1, -1} };
    afficherListeCoords(quad);
    afficherListeCoords(quad);
```

⇐ *Beaucoup de temps et de code entre les 2 appels...*

# La qualification const

- ***Chercher l'erreur ( 4H de debug )***



```
void afficherCoords(Coords& c)
{
    std::cout << c.x << " " << c.y;
    if ( c.x!=0 && (c.x=-c.y || c.x==c.y) )
        std::cout << " Diagonale";
    std::cout << std::endl;
}
```

**1er appel**

```
0 0
0 1
1 1 Diagonale
1 0
1 -1 Diagonale
```

**2ème appel**

```
0 0
0 1
1 1 Diagonale
0 0
1 -1 Diagonale
```

?

```
int main()
{
    std::vector<Coords> quad = { {0,0}, {0,1}, {1,1}, {1, 0}, {1, -1} };
    afficherListeCoords(quad);
    afficherListeCoords(quad);
}
```

***Beaucoup de temps et de code entre les 2 appels...***

*0 error(s), 0 warning(s)*

# La qualification const

- *Le sous-programme afficherCoords n'a pas vocation à modifier les données* ***: mais il modifie les données***

C++

```
void afficherCoords(Coords& c)
{
    std::cout << c.x << " " << c.y;
    if ( c.x!=0 && (c.x=-c.y || c.x==c.y) )
        std::cout << " Diagonale";
    std::cout << std::endl;
}
```

**= au lieu de ==**

# La qualification const

*Quel est le problème ?*

*Pour ne pas tomber*
*il suffit de ne pas mettre*
*le pied au mauvais endroit !*

La programmation c'est
carrément moins dangereux !

Seuls les débilos font
= à la place de == !

*Vraiment ?*
*Qui prendra le risque ?*

# La qualification const

## Acceptabilité des risques pour les projets C/C++

### *Inacceptables*

- Santé
- Transport
- Commerce
- Finance / banque
- Industries lourdes

- Infrastructures S.I.
  - OS
  - Serveurs
    - → Fichiers
    - → BDD
    - → HTTPS
  - Compilateurs
  - Machines virtuelles
  - Bibliothèques

### *À éviter*

- Jeux / divertissement
- Bureautique
- Multimédia
- Création
- Navigateurs ?

### *Acceptables*

- Les exercices quand on apprend la programmation C++

```
if ( x = 3 )
    std::cout<<"OK\n";
else
    std::cout<<"?\n";
```

- Quoi d'autre ?

# La qualification const

- *Les problèmes de sécurité du passage par référence comme façon d'améliorer les performances en évitant les copies de données :*
  - *Augmente les risques de **propager** des corruptions de données en multipliant les lignes de codes qui ont accès en écriture aux données initiales de l'appelant*
  - *Rend confus le **rôle des paramètres** : est-ce qu'un paramètre est par référence pour pouvoir être modifié ou juste pour optimiser ?*
  - *Rend difficile le **débogage** en cas de problème, qui pourrait croire qu'une innocente fonction d'affichage peut corrompre ses données ?*

# La qualification const

- *La qualification* **`const`** *complète une déclaration pour indiquer que l'objet ou la donnée référencé ou pointé ne doit pas être modifié par l'appelé*
- ***Même pas par accident !***

C++

```
void afficherCoords(const Coords& c)
{
    std::cout << c.x << " " << c.y;
    if ( c.x!=0 && (c.x=-c.y || c.x==c.y) )
        std::cout << " Diagonale";
    std::cout << std::endl;
}
```

*error: assignment of member 'Coords::x'*
*in read-only object*

```
int main()
{
    std::vector<Coords> quad = { {0,0}, {0,1}, {1,1}, {1, 0}, {1, -1} };
    afficherListeCoords(quad);
    afficherListeCoords(quad);
```

# La qualification const

- *La qualification* **`const`** *complète une déclaration pour indiquer que l'objet ou la donnée référencé ou pointé ne doit pas être modifié par l'appelé*
- ***Même pas par accident !***

C++

```
void afficherCoords(const Coords& c)
{

                    c.x=-c.y


}
```

```
error: assignment of member 'Coords::x'
       in read-only object
```

# La qualification const

- *Le développeur du code appelé ne risque plus de **modifier accidentellement** une donnée entrante*
- *Le développeur du code appelant peut être sûr que c'est une **donnée entrante** et **non modifiable***

**C++**

```
void afficherCoords(const Coords& c)
{

}
```

```
0 error(s), 0 warning(s)
```

**Après correction rapide du code fautif par le dev. du code appelé...**

**Le dev. du code appelant peut en toute confiance considérer que les données ne seront pas modifiées, sans avoir à lire le code appelé !**

# La qualification const

- *Une donnée non-const peut être confiée à un paramètre const*
- *Une donnée const peut être confiée à un paramètre const*

C++

```
void afficherCoords(const Coords& c)
{
    std::cout << c.x << " " << c.y;
}

void afficherListeCoords(const std::vector<Coords>& lst)
{
    for (size_t i=0; i<lst.size(); ++i)
        afficherCoords(lst[i]);
}

int main()
{
    std::vector<Coords> quad = { {0,0}, {0,1}, {1,1}, {1, 0}, {1, -1} };
    afficherListeCoords(quad);
```

# La qualification const

- *Une donnée non-const peut être confiée à un paramètre const*
- *Une donnée const peut être confiée à un paramètre const*

C++

```
void afficherCoords(const Coords& c)
{
    std::cout << c.x << " " << c.y;
}

void afficherListeCoords(const std::vector<Coords>& lst)
{
    for (size_t i=0; i<lst.size(); ++i)
        afficherCoords(lst[i]);
}

int main()
{
    std::vector<Coords> quad = { {0,0}, {0,1}, {1,1}, {1, 0}, {1, -1} };
    afficherListeCoords(quad);
```

const lst[i]
*vers*
const c

non-const quad
*vers*
const lst

# La qualification const

- *Une donnée const ne peut pas être confiée à un paramètre non-const !*
- *Le qualité de constance est contagieuse de l'appelant vers l'appelé : **tous les appelés doivent coopérer***

C++

```
void afficherCoords(Coords& c)
{
    std::cout << c.x << " " << c.y;
}

void afficherListeCoords(const std::vector<Coords>& lst)
{
    for (size_t i=0; i<lst.size(); ++i)
        afficherCoords(lst[i]);
}
```

const lst[i]
*vers*
non-const c

***error: binding 'const value_type {aka const Coords}' to reference of type 'Coords&' discards qualifiers***

# La qualification const

- *Indiquer et utiliser systématiquement la qualification* **`const`** *partout où cela fait sens est une* ***discipline***

- *Cette discipline doit partir du bas : les codes de bas niveau (appelés) sont ceux qui autorisent ou pas l'utilisation de const par les codes clients (appelants)*
  *Voir slide précédent*

- *Quand tous les développeurs adhèrent à cette rigueur le code est caractérisé par sa «* ***Const Correctness*** *»*

- ***Celà fait partie des « bonnes pratiques »***

# La qualification const

***https://isocpp.org/wiki/faq/const-correctness***

## FAQ Should I try to get things `const` correct "sooner" or "later"?

At the very, very, *very* beginning.

Back-patching `const` correctness results in a snowball effect: every `const` you add "over here" requires four more there."

Add `const` early and often.

## FAQ What do "`X const& x`" and "`X const* p`" mean?

`X const& x` is equivalent to `const X& x`, and `X const* x` is equivalent to `const X* x`.

# La qualification const

- *Au début ce n’est pas très gratifiant, ça n’ajoute aucune fonctionnalité au programme et souvent ça bloque la compilation, on se sent gêné, mais …*

- *C’est sécure*

- *Code de qualité*

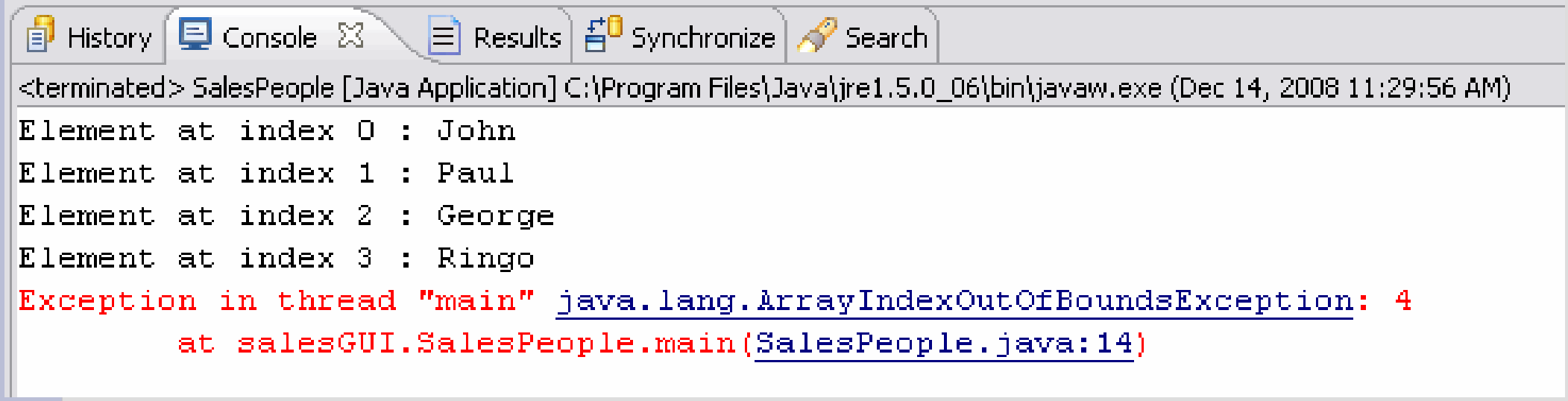
- *C’est pro*

# La qualification const

- *Attention cependant, le C++ est un code exécuté sans superviseur et les indices tableau non contrôlés : n'importe quel sous-programme peut écrabouiller n'importe quel octet de l'application !*
- *Il suffit d'un seul* **`tab[i]=50;`** *avec i trop grand !*
- *En C++ const est une protection des données contre certains types d'accidents, pas contre toutes les erreurs de la vie du programmeur, ni la malveillance...*

# La qualification const

- *Les langages supervisés offrent un niveau de protection des données supérieur*
- *Ils peuvent crasher mais ils disent où et pourquoi*



# La qualification const

- *Les langages supervisés offrent un niveau de protection des données supérieur (mais moins de perfs)*

*Programmation de haut niveau en toute décontraction*

# COURS 4

**A) Structs simples**
**B) Pointeurs * et Références &**
**C) La qualification const**
**D) Valeurs paramètres par défaut**
**E) Surcharge de fonctions**
**F) Surcharge d'opérateurs**
**G) Allocation dynamique new/delete**

# Valeurs paramètres par défaut

# Valeurs paramètres par défaut

- *Une option sympathique du C++ :*
  *donner une valeur initiale aux paramètres*
  *quand ceux-ci ne sont pas explicitement donnés*
- *Ce sont des **valeurs par défaut***

**C++**

```
void dessinerCarre(int taille,
                   char remplissage = '*',
                   std::string titre = "Carré")
{
    std::cout << titre << std::endl;
    for (int lig=0; lig<taille; ++lig)
    {
        for (int col=0; col<taille; ++col)
            std::cout << remplissage;
        std::cout << std::endl;
    }
}

int main()
{
    dessinerCarre(3);
}
```

*valeurs par défaut*

```
Carré
***
***
***
```

*2ème et 3ème paramètres pas explicitement donnés*

# Valeurs paramètres par défaut

- *Une option sympathique du C++ :*
  *donner une valeur initiale aux paramètres*
  *quand ceux-ci ne sont pas explicitement donnés*
- *Ce sont des **valeurs par défaut***

C++

```
void dessinerCarre(int taille,
                   char remplissage = '*',
                   std::string titre = "Carré")
{
    std::cout << titre << std::endl;
    for (int lig=0; lig<taille; ++lig)
    {
        for (int col=0; col<taille; ++col)
            std::cout << remplissage;
        std::cout << std::endl;
    }
}

int main()
{
    dessinerCarre(2, '+');
```

```
Carré
++
++
```

# Valeurs paramètres par défaut

- *Une option sympathique du C++ :*
  *donner une valeur initiale aux paramètres*
  *quand ceux-ci ne sont pas explicitement donnés*
- *Ce sont des **valeurs par défaut***

C++

```
void dessinerCarre(int taille,
                   char remplissage = '*',
                   std::string titre = "Carré")
{
    std::cout << titre << std::endl;
    for (int lig=0; lig<taille; ++lig)
    {
        for (int col=0; col<taille; ++col)
            std::cout << remplissage;
        std::cout << std::endl;
    }
}

int main()
{
    dessinerCarre(4, '#', "Gros carré avec dièses");
```

```
Gros carré avec dièses
####
####
####
####
```

# Valeurs paramètres par défaut

- *Les paramètres par défaut sont toujours en dernier*
- *A l'appel ils sont spécifiés dans l'ordre du 1er au dernier*
- ***On ne peut pas « sauter » un paramètre***

C++

```
void dessinerCarre(int taille,
                   char remplissage = '*',
                   std::string titre = "Carré")
{
    std::cout << titre << std::endl;
    for (int lig=0; lig<taille; ++lig)
    {
        for (int col=0; col<taille; ++col)
            std::cout << remplissage;
        std::cout << std::endl;
    }
}

int main()
{
    dessinerCarre(3, "Carré avec étoiles");
```

*error: invalid conversion from 'const char*' to 'char'*

# Valeurs paramètres par défaut

- *Les paramètres par défaut sont toujours en dernier*
- *A l'appel ils sont spécifiés dans l'ordre du 1er au dernier*
- ***On ne peut pas « sauter » un paramètre***

C++

```
void dessinerCarre(int taille,
                   char remplissage = '*',
                   std::string titre = "Carré")
{
    std::cout << titre << std::endl;
    for (int lig=0; lig<taille; ++lig)
    {
        for (int col=0; col<taille; ++col)
            std::cout << remplissage;
        std::cout << std::endl;
    }
}

int main()
{
    dessinerCarre(3,"Carré avec étoiles");
}
```

*error: invalid conversion from 'const char*' to 'char'*

# Valeurs paramètres par défaut

- *Les paramètres par défaut sont toujours en dernier*
- *A l'appel ils sont spécifiés dans l'ordre du 1er au dernier*
- ***Ici OK, les 3 paramètres sont donnés explicitement***

C++

```
void dessinerCarre(int taille,
                   char remplissage = '*',
                   std::string titre = "Carré")
{
    std::cout << titre << std::endl;
    for (int lig=0; lig<taille; ++lig)
    {
        for (int col=0; col<taille; ++col)
            std::cout << remplissage;
        std::cout << std::endl;
    }
}

int main()
{
    dessinerCarre(3, '*', "Carré avec étoiles");
```

```
Carré avec étoiles
***
***
***
```

# Valeurs paramètres par défaut

- *On va souvent utiliser une/des **valeur(s) par défaut** pour **neutraliser** un/des paramètre(s)*
- *On verra une alternative au chapitre suivant (surcharge)*

C++

```cpp
/// 0 est l'élément neutre pour l'addition
int somme(int a, int b, int c=0)
{
    return a + b + c;
}



int main()
{
    std::cout << somme(1, 2) << " " << somme(1, 2, 3) << std::endl;
```

```
3 6
```

# Valeurs paramètres par défaut

- *On va souvent utiliser une/des **valeur(s) par défaut** pour **neutraliser** un/des paramètre(s)*
- *On verra une alternative au chapitre suivant (surcharge)*

C++

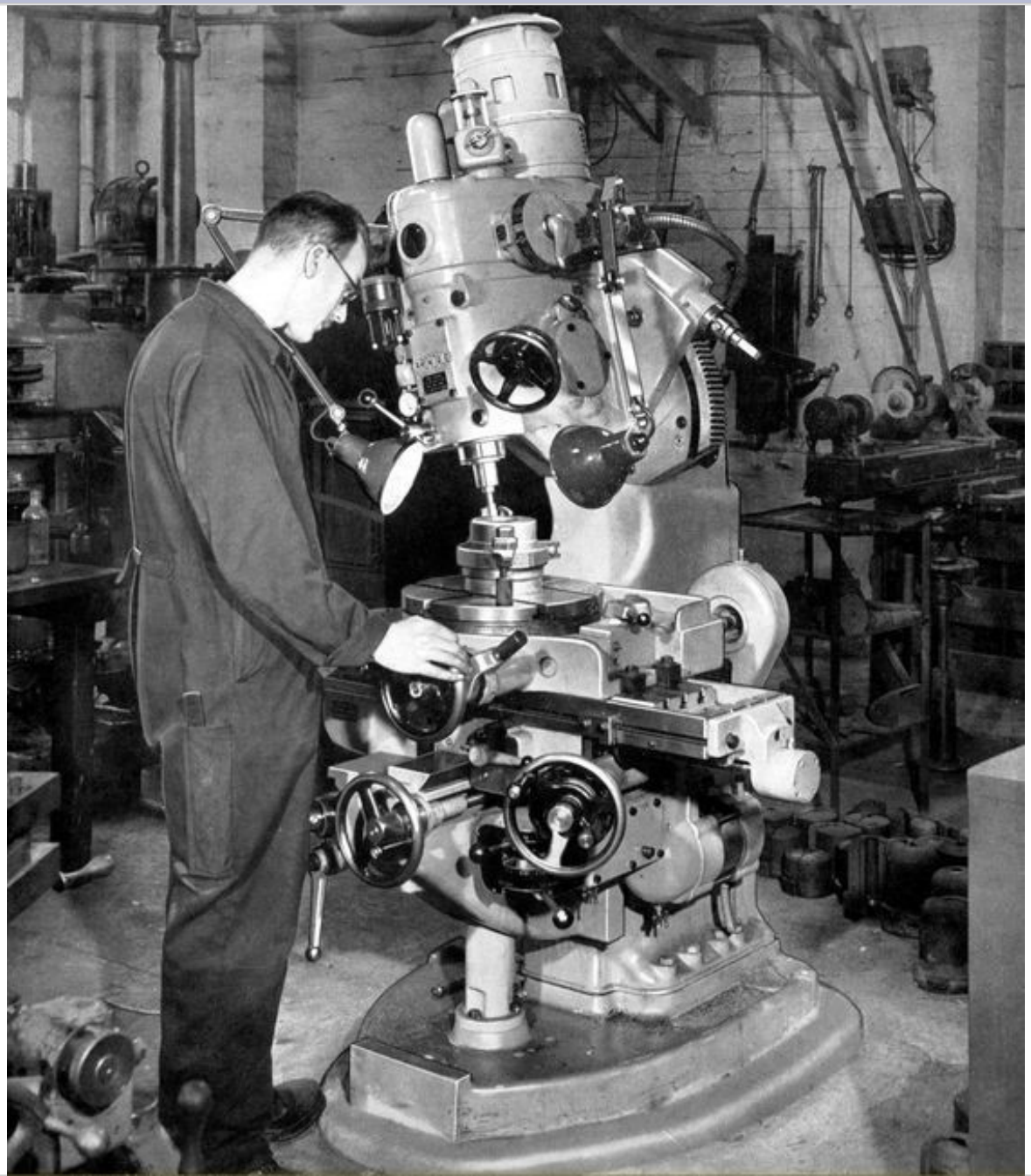
```
/// La plus petite valeur possible est élément neutre pour le max
int maxi(int a, int b, int c=std::numeric_limits<int>::min())
{
    if (a>b && a>c)
        return a;
    if (b>c)
        return b;
    return c;
}


int main()
{
    std::cout << maxi(1, 2) << " " << maxi(1, 2, 3) << std::endl;
```

```
2 3
```

# COURS 4

A) Structs simples
B) Pointeurs * et Références &
C) La qualification const
D) Valeurs paramètres par défaut
E) Surcharge de fonctions
F) Surcharge d'opérateurs
G) Allocation dynamique new/delete

# Surcharge de fonctions



# Surcharge de fonctions

- *On n'est pas obligé de donner des noms différents à des sous-programmes qui ont un rôle similaire mais qui prennent des listes de paramètres différentes*
- *C'est la surcharge (overloading)*

**C++**

```
/// moyenne à trois paramètres
float moyenne(float a, float b, float c)
{
    return (a + b + c) / 3.0;
}

/// moyenne à deux paramètres
float moyenne(float a, float b)
{
    return (a + b) / 2.0;
}

int main()
{
    std::cout << moyenne(1, 2) << " " << moyenne(1, 2, 3) << std::endl;
```

*On dit que la fonction moyenne est **surchargée***

*Fonction ou sous-programme : synonymes*

```
1.5 2
```

# Surcharge de fonctions

- *On n'est pas obligé de donner des noms différents à des sous-programmes qui ont un rôle similaire mais qui prennent des listes de paramètres différentes*
- *Le compilateur met en correspondance (overload resolution)*

C++

```
/// moyenne à trois paramètres
float moyenne(float a, float b, float c)
{
    return (a + b + c) / 3.0;
}

/// moyenne à deux paramètres
float moyenne(float a, float b)
{
    return (a + b) / 2.0;
}

int main()
{
    std::cout << moyenne(1, 2) << " " << moyenne(1, 2, 3) << std::endl;
```

Conversion de 3 int vers 3 floats OK

Conversion de 2 int vers 2 floats OK

```
1.5 2
```

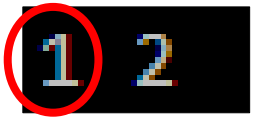
# Surcharge de fonctions

- *On n'est pas obligé de donner des noms différents à des sous-programmes qui ont un rôle similaire mais qui prennent des listes de paramètres différentes*
- *Ici l'approche paramètre par défaut n'était pas correcte*

C++

```
/// 0 est l'élément neutre pour la moyenne ?
float moyenne(float a, float b, float c=0)
{
    return (a + b + c) / 3.0;
}

int main()
{
    std::cout << moyenne(1, 2) << " " << moyenne(1, 2, 3) << std::endl;
```

FAUX !



# Surcharge de fonctions

- *On n'est pas obligé de donner des noms différents à des sous-programmes qui ont un rôle similaire mais qui prennent des listes de paramètres différentes*
- *La surcharge joue aussi sur les **types** des paramètres*

**saisies.h**
*Prototypes*

```
/// Saisie pour des phrases
void saisie(std::string& x);

/// Saisie pour des booléens oui/non
void saisie(bool& x);

/// Saisie pour des entiers positifs
void saisie(unsigned int& x);
```

# Surcharge de fonctions

- *On n'est pas obligé de donner des noms différents à des sous-programmes qui ont un rôle similaire mais qui prennent des listes de paramètres différentes*
- *Surcharge combinée avec paramètre par défaut !*

**saisies.h**
*Prototypes*

```
/// Saisie pour des phrases
void saisie(std::string& x, std::string message = "");

/// Saisie pour des booléens oui/non
void saisie(bool& x, std::string message = "");

/// Saisie pour des entiers positifs
void saisie(unsigned int& x, std::string message = "");
```

*valeur par défaut*

# Surcharge de fonctions

- *On n'est pas obligé de donner des noms différents à des sous-programmes qui ont un rôle similaire mais qui prennent des listes de paramètres différentes*
- *Chaque implémentation va être spécifique...*

**saisies.cpp**
*Implémentations*

```
/// Saisie pour des phrases
void saisie(std::string& x, std::string message)
{
    if (!message.empty())
        std::cout << message;

    std::getline(std::cin, x);
}
```

*Noter ici pas de valeur par défaut :*
*la valeur par défaut est donnée*
***dans la déclaration (prototype)***
***pas dans la définition (implémentation)***

*la substitution du paramètre*
*par la valeur par défaut*
*s'opère au niveau de l'appelant ...*

# Surcharge de fonctions

- *On n'est pas obligé de donner des noms différents à des sous-programmes qui ont un rôle similaire mais qui prennent des listes de paramètres différentes*
- *Chaque implémentation va être spécifique...*

**saisies.cpp**
*Implémentations*

```
/// Saisie pour des booléens oui/non
void saisie(bool& x, std::string message)
{
    if (!message.empty())
        std::cout << message;

    std::string ligne;
    std::getline(std::cin, ligne);
    while ( ligne!="non" && ligne!="oui" )
    {
        std::cout << "Reponse [oui/non] attendue, recommencer : ";
        std::getline(std::cin, ligne);
    }

    x = ligne=="oui";
}
```

# Surcharge de fonctions

- *On n'est pas obligé de donner des noms différents à des sous-programmes qui ont un rôle similaire mais qui prennent des listes de paramètres différentes*
- *Chaque implémentation va être spécifique...*

**saisies.cpp**
*Implémentations*

```
/// Saisie pour des entiers positifs
void saisie(unsigned int& x, std::string message)
{
    if (!message.empty())
        std::cout << message;

    std::string ligne;
    bool correct;
    do
    {
        std::getline(std::cin, ligne);
        correct = ligne.size()>0 && ligne.size()<10;
        for (size_t i=0; correct && i<ligne.size(); ++i)
            if ( ligne[i]<'0' || ligne[i]>'9' )
                correct = false;
        if (!correct)
            std::cout << "Entier positif attendu, recommencer : ";
    } while (!correct);

    x = std::stoul(ligne);
}
```

*Seule façon de « blinder » une saisie utilisateur : entrer une **ligne** sous forme de chaîne puis analyser la ligne puis accepter ou rejeter*

# Surcharge de fonctions

- *On n'est pas obligé de donner des noms différents à des sous-programmes qui ont un rôle similaire mais qui prennent des listes de paramètres différentes*
- *Ce gros travail est payant au niveau du* ***code client***

**main.cpp**
*Code utilisateur*

```
int main()
{

    std::string  val1;
    bool         val2;
    unsigned int val3;

    saisie(val1);
    saisie(val2);
    saisie(val3);

    std::cout << std::endl;

    std::cout << "valeur 1 saisie : " << val1 << std::endl;
    std::cout << "valeur 2 saisie : " << val2 << std::endl;
    std::cout << "valeur 3 saisie : " << val3 << std::endl;
```

# Surcharge de fonctions

- *On n'est pas obligé de donner des noms différents à des sous-programmes qui ont un rôle similaire mais qui prennent des listes de paramètres différentes*
- *Ce gros travail est payant au niveau du* ***code client***

**main.cpp**
*Code utilisateur*

```
int main()
{

    std::string  val1;
    bool         val2;
    unsigned int val3;

    saisie(val1, "Une phrase SVP : ");
    saisie(val2, "Avez vous vu le Big Bang [oui/non] ? ");
    saisie(val3, "Votre age SVP : ");

    std::cout << std::endl;

    std::cout << "valeur 1 saisie : " << val1 << std::endl;
    std::cout << "valeur 2 saisie : " << val2 << std::endl;
    std::cout << "valeur 3 saisie : " << val3 << std::endl;
```

# Surcharge de fonctions

- *Un cession interactive avec le code précédent*
- *En vert les saisies utilisateur*
- *Le blindage des entiers positifs est à améliorer...*

```
Une phrase SVP : Bonjour le Monde !
Avez vous vu le Big Bang [oui/non] ? peut-être
Reponse [oui/non] attendue, recommencer : euh
Reponse [oui/non] attendue, recommencer : oui
Votre age SVP : 15000000000
Entier positif attendu, recommencer : quinze milliards
Entier positif attendu, recommencer : -1
Entier positif attendu, recommencer : 9999999999
Entier positif attendu, recommencer : 999999999

valeur 1 saisie : Bonjour le Monde !
valeur 2 saisie : 1
valeur 3 saisie : 999999999

Process returned 0 (0x0)   execution time : 46.378 s
Press any key to continue.
```

***OVERLORD ≠ OVERLOAD***

# COURS 4

A) Structs simples
B) Pointeurs * et Références &
C) La qualification const
D) Valeurs paramètres par défaut
E) Surcharge de fonctions
F) Surcharge d’opérateurs
G) Allocation dynamique new/delete

# Surcharge d’opérateurs

# Surcharge d'opérateurs

- *En C++ on peut "customiser" les **opérateurs** qui sont en fait considérés **comme des fonctions** surchargées*
- *Le nom des ces fonctions est operator+ operator- operator* operator/ operator% etc...*

**C++**

```
+   -   *   /   %   ^   &   |   ~   !   =
<   >   +=   -=   *=   /=   %=   ^=   &=
|=   <<   >>   >>=   <<=   ==   !=   <=
>=   &&   ||   ++   --   ,   ->   () []
```

https://en.cppreference.com/w/cpp/language/operators

https://isocpp.org/wiki/faq/operator-overloading

# Surcharge d'opérateurs

- *En C++ on peut "customiser" les* ***opérateurs*** *qui sont en fait considérés* ***comme des fonctions*** *surchargées*
- *Utiliser + entre 2 variables de type Type appellera*

```
Type operator+(const Type& t1, const Type& t2)
```

**C++**

```
+   -   *   /   %   ^   &   |   ~   !   =
<   >   +=   -=   *=   /=   %=   ^=   &=
|=   <<   >>   >>=   <<=   ==   !=   <=
>=   &&   ||   ++   --   ,   ->   () []
```

https://en.cppreference.com/w/cpp/language/operators

https://isocpp.org/wiki/faq/operator-overloading

# Surcharge d'opérateurs

- *En C++ on peut "customiser" les **opérateurs** qui sont en fait considérés **comme des fonctions** surchargées*
- *On pourra définir ces fonctions opérateurs pour enrichir le langage et faciliter l'écriture du code...*

**C++**

```cpp
/// Somme vectorielle (version longue)
Coords operator+(const Coords& c1, const Coords& c2)
{
    Coords s;
    s.x = c1.x + c2.x;
    s.y = c1.y + c2.y;
    return s;
}

int main()
{
    Coords a{1, 2};
    Coords b{2, 3};
    Coords c;
    c = a + b;

    std::cout << "(" << c.x << ", " << c.y << ")" << std::endl;
```

```
(3, 5)
```

# Surcharge d'opérateurs

- *En C++ on peut "customiser" les **opérateurs** qui sont en fait considérés **comme des fonctions** surchargées*
- *On pourra définir ces fonctions opérateurs pour enrichir le langage et faciliter l'écriture du code...*

**C++**

```cpp
/// Somme vectorielle (version longue)
Coords operator+(const Coords& c1, const Coords& c2)
{
    Coords s;
    s.x = c1.x + c2.x;
    s.y = c1.y + c2.y;
    return s;
}

int main()
{
    Coords a{1, 2};
    Coords b{2, 3};
    Coords c;
    c = a + b;

    std::cout << "(" << c.x << ", " << c.y << ")" << std::endl;
```

**a+b** *déclenche l'appel* **operator+(a, b)**

```
(3, 5)
```

# Surcharge d'opérateurs

- *En C++ on peut "customiser" les **opérateurs** qui sont en fait considérés **comme des fonctions** surchargées*
- *On pourra définir ces fonctions opérateurs pour enrichir le langage et faciliter l'écriture du code...*

**C++**

```
/// Somme vectorielle (version courte)
Coords operator+(const Coords& c1, const Coords& c2)
{
    return {c1.x + c2.x, c1.y + c2.y};
}

/// Produit scalaire
double operator*(const Coords& c1, const Coords& c2)
{
    return c1.x*c2.x + c1.y*c2.y;
}

/// Multiplication par un réel
Coords operator*(double m, const Coords& c)
{
    return {m*c.x, m*c.y};
}
```

# Surcharge d'opérateurs

- *En C++ on peut "customiser" les **opérateurs** qui sont en fait considérés **comme des fonctions** surchargées*
- *En surchargeant l'opérateur d'insertion on rend les des classes d'objets compatibles avec std::cout*

C++

```
/// Opérateur d'insertion dans un flot d'affichage
std::ostream& operator<<(std::ostream& out, const Coords& c)
{
    out << "(" << c.x << ", " << c.y << ")";
    return out;
}

int main()
{
    Coords a{1, 2};
    Coords b{2, 3};

    std::cout << a << std::endl;
    std::cout << b << std::endl;
```

```
(1, 2)
(2, 3)
```

# Surcharge d'opérateurs

- *Finalement on peut **intégrer** les objets d'une classe dans le langage presque comme un type élémentaire...*
- *Complique la vie de ceux qui développent une classe mais facilite le travail des utilisateurs de la classe*

**C++**

```
int main()
{
    Coords a{1, 2};
    Coords b{2, 3};

    std::cout << a << std::endl;
    std::cout << b << std::endl << std::endl;

    Coords c;
    c = a + b;
    std::cout << c << std::endl << std::endl;

    Coords d = 4*b;
    std::cout << d << std::endl << std::endl;

    std::cout << a+b << std::endl;
    std::cout << a*b << std::endl;
    std::cout << -1*a+3*b << std::endl;
    std::cout << a*b*(a+b) << std::endl;
```

```
(1, 2)
(2, 3)

(3, 5)

(8, 12)

(3, 5)
8
(5, 7)
(24, 40)
```

# COURS 4

**A) Structs simples**
**B) Pointeurs * et Références &**
**C) La qualification const**
**D) Valeurs paramètres par défaut**
**E) Surcharge de fonctions**
**F) Surcharge d'opérateurs**
**G) Allocation dynamique new/delete**

# Allocation dynamique new/delete

# Allocation dynamique new/delete

| | C | C++ |
|---|---|---|
| allouer | malloc | new |
| libérer | free | delete |

C++

```
void afficher(const Coords& c);
void saisir(Coords& c);

int main()
{
    Coords* pa = nullptr;

    pa = new Coords;

    saisir(*pa);
    afficher(*pa);

    delete pa;

    return 0;
}
```

# Allocation dynamique new/delete

| | C | C++ |
|---|---|---|
| allouer | malloc | new |
| libérer | free | delete |

C++

```
void afficher(const Coords& c);
void saisir(Coords& c);

int main()
{
    Coords* pa = nullptr;
                        allouer
    pa = new Coords;

    saisir(*pa);
    afficher(*pa);      utiliser !

    delete pa;
                        libérer
    return 0;
}
```

# Allocation dynamique new/delete

| | C | C++ |
|---|---|---|
| allouer | malloc | new |
| libérer | free | delete |

C++

```
void afficher(const Coords& c);
void saisir(Coords& c);

int main()
{
    Coords* pa = nullptr;

    pa = new Coords;

    saisir(*pa);
    afficher(*pa);

    delete pa;

    return 0;
}
```

déréférencer ici, on a un **pointeur sur Coords**
les paramètres attendent un **Coords**
( une référence sur Coords attend le type Coords, pas Coords* )

# Allocation dynamique new/delete

- *C'était un exemple d'**allocation dynamique***
- ***Si on peut éviter d'allouer on évite !***
- *Ici il est préférable d'utiliser une variable **automatique***

C++

```
void afficher(const Coords& c);
void saisir(Coords& c);

int main()
{
    Coords a;



    saisir(a);
    afficher(a);



    return 0;
}
```

# Allocation dynamique new/delete

- *C'était un exemple d'**allocation dynamique***
- ***Si on peut éviter d'allouer on évite !***
- *Ici il est préférable d'utiliser une variable **automatique***

C++

```
void afficher(const Coords& c);
void saisir(Coords& c);

int main()
{
    Coords a;

    saisir(a);
    afficher(a);

    return 0;
}
```

déclarer suffit à allouer l'espace de stockage des données

utiliser !

libération automatique à la fermeture du scope

# Allocation dynamique new/delete

- *Initialisation d'une variable **automatique***
- *Il y a des variantes (en général équivalentes)*

```
Coords a = {5, 6};  // Historique (compatible C) implique une copie
Coords a(5, 6);     // C++ classique (C++98) appel au constructeur
Coords a{5, 6};     // C++ moderne (C++11) braced initialization
```

C++

```
void afficher(const Coords& c);
void saisir(Coords& c);

int main()
{
    Coords a{5, 6};



    afficher(a);



    return 0;
}
```

# Allocation dynamique new/delete

- *Initialisation d'un objet alloué **dynamiquement***
- *C'est nouveau (avec malloc on ne pouvait pas)*
- *On verra avec les classes : ici constructeur implicite*

C++

```
void afficher(const Coords& c);
void saisir(Coords& c);

int main()
{
    Coords* pa = nullptr;

    pa = new Coords{5, 6};


    afficher(*pa);


    delete pa;

    return 0;
}
```

# Allocation dynamique new/delete

- *Bon mais alors si on peut faire en gros les même choses avec des données automatiques alors les données allouées dynamiquement, ça sert à quoi ?*
- *Par exemple à retourner des données sans copie*

C++

```
Coords* faireCoordsDiago(double z)
{
    Coords* pc = nullptr;
    pc = new Coords;
    pc->x = z;
    pc->y = z;
    return pc;
}
```

```
int main()
{
    Coords* pa = nullptr;

    pa = faireCoordsDiago(3);

    afficher(*pa);

    delete pa;
    return 0;
}
```

**Communication à l'appelant d'un nouvel espace de stockage : 4 octets à transmettre quel que soit la taille de l'espace de stockage**

# Allocation dynamique new/delete

- *Bon mais alors si on peut faire en gros les même choses avec des données automatiques alors les données allouées dynamiquement, ça sert à quoi ?*
- *Par exemple à retourner des données sans copie*

**C++**

**Équivalent au code précédent avec une rédaction plus compacte**

```
Coords* faireCoordsDiago(double z)
{
    return new Coords{z, z};
}

int main()
{
    Coords* pa = faireCoordsDiago(3);


    afficher(*pa);

    delete pa;
    return 0;
}
```

**Attention avec un objet alloué retourné par une fonction appelée, l'appelant devient responsable de la gestion du cycle de vie de cet objet. Il doit soit le libérer soit en confier la responsabilité à un autre etc...**

# Allocation dynamique new/delete

- *Un « retour par valeur » copie plus de données*
- *Ici un passage par référence de* a *serait possible parce que l'appelant a déjà l'espace de stockage à remplir mais ce n'est pas toujours le cas*

C++

```
Coords faireCoordsDiago(double z)
{
    Coords c;

    c.x = z;
    c.y = z;
    return c;
}
```

```
int main()
{
    Coords  a;

    a  = faireCoordsDiago(3);

    afficher(a);


    return 0;
}
```

**Copie à l'appelant des données**
**nombre d'octets à transmettre proportionnel à la taille des données : potentiellement très grand !**

# Allocation dynamique new/delete

- *Là où l'allocation dynamique est indispensable c'est lors de la création des objets persistants qui vont peupler nos collections : « les instances » du modèle*
- *Les objets arrivent et quittent un logiciel orienté objet en fonction des besoins :*
  ***besoin d'un nouvel objet -> appeler new***
  ***plus besoin d'un objet -> appeler delete***
- *Slide suivant :*
  *un ajout « tant qu'on veut » de nouvelles coords dans une collection de Coords, un cas typique d'utilisation de l'allocation dynamique*

# Allocation dynamique new/delete

```
void utiliser(std::vector<Coords*>& lst);
void saisir(Coords& c);

int main()
{
    std::vector<Coords*> mesCoords;

    bool continuer;
    do
    {
        Coords* nouveau = new Coords;
        saisir(*nouveau);

        mesCoords.push_back(nouveau);

        std::cout << "continuer [true/false] ?" << std::endl;
        std::cin >> std::boolalpha >> continuer;
    } while (continuer);

    utiliser(mesCoords);

    for(size_t i=0; i<mesCoords.size(); ++i)
        delete mesCoords[i];

    return 0;
}
```

**Fabriquer des objets en fonction de la demande utilisateur, un cas typique d'utilisation de l'allocation**

# Sécurité & savoir-faire en 1913

# Relire les cours

- *Les cours font beaucoup de slides mais beaucoup de slides se répètent pour faire des « effets d'animation » par exemple avec l'apparition de commentaires etc...*
- *En mode défilement cette présentation peut être fastidieuse. En configurant le lecteur de pdf vous pouvez faire défiler les slides comme en cours :*