

LANGUAGE OBJ. AV. (C++) MASTER 1

Ufr d'Informatique - Yan Jurski

CORRECTIONS :

- **QCM 2024-2025**
- **EXAMEN 2022-2023**
- **RATTRAPAGE 2022-2023**

Question 1

Voici une liste d'affirmations :

1. À la compilation, un fichier `.o` est généré pour chaque fichier source du programme.
2. Les déclarations d'un programme sont normalement placées dans les fichiers `.hpp`.
3. L'extrait de code

```
int t[] { 4, 0, 9 };  
int *p{t};  
*p++;  
cout << "Décalage : " << p - t << " - Contenu : "  
for (int x:t) cout << x << " ";
```

affiche : |Décalage : 0 - Contenu : 5 0 9|

4. Le programme ci-dessous compile :

```
#include <vector>  
  
class Forest;  
class Tree {  
public:  
    int label;  
    Forest children;  
};  
  
class Forest {  
public:  
    std::vector<Tree> trees;  
};
```

Question 1

Voici une liste d'affirmations :

1. À la compilation, un fichier .o est généré pour chaque fichier source du programme.
2. Les déclarations d'un programme sont normalement placées dans les fichiers .hpp.
3. L'extrait de code

```
int t[] { 4, 0, 9 };
int *p{t};
*p++;
cout << "Décalage : " << p - t << " - Contenu : "
for (int x:t) cout << x << " ";
```

affiche : |Décalage : 0 - Contenu : 5 0 9|

4. Le programme ci-dessous compile :

```
#include <vector>

class Forest;
class Tree {
public:
    int label;
    Forest children;
};

class Forest {
public:
    std::vector<Tree> trees;
};
```

Inexact.

Dans vos sources vous avez des .cpp qui eux seront effectivement associés à des .o mais vous avez aussi des .hpp (ou des .hpp)

Question 1

Voici une liste d'affirmations :

1. À la compilation, un fichier .o est généré pour chaque fichier source du programme.
2. Les déclarations d'un programme sont normalement placées dans les fichiers .hpp.
3. L'extrait de code

```
int t[] { 4, 0, 9 };
int *p{t};
*p++;
cout << "Décalage : " << p - t << " - Contenu : "
for (int x:t) cout << x << " ";
```

affiche : |Décalage : 0 - Contenu : 5 0 9|

4. Le programme ci-dessous compile :

```
#include <vector>

class Forest;
class Tree {
public:
    int label;
    Forest children;
};

class Forest {
public:
    std::vector<Tree> trees;
};
```

Exact.
C'est la bonne pratique

Question 1

Voici une liste d'affirmations :

1. À la compilation, un fichier .o est généré pour chaque fichier source du programme.
2. Les déclarations d'un programme sont normalement placées dans les fichiers .hpp.
3. L'extrait de code

```
int t[] { 4, 0, 9 };
int *p{t};
*p++;
cout << "Décalage : " << p - t << " - Contenu : "
for (int x:t) cout << x << " ";
```

affiche : |Décalage : 0 - Contenu : 5 0 9|

4. Le programme ci-dessous compile :

```
#include <vector>

class Forest;
class Tree {
public:
    int label;
    Forest children;
};

class Forest {
public:
    std::vector<Tree> trees;
};
```

Inexact.

La question porte sur la priorité de ++ sur *
Le ++ opère d'abord sur l'adresse de p, puis on déréférence son ancienne valeur.

L'expression *p++ est donc évaluée à 4, qui n'est pas utilisé.
p, modifié, pointe sur le second élément de t

Décalage : 1
Contenu : 4 0 9

Question 1

Voici une liste d'affirmations :

1. À la compilation, un fichier .o est généré pour chaque fichier source du programme.
2. Les déclarations d'un programme sont normalement placées dans les fichiers .hpp.
3. L'extrait de code

```
int t[] { 4, 0, 9 };  
int *p{t};  
*p++;  
cout << "Décalage : " << p - t << " - Contenu : "  
for (int x:t) cout << x << " ";
```

affiche : |Décalage : 0 - Contenu : 5 0 9|

4. Le programme ci-dessous compile :

```
#include <vector>  
  
class Forest;  
class Tree {  
public:  
    int label;  
    Forest children;  
};  
  
class Forest {  
public:  
    std::vector<Tree> trees;  
};
```

Inexact.
Malgré la déclaration préalable.
On indique ici qu'on utilise une variable Forest.

Quelle est sa taille ?
Comment la construire ?

Ces questions ne se poseraient pas avec un pointeur ou une référence

Question 1

Voici une liste d'affirmations :

1. À la compilation, un fichier .o est généré pour chaque fichier source du programme.
2. Les déclarations d'un programme sont normalement placées dans les fichiers .hpp.
3. L'extrait de code

```
int t[] { 4, 0, 9 };
int *p{t};
*p++;
cout << "Décalage : " << p - t << " - Contenu : "
for (int x:t) cout << x << " ";
```

affiche : |Décalage : 0 - Contenu : 5 0 9|

4. Le programme ci-dessous compile :

```
#include <vector>

class Forest;
class Tree {
public:
    int label;
    Forest children;
};

class Forest {
public:
    std::vector<Tree> trees;
};
```

Quelle est sa taille ?
Comment la construire ?

Ces même questions se poseraient ici alors qu'elles auraient été laissée en suspend pour Tree ...

5. Le programme ci-dessous compile :

```
#include <iostream>
class B;
class C;
class A {
    private:
        static int x;
        friend class B;
};

class B {
    friend class C;
};

class C {
    public:
        static void f();
};

int A::x = 42;
void C::f() {
    std::cout << A::x << std::endl;
}

int main() {
    C::f();
}
```

5. Le programme ci-dessous compile :

```
#include <iostream>
class B;
class C;
class A {
    private:
        static int x;
        friend class B;
};

class B {
    friend class C;
};

class C {
    public:
        static void f();
};

int A::x = 42;
void C::f() {
    std::cout << A::x << std::endl;
}

int main() {
    C::f();
}
```

Inexact.
x est privé dans A

L'amitié se limite à la
classe déclarée friend.

Rq : ce serait le cas
même si C hérite de B

6. Dans le fichier 'Livre.hpp' ci-dessous, il n'est, pour l'instant, pas nécessaire d'insérer `#include "Personne.hpp"` au début.

```
class Personne;
class Livre {
    public:
        Personne *auteur;
};
```

7. Le programme ci-dessous

```
#include <iostream>

class A {
public:
    A() { std::cout << "A()"; }
    ~A() { std::cout << "~A()"; }
};

class B : public A {
public:
    B() { std::cout << "B()"; }
    ~B() { std::cout << "~B()"; }
};

int main() { B b; }
```

affiche : A()B()~B()~A()

6. Dans le fichier 'Livre.hpp' ci-dessous, il n'est, pour l'instant, pas nécessaire d'insérer `#include "Personne.hpp"` au début.

```
class Personne;
class Livre {
    public:
        Personne *auteur;
};
```

7. Le programme ci-dessous

```
#include <iostream>

class A {
public:
    A() { std::cout << "A()"; }
    ~A() { std::cout << "~A()"; }
};

class B : public A {
public:
    B() { std::cout << "B()"; }
    ~B() { std::cout << "~B()"; }
};

int main() { B b; }
```

affiche : A()B()~B()~A()

Exact.

La déclaration préalable d'une classe utilisée avec un pointeur ne pose pas de problème dans Livre puisque :

- sa taille est celle d'un pointeur
- on n'a pas à la construire
- aucune méthode particulière n'est appliquée sur auteur

6. Dans le fichier 'Livre.hpp' ci-dessous, il n'est, pour l'instant, pas nécessaire d'insérer `#include "Personne.hpp"` au début.

```
class Personne;
class Livre {
    public:
        Personne *auteur;
};
```

7. Le programme ci-dessous

```
#include <iostream>

class A {
public:
    A() { std::cout << "A()"; }
    ~A() { std::cout << "~A()"; }
};

class B : public A {
public:
    B() { std::cout << "B()"; }
    ~B() { std::cout << "~B()"; }
};

int main() { B b; }
```

affiche : A()B()~B()~A()

Exact.

La construction de `B:A()` `{...}` est implicite.

Puis la destruction se fait dans l'ordre inverse

Rq : Le caractère non virtuel des destructeurs n'intervient pas ici. B connaît ses « parties »

8. Le programme ci-dessous

```
#include <iostream>

class A {
public:
    A() { std::cout << "A()"; }
    ~A() { std::cout << "~A()"; }
};

class B : public A {
public:
    B() { std::cout << "B()"; }
    ~B() { std::cout << "~B()"; }
};

int main() {
    A *obj = new B;
    delete obj;
}
```

affiche : A()B()~B()~A()

8. Le programme ci-dessous

```
#include <iostream>

class A {
public:
    A() { std::cout << "A()"; }
    ~A() { std::cout << "~A()"; }
};

class B : public A {
public:
    B() { std::cout << "B()"; }
    ~B() { std::cout << "~B()"; }
};

int main() {
    A *obj = new B;
    delete obj;
}
```

affiche : A()B()~B()~A()

Inexact.

Le delete invoque le destructeur du type déclaré qui n'est pas virtuel, et donc reste au niveau de A.

A()B() à la création
~A() à la destruction.

Question 2 On donne la classe A suivant

```
class A {  
public:  
    A() : x_(0) { cout << "A() "; }  
    A(int x) : x_(x) { cout << "A(" << x << "  
    ~A() { cout << "~A():" << x_; }  
  
private:  
    int x_;  
};
```

A l'exécution du code suivant, quel est le bon affichage ?

```
int main() {  
    A a{12};  
    A b;  
}
```

Question 3

A l'exécution du code suivant, quel est le bon affichage ?

```
int main() {  
    for (int i = 0; i < 2; i++) {  
        A a{i};  
    }  
}
```

Question 4 A l'exécution du code suivant

```
int main() {  
    A* a_ptr[2]{NULL, NULL};  
    for (int i = 0; i < 2; i++) {  
        a_ptr[i] = new A(i);  
    }  
  
    for (int i = 0; i < 2; i++) {  
        delete a_ptr[i];  
    }  
}
```


Question 2 On donne la classe A suivant

```
class A {  
public:  
    A() : x_(0) { cout << "A() "; }  
    A(int x) : x_(x) { cout << "A(" << x << "  
    ~A() { cout << "~A():" << x_; }  
  
private:  
    int x_;  
};
```

A l'exécution du code suivant, quel est le bon affichage ?

```
int main() {  
    A a{12};  
    A b;  
}
```

A(12) A() ~A() : 0 ~A() : 12

Question 3

A l'exécution du code suivant, quel est le bon affichage ?

```
int main() {  
    for (int i = 0; i < 2; i++) {  
        A a{i};  
    }  
}
```

Question 4 A l'exécution du code suivant

```
int main() {  
    A* a_ptr[2]{NULL, NULL};  
    for (int i = 0; i < 2; i++) {  
        a_ptr[i] = new A(i);  
    }  
  
    for (int i = 0; i < 2; i++) {  
        delete a_ptr[i];  
    }  
}
```

Question 2 On donne la classe A suivant

```
class A {
public:
    A() : x_(0) { cout << "A() "; }
    A(int x) : x_(x) { cout << "A(" << x << "
    ~A() { cout << "~A():" << x_; }

private:
    int x_;
};
```

A l'exécution du code suivant, quel est le bon affichage ?

```
int main() {
    A a{12};
    A b;
}
```

Question 3

A l'exécution du code suivant, quel est le bon affichage ?

```
int main() {
    for (int i = 0; i < 2; i++) {
        A a{i};
    }
}
```

Question 4 A l'exécution du code suivant

```
int main() {
    A* a_ptr[2]{NULL, NULL};
    for (int i = 0; i < 2; i++) {
        a_ptr[i] = new A(i);
    }

    for (int i = 0; i < 2; i++) {
        delete a_ptr[i];
    }
}
```

A (0) ~ A () : 0 A (1) ~ A () : 1

Question 2 On donne la classe A suivant

```
class A {  
public:  
    A() : x_(0) { cout << "A() "; }  
    A(int x) : x_(x) { cout << "A(" << x << "  
    ~A() { cout << "~A():" << x_; }  
  
private:  
    int x_;  
};
```

A l'exécution du code suivant, quel est le bon affichage ?

```
int main() {  
    A a{12};  
    A b;  
}
```

Question 3

A l'exécution du code suivant, quel est le bon affichage ?

```
int main() {  
    for (int i = 0; i < 2; i++) {  
        A a{i};  
    }  
}
```

Question 4 A l'exécution du code suivant

```
int main() {  
    A* a_ptr[2]{NULL, NULL};  
    for (int i = 0; i < 2; i++) {  
        a_ptr[i] = new A(i);  
    }  
  
    for (int i = 0; i < 2; i++) {  
        delete a_ptr[i];  
    }  
}
```

A (0) A (1) ~ A () : 0 ~ A () : 1

Question 5 On donne le code suivant :

```
int f(int t) { return t; }
int g(int& t) { return t; }
int& h(int& t) { return t; }

int main(){
    int a = 67;
    int b = a;
    int& c = a;
    a = 12;
    cout << a << ", " << b << ", " << c << endl;

    int w = f(a);
    int x = g(a);
    int y = h(a);
    int& z = h(a);
    a = 95;
    cout << a << ", " << b << ", " << c << endl;
    cout << w << ", " << x << ", " << y << ", " << z << endl;

    return 0;
}
```

Question 5 On donne le code suivant :

12, 67, 12

```
int f(int t) { return t; }
int g(int& t) { return t; }
int& h(int& t) { return t; }

int main(){
    int a = 67;
    int b = a;
    int& c = a;
    a = 12;
    cout << a << ", " << b << ", " << c << endl;

    int w = f(a);
    int x = g(a);
    int y = h(a);
    int& z = h(a);
    a = 95;
    cout << a << ", " << b << ", " << c << endl;
    cout << w << ", " << x << ", " << y << ", " << z << endl;

    return 0;
}
```

Question 5 On donne le code suivant :

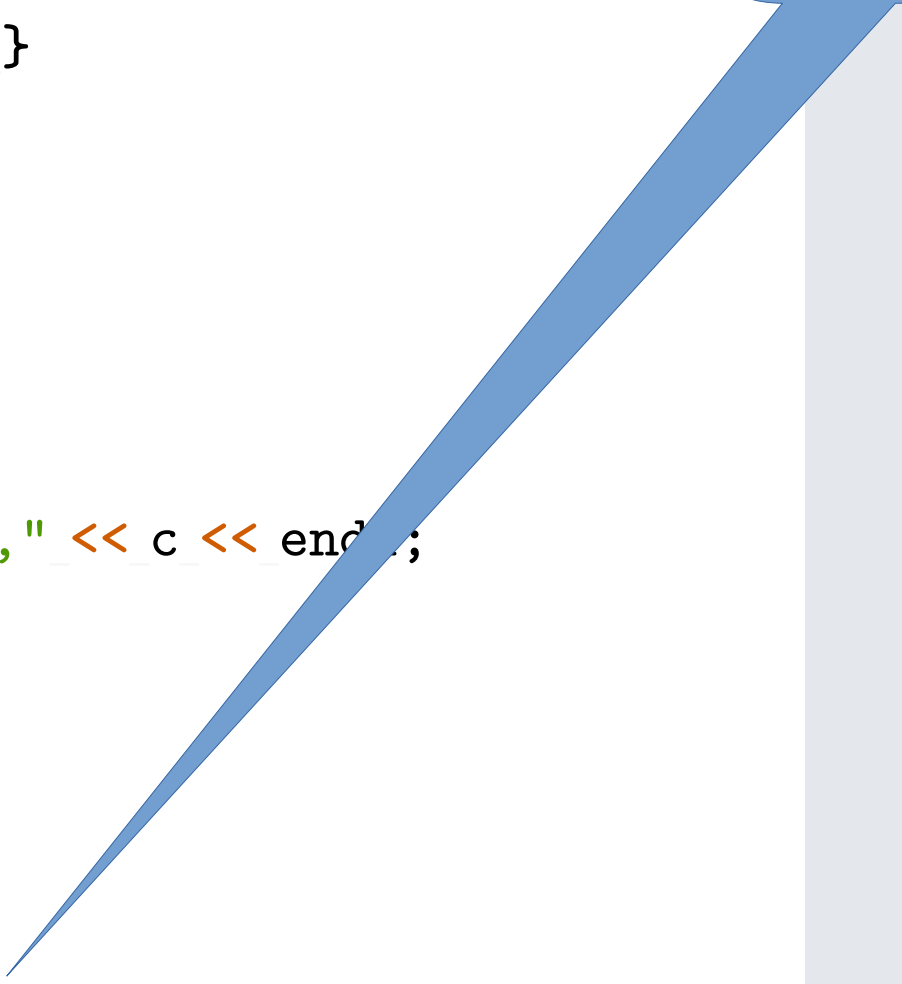
```
int f(int t) { return t; }
int g(int& t) { return t; }
int& h(int& t) { return t; }

int main(){
    int a = 67;
    int b = a;
    int& c = a;
    a = 12;
    cout << a << ", " << b << ", " << c << endl;

    int w = f(a);
    int x = g(a);
    int y = h(a);
    int& z = h(a);
    a = 95;
    cout << a << ", " << b << ", " << c << endl;
    cout << w << ", " << x << ", " << y << ", " << z << endl;

    return 0;
}
```

12, 67, 12
95, 67, 95



Question 5 On donne le code suivant :

```
int f(int t) { return t; }
int g(int& t) { return t; }
int& h(int& t) { return t; }

int main(){
    int a = 67;
    int b = a;
    int& c = a;
    a = 12;
    cout << a << ", " << b << ", " << c << endl;

    int w = f(a);
    int x = g(a);
    int y = h(a);
    int& z = h(a);
    a = 95;
    cout << a << ", " << b << ", " << c << endl;
    cout << w << ", " << x << ", " << y << ", " << z << endl;

    return 0;
}
```

12, 67, 12
95, 67, 95
12, 12, 12, 95

Question 6

Soit la définition de la classe Retour suivante :

```
class Retour {  
public:  
    Retour() : x(2024) {}  
    ... retour (); // à compléter  
  
private:  
    int x;  
};
```

Cocher les cases qui permettent de compléter ce code sans faire d'erreur à la compilation :

- ☐ A `const int& retour() const { return &x; }`
- ☐ B `int& retour() { return 2024; }`
- ☐ C `int retour() { return 2024; }`
- ☐ D `const int& retour() const { return x; }`
- ☐ E `int& retour() { return x; }`
- ☐ F `int retour() { return &x; }`

Question 6

Soit la définition de la classe Retour suivante :

```
class Retour {  
public:  
    Retour() : x(2024) {}  
    ... retour (); // à compléter  
  
private:  
    int x;  
};
```

Cocher les cases qui permettent de compléter ce code sans faire d'erreur à la compilation :

- ☐ A `const int& retour() const { return &x; }`
- ☐ B `int& retour() { return 2024; }`
- ☐ C `int retour() { return 2024; }`
- ☐ D `const int& retour() const { return x; }`
- ☐ E `int& retour() { return x; }`
- ☐ F `int retour() { return &x; }`

non : &x est une adresse

Question 6

Soit la définition de la classe Retour suivante :

```
class Retour {  
public:  
    Retour() : x(2024) {}  
    ... retour (); // à compléter  
  
private:  
    int x;  
};
```

Cocher les cases qui permettent de compléter ce code sans faire d'erreur à la compilation :

- ☐ A const int& retour() const { return &x; }
- ☐ B int& retour() { return 2024; }
- ☐ C int retour() { return 2024; }
- ☐ D const int& retour() const { return x; }
- ☐ E int& retour() { return x; }
- ☐ F int retour() { return &x; }

non : 2024 est un entier constant présent temporairement sur la pile, non référençable librement.

Question 6

Soit la définition de la classe Retour suivante :

```
class Retour {  
public:  
    Retour() : x(2024) {}  
    ... retour (); // à compléter  
  
private:  
    int x;  
};
```

Cocher les cases qui permettent de compléter ce code sans faire d'erreur à la compilation :

- ☐ A const int& retour() const { return &x; }
- ☐ B int& retour() { return 2024; }
- ☒ C int retour() { return 2024; }
- ☒ D const int& retour() const { return x; }
- ☒ E int& retour() { return x; }
- ☐ F int retour() { return &x; }

oui: 2024 est copié en retour

Question 6

Soit la définition de la classe Retour suivante :

```
class Retour {  
public:  
    Retour() : x(2024) {}  
    ... retour (); // à compléter  
  
private:  
    int x;  
};
```

Cocher les cases qui permettent de compléter ce code sans faire d'erreur à la compilation :

- ☐ A `const int& retour() const { return &x; }`
- ☐ B `int& retour() { return 2024; }`
- ☐ C `int retour() { return 2024; }`
- ☐ D `const int& retour() const { return x; }`
- ☐ E `int& retour() { return x; }`
- ☐ F `int retour() { return &x; }`

oui: et le const global impose d'avoir le const en retour si c'est une référence

Question 6

Soit la définition de la classe Retour suivante :

```
class Retour {  
public:  
    Retour() : x(2024) {}  
    ... retour (); // à compléter  
  
private:  
    int x;  
};
```

Cocher les cases qui permettent de compléter ce code sans faire d'erreur à la compilation :

- ☐ A const int& retour() const { return &x; }
- ☐ B int& retour() { return 2024; }
- ☐ C int retour() { return 2024; }
- ☐ D const int& retour() const { return x; }
- ☐ E int& retour() { return x; }
- ☐ F int retour() { return &x; }

oui

Question 6

Soit la définition de la classe Retour suivante :

```
class Retour {  
public:  
    Retour() : x(2024) {}  
    ... retour (); // à compléter  
  
private:  
    int x;  
};
```

Cocher les cases qui permettent de compléter ce code sans faire d'erreur à la compilation :

- ☐ A const int& retour() const { return &x; }
- ☐ B int& retour() { return 2024; }
- ☐ C int retour() { return 2024; }
- ☐ D const int& retour() const { return x; }
- ☐ E int& retour() { return x; }
- ☐ F int retour() { return &x; }

non : &x est une adresse

Question 7 On donne la déclaration de la classe A suivante :

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

class Machin{
private :
    const int n;
    int y;
    mutable int u;
public:
    Machin (int n, int y, int u)
        :n{n},y{y},u{u}{}
    int f() const;
    void g(int z) const;
    int h(int const z);
};
```

Cocher les définitions qui compilent :

- ☐ `void Machin::g(int z) const { n = z;}`
- ☐ `int Machin::h(int const z) { n= 2*z; return n; }`
- ☐ `int Machin::f() const { y = 0; return u;}`
- ☒ `int Machin::h(int const z) { u= z; return n; }`
- ☒ `int Machin::f() const { u = 0; return y;}`

Question 7 On donne la déclaration de la classe A suivante :

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

class Machin{
private :
    const int n;
    int y;
    mutable int u;
public:
    Machin (int n, int y, int u)
        :n{n},y{y},u{u}{}
    int f() const;
    void g(int z) const;
    int h(int const z);
};
```

Cocher les définitions qui compilent :

- ☐ A `void Machin::g(int z) const { n = z;}`
- ☐ B `int Machin::h(int const z) { n= 2*z; return n; }`
- ☐ C `int Machin::f() const { u = 0; return u;}`
- ☒ `int Machin::h(int const z) { u= z; return n; }`
- ☒ `int Machin::f() const { u = 0; return y;}`

non : n est déclaré const

Question 7 On donne la déclaration de la classe A suivante :

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

class Machin{
private :
    const int n;
    int y;
    mutable int u;
public:
    Machin (int n, int y, int u)
        :n{n},y{y},u{u}{}
    int f() const;
    void g(int z) const;
    int h(int const z);
};
```

Cocher les définitions qui compilent :

- ☐ **A** `void Machin::g(int z) const { n = z;}`
- ☐ **B** `int Machin::h(int const z) { n= 2*z; return n; }`
- ☐ **C** `int Machin::f() const { y = 0; return u;}`
- ☒ `int Machin::h(int const z) { u= z; return n; }`
- ☒ `int Machin::f() const { u = 0; return y;}`

non : n est déclaré const

Question 7 On donne la déclaration de la classe A suivante :

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

class Machin{
private :
    const int n;
    int y;
    mutable int u;
public:
    Machin (int n, int y, int u)
        :n{n},y{y},u{u}{}
    int f() const;
    void g(int z) const;
    int h(int const z);
};
```

Cocher les définitions qui compilent :

- ☐ **A** `void Machin::g(int z) const { n = z;}`
- ☐ **B** `int Machin::h(int const z) { n= 2*z; return n; }`
- ☐ **C** `int Machin::f() const { y = 0; return u;}`
- ☒ `int Machin::h(int const z) { u= z; return n; }`
- ☒ `int Machin::f() const { u = 0; return y;}`

non : ce const signifie que les attributs de l'objet courant ne changent pas

Question 7 On donne la déclaration de la classe A suivante :

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

class Machin{
private :
    const int n;
    int y;
    mutable int u;
public:
    Machin (int n, int y, int u)
        :n{n},y{y},u{u}{}
    int f() const;
    void g(int z) const;
    int h(int const z);
};
```

Cocher les définitions qui compilent :

- ☐ A `void Machin::g(int z) const { n = z;}`
- ☐ B `int Machin::h(int const z) { n= 2*z; return n; }`
- ☐ C `int Machin::f() const { y = 0; return u;}`
- ☒ `int Machin::h(int const z) { u= z; return n; }`
- ☒ `int Machin::f() const { u = 0; return y;}`

oui:

- la modification de u est possible
- le type retour est une copie

Question 7 On donne la déclaration de la classe A suivante :

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

class Machin{
private :
    const int n;
    int y;
    mutable int u;
public:
    Machin (int n, int y, int u)
        :n{n},y{y},u{u}{}
    int f() const;
    void g(int z) const;
    int h(int const z);
};
```

Cocher les définitions qui compilent :

- ☐ A `void Machin::g(int z) const { n = z;}`
- ☐ B `int Machin::h(int const z) { n= 2*z; return n; }`
- ☐ C `int Machin::f() const { y = 0; return u;}`
- ☒ `int Machin::h(int const z) { u= z; return n; }`
- ☒ `int Machin::f() const { u = 0; return y;}`

oui:
même si f est const, u est mutable

Question 8 On donne la déclaration et définition de la classe Bidule suivante :

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

class Bidule{
private:
    int x;
public :
    Bidule(int n):x{n}{};
    virtual ~Bidule(){}
    Bidule( const Bidule& a):x{a.x}{}
    int getX(){return x;}
};

Bidule g (Bidule b){ return b; }
int h (Bidule *c){ return c->getX(); }
```

Dans cet exercice et le suivant, on compile en refusant les optimisations liées aux constructeurs.

A l'exécution du code suivant, combien y a-t-il d'appels à des constructeurs directs, à des constructeurs de copies ?

```
int main() {
    Bidule *r = new Bidule(8);
    int i {h(r)};
    delete r;
    return EXIT_SUCCESS;
}
```

Question 8 On donne la déclaration et définition de la classe Bidule suivante :

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

class Bidule{
private:
    int x;
public :
    Bidule(int n):x{n}{};
    virtual ~Bidule(){}
    Bidule( const Bidule& a):x{a.x}{}
    int getX(){return x;}
};

Bidule g (Bidule b){ return b; }
int h (Bidule *c){ return c->getX(); }
```

1 construction
0 copie

Dans cet exercice et le suivant, on compile en refusant les optimisations liées aux constructeurs.

A l'exécution du code suivant, combien y a-t-il d'appels à des constructeurs directs, à des constructeurs de copies ?

```
int main() {
    Bidule *r = new Bidule(8);
    int i {h(r)};
    delete r;
    return EXIT_SUCCESS;
}
```

Question 8 On donne la déclaration et définition de la classe Bidule suivante :

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

class Bidule{
private:
    int x;
public :
    Bidule(int n):x{n}{};
    virtual ~Bidule(){}
    Bidule( const Bidule& a):x{a.x}{}
    int getX(){return x;}
};

Bidule g (Bidule b){ return b; }
int h (Bidule *c){ return c->getX(); }
```

Dans cet exercice et le suivant, on compile en refusant les optimisations liées aux constructeurs.

Question 9

A l'exécution du code suivant, combien y a-t-il d'appels à des constructeurs directs, à des constructeurs de copies ?

```
int main() {
    Bidule p{5};
    Bidule *q{&p};
    *q = g(p);
    return EXIT_SUCCESS;
}
```

Question 8 On donne la déclaration et définition de la classe Bidule suivante :

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

class Bidule{
private:
    int x;
public :
    Bidule(int n):x{n}{};
    virtual ~Bidule(){}
    Bidule( const Bidule& a):x{a.x}{}
    int getX(){return x;}
};

Bidule g (Bidule b){ return b; }
int h (Bidule *c){ return c->getX(); }
```

construction (1)

Dans cet exercice et le suivant, on compile en refusant les optimisations liées aux constructeurs.

Question 9

A l'exécution du code suivant, combien y a-t-il d'appels à des constructeurs directs, à des constructeurs de copies ?

```
int main() {
    Bidule p{5};
    Bidule *q{&p};
    *q = g(p);
    return EXIT_SUCCESS;
}
```


Question 8 On donne la déclaration et définition de la classe Bidule suivante :

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

class Bidule{
private:
    int x;
public :
    Bidule(int n):x{n}{};
    virtual ~Bidule(){}
    Bidule( const Bidule& a):x{a.x}{}
    int getX(){return x;}
};

Bidule g (Bidule b){ return b; }
int h (Bidule *c){ return c->getX(); }
```

copie (1)

Dans cet exercice et le suivant, on compile en refusant les optimisations liées aux constructeurs.

Question 9

A l'exécution du code suivant, combien y a-t-il d'appels à des constructeurs directs, à des constructeurs de copies ?

```
int main() {
    Bidule p{5};
    Bidule *q{&p};
    *q = g(p);
    return EXIT_SUCCESS;
}
```

Question 8 On donne la déclaration et définition de la classe Bidule suivante :

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

class Bidule{
private:
    int x;
public :
    Bidule(int n):x{n}{};
    virtual ~Bidule(){}
    Bidule( const Bidule& a):x{a.x}{}
    int getX(){return x;}
};

Bidule g (Bidule b){ return b; }
int h (Bidule *c){ return c->getX(); }
```

copie (2)

Dans cet exercice et le suivant, on compile en refusant les optimisations liées aux constructeurs.

Question 9

A l'exécution du code suivant, combien y a-t-il d'appels à des constructeurs directs, à des constructeurs de copies ?

```
int main() {
    Bidule p{5};
    Bidule *q{&p};
    *q = g(p);
    return EXIT_SUCCESS;
}
```

Question 8 On donne la déclaration et définition de la classe Bidule suivante :

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

class Bidule{
private:
    int x;
public :
    Bidule(int n):x{n}{};
    virtual ~Bidule(){}
    Bidule( const Bidule& a):x{a.x}{}
    int getX(){return x;}
};

Bidule g (Bidule b){ return b; }
int h (Bidule *c){ return c->getX(); }
```

cette affectation n'est pas faite par appel au constructeur de copie

Dans cet exercice et le suivant, on compile en retenant les optimisations liées aux constructeurs.

Question 9

A l'exécution du code suivant, combien y a-t-il d'appels à des constructeurs directs, à des constructeurs de copies ?

```
int main() {
    Bidule p{5};
    Bidule *q{&p};
    *q = g(p);
    return EXIT_SUCCESS;
}
```

Question 8 On donne la déclaration et définition de la classe Bidule suivante :

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

class Bidule{
private:
    int x;
public :
    Bidule(int n):x{n}{};
    virtual ~Bidule(){}
    Bidule( const Bidule& a):x{a.x}{}
    int getX(){return x;}
};

Bidule g (Bidule b){ return b; }
int h (Bidule *c){ return c->getX(); }
```

1 construction
2 copies

Dans cet exercice et le suivant, on compile en refusant les optimisations liées aux constructeurs.

Question 9

A l'exécution du code suivant, combien y a-t-il d'appels à des constructeurs directs, à des constructeurs de copies ?

```
int main() {
    Bidule p{5};
    Bidule *q{&p};
    *q = g(p);
    return EXIT_SUCCESS;
}
```

Question 10 On donne le code suivant :

```
class A{
private:
    int x;
public:
    A(int x): x{x}{}
    int get() const { return x; }
};

class B{
private:
    A a;
public:
    B(int x): a{x}{}
    int get() const { return a.get(); }
};
```

```
class C{
private:
    B *b;
public:
    C(int x): b{new B(x)}{}
    int get() const { return b->get(); }
};

int main(){
    int x = 6;
    C c{x};

    if(c.get()%2 == 0) { C c2{c}; }
    cout << c.get() << endl;

    return 0;
}
```

Cocher les cases qui donnent une réponse correcte :

- ☐ le code ne termine pas lors de l'exécution
☐ le code provoque une fuite de mémoire

- ☐ le code ne compile pas
☐ il y a des erreurs à l'exécution

Question 10 On donne le code suivant :

```
class A{
private:
    int x;
public:
    A(int x): x{x}{}
    int get() const { return x; }
};

class B{
private:
    A a;
public:
    B(int x): a{x}{}
    int get() const { return a.get(); }
};
```

```
class C{
private:
    B *b;
public:
    C(int x): b{new B(x)}{}
    int get() const { return b->get(); }
};

int main()
{
    int x = 1;
    C c1{x};
    if (c1.get()%2 == 0) { C c2{c1}; }
    c1.get() << endl;
    return 0;
}
```

Cocher les cases qui donnent une réponse correcte :

- | | |
|--|---|
| <input checked="" type="checkbox"/> A le code ne termine pas lors de l'exécution | <input type="checkbox"/> C le code ne compile pas |
| <input type="checkbox"/> B le code provoque une fuite de mémoire | <input type="checkbox"/> D il y a des erreurs à l'exécution |

ce new n'est jamais libéré

Question 10 On donne le code suivant :

```
class A{
private:
    int x;
public:
    A(int x): x{x}{}
    int get() const { return x; }
};

class B{
private:
    A a;
public:
    B(int x): a{x}{}
    int get() const { return a.get(); }
};
```

```
class C{
private:
    B *b;
public:
    C(int x): b{new B(x)}{}
    int get() const { return b->get(); }
};

int main(){
    int x = 6;
    C c{x};

    if(c.get()%2 == 0) { C c2{c}; }
    cout << c.get() << endl;

    return 0;
}
```

Cocher les cases qui donnent une réponse correcte :

- ☒ le code ne termine pas lors de l'exécution
☐ le code provoque une fuite de mémoire

- ☐ le code ne compile pas
☐ il y a des erreurs à l'exécution

passer un peu de temps pour écarter les autres cas

Question 12

Soit le code suivant :

```
class A {  
    public :  
    void f_pub() ;  
    protected :  
    void f_prot();  
};  
  
class B : protected A {  
    public :  
    void g() ;  
};  
  
class C: public B {  
    public :  
    void h();  
};  
  
void A::f_pub() {  
    A aa; B ba; C ca;
```

```
    aa.f_prot(); // (1)  
    ba.f_prot(); // (2)  
    ca.f_prot(); // (3)  
}  
  
void A::f_prot() {};  
  
void B::g() {  
    A ab; B bb; C cb;  
    ab.f_prot(); // (4)  
    bb.f_prot(); // (5)  
    cb.f_prot(); // (6)  
};  
  
void C::h() {  
    A ac; B bc; C cc;  
    ac.f_prot(); // (7)  
    bc.f_prot(); // (8)  
    cc.f_prot(); // (9)  
}
```

Cocher les cases qui correspondent aux appels (identifiés par le numéro en commentaire) qui **ne compilent pas** :

Question 12

Soit le code suivant :

```
class A {  
    public :  
    void f_pub() ;  
    protected :  
    void f_prot();  
};  
  
class B : protected A {  
    public :  
    void g() ;  
};  
  
class C: public B {  
    public :  
    void h();  
};  
  
void A::f_pub() {  
    A aa; B ba; C ca;
```

```
    aa.f_prot(); // (1)  
    ba.f_prot(); // (2)  
    ca.f_prot(); // (3)  
}  
  
void A::f_prot() {};  
  
void B::g() {  
    A ab; B bb; C cb;  
    ab.f_prot(); // (4)  
    bb.f_prot(); // (5)  
    cb.f_prot(); // (6)  
};  
  
void C::h() {  
    A ac; B bc; C cc;  
    ac.f_prot(); // (7)  
    bc.f_prot(); // (8)  
    cc.f_prot(); // (9)  
}
```

(1) ok :
f_pub méthode de A
manipule un A.

Cocher les cases qui correspondent aux appels (identifiés par le numéro en commentaire) qui **ne compilent pas** :

Question 12

Soit le code suivant :

```
class A {
    public :
    void f_pub() ;
    protected :
    void f_prot();
};

class B : protected A {
    public :
    void g() ;
};

class C: public B {
    public :
    void h();
};

void A::f_pub() {
    A aa; B ba; C ca;
```

```
    aa.f_prot(); // (1)
    ba.f_prot(); // (2)
    ca.f_prot(); // (3)
}

void A::f_prot() {};

void B::g() {
    A ab; B bb; C cb;
    ab.f_prot(); // (4)
    bb.f_prot(); // (5)
    cb.f_prot(); // (6)
};

void C::h() {
    A ac; B bc; C cc;
    ac.f_prot(); // (7)
    bc.f_prot(); // (8)
    cc.f_prot(); // (9)
}
```

(2, 3) non :
f_pub méthode de A
manipule des non A

Cocher les cases qui correspondent aux appels (identifiés par le numéro en commentaire) qui **ne compilent pas** :

Question 12

Soit le code suivant :

```
class A {  
    public :  
    void f_pub() ;  
    protected :  
    void f_prot();  
};  
  
class B : protected A {  
    public :  
    void g() ;  
};  
  
class C: public B {  
    public :  
    void h();  
};  
  
void A::f_pub() {  
    A aa; B ba; C ca;
```

```
    aa.f_prot(); // (1)  
    ba.f_prot(); // (2)  
    ca.f_prot(); // (3)  
}  
  
void A::f_prot() {};  
  
void B::g() {  
    A ab; B bb; C cb;  
    ab.f_prot(); // (4)  
    bb.f_prot(); // (5)  
    cb.f_prot(); // (6)  
};  
  
void C::h() {  
    A ac; B bc; C cc;  
    ac.f_prot(); // (7)  
    bc.f_prot(); // (8)  
    cc.f_prot(); // (9)  
}
```

(5, 6) ok :
g méthode de B
manipule des B

Cocher les cases qui correspondent aux appels (identifiés par le numéro en commentaire) qui **ne compilent pas** :

Question 12

Soit le code suivant :

```
class A {  
    public :  
    void f_pub() ;  
    protected :  
    void f_prot();  
};  
  
class B : protected A {  
    public :  
    void g() ;  
};  
  
class C: public B {  
    public :  
    void h();  
};  
  
void A::f_pub() {  
    A aa; B ba; C ca;
```

```
    aa.f_prot(); // (1)  
    ba.f_prot(); // (2)  
    ca.f_prot(); // (3)  
}  
  
void A::f_prot() {};  
  
void B::g() {  
    A ab; B bb; C cb;  
    ab.f_prot(); // (4)  
    bb.f_prot(); // (5)  
    cb.f_prot(); // (6)  
};  
  
void C::h() {  
    A ac; B bc; C cc;  
    ac.f_prot(); // (7)  
    bc.f_prot(); // (8)  
    cc.f_prot(); // (9)  
}
```

(7,8) non :
h méthode de C
manipule des non C

Cocher les cases qui correspondent aux appels (identifiés par le numéro en commentaire) qui **ne compilent pas** :

Question 12

Soit le code suivant :

```
class A {  
    public :  
    void f_pub() ;  
    protected :  
    void f_prot();  
};  
  
class B : protected A {  
    public :  
    void g() ;  
};  
  
class C: public B {  
    public :  
    void h();  
};  
  
void A::f_pub() {  
    A aa; B ba; C ca;
```

```
    aa.f_prot(); // (1)  
    ba.f_prot(); // (2)  
    ca.f_prot(); // (3)  
}  
  
void A::f_prot() {};  
  
void B::g() {  
    A ab; B bb; C cb;  
    ab.f_prot(); // (4)  
    bb.f_prot(); // (5)  
    cb.f_prot(); // (6)  
};  
  
void C::h() {  
    A ac; B bc; C cc;  
    ac.f_prot(); // (7)  
    bc.f_prot(); // (8)  
    cc.f_prot(); // (9)  
}
```

(9) oui :
h méthode de C
manipule un C

Cocher les cases qui correspondent aux appels (identifiés par le numéro en commentaire) qui **ne compilent pas** :

Question 13

On considère le code suivant :

```
1  class D {  
2      public :  
3      ~D() {cout << "D::~~D() called";}   
4  };  
5  
6  class E : public D{  
7      protected :  
8      ~E() {cout << "E::~~E() called";}   
9  };  
10  
11 class F : public E{
```

```
12     public :  
13     ~F() {cout << "F::~~F() called";}   
14 };  
15  
16 int main() {  
17     E e;  
18     F f;  
19     D* dp =new F();  
20     delete dp;  
21     return EXIT_SUCCESS;  
22 }
```

Cocher l'affirmation correcte :

- ☒ la ligne 18 est acceptable, la ligne 17 fait échouer la compilation
- ☐ avec les lignes 17 et 18 le code compile
- ☐ la ligne 17 est acceptable mais la ligne 18 fait échouer la compilation
- ☐ la ligne 17 et la ligne 18 font échouer la compilation

Question 13

On considère le code suivant :

```
1 class D {  
2     public :  
3     ~D() {cout << "D::~~D() called";}  
4 };  
5  
6 class E : public D{  
7     protected :  
8     ~E() {cout << "E::~~E() called";}  
9 };  
10  
11 class public E{
```

```
12     public :  
13     ~F() {cout << "F::~~F() called";}  
14 };  
15  
16 int main() {  
17     E e;  
18     F f;  
19     D* dp =new F();  
20     delete dp;  
21     return EXIT_SUCCESS;  
22 }
```

Cochez l'information correcte :

- ☒ la ligne 8 est acceptable, la ligne 17 fait échouer la compilation
- ☐ avec les lignes 17 et 18 le code compile
- ☐ la ligne 8 est acceptable mais la ligne 18 fait échouer la compilation
- ☐ la ligne 18 font échouer la compilation

à cause de ce protected le destructeur n'est pas accessible partout.

Question 13

On considère le code suivant :

```
1  class D {  
2      public :  
3      ~D() {cout << "D::~~D() called";}   
4  };  
5  
6  class E : public D{  
7      protected :  
8      ~E() {cout << "E::~~E() called";}   
9  };  
10  
11 class F : public E{
```

```
12     public :  
13     ~F() {cout << "F::~~F() called";}   
14 };  
15  
16 int main() {  
17     E e;  
18     F f;  
19     D* dp =new F();  
20     delete dp;  
21     return EXIT_SUCCESS;  
22 }
```

Cocher l'affirmation correcte :

- ☒ la ligne 18 est acceptable, la ligne 17 fait échouer la compilation
- ☐ avec les lignes 17 et 18 le code compile
- ☐ la ligne 17 est acceptable mais la ligne 18 fait échouer la compilation
- ☐ la ligne 17 et la ligne 18 font échouer la compilation

à la sortie du main() e et f sont détruits.
Le destructeur de e est inaccessible :
la ligne 17 fait échouer la compilation.

Question 13

On considère le code suivant :

```
1  class D {  
2      public :  
3      ~D() {cout << "D::~~D() called";}   
4  };  
5  
6  class E : public D{  
7      protected :  
8      ~E() {cout << "E::~~E() called";}   
9  };  
10  
11 class F : public E{
```

```
12     public :  
13     ~F() {cout << "F::~~F() called";}   
14 };  
15  
16 int main() {  
17     E e;  
18     F f;  
19     D* dp =new F();  
20     delete dp;  
21     return EXIT_SUCCESS;  
22 }
```

Cocher l'affirmation correcte :

- ☒ la ligne 18 est acceptable, la ligne 17 fait échouer la compilation
- ☐ avec les lignes 17 et 18 le code compile
- ☐ la ligne 17 est acceptable mais la ligne 18 fait échouer la compilation
- ☐ la ligne 17 et la ligne 18 font échouer la compilation

la destruction de f reste possible :
Le destructeur de f se charge de faire appel à celui de sa classe mère. Avec protected il est accessible : pas de pb avec la ligne 18 seule.

On considère le code suivant :

```
class Base {  
    public:  
    virtual void show() { cout << "Base";}  
};  
  
class Derived : public Base {  
    public:  
    void show() { cout << "Derived"; }  
};  
  
class Wrapper {  
    public:  
    Wrapper(Base& b) : ref(b) {}  
    void callShow() { ref.show(); }  
    void changeObject(Base& newRef) {
```

```
        ref = newRef;  
    }  
  
    private:  
    Base& ref;  
};  
  
int main(){  
    Derived d;  
    Base b;  
    Wrapper wrapper(d);  
    wrapper.callShow();  
    wrapper.changeObject(b);  
    wrapper.callShow();  
    return EXIT_SUCCESS;  
}
```

Cocher la case qui donne l'affichage obtenu à l'exécution du code :

On considère le code suivant :

```
class Base {  
    public:  
    virtual void show() { cout << "Base";}  
};  
  
class Derived : public Base {  
    public:  
    void show() { cout << "Derived"; }  
};  
  
class Wrapper {  
    public:  
    Wrapper(Base& b) : ref(b) {}  
    void callShow() { ref.show(); }  
    void changeObject(Base& newRef) {
```

```
        ref = newRef;  
    }  
  
    private:  
    Base& ref;  
};  
  
int main(){  
    Derived d;  
    Base b;  
    Wrapper wrapper(d);  
    wrapper.callShow();  
    wrapper.changeObject(b);  
    wrapper.callShow();  
    return SUCCESS;  
}
```

Cocher la case qui donne l'affichage obtenu à l'exécution du code :

à son initialisation wrapper a stocké dans ref un alias vers d, Dérivé.
callShow invoque show, virtual : « Derived »

On considère le code suivant :

```
class Base {  
    public:  
    virtual void show() { cout << "Base";}  
};  
  
class Derived : public Base {  
    public:  
    void show() { cout << "Derived"; }  
};  
  
class Wrapper {  
    public:  
    Wrapper(Base& b) : ref(b) {}  
    void callShow() { ref.show(); }  
    void changeObject(Base& newRef) {
```

```
        ref = newRef;  
    }  
  
    private:  
    Base& ref;  
};  
  
int main(){  
    Derived d;  
    Base b;  
    Wrapper wrapper(d);  
    wrapper.callShow();  
    wrapper.changeObject(b);  
    wrapper.callShow();  
    return EXIT_SUCCESS;  
}
```

Cocher la case qui donne l'affichage obtenu à l'exécution du code :

changeObject reçoit b de la classe de Base, en référence.

On considère le code suivant :

```
class Base {  
    public:  
    virtual void show() { cout << "Base";}  
};  
  
class Derived : public Base {  
    public:  
    void show() { cout << "Derived"; }  
};  
  
class Wrapper {  
    public:  
    Wrapper(Base& b) : ref(b) {}  
    void callShow() { ref.show(); }  
    void changeObject(Base& newRef) {
```

```
        ref = newRef;  
    }  
  
    private:  
    Base& ref;  
};  
  
int main(){  
    Derived d;  
    Base b;  
    Wrapper wrapper(d);  
    wrapper.callShow();  
    wrapper.changeObject(b);  
    wrapper.callShow();  
    return EXIT_SUCCESS;  
}
```

Cocher la case qui donne l'affichage obtenu par le code :

cette affectation ne change pas
le fait que ref est un alias
vers d.

On considère le code suivant :

```
class Base {
public:
    virtual void show() { cout << "Base"; }
};

class Derived : public Base {
public:
    void show() { cout << "Derived"; }
};

class Wrapper {
public:
    Wrapper(Base& b) : ref(b) {}
    void callShow() { ref.show(); }
    void changeObject(Base& newRef) {
```

```
        ref = newRef;
    }

private:
    Base& ref;
};

int main(){
    Derived d;
    Base b;
    Wrapper wrapper(d);
    wrapper.callShow();
    wrapper.changeObject(b);
    wrapper.callShow();
    return EXIT_SUCCESS;
}
```

Cocher la case qui donne l'affichage obtenu à l'exécution du code :

à nouveau callshow :
« Derived »

On considère le code suivant :

```
class Base {
public:
    virtual void show() { cout << "Base"; }
};

class Derived : public Base {
public:
    void show() { cout << "Derived"; }
};

class Wrapper {
public:
    Wrapper(Base& b) : ref(b) {}
    void callShow() { ref.show(); }
    void changeObject(Base& newRef) {
```

```
        ref = newRef;
    }

private:
    Base& ref;

int main(){
    Derived d;
    Wrapper w(d);
    w.callShow();
    w.changeObject(b);
    w.callShow();
    return EXIT_SUCCESS;
}
```

Cocher la case qui donne l'affichage obtenu à l'exécution.

Rq : deux opérateurs= existent par défaut
celui de Derived :
Derived& : operator=(Derived &)
qui serait écarté car newRef est typé
comme Base.
Par ailleurs il ne serait pas disponible
car l'opérateur= par défaut n'est pas
virtual.

On considère le code suivant :

```
class Base {  
    public:  
    virtual void show() { cout << "Base";}  
};  
  
class Derived : public Base {  
    public:  
    void show() { cout << "Derived"; }  
};  
  
class Wrapper {  
    public:  
    Wrapper(Base& b) : ref(b) {}  
    void callShow() { ref.show(); }  
    void changeObject(Base& newRef) {
```

```
        ref = newRef;  
    }  
    private:  
    Base& ref;  
};  
  
int main(){  
    Derived d;  
    Wrapper w(d);  
    w.callShow();  
    w.changeObject(d);  
    w.callShow();  
    return EXIT_SUCCESS;  
}
```

Cocher la case qui donne l'affichage obtenu à l'exécution.

Rq : deux opérateurs existent par défaut
celui de Base:
Base& : operator=(Base &)
qui est celui qui est exécuté

On considère le code suivant :

```
class Base {  
    public:  
    virtual void show() { cout << "Base";}  
};  
  
class Derived : public Base {  
    public:  
    void show() { cout << "Derived"; }  
};  
  
class Wrapper {  
    public:  
    Wrapper(Base& b) : ref(b) {}  
    void callShow() { ref.show(); }  
    void changeObject(Base& newRef) {
```

```
        ref = newRef;  
    }  
  
    private:  
    Base& ref;  
};  
  
int main(){  
    Derived d;  
    Base b;  
    Wrapper wrapper(d);  
    wrapper.callShow();  
    wrapper.changeObject(b);  
    wrapper.callShow();  
    return EXIT_SUCCESS;  
}
```

Cocher la case qui donne l'affichage obtenu par le code :

l'affectation se fait sur les éventuels champs propres à Base (sans modifier le type de ref)