

Support de cours

Cours:

Introduction à la programmation orientée objet (en C++)

Vidéo:

W15-03-polymasq-CPP-pt2

Concepts (extraits des sous-titres générés automatiquement) :

Hiérarchie a. Portée c.. Fait toutes. Opérateur de résolution de portée. Instances b. Sous-classe c. Seule méthode. A. donc. Instance c de la classe c. Aspect substitution. Règles de substitution. Résolution dynamique des liens. Méthode. Méthodes de a. tous. Classe b.



[vers la recherche de séquences vidéo](#)

(dans Introduction à la programmation orientée objet (en C++).)



[vers la vidéo](#)

Center for Digital Education. Plus de matériel de soutien pédagogique ici :

<https://www.epfl.ch/education/educational-initiatives/cede/educational-technologies-gallery/boocs-en/>

Masquage, substitution et surcharge

(Partie 2)

Introduction à la programmation orientée objet (en C++)

Jean-Cédric Chappelier, Jamila Sam et Vincent Lepetit

...

notes

résumé

0m 0s



Masquage, substitution et surcharge : exemples

A
B

```
int main(){
    B b;
    //b.m1(2);    // NON : no matching function for call to 'B::m1(int)'
    b.A::m1(2); // ... mais elle est bien là
    b.m1("2");

    C c;    C::m1(double)
    c.m1(2); // Attention ici : c'est celle avec double !!
    //c.m1("2"); // NON : no matching function
    c.A::m1("2"); // OK
    c.A::m1(2);   // OK, et là c'est celle avec int

    return 0;
}
```

Pour l'instance C de la classe C, on ne peut pas appeler ni la méthode avec les chaînes de caractères ni la méthode avec les entiers. Attention ici un piège. Cet appel ici `c.m1(2)` n'appelle pas la méthode avec les entiers puisqu'elle est masquée. Si vous faites exécuter ce code, ce que vous allez avoir comme message

notes

résumé

0m 1s



```
class A {
public:
    virtual void m1(int i) const { cout << "A::m1(int) : " << i << endl; }
    // surcharge :
    virtual void m1(string const& s) const { cout << "A::m1(string) : " << s << endl; }
};

class B : public A {
public:
    // substitution de l'une des deux, l'autre devient hors de portée (masquage)
    virtual void m1(string const& s) const { cout << "B::m1(string)" << endl; }
};

class C : public A {
public:
    // introduction d'une 3e => masquage des 2 autres
    virtual void m1(double x) const { cout << "C::m1(double) : " << x << endl; }
};
```

à l'appel c'est : C::m1(double) puisqu'en effet c'est la seule méthode qui est appellable directement sans démasquage dans la portée C. et que un entier peut être converti en un double. Donc ici il y a une conversion d'un entier en un double pour appeler la seule méthode qui est dans C. Bien sûr comme précédemment, nous pouvons démasquer les méthodes avec chaînes de caractères et avec un entier. en utilisant l'opérateur de résolution de portée c.A::m1 avec une chaîne de caractère appellera bien sûr la méthode de A qui a une chaîne de caractère. Cet appel, ici, avec un entier cette fois-ci démasquage de la méthode m1 de la superclasse A héritée par la sous-classe C fera un appel effectivement à cette méthode, ici . Voila donc pour les aspects surcharge et masquage, Voyons maintenant l'aspect substitution. En fait toutes ces méthodes ont de fait été déclarées comme virtuelles. Il suffit simplement d'avoir déclaré, ici, les méthodes comme virtuelles pour que automatiquement elles soient virtuelles dans les sous-classes. Ici, comme on a exactement la même méthode ce virtual est transitif. Il n'est pas nécessaire de le répéter. Par contre ici c'est une nouvelle méthode. Donc on a le choix de la mettre comme virtuelle ou non. Toutes ces méthodes ayant été déclarées comme virtuelles il s'agit bien d'une substitution. Les règles de substitution suivent les mêmes règles que les règles de masquage.

notes

résumé

0m 37s



A
B

```
int main(){
    B b;
    //b.m1(2);    // NON : no matching function for call to 'B::m1(int)'
    b.A::m1(2); // ... mais elle est bien là
    b.m1("2");

    C c;    C::m1(double)
    c.m1(2); // Attention ici : c'est celle avec double !!
    //c.m1("2"); // NON : no matching function
    c.A::m1("2"); // OK
    c.A::m1(2);   // OK, et là c'est celle avec int

    return 0;
}
```

Simplement on a une résolution dynamique des liens qui va se passer, si tant est que l'on ait des pointeurs ou des références,

notes

résumé

2m 13s

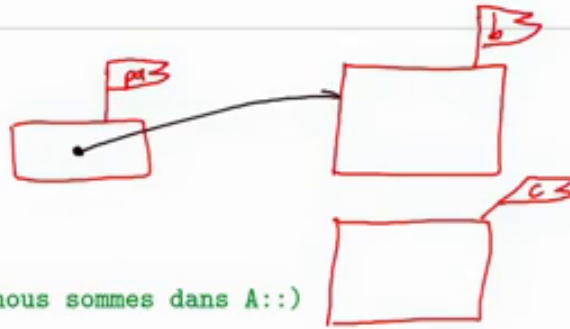


```
int main() {
    B b;
    C c;
    A* pa(nullptr);

    pa = &b;
    pa->m1("2");
    pa->m1(2);    // OK (nous sommes dans A::)

    pa = &c;
    pa->m1(2.1); /* Attention ici : c'est celle avec int !!
                  *    Nous sommes dans A::                */
    // pa->C::m1(2.1); // Impossible ! A n'hérite pas de C !!

    return 0;
}
```



ce qui n'était pas le cas ici. Ici nous avons des instances directes donc pour voir l'effet de la substitution et l'effet de la résolution dynamique des liens il faut donc passer par des pointeurs. Ce que nous faisons donc ici sur ce nouvel exemple. où nous avons toujours les instances B et C des classes, respectivement B et C, mais où nous allons donc passer sur un pointeur sur un A que l'on appelle donc pa. Dans une première partie, pa va pointer sur b et dans une seconde partie, pa pointera sur c. Ici on fait pointer pa sur b. En mémoire nous avons la situation suivante.

notes

résumé

2m 20s



```
class A {
public:
    virtual void m1(int i) const { cout << "A::m1(int) : " << i << endl; }
    // surcharge :
    virtual void m1(string const& s) const { cout << "A::m1(string) : " << s << endl; }
};

class B : public A {
public:
    // substitution de l'une des deux, l'autre devient hors de portée (masquage)
    virtual void m1(string const& s) const { cout << "B::m1(string)" << endl; }
};

class C : public A {
public:
    // introduction d'une 3e => masquage des 2 autres
    virtual void m1(double x) const { cout << "C::m1(double) : " << x << endl; }
};
```

pa pointe ici sur b et l'on appelle donc la méthode m1 au travers de ce pointeur pa.

notes

résumé

3m 1s

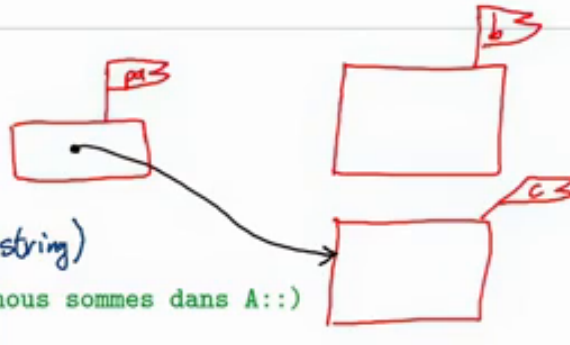


```
int main() {
    B b;
    C c;
    A* pa(nullptr);
```

```
    pa = &b;
    pa->m1("2");
    pa->m1(2);

    pa = &c;
    pa->m1(2.1); /* Attention ici : c'est celle avec int !!
                  * Nous sommes dans A:: */
    // pa->C::m1(2.1); // Impossible ! A n'hérite pas de C !!

    return 0;
}
```



Or il se trouve que m1 est une méthode virtuelle qui est substituée dans la classe B. Ici c'est bien le polymorphisme qui entre en jeu. C'est donc bien la méthode m1 de la classe B qui est appelée et qui affichera donc le message B::m1(string) Par contre même si pa pointe sur un B, il n'en reste pas moins un pointeur sur A. Donc en tant que tel, je vous rappelle qu'en C++ c'est le type qui prime lorsqu'il n'y a pas de polymorphisme on a une résolution statique des liens. On résout en fonction du type. Donc ici pa est un pointeur sur un A et donc en tant que tel accède aux méthodes de A. Tous ces appels à ses méthodes sont de la classe A. On dit que ce sont des appels dans la portée de A et donc on a tout à fait le droit même si pa pointe sur un B, d'appeler la méthode m1 avec un entier. On est bien dans la portée de A. Donc ici, ce sera bien la méthode de A qui va être appelée A::m1(int) De même si pa pointe sur C, alors bien que l'on pointe sur C on est toujours dans la portée de A puisqu'on travaille avec un pointeur sur un A.

notes

résumé

3m 9s




```
class A {
public:
    virtual void m1(int i) const { cout << "A::m1(int) : " << i << endl; }
    // surcharge :
    virtual void m1(string const& s) const { cout << "A::m1(string) : " << s << endl; }
};

class B : public A {
public:
    // substitution de l'une des deux, l'autre devient hors de portée (masquage)
    virtual void m1(string const& s) const { cout << "B::m1(string)" << endl; }
};

class C : public A {
public:
    // introduction d'une 3e => masquage des 2 autres
    virtual void m1(double x) const { cout << "C::m1(double) : " << x << endl; }
};
```

Ici, on va appeler la méthode. Attention, même si on passe un double, on reste dans la portée de A et on va donc appeler ici la méthode qui passe un entier. Ce double va être converti en entier, pour appeler, dans la portée de A,

notes

résumé

4m 25s



```
int main() {
    B b;
    C c;
    A* pa(nullptr);

    pa = &b;
    pa->m1("2");
    pa->m1(2);

    pa = &c;
    pa->m1(2.1); /* Attention ici : c'est celle avec int !!
    // pa->C::m1(2.1); // Impossible ! A n'hérite pas de C !!

    return 0;
}
```

Handwritten notes and diagram:

- Diagram showing a pointer `pa` pointing to a box representing object `b` (type `B`).
- Another box representing object `c` (type `C`) is shown below it.
- Handwritten notes: `B::m1(string)`, `A::m1(int)`, `// OK (nous sommes dans A::)`.
- Handwritten note: `A::*` `* Nous sommes dans A:: *`.

la seule méthode que l'on peut appeler avec un double. C'est celle-ci où l'on convertit le double en entier. Ce qui est tout à fait possible en C++ et c'est bien cette méthode qui va être appelée. la méthode qui est dans C n'est pas accessible dans la portée de A Si un C est un A, un A lui n'est pas du tout un C. Ici, puisqu'on est dans la portée de A c'est bien dans cette portée-là qu'on va résoudre le nom `m1`. C'est donc bien `m1` avec un `int` qui va être appelée et donc ici on aura bien un affichage de `A::m1` pour un `int`. Pour les mêmes raisons on ne peut absolument pas essayer de démasquer la méthode avec un double en essayant d'utiliser l'opérateur de résolution de portée ici sur C. Ceci est totalement illicite puisqu'on passe par un pointeur sur un A et que un A n'est pas un C, C'est un C qui est un A. Mais un A n'est pas un C. Dans la hiérarchie A est en haut de la hiérarchie. C'est C qui est un A mais absolument pas A qui est un C. Donc ceci ne compile pas. C'est interdit par le compilateur. Ce n'est pas A qui hérite de C, mais C qui hérite de A. mais C qui hérite de A.

notes

résumé

4m 41s

