

Pourquoi C++

Programmation orientée objet

Prof. John Iacono

Traditionnel vs orienté objet

- Traditionnel
- Les données et les fonctions sont séparées
- Les fonctions peuvent fonctionner avec un type de données
- Les types intégrés sont spéciaux. par exemple. + fonctionne avec des entiers.
- Les fonctions exécutent le même code
- Aucune restriction d'accès. N'importe quelle fonction peut être exécutée n'importe où.
- Orienté objet
- Les données et les fonctions sont ensemble dans une classe. (méthode=fonction dans une classe)
- Les fonctions peuvent fonctionner avec de nombreux types de données
- Peut faire fonctionner + et d'autres opérateurs avec vos types
- La même méthode peut exécuter un code différent en fonction de l'objet sous-jacent
- Restrictions d'accès très développées

Pourquoi C++

- C a été conçu pour être rapide, avec des opérations qui suivaient de près ce qui pouvait être fait dans les pouces.
- C++ ajoute l'orientation objet au C, mais d'une manière qui ne sacrifie pas la vitesse.
- S'il y a deux façons de faire quelque chose, et que les deux ont des avantages et des inconvénients, le C++ fait souvent les choses de deux manières différentes
- Cela pourrait être appelé riche ou fou, selon votre point de vue
- Après avoir appris le C++, d'autres langages seront faciles à apprendre

Les idées directrices du C++

- Les programmeurs devraient être libres de choisir leur propre style de programmation
- Autoriser une fonctionnalité utile est plus important que d'empêcher toute utilisation abusive possible de C++.
- Aucune violation implicite du type system (mais autorise les violations explicites, c'est-à-dire celles explicitement demandées par le programmeur).
- Les types créés par l'utilisateur doivent avoir la même prise en charge et les mêmes performances que les types intégrés.
- Il ne devrait pas y avoir de langage sous C++ (à l'exception du langage assembleur).

C++ est fortement typé

- Chaque objet a un type
- En tant que programmeur, vous devez être conscient du type de tout et de la relation des types les uns avec les autres
- Le compilateur vérifie la compatibilité des types
- Pas de tyage de canard

fred.quack()

- Le programmeur peut forcer les conversions
- Python et Javascript n'ont pas de types, Typescript est le langage Javascript avec typé ajouté pour développer plus facilement de gros projets.



Il y a plusieurs façons de faire les choses en C++

- Certains sont dus à l'histoire. C, C++, C++17
- Certains sont dus à la vitesse et à l'espace. Il ne devrait pas y avoir de langage plus rapide que le C++.
- Vous pouvez choisir de coder de manière sûre ou sans restriction
- C++ est très riche
 - Prise en charge de l'héritage multiple
 - Pointeurs, références, rvalues, expressions lambda,

Exemple 1 : Vector

```
vector<int> v(10,0);  
vector<int> v2(10,0);  
v[20]=5;
```

```
for (auto i:v) cout<<i<<" "; cout<<endl;  
for (auto i:v2) cout<<i<<" "; cout<<endl;
```

v et v2 ont chacun 10 zéros

Attendez.... la v1 a une longueur de 10 !!!

Plus rapide!

Plus sûr !

0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	5	0

J'ai corrompu la v2[8] !

C'est votre choix !

```
vector<int> v(10,0);  
vector<int> v2(10,0);  
v.at(20)=5  
for (auto i:v) cout<<i<<" "; cout<<endl;  
for (auto i:v2) cout<<i<<" "; cout<<endl;
```

Maintenant, j'obtiens une erreur et le programme s'arrête plutôt que de corrompre les données

libc++abi: terminating with uncaught exception of type std::out_of_range: vector

Exemple 2 : Virtuel

Virtual indique que la méthode sera appelée en fonction du type de l'objet au lieu du type de la variable.

Il est également un peu plus lent et prend plus de place.

```
struct animal{  
    void whatAreYou(){cout<<"I am a animal"\  
    virtual void makeNoise(){cout<<"Generic animal noise"\  
};  
  
struct duck:animal {  
    void whatAreYou(){cout<<"I am a duck"\  
    virtual void makeNoise() {cout<<"Quack"\  
};
```

Le canard hérite de l'animal. Par conséquent, tous les canards sont des animaux.

```
animal *a=new animal();  
duck   *d=new duck();  
animal *b=d;  
  
cout<<"a pointer to an animal that points to an animal:"\  
a->whatAreYou();      a pointer to an animal that points to an animal:  
                        I am a animal  
a->makeNoise();        Generic animal noise  
  
cout<<endl<<"a pointer to an duck that points to an duck:"\  
d->whatAreYou();      a pointer to an duck that points to an duck:  
                        I am a duck  
d->makeNoise();        Quack  
  
cout<<endl<<"a pointer to an animal that points to an duck:"\  
b->whatAreYou();      a pointer to an animal that points to an duck:  
                        I am a animal  
b->makeNoise();        Quack
```

Un pointeur de type animal doit-il se comporter comme un canard ou un animal générique ?

Virtuel ou pas ?
C'est ton choix!

Où C++ est utilisé

- Grands projets
- Endroits où la vitesse et l'utilisation de l'espace sont primordiales. Un programme C++ peut facilement être 100 fois plus rapide et plus petit que le même programme en Python.
- Appareils embarqués
- Systèmes d'exploitation
- Endroits où vous devez accéder au matériel (par exemple, programmation GPU)
- Non utilisé pour :
 - Développement web front-end (mais est utilisé pour programmer des programmes back-end tels que des bases de données)
 - Petits programmes où vous exécutez les choses rapidement
 - Développement interactif. C++ est compilé et ne fonctionne pas bien dans les environnements comme les notebooks Jupyter.

Encapsulation

Programmation orientée objet

Prof. John Iacono

Introduction

- Faire fonctionner C++ avec des fractions
 - Stockez les fractions toujours dans les termes les plus bas (jamais 9/12, 3/4 à la place)
 - Ajouter une fraction à une autre
 - Copier une fraction dans une autre
 - Convertir une fraction en chaîne pour que nous puissions l'imprimer
- Visite historique des techniques pour ce faire
- En C++, il y a plusieurs façons de faire la même chose. Pour comprendre pourquoi cela aide à comprendre l'histoire.
- Cela servira également de (ré)-introduction aux concepts d'encapsulation et d'abstraction

Version 1 : Chaque fraction est deux nombres entiers

Aucune struct, classe ou référence. Les pointeurs sont ok.

Tout d'abord, écrivons les en-têtes de fonction.
Nous les mettrons en œuvre après

```
void fractionCreate(int *n, int *d, int num, int den);
void fractionAddTo(int *n1, int *d1, int n2, int d2);
void fractionCopy(int *n1, int *d1, int n2, int d2);
void fractionToString(int n, int d, char *s, int stringMaxLen);
```

Un pointeur vers le premier caractère de la chaîne

La longueur de la chaîne

Pas de références en C, donc on passe un pointeur sur un entier quand on veut changer

Comment nous voulons que cela fonctionne

Nous utilisons & pour obtenir l'adresse de l'entier et le transformer en un pointeur

```
int fn,fd;
char s[10];
fractionCreate(&fn,&fd,9,12);
fractionToString(fn,fd,s,10);
printf ("%s\n",s);
```

3 / 4

9/12 a été correctement réduit à 3/4

Méthode d'Euclide

Première version : Le code

```
void fractionCreate(int *n,int *d, int num, int den){  
    *n=num;  
    *d=den;  
    reduce(n,d);  
}
```

Besoin de réduire aux termes les plus bas à deux endroits, codons-nous une fonction distincte

```
void fractionAddTo(int *n1, int *d1,int n2, int d2){  
    *n1 = *n1 * d2 + n2 * *d1;  
    *d1 *= d2;  
    reduce(n1,d1);  
}
```

```
int gcd(int x, int y){  
    if (x==0)  
        return y;  
    else return gcd(y%x,x);  
}
```

```
void reduce(int *n, int *d){  
    int g=gcd(*n,*d);  
    *n/=g;  
    *d/=g;  
    if (*d<0){  
        *n*=-1;  
        *d*=-1;  
    }  
}
```

```
void fractionCopy(int *n1, int *d1,int n2, int d2){  
    *n1=n2;  
    *d1=d2;  
}
```

Termes les plus bas, dénominateur positif

```
void fractionToString(int n, int d, char *s,int stringMaxLen){  
    sprintf(s,stringMaxLen,"%i/%i",n,d);  
}
```

snprintf est la fonction pour « imprimer » dans une chaîne en C. Elle utilise une chaîne de formatage avec de nombreuses options %i indique que n (et d) sont des entiers

Première version : La démo

```
int f1n,f2n,f1d,f2d;
char s1[20],s2[20];
fractionCreate(&f1n,&f1d,1,1);
for (int i=2;i<=10;++i){
    fractionToString(f1n,f1d,s1,20);
    fractionCreate(&f2n,&f2d,1,i);
    fractionToString(f2n,f2d,s2,20);
    printf("%s+%s=",s1,s2);
    fractionAddTo(&f1n,&f1d,f2n,f2d);
    fractionToString(f1n,f1d,s1,20);
    printf("%s\n",s1);
```

```
1/1+1/2=3/2
3/2+1/3=11/6
11/6+1/4=25/12
25/12+1/5=137/60
137/60+1/6=49/20
49/20+1/7=363/140
363/140+1/8=761/280
761/280+1/9=7129/2520
7129/2520+1/10=7381/2520
```

Qu'est-ce qui pourrait être amélioré ?

- Maintenir ce qui est logiquement une fraction sur deux entiers est ennuyeux et sujet aux erreurs
- Le code n'est pas très lisible
- Devoir se rappeler quand utiliser le &
- Voici le code pour faire la même chose avec les flotteurs. Pourquoi notre code pour les fractions est-il meilleur ?

```
float f1;
for (int i=2;i<=10;++i){
    float f2=1.0/i;
    cout<<f1<<"+"<<f2;
    f1+=f2;
    cout<<f1<<endl;
```

Deuxième version : struct

- `struct fraction {
 int n;
 int d;
};`
- Cela définit un nouveau type, fraction, qui est la combinaison de deux entiers appelés n et d
- Pour déclarer une fraction, faites-le comme n'importe quel type :

`fraction f;`

- Pour accéder à n et f, utilisez . sur toute variable de type fraction :

`f.n=10; f.d=9;`

Deuxième version : C utilisant la struct (on est encore en 1972)

```
void fractionCreate(int *n, int *d, int num, int den);
void fractionAddTo(int *n1, int *d1, int n2, int d2);
void fractionCopy(int *n1, int *d1, int n2, int d2);
void fractionToString(int n, int d, char *s, int stringMaxLen);
```



Utilisez le fraction struct

```
fraction fractionCreate(int num, int den);
void fractionAddTo(fraction *f1, fraction f2);
void fractionCopy(fraction *f1, fraction f2);
void fractionToString(fraction f, char *s, int stringMaxLen);
```

Maintenant qu'une fraction est un objet unique, nous pouvons la renvoyer ; avant, nous ne pouvions pas retourner à la fois n et d, nous devions donc leur passer des pointeurs

Un peu plus sympa. Il n'y a qu'un f pour la fraction, plus de fn et fd

```
int fn,fd;
char s[10];
fractionCreate(&fn,&fd,9,12);
fractionToString(fn,fd,s,10);
printf("%s\n",s);
```



```
char s[10];
fraction f=fractionCreate(9,12);
fractionToString(f,s,10);
printf("%s\n",s);
```

Deuxième version : La démo

```
int f1n,f2n,f1d,f2d;
char s1[20],s2[20];
fractionCreate(&f1n,&f1d,1,1);
for (int i=2;i<=10;++i){
    fractionToString(f1n,f1d,s1,20);
    fractionCreate(&f2n,&f2d,1,i);
    fractionToString(f2n,f2d,s2,20);
    printf("%s+%s=",s1,s2);
    fractionAddTo(&f1n,&f1d,f2n,f2d);
    fractionToString(f1n,f1d,s1,20);
    printf("%s\n",s1);
```



```
fraction f1,f2;
char s1[20],s2[20];
f1=fractionCreate(1,1);
for (int i=2;i<=10;++i){
    fractionToString(f1,s1,20);
    f2=fractionCreate(1,i);
    fractionToString(f2,s2,20);
    printf("%s+%s=",s1,s2);
    fractionAddTo(&f1,f2);
    fractionToString(f1,s1,20);
    printf("%s\n",s1);
```

```
1/1+1/2=3/2
3/2+1/3=11/6
11/6+1/4=25/12
25/12+1/5=137/60
137/60+1/6=49/20
49/20+1/7=363/140
363/140+1/8=761/280
761/280+1/9=7129/2520
7129/2520+1/10=7381/2520 }
```

Qu'est-ce qui pourrait être amélioré ?

- Maintenir ce qui est logiquement une fraction sur deux entiers est ennuyeux et sujet aux erreurs
- Le code n'est pas très lisible
- Devoir se rappeler quand utiliser le &
- Voici le code pour faire la même chose avec les flotteurs. Pourquoi notre code pour les fractions est-il meilleur ?

Prochaine étape : modifiez le code afin que nous n'ayons pas besoin d'un & ici

```
float f1;
for (int i=2;i<=10;++i){
    float f2=1.0/i;
    cout<<f1<<"+"<<f2;
    f1+=f2;
    cout<<f1<<endl;
```

Troisième version : chaînes et références C++

- Une référence est comme un pointeur déréférencé en permanence, qui existe en C++ mais pas en C :

Pointeurs

```
void addoneA(int *i) {*i += 1;}
```

```
int j=0;  
addoneA(&j);
```

Je pourrais écrire `i+=1`, ce qui ferait avancer le pointeur d'un (et ne ferait rien). Des erreurs comme celle-ci sont difficiles à trouver et à déboguer !

Références

```
void addoneB(int &i) {i += 1;}
```

```
int j=0;  
addoneB(j);
```

C'est plus facile à suivre.

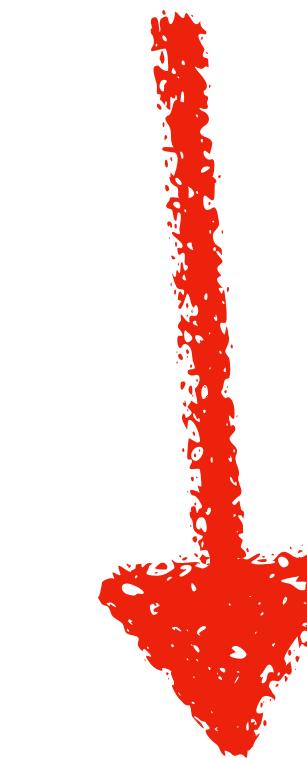
On ne peut pas changer ce à quoi la référence fait référence, ce qui limite ce que vous pouvez faire et limite les erreurs que vous pouvez faire

Troisième version : avec des références au lieu de pointeurs

```
fraction fractionCreate(int num, int den);
void fractionAddTo(fraction *f1, fraction f2);
void fractionToString(fraction f, char *s, int stringMaxLen);
```

```
void reduce(fraction *f)
```

Utiliser des références



```
void reduce(fraction &f);
```

```
fraction fractionCreate(int n,int d);
fraction &fractionAddTo(fraction &f1, fraction f2);
void fractionToString(fraction f, char *s, int stringMaxLen);
```

```
fraction f1=fractionCreate(1,3);
fraction f2=fractionCreate(1,4);
fraction f3=fractionCreate(1,5);
char s[10];
fractionToString(fractionAddTo(fractionAddTo(f1,f2),f3),s,10);
printf("1/3+1/4+1/5=%s\n",s);
```

En renvoyant une référence à f1,
plusieurs instructions peuvent être
enchaînées

Troisième version : le code qui change

```
void reduce(fraction *f){  
    int g=gcd(f->n,f->d);  
    f->n/=g;  
    f->d/=g;  
    if (f->d<0){  
        f->n*=-1;  
        f->d*=-1;  
    }  
}
```

```
void fractionAddTo(fraction *f1,fraction f2){  
    f1->n = f1->n * f2.d + f2.n * f1->d;  
    f1->d *= f2.d;  
    reduce(f1);
```

Reduce prend une fraction* et f1 est une fraction*

```
void reduce(fraction &f){  
    int g=gcd(f.n,f.d);  
    f.n/=g;  
    f.d/=g;  
    if (f.d<0){  
        f.n*=-1;  
        f.d*=-1;  
    }  
}
```

```
fraction &fractionAddTo(fraction &f1, fraction f2)  
{  
    f1.n = f1.n * f2.d + f2.n * f1.d;  
    f1.d *= f2.d;  
    reduce(f1);  
    return f1;  
}
```

Reduce prend une fraction& et f1 est une fraction&

Mettez les fonctions dans la Fraction !

Vers les objets

Définition de notre type de fraction

```
struct fraction {  
    int n;  
    int d;  
};
```

Définition des fonctions utilisant le type fraction

```
fraction fractionCreate(int n,int d);  
void reduce(fraction &f);  
fraction &fractionAddTo(fraction &f1,fraction f2);  
void fractionToString(fraction f, char *s,int stringMaxLe
```

Une méthode portant le même nom que la structure/classe est appelée un constructeur et est appelée chaque fois qu'une fraction est créée. Il remplace FractionCreate

```
struct Fraction{  
    int n;  
    int d;  
    Fraction(int n,int d);  
    void reduce();  
    string toString();  
    void addTo(Fraction f2);  
};
```

They are now called methods

Ces trois méthodes ont vu leur premier paramètre supprimé, qui était une fraction. Ceci est maintenant représenté par la fraction qu'ils sont à l'intérieur

Renvoie une nouvelle fraction

Ces trois éléments prennent tous une fraction comme premier paramètre. C'est la fraction sur laquelle ils agissent

```
struct fraction {  
    int n;  
    int d;  
};
```

Ancienne
façon

```
fraction fractionCreate(int n,int d);  
void reduce(fraction &f);  
fraction &fractionAddTo(fraction &f1,fraction f2);  
void fractionToString(fraction f, char *s,int stringMaxLen);
```

```
fraction f1=fractionCreate(1,3);  
fraction f2=fractionCreate(1,4)  
fractionAddTo(f1,f2);
```

```
fraction &fractionAddTo(fraction &f1, fraction f2){  
    f1.n =  
        f1.n * f2.d + f2.n * f1.d;  
    f1.d *= f2.d;  
    reduce(f1);  
    return f1;  
}
```

Change f1 à
*this

AddTo

Nouvelle façon

```
struct Fraction{  
    int n;  
    int d;  
    Fraction(int n,int d);  
    void reduce();  
    string toString();  
    void addTo(Fraction f2);  
};
```

```
fraction f1(1,3);  
fraction f2(1,4)  
f1.addTo(f2);
```

this indique que
addTo est une
méthode de
Fraction

```
void Fraction::addTo(Fraction f2){  
    this->n =  
        this->n * f2.d + f2.n * this->d;  
    this->d *= f2.d;  
    this->reduce();  
    return *this  
}
```

Comme reduce est maintenant une
méthode, nous l'appelons sur une
fraction, l'ancienne f1, qui est *this

```
struct fraction {  
    int n;  
    int d;  
};
```

Ancienne façon

FractionCreate/Constructor

```
fraction fractionCreate(int n,int d);  
void reduce(fraction &f);  
fraction &fractionAddTo(fraction &f1,fraction f2);  
void fractionToString(fraction f, char *s,int stringMaxLen);
```

```
fraction f1=fractionCreate(1,3);  
fraction f2=fractionCreate(1,4);  
fractionAddTo(f1,f2);
```

```
fraction fractionCreate(int n,int d){  
    fraction f;  
    f.n=n;  
    f.d=d;  
    reduce(f);  
    return f;  
}
```

Une fraction devait être créée et retournée

fractionCreate renvoie une fraction

Nouvelle façon

```
struct Fraction{  
    int n;  
    int d;  
    Fraction(int n,int d);  
    void reduce();  
    string toString();  
    void addTo(Fraction f2);  
};
```

```
fraction f1(1,3);  
fraction f2(1,4)  
f1.addTo(f2);
```

Le constructeur est appelé dans le cadre de la création d'une variable de type fraction

```
Fraction::Fraction(int n,int d){  
    this->n=n;  
    this->d=d;  
    this->reduce();  
}
```

La fraction créée/retournée est *this

Des détails

this-> peut être omis pour accéder aux variables d'instance et aux méthodes

```
Fraction &Fraction::addTo(Fraction f2){  
    this->n =  
        this->n * f2.d + f2.n * this->d;  
    this->d *= f2.d;  
    this->reduce();  
    return *this  
}
```

supprimez
this->

```
Fraction &Fraction::addTo(Fraction f2){  
    n =  
        n * f2.d + f2.n * d;  
    d *= f2.d;  
    reduce();  
    return *this  
}
```

ici, nous avons encore besoin de
this

Access control

- Maintenant que nous avons combiné les données et les fonctions en un seul objet avec des variables d'instance et des méthodes, nous pouvons décider que certaines variables d'instance et méthodes ne doivent pas être accédées depuis l'extérieur de la classe.
- Cela peut être fait en utilisant private: et public:
- Il est considéré comme une bonne pratique de rendre les données privées dans la plupart des cas et de créer des méthodes pour accéder/modifier les données.
- struct et class sont identiques, les deux définissent des objets. La seule différence est que dans une structure, la valeur par défaut est public, tandis que dans une classe, la valeur par défaut est privée.
- Nous utilisons généralement des structs lorsque tout devrait être public, souvent s'il n'y a pas de méthodes, et class lorsqu'il y a un mélange de public et de privé

```
struct Fraction{  
    int n;  
    int d;  
  
    Fraction(int n,int d);  
    void reduce();  
    string toString();  
    void addTo(Fraction f2);  
};
```

```
class Fraction{  
    int n;  
    int d;  
  
    void reduce();  
  
public:  
    Fraction(int n,int d);  
    string toString();  
    void addTo(Fraction f2);  
};
```

Reduce est uniquement pour un usage interne

Pourquoi le contrôle d'accès est important

```
struct Fraction{  
    int n;  
    int d;  
    Fraction(int n,int d);  
    void reduce();  
    string toString();  
    void addTo(Fraction f2);  
};
```

```
class Fraction{  
    int n;  
    int d;  
    void reduce();  
public:  
    Fraction(int n,int d);  
    string toString();  
    void addTo(Fraction f2);  
};
```

On essaie de changer le numérateur

```
fraction f=fractionCreate(7,18);  
f.n=6;  
cout<<fractionToString(f)<<endl;
```

6/18

Pas en termes les plus bas !!!!

```
fractions.cpp:318:19: error: 'int  
attempt4::Fraction::n' is private within this  
context  
318 | f.n=6;
```

C'est une erreur ! Ce qui est bon. Cela nous empêche de gâcher la fraction

Setters et getters

```
struct Fraction{  
    int n;  
    int d;  
    Fraction(int n,int d);  
    void reduce();  
    string toString();  
    void addTo(Fraction f2);  
};
```

```
fraction f=fractionCreate(7,18);  
f.n=6;  
cout<<fractionToString(f)<<endl;
```

6/18

Pas en termes les plus bas !!!!

```
class Fraction{  
    int n;  
    int d;  
    void reduce();  
public:  
    Fraction(int n,int d);  
    string toString();  
    Fraction &addTo(Fraction f2);  
    int getNum(){return n;}  
    int getDen(){return d;}  
    void setNum(int num){n=num;reduce();}  
    void setDen(int den){d=den;reduce();}  
};
```

Fraction f(7,18);

f.setNum(6);

```
cout<<f.toString()<<endl;
```

1/3

Le corps d'une méthode peut être séparé (comme avant) ou à l'intérieur de la déclaration de classe

Nous nous assurons que la réduction est appelée

Parfait!

Les opérateurs

```
float f1(0.2);  
float f2(0.3);  
f2+=f1;  
cout<<f2<<endl;
```

0.5

```
Fraction f1(2,10);  
Fraction f2(3,10);  
f2.addTo(f1);  
cout<<f2.toString()<<endl;
```

2

Pouvons-nous faire fonctionner += avec nos fractions ?

```
class Fraction{  
    . . .  
    Fraction &addTo(Fraction f2); // implemented later  
    Fraction &operator +=(Fraction f2){return addTo(f2);} };
```

Un opérateur est juste une fonction avec un nom spécial

```
Fraction f1(2,10);  
Fraction f2(3,10);  
f2+=f1;  
cout<<f2.toString()<<endl;
```

1/2

Nous sommes venus de loin

Nous pouvons maintenant utiliser les fractions presque de la même manière que le type double

Première version

```
int f1n,f2n,f1d,f2d;
char s1[20],s2[20];
fractionCreate(&f1n,&f1d,1,1);
for (int i=2;i<=10;++i){
    fractionToString(f1n,f1d,s1,20);
    fractionCreate(&f2n,&f2d,1,i);
    fractionToString(f2n,f2d,s2,20);
    printf("%s+%s=",s1,s2);
    fractionAddTo(&f1n,&f1d,f2n,f2d);
    fractionToString(f1n,f1d,s1,20);
    printf("%s\n",s1);
```

```
1/1+1/2=3/2
3/2+1/3=11/6
11/6+1/4=25/12
25/12+1/5=137/60
137/60+1/6=49/20
49/20+1/7=363/140
363/140+1/8=761/280
761/280+1/9=7129/2520
7129/2520+1/10=7381/2520
```

double

```
float f1{0};
for (int i=2;i<=10;++i){
    float f2{1.0/i};
    cout<<f1<<"+"<<f2;
    f1+=f2;
    cout<<f1<<endl;
}
```

```
0+0.50.5
0.5+0.3333330.833333
0.833333+0.251.08333
1.08333+0.21.28333
1.28333+0.1666671.45
1.45+0.1428571.59286
1.59286+0.1251.71786
1.71786+0.1111111.82897
1.82897+0.11.92897
```

Dernière version

```
Fraction f1{0};
for (int i=2;i<=10;++i){
    Fraction f2{1,i};
    cout<<f1.toString()<<"+"<<f2.toString();
    f1+=f2;
    cout<<f1.toString()<<endl;
}
```

```
1/1+1/2=3/2
3/2+1/3=11/6
11/6+1/4=25/12
25/12+1/5=137/60
137/60+1/6=49/20
49/20+1/7=363/140
363/140+1/8=761/280
761/280+1/9=7129/2520
7129/2520+1/10=7381/2520
```

Nous pourrions supprimer le besoin d'appeler `toString` en surchargeant l'opérateur `<<`

Summary

- À première vue, la programmation orientée objet n'est qu'un moyen de rassembler des données et des fonctions. C'est ce qu'on appelle l'encapsulation
- C'est aussi un moyen de définir de nouveaux types et de leur donner la possibilité d'agir comme des types intégrés. Cela inclut la possibilité de faire travailler les opérateurs sur eux
- Nous pouvons également, via public et privé, masquer certaines méthodes et données d'accès en dehors de la classe. C'est ce qu'on appelle l'abstraction. Cette séparation des parties privées et publiques d'une classe est extrêmement importante à mesure qu'un projet prend de l'ampleur.
- Certaines personnes considèrent les méthodes d'appel d'un objet comme un moyen de transmettre des messages entre les objets. Des langages tels que smalltalk accentuent ce point de vue.
- Nous n'avons pas discuté ici de l'héritage, grâce auquel nous pouvons créer de nouvelles classes à partir des classes existantes. Nous allons d'abord nous assurer que vous comprenez tout sur la programmation orientée objet sans héritage.

Durée de vie des objets I : variables automatiques

Programmation orientée objet

Prof. John Iacono

Plan de l'exposé

- Nous allons discuter quelques sujets à propos des variables. Les variables sont où vos données sont stockées et il est important de maîtriser les détails de les différents types de variables. On commence avec le cas le plus simple:
 - Pas d'allocation dynamique. Pas de `new`, `malloc` ou pointeurs intelligentes.
 - Pas de références
 - Pas d'inheritance
 - Pas de templates
- Avec ces hypothèses, chaque variable est associée à une seule instance/objet
- Ces variables qui créent un objet lorsqu'elles entrent dans la portée et le suppriment lorsqu'elles sortent de la portée sont appelées variables automatiques
- Les variables automatiques sont le moyen le plus courant de créer et de détruire des objets
- Il y aura des autres leçons pour chaque de ces cas plus compliqués dans en temps voulu.

Durées de vie

- Il existe quatre durées de vie possibles pour les objets:
- **Global:** La variable existe pour la durée de vie du programme. Il n'y a qu'une seule variable pour chaque déclaration. Il est créé au début du programme et détruit à la fin du programme. Si le programme se termine anormalement, il peut ne pas être correctement détruit.
- **Local:** La variable existe pour la durée de vie du bloc dans lequel elle se trouve. Ce bloc est une fonction / méthode ou un bloc interne défini avec des accolades: { }. Il est créé lorsque l'exécution atteint l'endroit où il est déclaré et détruit lorsque l'exécution atteint la fin du bloc. Les paramètres des fonctions et des méthodes sont aussi des variables locales.
- **Instance:** Une variable d'instance est définie dedans une classe. Il est associé à une instance particulière d'une variable de sa classe et a la même durée de vie que cette instance.
- **Temporaire :** créé lorsqu'il est renvoyé par une fonction utilisant return-by-value ou par un appel direct à un constructeur. Détruit à la fin de l'expression.
- **Dynamique:** créer et détruire manuellement des instances (une future leçon)

Une classe pour nous aider à voir les lifteimes

- Chaque instance stocke une chaîne.
- **Created:** et la chaîne est imprimé lorsqu'une instance de **C** est créée avec une chaine
- **Created via copy:** et la chaîne est imprimé lorsqu'une instance de **C** est créée à avec une objet de type **C**
- **destroyed:** et la chaîne est imprimé lorsque l'instance atteint la fin de sa durée de vie.
- Cela se fait en utilisant des constructeurs et des destructeurs, qui auront leur propre leçons

```
struct C {  
    string st;  
    C(string s): st(s) {cout<<" Created: "<<st<<endl;}  
    C(C &c):st{c.st} {cout<<" Created via copy: "<<st<<endl;}  
    ~C() {cout<<" Destroyed: "<<st<<endl;}  
};
```

Constructeur

Constructeur de copie

Destructeur, appelé lorsque l'instance de C atteint la fin de sa vie

Variables locales

- Les variables locales normales commencent leur vie à leur déclaration et finissent leur vie à la fin de leur bloc.
- Les paramètres de fonction/méthode utilisant le passage par valeur commencent leur vie à l'appel de la fonction via une copie et se terminent lorsque la fonction/méthode revient
- Avec la récursivité, il peut y avoir plusieurs instances associées à une seule variable

Variables

The diagram illustrates the concept of value copying. On the left, the word "Variables" is written in large, bold black letters. In the center, there is a blue rectangular box containing the text "Le passage par valeur copie lv1 dans x". To the right of the box, the word "es : exemple" is written in large, bold black letters.

```
void doubleString(C x){  
    cout<<"start doubleString"<<endl;  
    C y{x.st+x.st};  
    cout<<"end doubleString"<<endl;  
}
```

```
void demoLocal() {
    cout<<"start demoLocal"<<endl;
    C lv1{"lv1"};
    for (int i=0;i<5;i++) {C lv2{"lv2"};
        cout<<"loopover"<<endl;
        doubleString(lv1);
        cout<<"end demoLocal"<<endl;
}
```

```
int main() {
    cout<<"Start Main"<<endl;
    demoLocal();
    cout<<"End Main"<<endl;
}
```

Le passage par valeur copie lv1 dans x

y est une variable locale initialisée avec lv1lv1, et est détruite lorsque doubleString se termine

lv1 est créé

```
Start Main
start demoLocal
    Created: lv1
    Created: lv2
    Destroyed: lv2
    Created: lv2
    Destroyed: lv2
    Created: lv2
    Destroyed: lv2
    Created: lv2
    Destroyed: lv2
    Created: lv2
    Destroyed: lv2
loopover
    Created via co
start doubleString
    Created: lv1lv
end doubleString
    Destroyed: lv1
```

2 créé et détruit à
chaque exécution de
la boucle

x est créé avec une copie avant que le corps de doubleString s'exécute

y est une variable locale initialisée avec lv1 lv1

y est détruit lorsque
doubleString se termine

Instance variables

- Les variables d'instance sont déclarées à l'intérieur d'une classe, et il y a une instance pour chaque instance d'une classe
- La durée de vie des variables d'instance est exactement celle de la classe dont elles font partie.
- Si st est une variable d'instance d'une classe/struct C, x est une variable locale de type x, st est accessible
 - en utilisant st ou this->st dans la classe/struct
 - en utilisant x.st dans le cadre de x (en supposant que st est public)

Variables d'instance : exemple

```
struct person{  
    C _prenom, _nom;  
  
    person(string prenom, string nom) : _prenom(prenom), _nom(nom) {  
        cout << " Person created: " << _prenom.st << " " << _nom.st << endl;  
    }  
  
    ~person(){  
        cout << " Person destroyed: " << _prenom.st << " " << _nom.st << endl;  
    } // _nom and _prenom destroyed here  
};
```

1 : chaque instance de la classe person a deux variables d'instance de type C

2 : Ceci appelle le constructeur de person, qui appelle les constructeurs des deux variables d'instance de type

4. Ensuite, nous imprimons que la personne a été créée

6. Lorsque moi est détruit, le corps du destructeur est exécuté, puis _nom et _prenom sont détruits

```
void demoInstance(){  
    cout << "start demoInstance" << endl;  
    person me{"John", "Iacono"};  
    {  
        person jean{"Jean", "Cardinal"};  
    } // Jean Cardinal Destroyed  
    cout << "end demoInstance" << endl;  
} // John Iacono Destroyed
```

2 : Ici on crée une personne John Iacono et on la stocke dans la variable me

Le même processus se produit pour Jean Cardinal, mais comme il est à l'intérieur d'un bloc imbriqué dans le bloc de la fonction, il est créé après John Iacono et détruit avant

5. À la fin du bloc, je suis détruit

```
start demoInstance  
Created: John  
Created: Iacono  
Person created: John Iacono  
Created: Jean  
Created: Cardinal  
Person created: Jean Cardinal  
Person destroyed: Jean Cardinal  
Destroyed: Cardinal  
Destroyed: Jean  
end demoInstance  
Person destroyed: John Iacono  
Destroyed: Iacono  
Destroyed: John  
Destroyed: John
```

Global Variables

- Les variables globales normales sont déclarées en dehors de toute classe ou fonction.
- Avoir une durée de vie égale à celle du programme.
- les espaces de noms peuvent être utilisés pour éviter les conflits de noms.
C'est une bonne pratique dans les fichiers qui doivent être inclus, car si vous déclarez deux variables globales avec le même nom de types différents, cela provoquera une erreur de compilation.

Variables globales : exemple

```
C x{"Global x"};
namespace N{
    C x{"Global x in namespace N"};
}

int main(){
    cout<<"Start Main"<<endl;
    cout<<x.st<<endl;
    cout<<N::x.st<<endl;
    cout<<"End Main"<<endl;
}
```

Deux variables globales sont créées,
toutes deux nommées x, une à
l'intérieur de l'espace de noms N

Nous pouvons imprimer les deux, en
utilisant N::x pour obtenir le x à
l'intérieur de l'espace de noms N

Ces variables globales sont créées
avant le démarrage principal

```
Created: Global x
Created: Global x in namespace N
Start Main
Global x
Global x in namespace N
End Main
Destroyed: Global x in namespace N
Destroyed: Global x
```

Les deux variables globales sont
détruites après les extrémités
principales

Les variables statiques sont des variables globales

- Toute variable déclarée statique est une variable globale
- Les variables statiques peuvent apparaître dans des fonctions, des méthodes ou des classes
- Les variables de classe statiques peuvent être référencées comme une variable d'instance normale ou en utilisant uniquement le nom de la classe : `classname::variablename`. En effet, alors que les variables d'instance ont besoin d'une instance, les variables statiques sont globales et ne dépendent d'aucune instance.

Variables de classe statiques : exemple

```
struct person2{
    string _prenom,_nom;
    static int count;
    person2(string prenom,string nom):_prenom(prenom),_nom(nom){
        cout<<" Person created: "<<_prenom<<" "<<_nom<<endl;
        count++;
    }
    ~person2(){
        cout<<" Person destroyed: "<<_prenom<<" "<<_nom<<endl;
        count--;
    }
};
```

int person2::count=0;

Cette classe person2 aura une variable de classe statique qui garde une trace du nombre d'instances de personnes qui existent actuellement

Donc, chaque instance de person2 a son propre nom et prenom, mais ils partagent tous le compte

On l'incrémente dans le constructeur

On le décrémente dans le destructeur

La variable est initialisée ici. Vous ne pouvez pas le faire en ligne dans la définition de la classe

```
start demoStatic
person2::count = 0
Person created: John Iacono
person2::count = 1
Person created: Jean Cardinal
person2::count = 2
Person destroyed: Jean Cardinal
person2::count = 1
end demoStatic
Person destroyed: John Iacono
```

```
void demoStatic(){
    cout<<"start demoStatic"<<endl;
    cout<<" person2::count = "<<person2::count<<endl;
    person2 me{"John","Iacono"};
    cout<<" person2::count = "<<person2::count<<endl;
    {
        person2 jean{"Jean","Cardinal"};
        cout<<" person2::count = "<<person2::count<<endl;
    } // Jean Cardinal Destroyed
    cout<<" person2::count = "<<person2::count<<endl;
    cout<<"end demoStatic"<<endl;
} // John Iacono Destroyed
```

Voici le même exemple que précédemment, sauf que nous imprimons les comptes

Au lieu de person2::count, nous aurions pu écrire jean.count ou john.count. Les trois sont exactement la même variable.

Pourquoi des variables statiques ?

Pourquoi ne pas simplement utiliser des variables globales normales ?

- L'encapsulation est le principe central de la programmation orientée objet
- L'encapsulation regroupe les données et le code en une seule unité, le cas échéant
- Comme le nombre ne concerne que la classe person2, le placer à l'intérieur de la classe person2 en tant que variable de classe statique plutôt que variable globale indique clairement où il est pertinent.
- Cela évite également les erreurs accidentnelles. Un nombre de variables globales peut être à l'origine de nombreuses erreurs.

Les problèmes des variables globales

```
int cnt=0;                                Ici, nous utilisons une variable globale cnt

struct person3{
    string _prenom,_nom;
    person3(string prenom,string nom):_prenom{prenom},_nom{nom}{

        cout<<" Person created: "<<_prenom<<" "<<_nom<<endl;
        cnt++;
    }
    ~person3(){
        cout<<" Person destroyed: "<<_prenom<<" "<<_nom<<endl;
        cnt--;
    } // _nom and _prenom destroyed here
};
```

```
start demoBadIdea
0 1 2 3 4 5 6 7 8 9
Person created: John Iacono
cnt = 11
end demoBadIdea
```

Mais le compte est de 11 !

```
void demoBadIdea(){

    cout<<"start demoBadIdea"<<endl;
    for (cnt=0;cnt<10;cnt++) {cout<<cnt<<" ";}
    cout<<endl;
    person3 me{ "John" , "Iacono" };
    cout<<" cnt = "<<cnt<<endl;
    cout<<"end demoBadIdea"<<endl;
}
```

Nous avons oublié d'écrire `int cnt!`
Mais nous avons eu de la "chance" et seul
il y avait une variable globale avec le
même nom qui a été utilisée

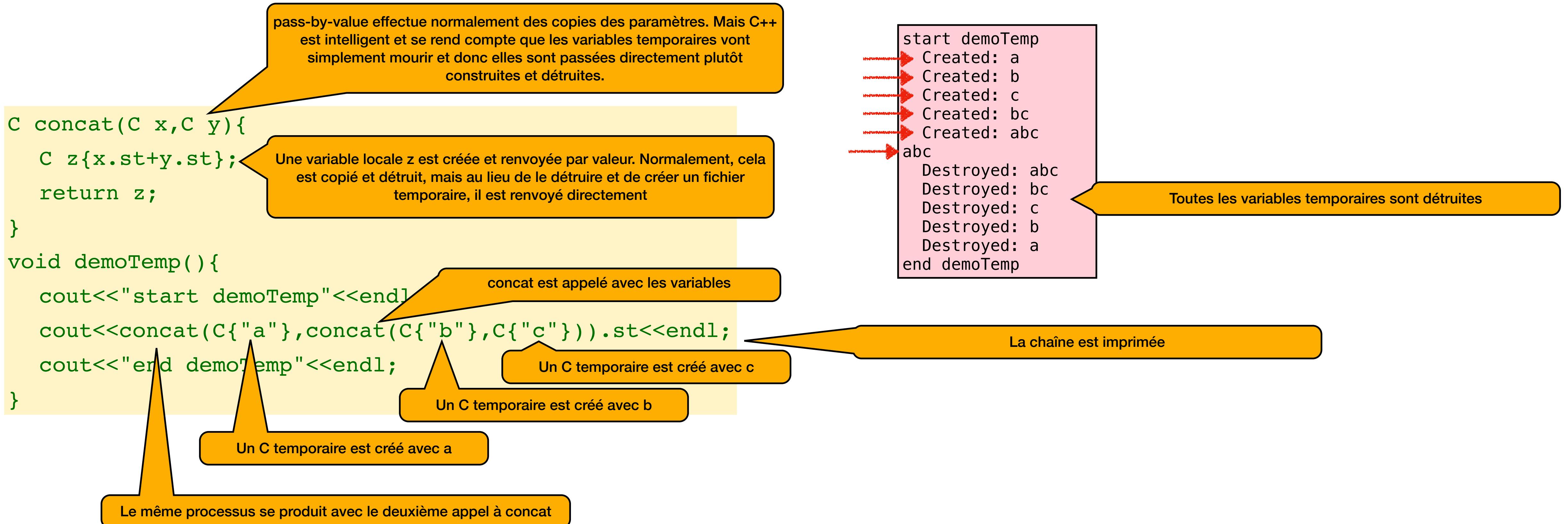
Nous ne créons qu'une seule personne3

Objets temporaires

(et copie élision)

- Les objets temporaires sont créés et détruits automatiquement
- Les objets temporaires n'ont pas de nom de variable
- Des variables temporaires sont créées
 - Par les valeurs renvoyées de la fonction et des opérateurs
 - Lors de l'appel direct d'un constructeur en utilisant le nom de la classe pour créer une instance de la classe. ex. : `string{"bonjour !"}`
- Les valeurs temporaires ne durent que le temps que l'expression
- C++ a des règles quelque peu compliquées pour quand les compilateurs sont autorisés ou obligés d'ignorer la copie pour des raisons d'efficacité, c'est généralement quand un objet mourrait et qu'un objet identique serait créé.

Variables temporaires : exemple



Durées de vie variables I : résumé

- Vous avez 3 choix de durées de vie pour les variables automatiques : local, instance et global en plus des objets temporaires.
- Vous devez bien les comprendre, car la création et la destruction d'instances sont essentielles à la programmation orientée objet
- Le point de création est souvent plus évident que le point de destruction
- Pour chaque donnée, vous devez décider quelle est la durée de vie appropriée
- Pour les variables globales, si elles ne doivent être utilisées que dans une certaine classe, fonction ou méthode, elles doivent y être des variables statiques. S'ils doivent être utilisés par un certain ensemble de fonctions ou dans un fichier qui sera inclus dans d'autres, ils doivent être dans un espace de noms. Cela évite les conflits de noms et suit le principe de l'encapsulation.
- Si aucune de ces trois durées de vie ne répond aux besoins de ce que vous essayez de faire, alors vous devez utiliser l'allocation dynamique de votre instance, où vous contrôlez manuellement la création et la destruction d'objets et est décrite dans une prochaine conférence.

Pointeurs et références : une revue

Programmation orientée objet

Prof. John Iacono

Pointeurs

- Un pointeur n'est qu'une adresse en mémoire. Tous les pointeurs ont la même taille.
- Le type d'un pointeur indique quel est le type de données qui (devraient être) stockées dans l'adresse mémoire du pointeur
- `*` est utilisé de trois manières différentes. Pour déclarer un pointeur, et pour déréférencer un pointeur :
 - `int *x;` // Déclare un nouveau pointeur `x` qui doit pointer sur un entier. `x` est de type `int *`
 - `*x=57;` //Regarde l'adresse mémoire dans `x` et la changer à l'entier `57`
 - `cout<<*x;` //Regarde l'adresse mémoire dans `x` et renvoie le contenu, le traitant comme un entier
- Une utilisation de `&` est l'opérateur d'adresse. Si `x` est de type `T`, `&x` est de type `T *`
- `*&x` est identique à `x` !

```
int x=50;
int *p=&x; // p pointe vers l'adresse mémoire de x*
*p=20;
cout<<x; //imprimera 20
```

- Un pointeur ne pointant actuellement pas vers quelque chose devrait avoir la valeur `nullptr` (à partir de C++11). L'ancien code utilise `NULL` ou `0`.
- Un pointeur peut être utilisé dans une expression logique et est faux si et seulement s'il est `nullptr/NULL/0`.

Pointeurs : utilisation

- Où obtenez-vous l'adresse sur laquelle placer le pointeur ?
 - Utiliser & sur une autre variable
 - D'un autre pointeur
 - From `new T`, qui a créé une nouvelle instance de type `T` et renvoie un pointeur vers celle-ci
- Danger des pointeurs : il y en a beaucoup
 - Différents types de pointeurs peuvent être convertis à l'aide de transtypages (typecasts) sans protection
 - Les données vers lesquelles pointe le pointeur peuvent ne plus être là
 - Arithmétique du pointeur
- Règle de base : vous, le programmeur, êtes responsable de vous assurer que lorsque vous déréférez un pointeur, il pointe vers quelque chose qui est toujours là et a le type approprié.
- Les erreurs de pointeur sont difficiles à déboguer car elles sont souvent intermittentes et peuvent apparaître ou disparaître en raison des options du compilateur, de l'ordinateur et du compilateur. "mais ça marche bien sur mon ordinateur...."

Exemple : classe d'incrémantation

- La classe a deux compteurs, `a` et `b`, tous deux initialement mis à 0
- L'un des deux compteurs est le compteur actif, initialement `a`
- La classe a trois méthodes :
 - `increment` : incrémente le compteur actif
 - `swap` : change le compteur actif
 - `toString` : donne une représentation sous forme de chaîne des deux compteurs,
- Comment représenter le compteur actif ? Nous utiliserons un pointeur vers le compteur actif.

Exemple : incrémentor

```
class incrementor{  
    int a=0,b=0;  
    int *active=&a;  
public:  
    void increment() {(*active)++;}  
    void swap() {active=(active==&a)?&b:&a;}  
    string toString() {return "a:"+to_string(a)+" b:"+to_string(b);}  
};
```

swap détermine si active pointe vers l'adresse mémoire de a ; si c'est le cas, active reçoit la valeur &b, sinon &a

Remarquez que nous ne changeons jamais a et b directement

La variable active est un pointeur sur un entier. Il est initialisé à l'adresse mémoire d'a

increment incrémente l'entier sur lequel pointe le pointeur actif

a:0 b:1
a:0 b:2
a:0 b:3
a:0 b:4
a:0 b:5
a:0 b:6
a:1 b:6
a:2 b:6
a:3 b:6
a:4 b:6
a:5 b:6
a:6 b:6
a:6 b:7
a:6 b:8
a:6 b:9
a:6 b:10
a:6 b:11
a:6 b:12
a:7 b:12
a:8 b:12

Après six incréments, il appelle swap

```
void testIncrementor(){  
    incrementor I;  
    for (int i=0;i<20;i++){  
        if (i%6==0) I.swap();  
        I.increment();  
        cout<<I.toString()<<endl;  
    }  
}
```

swap est appelé tous les six incréments

Ce code évite les problèmes courants avec les pointeurs :

active est un int * et n'est défini que avec l'adresse des entiers

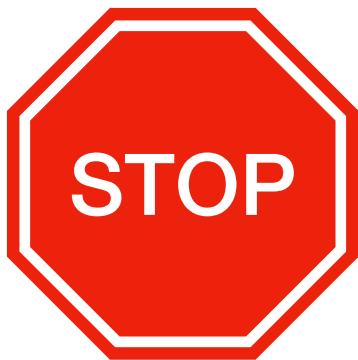
active a la même durée de vie que a et b, qui sont les seules choses vers lesquelles active pointe

Exemple 2 : minimum

Les nombreuses façons dont les choses peuvent mal tourner

- Objectif : écrire une fonction `minimum` qui fonctionne avec deux entiers et renvoie un pointeur vers le plus petit
- Je vais montrer plusieurs façons de coder cela
- La plupart ont tort
- Cependant, tout compiler!
- Certains donnent des avertissements au compilateur. N'ignorez jamais les avertissements du compilateur !

Minimum : tentative 1



Quel est le problème avec ce code? Pouvez-vous le voir?

```
int *min1(int a,int b){  
    int min=(a<b)?a:b;  
    return &min;  
}  
min est une variable  
locale qui n'existe  
plus lorsque la  
fonction retourne
```

Le code a l'air assez innocent

Mais attendez, on
renvoie un pointeur
vers `min`, pas vers `a`
ou `b`!

```
void testMinA(){  
    int a=20;  
    int b=10;  
    int *m=min1(a,b);  
    *m += 5;  
    cout<<"a:"<<a<<" b:"<<b<<endl;  
    // We hope for a:20 b:15  
}
```

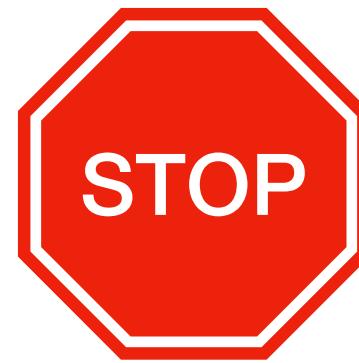
Comme `b` est le minimum de `a` et `b`, nous
voulons que cette ligne incrémente `b`

Cela pourrait faire planter votre programme ou
corrompre d'autres variables, car `m` n'est pas un
pointeur vers quoi que ce soit de valide. Sur mon
ordinateur, cela n'a rien fait.

`a` et `b` sont inchangés ! Qu'est ce qui ne
s'est pas bien passé?

a:20 b:10

Minimum : tentative 2



Quel est le problème avec ce code? Peux-tu le voir?

```
int *min1(int a,int b){  
    int min=(a<b)?a:b;  
    return &min;  
}
```

```
int *min2(int a,int b){  
    return (a<b)?&a:&b;  
}
```

Nous avons amélioré le code pour retourner `a` ou `b`. Est-ce que ça marche maintenant ?

```
void testMinB(){  
    int a=20;  
    int b=10;  
    int *m=min2(a,b);  
    *m += 5;  
    cout<<"a:"<<a<<" b:"<<b<<endl;  
    // We hope for a:20 b:15  
}
```

Que sont `a` et `b`? Ce sont des VARIABLES LOCALES ! Ce ne sont pas les mêmes que les `a` et `b` dans `testMinB`. Ce sont des copies. Leur durée de vie se termine lorsque la fonction retourne

Hé, c'est quoi exactement `m`? C'est un pointeur sur ce que `min1` a renvoyé

Dans ce cas, puisque nous avons fait quelque chose de manifestement mal, le compilateur C++ a été assez gentil pour émettre un avertissement.

Donc `m` est un pointeur vers une variable locale dont la durée de vie est terminée. Il s'agit d'une erreur qui pourrait provoquer le plantage du programme ou un comportement erratique. Il pourrait même changer une autre variable qui occupe maintenant l'adresse mémoire où les variables locales étaient

Cela ne fonctionne toujours pas!
a:20 b:10

```
dynamic.cpp:32:16: warning: address of stack memory  
associated with parameter 'a' returned [-Wreturn-stack-  
address]  
    return (a<b)?&a:&b;  
^  
dynamic.cpp:32:19: warning: address of stack memory  
associated with parameter 'b' returned [-Wreturn-stack-  
address]  
    return (a<b)?&a:&b;  
^
```

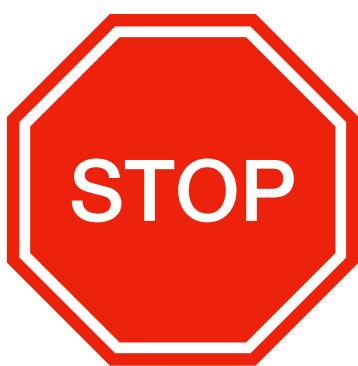
Qu'est-ce qui n'allait pas avec les tentatives 1 et 2 ?

- Nous voulons que `min2` renvoie un pointeur vers `a` ou un pointeur vers `b`.
- Mais `min2` copie `a` et `b` dans ses paramètres, également appelés `a` et `b`, et n'a pas accès à l'adresse des `a` et `b` de `testMinB` !
- Solution : `min` devrait prendre comme paramètres des pointeurs vers des entiers, et `testMinB` devrait donner à `min` les adresses de `a` et `b`

```
int *min2(int a,int b){  
    return (a<b)?&a:&b;  
}
```

```
void testMinB(){  
    int a=20;  
    int b=10;  
    int *m=min2(a,b);  
    *m += 5;  
    cout<<"a:"<<a<<" b:"<<b<<endl;  
    // We hope for a:15 b:20  
}
```

Minimum : tentative 3



Quel est le problème avec ce code? Pouvez-vous le voir?

```
int *min2(int a,int b){  
    return (a<b)?&a:&b;  
}
```

min3 prend maintenant des pointeurs vers des entiers

```
int *min3(int *a,int *b){  
    return (a<b)?a:b;  
}
```

a et b sont des pointeurs ! Ceci compare les adresses de a et b, pas les valeurs des entiers vers lesquels a et b pointent.

```
void testMinCa(){  
    int a=20;  
    int b=10;  
    int *m=min3(&a,&b);  
    *m += 5;  
    cout<<"a:"<<a<<" b:"<<b<<endl;  
    // We hope for a:20 b:15  
}
```

Lors de l'appel de min3, nous donnons les adresses de a et b

Ici, a a été déclaré en premier, il a donc reçu une adresse inférieure (vous ne pouvez pas compter dessus) et il a donc été renvoyé par min

a:20 b:15

Ce n'est pas ce que nous voulons. Comment pouvons-nous résoudre ce problème?

Ici, b a été déclaré en premier et il a donc été renvoyé par min

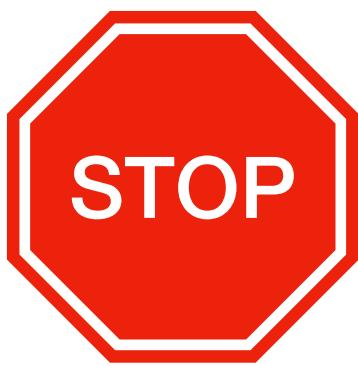
```
void testMinCb(){  
    int b=10;  
    int a=20;  
    int *m=min3(&a,&b);  
    *m += 5;  
    cout<<"a:"<<a<<" b:"<<b<<endl;  
    // We hope for a:20 b:15  
}
```

On inverse ces deux lignes. Cela ne devrait pas avoir d'importance

Qu'est-il arrivé????

a:25 b:10

Minimum : tentative 4



Quel est le problème avec ce code? Pouvez-vous le voir?

```
int *min3(int *a,int *b){  
    return (a<b)?a:b;  
}
```

```
int *min4(int *a,int *b){  
    return (*a<*b)?a:b;  
}
```

Comparons maintenant les nombres entiers vers lesquels a et b pointent

```
void testMinDa(){  
    int a=20;  
    int b=10;  
    int *m=min4(&a,&b);  
    *m += 5;  
    cout<<"a:"<<a<<" b:"<<b<<endl;  
    // We hope for a:20 b:15  
}
```

a:20 b:15

Rien! Ça fonctionne maintenant!

```
void testMinDb(){  
    int b=10;  
    int a=20;  
    int *m=min4(&a,&b);  
    *m += 5;  
    cout<<"a:"<<a<<" b:"<<b<<endl;  
    // We hope for a:20 b:15  
}
```

a:20 b:15

Les références

- De manière simplifiée, une référence équivaut à un pointeur (sauf que `nullptr`) dont vous ne pouvez pas modifier la valeur. Vous pouvez accéder et changer à ce qui est pointé.
- La syntaxe est la suivante (**&** a deux significations complètement différentes !):
 - p est un pointeur vers x:
`int *p=&x;`
r est une référence vers x :
`int &r=x;`
 - Avec le pointeur `p`, pour obtenir la valeur de l'entier vers lequel il pointe, on écrit `*p`
 - Avec la référence `r`, pour obtenir la valeur de l'entier auquel il se réfère, on écrit simplement `r`
- En commun avec les pointeurs, la déclaration d'une référence ne provoque pas la création d'un objet ni n'affecte sa durée de vie. L'objet est déjà là.

Pourquoi des références

- Les pointeurs peuvent faire beaucoup de choses que vous ne voudriez peut-être pas faire, et que les références ne peuvent pas faire :

`p+=7 ; //Déplacer le pointeur vers l'avant`

`p=&z; //Changer le pointeur pour pointer vers autre chose`

`int *p=nullptr; // Initialiser un pointeur vers un nullptr`

`if (p==nullptr); //Vérifier si un pointeur est un nullptr`

`if (p1>p2) //Comparer les adresses mémoire des pointeurs`

- La syntaxe des références est souvent beaucoup plus agréable, notamment pour le passage de paramètres
- Moins de bogues. Toutes les erreurs de pointeur avec un minimum 1-3 sont impossibles à faire avec des références !
- Morale : au lieu de pointeurs, utilisez des références lorsque le pointeur serait pointer vers un seul vrai objet.

Minimum tentative 5 : avec références

Pointeurs

```
int *min4(int *a,int *b){  
    return (*a<*b)?a:b;  
}
```

Nous ne modifions jamais ces pointeurs,
alors convertissons-les en références

```
void testMinDa(){  
    int a=20;  
    int b=10;  
    int *m=min4(&a,&b);  
    *m += 5;  
    cout<<"a:"<<a<<" b:"<<b<<endl;  
    // We hope for a:20 b:15  
}
```

a:20 b:15

Références

```
int &min5(int &a,int &b){  
    return (a<b)?a:b;  
}
```

Pas besoin de déréférer. Aussi aucune chance d'erreur comme dans la tentative 3

```
void testMinE(){  
    int a=20;  
    int b=10;  
    int &m=min5(a,b);  
    m += 5;  
    cout<<"a:"<<a<<" b:"<<b<<endl;  
    // We hope for a:20 b:15  
}
```

a:20 b:15

Pointeurs et références : résumé

- Les pointeurs vous permettent d'accéder à des objets créés ailleurs à l'aide d'une adresse mémoire. Vous obtenez normalement cette adresse mémoire à partir de :
 - Utilisation de l'opérateur d'adresse : `&`
 - D'un autre pointeur
 - A partir d'une fonction qui alloue de la mémoire et renvoie une adresse mémoire, que nous décrirons plus tard. Il s'agit notamment du `malloc` et du `new`
- Les références sont des pointeurs qui sont fixés à une certaine adresse. Cette restriction réduit les erreurs et rend le code plus lisible. Utilisez des références au lieu de pointeurs lorsque cela est possible.
- Les références sont couramment utilisées dans les paramètres des fonctions/méthodes. Cela permet de modifier le paramètre sans que l'appelant utilise l'opérateur address-of pour obtenir un pointeur. Nous en discuterons davantage dans la vidéo sur le passage des paramètres.
- Les références n'existent pas en C, seulement en C++. Ainsi, le code C utilise des pointeurs là où C++ utiliserait normalement des références.

Façons de passer des paramètres

Programmation orientée objet

Prof. John Iacono

- Passage par valeur

```
void f(string x)
```

- Passage par référence

```
void f(string &x)
```

- Passage par référence constante

```
void f(const string &x)
```

- Passage par référence rvalue

```
void f(string &&x)
```

- Passage par pointeur

```
void f(string *x)
```

Passage par référence:

```
string y="Hello";
```

```
f(y);
```

y et x sont le même objet. Si f change y, x changera aussi.

Passage par valeur:

```
string y="Hello";
```

```
f(y);
```

Ceci fait une copie de y en appelant le constructeur de copie de string. f ne peut pas changer y.

Passage par référence rvalue:

```
string y="Hello";
```

```
string z="World";
```

```
f(y+z);
```

f sait que x est une valeur intermédiaire (une rvalue), dans ce cas une instance de type chaîne formée par l'opérateur +. Ainsi, f pourra peut-être en profiter, sachant que x mourra après la fin de f.

Passage par référence constante:

```
string y="Hello";
```

```
f(y);
```

x et y sont la même chaîne. Toute modification de x dans est interdit.

Passage par pointeur:

```
Thing y;
```

```
f(&y);
```

Ceci est juste passer par valeur d'un pointeur. Dans f, *x est exactement la même chose que y. L'ancien code C l'utilise car C n'a pas de passage par référence.

Qu'est-ce
qu'une rvalue ?

	Copie l'objet ?	Peut modifier l'objet ?	Peut appeler avec un rvalue/temporaire
Passage par valeur <code>void f1(string s)</code>	Yes	Peut modifier la copie, pas l'original	Yes
Passage par référence constante <code>void f2(const string &s)</code>	No	No	Yes
Passage par référence <code>void f3(string &s)</code>	No	Yes	No
Passage par référence rvalue <code>void f4(string &&s)</code>	No	Oui, mais l'original était une rvalue	Oui, ne peut être appelé qu'avec une rvalue

```
string s="hello";
string t="world";
f1(s+t);
f2(s+t);
f3(s+t);
f4(s+t);
```

Lequel utiliser ?

Avez-vous besoin de modifier l'objet d'origine passé ?

Non

Avez-vous réellement
besoin d'une copie de
l'objet ?

La copie prend du
temps et doit être
évitée si possible

Oui

Passage par référence
`void f(string &f)`

Non

L'objet est-il très petit
ou de type intégré (par
exemple, int ou
pointeur)

Oui

Passage par valeur
`void f(string s)`

Oui

Utiliser le passage par
référence constante
`void f(const string &f)`

Non

Pourriez-vous implémenter
la fonction plus
rapidement si l'objet était
une rvalue ?

Yes

Ecrire une autre
fonction avec passage
par référence rvalue
`void f(string &&f)`

Exercice

Écrivez une fonction `replaceDifferentWith(?, ?, ?)` qui fonctionne comme ceci :

```
string s1="hello world!";
string s2="Hello_World?";
replaceDifferentWith(s1,s2,'-');
cout<<s1; Prints "-hello--orld-"
cout<<s2; Prints "Hello_World?"
```

Bien sûr, faites attention au bon type de passage de paramètre !

Constructeurs et destructeurs

Programmation orientée objet

Prof. John Iacono

Constructeurs et destructeurs : objectif

- Les constructeurs sont exécutés lorsqu'une instance d'un objet est créée. Ils initialisent les variables d'instance et font tout ce qui doit être fait à la création.
- Les destructeurs sont exécutés lorsqu'une instance d'un objet atteint la fin de sa durée de vie. Il effectue tous les nettoyages nécessaires. Souvent il libère le la mémoire dynamique dont cette instance est responsable.

Points principaux

- Le(s) constructeur(s) et le destructeur sont des méthodes spéciales
 - Les constructeurs ont pour nom le nom de la classe
 - Les destructeurs ont pour nom un tilde suivi du nom de la classe
 - Les constructeurs et les destructeurs n'ont pas de valeurs de retour (même pas void)
- Un constructeur est appelé lors de la création d'un objet, éventuellement avec des paramètres
 - Comme d'autres fonctions/méthodes Il peut y avoir plusieurs constructeurs avec des paramètres différents
 - Si vous n'écrivez pas de constructeur, C++ en fait un pour vous
 - Il existe plusieurs manières d'appeler un constructeur. Ceci est déroutant!
 - Les constructeurs sont responsables de l'initialisation des variables d'instance de la classe
 - Les constructeurs pour copier and déplacer un object de la même type sera dans le prochain leçon !
- Le destructeur est appelé lorsque l'objet est détruit
 - Il n'y a qu'un seul destructeur. Il n'a pas de paramètres. Jamais.
 - Si vous n'écrivez pas de constructeur, C++ en fait un pour vous

Notions de base : définir un constructeur

```
class Student {  
    string name;  
    Photo *photo;  
    Student(string name1,Photo photo1):name{name1},photo{new Photo{photo1}}{};  
    ...  
}
```

Dans la classe Student

En dehors de la classe
Etudiante

```
class Student {  
    string name;  
    Photo *photo;  
    Student(string,Photo);  
    ...  
}  
  
Student::Student(string name1,Photo photo1):name{name1},photo{new Photo{photo1}}{};
```

- Le nom du constructeur est le nom de la classe
- Les paramètres
- La liste des initialiseurs de membres: this is where the constructor calls the constructors of the class's instance variables
- Le code principal du constructeur. Often empty if all the constructor needs to do is initialize the instance variables.

```
class Student {  
    string name;  
    Photo *photo;  
    Student(string name1,Photo photo1):name{name1},photo{new Photo{photo1}}{};  
    ...  
}
```

Certaines personnes aiment utiliser les mêmes noms de variables pour les variables d'instance et les paramètres du constructeur

La liste d'initialisation doit faire référence à des variables d'instance

```
class Student {  
    string name;  
    Photo *photo;  
    Student(string name,Photo photo):name{name},photo{new Photo{photo}}{};  
};
```

Ce doit être le paramètre, car dans la liste d'initialisation, les variables d'instance ne sont pas encore disponibles pour l'utilisation

Constructeurs par défaut

- Ce sont des constructeurs qui ne prennent aucun paramètre
- Si votre classe n'a pas de constructeur, C++ crée un constructeur par défaut pour vous
- Si vous écrivez un constructeur, C++ ne vous donnera pas de constructeur par défaut
- Les classes sans constructeurs par défaut nécessitent que les constructeurs soient appelés avec des paramètres. C'est une cause fréquente d'erreurs.

Exemple : aucun constructeur par défaut

```
class Point {  
    int x,y;  
public:  
    Point(int x,int y):x{x},y{y}{}  
    int getX() {return x;}  
    int getY() {return y;}  
};  
  
Point p; //ERROR  
Point q(4,5); //OK
```

Nous avons défini un constructeur avec des paramètres, donc il n'y a plus de constructeur par défaut

C'est simple, nous devons appeler un constructeur dans le cadre de la déclaration d'un point de variable

```
constructor.cpp:47:8: error: no matching constructor for initialization of 'Point'  
    Point p;  
           ^  
  
constructor.cpp:6:7: note: candidate constructor (the implicit copy constructor) not  
viable: requires 1 argument, but 0 were provided  
class Point {  
           ^  
  
constructor.cpp:6:7: note: candidate constructor (the implicit move constructor) not  
viable: requires 1 argument, but 0 were provided  
constructor.cpp:9:3: note: candidate constructor not viable: requires 2 arguments, but  
0 were provided  
    Point(int x,int y):x{x},y{y}{};
```

```
class Rectangle {  
    Point p1,p2;  
public:  
    void setUL(int x,int y){p1=Point(x,y);}  
    void setLR(int x,int y){p2=Point(x,y);}  
}
```

Ici, nous avons une classe Rectangle, sans constructeur déclaré et a normalement un constructeur par défaut. Mais cela ne fonctionne pas, car comment peut-on initialiser `p1` et `p2` ?

C'est l'erreur. Il dit que parce que `p1` est un rectangle, qui n'a pas de constructeur par défaut, Rectangle ne peut pas avoir de constructeur par défaut, même un par défaut. Il est donc sans constructeur, une erreur

```
Rectangle r;  
  
constructor.cpp:50:12: error: call to implicitly undefined function 'Rectange'  
    Rectangle r;  
           ^  
  
constructor.cpp:16:9: note: default constructor of 'Rectange' is implicitly deleted  
because field 'p1' has no default constructor  
    Point p1,p2;  
           ^  
1 error generated.  
make: *** [constructor] Error 1
```

```
class Rectangle {  
    Point p1,p2;  
public:  
    void setUL(int x,int y){p1=Point(x,y);}  
    void setLR(int x,int y){p2=Point(x,y);}  
    Rectange(int x1,int y1,int x2,int y2){  
        setUL(x1,y1);setLR(x2,y2); //Error!!!  
    }  
}  
  
Rectangle r{1,2,3,4};
```

Ici, nous essayons de résoudre le problème en créant un constructeur. Mais le code est toujours faux, car lorsqu'il atteint `setUL`, `p1` devrait déjà être construit, ce qui n'est pas possible car nous n'avons pas appelé son constructeur. Comment faisons-nous cela?

```
constructor.cpp:30:3: error: constructor for 'Rectange' must explicitly initialize the  
member 'p1' which does not have a default constructor  
    Rectange(int x1,int y1,int x2,int y2){  
           ^  
constructor.cpp:26:9: note: member is declared here  
    Point p1,p2;  
           ^  
constructor.cpp:6:7: note: 'Point' declared here  
class Point {  
           ^  
constructor.cpp:30:3: error: constructor for 'Rectange' must explicitly initialize the  
member 'p2' which does not have a default constructor  
    Rectange(int x1,int y1,int x2,int y2){  
           ^  
constructor.cpp:26:12: note: member is declared here  
    Point p1,p2;  
           ^  
constructor.cpp:6:7: note: 'Point' declared here  
class Point {  
           ^  
2 errors generated.
```

```
class Rectangle {  
    Point p1,p2;  
public:  
    void setUL(int x,int y){p1=Point(x,y);}  
    void setLR(int x,int y){p2=Point(x,y);}  
    Rectange(int x1,int y1,int x2,int y2){  
        p1{x1,y1},p2{x2,y2}{}  
    }  
}  
  
Rectangle r{1,2,3,4};
```

Ici on répare ! Cela montre que la liste d'initialisation des membres n'est pas un raccourci, c'est le moyen requis pour initialiser n'importe quelle variable membre sans constructeur par défaut

Constructeur par défaut : exercice

- Quel est le problème avec ces deux morceaux de code?
- Faites-le correctement
- Nappelez le constructeur de **p** qu'une seule fois !

```
void exercise(int a,int b){  
      
    // Attempt 1  
    Point p;  
    if (a<b) p=Point(a,b); else p=Point(b,a);  
    cout<<p.getX()<<" "<<p.getY()<<endl;  
    return p  
}  
}
```

```
void exercise(int a,int b){  
      
    // Attempt 2  
    if (a<b) Point p{a,b}; else Point p{b,a};  
    cout<<p.getX()<<" "<<p.getY()<<endl;  
    return p  
}  
}
```

Classes versus types standards

- Un constructeur d'une classe est toujours appelé à la création d'une instance de la classe.
- Mais... C++ ne garantit aucune initialisation des types standards.

```
int x;  
C y;    // Same as C y{} or C y();
```

- Nous n'avons aucune idée de ce qu'il y a dans `x`, et cela peut varier en fonction du compilateur et même des drapeaux passés au compilateur.
- Cependant, `y` a été initialisé en utilisant le constructeur `C::C()`!

Classes versus types standards : exemple

- Considérez:

```
for (int x;x<10;x++) cout<<x<<endl;
```

- `x` n'est pas initialisé ! Ça peut être n'importe quoi !

- Considérez:

```
for (vector<int> v;
      v.size()<10;
      v.push_back(1))
cout<<v.size()<<endl;
```

- `v` est correctement construit.

Appel d'un constructeur : initialisation

Appel d'un constructeur : initialisation

- Les constructeurs sont appelés au début de la durée de vie d'un objet.
- Les paramètres peuvent être passés aux constructeurs.
- Il peut y avoir plusieurs constructeurs.
- Selon le type d'objet, vous démarrez sa durée de vie de manière différente et cela se reflète dans les différentes manières d'appeler un constructeur.
 - Pour les variables globales et locales : **Point p{2,3};**
 - Pour créer un objet temporaire: **Point{2,3}**
 - Pour les objets alloués dynamiquement : **Point *p=new Point{2,3}**
 - Pour les variables d'instance dans le cadre d'une liste d'initialisation d'instance

```
Class Rect {  
    Point p1,p2;  
    Rect::Rect(Point a,Point b) :p1{a},p2{b}{ };  
    ...  
}
```

Appeler un destructeur

- Les destructeurs sont appelés à la fin de la durée de vie d'une instance.
 - A la fin du programme pour les variables globales et les variables membres statiques
 - A la fin d'un bloc pour les variables locales
 - Lorsque `delete` est appelé pour les variables allouées dynamiquement.
 - Lorsque l'expression est évaluée, pour un objet temporaire
 - Pour les variables d'instance, à la fin de la durée de vie de l'instance à laquelle elles appartiennent

Manières d'appeler un constructeur : local, global et dynamique

- Variables locales et globales

```
C x;  
C x{};  
C x(); //déclaration de  
fonction!!!!  
C x(5);  
C x{5};  
C x=5;
```

- Variables allouées dynamiquement

```
C *x=new C;  
C *x=new C{};  
C *x=new C();  
C *x=new C{5};  
C *x=new C(5);
```

Façons d'appeler un constructeur : variables d'instance

- Quand les variables d'instance sont-elles créées ?
 - Lorsqu'une instance de la classe à laquelle ils appartiennent est créée.
- Il existe une syntaxe spéciale pour appeler le constructeur d'une variable d'instance à partir du constructeur de la classe à laquelle elle appartient.

```
class C {  
    D x,y,z;  
public:  
    C(int a,int b):x{a+b},y{a-b},z{}{}  
};  
  
C v(5,4);
```

- Cela appelle le constructeur `C::C(5,4)` qui appelle :
 - Le constructeur `D::D(9)` pour construire la variable `v::x`.
 - Le constructeur `D::D(1)` pour construire la variable `v::y`.
 - Le constructeur `D::D()` pour construire la variable `v::z`.
- Il existe plusieurs raccourcis pour la liste d'initialisation

Variables d'instance : Raccourci #1 Omission de la construction par défaut

```
class C {  
    D x,y,z;  
public:  
    C(int a,int b):x{a+b},y{a-b},z{}{}  
};
```

```
class C {  
    D x,y,z;  
public:  
    C(int a,int b):x{a+b},y{a-b}{}  
};
```

```
class Player{
    string level;
    string name; int age;
public:
    Player(string name,int age):level("Beginner"),name{name},age{age}{};
    Player(string name):level("Beginner"),name{name},age{20}{};
    Player(int number,int age):level("Beginner"),name{"Player"+to_string(number)},age{age}{};
    Player(int number):level("Beginner"),name{"Player"+to_string(number)},age{20}{};
}
```

Tous les constructeurs mettent
level à la même chose (et ce n'est
pas un paramètre)

```
class Player{
    string level{"Beginner"}; // much better!!!
    string name; int age;
public:
    Player(string name,int age):name{name},age{age}{}
    Player(string name):name{name},age{20}{}
    Player(int number,int age):name{"Player"+to_string(number)},age{age}{}
    Player(int number):name{"Player"+to_string(number)},age{20}{}
};
```

Nous pouvons simplement initialiser
level ici à la place comme un
raccourci

```
class Player{
    string level{"Beginner"};
    string name; int age;
public:
    Player(string name,int age):name{name},age{age}{}
    Player(string name):Player{name,20}{}
    Player(int number,int age):Player{"Player"+to_string(number)},age{age}{};
    Player(int number):Player{number,20}{}
};
```

Un constructeur au lieu d'initialiser les
variables d'instance, peut appeler un autre
constructeur de la même classe. C'est ce
qu'on appelle la délégation et vous permet
d'isoler la logique du constructeur en un seul
endroit

```
class Player{
    string level{"Beginner"};
    string name;
    int age;
public:
    Player(string name,int age=20):name{name},age{age}{}
    Player(int number,int age=20):Player{"Player"+to_string(number)},age{age}{};
};
```

Comme toute autre fonction ou méthode,
nous pouvons avoir des valeurs par
défaut.

C'est beaucoup plus sympa que le code
d'origine et fait exactement la même
chose,

Variables membres statiques

- Rappelez-vous que les variables membres statiques ont donné une durée de vie globale. Ils ne doivent donc être initialisés qu'une seule fois.
- Comment initialiser `personCount` ?

```
class Person{
    string name;
    static int personCount=0;
public:
    Person(string name):name{name}{personCount++;}
};
```

- Cela ne fonctionne pas!

```
class Person{
    string name;
    static int personCount;
public:
    Person(string name):name{name}{personCount++;}
};

int Person::personCount=0;
```

Pourquoi cette restriction ? Dans un grand projet avec plusieurs fichiers de code source, la classe `Person` peut être incluse dans de nombreux fichiers qui sont compilés, mais cette initialisation ne peut apparaître que dans un seul fichier.

Listes d'initialisation versus parenthèse versus =

```
string x1{"Hello"};  
string x2("Hello");  
string x3="Hello";
```

- Déroutant!
- Pour de nombreuses classes, elles sont toutes identiques
- Pour certaines classes, elles peuvent toutes être différentes !
- La meilleure pratique consiste à en choisir un, sauf si vous avez une raison de faire autrement. J'utiliserai {}.
- Nous allons maintenant nous concentrer sur la différence entre () et {}

Listes d'initialisation: () vs {}

```
class C{
public:
    C(int a, int b){cout<<"Two ints"<<endl;}
};
```

```
C x1{1,2}; //Prints "Two ints"
C x2(1,2); //Prints "Two ints"
```

{} et {} semblent être identiques

La liste d'initialisation ne peut être appelée qu'avec {}, pas ()

```
class C{
public:
    C(int a, int b){cout<<"Two ints";}
    C(initializer_list<int> args){
        cout<<"args: "<<endl;
        for(auto x::args) cout<<x<<endl;
    }
    C(){cout<<"Default"<<endl;}
};
```

```
class C {
public:
    C(initializer_list<int> args){
        cout<<"args: ";
        for(auto x:args) cout<<x<< " ";
        cout<<endl;
    }
};
```

C'est une façon d'accepter un nombre variable d'arguments

```
C x1{1,2,3,4,5}; //Prints args:1,2,3,4,5
C x2(1,2,3,4,5); //ERROR
C x3{1,2}; //Prints args: 1 2
```

Aucun paramètre n'appellera la liste d'initialisation, un {} vide

```
class C{
public:
    C(int a, int b){cout<<"Two ints";}
    C(initializer_list<int> args){
        cout<<"args:"<<endl;
        for(auto x::args) cout<<x<<endl;
    }
};
```

Avec des paramètres normaux et une liste d'initialisation, () appelle la normale et {} appelle la liste

```
C x{1,2}; //Prints args:1,2
C x(1,2); //Prints "Two ints"
C x; //Error
C x{}; //Prints args:
C x(); //Declares a function x:
```

Je reçois un avertissement ici:
warning: empty parentheses interpreted as a function declaration

```
C x{1,2}; //Prints args:1,2
C x(1,2); //Prints "Two ints"
C x; //Prints "Default"
C x{}; //Prints "Default" (!!!)
C x(); //Declares a function x:
```

L'ajout d'un constructeur par défaut rend ce travail

Étonnamment, {} est appelé sur un constructeur par défaut s'il est disponible et non sur la liste d'initialisation !

Quel constructeur : résumé

- `C x` : utilise le constructeur par défaut
- `C x{ }` : utilise le constructeur par défaut s'il est présent, sinon le constructeur de la liste d'initialisation
- `C x{1,2}` : Utilise le constructeur de liste d'initialisation s'il est présent, sinon le constructeur normal
- `C x(1,2)` Utilise toujours le constructeur normal
- `C x()` : Une fonction `x` est déclarée
- L'ancien code utilise `()` exclusivement. Les listes d'initialisation et les constructeurs ont été ajoutés au C++ en 2011.
- La notation `{ }` est appelée ***initialisation uniforme*** car elle peut être utilisée pour initialiser n'importe quoi.

Exemple : `vector`

- Vous pouvez penser que vous pouvez ignorer cela en n'écrivant jamais de classes avec des listes d'initialisation et des constructeurs normaux.
- Mais votre code ne vit pas dans le vide, vous devez utiliser le code des autres
- Considérez la classe `std::vector` :

```
vector<int> v1{4,1}; // v1 has [4,1]
vector<int> v2(4,1); // v2 has [1,1,1,1]
```

Exemple : vector

```
vector();
explicit vector (const allocator_type& alloc);
explicit vector (size_type n, const allocator_type& alloc = allocator_type());
vector (size_type n, const value_type& val,
        const allocator_type& alloc = allocator_type()); /*1
template <class InputIterator>
vector (InputIterator first, InputIterator last,
        const allocator_type& alloc = allocator_type());
vector (const vector& x);
vector (const vector& x, const allocator_type& alloc);
vector (vector&& x);
vector (vector&& x, const allocator_type& alloc);
vector (initializer_list<value_type> il,
        const allocator_type& alloc = allocator_type()); /*2
```

Destructeurs!

Destructeurs!

- Facile : `~Classname()`
- Un seul
- Aucun type de retour (comme constructeur)
- Appelé en fin de vie de l'instance
- Destructeur vide fourni automatiquement par C++ (il y aura des exceptions)
- Souvent pas nécessaire sauf lors de l'utilisation de mémoire allouée dynamiquement
- Les destructeurs de toutes les variables membres sont appelés après la fin du code du destructeur.

Destructeurs et mémoire dynamique

- N'oubliez pas que lorsque vous utilisez `new` pour allouer de la mémoire dynamiquement, vous devez appeler `delete` lorsqu'il n'est plus nécessaire
- Lors du codage, il doit être clair qui a la responsabilité d'appeler `delete`
- Typiquement, le destructeur est chargé de libérer toute mémoire allouée dynamiquement par cette instance de la classe.

Exemple de destructeur :

```
class Person{  
    string name;  
    static int personCount;  
public:  
    Person(string name):name{name}{personCount++;}  
    ~Person(){personCount--;}  
    static int count(){return personCount;}  
};  
  
int Person::personCount=0;
```

Rappelez-vous qu'une variable statique est une variable globale mais se trouve dans l'espace de

Tant pour les variables statiques que pour les fonctions, elles ne sont PAS associées à une instance particulière

en dehors de la classe, vous pouvez utiliser
Person::personCount
et
Person::count()

Une fonction membre statique est comme une fonction déclarée en dehors de la classe, mais elle a accès aux parties privées de la classe.

Trois destructeurs sont appelés

```
void demoPerson(){  
    cout<<Person::count()<<endl;  
    Person p1{"John"};  
    {  
        Person p2{"Jack"};  
        Person p3{"Jill"};  
        Person p4{"Jane"};  
        cout<<Person::count()<<endl;  
    }  
    cout<<Person::count()<<endl;  
}
```

4

1

Ce que je ne vous ai pas (encore) dit

- Règles de conversion différentes pour () vs {}
- À traiter plus tard :
 - Listes d'initialisation et variables membres publiques
 - Références
 - Héritage
 - Construction de tableaux
 - Copier et déplacer

Résumé : constructeurs et destructeurs

- L'idée est simple : les constructeurs sont appelés lorsqu'un objet est créé et les destructeurs sont appelés et qu'un objet a atteint la fin de sa vie
- De nombreux détails
 - Syntaxe différente basée sur le membre local/global/statique/instance/alloué dynamiquement
 - `x()` vs `x{}` vs `x`
 - Différence dans l'initialisation des types intégrés par rapport aux classes
 - Nuance lorsqu'un constructeur par défaut et un destructeur sont fournis sans que vous les ayez codés
- Tout a une raison si tu la comprends bien.....

Durées de vie II: Pointeurs et l'Allocation de memoire dynamique

Programmation orientée objet

Prof. John Iacono

Durées de vie

- Global, Local, Instance, Temporaire : les instances sont créées et détruites automatiquement
- Avec `new` et `delete`, vous pouvez contrôler manuellement la création et la destruction d'instances
 - Pour créer une nouvelle instance de type `T`, utilisez `new`, qui retourne un pointeur vers l'objet nouvellement créé
`T* p=new T;`
 - Vous pouvez transmettre n'importe quel paramètre au constructeur lorsque vous utilisez `new` :
`string *p=new string{"Hello!"};`
 - Pour supprimer l'instance, utilisez `delete` sur un pointeur vers l'instance :
`delete p;`
`delete` appelle le destructeur et libère la mémoire utilisée par l'instance

Un grand pouvoir implique de grandes responsabilités

- Les règles sont simples : appelez `new` pour obtenir un nouvel objet pointeur et appelez `delete` une fois sur ce pointeur lorsque vous en avez terminé.
- De nombreuses façons de tout gâcher
- Le compilateur est de peu d'aide pour identifier les problèmes
- Un code incorrect ne conduit pas toujours à des erreurs évidentes
- Lorsque vous obtenez un pointeur de `new`, vous devez vous assurer
 - ce pointeur est toujours stocké quelque part
 - `delete` est appelé sur ce pointeur exactement une fois
 - ce pointeur n'est jamais déréférencé ou a été appelé avec `delete` après l'appel de `delete`
- Ces règles sont beaucoup plus faciles à suivre si vous ne stockez le pointeur que dans une variable à la fois. Si le pointeur est stocké dans plusieurs variables en même temps, une plus grande prudence doit être prise.
- Une bonne nouvelle : vous pouvez appeler `delete` sur `nullptr`. Cela ne fait rien. Donc, n'écrivez pas :
`if (p) delete p`, écrivez simplement : `delete p`

Erreurs avec la mémoire dynamique

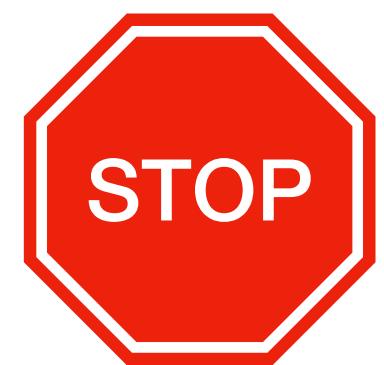
```
void bad1(){
    int i=0;
    int *p;
    for (int c=0;c<1000;c++){
        p=new int{c*c};
        i+=*p;
    }
    delete p;
    cout<<i<<endl;
}
```

332833500

Nouveau est appelé 1000 fois

Mais delete n'est appelé qu'une seule fois

999 objets restent alloués et nous n'avons aucun moyen d'appeler delete sur eux car nous n'avons aucune variable avec des pointeurs vers eux



Quel est le problème avec ce code? Pouvez-vous le voir?

Fuite de mémoire : lorsque vous oubliez d'appeler `delete` lorsque vous avez terminé avec un objet

Ce code compile et fonctionne. Mais si vous continuez appeler `bad1`, cela continuerait à utiliser de plus en plus de mémoire jusqu'à ce que votre ordinateur ralentisse et finisse par planter

Erreurs avec la mémoire dynamique 2

```
class student{
    string *name;
    string *town;
public:
    student(string *name,string *town):name{name},town{town}
{};
    ~student(){delete name;delete town;}
};

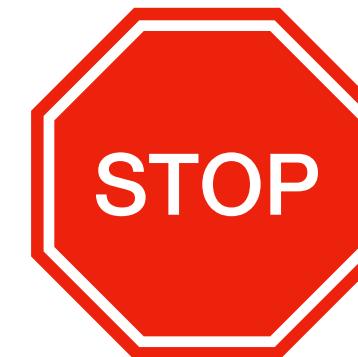
void bad2(){
    string *ixelles=new string{"ixelles"};
    student john(new string("john iacono"),ixelles);
    student jean(new string("jean cardinal"),ixelles);
}
```

Le destructeur est appelé et delete est appelé deux fois sur "ixelles"

Ici, le jean et le jean sortent tous les deux du champ d'application et tous les deux ont leurs destructeurs appelés

Regardez la chaîne "ixelles". Il n'y a qu'une seule chaîne de ce type et elle est créée ici

Un pointeur vers cette chaîne est donné à la fois à jean et à john



Quel est le problème avec ce code? Pouvez-vous le voir?

Double **delete** : se produit généralement lorsque vous copiez un pointeur sans savoir quelle copie doit être supprimée

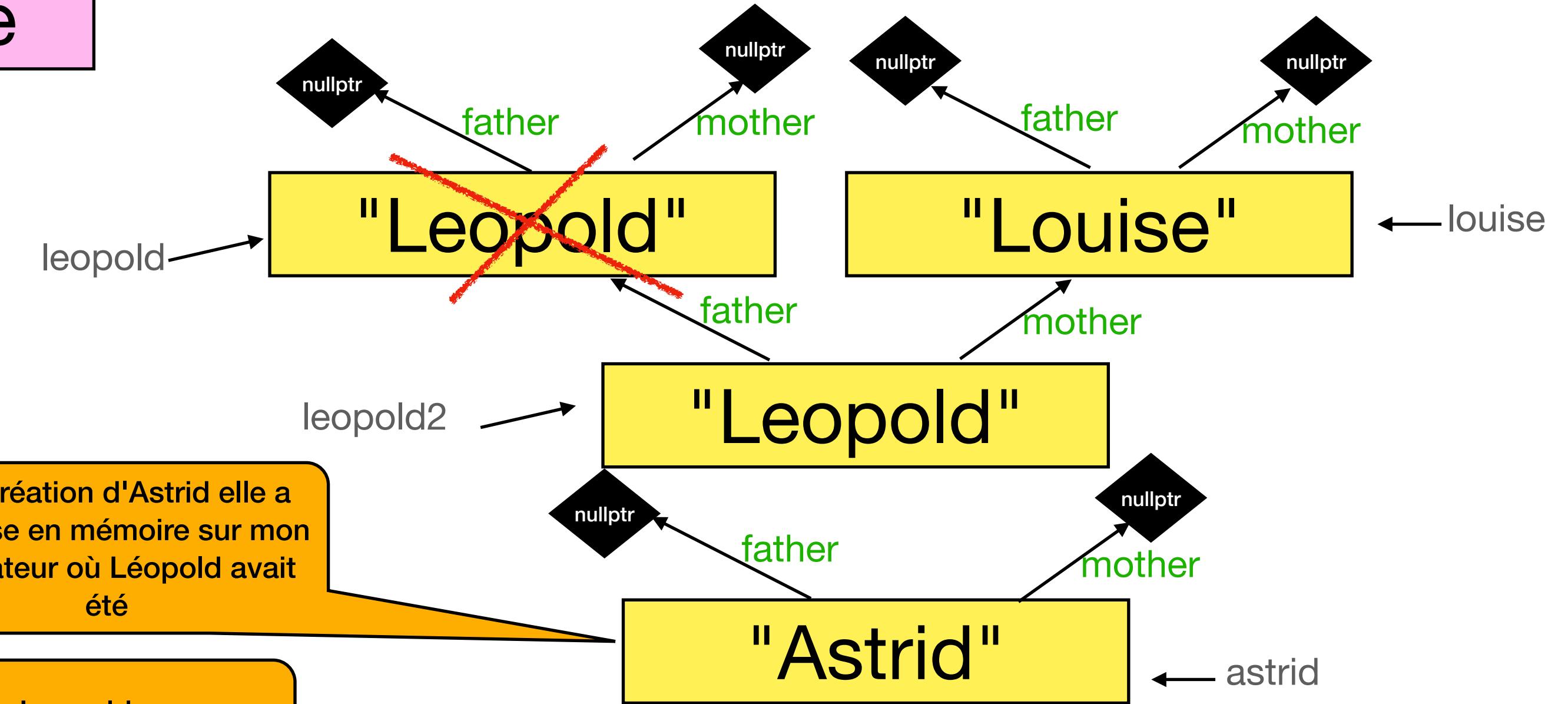
Ce code se compile mais plante lorsqu'il est exécuté

```
dynamic(22793,0x11dfa7e00) malloc: ***
error for object 0x7fd778c097b0: pointer
being freed was not allocated
```

Erreurs avec la mémoire dynamique 3

```
class person {  
    string name;  
    person *mother,*father;  
  
public:  
    person(string name,person *mother=nullptr,person  
*father=nullptr):name(name),mother(mother),father(father){;}  
    ~person(){delete mother;delete father;}  
    person *getMother() {return mother;}  
    person *getFather() {return father;}  
    string getName() {return name;}  
};  
  
void bad3(){  
    person *leopold=new person("Leopold");  
    person *louise=new person("Louise");  
    person *leopold2=new person("Leopold",louise,leopold);  
    cout<<leopold2->getFather()->getName()<<endl;  
    delete leopold;  
    cout<<leopold2->getFather()->getName()<<endl;  
    person *astrid=new person("Astrid");  
    cout<<leopold2->getFather()->getName()<<endl;  
}
```

Traçons-le



Problème : il y avait deux pointeurs vers Léopold. Lorsque `delete` a été appelée sur l'un, l'autre pointeur était toujours là

Pointeur suspendu : utiliser un pointeur vers quelque chose qui n'existe plus. Pourrait planter ou accéder/corrompre des données non liées

Remarques finales

- La mémoire dynamique vous donne un contrôle total sur la durée de vie d'un objet en utilisant `new` et `delete`
- Cela ouvre la possibilité à de nombreuses erreurs
- Nous verrons plus tard des "pointeurs intelligents" introduits en C++11 qui aident à éliminer de nombreuses erreurs courantes. Mais vous devez d'abord comprendre `new` and `delete` avec des pointeurs réguliers
- Nous discuterons de l'utilisation de `new` et `delete` pour créer plusieurs éléments à la fois dans une leçon ultérieure
- `new` alloue de la mémoire **ET** appelle le constructeur. `delete` libère la mémoire **ET** appelle le destructeur. Dans certains cas, il peut être nécessaire de séparer ces deux étapes, et C++ permet cela:
 - `malloc` et `free` proviennent de C et allouent et désallouent de la mémoire sans appeler aucun constructeur
 - La syntaxe : `new (p) string{"hello"}` n'alloue pas de mémoire et utilise la mémoire à l'emplacement `p` pour "placer" la chaîne et appeler le constructeur de la classe `string`. C'est ce qu'on appelle le « *placement new* ».
 - Ceci est utile dans les rares cas où vous souhaitez préalloquer une partie de la mémoire pour stocker certains objets de manière contiguë avec de l'espace libre pour les objets que vous ne souhaitez pas encore construire. La classe `vector` fait cela, par exemple.

Exemple de pile

Programmation orientée objet

Prof. John Iacono

Exemple de pile

- Le but de cet exemple est de rassembler des objets, mémoire dynamique, abstraction, encapsulation en un seul exemple non trivial
- À la fin de cette leçon, il aura des fonctionnalités de base, mais nous utiliserons la pile comme exemple courant dans les prochaines leçons pour ajouter plus de fonctionnalités et traiter plus de nuances quand on les apprend
- Je vais essayer de montrer le processus de conception
- Ouvrez le code sur votre ordinateur. Je vais essayer d'avoir les parties pertinentes sur chaque diapositive, mais veuillez faire une pause et regarder l'ensemble du code lorsque vous en aurez besoin.

Abstraction

- Lors du codage d'une classe, il existe deux niveaux de conception :
 - Quelles sont les méthodes et comportements souhaités selon l'expérience de l'utilisateur de la classe
 - Comment allons-nous implémenter les méthodes et stocker les données dans la classe
- Ces deux niveaux doivent rester aussi indépendants que possible. L'un des principaux principes de conception orientée objet est l'**abstraction** où les détails d'implémentation de la classe n'ont pas besoin d'être connus par l'utilisateur de la classe
- Plan:
 - Décrire à un niveau élevé ce qu'une pile devrait faire
 - Concevoir les signatures de fonction pour la partie publique de la classe
 - Décider à un niveau élevé de la mise en œuvre de la pile
 - Décidez comment les données seront stockées en interne et quelles seront les variables d'instance
 - Codez chaque méthode

Utiliser une pile

- Une pile stocke des objets et a deux opérations principales : push et pop.
 - **Push** : ajouter un élément en haut de la pile
 - **Pop** : supprimer (et retourner l'élément en haut de la pile)
- Nous allons mettre en œuvre deux autres opérations
 - **isEmpty** : retourne si la pile est vide
 - **top** : renvoie le premier élément de la pile sans le supprimer
- Nous allons coder une pile de chaînes (plus tard, nous le généraliserons)
- L'utilisateur doit s'assurer que pop et top ne sont pas appelés sur une pile vide ; si c'est le cas, le programme peut planter. Plus tard, nous discuterons des exceptions, qui sont une meilleure façon de gérer cela.

Codage de l'interface vers une pile

- Nous allons stocker des copies des chaînes
- Assez facile.... push, pop, top, isEmpty

```
class stringStack{  
public:  
    void push(const string &s);  
    string pop();  
    string top();  
    bool isEmpty();  
};
```

Les chaînes peuvent être grandes et donc une référence constante est probablement un meilleur choix que le passage par valeur

```
void testStringStack(){  
    stringStack s;  
    cout<<"isEmpty: "<<s.isEmpty()<<endl;  
    cout<<"Pushing First"<<endl;  
    s.push("First");  
    cout<<"Pushing Second"<<endl;  
    s.push("Second");  
    cout<<"Pushing Third"<<endl;  
    s.push("Third");  
    cout<<"Top: "<<s.top()<<endl;  
    cout<<"Popping: "<<s.pop()<<endl;  
    cout<<"Pushing Fourth"<<endl;  
    s.push("Fourth");  
    cout<<"Popping: "<<s.pop()<<endl;  
    cout<<"Popping: "<<s.pop()<<endl;  
    cout<<"isEmpty: "<<s.isEmpty()<<endl;  
}
```

```
isEmpty: 1  
Pushing First  
Pushing Second  
Pushing Third  
Top: Third  
Popping: Third  
Pushing Fourth  
Popping: Fourth  
Popping: Second  
isEmpty: 0
```

Comment stockons-nous les données ?

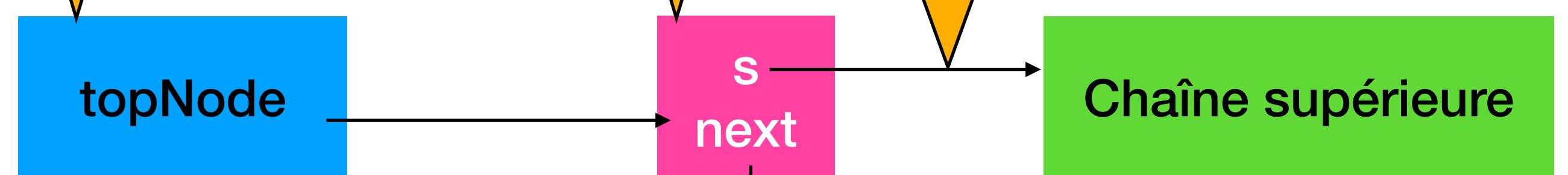
- Nous n'allons pas utiliser de tableaux (ou de vecteurs), que nous aborderons plus tard.
- Ainsi, nos classes ne peuvent avoir qu'un nombre constant de variables d'instance, chacune pouvant stocker un nombre constant de choses
- Mais une pile peut stocker beaucoup de choses !
- Solution : chaque instance d'une pile doit avoir plusieurs instances de quelque chose... mais quoi ?

Concevoir comment stocker les données

Qu'en est-il de notre classe stringStack ? De quoi avons-nous besoin pour accéder à la structure ? Juste le nœud supérieur. Cela peut ne pas

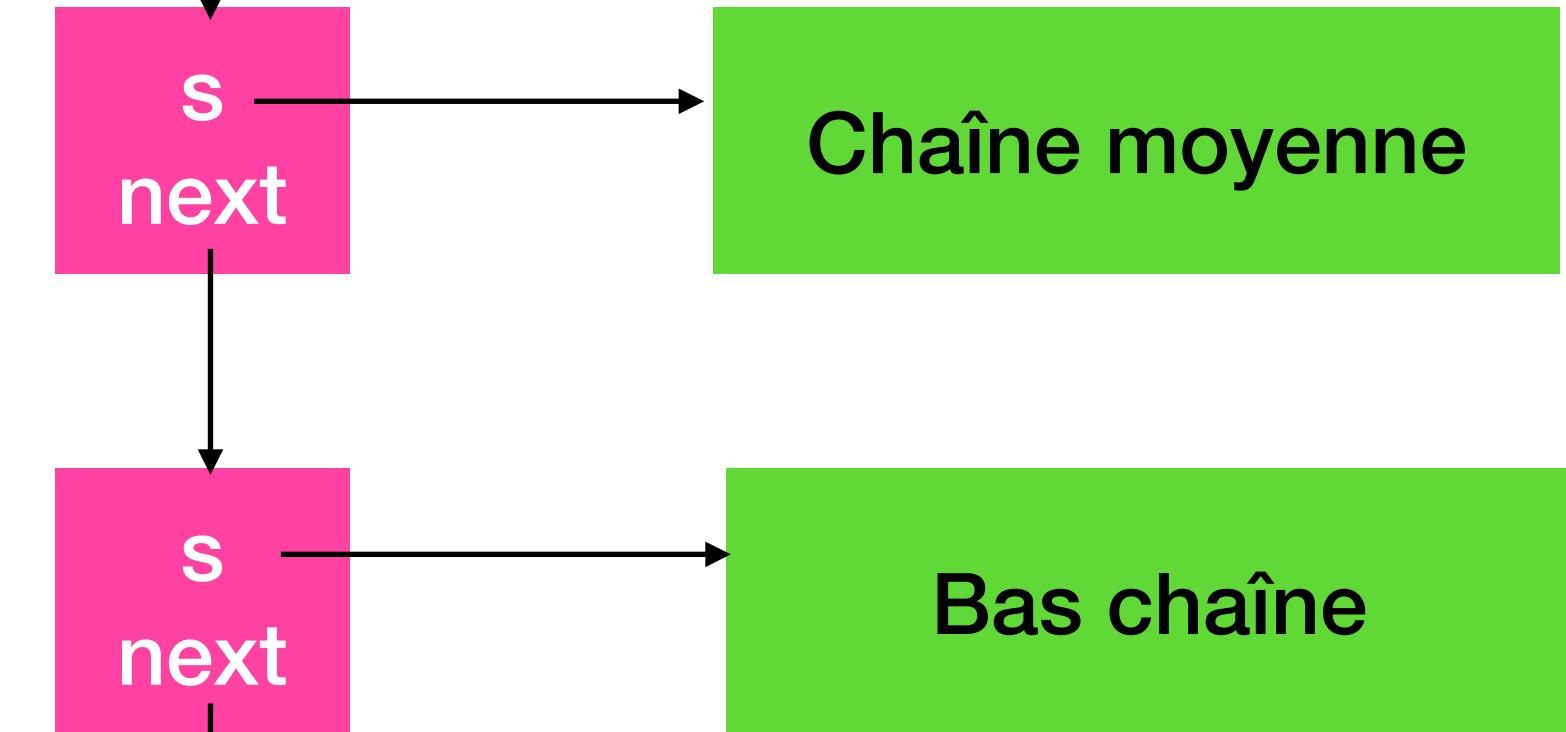
Ces cases sont-elles une chaîne Stack ? Non, c'est autre chose. Nous pouvons les définir. Appelons l'un d'eux un nœud

Chaque s a-t-il une flèche vers une case de la même couleur ? Oui. Donc une variable d'instance ordinaire est assez bonne



Cas extrême : structure vide

topNode



Types de données

string

node

stringStack

```
struct node {  
    string s;  
    node* next;  
};
```

```
class stringStack {  
    node* topNode;  
    ...
```

```
class stringStack{  
    struct node {  
        string s;  
        node* next;  
    };  
    node *topNode;  
public:  
    void push(const string &s)  
    string pop();  
    string top();  
    bool isEmpty();  
};
```

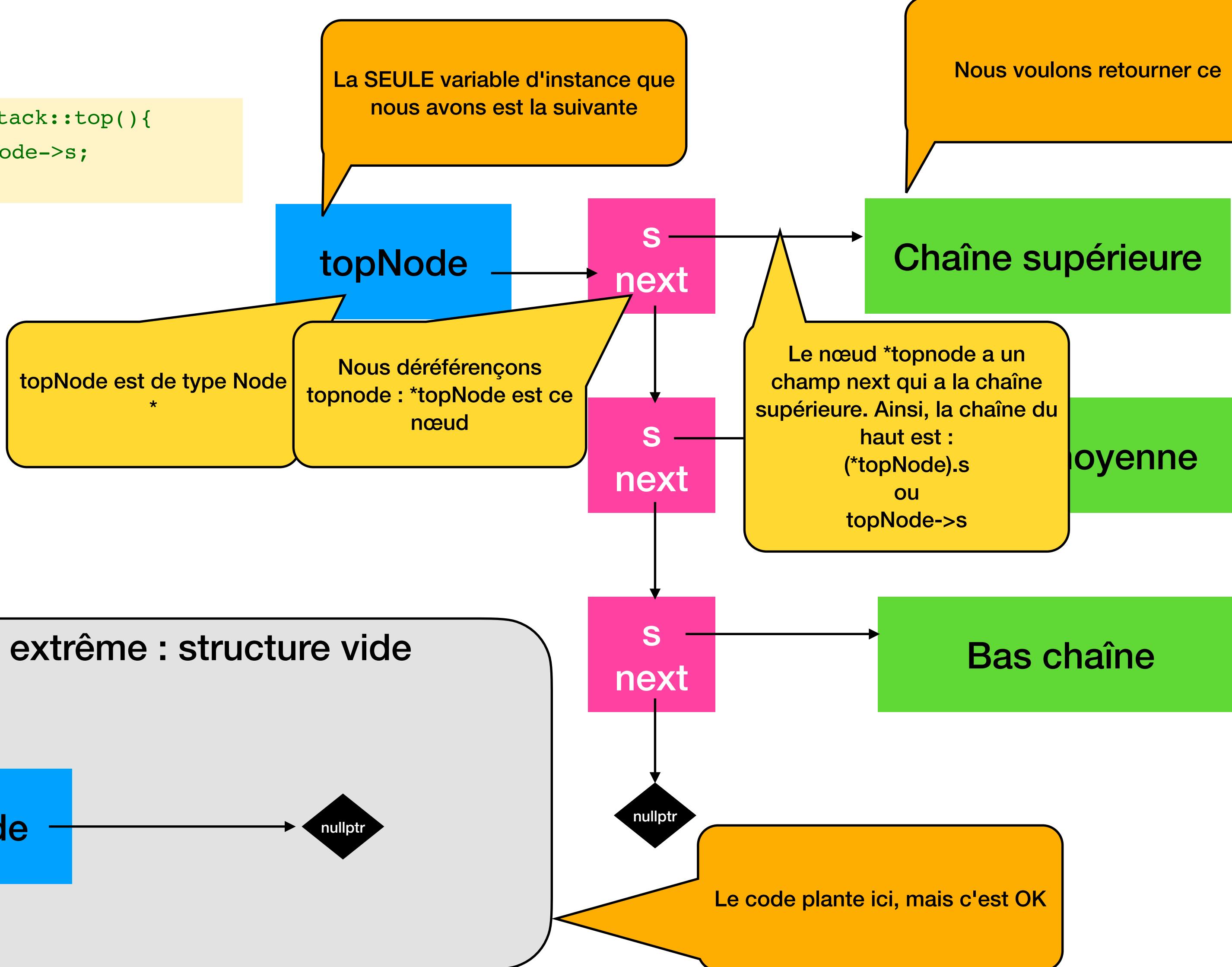
Nous mettons la structure de nœud à l'intérieur de stringStack. Encapsulation ! Vous pouvez placer des classes à l'intérieur d'autres classes, et vous devriez le faire lorsque la classe interne n'est utile qu'à

topNode est privé.
L'utilisateur de stringStack ne doit pas y toucher (ou se soucier

Il ne nous reste plus qu'à implémenter les méthodes

Next: top

```
string stringStack::top() {
    return topNode->s;
}
```



Data Types

string

node

stringStack

```
struct node {
    string s;
    node* next;
};
```

```
class stringStack {
    node* topNode;
    ...
}
```

```
class stringStack{
    struct node {
        string s;
        node* next;
    };
    node *topNode;
public:
    void push(const string &s);
    string pop();
    string top();
    bool isEmpty();
};
```

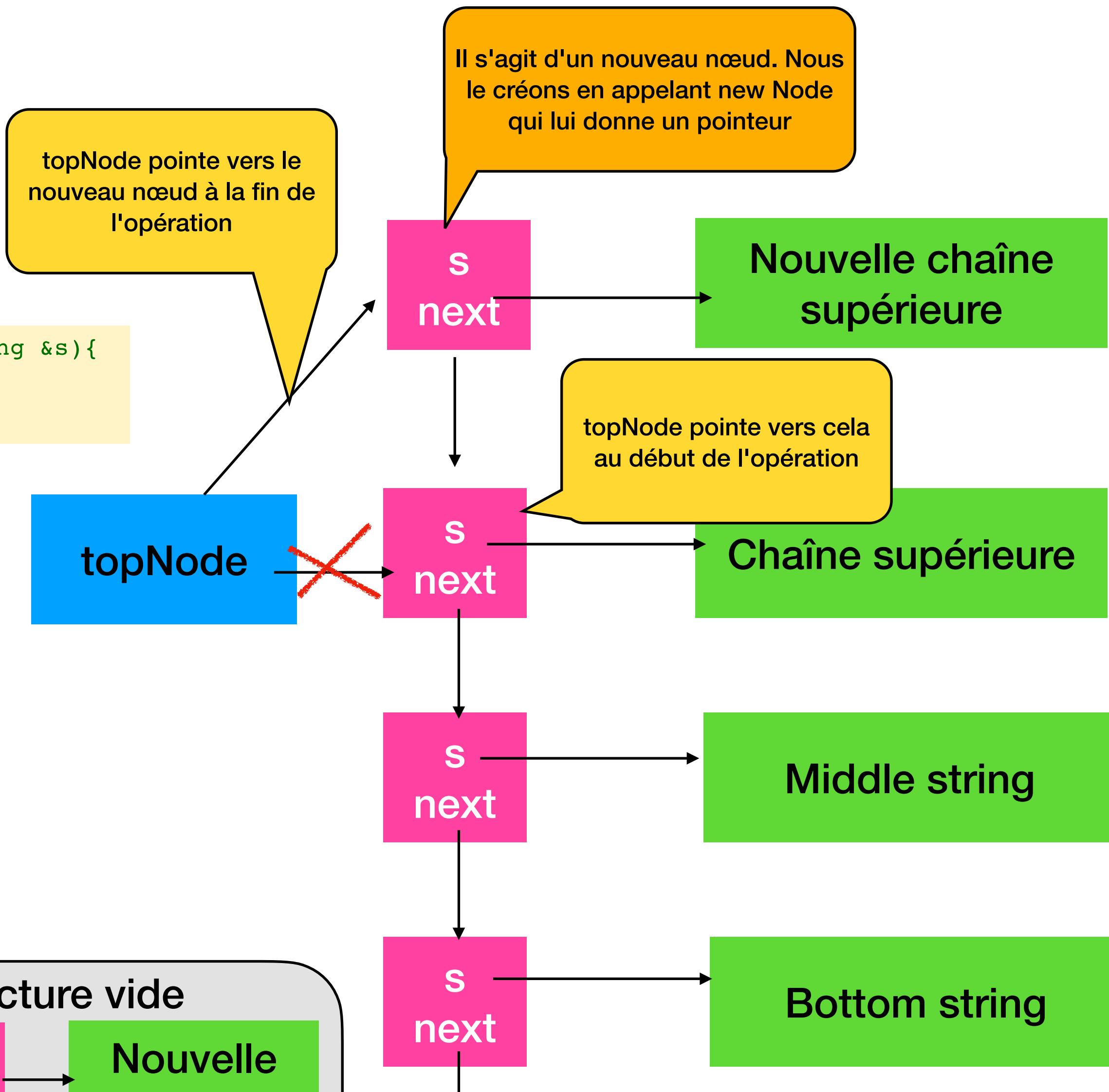
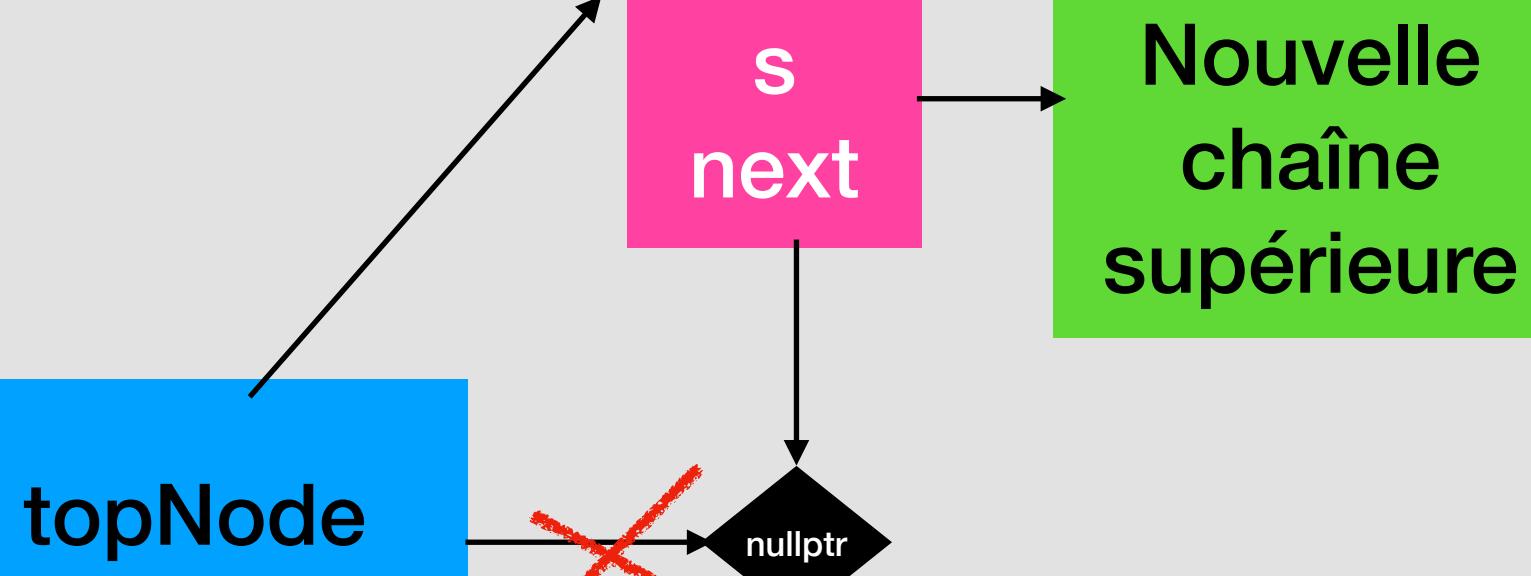
Push

```
void stringStack::push(const string &s){
    topNode=new node{s,topNode};
}
```

La chaîne du nouveau nœud est la chaîne donnée par l'utilisateur

Le nœud next du nouveau nœud est l'ancien topNode

Cas extrême : structure vide



Types de données

string

node

stringStack

```
struct node {
    string s;
    node* next;
};
```

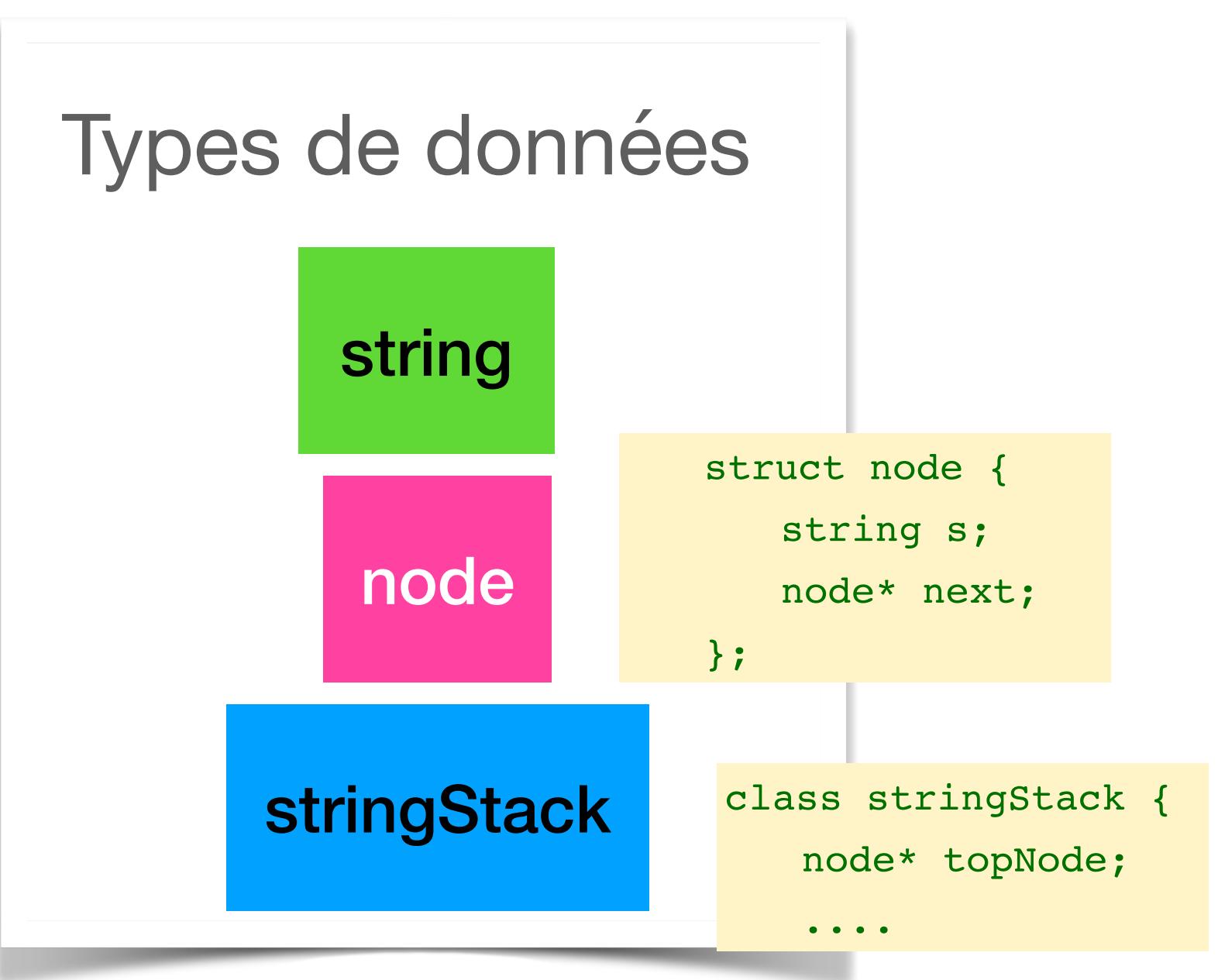
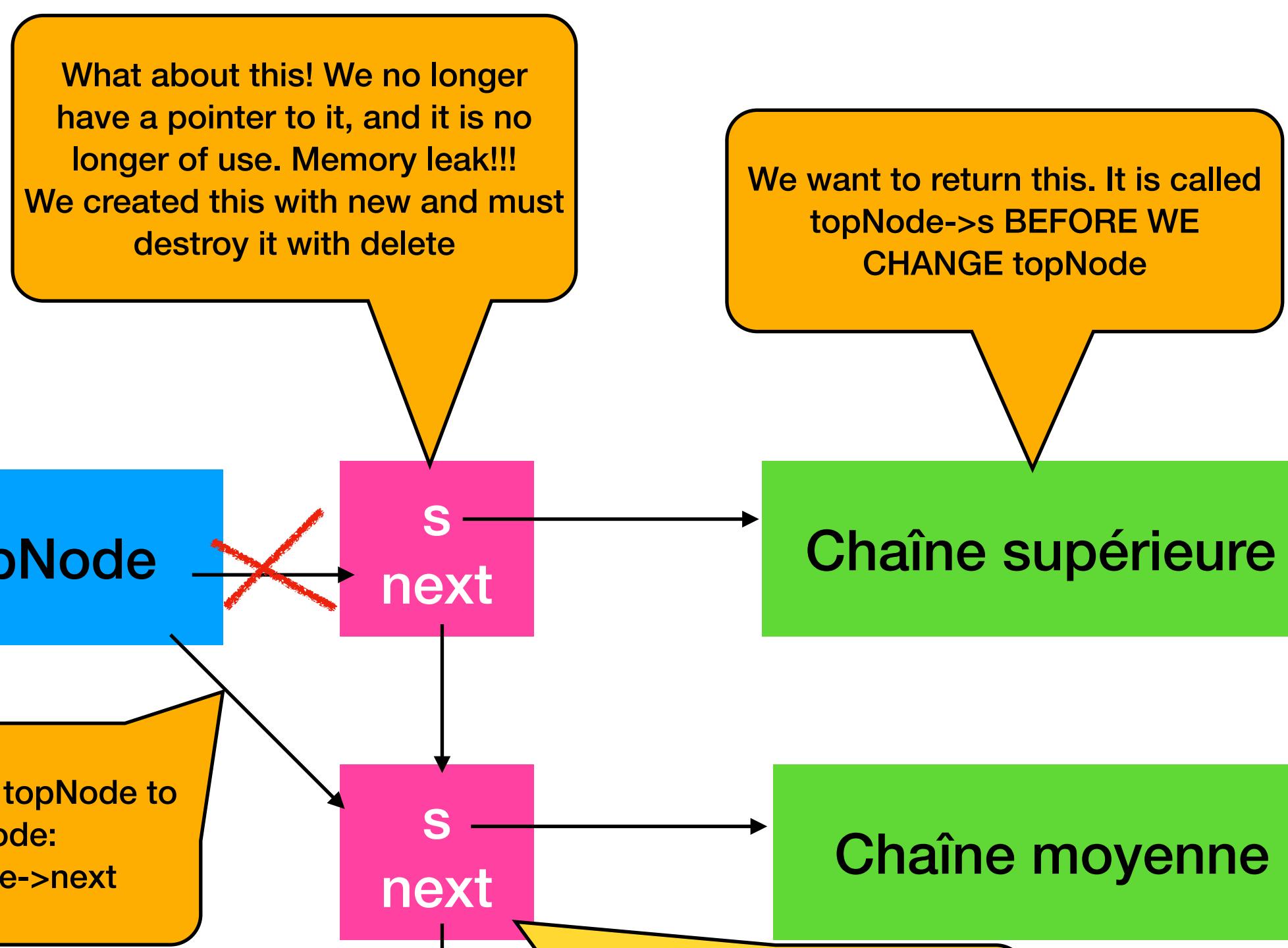
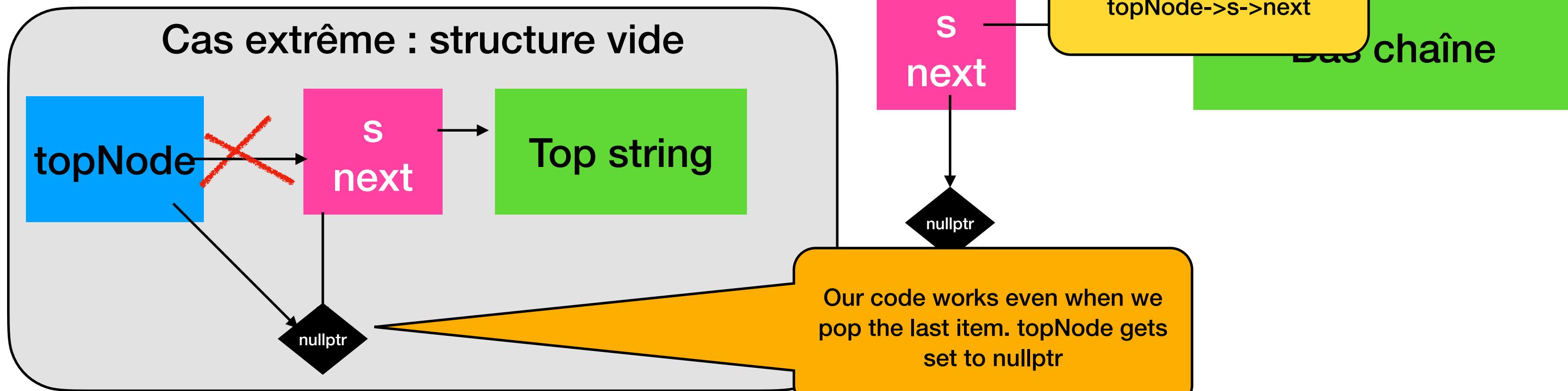
```
class stringStack {
    node* topNode;
    ...
}
```

```
class stringStack{
    struct node {
        string s;
        node* next;
    };
    node *topNode;
public:
    void push(const string &s);
    string pop();
    string top();
    bool isEmpty();
};
```

Next: pop

```
string stringStack::pop(){
    string toReturn{topNode->s};
    topNode=topNode->next;
    return toReturn;
}
```

```
string stringStack::pop(){
    string toReturn{topNode->s};
    node *oldTopNode=topNode;
    topNode=topNode->next;
    delete oldTopNode;
    return toReturn;
}
```



```

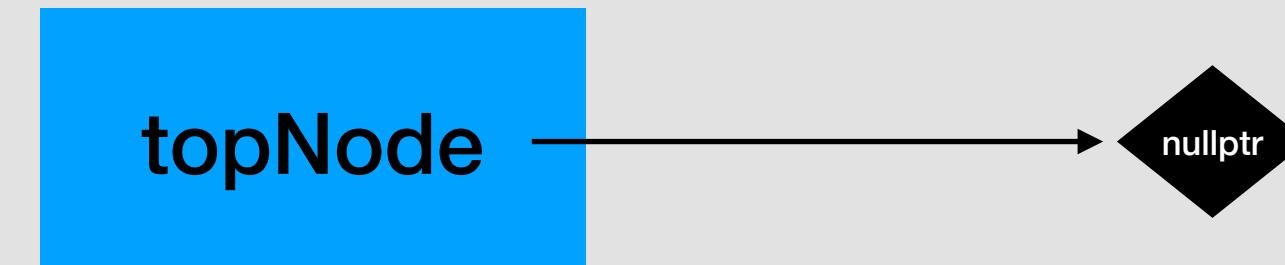
class stringStack{
    struct node {
        string s;
        node* next;
    };
    node *topNode;
public:
    void push(const string &s);
    string pop();
    string top();
    bool isEmpty();
};

```

Constructor?

- Besoin d'initialiser topNode à nullptr

Cas extrême : structure vide



```
stringStack::stringStack( ):topNode{nullptr} {}
```

Destructeur?

- Nous en avons besoin! Tous les nœuds créés doivent être détruits, sinon il y a une fuite de mémoire
- Nous avons déjà du code pour supprimer un seul élément : pop
- Ne réinventez pas la roue. La réutilisation du code est une partie importante de la programmation orientée objet. Cela isole également les erreurs.

```
stringStack::~stringStack() {
    while (!isEmpty()) pop();
}
```

```
stringStack::~stringStack() {
    while (topNode) {
        node *oldTopNode=topNode;
        topNode=topNode->next;
        delete oldTopNode;
    }
}
```

Code : tout sur une seule diapositive

Définition de classe

```
class stringStack{
    struct node {
        string s;
        node* next;
    };
    node *topNode;
public:
    stringStack();
    ~stringStack();
    void push(const string &s);
    string pop();
    string top();
    bool isEmpty();
};
```

Constructeur et destructeur

```
stringStack::stringStack():topNode{nullptr}{}

stringStack::~stringStack() {
    while (!isEmpty()) pop();
}
```

Autres méthodes: push,
pop, top, isEmpty

```
void stringStack::push(const string &s){
    topNode=new node{s,topNode};
}

string stringStack::pop(){
    string toReturn{topNode->s};
    node *oldTopNode=topNode;
    topNode=topNode->next;
    delete oldTopNode;
    return toReturn;
}

string stringStack::top(){
    return topNode->s;
}

bool stringStack::isEmpty() {
    return topNode==nullptr;
}
```

Exercices

- Ajoutez une méthode `size()` à la classe de pile qui renvoie le nombre de chaînes actuellement stockées. Codez ceci de deux manières :
 - Sans ajouter de variables d'instance supplémentaires à `stringStack` (plus difficile)
 - En ajoutant un entier en tant que variable d'instance privée de `stringStack`
- Quels sont les avantages et les inconvénients de chaque approche ?
- Écrivez une méthode `toString` qui renvoie le contenu de la pile bien formaté :
["top", "first", "second"]
- Écrivez un constructeur qui prend une liste d'initialisation contenant des chaînes et initialise la pile avec ces chaînes :

```
stringStack s{"bottom", "middle", "top"};
```

Le constructeur de copie et l'opérateur d'assignation

Programmation orientée objet

Prof. John Iacono

Que se passe-t-il lorsque nous copions une pile ?

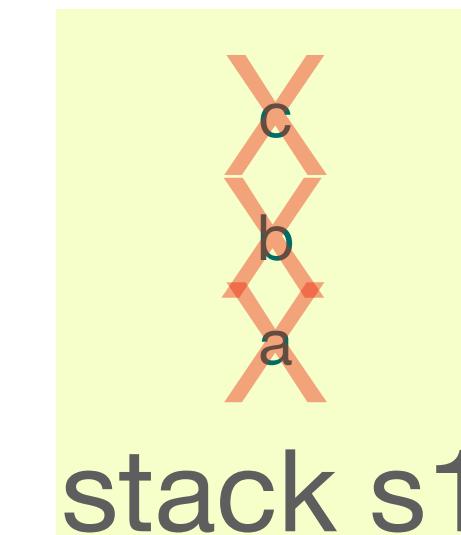
Notre classe de pile simple avec push, pop, top, isEmpty, constructeur et destructeur

```
class stringStack{  
    struct node {  
        string s;  
        node* next;  
        node(const string &s, node *next):s(s),next(next){;}  
    };  
    node *topnode=nullptr;  
public:  
    void push(const string &s){topnode=new node(s,topnode);};  
    string pop(){  
        string s{topnode->s};  
        node *oldtopnode=topnode;  
        topnode=topnode->next;  
        delete oldtopnode;  
        return s;  
    }  
    string &top(){return topnode->s;}  
    bool isEmpty() {return topnode==nullptr;}  
    ~stringStack() {while (!isEmpty()) pop();}  
};
```

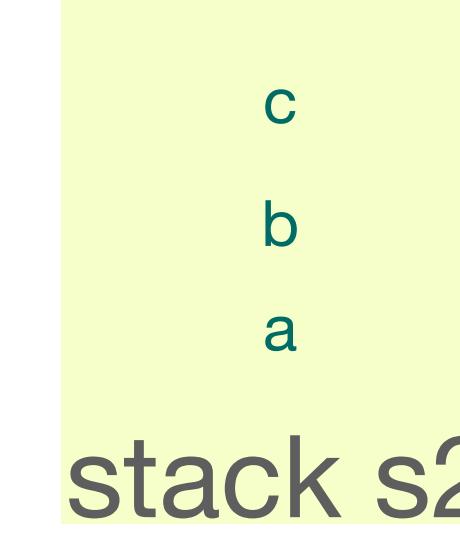
Un constructeur qui prend une référence à un objet du même type est appelé un **constructeur de copie**

C++ fait ce code pour nous :

```
stringStack(stringStack &other):topnode{other.topnode}{}  
C'est ce qu'on appelle le constructeur de copie par défaut  
Il copie simplement toutes les variables membres
```



stack s1



stack s2

```
void demo1(){  
    stringStack s1;  
    s1.push("a");  
    s1.push("b");  
    s1.push("c");  
    stringStack s2=s1;  
    cout<<"s1.top():"<<s1.top()<<" s2.top():"<<s2.top()<<endl; s1.top():c s2.top():c  
    s1.pop();  
    cout<<"s1.top():"<<s1.top()<<" s2.top():"<<s2.top()<<endl; s1.top():b s2.top():c  
    s1.pop();  
    cout<<"s1.top():"<<s1.top()<<" s2.top():"<<s2.top()<<endl; s1.top():a s2.top():a  
    s1.pop();  
    cout<<"s1.isEmpty():"<<s1.isEmpty()<<" s2.isEmpty():"<<s2.isEmpty()<<endl;  
}
```

Que se passe t-il ici? Supposons que s2 est une copie de s1

```
copy-constructor(48499,0x108f21e00) malloc: *** error for object  
0x7fc5ec405970: pointer being freed was not allocated  
copy-constructor(48499,0x108f21e00) malloc: *** set a breakpoint in  
malloc_error_break to debug  
Abort trap: 6
```

HUH?

Ok, nous ne comprenons pas la copie

???

Que se passe-t-il lorsque nous copions une pile ?

Types de données

```

class stringStack{
    struct node {
        string s;
        node* next;
        node(const string &s,node *next):s(s),next(next){;}
    };
    node *topnode=nullptr;
public:
    void push(const string &s){topnode=new node(s,topnode);}
    string pop(){
        string s{topnode->s};
        node *oldtopnode=topnode;
        topnode=topnode->next;
        delete oldtopnode;
        return s;
    }
    string &top(){return topnode->s;}
    bool isEmpty() {return topnode==nullptr;}
    ~stringStack() {while (!isEmpty()) pop();}
};

```

Le constructeur de copie par défaut ne fonctionne pas pour notre classe de pile !!

```

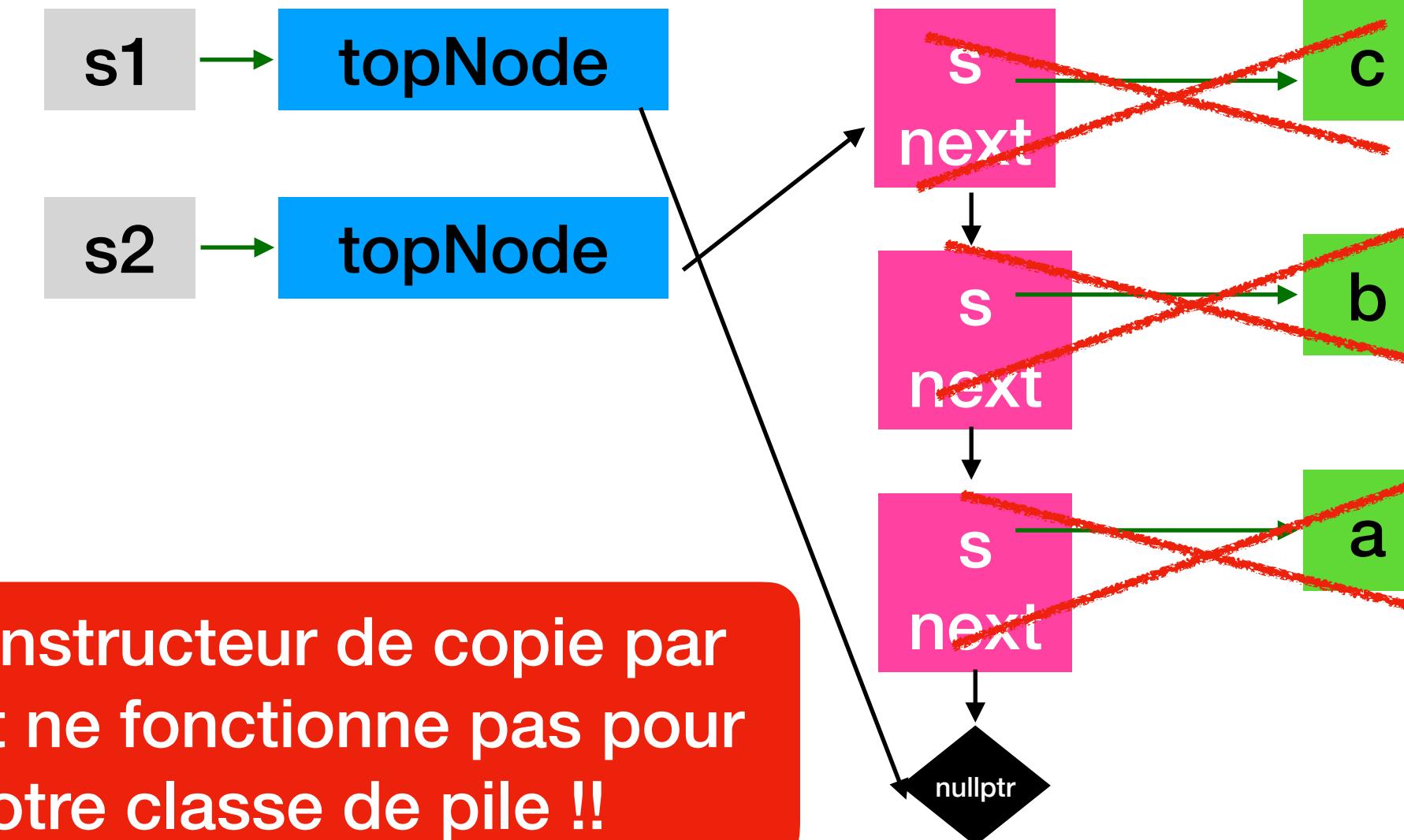
copy-
constructor(48499,0x108f21e00)
malloc: *** error for object
0x7fc5ec405970: pointer being
freed was not allocated
copy-
constructor(48499,0x108f21e00)
malloc: *** set a breakpoint
in malloc_error_break to debug
Abort trap: 6

```

A constructor that takes a reference to an object of the same type is called a **copy constructor**

Si vous n'écrivez pas de constructeur de copie,
C++ en fait un pour vous comme celui-ci qui appelle simplement
le constructeur de copie sur toutes les variables membres
(qui copie simplement les types intégrés) :

```
stringStack(stringStack &other):topnode{other.topnode}{}}
```



```

void demo1(){
    stringStack s1;
    s1.push("a");
    s1.push("b");
    s1.push("c");
    stringStack s2=s1;
    cout<<"s1.top():"<<s1.top()<<" s2.top():"<<s2.top()<<endl; s1.top():c s2.top():c
    s1.pop();
    cout<<"s1.top():"<<s1.top()<<" s2.top():"<<s2.top()<<endl; s1.top():b s2.top():c
    s1.pop();
    cout<<"s1.top():"<<s1.top()<<" s2.top():"<<s2.top()<<endl; s1.top():a s2.top():
    s1.pop();
    cout<<"s1.isEmpty():"<<s1.isEmpty()<<" s2.isEmpty():"<<s2.isEmpty()<<endl;
    s1.isEmpty():1 s2.isEmpty():0
}

```

Cela appelle le constructeur **stringStack(s1)** pour initialiser s2

Créer un constructeur de copie

Nous utiliserons une pile temporaire et push/pop pour copier

```
class stringStack{
    struct node {
        string s;
        node* next;
        node(const string &s,node *next):s(s),next(next){;}
    };
    node *topnode=nullptr;
public:
```

```
    void push(const string &s);
```

```
    string pop();
```

```
    string &top(){return topnode->s;}
```

```
    bool isEmpty() {return topnode==nullptr;}
```

```
-stringStack() {while (!isEmpty()) pop();}
```

```
stringStack();
```

```
stringStack(stringStack &other);
```

```
};
```

```
→ stringStack::stringStack(stringStack &other){
```

```
    stringStack tmp;
```

```
    while (!other.isEmpty())
        tmp.push(other.pop());
```

```
    while (!tmp.isEmpty()){
        auto s=tmp.pop();
```

```
        push(s);
        other.push(s);
    }
```

Pour désencombrer, toutes les méthodes avec des corps sur plus d'une ligne ont des implémentations en dehors de la classe (et cette diapositive)

En C++11, vous pouvez utiliser `auto` au lieu d'un nom de type si le compilateur peut le comprendre. Ici, `pop()` renvoie une chaîne, c'est donc le type de `s`

b
a
stack other

c
stack *this

stack tmp

```
void demo(){
    → stringStack s1;
    → s1.push("a");
    → s1.push("b");
    → s1.push("c");
    → stringStack s2=s1;
```

Maintenant, notre constructeur de copie s'appelle

```
cout<<"s1.top():"<<s1.top()<<" s2.top():"<<s2.top()<<endl;
s1.pop();
cout<<"s1.top():"<<s1.top()<<" s2.top():"<<s2.top()<<endl;
s1.pop();
cout<<"s1.top():"<<s1.top()<<" s2.top():"<<s2.top()<<endl;
s1.pop();
cout<<"s1.isEmpty():"<<s1.isEmpty()<<" s2.isEmpty():"<<s2.isEmpty()<<endl;
```

```
}
```

Et pour les fractions ?

```
class Fraction{  
    int n;  
    int d;  
    void reduce();  
  
public:  
    Fraction(int n,int d);  
    string toString();  
    Fraction &addTo(const Fraction &f2);  
    Fraction &operator +=(const Fraction &f2);  
    Fraction operator +(const Fraction &f2);  
};
```

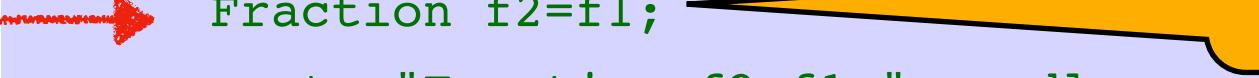
Si vous n'écrivez pas de constructeur de copie, C++ en fait un pour vous comme celui-ci qui appelle simplement le constructeur de copie sur toutes les variables membres (qui copie simplement les types intégrés) :

```
Fraction(Fraction &other):n{other.n},d{other.d}{}{}
```

En général, le constructeur de copie par défaut fourni par C++ est généralement tout ce dont vous avez besoin si vous n'avez pas de pointeurs

Vous avez presque toujours besoin d'un constructeur de copie personnalisé si vous utilisez des pointeurs et de la mémoire dynamique

Il y a, bien sûr, des exceptions. Vous devez vous demander s'il faut coder ou utiliser les constructeurs de copie par défaut pour chaque classe que vous écrivez

```
Fraction f1{5,7}; // Makes 5/7  
cout<<"Fraction f1:"<<f1.toString()<<endl;  
→ Fraction f2=f1;   
cout<<"Fraction f2=f1;"<<endl;  
cout<<"Fraction f1:"<<f1.toString()<<" Fraction f2:"<<f2.toString()<<endl;  
f2.addTo(f1);  
cout<<"f2.addTo(f1);"<<endl;  
cout<<"Fraction f1:"<<f1.toString()<<" Fraction f2:"<<f2.toString()<<endl;
```

Que se passe t-il ici?

Fraction f1:5/7
Fraction f2=f1;
Fraction f1:5/7 Fraction
f2:5/7
f2.addTo(f1);
Fraction f1:5/7 Fraction
f2:10/7

Est-ce correct??

Oui !

Constructeur de copie vs opérateur d'assignation : opérateur =

- stringStack s1; // Constructeur par défaut
- stringStack s2{s1}; // Constructeur de copie
- stringStack s3=s2; //Constructeur de copie
- Passe-par-valeur: //Le constructeur de copie est appellé avec la valeur de s1
- g2=g1; // operator =

g2 existe déjà !

Argh, nous devons coder autre chose pour créer la classe
stringStack

s est créé à l'aide du constructeur de copie appelé
avec le paramètre k

```
int stackCount(stringStack s) {  
    int count=0;  
    while (!s.isEmpty()) {s.pop();count++;}  
    return count  
}
```

```
stringStack k; k.push("a"); k.push("b");  
cout<<stackCount(k);
```

s est vide ici

Que se passe t-il ici?

k reste inchangé

Ajout de l'opérateur = à stringStack

```
class stringStack {  
    .....  
    stringStack(){  
        .....  
    }  
    stringStack(stringStack &other){  
        .....  
        constructeur de copie  
        stringStack tmp;  
        while (!other.isEmpty())  
            tmp.push(other.pop());  
        while (!tmp.isEmpty()){  
            auto s=tmp.pop();  
            push(s);  
            other.push(s);  
        }  
        opérateur =  
    }  
}
```

```
stringStack &operator=(stringStack &other){  
    .....  
    while (!isEmpty()) pop();  
    stringStack tmp;  
    while (!other.isEmpty())  
        tmp.push(other.pop());  
    while (!tmp.isEmpty()){  
        auto s=tmp.pop();  
        push(s);  
        other.push(s);  
    }  
    return *this;  
}  
~stringStack() {while (!isEmpty()) pop();}
```

Ne répétez pas le code !!!

Copier les éléments de `other`

Me vider

Copier les éléments de `other`

Renvoyer une référence à moi-même

Me vider

```
class stringStack {  
    .....  
    void makeEmpty(){while (!isEmpty()) pop();}  
    void copyFromOtherStack(stringStack &other){  
        .....  
        stringStack tmp;  
        while (!other.isEmpty())  
            tmp.push(other.pop());  
        while (!tmp.isEmpty()){  
            auto s=tmp.pop();  
            push(s);  
            other.push(s);  
        }  
    }
```

```
public:  
    ~stringStack() {makeEmpty();}  
    stringStack(){  
        .....  
    }  
    stringStack(stringStack &other){  
        copyFromOtherStack(other);  
    }  
    stringStack &operator=(stringStack &other){  
        makeEmpty();  
        copyFromOtherStack(other);  
        return *this;  
    }  
    ....
```

Bien mieux !!!!

Les trois grands

Copier le constructeur, l'assignation, le destructeur

- C++ fournit des versions des trois pour vous
 - Constructeur de copie : copie toutes les variables d'instance
 - Opérateur d'assignment : assignment sur toutes les variables d'instance
 - Destructeur : destructor sur toutes les variables d'instance
- Les versions défaut sont généralement correctes sauf si vous utilisez de la mémoire dynamique
- Si vous codez l'un de ces trois, vous devez généralement coder les trois !

Delete et default

Contrôle fin des méthodes fournies par C++

- `=delete`: Spécifie qu'une méthode que C++ fournit normalement doit être fournie.

```
struct C {  
    string s;  
    C(const C&)=delete;  
    C(string s):s{s}{};  
    C &operator = (const C&)=delete;  
};  
  
void testC(){  
    C x{"hello"};  
    C y{"world"};  
    C z=y; // ERROR!  
    y=x; //ERROR  
}  
  
deleteDefault.cpp:17:4: error:  
call to deleted constructor of  
'C'  
  
deleteDefault.cpp:11:5: note:  
candidate function has been  
explicitly deleted  
    C &operator = (const  
C&)=delete;
```

Copier le constructeur supprimé !

Opérateur = supprimé !

- `=default`: Spécifie qu'une méthode que C++ peut fournir doit être fournie

```
struct D {  
    string s;  
    D(string s):s{s}{};  
};  
  
struct E {  
    string s;  
    E(string s):s{s}{};  
    E()=default;  
};  
  
void testDE() {  
    D x{"Hello"};  
    D y; // ERROR  
    E z{"World"};  
    E w;  
}
```

Définir un constructeur signifie normalement que C++ ne vous donne normalement pas de constructeur par

```
deleteDefault.cpp:35:5: error:  
no matching constructor for  
initialization of 'D'  
    D y; // ERROR  
  
deleteDefault.cpp:23:2: note:  
candidate constructor not  
viable: requires single  
argument 's', but no arguments  
were provided  
    D(string s):s{s}{};
```

mais vous pouvez le rajouter

Opérateurs et amis

Programmation orientée objet

Prof. John Iacono

Un opérateur n'est qu'une fonction

- Si j'écris

```
int x=3,y=6,z=9;  
x=y+z;
```

- `+` est juste une fonction qui prend deux entiers et renvoie un autre entier.
- La fonction s'appelle `operator+`
- Lorsque vous codez `x+y` où `x` est de type `X` et `y` est de type `Y`, cela fonctionne si l'un des deux suivants est défini :

```
Z operator+(X x,Y y); // Pas dans aucune classe
```

```
Z X::operator+(Y y); // Une méthode de classe X
```

- Dans les deux cas `Z` est le type de l'expression `x+y`

Ajouter des fractions

- Option 1 : Ajouter `operator +(Fraction other)` à l'intérieur de Fraction

```
class Fraction{  
    int n,d;  
    void reduce();  
public:  
    Fraction(int n,int d);  
    Fraction operator +(Fraction f2);  
    ...
```

```
Fraction Fraction::operator +(Fraction f2){  
    return Fraction{ n * f2.d + f2.n * d, d * f2.d};  
}  
Fraction f1{4,6};  
Fraction f2{1,10};  
cout<<string{f1}<<"+"<<string{f2}<<"="<<string{f1+f2}<<endl;
```

2/3+1/10=23/30

- Option 2: Ajouter `operator +(Fraction f1, Fraction f2)` en tant que fonction distincte

```
class Fraction{  
    int n,d;  
    void reduce();  
public:  
    Fraction(int n,int d);  
    ...
```

Adding this friend declaration inside the Fraction class gives this function access to the private parts of the class

```
friend Fraction operator +(Fraction f1,Fraction f2);
```

```
Fraction operator +(Fraction f1,Fraction f2){  
    return Fraction{ f1.n * f2.d + f2.n * f1.d, f1.d * f2.d};  
}
```

```
fractions.cpp: In function 'lesson_operator::Fraction lesson_operator::operator+(lesson_operator::Fraction, lesson_operator::Fraction)':  
fractions.cpp:577:37: error: 'int lesson_operator::Fraction::n' is private  
within this context  
577 |         return Fraction{ f1.n * f2.d + f2.n * f1.d, f1.d *  
| f2.d};  
|  
fractions.cpp:535:21: note: declared private here  
535 |         int n;
```

n and d are private, and this is an
function, not a method of F

Addition de fractions et d'entiers

- Pour additionner une fraction et un entier
 - Option 1: coder une méthode dans `Fraction`:
`Fraction Fraction::operator +(int i);`
 - Option 2: code une fonction pas dans une classe : (un ami si besoin)
`Fraction operator +(Fraction f, int i);`
- Pour additionner un entier et une fraction:
 - Option 1: coder une méthode dans un `int`
~~`Fraction int::operator +(Fraction f);`~~
 - Option 2: code une fonction pas dans une classe : (un ami si besoin)
`Fraction operator +(int i, Fraction f);`

on ne peut pas ajouter de méthode à `int` !

C'est la seule option lorsque nous ne pouvons pas ajouter de méthodes à la classe de `x` dans `x+y`

Addition de fractions et d'entiers

```
class Fraction{  
    . . .  
    Fraction operator +(Fraction f2);  
    Fraction operator +(int i);  
    friend Fraction operator +(int i, Fraction f);  
    . . .  
};
```

Nous verrons plus tard qu'en codant une conversion de `int` en `Fraction` seul le premier opérateur+ sera nécessaire

```
Fraction Fraction::operator +(Fraction f2){  
    return Fraction{ n * f2.d + f2.n * d, d * f2.d};  
}
```

Addition fraction+fraction (method)

```
Fraction Fraction::operator +(int i){  
    return Fraction{n + i * d, d};  
}
```

Addition fraction+int (method)

```
No Fraction::  
Fraction operator +(int i,Fraction f){  
    Fraction{f.n + i * f.d, f.d};  
}
```

Addition int+fraction (not a method, friend)

Comparaisons

Supposons que vous voulez permettre à < <= == != > >= de fonctionner pour notre classe Fraction

- Avant C++20 : Codez six méthodes :

```
bool Fraction::operator <(Fraction other)
```

- Puis, codez aussi : <=, ==, !=, >, >=
- C'est énervant
- Avec C++20 : codez deux méthodes (pour les cas courants)



- `<=>` est le *spaceship operator*
 - `x <=> y` doit être négatif si $x>y$, 0 si $x==y$ et positif si $x<y$
 - Il existe un certain nombre de types de retour possibles pour l'opérateur `<=>`, l'utilisation de auto couvre les cas courants.
 - Si vous codez l'opérateur du spaceship, C++ vous donne <, <=, >, >=
 - Si vous codez l'opérateur == C++ vous donne l'opérateur !=
 - Un contrôle fin est possible avec `=default` (C++ crée une fonction qui effectue la comparaison sur toutes les variables membres) et `=delete` (supprime une fonction de comparaison que C++ ferait normalement pour vous)

Ajout de comparaisons à notre classe de fraction dans C++20

Codez opérateur `<=>`
et opérateur `==`

```
auto Fraction::operator <=>(Fraction other){  
    return n*other.d <=> other.n*d;  
}  
  
bool Fraction::operator ==(Fraction other){  
    return n*other.d == other.n*d;  
}
```

```
f1=Fraction{1,3};  
f2=Fraction{5,13};  
cout<<"f1:"<<string{f1}<<endl;  
cout<<"f2:"<<string{f2}<<endl;  
cout<<"f1<f2:"<< (f1<f2) <<endl;  
cout<<"f1<=f2:"<< (f1<=f2) <<endl;  
cout<<"f1==f2:"<< (f1==f2) <<endl;  
cout<<"f1!=f2:"<< (f1!=f2) <<endl;  
cout<<"f1>=f2:"<< (f1>=f2) <<endl;  
cout<<"f1>f2:"<< (f1>f2) <<endl;
```

f1:1/3
f2:5/13
f1<f2:1
f1<=f2:1
f1==f2:0
f1!=f2:1
f1>=f2:0
f1>f2:0

- Pourrait plutôt dire à C++ que nous voulons utiliser l'opérateur par défaut `==` :
`opérateur booléen ==(const Fraction &other) const=default;`
- Cela vérifiera l'égalité de `n` et `d`, ce qui est OK puisque nous stockons toujours la fraction dans les termes les plus bas.
- Doit utiliser pass-by-constant-reference et marquer la méthode comme `const` (voir la leçon sur `const`)
- Nous ne voulons pas de l'opérateur `<=>` par défaut, cela appellera simplement `<=>` sur `n`, puis `d` si les `n` sont égaux, ce qui n'est pas la façon dont nous comparons les fractions !

Quels opérateurs peuvent être surchargés

- De nombreux! Les plus courants incluent :
 - Arithmétique: + - * % ^ | & ~ << >>
 - Incrémenter/décrémenter: `++ --`
 - Logique: ! && ||
 - Assignation et assignation composée:
`= += -= *= /= %= &= |= <<= >>=`
 - Comparaison == != < > >= <= <=>
 - Appel de fonction: `X(a,b,c)`
 - Indice `X[a]`
 - Conversion de type `(int)X`
`static_cast<int>(X)`
- Opérateurs que vous pouvez surcharger mais il est rare de le faire :
 - `new, delete, new[], delete[]`
 - `->, &, *`
 - `,`
- Vous ne pouvez pas créer de nouveaux opérateurs !
- Vous ne pouvez pas surcharger :
 - `?:: :: .* sizeof()`

Example: operator ()

Une classe pour les fonctions quadratiques : $ax^2 + bx + c$

```
class quadratic{
    double a,b,c;
public:
    quadratic(double a,double b,double c):a{a},b{b},c{c}
    {};
    double operator ()(double x){
        return a*x*x+b*x+c;
    }
};
```

$2x^2 + 3x + 4$

```
quadratic Q{2,3,4};
cout<<"2*x^2+3*x+4="<<endl;
for (int x=0;x<10;x++)
    cout<<Q(x)<<" when x="<<x<<endl;
```

$2*x^2+3*x+4 =$
4 when $x=0$
9 when $x=1$
18 when $x=2$
31 when $x=3$
48 when $x=4$
69 when $x=5$
94 when $x=6$
123 when $x=7$
156 when $x=8$
193 when $x=9$

Exercice:

make >> et << push and pop dans une pile

- Code d'essai :

```
stringStack ss;
ss<<"a"<<"b"<<"c"<<"d"<<"e"<<"f"; //Pushes a to f
string s1,s2;
ss>>s1>>s2; //Pops into s1 then s2
cout<<s1<<" "<<s2<<endl; //Should print f e
while (!ss.isEmpty()){
    cout<<(ss>>s1,s1)<<endl; //Should print d c b a
}
```

Sortie

f
e
d
c
b
a

Résumé : opérateurs

- Les opérateurs sont comme des fonctions normales
- C++ a un nombre fixe d'opérateurs que vous pouvez surcharger
- Vous pouvez les surcharger en tant que méthodes ou fonctions indépendantes
- Quand surcharger les opérateurs ?
 - Quand il serait naturel de les utiliser avec votre classe
 - Lorsque vous souhaitez utiliser votre classe avec un autre code qui le nécessite (par exemple, une fonction `sort` qui suppose que les opérateurs de comparaison sont codés)

Conversion

Programmation orientée objet

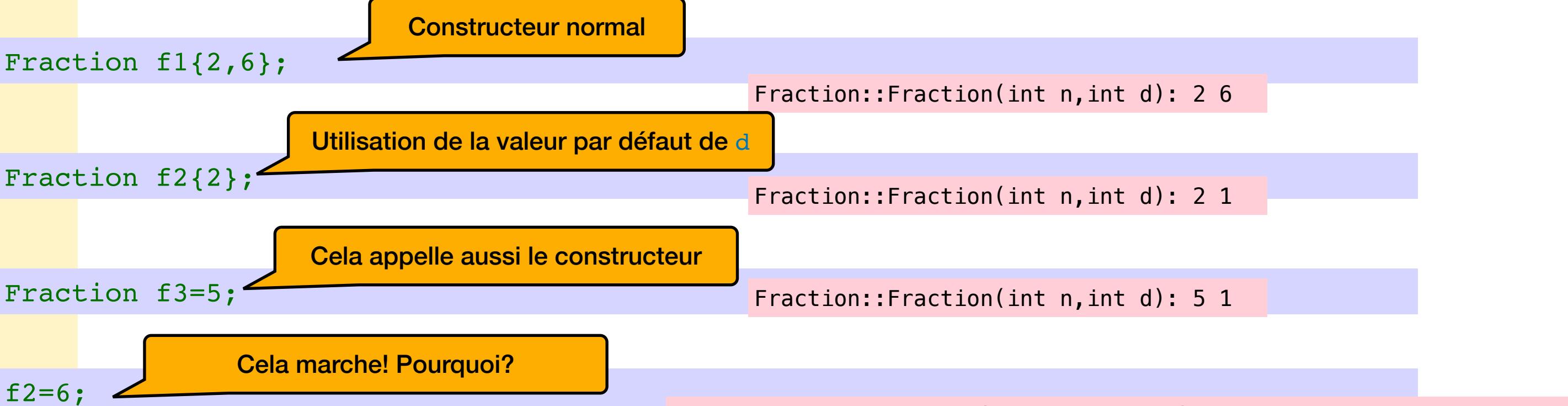
Prof. John Iacono

Conversion d'un entier en une fraction

Les constructeurs sont l'une des deux façons de définir une conversion

```
class Fraction{  
    int n;  
    int d;  
    void reduce();  
public:  
    Fraction(int n,int d=1);  
    string toString();  
    ... Fraction(Fraction &other);  
    Fraction &operator =(const Fraction &other);  
};
```

Nous ajoutons du code pour le constructeur de copie et l'opérateur = afin que nous puissions imprimer ce qui se passe



Voici les implémentations des deux constructeurs et de l'opérateur = avec `cout`

```
Fraction::Fraction(int n,int d):n{n},d{d}{  
    cout<<"Fraction::Fraction(int n,int d): "<<n<<" "<<d<<endl;  
    reduce();  
}  
Fraction::Fraction(Fraction &other):n{other.n},d{other.d}{  
    cout<<"Fraction::Fraction(Fraction &other): "<<other.n<<" "<<other.d<<endl;  
}  
Fraction &Fraction::operator =(const Fraction &other){  
    n=other.n;  
    d=other.d;  
    cout<<"Fraction &Fraction::operator =(Fraction &other): "<<other.n<<" "<<other.d<<endl;  
    return *this;  
}
```

Donc, cela a d'abord appelé le constructeur de `Fraction` avec `6` pour créer une `Fraction` temporaire

Ensuite, il a appelé le constructeur de copie sur `f2` avec la `Fraction` temporaire comme paramètre

Conversion implicite

Lorsque C++ convertit un type en un autre sans qu'aucun code n'indique qu'une conversion devrait avoir lieu

On ne peut pas ajouter un constructeur à float qui prend une fraction

Conversion d'une fraction en flottant

```
class Fraction{  
    int n;  
    int d;  
    void reduce();  
  
public:  
    Fraction(int n,int d=1); // Cela permet la conversion DE int À Fraction  
    string toString();  
    operator float (); // Cela permet la conversion d'une fraction en un flottant  
    ...  
}; // Aucun paramètre  
Aucune valeur de retour devant. Le type de retour est flottant !
```

```
Fraction::operator float() {  
    return static_cast<float>(n)/static_cast<float>(d);}
```

Nous devons convertir n et d en nombres flottants,
sinon nous obtenons une division entière, c'est-à-dire
 $10/3=3$

static_cast<T>(x)

Essaie de convertir x en type T

```
Fraction f{2,6};  
float fl=f;  
cout<<fl<<endl; // 0.333333  
  
cout<<f<<endl; // 0 .333333
```

Cela ne fonctionne que parce que nous
n'avons qu'une seule conversion de
Fraction vers quelque chose que cout sait
imprimer.

Si nous ajoutons l'opérateur double()
cout<<f<<endl;
Devient ambigu et donne cette erreur

```
12-conversion.cpp:122:7: error: use of overloaded operator '<<' is ambiguous (with operand types 'std::ostream' (aka 'basic_ostream<char>') and 'Fraction')  
    cout<<f<<endl;  
    ^~~~~~ ~  
/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX.sdk/usr/include/c++/v1/ostream:207:20: note: candidate function  
basic_ostream& operator<<(bool __n);  
/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX.sdk/usr/include/c++/v1/ostream:208:20: note: candidate function  
basic_ostream& operator<<(short __n);  
/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX.sdk/usr/include/c++/v1/ostream:209:20: note: candidate function  
basic_ostream& operator<<(unsigned short __n);  
/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX.sdk/usr/include/c++/v1/ostream:210:20: note: candidate function  
basic_ostream& operator<<(int __n);  
/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX.sdk/usr/include/c++/v1/ostream:211:20: note: candidate function  
basic_ostream& operator<<(unsigned int __n);  
/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX.sdk/usr/include/c++/v1/ostream:212:20: note: candidate function  
basic_ostream& operator<<(long __n);  
/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX.sdk/usr/include/c++/v1/ostream:213:20: note: candidate function  
basic_ostream& operator<<(unsigned long __n);  
/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX.sdk/usr/include/c++/v1/ostream:214:20: note: candidate function  
basic_ostream& operator<<(long long __n);  
/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX.sdk/usr/include/c++/v1/ostream:215:20: note: candidate function  
basic_ostream& operator<<(unsigned long long __n);  
/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX.sdk/usr/include/c++/v1/ostream:216:20: note: candidate function  
basic_ostream& operator<<(float __f);  
/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX.sdk/usr/include/c++/v1/ostream:217:20: note: candidate function  
basic_ostream& operator<<(double __f);  
/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX.sdk/usr/include/c++/v1/ostream:218:20: note: candidate function  
basic_ostream& operator<<(long double __f);  
1 error generated.
```

Limitation des conversions : explicit

```
vector<int> v{1,2,3,4,5};  
for (auto &i:v) cout<<i<<" ";cout<<endl;
```

1 2 3 4 5

```
vector<int> v2(10);  
for (auto &i:v2) cout<<i<<" ";cout<<endl;
```

0 0 0 0 0 0 0 0 0 0

Essayons

```
v=10  
for (auto &i:v) cout<<i<<" ";cout<<endl;
```

candidate function not viable:
no known conversion from 'int'
to 'const std::vector<int>' for
1st argument

```
v=static_cast< vector<int> >(10);  
for (auto &i:v) cout<<i<<" ";cout<<endl;
```

Nous pouvons utiliser static_cast pour forcer une conversion explicite

0 0 0 0 0 0 0 0 0 0

explicit empêche les conversions implicites

```
explicit vector( size_type count );
```

Généralement défini comme une sorte d'entier, par ex. unsigned long long

Mot-clé explicit

Vous pouvez l'ajouter à un constructeur ou à un opérateur de conversion pour empêcher les conversions implicites.

En vecteur, c'était une bonne idée, car cela empêche **v=10** de vider le vecteur et de le remplacer par un vecteur de **10** éléments construits par défaut, ce qui ne semble pas naturel

Quand les conversions implicites fonctionnent

Les règles sont complexes, essentiellement lorsqu'une seule conversion non ambiguë peut être appliquée

```
class Fraction{  
    int n;  
    int d;  
    void reduce();  
public:  
    Fraction(int n,int d=1);  
    string toString();  
    operator float ();  
....  
};
```

```
struct To {  
    To() = default;  
To(const struct From&) {}};  
  
struct From {  
    operator To() const {return To();}  
};
```

```
From f;  
To t1{f};  
To t2 = f; //ERROR ambiguous
```

```
struct C {  
    C(float f);  
    ...  
};
```

```
C c1(frac);  
C c2{frac};  
C c3=frac; //ERROR  
C c3=static cast<float>(frac);
```

Des règles de conversion légèrement différentes existent pour l'initialisation par copie et l'initialisation directe

Besoin de deux conversions

```
c1=frac; /ERROR  
c1=static cast<float>(frac);
```

```
struct Hello {  
    operator string(){return string("Hello world!");}  
};
```

```
Hello h;  
string s=h;  
cout<<s<<endl;
```

```
Hello h;  
cout<<h<<endl;
```

Cela fonctionnait plus tôt avec `float` mais échoue avec `string`!

`string` est défini comme `std::basic_string<char>`, et les règles pour les templates n'autorisent pas implicitement cette conversion.

```
cout<<static_cast<string>(h)<<endl;
```

Hello world!

Morale : utilisez `static_cast` pour forcer une conversion

Les erreurs du C++ peuvent être horribles. Surtout avec du code modélisé complexe.

Sommaire

- Deux façons de définir une conversion :
 - Un constructeur de classe `C` définit une conversion vers cette classe.
 - Un opérateur de conversion dans la classe `C` définit une conversion à partir de cette classe.
- Les conversions peuvent être implicites ou explicites
 - **Implicite** : Sans transtypage (typecast). Quand cela fonctionne est soumis à des règles complexes (en particulier pour les classes modélisées).
 - **Explicite** : Avec transtypage (typecast). L'utilisation de `static_cast<newtype>(V)` convertit `V` en `newtype`. L'ancien code utilise `(newtype)V` ; ne l'utilisez pas.
- La définition d'un constructeur ou d'un opérateur de conversion comme `explicit` empêche les conversions implicites. Quand vous écrivez un constructeur avec un seul paramètre vous devez vous demander toujours : est-ce que cela a un sens en tant que conversion ? Si non, marquez-le comme `explicite`.

const et constexpr

Programmation orientée objet

Prof. John Iacono

const = ne peut pas être changé

- les variables et les méthodes peuvent être const
- la variable est const : ne peut pas être modifiée après la déclaration
- la méthode est const : ne peut pas se changer (*this).
 - Qu'est-ce qu'un changement ?
 - Pour un type intégré c'est évident
- Pour les classes, une modification est un appel à une méthode non constante ou une modification à une variable d'instance.
- Vous ne pouvez pas éviter const car nous utilisons le passage par référence constante tout le temps !
- Vous devez marquer toutes les méthodes comme const le cas échéant

const est nécessaire!

```
class Fraction{  
    int n;  
    int d;  
    void reduce();  
  
public:  
    Fraction(int n,int d);  
    string toString();  
    int numerator(){return n;}  
    int denominator(){return d;}  
};
```

toString n'est pas const

```
void printFraction(const Fraction &f){  
    cout<<f.toString()<<endl;  
}
```

passer par référence constante

toString est appelé sur f

```
Fraction f{2,4};  
printFraction(f);
```

13-const.cpp:38:9: error: 'this' argument to member function
'toString' has type 'const Fraction', but function is not marked
const
 cout<<f.toString()<<endl;

const est nécessaire!

```
class Fraction{  
    int n;  
    int d;  
    void reduce();  
public:  
    Fraction(int n,int d);  
    string toString() const;  
    int numerator(){return n;}  
    int denominator(){return d;}  
};  
  
string Fraction::toString() const{  
    return to_string(n) + '/' + to_string(d);  
}
```

toString est const

numerator et denominator
ne sont pas const

```
void printFraction(const Fraction &f){  
    cout<<f.toString()<<endl;  
}
```

nous appelons toString sur f

f est const

ça casse encore !

```
Fraction f{2,4};  
printFraction(f);
```

```
13-const.cpp:37:19: error: 'this' argument to member function 'numerator'  
has type 'const Fraction', but function is not marked const  
    return to_string(numerator())+'/'+to_string(denominator());  
^~~~~~
```

```
13-const.cpp:19:6: note: 'numerator' declared here  
    int numerator(){return n;}  
^
```

```
13-const.cpp:37:46: error: 'this' argument to member function 'denominator'  
has type 'const Fraction', but function is not marked const  
    return to_string(denominator());  
^~~~~~
```

toString appelle numerator

```
int denominator(){return d;}  
  
string Fraction::toString() const{  
    return to_string(numerator())+'/'+to_string(denominator());  
}
```

Moral: mark all methods that don't change member variables as constant

```
class Fraction{  
    int n;  
    int d;  
    void reduce();  
  
public:  
    Fraction(int n,int d);  
    string toString() const;  
    int numerator() const {return n;}  
    int denominator() const {return d;}  
};
```

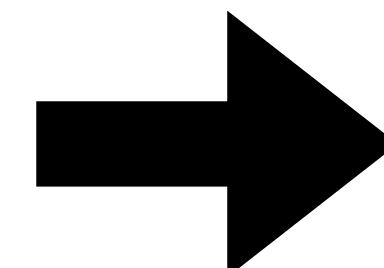
```
string Fraction::toString() const{  
    return to_string(n) + '/' + to_string(d);  
}
```

```
void printFraction(const Fraction &f){  
    cout<<f.toString()<<endl;  
}
```

```
Fraction f{2,4};  
printFraction(f);
```

2/4

tout va bien encore



```
string Fraction::toString() const{  
    return to_string(numerator()) + '/' + to_string(denominator());  
}
```

Tricherie const

- Une variable membre déclarée `mutable` peut être modifiée même si l'instance est `const`.
- Souvent utile pour le débogage

mutable

```
class Int {  
    int i;  
    mutable int readCount=0;  
public:  
    Int(int i):i{i}{};  
    Int &operator =(int newi){  
        i=newi;  
        return *this;  
    };  
    operator int() const {  
        readCount++;  
        return i;  
    }  
    int getReadCount() const {  
        return readCount;  
    }  
};
```

Conversion en **int** est **const**

Serait une erreur si
readcount n'était pas
mutable

```
void printkTimes(const Int &i, int k){  
    for (int j=0;j<k;j++){  
        cout<<i<<" ";  
    }  
    cout<<endl;  
}  
  
int testMutable(){  
    Int i(7);  
    printkTimes(i,10);  
    cout<<i.getReadCount()<<endl;  
}
```

appelle **operator**
int() comme
conversion implicite

7 7 7 7 7 7 7 7 7
10

pointeurs **const** : trois types

- Pour interdire la modification d'un pointeur (l'adresse du pointeur) :

```
int i=5, j=7;  
const int *p=&j; // pareil que int const *p=&j  
*p=7;  
p=&i; // Erreur, impossible de changer p
```

- Pour interdire la modification de ce vers quoi pointe un pointeur :

```
int i=5, j=7;  
int *const p=&j;  
p=&i;  
*p=7; // Erreur, impossible de changer p *p
```

- Pour interdire la modification du pointeur et de ce sur quoi le pointeur pointe :

```
int i=5, j=7;  
const int *const p=&j;  
p=&i; // Erreur, impossible de changer p p  
*p=7; // Erreur, impossible de changer p *p
```

const fait partie du système de types

- Une variable de type **T** a une conversion implicite en un **const T**
- Vous ne pouvez pas utiliser une variable de type **const T** où un **T** est nécessaire
- Vous pouvez supprimer de force le const d'un **x** de type **const T** comme suit :

const_cast<T>(x)

- C'est rarement une bonne idée
- Vous pouvez également convertir **x** pas **const T** en **const T** :

const_cast<const T>(x)

- C'est généralement sans danger

const ou ne pas const?

Parfois, vous avez besoin des deux !

```
class C {  
    string s;  
public:  
    C(string s):s{s}{};  
    string &getString(){return s;}  
};  
  
void printC(const C &s){  
    cout<<s.getString()<<endl;  
}
```

passer par référence

getString n'est pas const

```
C s{"Hello!"};  
s.getString()="Bonjour!";  
printC(s);
```

```
13-const.cpp:84:16: error: no viable overloaded '='  
    s.getString()="Bonjour!";  
~~~~~^~~~~~  
/Applications/Xcode.app/Contents/Developer/Platforms/  
MacOSX.platform/Developer/SDKs/MacOSX.sdk/usr/include/c++/v1/  
string:877:19: note: candidate function not viable: 'this'  
argument has type 'const std::string' (aka 'const  
basic_string<char>'), but method is not marked const  
    basic_string& operator=(const basic_string& __str);
```

```
class C {  
    string s;  
public:  
    C(string s):s{s}{};  
    const string &getString() {return s;}  
};  
  
13-const.cpp:53:9: error: 'this' argument to member  
function 'getString' has type 'const C', but function is  
not marked const  
cout<<s.getString()<<endl;
```

getString n'est toujours pas constant, il renvoie une constante

```
class C {  
    string s;  
public:  
    C(string s):s{s}{};  
    string &getString() const {return s;}  
};  
  
13-const.cpp:50:36: error: binding reference of type  
'basic_string<...>' to value of type 'const  
basic_string<...>' drops 'const' qualifier  
    string &getString() const {return s;}
```

getString est const et ne peut donc pas renvoyer une référence non constante à une variable d'instance

```
class C {  
    string s;  
public:  
    C(string s):s{s}{};  
    const string &getString() const {return s;}  
};  
  
13-const.cpp:84:16: error: no viable overloaded '='  
    s.getString()="Bonjour!";  
~~~~~^~~~~~
```

Nous ne pouvons pas changer la chaîne en bonjour en utilisant une référence constante

```
class C {  
    string s;  
public:  
    C(string s):s{s}{};  
    string &getString(){return s;}  
    const string &getString() const {return s;}  
};  
  
void printC(const C &s){  
    cout<<s.getString()<<endl;  
}
```

Solution: deux getStrings !

```
C s{"Hello!"};  
s.getString()="Bonjour!";  
printC(s);
```

Bonjour!

Le besoin de méthodes const et non const est courant !

Résumé : const

- Vous ne pouvez pas ignorer const!
- Règle de base :
 - marquez toutes les variables `const` que vous ne changerez pas
 - marquer toutes les méthodes `const` si elles ne modifient pas les variables d'instance ou n'appellent pas d'autres méthodes non `const`
- Vous devez penser à `const` avec chaque définition
- Si vous essayez d'ajouter `const` au code là où vous l'avez ignoré, cela conduit à un gâchis. Faites-le dès le début.
- Mon code dans ce cours a ignoré const, à partir de maintenant je vais l'utiliser correctement
- `const` est une partie importante du système de types en C++ et est essentiel pour ne pas permettre aux choses de changer alors qu'elles ne devraient pas changer.

constexpr

- `const` = ne change pas après l'initialisation
- `constexpr` peut être des variables ou des fonctions/méthodes
 - variables `constexpr` : valeur constante connue à la compilation
 - fonctions `constexpr` : si les paramètres sont `constexp`, le compilateur peut déterminer la valeur au moment de la compilation
- `constexp` a été introduit en C++11. Exactement ce qui est autorisé dans une fonction `constexp` a été étendu dans les versions ultérieures de C++
- Toutes les variables `constexpr` sont `const`, mais les fonctions `constexpr` ne renvoient un `constexp` que si les paramètres sont `constexp`
- Pourquoi?
 - Accélérez certains calculs en les effectuant au moment de la compilation
 - Parfois, vous avez besoin d'une valeur au moment de la compilation

Exemple : création d'un tableau de taille fixe

- C++ a un moyen de créer un tableau de taille fixe :

```
array<int, 3> A1;  
// Array of 3 ints,
```

```
int A2[3];  
// Ancienne façon
```

- La taille doit être connue au moment de la compilation.
- La meilleure pratique consiste à marquer toutes les fonctions comme `constexpr` chaque fois que possible

Exemple : création d'un tableau de taille fixe

- C++ a un moyen de créer un tableau de taille fixe :

```
array<int, 3> A1;  
// Array of 3 ints,
```

```
int A2[ 3 ];  
// Ancienne façon
```

- La taille doit être connue au moment de la compilation.
- La meilleure pratique consiste à marquer toutes les fonctions comme **constexpr** chaque fois que possible

```
int quadratic1(int a, int b, int c, int x) {  
    return a*x*x+b*x+c;  
}  
  
constexpr int quadratic2(int a, int b, int c, int x) {  
    return a*x*x+b*x+c;  
}  
  
void testconstexpr() {  
    array<int, 3> A1;  
    // array<int, quadratic1(1,2,3,4)> A2;  
    array<int, quadratic2(1,2,3,4)> A3;  
}
```

3 est OK

résultat inconnu à la compilation

13-const.cpp:73:13: error: non-type
template argument is not a constant
expression
array<int,
quadratic1(1,2,3,4)> A2;

Fonctionne puisque quadratic2 est un **constexpr**

const est essentiel à comprendre.
constexpr est moins important.

Déplacement

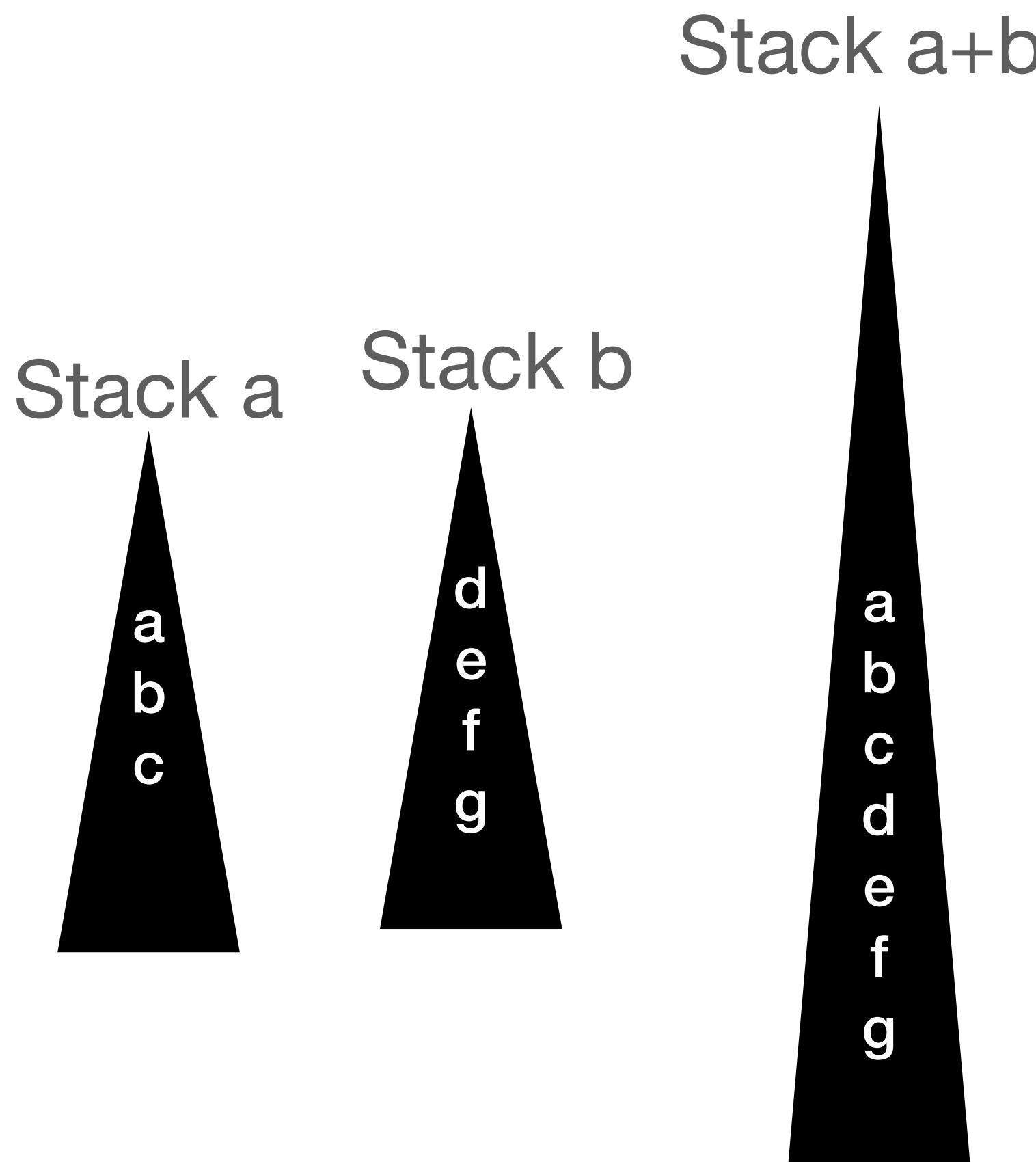
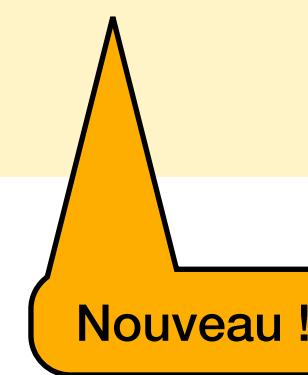
Programmation orientée objet

Prof. John Iacono

Combiner des piles

Ajout d'opérateur +

```
class stringStack{  
    struct node {  
        string s;  
        node* next;  
    };  
    node *topNode;  
    void makeEmpty();  
    void pushAllFrom(stringStack const &other);  
public:  
    stringStack();  
    stringStack(stringStack const &other);  
    ~stringStack();  
  
    void push(const string &s);  
    string pop();  
    string &top() const;  
    bool isEmpty() const;  
  
    stringStack &operator=(stringStack const &other);  
    stringStack operator+(stringStack const &other) const;  
    string toString() const;  
};
```



```
stringStack stringStack::operator+(stringStack const &other) const{  
    stringStack toReturn{*this};  
    toReturn.pushAllFrom(other);  
    return toReturn;
```

éléments de **b** ajoutés à la nouvelle pile

nouvelle pile retournée

- **a+b** renvoie un nouvel objet de pile
- **a** et **b** ne sont pas modifiés
- Ainsi, toutes les données de **a** et **b** doivent être copiées dans la nouvelle pile **a+b**

b ne change pas

nouvelle pile créée à partir d'**a**

a ne change pas

c=a+b

```
stringStack a;
stringStack b;

for (int i=0;i<10;i++){
    a.push(to_string(i));
    b.push(to_string(i+100));
}

cout<<"a "<<a.toString()<<endl; a: 9 8 7 6 5 4 3 2 1 0
cout<<"b "<<b.toString()<<endl; b: 109 108 107 106 105 104 103 102 101 100

stringStack c=a+b;

cout<<"c: "<<c.toString()<<endl; c: 109 108 107 106 105 104 103 102 101 100 9 8 7 6 5 4 3 2 1 0
```

```
void stringStack::pushAllFrom(stringStack const &other){
    cout<<"pushAllFrom: "<<other.toString()<<endl;
    stringStack tmp;
    for (auto p=other.topNode;p;p=p->next) tmp.push(p->s);
    while (!tmp.isEmpty()) push(tmp.pop());
}
```

Tout va bien, non ?

```
stringStack stringStack::operator+(stringStack const &other) const{
    stringStack toReturn{*this};
    toReturn.pushAllFrom(other);
    return toReturn;
}
```

```
stringStack &stringStack::operator =(stringStack const &other){
    makeEmpty();
    pushAllFrom(other);
    return *this;
}
```

```
pushAllFrom: 9 8 7 6 5 4 3 2 1 0
pushAllFrom: 109 108 107 106 105 104 103 102 101 100
pushAllFrom: 109 108 107 106 105 104 103 102 101 100 9 8 7 6 5 4 3 2 1 0
```

c=a+b

```
stringStack a;  
stringStack b;  
  
for (int i=0;i<10;i++){  
    a.push(to_string(i));  
    b.push(to_string(i+100));  
}  
  
cout<<"a "<<a.toString()<<endl;  
cout<<"b "<<b.toString()<<endl;  
  
stringStack c;  
c=a+b;  
  
cout<<"c: "<<c.toString()<<endl;
```

Que se passe t-il ici?

- l'opérateur + est appelé sur a avec le paramètre b.
- Un nouvel objet est créé et renvoyé. Ce temporaire n'a pas de nom (un **rvalue**).
- Le constructeur de copie de la pile est appelé pour initialiser c avec le résultat temporaire renvoyé par a+b

Tout de a et b est copié dans le nouvel objet de pile

Tout de l'objet de pile temporaire renvoyé par a+b est copié dans c

Une copie des données en a et b est nécessaire.
Deux copies c'est excessif !

Comment pouvons-nous résoudre ce problème?

Comme a+b renvoie une pile temporaire, pouvons-nous simplement voler les données de ce temporaire et les mettre dans c ?

Oui!

Déplacement assignation

```
stringStack a;
stringStack b;

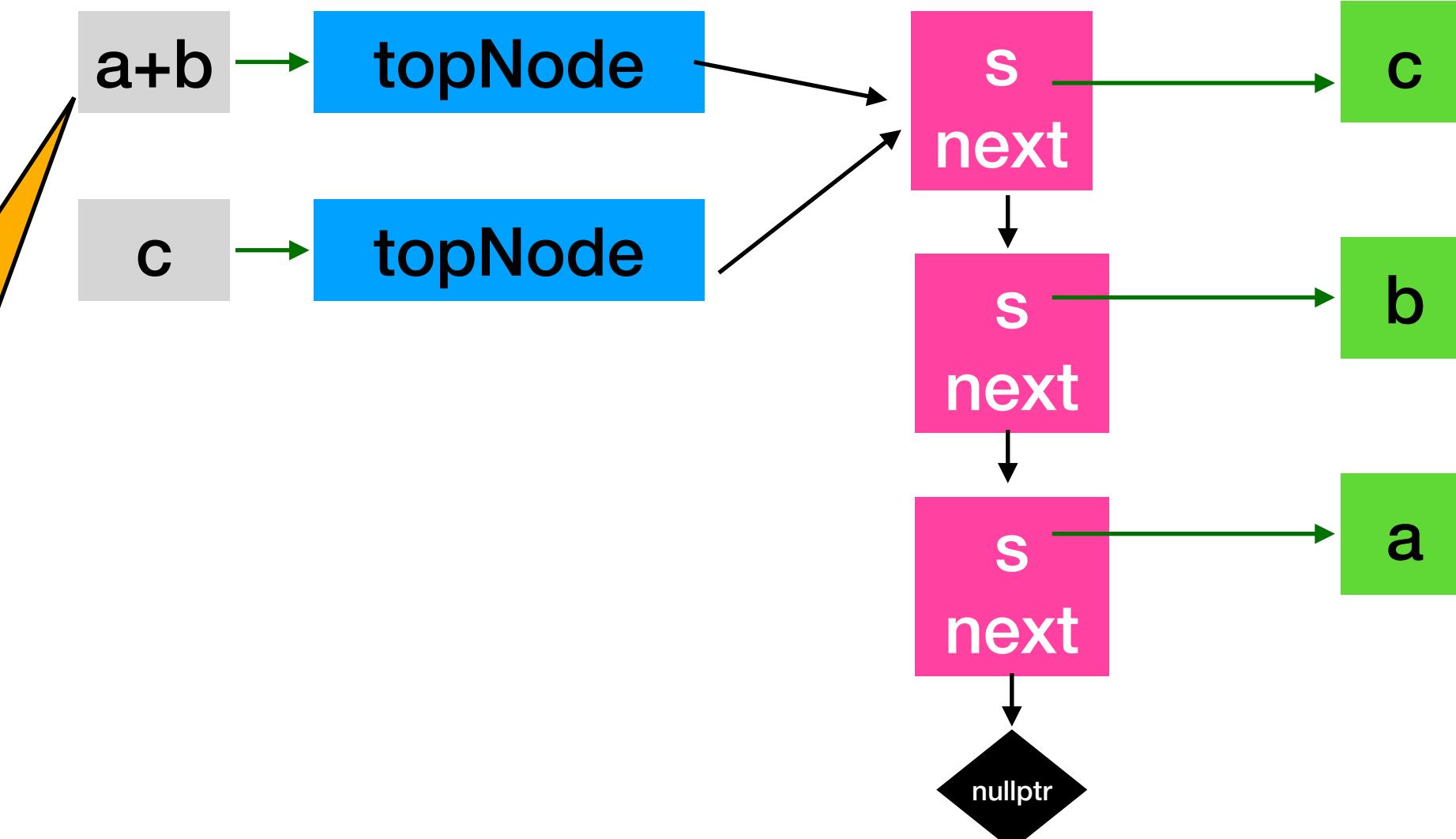
for (int i=0;i<10;i++){
    a.push(to_string(i));
    b.push(to_string(i+100));
}

cout<<"a "<<a.toString()<<endl;
cout<<"b "<<b.toString()<<endl;

stringStack c;
c=a+b;

cout<<"c: "<<c.toString()<<endl;
```

a+b is est temporaire et mourra
à la fin de l'expression. Il n'y a
donc aucun danger que ces
deux piles s'entremêlent



```
stringStack &stringStack::operator =(stringStack const &other){
    makeEmpty();
    pushAllFrom(other);
    return *this;
}
```

Ceci copie tous les éléments de other

```
stringStack &operator=(stringStack &&other){
    makeEmpty();
    topNode=other.topNode;
    other.topNode=nullptr;
    return *this;
}
```

Ceci n'est appelé que lorsque
other est une rvalue

Cela a topNode pointant vers topNode de
other. Rapide, quelle que soit la taille de la pile

Mais nous l'avons fait à la leçon 10 et c'était
une mauvaise idée ! Pourquoi ça marche ici ?

Constructeur de déplacement

Move Constructor

Encore un constructeur

Prend une autre pile

L'autre pile est une rvalue, c'est-à-dire une variable temporaire sur le point de mourir

Ici, nous volons le pointeur vers les données de l'autre pile.
Pas de copie !!!

```
stringStack(stringStack &&other) :topnode{other.topnode}{  
    other.topNode=nullptr; }
```

Nous mettons `nullptr` dans `other.topNode`. Ceci est important car `other` est sur le point de mourir et son destructeur est appelé. Nous ne voulons pas que ce destructeur appelle `delete` sur les nœuds qui sont partie de `*this` maintenant.

Copy constructor

```
stringStack::stringStack(stringStack const &other) :topNode{nullptr}{  
    pushAllFrom(other);  
}
```

Swap

```
stringStack a;  
stringStack b;  
  
for (int i=0;i<10;i++){  
    a.push(to_string(i));  
    b.push(to_string(i+100));  
}
```

```
stringStack tmp{a};  
a=b;  
b=tmp;
```

Copie !

Copie !

Copie !

Copie trois fois !

```
pushAllFrom: 9 8 7 6 5 4 3 2 1 0  
pushAllFrom: 109 108 107 106 105 104 103 102 101 100  
pushAllFrom: 9 8 7 6 5 4 3 2 1 0
```

a b tmp ne sont pas des rvalues. Ce
sont des **variables normales**

Swap

Toutes les copies des données des piles ont été éliminées.

```
stringStack a;  
stringStack b;  
  
for (int i=0;i<10;i++){  
    a.push(to_string(i));  
    b.push(to_string(i+100));  
}
```

```
stringStack(stringStack const &other);
```

```
stringStack tmp{a};
```

```
a=b;
```

```
b=tmp;
```

```
stringStack &operator=(stringStack const &other)
```

```
pushAllFrom: 9 8 7 6 5 4 3 2 1 0  
pushAllFrom: 109 108 107 106 105 104 103 102 101 100  
pushAllFrom: 9 8 7 6 5 4 3 2 1 0
```

Pouvons-nous faire croire à c++ que `a` est un rvalue ?

```
stringStack(stringStack &&other);
```

```
stringStack &operator=(stringStack &&other)
```

```
stringStack &operator=(stringStack &&other)
```

```
stringStack a;
```

```
stringStack b;
```

```
for (int i=0;i<10;i++){
```

```
    a.push(to_string(i));
```

```
    b.push(to_string(i+100));
```

```
}
```

```
stringStack tmp{std::move(a)};
```

```
a=std::move(b);
```

```
b=std::move(tmp);
```

`std::move` ne bouge rien !

Tout ce qu'il fait est de dire à C++ de traiter `a` comme une rvalue. Ici, cela signifie que le constructeur de déplacement est appelé, pas le constructeur de copie.

Remarque : après avoir appelé `std::move` sur quelque chose, vous promettez de ne plus accéder à ses données.

`a` n'est pas une rvalue/variable temporaire. C'est une variable normale. Ainsi, le constructeur de copie normal est appelé et copie les

Mais nous n'allons plus jamais accéder aux données de `a` comme dans la ligne suivante, nous assignons `b` à `a`.

Notez que `std::swap` existe et fonctionne comme ceci. Mais pour obtenir l'accélération, vous devez avoir le constructeur de déplacement et assignation codés dans votre classe.

Quoi coder

- Lorsque votre classe utilise de la mémoire dynamique, vous devez coder :
 - constructeur de copie : `classname(const classname &)`
 - opérateur d'assignation `operator=(const classname &)`
- destructeur `~classname`
- Vous devez également coder pour des raisons d'efficacité les versions qui seront appelées à la place si le paramètre est une rvalue :
 - constructeur de déplacement: `classname(const classname &&)`
 - opérateur d'assignation de déplacement:
`operator=(const classname &&)`

Templates

Programmation orientée objet

Prof. John Iacono

Rappelez-vous notre classe de pile?

```
class stringStack{
    struct node {
        string s;
        node* next;
        node(string s, node *next):s(s),next(next){;}
    };
    node *top=nullptr;
public:
    void push(string s){top=new node(s,top);}
    string pop(){
        string s{top->s};
        node *oldtop=top;
        top=top->next;
        delete oldtop;
        return s;
    }
    bool isEmpty() const {return top==nullptr;}
    ~stringStack() {while (!isEmpty()) pop();}
};
```

```
class intStack{
    struct node {
        int s;
        node* next;
        node(int s, node *next):s(s),next(next){;}
    };
    node *top=nullptr;
public:
    void push(int s){top=new node(s,top);}
    int pop(){
        int s{top->s};
        node *oldtop=top;
        top=top->next;
        delete oldtop;
        return s;
    }
    bool isEmpty() const {return top==nullptr;}
    ~intStack() {while (!isEmpty()) pop();}
};
```

```
template <typename T>
class Stack{
    struct node {
        T s;
        node* next;
        node(T s, node *next):s(s),next(next){;}
    };
    node *top=nullptr;
public:
    void push(T s){top=new node(s,top);}
    T pop(){
        T s{top->s};
        node *oldtop=top;
        top=top->next;
        delete oldtop;
        return s;
    }
    bool isEmpty() const {return top==nullptr;}
    ~Stack() {while (!isEmpty()) pop();}
};
```

```
stringStack k;
k.push("Bonjour");
k.push("Au Revoir");
string s{k.pop()};
cout<<"Pop: "<<s<<endl;
```

Au Revoir

```
intStack k;
k.push(10);
k.push(12);
int i{k.pop()};
cout<<"Pop: "<<i<<endl;
```

12

```
Stack<string> k;
k.push("Bonjour");
k.push("Au Revoir");
string s{k.pop()};
cout<<"Pop: "<<s<<endl;
```

Au Revoir

```
Stack<int> k;
k.push(10);
k.push(12);
int i{k.pop()};
cout<<"Pop: "<<i<<endl;
```

12

C'était trop facile

Nous allons maintenant avoir un exemple plus compliqué

But:

Code:

```
middle(x,y,z)
```

qui retourne le milieu des trois valeurs.

```
middle(1,10,100)
```

```
middle(10,1,100)
```

```
middle(1,100,10)
```

devraient tous retourner 10

La fonction `middle` devrait fonctionner avec un large éventail de types.

Tentative 1: templates de fonctions

dog d4{middle1(d1,d2,d3)};

```
template <typename T>
T middle1(T a,T b,T c){
    if (a<=b && b<=c || c<=b && b<=a) return b;
    if (a<=c && c<=b || b<=c && c<=a) return c;
    return a;
}
```

cout << middle1<int>(1,20,15);

15

cout << middle1(1,20,15);

15

Vous pouvez omettre le type si C++ peut le déterminer

```
float a{1.3},b{2.5},c{0.4};
cout<<middle1(a,b,c);
```

1.3

cout<<middle1(string{"a"},string{"b"},string{"c"})

b

```
float a{1.3},b{2.5},c{0.4};
middle1(a,b,c)+=100;
```

```
middle.cpp:87:16: error: expression is not assignable
    middle1(a,b,c)+=100; //error
               ~~~~~~^~~~~~^
1 error generated.
```

```
float a{1.3},b{2.5},c{0.4};
middle1<int &>(a,b,c)+=100;
cout<<"a:"<<a<<" b:"<<b<<" c:"<<c<<endl;
```

a:101.3 b:2.5 c:0.4

Mais parfois C++ a besoin d'aide

```
struct dog{
    string name;
    int age;
    float weight;
};
```

```
dog d1{"fluffy",4,4.5};
dog d2{"snowflake",10,8.2};
dog d3{"guardian",20,4.0};
```

```
middle.cpp:8:7: error: invalid operands to binary expression ('dog' and 'dog')
    if (a<=b && b<=c || c<=b && b<=a) return b;
               ~~~~^~~~~~^
```

```
middle.cpp:95:9: note: in instantiation of function template specialization 'middle1<dog>' requested here
    dog d4{middle1(d1,d2,d3)};
```

Point clé:

Chaque fois que vous utilisez une fonction/classe/méthode basée sur un template avec un paramètre différent, le compilateur C++ compile le template avec les paramètres. Ce processus est appelé instantiation.

C'est vraiment comme si vous écriviez des fonctions séparées pour chaque choix de paramètres.

Comprendre cela est essentiel pour comprendre les messages d'erreur

Tentative 1

```
template <typename T>
T middle1(T a,T b,T c){
    if (a<=b && b<=c || c<=b && b<=a) return b;
    if (a<=c && c<=b || b<=c && c<=a) return c;
    return a;
}
```

```
struct dog{
    string name;
    int age;
    float weight;
};

dog d1{"fluffy",4,4.5};
dog d2{"snowflake",10,8.2};
dog d3{"guardian",20,4.0};

dog d4{middle1(d1,d2,d3)};
```

```
middle.cpp:8:7: error: invalid operands to binary expression ('dog' and 'dog')
    if (a<=b && b<=c || c<=b && b<=a) return b;
          ^~ ~
middle.cpp:95:9: note: in instantiation of function template specialization 'middle1<dog>' requested here
    dog d4{middle1(d1,d2,d3)};
```

Tentative 2: operator <=

```
bool operator <= (const dog &a, const dog &b){
    return a.name<=b.name;
}
```

```
dog d4{middle1(d1,d2,d3)};
cout<<d4.name<<endl;
```

guardian

Et si parfois vous vouliez choisir le chien du milieu en fonction parfois de son nom et parfois de son âge ?

Nous ne pouvons pas redéfinir l'opérateur <= deux fois !

Nous avons besoin d'un moyen de définir des fonctions de comparaison et de les utiliser comme paramètres

Foncteurs

```
struct dog{  
    string name;  
    int age;  
    float weight;  
};  
  
dog d1{"fluffy",4,4.5};  
dog d2{"snowflake",10,8.2};  
dog d3{"guardian",20,4.0};
```

```
struct dogLEage{  
    bool operator()(const dog &a,const dog &b){return a.age<=b.age;}  
};
```

```
dogLEage dleage;  
cout<<dleage(d1,d2)<<endl;  
  
cout<<dogLEage{}(d1,d2)<<endl;
```

Raccourci

Cette classe amusante n'a pas de données et n'a que l'**opérateur()** défini. C'est ainsi que vous définissez une fonction que vous souhaitez passer en paramètre.

Le foncteur est une classe.
Pour l'utiliser, définissez une variable de la classe et utilisez cette variable comme fonction.

Cela fonctionne mais c'est sûr que c'est moche !

Ancien code

```
template <typename T>  
T middle1(T a,T b,T c){  
    if (a<=b && b<=c || c<=b && b<=a) return b;  
    if (a<=c && c<=b || b<=c && c<=a) return c;  
    return a;  
}
```

Nouveau code

```
template <typename T,typename compare>  
T middle2(T a,T b,T c){  
    compare cmp;  
    if (cmp(a,b) && cmp(b,c) || cmp(c,b) && cmp(b,a)) return b;  
    if (cmp(a,c) && cmp(c,b) || cmp(b,c) && cmp(c,a)) return c;  
    return a;  
}
```

```
dog d5{middle2<dog,dogLEage>(d1,d2,d3)};  
cout<<d5.name<<endl;
```

snowflake

Mais maintenant ça ne marche que si on lui donne le foncteur **compare** !

Cela ne fonctionne plus pour int, string, etc.

```
out<<middle2<int,lessEqual<int>>(1,20,15)
```

Voici un foncteur templaté qui appelle juste `<=` sur la classe

```
template <typename T>  
struct lessEqual {  
    bool operator ()(T &a,T &b){return a<=b;}  
};
```

Paramètres par défaut de template

```
template <typename T,typename compare>
T middle2(T a,T b,T c){
    compare cmp;
    if (cmp(a,b) && cmp(b,c) || cmp(c,b) && cmp(b,a)) return b;
    if (cmp(a,c) && cmp(c,b) || cmp(b,c) && cmp(c,a)) return c;
    return a;
}
```

```
template <typename T,typename compare=lessEqual<T> >
T middle3(T a,T b,T c){
    compare cmp;
    if (cmp(a,b) && cmp(b,c) || cmp(c,b) && cmp(b,a)) return b;
    if (cmp(a,c) && cmp(c,b) || cmp(b,c) && cmp(c,a)) return c;
    return a;
}
```

```
template <typename T>
struct lessEqual {
    bool operator ()(T &a,T &b){return a<=b;}
};
```

```
cout<<middle2<int,lessEqual<int>>(1,20,15)
```

15

Cela fonctionne mais c'est
sûr que c'est moche !

```
cout<<middle3(1,20,15)<<endl
```

15

Bien mieux !

Spécialisation de template

Supposons que nous n'ayons pas défini l'opérateur `<=` sur la class dog

```
template <typename T,typename compare=lessEqual<T> >
T middle3(T a,T b,T c){
    compare cmp;
    if (cmp(a,b) && cmp(b,c) || cmp(c,b) && cmp(b,a)) return b;
    if (cmp(a,c) && cmp(c,b) || cmp(b,c) && cmp(c,a)) return c;
    return a;
}
```

```
template <typename T>
struct lessEqual {
    bool operator ()(T &a,T &b){return a<=b; }
};
```

Ajoutons une spécialisation !

Ceci est le template général de fonction lessEqual

Ceci est la spécialisation de le template lessEqual à la classe dog pour comparer les noms de chien

```
bool operator <=(const dog &a, const dog &b){
    return a.name<=b.name;
}
```

```
dog d5{middle3(d1,d2,d3)};
```

Error!
Tries:
 middle3<dog,lessEqual<dog>(d1,d2,d3)
Which calls:
 lessEqual<dog>(d1,d2)
Which runs
 d1<=d2
Which does not work!

Pouvons-nous faire fonctionner lessEqual<dog> directement pour comparer les noms de chiens ?

```
template <typename T>
struct lessEqual {
    bool operator ()(T &a,T &b){return a<=b; }
};

template <>
struct lessEqual<dog>{
    bool operator()(const dog &a,const dog &b){return a.name<=b.name; }
};
```

```
cout<<middle3(1,20,15)<<endl;
cout<<middle3(d1,d2,d3).name<<endl;
```

```
struct dog{
    string name;
    int age;
    float weight;
};

dog d1{"fluffy",4,4.5};
dog d2{"snowflake",10,8.2};
dog d3{"guardian",20,4.0};
```

Utilise lessEqual<T>

Utilise la spécialisation lessEqual<dog>

Expressions lambda

```
template <typename T,typename compare=lessEqual<T> >
T middle3(T a,T b,T c){
    compare cmp;
    if (cmp(a,b) && cmp(b,c) || cmp(c,b) && cmp(b,a)) return b;
    if (cmp(a,c) && cmp(c,b) || cmp(b,c) && cmp(c,a)) return c;
    return a;
}
```

```
template <typename T>
struct lessEqual {
    bool operator ()(T &a,T &b){return a<=b;}
};
```

```
template <>
struct lessEqual<dog>{
    bool operator()(const dog &a,const dog &b){return a.name<b.name;}
};
```

```
cout<<middle3(1,20,15)<<endl;      15
cout<<middle3(d1,d2,d3).name<<endl; guardian
```

```
struct dogLEage{
    bool operator()(const dog &a,const dog &b){return a.age<=b.age;}
};
```

```
cout<< middle3<dog,dogLEage>(d1,d2,d3).name<<endl;
snowflake
```

```
struct dog{
    string name;
    int age;
    float weight;
};

dog d1{"fluffy",4,4.5};
dog d2{"snowflake",10,8.2};
dog d3{"guardian",20,4.0};
```

```
struct dogLEage{
    bool operator()(const dog &a,const dog &b){return a.age<=b.age;}
};

dogLEage dle;
auto dleLambda=[ ](dog &a,dog &b){return a.age<=b.age;};

cout<<dle(d1,d2)<<endl;
cout<<dleLambda(d1,d2)<<endl;
```

Il s'agit d'une expression lambda. C'est un raccourci !

C'est la même chose que définir une classe, coder l'opérateur () et déclarer une variable de la classe.

```
cout<<middle3<dog,dleLambda>(d1,d2,d3).name<<endl;
```

Error! dleLambda est une variable pas un type!

Seuls les types peuvent être dans < >

Alors, comment passons-nous de l'expression lambda à la fonction ? C'est une variable, donc comme un paramètre normal comme celui-ci :

```
cout<<middle4(d1,d2,d3,dleLambda).name<<endl;
```

Manque d'espace, laisse-moi nettoyer le diapo

Expressions lambda

Nous devons déplacer cmp
d'une variable locale à un
paramètre

```
template <typename T,typename compare=lessEqual<T> >  
T middle3(T a,T b,T c){  
    compare cmp;  
    if (cmp(a,b) && cmp(b,c) || cmp(c,b) && cmp(b,a)) return b;  
    if (cmp(a,c) && cmp(c,b) || cmp(b,c) && cmp(c,a)) return c;  
    return a;  
}
```

```
struct dog{  
    string name;  
    int age;  
    float weight;  
};  
  
dog d1{"fluffy",4,4.5};  
dog d2{"snowflake",10,8.2};  
dog d3{"guardian",20,4.0};
```

```
template <typename T>  
struct lessEqual {  
    bool operator ()(T &a,T &b){return a<=b;}  
};  
  
template <>  
struct lessEqual<dog>{  
    bool operator()(const dog &a,const dog &b){return a.name<b.name;}  
};
```

```
auto dleLambda=[ ](dog &a,dog &b){return a.age<=b.age;}  
cout<<middle4(d1,d2,d3,dleLambda).name<<endl;
```

```
template <typename T, typename compare=lessEqual<T> >  
T middle4(T a,T b,T c, compare cmp=compare{}){  
    if (cmp(a,b) && cmp(b,c) || cmp(c,b) && cmp(b,a)) return b;  
    if (cmp(a,c) && cmp(c,b) || cmp(b,c) && cmp(c,a)) return c;  
    return a;  
}
```

Tout fonctionne maintenant !
Pas besoin de spécifier les
paramètres du template !

```
cout<<middle4(d1,d2,d3,[ ](dog &a,dog &b){return a.age<=b.age;}).name<<endl;  
cout<<middle4(d1,d2,d3,[ ](dog &a,dog &b){return a.age<=b.weight;}).name<<endl;  
cout<<middle4(d1,d2,d3).name<<endl;  
cout<<middle4(1,20,15)<<endl;
```

cmp reçoit le expression lambda
compare reçoit le type de l'expression lambda

compare reçoit lessEqual<int>
qui utilise lessEqual<T>
qui utilise le <= opérateur sur les entiers

Puis dans le paramètre par défaut :
cmp=compare{}

Ce qui est le même que :
cmp=lessEqual<int>{}

compare reçoit lessEqual<dog>
qui est une spécialisation de lessEqual<T>
qui utilise le nom du dog

Puis dans le paramètre par défaut :
cmp=compare{}

Ce qui est le même que :
cmp=lessEqual<dog>{}

Ne réinventez pas la roue

Version finale!!!!

```
struct dog{  
    string name;  
    int age;  
    float weight;  
};
```

```
template <typename T>  
struct lessEqual {  
    bool operator ()(T &a,T &b){return a<=b;}  
};  
  
template <>  
struct lessEqual<dog>{  
    bool operator()(const dog &a,const dog &b){return a.name<b.name;}  
};  
  
template <typename T, typename compare=lessEqual<T> >  
T middle(T a,T b,T c, compare cmp=compare{}){  
    if (cmp(a,b) && cmp(b,c) || cmp(c,b) && cmp(b,a)) return b;  
    if (cmp(a,c) && cmp(c,b) || cmp(b,c) && cmp(c,a)) return c;  
    return a;  
}
```

Cela existe déjà en C++. Il s'appelle std::less_equal

```
using namespace std;  
  
struct dog{  
    string name;  
    int age;  
    float weight;  
};  
  
template <>  
struct less_equal<dog>{  
    bool operator()(const dog &a,const dog &b){return a.name<b.name;}  
};  
  
template <typename T,typename compare=less_equal<T> >  
T middle(T a,T b,T c,compare cmp=compare{}){  
    if (cmp(a,b) && cmp(b,c) || cmp(c,b) && cmp(b,a)) return b;  
    if (cmp(a,c) && cmp(c,b) || cmp(b,c) && cmp(c,a)) return c;  
    return a;  
}  
  
cout<<middle(d1,d2,d3,[](dog &a,dog &b){return a.age<=b.age;}).name<<endl;  
cout<<middle(d1,d2,d3,[](dog &a,dog &b){return a.age<=b.weight;}).name<<endl;  
cout<<middle(d1,d2,d3).name<<endl;  
cout<<middle(1,20,15)<<endl;
```

snowflake (middle age)
fluffy (middle weight)
guardian (middle name)
15

std::min

Defined in header `<algorithm>`

```
template< class T >
const T& min( const T& a, const T& b );
```

(1) (until C++14)

```
template< class T >
constexpr const T& min( const T& a, const T& b );
```

(since C++14)

```
template< class T, class Compare >
const T& min( const T& a, const T& b, Compare comp );
```

(2) (until C++14)

```
template< class T, class Compare >
constexpr const T& min( const T& a, const T& b, Compare comp );
```

(since C++14)

```
template< class T >
T min( std::initializer_list<T> ilist );
```

(3) (since C++11)

(until C++14)

```
template< class T >
constexpr T min( std::initializer_list<T> ilist );
```

(since C++14)

```
template< class T, class Compare >
T min( std::initializer_list<T> ilist, Compare comp );
```

(4) (since C++11)

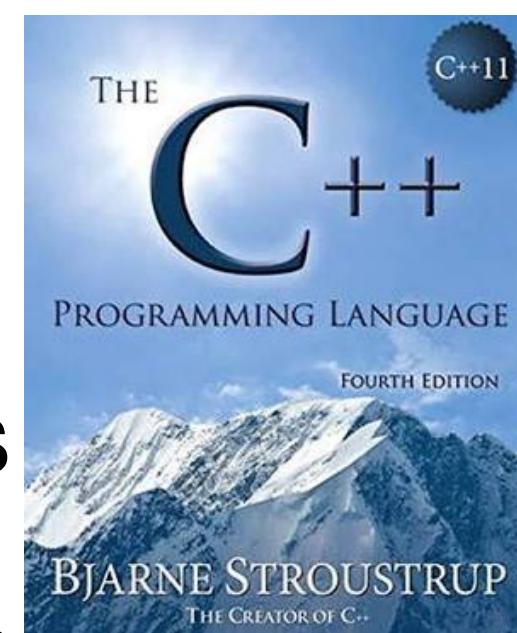
(until C++14)

```
template< class T, class Compare >
constexpr T min( std::initializer_list<T> ilist, Compare comp );
```

(since C++14)

Templates: summary

- Les templates peuvent être appliqués aux classes, méthodes et fonctions
- Les paramètres de templates sont des types, les paramètres normaux sont des instances d'un type
- Les templates sont compilés séparément avec chaque choix de paramètres de template qui se produit dans votre code, c'est ce qu'on appelle l'instanciation.
- En conséquence, les messages d'erreur sont souvent déroutants. C++20 essaie de résoudre ce problème avec une nouvelle fonctionnalité : concepts
- La spécialisation du template est utilisée lorsque vous souhaitez exécuter un code différent sur certains paramètres d'un template générique.
- Les template peuvent avoir des valeurs par défaut
- Les possibilités et les règles complètes sont complexes. 3 chapitres et >100 pages dans
- Lambda expression=foncteur=classe avec juste operator()=façon de passer une fonction



Pointeurs intelligents

Programmation orientée objet

Prof. John Iacono

Existe-t-il un choix intermédiaire ?

Variables locales/ membres

- Durée de vie fixe
- Doit être instancié (ne peut pas être `nullptr`)

Oui !

Un pointeur « intelligent »

- N'autoriser qu'un seul pointeur vers quelque chose d'alloué avec `new`
- `Delete` est automatiquement appelé lorsque le pointeur intelligent atteint la fin de sa durée de vie
- Le pointeur intelligent peut être déplacé vers un nouveau pointeur intelligent mais pas copié
- Le pointeur intelligent est toujours valide ou `nullptr`
- Pas aussi flexible que `new/delete`, mais c'est souvent tout ce dont nous avons besoin
- Pouvons-nous faire un tel pointeur intelligent?

Mémoire allouée dynamiquement

- Flexibilité totale : créé avec `new` et supprimé avec `delete`
- Pas besoin de le créer
- Mais...
 - Oublier d'appeler `delete` est une fuite de mémoire
 - Appeler `delete` deux fois est un plantage
 - Faire de l'arithmétique sur des pointeurs est autorisé et sujet aux erreurs
 - Déclarer un pointeur ne l'initialise pas à `nullptr`
 - `delete` ne met `nullptr` dans le pointer
 - Ces erreurs sont courantes, souvent intermittentes, non détectées par le compilateur et difficiles à déboguer

Notre pointeur intelligent

- C'est une classe calquée sur le type d'objet vers lequel elle pointera
- Il contient et représente un pointeur vers quelque chose d'un type inconnu, disons T.
- Il peut être construit avec un pointeur normal vers un T
- Un smartP peut être déplacé vers un autre smartP, lors de la construction ou plus tard.
- Le constructeur par défaut le définit sur nullptr
- Peut se comparer à nullptr et traiter comme booléen
- La copie des smartP est interdite
- Appeler * et -> sur un smartP<T> fonctionne comme s'il s'agissait d'un pointeur vers T
- Lorsque le smartP sort de la portée, il appelle delete automatiquement sur l'instance de T qu'il gère

```
{  
    smartP<Thing> p{new Thing};  
    Constructeur  
    smartP<Thing> p2{move(p)};  
    Constructeur de déplacement  
    p=move(p2);  
    operator = de déplacement  
    smartP<Thing> p3; //nullptr  
    Définir sur nullptr par défaut  
    if (p3) {do something}  
    Peut être utilisé comme un booléen  
    p3=p; //ERROR CAN NOT COPY  
  
    Thing t=*p;  
    p->someMethod();  
    le déréférencement  
    fonctionne comme prévu  
}  
l'instance de Thing est  
supprimée
```

Notre pointeur intelligent : interface

```
template <typename T>  
class smartP {  
    smartP();
```

Représentera un pointeur de type T*

```
explicit smartP(T *p);
```

Initialiser avec un pointeur. Je ne veux pas de conversions

```
smartP(smartP<T> &&other);  
smartP &operator =(smartP<T> &&other);
```

Déplacement

```
smartP(const smartP<T> &other)=delete;  
smartP &operator =(const smartP<T> &other)=delete;
```

Copie interdite

```
~smartP();
```

Destructeur, devrait appeler delete

```
T &operator *();  
T *operator ->();
```

Support * et -> comme un pointeur normal

```
friend bool operator ==<T>(nullptr_t, const smartP<T>&);  
friend bool operator ==<T>(const smartP<T>&, nullptr_t);
```

Autoriser les comparaisons
avec nullptr

Notre pointeur intelligent : code

Stocker un pointeur normal en tant que variable membre privée

```
template <typename T>
class smartP {
    T *p;
public:
    smartP():p{nullptr}{};  
    explicit smartP(T *p):p{p}{};  
    smartP(smartP<T> &&other):p{other.p}{other.p=nullptr;}  
    smartP &operator =(smartP<T> &&other){
```

Le constructeur par défaut définit p sur nullptr

Le constructeur régulier prend un pointeur et le stocke dans p

Le constructeur de déplacement prend le pointeur de other et définit le pointeur de other sur nullptr

L'assignation de déplacement fait la même chose mais s'assure de supprimer d'abord tout p existant (rappelez-vous que `delete` sur `nullptr` est OK et ne fait rien)

```
        delete p;  
        p=other.p;  
        other.p=nullptr;  
        return *this;
```

la copie n'est pas autorisée !

les opérateurs * et -> font comme prévu.

```
smartP &operator =(const smartP<T> &other)=delete;  
smartP(const smartP<T> &other)=delete;  
~smartP(){delete p;}  
T &operator *() {return *p;}  
T *operator ->() {return p;}  
operator bool(){return p;}
```

Lorsque le smartP sort de la portée, le destructeur est appelé qui appellera `delete` sur p

les opérateurs amis == != sont ignorés ici. Voir le code complet

node * est remplacé par
smartP<node>

Utilisation de notre pointeur intelligent : stack

```
template <typename T>
class Stack{
    struct node {
        T s;
        node* next;
        node(const T &s, node *next):s(s),next(next){}
    };
    node *top=nullptr;
public:
    void push(const T &s){top=new node(s,top);}
    T pop(){
        T s{top->s};
        node *oldtop=top;
        top=top->next;
        delete oldtop;
        return s;
    }
    bool isEmpty() {return top;}
    ~Stack() {while (!isEmpty()) pop();}
};
```

J'ai dû me souvenir de mettre top à nullptr

```
template <typename T>
class Stack{
    struct node {
        T s;
        smartP<node> next;
        node(const T &s, smartP<node> &&next):s(s),next(move(next)){}
    };
    smartP<node> top;
public:
    void push(const T &s){
        top=smartP<node>{new node(s,move(top))};
    }
    T pop(){
        T s{top->s};
        auto tmp=move(top->next);
        top=move(tmp);
        return s;
    }
    bool isEmpty() {return top;}
};
```

Le destructeur et `delete` sont supprimés !

Lorsque vous créez un nœud, vous devez lui donner le prochain pointeur intelligent via un mouvement ! Comme ça il n'y en a qu'un

Le constructeur par défaut définit p sur nullptr

Besoin d'utiliser move pour obtenir une référence rvalue

Nous indiquons explicitement quand le pointeur intelligent est déplacé

Sans cela, le pointeur intelligent serait copié, ce qui n'est pas autorisé

Démo : appel automatique de delete

```
template <typename T>
class Stack{
    struct node {
        T s;
        smartP<node> next;
        node(const T &s, smartP<node> &&next):s{s},next(move(next)){
            cout<<"Creating node:"<<s<<endl;
        ~node(){cout<<"Destroying node:"<<s<<endl;}
    };
    smartP<node> top;
public:
    void push(const T &s){
        top=move(smartP<node>{new node(s,move(top))});
    }
    T pop(){
        T s{top->s};
        auto tmp=move(top->next);
        top=move(tmp);
        return s;
    }
    .....
};
```

Ajouter du code à imprimer
lorsqu'un nœud est créé/détruit

pop n'appelle plus supprimer directement !
Mais le nœud est supprimé automatiquement
lorsque top obtient une nouvelle valeur

```
Stack<int> s;
for (auto i:{1,4,8,6,4}) s.push(i);
while (!s.isEmpty()) cout<<s.pop()<<endl;
```

Creating node:1
Creating node:4
Creating node:8
Creating node:6
Creating node:4
Destroying node:4
4
Destroying node:6
6
Destroying node:8
8
Destroying node:4
4
Destroying node:1
1

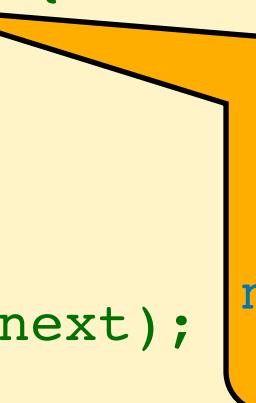
C++: std::unique_ptr<T>

```
template <typename T>
class Stack{
    struct node {
        T s;
        smartP<node> next;
        node(const T &s, smartP<node> &&next):s{s},next{move(next)}{}
    };
    smartP<node> top;
public:
    void push(const T &s){
        top=smartP<node>{new node(s,move(top))};
    }
    T pop(){
        T s{top->s};
        auto tmp=move(top->next);
        top=move(tmp);
        return s;
    }
    bool isEmpty() {return top==nullptr;}
};
```

Le STL de C++ fournit un pointeur intelligent comme le nôtre.

Il s'appelle std::unique_ptr<T>

```
class StackUP{
    struct node {
        T s;
        unique_ptr<node> next;
        node(const T &s,unique_ptr<node> &&next):s{s},next{move(next)}{}
    };
    unique_ptr<node> top;
public:
    void push(const T &s){
        top=unique_ptr<node>{new node(s,move(top))};
    }
    T pop(){
        T s{top->s};
        auto tmp=move(top->next);
        top=move(tmp);
        return s;
    }
    bool isEmpty() {return top==nullptr;}
};
```



make_unique combine la création du pointeur intelligent et l'appel de newtop=move(make_unique<node>(s,move(top)));

Autres classes de pointeurs intelligents : shared_ptr

- unique_ptr permet à un seul objet pointeur intelligent d'avoir un pointeur vers un élément qu'il gère
- Lorsque le unique_ptr meurt, il appelle delete sur l'élément qu'il gère
- Le déplacement est autorisé, la copie est interdite
- Aussi rapide que d'utiliser un pointeur nouveau/supprimer manuellement
- En interne, il suffit de stocker le pointeur
- shared_ptr permet à plusieurs objets pointeurs intelligents d'avoir un pointeur vers un élément qu'il gère
- Lorsque tous les objets shared_ptr d'un élément meurent, il appelle delete sur l'élément qu'il gère
- Le déplacement et la copie sont tous deux autorisés
- Très légèrement plus lent et plus grand que l'utilisation d'un pointeur et nouveau/supprimer manuellement
- En interne, doit stocker le pointeur et le nombre d'objets shared_ptr qui utilisent le pointeur sont actifs.

Lequel utiliser

- Si une simple variable fonctionne (locale, paramètre, membre, classe, globale...) utilisez-la. Cela ne fonctionnera pas si la durée de vie n'est pas appropriée, ou si parfois vous ne voulez pas créer d'objet (par exemple, top ne devrait pas être un nœud lorsque la classe de pile démarre)
- Si une seule variable pointe vers l'objet (qui peut être nullptr) et que l'objet doit mourir lorsque la variable pointant vers lui, utilisez un unique_ptr.
- Si plusieurs variables pointent vers un objet et que l'objet doit mourir avec le dernier objet pointant vers lui, utilisez shared_ptr
- Si aucun de ces éléments ne fonctionne, utilisez new/delete manuellement

Moral

- Il est important de comprendre new/delete, mais les pointeurs intelligents de C++11 devraient être utilisés dans la plupart des circonstances si vous avez le choix
- shared_ptr correspond à la façon dont les variables se comportent en Python
- En Java, les types intégrés comme int se comportent comme en C++, et les objets autres que les types intégrés se comportent comme shared_ptr.
- C++ a beaucoup plus d'options, cela le rend riche, compliqué et très rapide quand il le faut.

Stockage de données 1D

Programmation orientée objet

Prof. John Iacono

Stockage de données 1D

Nous avons réussi à éviter cela jusqu'à présent!

- Nous parlons de choses comme la liste de Python
- Les éléments sont définis et accessibles à l'aide de crochets : `x=A[i]`, `A[i]=x`, etc. Ces opérations sont rapides et ne dépendent pas de la taille du tableau.
- Plusieurs façons de le faire en C++
- Aucun d'entre eux n'a de tranches `A[5:10]` ou n'utilise d'indices négatifs (par exemple `A[-1]`) comme en Python.
- Tous nécessitent que tous les éléments soient du même type (mais ce type peut être un pointeur)
- Nous utilisons le mot « tableau/array » pour décrire toutes ces structures de manière générique. Il existe une classe dans la STL appelée `std::array`, et nous dirons toujours « STL array » ou « `std::array` » pour faire la distinction.

C-Style Arrays

- Exemple : `int x[7];`
- La taille doit être connue au moment de la compilation :

```
void f(int sz){ int x[sz] ; //ERREUR!!!
```

- Pas une classe ; pas de méthodes ! Pas de `.size()`.
- Se comporte comme un pointeur et est converti automatiquement en pointeur :
`x[3]` est identique à `* (x+3)`
- Aucune vérification pour voir si l'index est dans la plage. L'utilisation d'un index légèrement hors de portée corrompt probablement les données à proximité sans générer d'erreur.
- Rapide et compact

Tableaux de style C++ : allocation dynamique

- Crée avec `new`, supprimé avec `delete` :

```
int sz=10; Fraction *A=new Fraction[sz] ; .... delete [ ]A ;
```

- Le type de `A` a besoin d'un constructeur par défaut !
- Contrôler manuellement la création et la suppression
- La taille n'a pas besoin d'être connue au moment de la compilation
- Pas de vérification pour voir si un accès est à portée
- Pas une classe, juste un pointeur vers un bloc de mémoire avec plus d'un élément
- Tous les problèmes de la mémoire allouée dynamiquement normale plus les erreurs possibles de vérification de plage. `delete A != delete []A`
- Rapide et compact

Classe array de STL

- Substitut pour les tableaux C :

```
Fraction A[10];
```

devient:

```
array<Fraction, 10> A;
```

- Certains avantages, dont

```
A.size() // 10
```

- Compact (ne stocke même pas la taille, le compilateur le sait)
- `A.at(x)` et `A[x]` sont les mêmes sauf qu'à `A.at(x)` générera une erreur si `x` est hors de portée, et `A[x]` corrompre la mémoire à proximité, probablement en silence
- `A.at(x)` est légèrement plus lent que `A[x]` car il doit effectuer la vérification.
- La taille peut être transmise via des modèles aux fonctions.

La classe `vector` de STL

- Un `vector` de `shared_ptr` est la chose C++ la plus similaire à une liste Python

```
vector<Fraction> A(10) ;
```

- La taille n'est pas fixe :

```
A.push_back( Fraction(2,3) ) // Déplace la fraction dans le vecteur
```

```
A.emplace_back(2,3); // Ajoute une fraction au vecteur, en appelant directement le constructeur de Fraction avec 2,3
```

- De nombreuses méthodes à manipuler, comme la liste Python. Voir la documentation.
- Tout comme le tableau STL, `A[x]` ne vérifie pas si `x` est dans les limites et `A.at(x)` le fait.
- Prend en charge les opérations de copie et de déplacement rapide sur l'ensemble du tableau et permet de déplacer des éléments vers et depuis le tableau.
- Est implémenté en interne en stockant un tableau de taille fixe et en redimensionnant selon les besoins

Conseils généraux

- Utilisez `std::vector`, à moins que la taille fixée ne soit connue au moment de la compilation lorsque vous souhaiterez peut-être utiliser `std::array`
- Utilisez `.at(10)` et non `[10]` pour obtenir la vérification de la plage
- N'utilisez pas de tableaux à l'ancienne à moins qu'il n'y ait une raison spécifique pour laquelle `std::vector` n'est pas assez bon

Exercice

Quel est le problème avec ce code?

```
#include <iostream>
#include <vector>
#include <string>

using namespace std;

struct Tree{
    Tree *left,*right;
    int color;
    Tree(Tree *left,Tree *right):left(left),right(right),color{0}(){}
    Tree():Tree{nullptr,nullptr}{};
    void setColor(int newColor){
        color=newColor;
        if (left) left->setColor(newColor);
        if (right) right->setColor(newColor);
    }
};

int main(){
    vector<Tree> T;
    T.emplace_back();
    T.emplace_back();
    for (int i=0;i<50;i++)
        T.emplace_back(&T[i],&T[i+1]);
    T[20].setColor(1);
    for (auto t:T) cout<<t.color<<" ";
    cout<<endl;
}
```

STL: The Standard Template Library

- A beaucoup de deux parties principales:
 - Conteneurs : objets qui stockent des choses (par exemple `std::vector`, `std::array`)
 - Algorithmes : fonctions qui font quelque chose avec les données (par exemple `std::min`)
- Très riche! Avant de coder quelque chose qui est probablement d'usage général, vérifiez s'il est dans la STL
- Rapide
- Contenants pouvant être utiles :
 - `stl::unordered_map` : similaire à la classe de dictionnaire de Python. Comme en Python, utilise le hachage
 - `stl::tuple` : similaire à `std::array` mais où vous pouvez spécifier un type différent pour chaque position
 - `stl::set` : stocke les éléments qui ont un ordre trié, prenant en charge une recherche rapide exacte et inexacte. Il n'y a pas d'équivalent en Python.
 - `stl::initializer_list` : utilisé pour le passage de paramètres, pour les boucles comme `for (auto x:{1,2,3})`
- Chaque conteneur STL est conçu de manière à disposer d'un schéma de nommage cohérent pour leurs méthodes et propriétés. Cela les fait bien fonctionner avec les algorithmes, et si vous créez vos propres conteneurs, il est bon d'utiliser les conventions de la STL lorsque cela est possible.

Iterators

Programmation orientée objet

Prof. John Iacono

Qu'est-ce qu'un itérateur ?

C'est un moyen générique de parcourir des données 1D

- Ici, nous aurons une discussion simple sur ce que l'on appelle les forward iterators.
- Les itérateurs sont un moyen de parcourir des données 1D
- Pour apprendre les itérateurs, je vous montrerai comment faire fonctionner ce code pour une pile **s** :

```
for (auto i=s.begin(); i!=s.end(); ++i){cout<<*i<<endl;}
```

- Les itérateurs sont les objets renvoyés par les méthodes **begin()** et **end()**
- Ce sont des instances d'une nouvelle classe
- Ils doivent avoir **++ ==** et ***** pour que ce code fonctionne

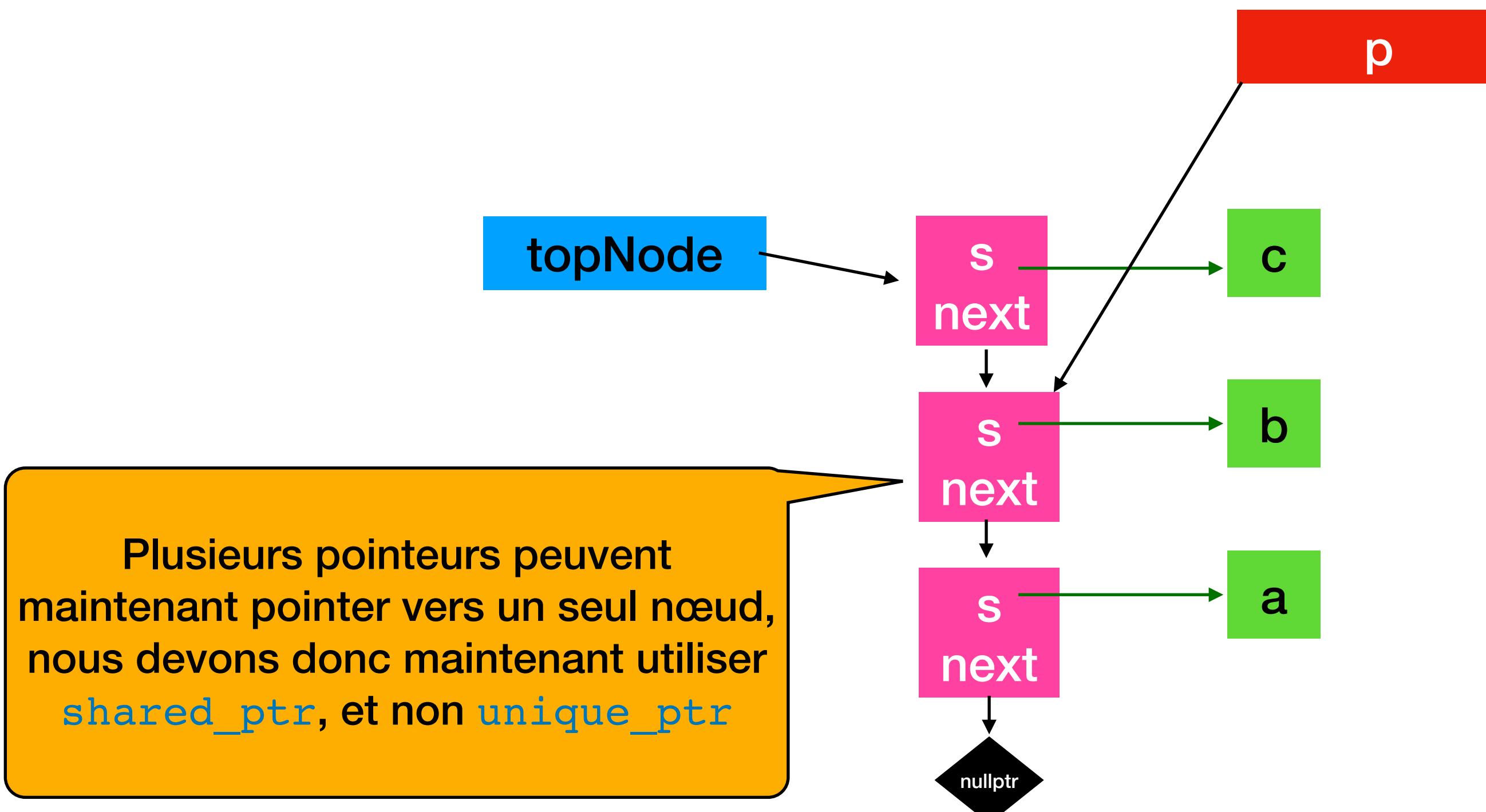
Créer une classe d'itérateur pour la pile

- Comment l'appeler ? Pourquoi pas `Iterator`
- Où le déclarer ? À l'intérieur de la pile. C'est donc `Stack::Iterator`
- Que doit-il stocker ? Un pointeur vers un nœud
- Quel genre de pointeur ? Un pointeur intelligent, bien sûr. Appelle ça `p`
- Quel type de pointeur intelligent ? L'itérateur pointera vers des nœuds tels que ceux pointés par `top` qui ont déjà un pointeur pointant vers eux. Ainsi `shared_ptr`. Tous les pointeurs de la classe `Stack` doivent maintenant être `shared_ptr`.
- `++` devrait juste déplacer `p` vers `p -> next`
- `*` doit renvoyer une référence aux données dans le nœud pointé vers : `p->data`
- `==` devrait juste comparer les deux pointeurs `p` en utilisant `==`. C'est le comportement par défaut de `==` si nous l'activons puisque `p` est la seule variable membre.

```
class Iterator{  
    shared_ptr<node> p;  
public:  
    Iterator(shared_ptr<node> p):p(p){};  
    T &operator *(){return p->data;};  
    Iterator &operator++(){p=p->next;return *this;};  
    bool operator ==(const Iterator&)const=default;  
};
```

Diagramme

Quatre types différents maintenant



Types de données

T

node

stack<T>

stringStack::Iterator

→ Variable d'instance normale

→ pointeur intelligent

begin() et end()

- Rappelez-vous que c'est le code que nous voulons travailler:

```
for (auto i=s.begin();
      i!=s.end();
      ++i)
cout<<*i<<endl;
```

- `s.begin()` doit représenter le premier élément de la pile, le `top`
- `s.end()` doit représenter la valeur au-delà de la fin, ce que nous obtenons en exécutant `++` sur l'itérateur de la dernière valeur. Dans notre cas, exécuter `++` sur la dernière valeur définit l'itérateur pour qu'il pointe vers le `next` du dernier nœud, qui est `nullptr`

```
template <typename T>
class Stack{
    struct node {
        T data;
        shared_ptr<node> next;
        node(const T &data,shared_ptr<node> &&next):
            data{data},next{move(next)}{}
    };
    shared_ptr<node> top;
public:
    class Iterator{
        shared_ptr<node> p;
    public:
        Iterator(shared_ptr<node> p):p{p}(){}
        T &operator *(){return p->data;}
        Iterator &operator++(){p=p->next;return *this;}
        bool operator ==(const Iterator&)const=default;
    };
    Iterator begin(){return Iterator{top};}
    Iterator end(){return Iterator{nullptr};}
    void push(const T &data){
        top=move(shared_ptr<node>{new node(data,move(top))});
    }
    T pop(){
        T s{top->data};
        auto tmp=move(top->next);
        top=move(tmp);
        return data;
    }
    bool isEmpty() {return top==nullptr;}
};
```

```

Stack<int> s;
for (auto i:{1,4,8,6,4}) s.push(i);
for (auto i=s.begin();i!=s.end();++i){
    cout<<*i<<endl;
}

```

4
6
8
4
1

```

for (auto i=begin(s);i!=end(s);++i){
    cout<<*i<<endl;
}

```

4
6
8
4
1

```

for (auto &i:s){
    cout<<i<<endl;
}

```

4
6
8
4
1

```

template <typename T>
class Stack{
    struct node {
        T data;
        shared_ptr<node> next;
        node(const T &data,shared_ptr<node> &&next):
            data{data},next{move(next)}{}
    };
    shared_ptr<node> top;
public:
    class Iterator{
        shared_ptr<node> p;
    public:
        Iterator(shared_ptr<node> p):p{p}(){}
        T &operator *(){return p->data;};
        Iterator &operator++(){p=p->next;return *this;};
        bool operator ==(const Iterator&)const=default;
    };
    Iterator begin(){return Iterator{top};};
    Iterator end(){return Iterator{nullptr};};
    void push(const T &data){
        top=move(shared_ptr<node>{new node(data,move(top))});
    }
    T pop(){
        T s{top->data};
        auto tmp=move(top->next);
        top=move(tmp);
        return data;
    }
    bool isEmpty() {return top==nullptr;}
};

```

```
Stack<int> s;  
for (auto &i:{1,4,8,6,4}) s.push(i);  
for (auto &i=s.begin();i!=s.end();++i){  
    cout<<*i<<endl;  
}
```

Le range-based `for` loop fonctionne également avec les listes d'initialisation

4
6

auto ici est `Stack<int>::Iterator`

4
1

auto ici est `Stack<int>::Iterator`

```
for (auto i=begin(s);i!=end(s);++i){  
    cout<<*i<<endl;  
}
```

`std::begin(s)` et `std::end(s)`
appellent `s.begin()` and `s.end()`

4
6
8

```
for (auto &i:s){  
    cout<<i<<endl;  
}
```

4
6
8

C'est appelé range-based `for` loop.

Cela fonctionne sur tout ce qui `begin` et `end` renvoient des itérateurs qui ont `++` `==` et `*` codé

auto ici est `int`. D'où est-ce que sa vient? Du type de
`*(s.begin())`

Il fonctionne également avec toutes les classes STL

Itérateurs : ce n'était qu'une introduction

- C++ définit un certain nombre de types d'itérateurs, avec des fonctionnalités spécifiques : input, output, forward, bidirectionnel, accès aléatoire
 - Les itérateurs bidirectionnels prennent en charge non seulement `i++` mais `i--`
 - Les itérateurs à accès aléatoire autorisent `i[10]` et `i+=10`
- Il existe des moyens de marquer un itérateur quant au type qu'il utilise par héritage (que nous n'avons pas encore couvert), de sorte qu'un algorithme peut faire différentes choses en fonction des types d'itérateur fournis.
- Les classes STL telles que `vector` fournissent :
 - `begin` et `end` (itération vers l'avant)
 - `rbegin` et `rend` (itérer en arrière)
 - `cbegin` et `cend` (itérer avec un itérateur constant, ne pas modifier les données)
 - `crbegin` et `crend` (itérateur constant inverse)

Exercice : deuxième

- Écrivez le code pour la `second` qui renvoie le deuxième élément de n'importe quel objet qui a un itérateur accessible à l'aide de `begin()` et `end()`
- Cela devrait fonctionner comme ceci :
- ```
Stack<int> s;
for (auto i:{1,2,3,4,5}) s.push(i);
cout<<"second of stack:"<<second(s)<<endl;
auto il={1,2,3,4,5};
cout<<"second of initializer list:"<<second(il)<<endl;
auto v=vector<string>({"first", "second", "third"});
cout<<"second of vector:"<<second(v)<<endl;
```
- Astuce : `auto` est un type de retour valide depuis C++17

```
second of stack:4
second of initializer list:2
second of vector:second
```

# **Héritage unique**

**Programmation orientée objet**

**Prof. John Iacono**

```

class Person{
 string firstName,lastName;
public:
 Person(string firstName,string lastName)
 :firstName(firstName),lastName(lastName){}
 const string &getFirstName() const {return firstName;}
 const string &getLastName() const {return lastName;}
 void print() const {cout<<lastName<<", "<<firstName<<endl;}
};

class Student{
 Person person;
 string program;
public:
 Student(string firstName,string lastName,string program)
 :person(firstName,lastName),program(program){}
 const string &getFirstName() const {return person.getFirstName();}
 const string &getLastName() const {return person.getLastName();}
 void print() const {cout<<person.getLastName()<<",
 "<<person.getFirstName()<<" ("<<program<<") "<<endl;}
};

```

Un étudiant A une personne  
A la place, nous voulons qu'un étudiant SOIT une personne

```

Person p{"Joe","Person"};
Student s{"Jane","Student","CS:BA2"};

p.print();
s.print();

```

Person, Joe  
Student, Jane (CS:BA2)

```

template <class T>
string lastNameAllCaps(const T& x){
 string toReturn{x.getLastName()};
 for (string::size_type i=0;i<toReturn.size();i++)
 toReturn[i]=toupper(toReturn[i]);
 return toReturn;
}

```

```

cout<<lastNameAllCaps(p)<<endl;
cout<<lastNameAllCaps(s)<<endl;

```

PERSON  
STUDENT

```

vector<Person> A;
A.push_back(p);
A.push_back(s); // ERROR!

```

19-inherit.cpp:54:4: error: no matching member function for call to 'push\_back'

Apparemment, les étudiants ne sont pas des People !

```

class Person{
 string firstName,lastName;
public:
 Person(string firstName,string lastName)
 :firstName(firstName),lastName(lastName){}
 const string &getFirstName() const {return firstName;}
 const string &getLastName() const {return lastName;}
 void print() const {cout<<lastName<< ", " <<firstName<<endl;}
};

```

Ces méthodes font désormais également partie de Person !

Et print?

Un étudiant hérite de Person.  
Un étudiant EST une personne  
Vous pouvez accéder à toutes les données/  
méthodes de la personne comme si elles  
étaient les vôtres (et publiques/protégées)

Nouveau

Vieux

```

class Student{
 Person person;
 string program;
public:
 Student(string firstName,string lastName,string program)
 :person(firstName,lastName),program(program){}
 const string &getFirstName() const {return person.getFirstName();}
 const string &getLastName() const {return person.getLastName();}
 void print() const {
 cout<<person.getLastName()<< ", "
 <<person.getFirstName()<< " (" <<program<<") "<<endl;}
};

```

Un étudiant a une variable  
membre appelée person de  
type Person

Le constructeur initialise Person

Avons besoin de person.

```
class Student:public Person{
```

```
 string program;
public:
```

Le constructeur initialise la personne dont nous héritons

```
 Student(string firstName,string lastName,string program)
 :Person(firstName,lastName),program(program){};
```

```
void print() const {
```

```
 cout<<getLastName()<< ", "
 <<getFirstName()<< " (" <<program<<") "<<endl;}
```

Peut appeler directement en tant  
qu'étudiant est une personne

Ici, nous avons dû coder ces méthodes,  
qui appelaient les méthodes du même  
nom sur Person.

```

class Person{
 string firstName,lastName;
public:
 Person(string firstName,string lastName)
 :firstName{firstName},lastName{lastName}{}
 const string &getFirstName() const {return firstName;}
 const string &getLastName() const {return lastName;}
 void print() const {cout<<lastName<<" , "<<firstName<<endl;}
};

```

```

class Student:public Person{
 string program;
public:
 Student(string firstName,string lastName,string program)
 :Person{firstName,lastName},program{program}{}
 void print() const {
 cout<<getLastName()<<" ,
 <<getFirstName()<<" ("<<program<<")"<<endl;
 }
};

```

```

Person p{"Joe","Person"};
Student s{"Jane","Student","CS:BA2"};

```

Person::print

```

p.print();
s.print();

```

Student::print

Person, Joe  
Student, Jane (CS:BA2)

```

Person &p2=s;

```

Une Student est une Person!

```

p2.print();

```

Person::Print

Student, Jane

Pouvons-nous passer cet appel Student::Print ? Après tout, p2 est vraiment un étudiant !

```

class Person{
 string firstName,lastName;
public:
 Person(string firstName,string lastName)
 :firstName{firstName},lastName{lastName}{}
 const string &getFirstName() const {return firstName;}
 const string &getLastName() const {return lastName;}
 virtual void print() {cout<<lastName
 <<", "<<firstName<<endl;}
 virtual ~Person(){}
};

```

Les classes avec des méthodes virtuelles ont besoin de destructeurs

Marquer une méthode virtuelle signifie que les méthodes seront appelées en fonction du type de l'instance, et non du type du pointeur/référence

```

class Student:public Person{
 string program;
public:
 Student(string firstName,string lastName,string program)
 :Person(firstName,lastName),program{program}{}
 void print() {cout<<getLastName()<<", "<<
 getFirstName()<<" ("<<program<<")"<<endl;}
};

```

```

Person p{"Joe","Person"};
Student s{"Jane","Student","CS:BA2"};

```

Person::print

```

p.print();
s.print();

```

Student::print

Person, Joe  
Student, Jane (CS:BA2)

```

Person &p2=s;

```

Une Student est une Person!

```

p2.print();

```

Student::print

~~Student, Jane~~

Student, Jane (CS:BA2)

```

vector< shared_ptr<Person> > A;
for (int i=1;i<5;++i){

```

A.push\_back(make\_shared<Person>(

"Person",to\_string(i)));

A.push\_back(make\_shared<Student>(

"Student",to\_string(i),"BS:BA2"));

}

cout<<"Printing all in A"<<endl;

for (auto &x:A) x->print();

Printing all in A  
1, Person

1, Student (BS:BA2)  
2, Person

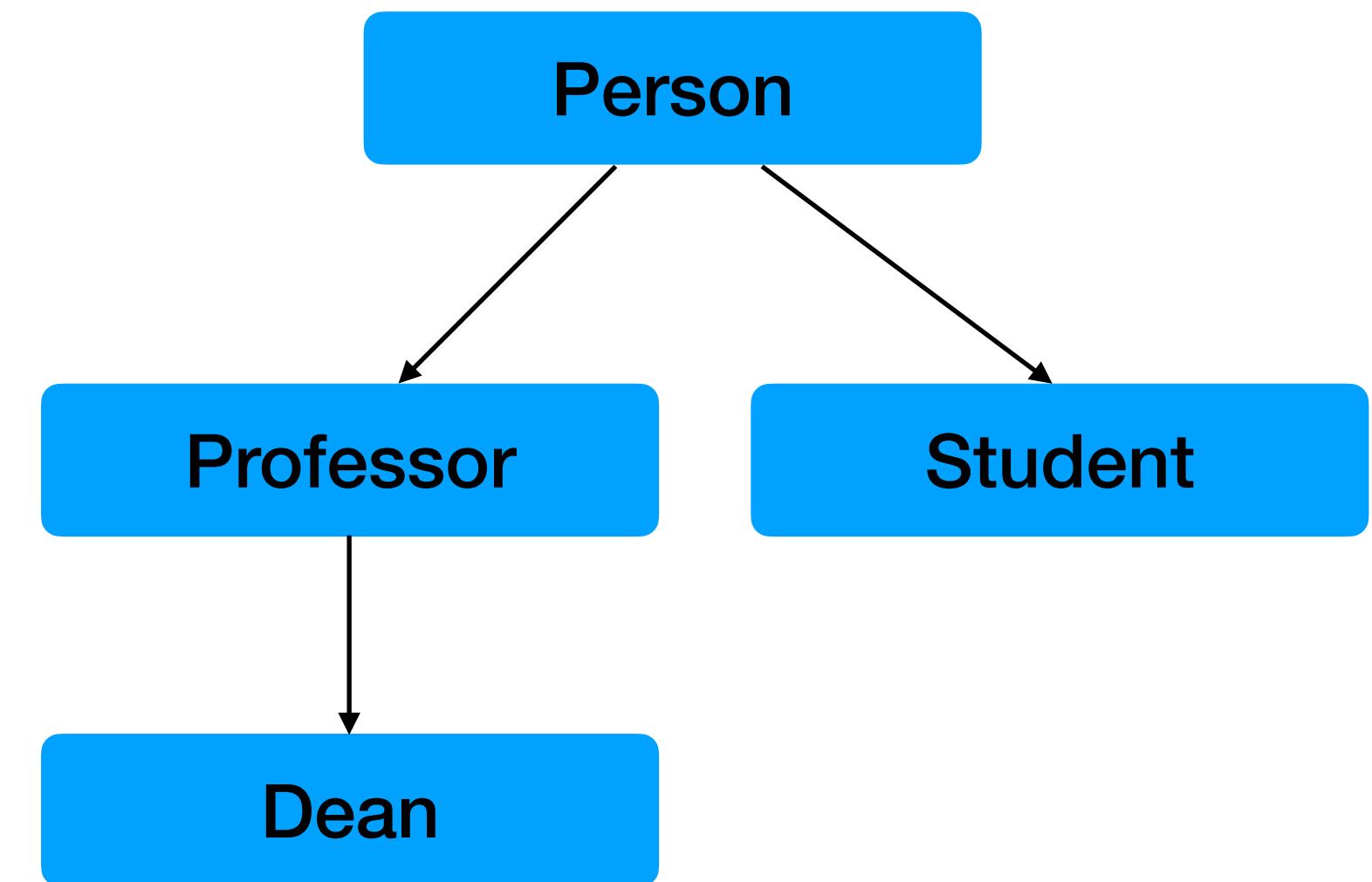
2, Student (BS:BA2)  
3, Person

3, Student (BS:BA2)  
4, Person

4, Student (BS:BA2)

# Terminologie

- Il s'agit d'une hiérarchie de classes.
- **Person** est la classe de base de la hiérarchie
- **Student** est une classe enfant/classe dérivée/sous-classe de **Person**
- **Person** est la classe parente/classe de base/superclasse de **Student**
- **Professor**, **Student** et le **Dean** sont des classes descendantes de la **Person**
- **Professeur** et **Person** sont des classes ancêtres de **Dean**
- Conversion d'étudiant en personne est upcasting
- Convertir une personne en étudiant est downcasting



# Conversion

- Nous supposons que `Student` est une sous-classe de `Person`
- Un `Student` peut être converti en `Person` mais sera "tranché". C'est-à-dire qu'il perdra toutes les méthodes et données spécifiques à `Student`. C'est souvent mauvais.
- Un `Student *` peut être pointé vers un `Person *`. Les méthodes virtuelles seront appelées en utilisant le type réel et non le type du pointeur.
- Ce qui précède est une conséquence du fait que les variables d'un type donné sont de taille fixe. Ainsi, mettre une variable `Student` dans une `Person` signifie qu'il n'y a pas d'espace pour la variable d'instance de programme de `Student` et qu'elle doit être coupée. Il n'y a pas de problème avec les pointeurs car tous les pointeurs ont la même taille (bien que la taille de ce vers quoi ils pointent puisse être différente).
- Une `Person *` qui pointe vers un `Student` peut être convertie en `Student *` en utilisant `dynamic_cast` (ou `dynamic_pointer_cast` pour un pointeur intelligent). Ce cast renverra `nullptr` si l'objet n'est pas vraiment un `Student *`
- Les références se comportent comme des pointeurs.

```

vector< shared_ptr<Person> > A;
for (int i=1;i<5;++i){
 A.push_back(make_shared<Person>(
 "Person",to_string(i)));
 A.push_back(make_shared<Student>(
 "Student",to_string(i),"BS:BA2"));
}
cout<<"Printing all in A"=>endl;
for (auto &x:A) x->print();

```

Pas un pointeur.

A peut stocker des pointeurs vers la personne et l'étudiant

```

vector< Person > A2;
for (int i=1;i<5;++i){
 A2.push_back(Person{
 "Person",to_string(i)});
 A2.push_back(Student{
 "Student",to_string(i),"BS:BA2"});
}
cout<<"Printing all in A2 (sliced)"=>endl;
for (auto &x:A2) x.print();

```

Les informations du programme sont perdues

L'ajout des étudiants au vecteur de la personne les coupe

Printing all in A2 (sliced)  
1, Person  
1, Student  
2, Person  
2, Student  
3, Person  
3, Student  
4, Person  
4, Student

Seulement Person::Draw est appelé.

```

vector< shared_ptr<Person> > A;
for (int i=1;i<5;++i){
 A.push_back(make_shared<Person>(
 "Person",to_string(i)));
 A.push_back(make_shared<Student>(
 "Student",to_string(i),"BS:BA2"));
}

cout<<"Printing students only"=>endl;
for (auto &x:A) {
 shared_ptr<Student> s=
 dynamic_pointer_cast<Student>(x);
 if (s) s->print();
}

Printing students only
1, Student (BS:BA2)
2, Student (BS:BA2)
3, Student (BS:BA2)
4, Student (BS:BA2)

```

```

class Animal {
protected:
 string name;
public:
 virtual void makeSound() const=0;
 Animal(const string &name):name{name}(){}
 const string getName() const {return name;}
};

class Duck:public Animal {
public:
 using Animal::Animal;
 void makeSound() const override
 {cout<<name<<" says quack"<<endl;}
};

class Cow:public Animal {
public:
 using Animal::Animal;
 void makeSound() const override
 {cout<<name<<" says moo"<<endl;}
};

```

Ici makeSound est défini, donc Duck n'est pas abstrait

Cela fonctionne avec n'importe quelle classe qui a une méthode makeSound, y compris Mushroom

Animal est ce qu'on appelle une classe abstraite, ce qui signifie que vous ne pouvez pas créer un objet de type Animal

Cela dit qu'il n'y a AUCUNE implémentation de makeSound

```
Animal a{"Archie"};
```

**19-inherit.cpp:207:9: error: variable type 'n4::Animal' is an abstract class**

```
Animal a{"Archie"};
```

```

class Mushroom {
public:
 void makeSound() const
 {cout<<"shroom"<<endl;}
};

Duck d{"Roger"};
Cow c{"Camilla"};
Mushroom m;

```

N'hérite pas d'Animal

Cela ne fonctionne qu'avec Animal (et les classes dérivées d'Animal). Pas de champignon !

```

template <class T>
void soundFiveTimesV1(const T &x){
 for (int i=0;i<5;i++)
 x.makeSound();
}

```

```

soundFiveTimesV1(d);
soundFiveTimesV1(m);

```

```

void soundFiveTimesV2(const Animal &x){
 for (int i=0;i<5;i++)
 x.makeSound();
}

```

```

soundFiveTimesV2(d);
soundFiveTimesV2(m); //ERROR

```

```

Roger says quack
shroom
shroom
shroom
shroom
shroom

```

```

void allMakeSound(vector <Animal*> A)
{for (auto &a:A) a->makeSound();}

allMakeSound({&d,&c,&d,&c});

```

Nous pouvons combiner des canards et des vaches car ce sont tous des animaux

```

Roger says quack
Camilla says moo
Roger says quack
Camilla says moo

```

# Override

Override n'entraîne pas le override. Ce qu'il fait, c'est que si le remplacement ne se produit pas, il génère une erreur de compilateur afin que vous puissiez trouver ce que vous avez fait

```
class Person{
 string firstName,lastName;
public:
 Person(string firstName,string lastName)
 :firstName(firstName),lastName(lastName){}
 const string &getFirstName() const {return firstName;}
 const string &getLastName() const {return lastName;}
 virtual void print() const {cout<<lastName<<", "<<firstName<<endl;}
 virtual ~Person(){}
};

class Student:public Person{
 string program;
public:
 Student(string firstName, string lastName, string program)
 :Person(firstName,lastName),program(program){}
 void print() {cout<<lastName<<", "<<firstName<<" ("<<program<<")"<<endl;}
};
```

Ce print est const

J'aurais dû ajouter override ici. Vous devez ajouter override chaque fois que vous avez l'intention de remplacer une méthode

Ce print n'est pas const. À quelques exceptions près, les signatures de fonction doivent être exactement les mêmes pour remplacer

```
Student s("Joe","Student","INFOMA1");
Person &p=s;
p.print();
```

Student, Joe

```
19-inherit.cpp:244:7: warning:
'n5::Student::print' hides overloaded
virtual function [-Woverloaded-virtual]
 void print()
{cout<<getLastName()<<",
"<<getFirstName()<<" ("<<program<<")"<<en
dl;} ^

19-inherit.cpp:235:15: note: hidden
overloaded virtual function
'n5::Person::print' declared here:
different qualifiers ('const' vs
unqualified)
 virtual void print() const
{cout<<lastName<<", "<<firstName<<endl;} ^

1 warning generated.
```

# Contrôle d'accès

- `public` : en accès libre
- `protected` : accessible depuis les classes descendantes
- `private` : accessible uniquement depuis cette classe et ses amis, pas depuis les classes descendantes
- `class Student : public Person` public/privé/protégé en `Person` est le même en `Student`
- `class Student : private Person` Tout en `Person` est privé dans `Student`
- `class Student : protected Person` Tout ce qui est en `Person` qui est public devient privé dans `Student`, le reste reste le même
- Règle générale, en particulier pour les grands projets, rendre les données privées

# Autres détails

## Choses que vous pouvez voir

- `using` : utilisé pour apporter quelque chose de l'espace de noms d'un ancêtre dans l'espace de noms actuel. Souvent utile pour les constructeurs. Exemple sur la diapositive suivante
- `final` : utilisé pour indiquer qu'une méthode ne peut jamais être remplacée dans une classe descendante.
- Incompatibilité de type de retour. Lorsque vous remplacez, les types de retour doivent correspondre, à l'exception du fait que les types de retour peuvent être "promus". Par exemple, si une méthode de `Person` a renvoyé une `Person`, la même méthode de `Student` pourrait renvoyer un `Student`.
- `typeid(Classname)` peut être utilisé pour obtenir des informations sur le type. Habituellement `dynamic_cast` est mieux car généralement vous ne voulez pas savoir si un objet est EXACTEMENT un certain type, vous voulez savoir s'il peut être converti en un certain type.

# Utilisation de Using

## Créer un vecteur qui vérifie la plage lorsque vous utilisez A[ ]

```
template <typename T>
class SafeVector: vector<T> {
public:
 T &operator[](size_type i){return this->at(i);}
 const T&operator [](size_type i) const {return this->at(i);}
};
```

```
template <typename T>
class SafeVector: vector<T> {
public:
 using size_type=typename vector<T>::size_type;
 T &operator[](size_type i){return this->at(i);}
 const T&operator [](size_type i) const {return this->at(i);}
};
```

```
template <typename T>
class SafeVector: vector<T> {
public:
 using size_type=typename vector<T>::size_type;
 using vector<T>::vector;
 T &operator[](size_type i){return this->at(i);}
 const T&operator [](size_type i) const {return this->at(i);}
};
```

```
19-inherit.cpp:264:16: error: unknown type name 'size_type'
 T &operator[](size_type i){return this->at(i);}

SafeVector<int> A={1,2,3,4,5};
A[100]=6;
```

```
19-inherit.cpp:269:18: error: no matching constructor
for initialization of 'SafeVector<int>'
 SafeVector<int> A={1,2,3,4,5};
```

```
SafeVector<int> A={1,2,3,4,5};
A[100]=6;
```

Compiles!

```
libc++abi: terminating with uncaught exception of type
std::out_of_range: vector!
```

# 3 concepts distincts

```
class B {
 void someMethod();

}

class A {
 B b;

 void someMethod(){
 b.someMethod();
 }
}
```

```
class A{
 class B {
 void someMethod();

 }
 B b;
 void someMethod(){
 b.someMethod();

 }

A::B b;
```

```
class B {
 void someMethod();

}

class A:public B{
 void someMethod(){
 B::someMethod();

 }
}
```

A et B sont des classes séparées

A a un B

B est une classe interne de A

A a un B

En dehors de A, ne peut pas utiliser uniquement B, doit être B::A

A est une sous-classe de B

A est un B

# Sommaire

- Le polymorphisme est un outil puissant et l'une des fonctionnalités clés du C++ ou d'autres langages orientés objet. Il s'agit de pouvoir utiliser un objet sans connaître son type exact.
- Appelez le constructeur de la classe parente dans la liste d'initialisation.
- Classes avec méthodes virtuelles
  - Besoin de savoir de quelle classe ils sont. Ces informations de type stockées dans des objets avec des méthodes virtuelles sont appelées RTTI (Runtime information).
  - Cela les rend légèrement plus grandes et l'appel de méthodes virtuelles prend un peu plus de temps car au moment de l'exécution, le type doit être déterminé.
  - Besoin d'avoir un destructeur virtuel
- Conversion
  - La conversion en classe parent (upcasting) peut toujours être effectuée
  - La conversion en une classe descendante (downcasting) peut être effectuée en toute sécurité avec `dynamic_cast`. Cela ne fonctionne qu'avec RTTI.
- `override` ne remplace rien. Il génère une erreur si la méthode ne remplace pas. C'est facultatif mais fortement recommandé.
- contrôle d'accès disponible via `public/private/protected`
- Sans `virtual`, une méthode est appelée selon le type de la variable. Avec `virtual`, une méthode est appelée en fonction du type de l'objet. Une méthode qui remplace une méthode virtuelle est virtuelle ; le keyword `virtuel` est facultatif et n'a aucun effet dans ce cas.

# **Héritage multiple**

## **Programmation orientée objet**

**Prof. John Iacono**

```
class Stringable{
public:
 virtual string toString() const=0;
};
```

Stringable est abstrait et n'a pas de données

```
class Floatable{
public:
 virtual float toFloat() const=0;
};
```

```
class Person:public Stringable{
 string firstName,lastName;
public:
 Person(string firstName,string lastName);
 string toString() const override;
};
```

```
class Fraction:public Stringable,public Floatable{
 int n;
 int d;
 void reduce();
public:
 Fraction(int n,int d);
 string toString() const override;
 float toFloat() const override;
 ...
};
```

```
class FeetInches:public Stringable,public Floatable{
 int feet;
 int inches;
public:
 FeetInches(int feet,int inches);
 string toString() const override;
 float toFloat() const override;
};
```

```
ostream &operator <<(ostream &os, const Stringable &x){
 return os<<x.toString();
}
```

std::cout est une instance de type std::ostream

```
Fraction f{2,3};
FeetInches fi{6,10};
Person p{"Joe", "Smith"};
```

```
for (auto &x:initializer_list<Stringable*>{&f,&fi,&p})
 cout<<x<<endl;
```

2/3  
6'10"  
Smith, Joe

```
for (auto &x:initializer_list<Floatable*>{&f,&fi})
 cout<<x->toFloat()<<endl;
```

0.666667  
6.83333

Maintenant, tout ce qui est un Stringable peut être imprimé en

# Héritage multiple

Cet exemple était facile (mais commun)

- En général, vous devez appeler les constructeurs de toutes les classes parentes (dans l'exemple, nous avons utilisé des constructeurs par défaut). Ils seront appelés dans l'ordre où ils apparaissent
- Les destructeurs sont appelés dans l'ordre inverse des constructeurs
- Que se passe-t-il si les deux classes parentes ont des méthodes ou des variables d'instance avec le même nom ?
  - Solution : ambigu = erreur
- Que faire si plusieurs classes parentes héritent de la même classe (le problème du diamant)
  - C'est compliqué. Détails à suivre.

```

class Person{
 string firstName,lastName;
public:
 Person(string firstName,string lastName)
 :firstName(firstName),lastName(lastName){}
 const string &getFirstName() const {return firstName;}
 const string &getLastName() const {return lastName;}
 void setFirstName(const string &newFirstName) {firstName=newFirstName;}
 void setLastName(const string &newLastName) {lastName=newLastName;}
 virtual void print() const {cout<<lastName<<, " <<firstName<<endl;}
 virtual ~Person(){}
};

```

```

class Student:public Person{
 string program;
public:
 Student(string firstName,string lastName,string program)
 :Person(firstName,lastName),program(program){};
 void print() const {
 cout<<getLastName()<<, " <<getFirstName()<<" (" <<program<<")<<endl;
 };

```

```

class Employee:public Person{
 int salary;
public:
 int getSalary() const {return salary;}
 Employee(string firstName,string lastName,int salary)
 :Person(firstName,lastName),salary(salary){}
 void print() const {
 cout<<getLastName()<<, " <<getFirstName()<<" [€]<<salary<<"]<<endl;
 };

```

```

class StudentWorker:public Student,public Employee{
public:
 StudentWorker(string firstName,string lastName,string program,int salary)
 :Student(firstName,lastName,program),Employee(firstName,lastName,salary){}
};

```

Comme un StudentWorker est un Student et un Employee, il appelle les constructeurs de ces deux

StudentWorker j{"John", "Iacono", "MSCS", 1000};  
j.print(); //ERROR  
20-multiple.cpp:53:10: error: member 'print'  
found in multiple base classes of different  
types

Ambigu

j.Student::print();  
j.Employee::print();

OK

Iacono, John (MSCS)  
Iacono, John [€1000]

```

class Person{
 string firstName,lastName;
public:
 Person(string firstName,string lastName)
 :firstName(firstName),lastName(lastName){}
 const string &getFirstName() const {return firstName;}
 const string &getLastName() const {return lastName;}
 void setFirstName(const string &newFirstName) {firstName=newFirstName;}
 void setLastName(const string &newLastName) {lastName=newLastName;}
 virtual void print() const {cout<<lastName<<, " <<firstName<<endl;}
 virtual ~Person(){}
};

```

```

class Student:public Person{
 string program;
public:
 Student(string firstName,string lastName,string program)
 :Person(firstName,lastName),program(program){}
 void print() const {
 cout<<getLastName()<<, " <<getFirstName()<<" (" <<program<<")<<endl;}
};

```

```

class Employee:public Person{
 int salary;
public:
 int getSalary() const {return salary;}
 Employee(string firstName,string lastName,int salary)
 :Person(firstName,lastName),salary(salary){}
 void print() const {
 cout<<getLastName()<<, " <<getFirstName()<<" [€]<<salary<<]<<endl;}
};

```

```

class StudentWorker:public Student,public Employee{
public:
 StudentWorker(string firstName,string lastName,string program,int salary)
 :Student(firstName,lastName,program),Employee(firstName,lastName,salary){}
};

```

StudentWorker j{"John", "Iacono", "MSCS", 1000};  
 cout<<j.getFirstName()<<endl;//ERROR  
 20-multiple.cpp:56:10: error: non-static  
 member 'getFirstName' found in multiple base-  
 class subobjects of type 'n1::Person':

Ambigu!  
 Pourquoi??

```

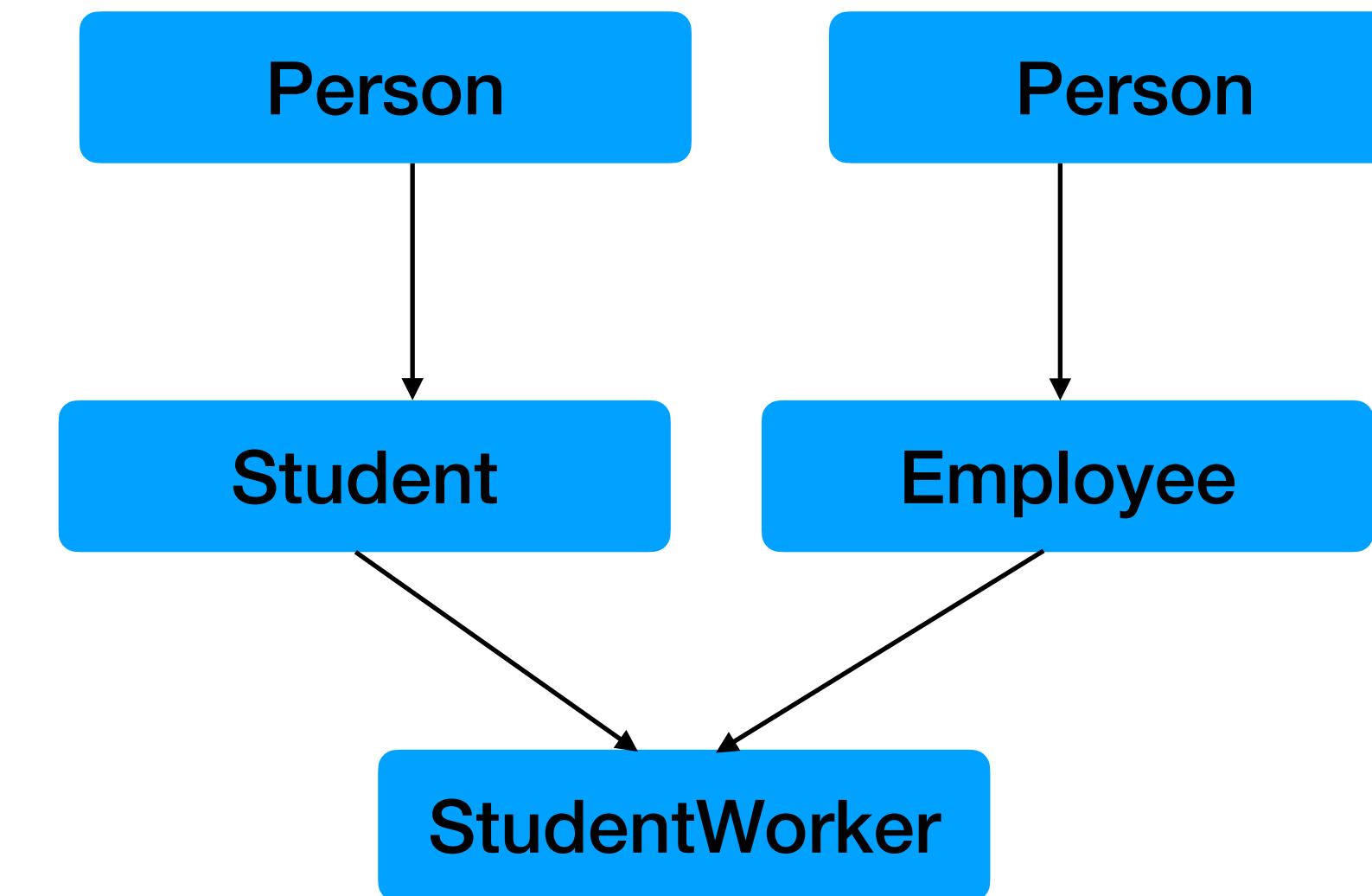
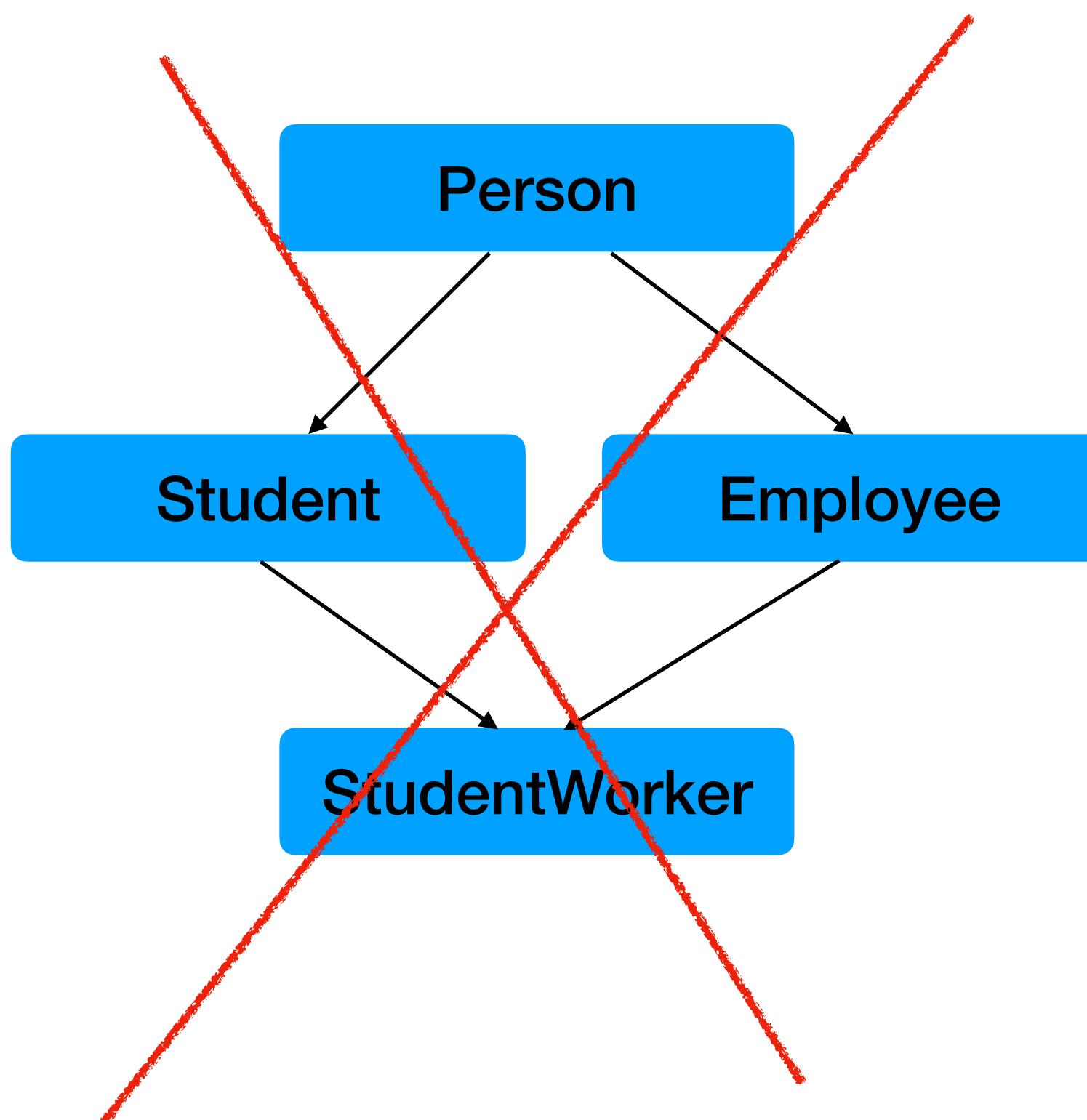
cout<<j.Student::getFirstName()<<endl;
cout<<j.Employee::getFirstName()<<endl;
j.Student::setFirstName("Richard");
cout<<j.Student::getFirstName()<<endl;
cout<<j.Employee::getFirstName()<<endl;

```

|         |
|---------|
| John    |
| John    |
| Richard |
| John    |

Qu'est-ce qui se passe ici???

# Nous avons la mauvaise image



Il y a DEUX personnes à l'intérieur de Studentworker !

```

class Person{
 string firstName,lastName;
public:
 Person(string firstName,string lastName)
 :firstName(firstName),lastName(lastName){cout<<"Constructor Person";}
 const string &getFirstName() const {return firstName;}
 const string &getLastName() const {return lastName;}
 void setFirstName(const string &newFirstName) {firstName=newFirstName;}
 void setLastName(const string &newLastName) {lastName=newLastName;}
 virtual void print() const {cout<<lastName<<, " <<firstName<<endl;}
 virtual ~Person(){}
};

```

```

class Student:public Person{
 string program;
public:
 Student(string firstName,string lastName,string program)
 :Person(firstName,lastName),program(program){cout<<"Constructor Student";}
 void print() const {
 cout<<getLastName()<<, "<<getFirstName()<< (" <<program<< ")" <<endl;}
}

```

```

class Employee:public Person{
 int salary;
public:
 int getSalary() const {return salary;}
 Employee(string firstName,string lastName,int salary)
 :Person(firstName,lastName),salary(salary){cout<<"Constructor Employee";}
 void print() const {
 cout<<getLastName()<<, "<<getFirstName()<< [€"<<salary<<]" <<endl;}
}

```

```

class StudentWorker:public Student,public Employee{
public:
 StudentWorker(string firstName,string lastName,string program,int salary)
 :Student(firstName,lastName,program),Employee(firstName,lastName,salary){
 cout<<"Constructor StudentWorker";}
}

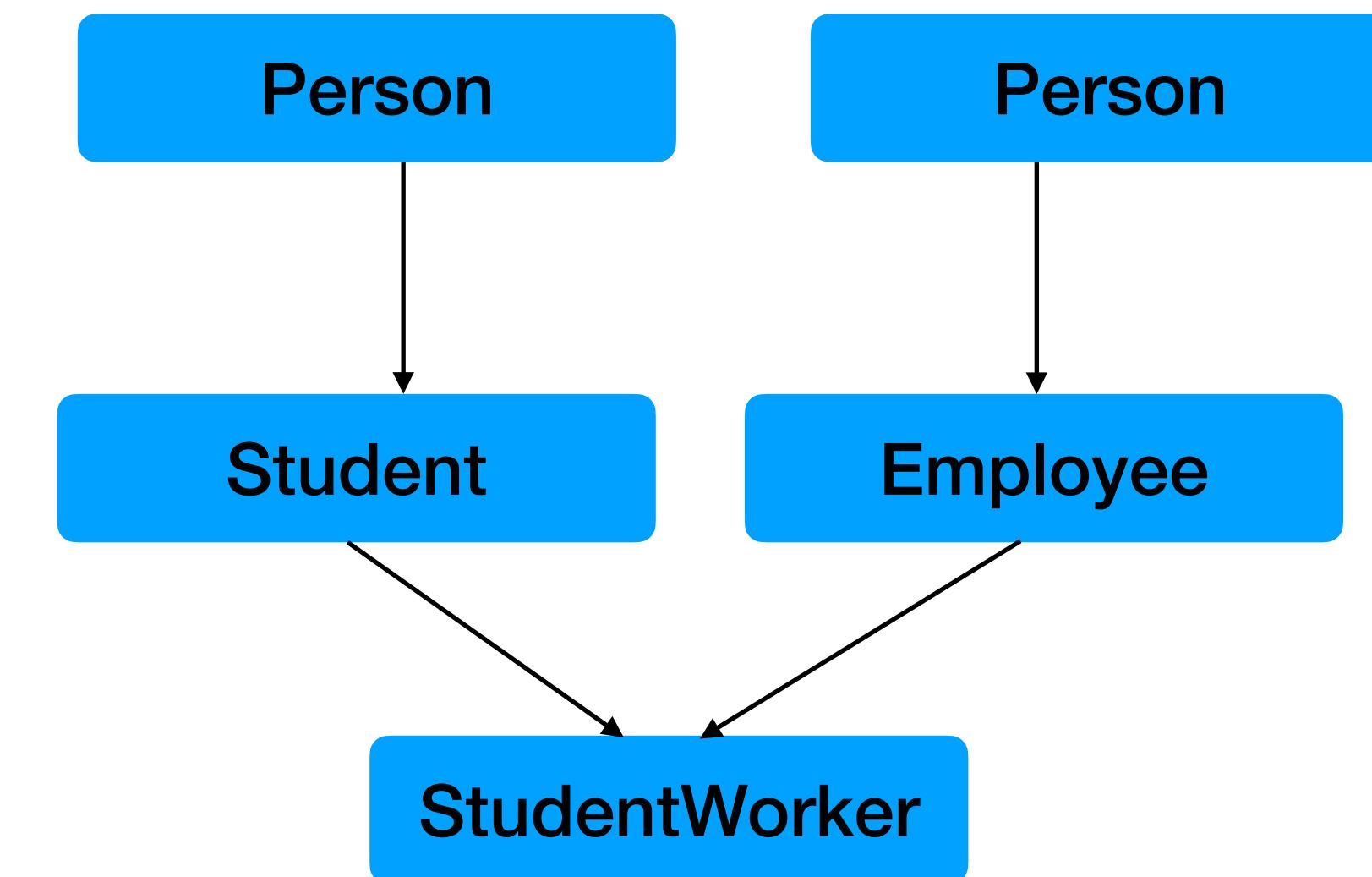
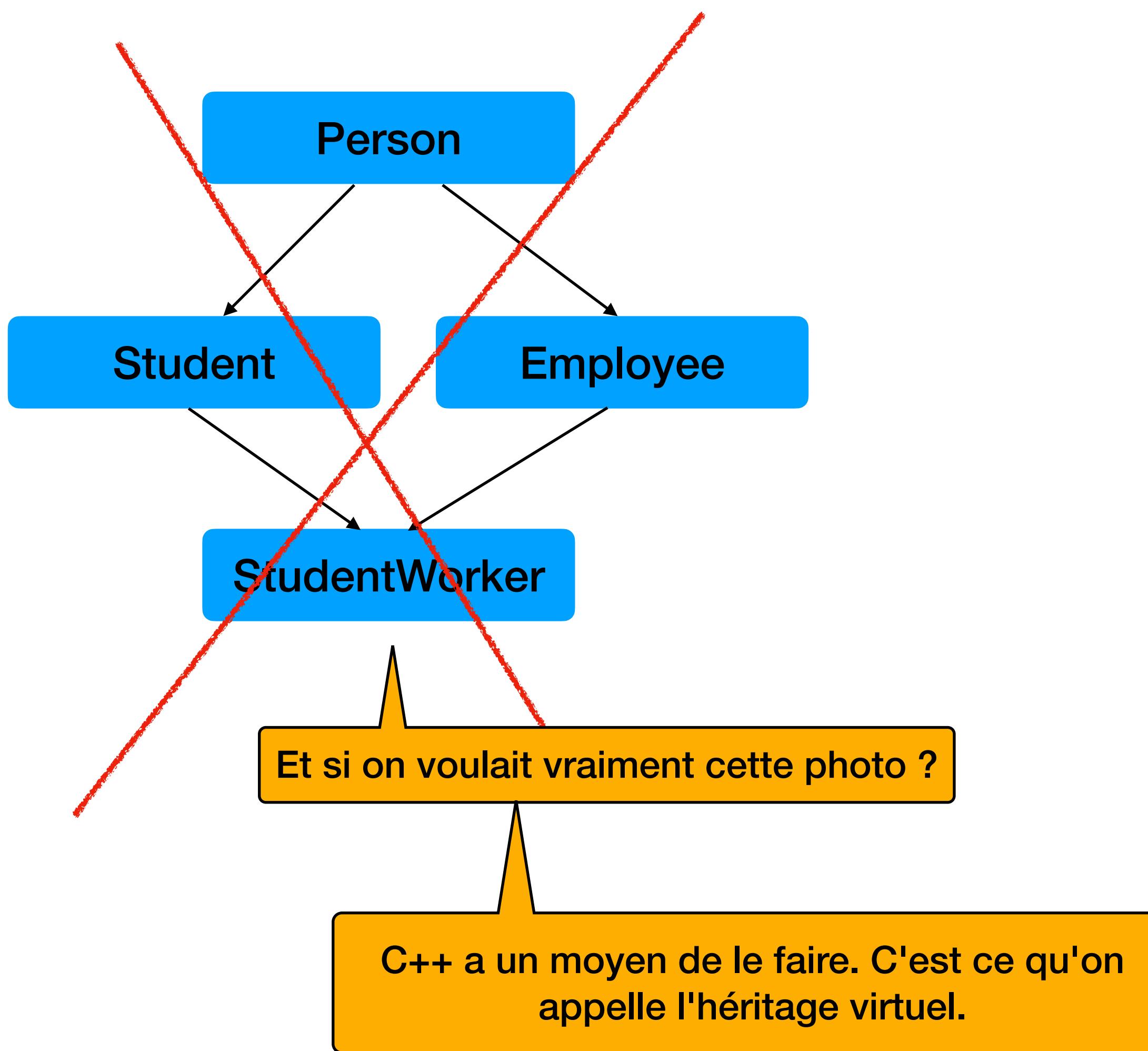
```

```
StudentWorker j{"John", "Iacono", "MSCS", 1000};
```

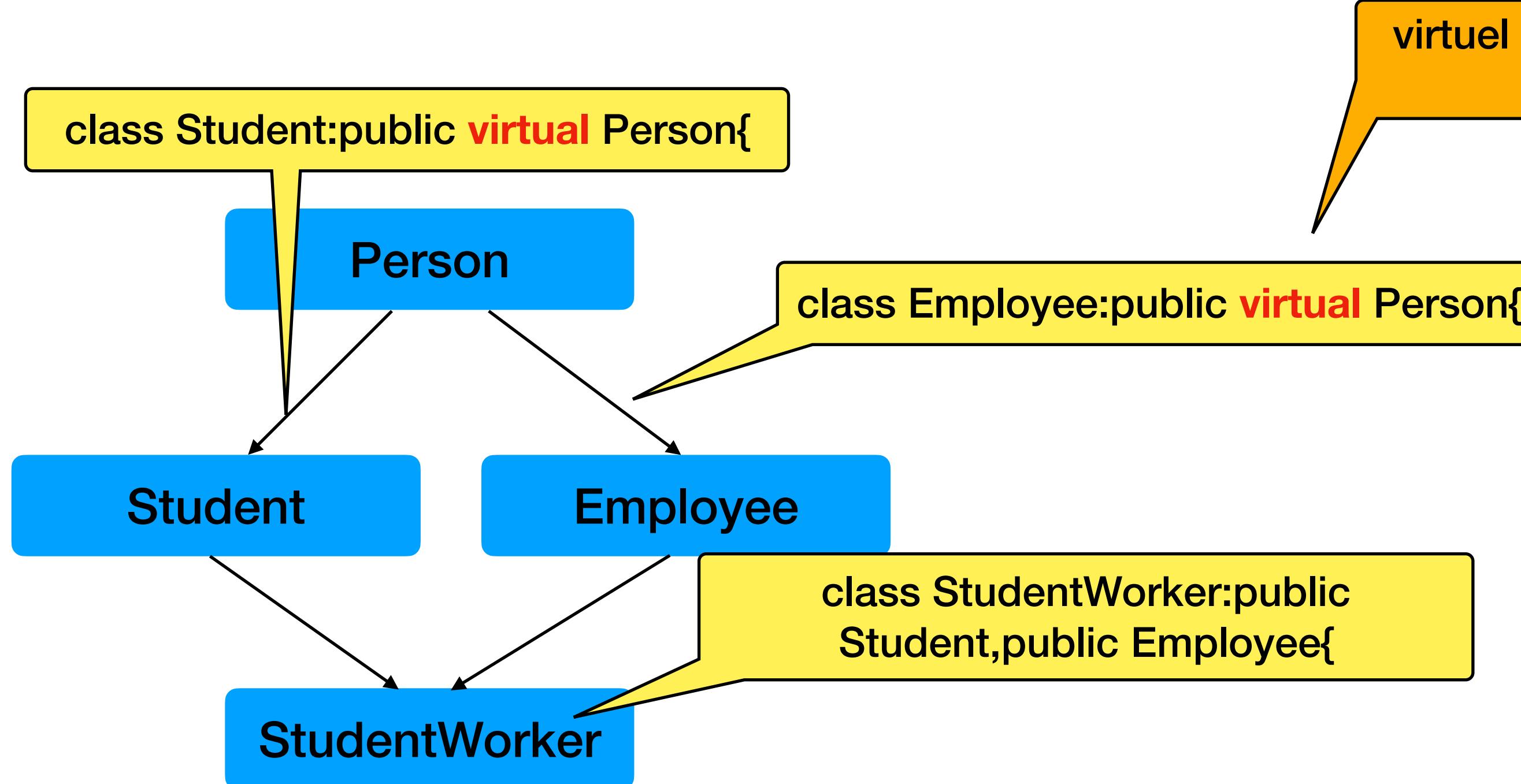
**Constructor Person**  
**Constructor Student**  
**Constructor Person**  
**Constructor Employee**  
**Constructor StudentWorker**

Le constructeur de Person a été appelé deux fois.  
 Une fois pour la Person qui est le parent de Student  
 Une fois pour la Person qui est le parent de Employee

# Héritage virtuel



# Héritage virtuel



virtuel signifie qu'une seule personne  
être présent

Qui est chargé d'appeler le constructeur  
de Person ?

- Pour une Person, elle a son propre constructeur
- Pour un Student, il appelle le constructeur de Person
- Pour un Employee, il appelle le constructeur de Person
- Pour un StudentWorker, il DOIT appeler le constructeur de Person, et cela se produit avant l'appel du constructeur à Student ou Employee. Lorsque Student et Employee sont construits, leurs appels de constructeur à Person sont supprimés car Person existe déjà.

```

class Person{
 string firstName,lastName;
public:
 Person(string firstName,string lastName)
 :firstName(firstName),lastName(lastName){cout<<"Constructor Person";}
 const string &getFirstName() const {return firstName;}
 const string &getLastName() const {return lastName;}
 void setFirstName(const string &newFirstName) {firstName=newFirstName;}
 void setLastName(const string &newLastName) {lastName=newLastName;}
 virtual void print() const {cout<<lastName<<, " <<firstName<<endl;}
 virtual ~Person(){}
};
```

Nouveau

```

class Student:public virtual Person{
 string program;
public:
 Ceci est ignoré lors de la construction d'un StudentWorker
 Student(string firstName,string lastName,string program)
 :Person(firstName,lastName),program(program){cout<<"Constructor Student";}
 void print() const {
 cout<<getLastName()<<, " <<getFirstName()<< " (<<program<<)"<<endl;}
};
```

```
class Employee:public virtual Person{
```

```

 int salary;
public:
 int getSalary() const {return salary;}
 Employee(string firstName,string lastName,int salary)
 :Person(firstName,lastName),salary(salary){cout<<"Constructor Employee";}
 void print() const {
 cout<<getLastName()<<, " <<getFirstName()<< [€]<<salary<<]"<<endl;}
};
```

```
StudentWorker j{"John", "Iacono", "MSCS", 1000};
```

Person n'est construite qu'une seule fois

Constructor Person  
Constructor Student  
Constructor Employee  
Constructor StudentWorker

```

class StudentWorker:public Student,public Employee{
Nouveau:Person(firstName,lastName),
 Student(firstName,lastName,program),Employee(firstName,lastName,salary){
 cout<<"Constructor StudentWorker";}
};
```

Ce n'est plus une erreur

```
cout<<j.getFirstName(); Richard
```

John  
John

Un seul prénom

Richard  
Richard

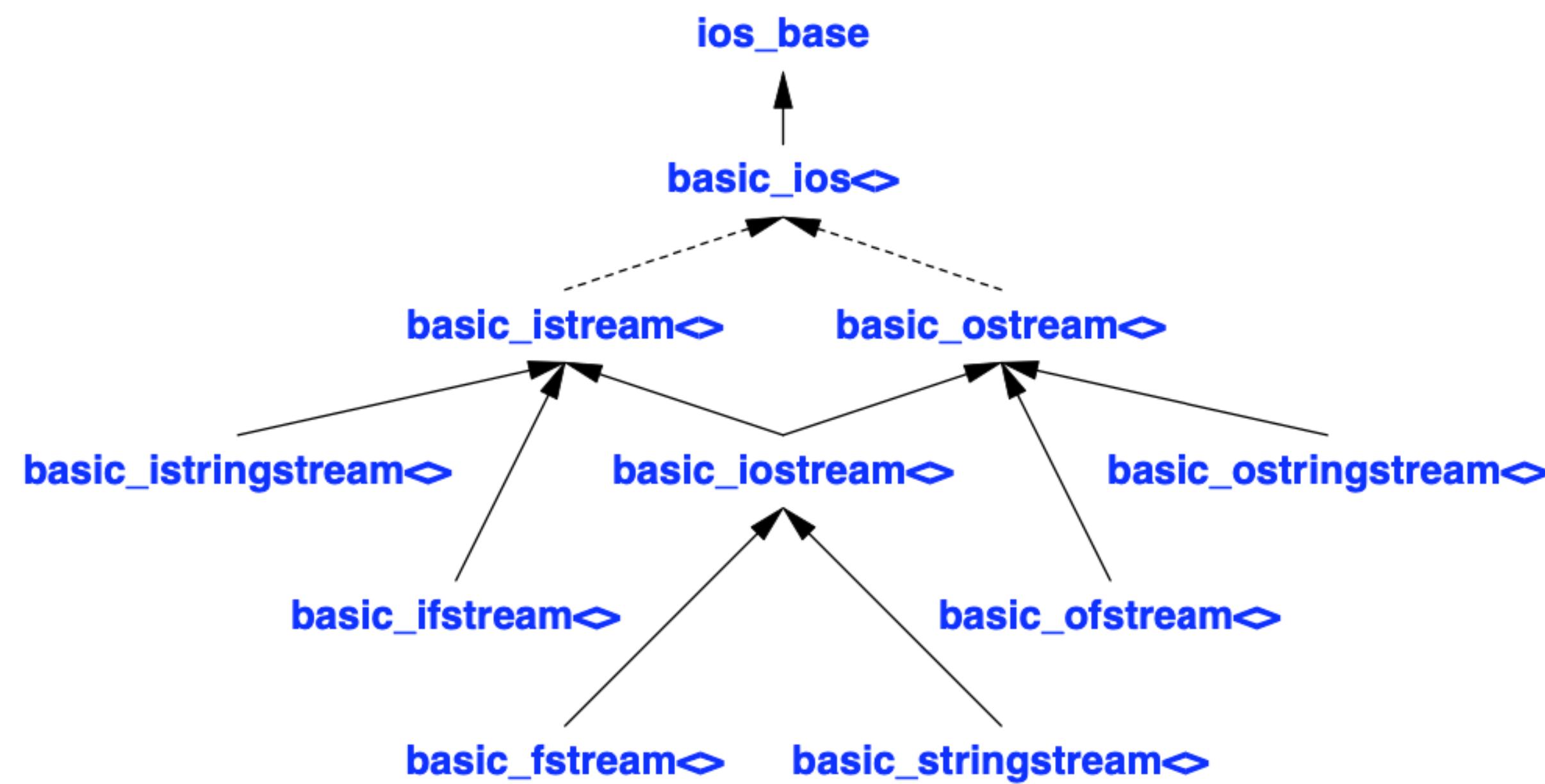
# Sommaire

- L'héritage multiple fonctionne en grande partie comme l'héritage simple
- L'ambiguïté conduit à une erreur de compilation
- « Problème de diamant » : C++ ne permet d'hériter qu'une seule classe via l'héritage virtuel. Dans ce cas, la construction de la classe partagée est de la responsabilité de la classe la plus élevée qui possède tous les chemins d'héritage vers la classe partagée.
- Les classes virtuellement héritées sont construites en premier

# Un exemple d'héritage multiple

## 38.2 The I/O Stream Hierarchy

An **istream** can be connected to an input device (e.g., a keyboard), a file, or a **string**. Similarly, an **ostream** can be connected to an output device (e.g., a text window or an HTML engine), a file, or a **string**. The I/O stream facilities are organized in a class hierarchy:



# **Patrons de conception : introduction et observateur**

**Programmation orientée objet**

**Prof. John Iacono**

# Concepts de niveau supérieur

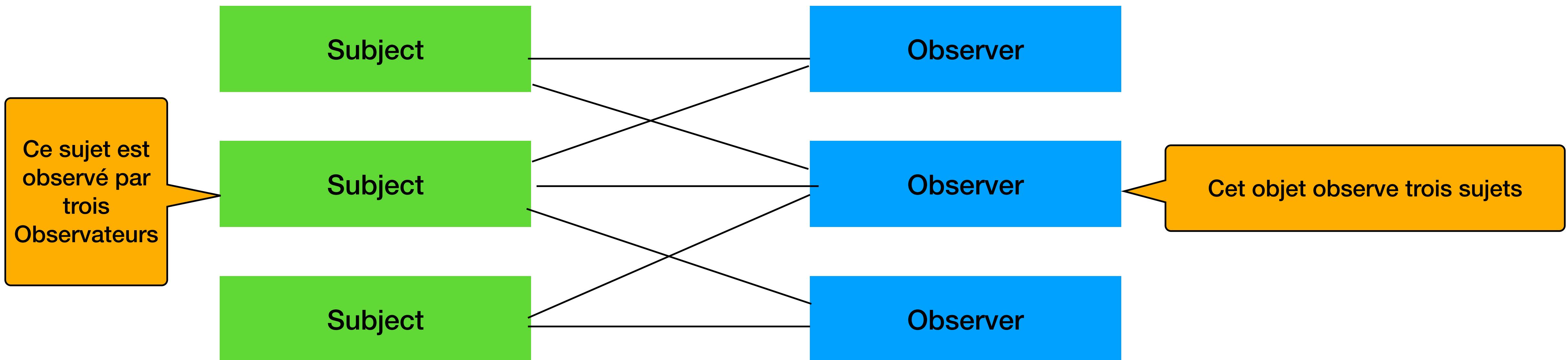
- Nous avons fini de parler de la mécanique du C++
- Vous devez connaître toutes les fonctionnalités fondamentales et comment les utiliser
- Il y a bien sûr beaucoup de choses que vous (et moi) ne savons pas, en particulier dans la bibliothèque standard. Tout ce que vous voyez de nouveau dans le code C++, recherchez-le de la même manière que vous chercheriez un nouveau mot lors de l'apprentissage d'une langue. Par exemple: `std::optional`
- Nous allons maintenant parler des problèmes de conception de niveau supérieur dans la programmation orientée objet. C'est comme nous avons fini de la grammaire et commencé la composition.
- Les patrons de conception sont des solutions réutilisables aux problèmes de conception courants. Ils ne sont pas un code spécifique ou une partie de la langage.
- Je vais donner quelques exemples de modèles de conception courants.
- Les patrons de conception donnent des noms à des approches communes et permettent ainsi aux programmeurs de se parler à un niveau supérieur.

# Observateur

# Observateur patron de conception

Pourrait être de nombreux sujets et observateurs

Les sujets et les observateurs pourraient tous être des instances de différentes classes



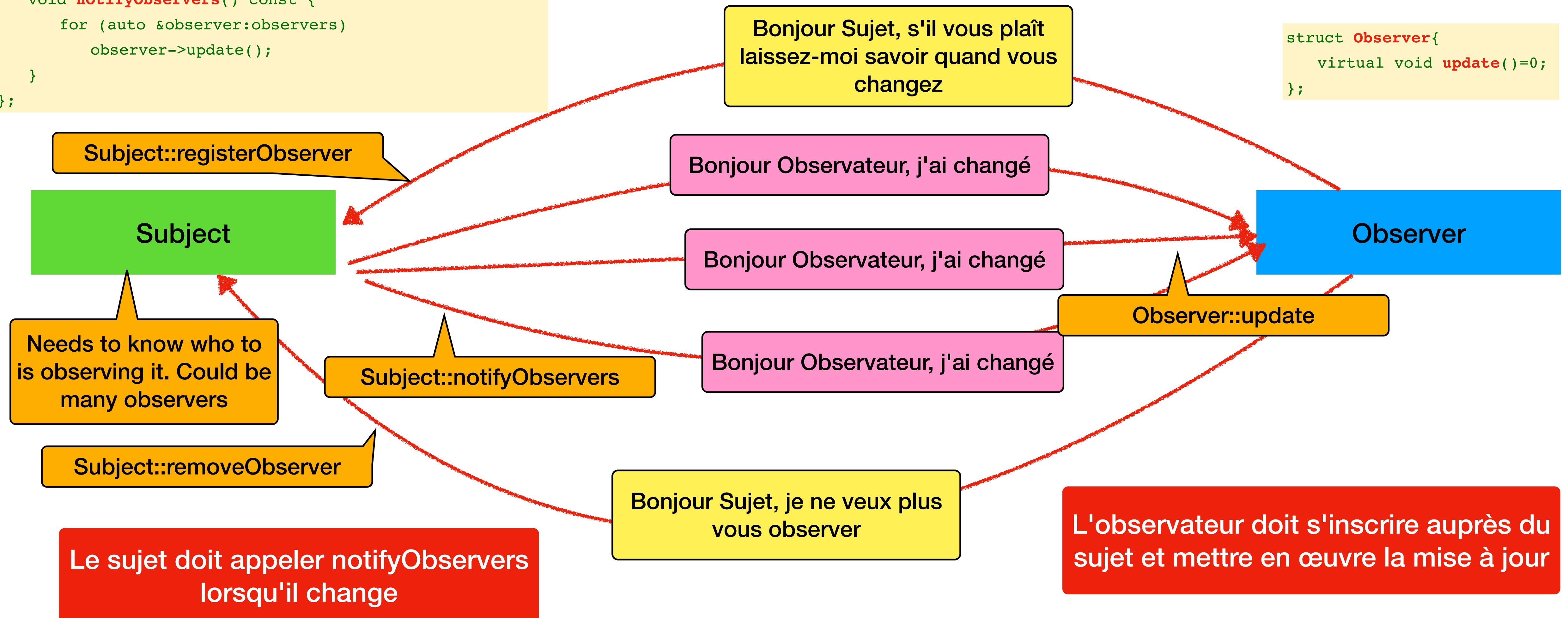
```

class Subject{
 vector <Observer *> observers;
public:
 void registerObserver(Observer *observer){
 observers.push_back(observer);
 }
 void removeObserver(Observer *observer){
 remove(begin(observers),end(observers),observer);
 }
 void notifyObservers() const {
 for (auto &observer:observers)
 observer->update();
 }
};

```

# Observateur patron de conception

## La communication devrait ressembler à ceci

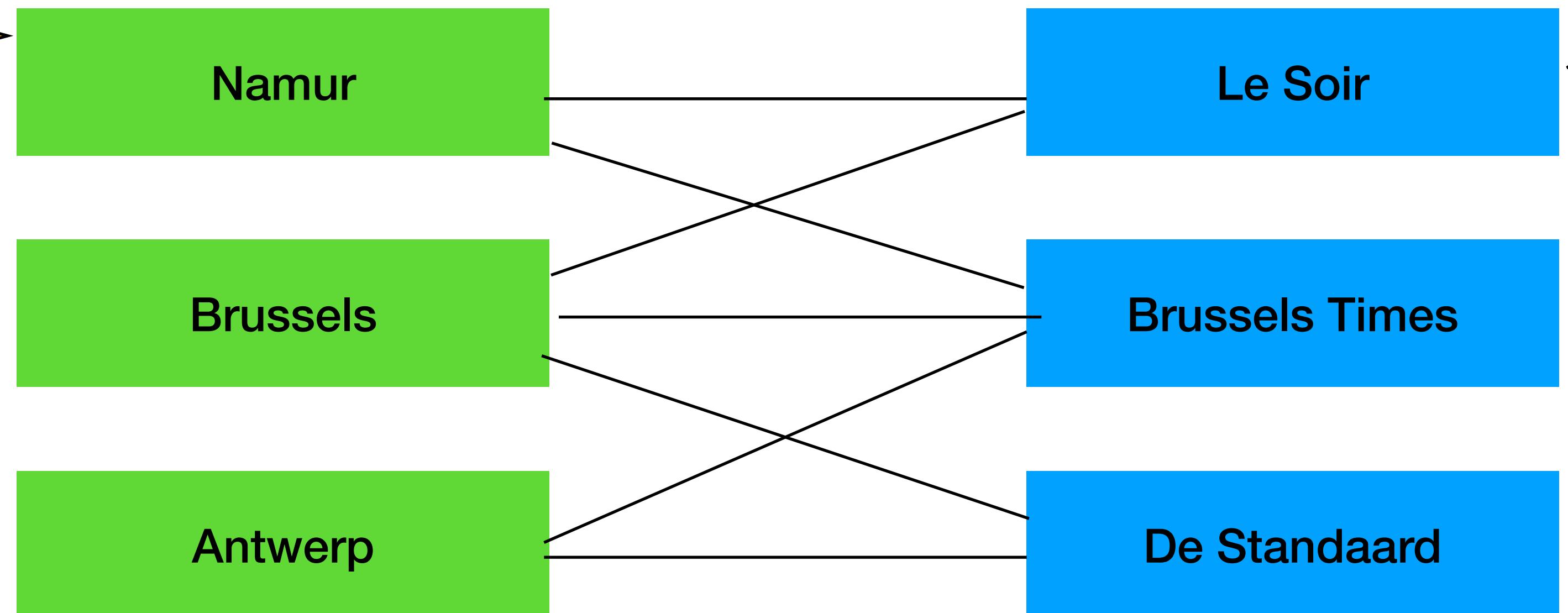


```

class WeatherStation{
 string location;
 float temperature;
public:
 WeatherStation(string location,float temperature)
 :location{location},temperature{temperature}{}
 float getTemperature() const {return temperature;}
 string getLocation() const {return location;}
 void setTemperature(float newTemperature) {
 temperature=newTemperature;
 }
};

```

Ce sont des stations météo



# Exemple : Météo

Chacun d'entre eux veut être averti lorsque la météo change dans les villes qui l'intéressent

Ce seront des instances de WeatherReport et nous voulons qu'elles impriment la météo lorsqu'elle change

```
WeatherStation
brusselsStation{"Brussels",0},
namurStation{"Namur",0},
antwerpStation{"Antwerp",0};
```

```
WeatherReport
belgiumReport{"Brussels Times", {&brusselsStation,&namurStation,&antwerpStation}},
leSoirReport{"Le Soir", {&brusselsStation,&namurStation}},
deStandaardReport{"De Standaard", {&brusselsStation,&antwerpStation}};
```

```
brusselsStation.setTemperature(20);
```

Weather report for Brussels Times  
Brussels:20  
Namur:0  
Antwerp:0  
Weather report for Le Soir  
Brussels:20  
Namur:0  
Weather report for De Standaard  
Brussels:20  
Antwerp:0

```
namurStation.setTemperature(15);
```

Weather report for Brussels Times  
Brussels:20  
Namur:15  
Antwerp:0  
Weather report for Le Soir  
Brussels:20  
Namur:15

```
namurStation.setTemperature(5);
```

Weather report for Brussels Times  
Brussels:20  
Namur:5  
Antwerp:0  
Weather report for Le Soir  
Brussels:20  
Namur:5

# Comment nous voulons que cela fonctionne

```

class Subject{
 vector<Observer*> observers;
public:
 void registerObserver(Observer *observer){
 observers.push_back(observer);
 }
 void removeObserver(Observer *observer){
 remove(begin(observers), end(observers), observer);
 }
 void notifyObservers() const {
 for (auto &observer:observers)
 observer->update();
 }
};

class WeatherStation:public Subject{
 string location;
 float temperature;
public:
 WeatherStation(string location, float temperature)
 :location{location},temperature{temperature}{}
 float getTemperature() const {return temperature;}
 string getLocation() const {return location;}
 void setTemperature(float newTemperature) {
 temperature=newTemperature;
 notifyObservers();
 }
};

```

Tout ce que la station doit faire est d'hériter du Subject et d'appeler notifyObservers quand il change

```

struct Observer{
 virtual void update()=0;
};

class WeatherReport:public Observer{
 string name;
 vector<WeatherStation*> stations;
public:
 WeatherReport(string name, vector<WeatherStation*> stations)
 :name{name}, stations{stations}{}
 for (auto station:stations)
 station->registerObserver(this);
 ~WeatherReport(){
 for (auto station:stations)
 station->removeObserver(this);
 }
 void update() override {
 cout<<"Weather report for "<<name<<endl;
 for (auto station:stations)
 cout<<" "<<station->getLocation()<<": "<<station->getTemperature()<<endl;
 }
};

```

```

WeatherStation
 brusselsStation{"Brussels",0},
 namurStation{"Namur",0},
 antwerpStation{"Antwerp",0};
WeatherReport
 belgiumReport{"Brussels Times", {&brusselsStation, &namurStation, &antwerpStation}},
 leSoirReport{"Le Soir", {&brusselsStation, &namurStation}},
 deStandaardReport{"De Standaard", {&brusselsStation, &antwerpStation}};
 cout<<"Updating Brussels"<<endl;
 brusselsStation.setTemperature(20);
 cout<<"Updating Namur"<<endl;
 namurStation.setTemperature(15);
 cout<<"Updating Antwerp"<<endl;
 namurStation.setTemperature(5);

```

# Exemple : Météo

Chaque rapport est initialisé avec un vecteur des stations qui l'intéresse et s'enregistre avec d'elles

Se retirer en tant qu'observateur avant la mort

La mise à jour est mise en œuvre pour obtenir les données météorologiques et imprimer un rapport

```

Updating Brussels
Weather report for Brussels Times
Brussels:20
Namur:0
Antwerp:0
Weather report for Le Soir
Brussels:20
Namur:0
Weather report for De Standaard
Brussels:20
Antwerp:0
Updating Namur
Weather report for Brussels Times
Brussels:20
Namur:15
Antwerp:0
Weather report for Le Soir
Brussels:20
Namur:15
Updating Antwerp
Weather report for Brussels Times
Brussels:20
Namur:5
Antwerp:0
Weather report for Le Soir
Brussels:20
Namur:5

```

# Sommaire : observateur patron de conception

- Le observateur patron de conception est utile lorsque vous avez des objets qui doivent être informés des changements dans d'autres objets
- Il est basé sur l'idée que les observateurs souscrivent aux sujets, et les sujets informent les observateurs lorsqu'ils changent
- Nous avons donné une implémentation simple.
  - Il a séparé le Subject/Observer de WeatherStation/WeatherReport
  - Il peut être utilisé avec plusieurs classes qui héritent de Subject et Observer. Certaines classes pourraient même hériter des deux !
- Des variations sur cette implémentation sont certainement possibles
  - pointeurs intelligents
  - Méthodes d'appel autres que update (fonction lambda)
  - Plusieurs types de update
  - Communiquer avec update l'identité du sujet qui effectuant l'update

# **Singleton**

**Programmation orientée objet**

**Prof. John Iacono**

# Singleton patron de conception

- Parfois, vous voulez au plus UNE instance d'une classe donnée.
- Exemple : une classe Log qui vous permet d'écrire des messages dans un fichier journal et de suivre le nombre de messages.
- Il n'y a qu'un seul fichier journal. Il ne doit être ouvert qu'une seule fois, et seulement si nécessaire.

# Classe de journal de base

```
class Log{
 int linesOutput=0;
 ofstream logfile;
public:
 Log():logfile{"22-singleton-log.txt"}{};
 void addMessage(const string &message) {
 logfile<<message<<endl;
 ++linesOutput;
 }
 int getLinesOutput() const {return linesOutput;}
};
```

# Singleton

```
class Log{
 static unique_ptr<Log> singleton;
 int linesOutput=0;
 ofstream logfile;
 Log():logfile{"22-singleton-log.txt"}{};
public:
 void addMessage(const string &message) {
 logfile<<message<<endl;
 ++linesOutput;
 }
 int getLinesOutput() const {return linesOutput;}
 static Log &get()
 {
 if (!singleton)
 singleton=unique_ptr<Log>(new Log());
 return *singleton;
 }
};

unique_ptr<Log> Log::singleton;
```

This stores a single STATIC pointer to a Log

The constructor is PRIVATE!  
Thus the user can not create a instance of Log

This gets returns the single static pointer to the Log, and constructs it if it has not been constructed yet

```
void f(){
 auto &log=Log::get();
 for (int i=1;i<4;i++){
 log.addMessage(to_string(i));
 }
}

int main(){
 Log::get().addMessage("Start");
 f();
 Log::get().addMessage("End");
 cout<<"Lines in log"<<Log::get().getLinesOutput()<<endl;
}
```

Output

Lines in log: 5

Start  
1  
2  
3  
End

23-singleton-log.txt

# Singleton : résumé

- Le modèle de conception singleton impose qu'il y ait au plus une instance d'une classe
- Une meilleure alternative à une variable globale.
  - L'instance ne sera construite que si elle est utilisée
  - L'accès au constructeur est privé
  - Toute la logique relative à la classe est à l'intérieur de la classe
- Couramment utilisé

# **Décorateur et Méthode Usine**

## **Programmation orientée objet**

**Prof. John Iacono**

# Patron de conception : Décorateur

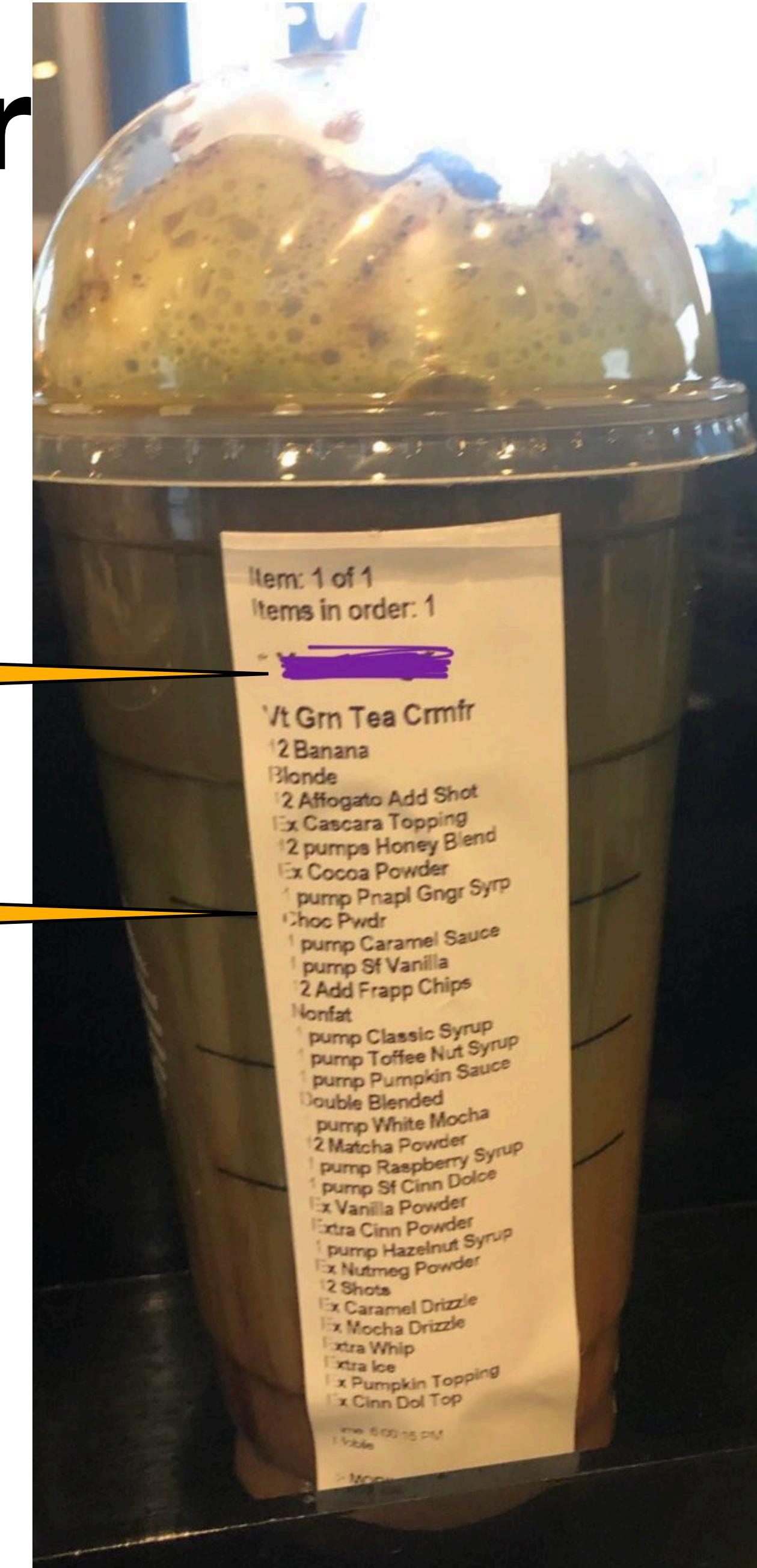
- Il y a une ou des classes auxquelles nous souhaitons ajouter des choses supplémentaires
- Ces choses sont appelées décos
- L'exemple du jour : le café



C'est un crabe décorateur

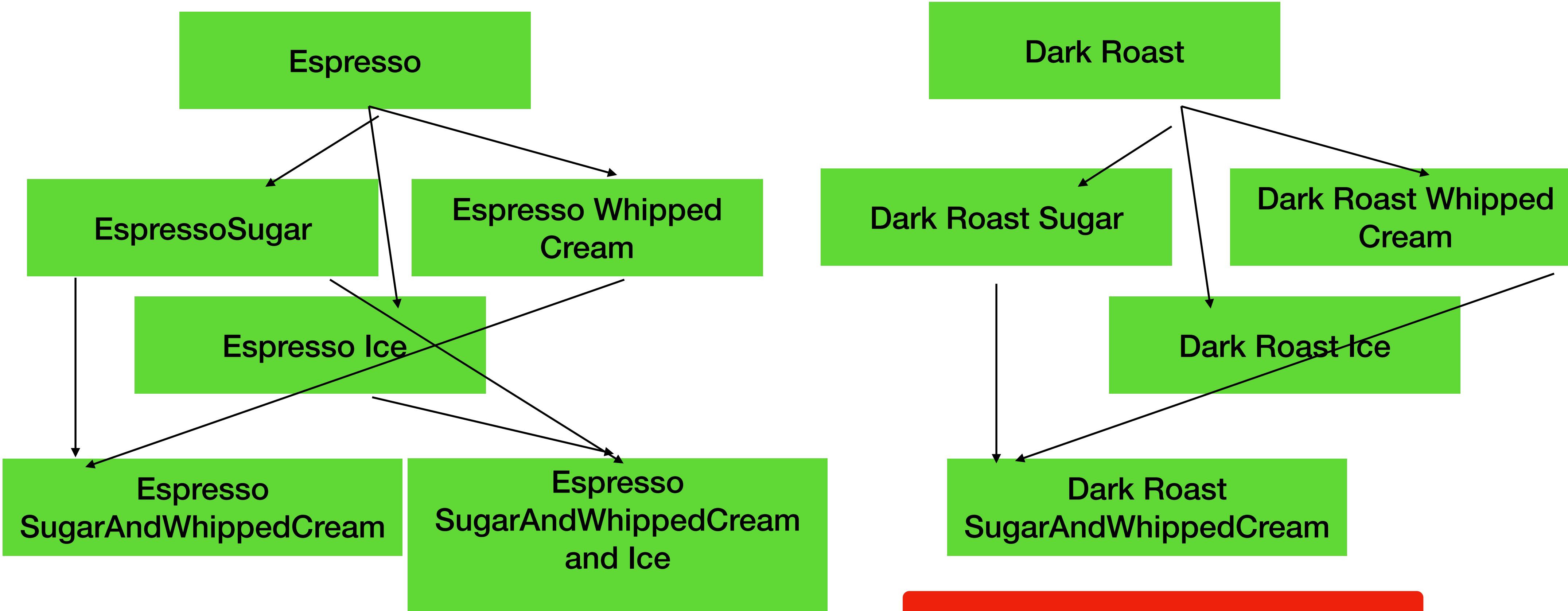
Base : Thé vert Matcha Frappuccino

Décos : extra glace, chocolat en poudre, sauce potiron, sirop de framboise, ....



# Mauvaise idée 1

Chaque boisson est sa propre classe



Besoin de  $2^N$  classes s'il y a N  
décorations

# Mauvaise idée 2

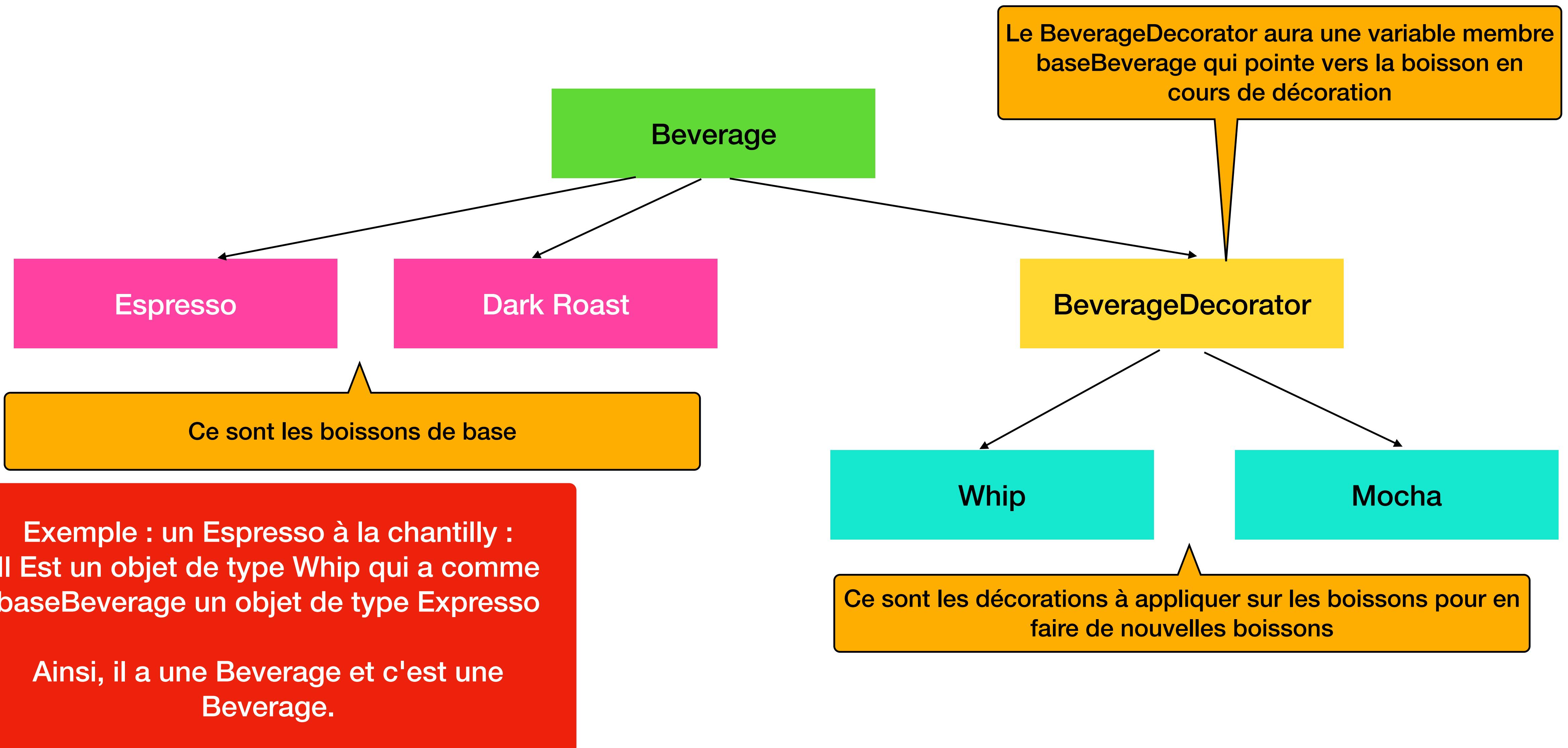
## Une classe

```
class Beverage{
 ...
public:
 void addSugar();
 void addCream();
 void addCocoa();
 void shakenNotStirred();
 void addPumpkinSpice();
 ...
};
```

Comment puis-je supprimer l'épice de citrouille?

# Décorateur

# Patron de conception : Décorateur



Toutes les boissons auront un coût et une description

Un Dark Roast de base coûte 1,50 €

```
struct Beverage{
 float virtual cost() const=0;
 string virtual description() const=0;
 virtual ~Beverage(){};
};
```

Tous les décorateurs hériteront de BeverageDecorator

Il existe une variable d'instance baseBeverage qui stocke un pointeur vers la boisson en cours de décoration

```
class DarkRoast:public Beverage{
 float cost() const override {return
1.5;}
 string description() const override {
 return "DarkRoast";}
};
```

```
class BeverageDecorator:public Beverage{
protected:
 const shared_ptr<Beverage> baseBeverage;
public:
 BeverageDecorator(shared_ptr<Beverage>
baseBeverage)
 :baseBeverage(baseBeverage){};
};
```

```
class Espresso:public Beverage{
 int shots;
public:
 Espresso(int shots=1):shots(shots){};
 float cost() const override {return 2+0.5*shots;}
 string description() const override {
 switch (shots) {
 case 1:return "Espresso";
 case 2:return "Double Espresso";
 case 3:return "Triple Espresso";
 default:return to_string(shots)+"-Espresso";
 }
 }
};
```

faire fonctionner l'impression

```
ostream &operator <<(ostream &os,const Beverage &b){
 return os<<b.description()<<" €"<<b.cost()<<endl;
}
```

```
class Whip:public BeverageDecorator{
using BeverageDecorator::BeverageDecorator;
 float cost() const override {
 return 0.5+baseBeverage->cost();}
 string description() const override {
 return baseBeverage->description()+"
+Whip";
 }
};
```

```
auto b1=make_shared<Espresso>(2);
cout<<*b1;
```

Double Espresso: €3

```
auto b2=make_shared<Whip>(b1);
cout<<*b2;
```

Double Espresso +Whip: €3.5

```
auto b3=make_shared<Whip>(make_shared<Mocha>(make_shared<DarkRoast>()));
cout<<*b3;
```

DarkRoast +Mocha +Whip: €3

Le fouet coûte 0,5 € en plus du coût de la boisson fouettée. Nous ajoutons le mot "whip" à la description

```
class Mocha:public BeverageDecorator{
using BeverageDecorator::BeverageDecorator;
 float cost() const override {
 return 1+baseBeverage->cost();}
 string description() const override {
 return baseBeverage->description()+"
+Mocha";
};
```

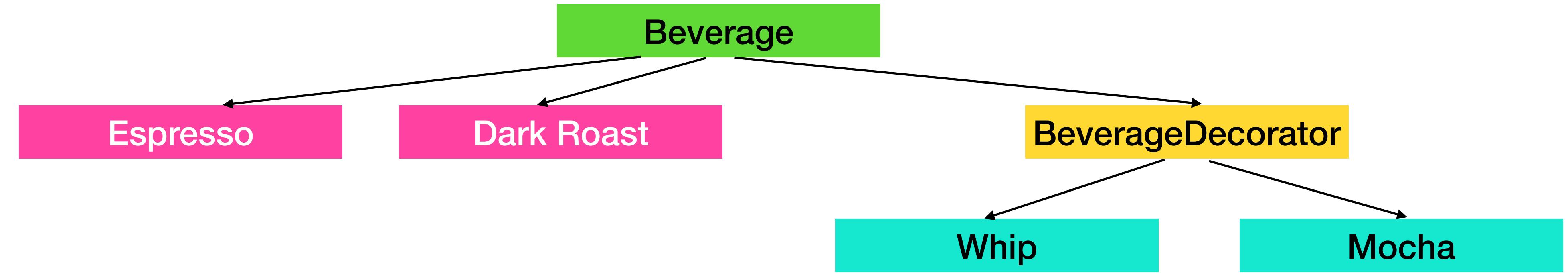
Un moka est une boisson, il peut donc être décoré aussi

# Décorateur : résumé

- Les décorateurs sont un moyen de donner des fonctionnalités supplémentaires aux classes d'une manière qui peut être combinée.
- Ils utilisent à la fois des variables d'héritage et d'instance : un moka A une boisson et EST une boisson
- Couramment utilisé, en particulier dans des situations telles que les interfaces graphiques.

# Méthode d'usine

# Peut-on améliorer la construction ?



```
auto b3=make_shared<Whip>(make_shared<Mocha>(make_shared<DarkRoast>()));
cout<<*b3;
```

C'est énervant

Ae serait mieux

```
auto b4=????({DarkRoast,Mocha,Whip})
```



- Qu'est-ce que doit ??? être?
- Il doit construire et renvoyer un pointeur vers la bonne boisson
- Il ne peut pas s'agir d'un constructeur, car un constructeur renvoie toujours une instance d'un type donné
- Cela ne devrait pas être une méthode, car une méthode nécessite qu'une instance existe déjà
- Solution : méthode statique (c'est-à-dire une fonction qui vit à l'intérieur d'une classe)

- Qu'est-ce que c'est?
- Une énumération

```
enum BeverageDescriptors {
 Espresso, DoubleEspresso, DarkRoast, Whip, Mocha
};
```

# Peut-on améliorer la construction ?

```
struct Beverage{
 float virtual cost() const=0;
 string virtual description() const=0;
 virtual ~Beverage(){};
 enum BeverageDescriptors {
 B_Espresso,B_DoubleEspresso,B_DarkRoast,D_Whip,D_Mocha
 };
 static shared_ptr<Beverage> makeBeverage(
 initializer_list<BeverageDescriptors> bd);
};
```

makeBeverage est notre méthode d'usine

Nous appelons makeBeverage de la classe Beverage

```
auto b4=Beverage::makeBeverage({Beverage::B_DoubleEspresso,Beverage::D_Mocha});
cout<<*b4;
```

```
shared_ptr<Beverage> Beverage::makeBeverage(initializer_list<BeverageDescriptors> bd){

 shared_ptr<Beverage> beverage;
 auto currentItem=begin(bd);

 switch (*currentItem){
 case B_Espresso: beverage=make_shared<Espresso>();break;
 case B_DoubleEspresso: beverage=make_shared<Espresso>(2);break;
 case B_DarkRoast: beverage=make_shared<DarkRoast>();break;
 default: throw "Beverage Base Wrong";
 }

 for (++currentItem; currentItem!=end(bd);++currentItem)
 switch (*currentItem){
 case D_Whip: beverage=make_shared<Whip>(beverage);break;
 case D_Mocha: beverage=make_shared<Mocha>(beverage);break;
 default: throw "Beverage Decorator Error";
 }

 return beverage;
}
```

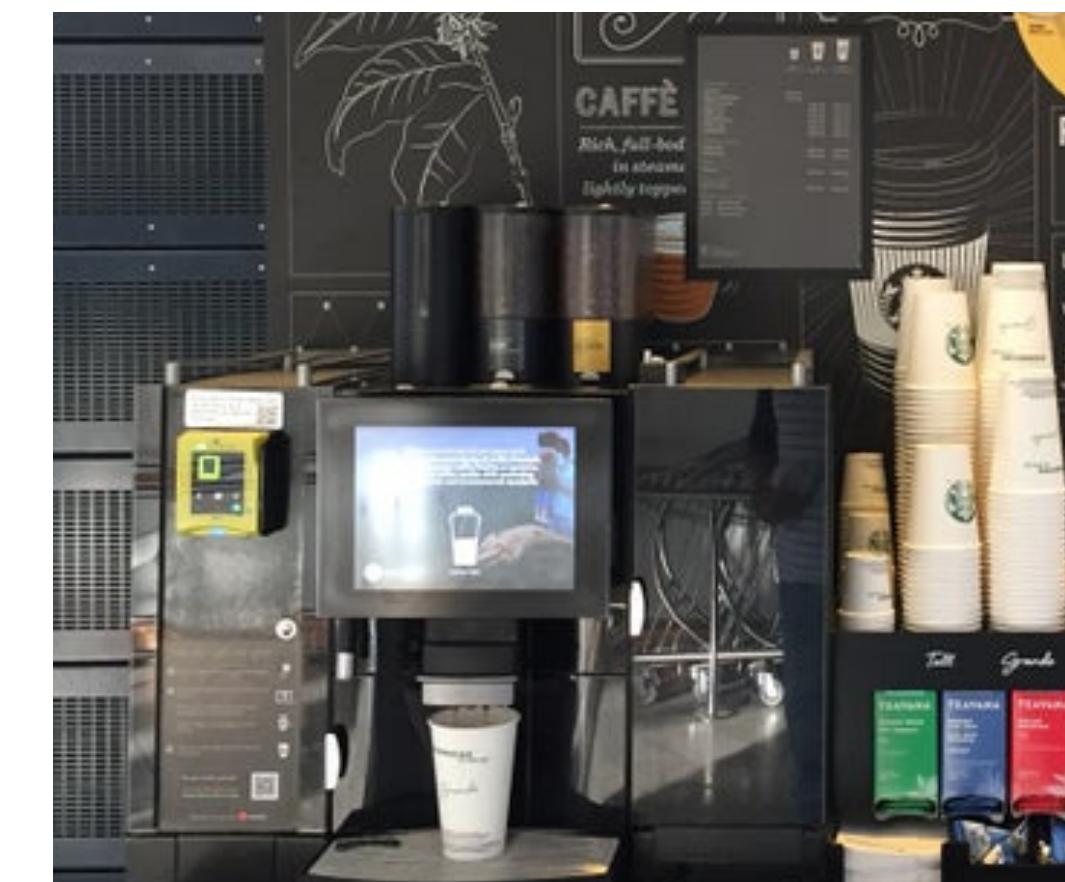
Nous construisons d'abord la boisson de base

On ajoute ensuite les décorateurs

Double Espresso +Mocha: €4

# Méthode usine : résumé

- Les constructeurs C++ ne peuvent construire que des méthodes d'une classe
- Les méthodes/fonctions peuvent renvoyer un pointeur de type pointeur de la classe de base, qui pourrait être de n'importe quel type qui descendant de la classe de base
- Une méthode usine est une méthode non constructeur, généralement statique, capable de construire et de renvoyer un pointeur/référence à une instance de plusieurs classes
- Les méthodes d'usine peuvent simplifier la construction
- Dans les cas plus complexes, le patron de conception Builder est utilisé. C'est quand vous avez une classe distincte ou une hiérarchie de classes qui est responsable pour créer des instances d'une autre classe
- Ex : BeverageBuilder



# **Modèle-vue-contrôleur (MVC)**

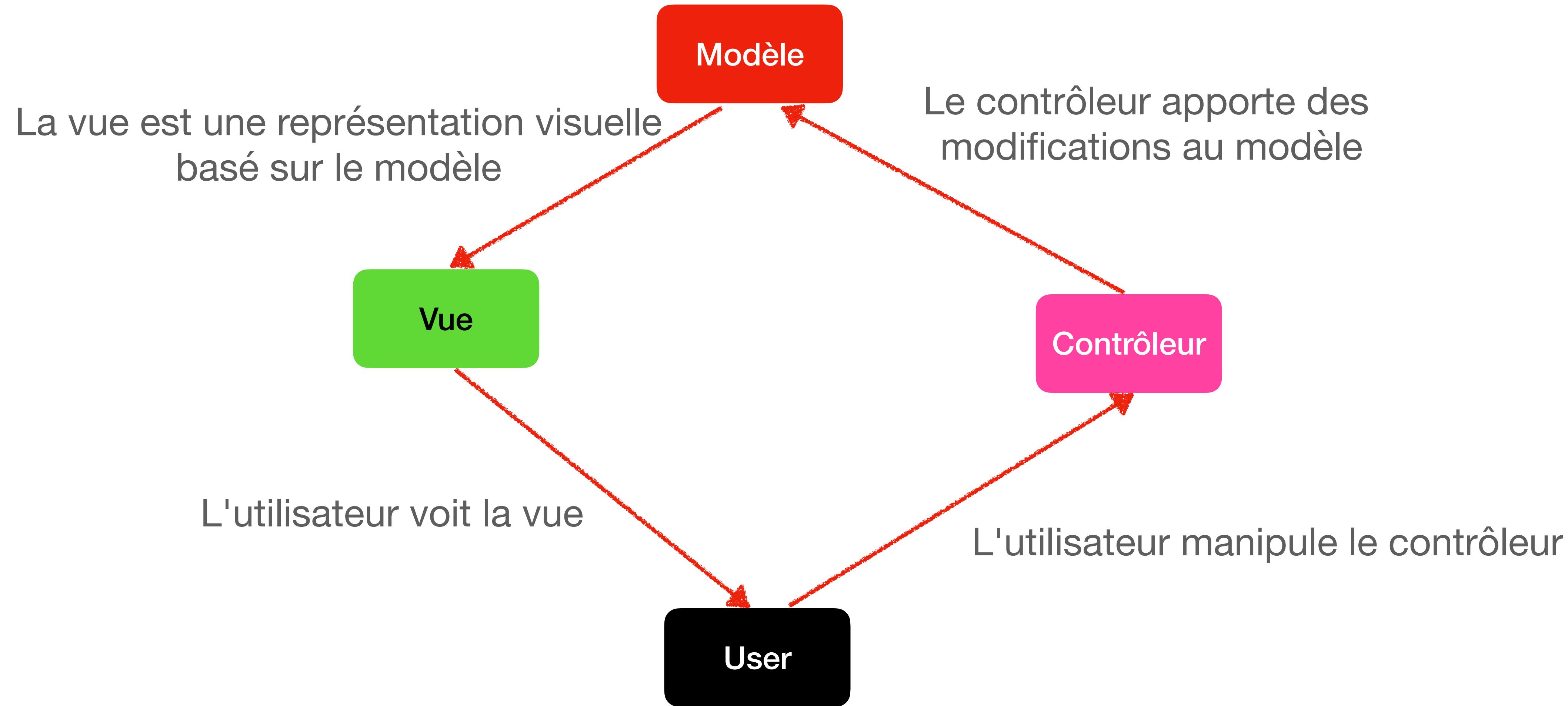
## **Programmation orientée objet**

**Prof. John Iacono**

# Modèle-vue-contrôleur (MVC)

Un modèle de conception pour les applications graphiques interactives

- Une bonne conception orientée objet sépare les différentes fonctionnalités
- Dans une application graphique, MVC se sépare en trois parties :
  - Modèle : les données, l'état et la logique sous-jacents.
  - Vue : Responsable d'afficher à l'utilisateur les données du modèle.
  - Contrôleur : traitez les entrées de l'utilisateur et apportez des modifications au modèle.



# **Exemple : démo**

# MVC

## Demo code

```
class Board{
 bitset<64> bits;
 int currentx=0,currenty=0;
public:
 int getCurrentx() {return currentx;}
 int getCurrenty() {return currenty;}
 bool getSquare(int x,int y){return bits[8*y+x];}
 bool getSquare(){return getSquare(getCurrentx(),getCurrenty());}
 void setSquare(bool value,int x,int y){bits[8*y+x]=value;}
 void setSquare(bool value){setSquare(value,getCurrentx(),getCurrenty());}
 void setCurrentx(int x) {currentx=(x+8)%8;}
 void setCurrenty(int y) {currenty=(y+8)%8;}
};
```

Modèle

```
class ControllBoard {
 shared_ptr<Board> board; Modèle
public:
 ControllBoard(shared_ptr<Board> board):board(board){}
 void processKey(char key){
 switch (key){
 case ' ':
 case '5': board->setSquare(!board->getSquare());break;
 case '4': board->setCurrentx(board->getCurrentx()-1);break;
 case '6': board->setCurrentx(board->getCurrentx()+1);break;
 case '8': board->setCurrenty(board->getCurrenty()-1);break;
 case '2': board->setCurrenty(board->getCurrenty()+1);break;
 case 'q': exit(0);
 }
 }
};
```

Contrôleur

```
class DisplayBoard {
 shared_ptr<Board> board; Modèle
public:
 DisplayBoard(shared_ptr<Board> board):board(board){}
 void display(){
 for (int y=0;y<8;++y){
 for (int x=0;x<8;x++){
 cout<<board->getSquare(x,y);
 cout<<((x==board->getCurrentx() && y==board->getCurrenty())?"< ":" ");
 }
 cout<<endl;
 }
 }
};
```

Vue

```
int main(){
 auto b=make_shared<Board>(); Modèle
 DisplayBoard db{b}; Vue
 ControllBoard controller{b}; Contrôleur
 while (true) {
 system("clear");
 db.display(); Vue
 char c;
 cin.get(c);
 controller.processKey(c); Contrôleur
 }
}
```

main

# Résumé : MVC

- MVC est un bon point de départ lors de la conception d'applications interactives avec une interface visuelle
- Il garde les données et la logique de base sur les données séparées de la visualisation et de la réponse à l'utilisateur
- Exemple : un jeu d'échecs, où l'ordinateur peut jouer
  - Le modèle aurait une représentation du plateau (c'est-à-dire un tableau 2D), des moyens de vérifier si un coup est valide et l'IA pour que l'ordinateur génère un coup
  - La vue afficherait le tableau
  - Le contrôleur donnerait des mouvements au modèle en fonction des entrées de l'utilisateur
- Le modèle est généralement très portable et n'aura pas besoin de changer
  - La vue et la manette seront très différentes pour un terminal jeu/versions Windows/X-windows/Mac/iOS/Android/Web. Ce ne sont pas des standards.
  - La personne qui fait la logique des échecs est probablement différente de la personne à laquelle on a donné la version iOS et à qui l'on a demandé de créer une version Android.

# **Autres langages : Java et Python**

**Programmation orientée objet**

**Prof. John Iacono**

# Autres langages orientés objet

- La plupart des langages populaires utilisés sont orientés objet
- Si vous connaissez bien le C++, il ne devrait pas être difficile de commencer à coder dans un autre langage après quelques heures d'apprentissage
- À titre d'exemple, je soulignerai les différences entre C++, Java et Python
- Rappelez-vous les 4 principaux langages de programmation actuellement utilisés :
  - #1 Python
  - #2 C
  - #3 Java
  - #4 C++

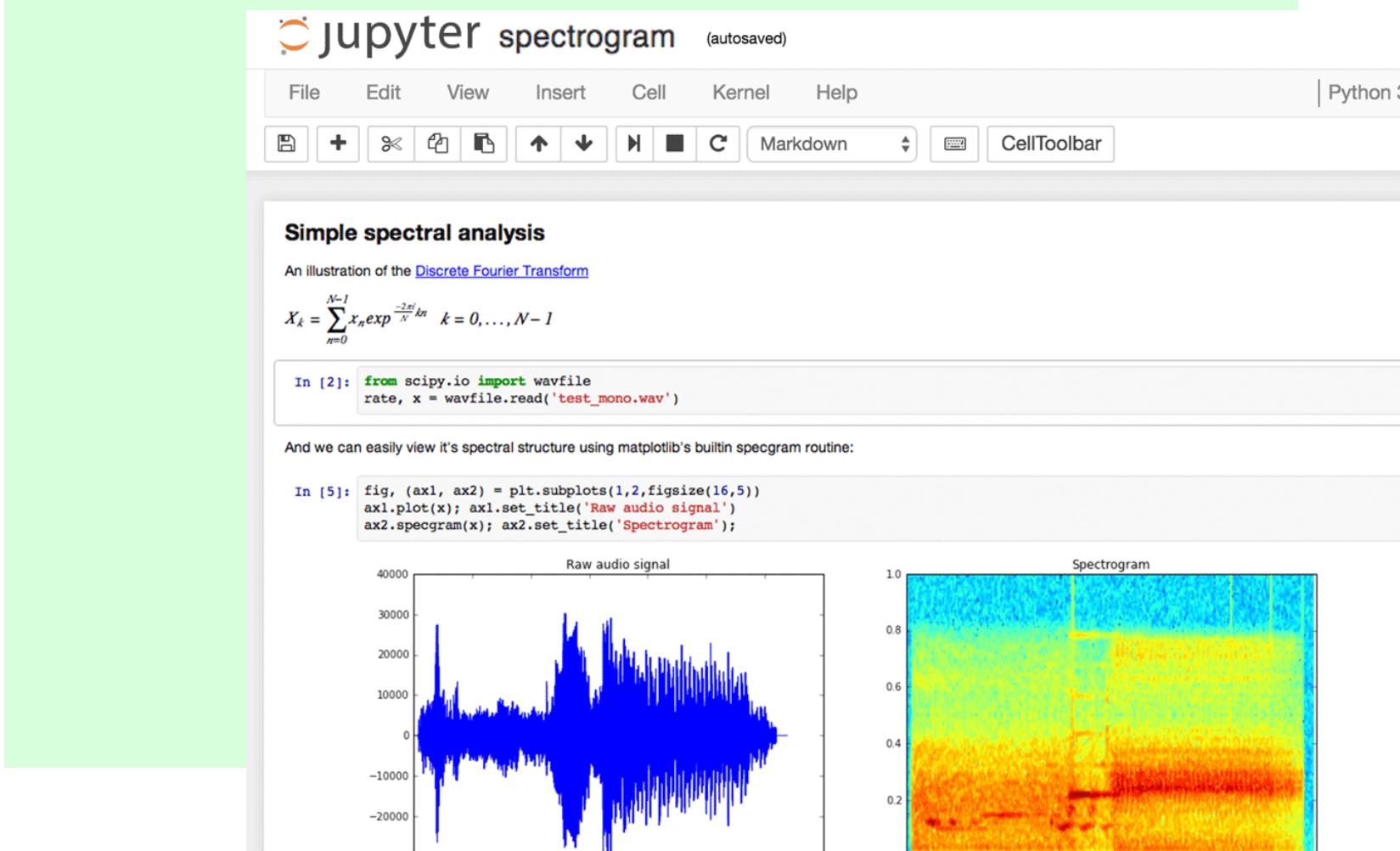
# Utilisation

## C++

- C++ est généralement compilé dans le langage machine d'un processeur spécifique.

## Python

- Python est généralement interprété
- L'utilisation interactive via Jupyter Notebook est populaire



## Java

- Java est généralement compilé en bytecode Java Virtual Machine (JVM) qui est une sorte de langage d'assemblage conçu pour être indépendant de la machine.

- Toute machine avec une implémentation JVM peut exécuter n'importe quel programme JVM

# Syntaxe

## C++

- Tu sais ça

C++ a une vingtaine de classes de conteneurs différentes dans la STL, ainsi que des tableaux normaux.

Python a choisi list/tuple/dictionary comme étant "spécial" et faisant partie du langage.

## Python

- Utilise l'indentation
- Syntaxe spéciale pour les listes, les tuples et les dictionnaires
- List: (comme `std::vector` en C++)  
`[1,2,3,4]`
- Tuple (comme `std::tuple` en C++)  
`(1,2,3,4)`
- Dictionary (comme `std::unordered_map` in C++)  
`D={"john":19,"stefan":12}  
print(D["john"])`

## Java

- Comme C++

# Variables

## C++

- `T f(a,b)`
- `T F=T(a,b)`
- `T *f=new T(a,b)`
- `T &f=f2`
- `shared_ptr<T> f =make_shared<T>(a,b)`
- `unique_ptr<T> f =make_unique<T>(a,b)`

variables régulières,  
pointeurs, références,  
pointeurs intelligents

## Python

- `f=T(a,b)`

Tout en Python se comporte  
comme un C++ shared\_ptr

Tout en Python est un objet

Dans la terminologie Python et Java, le  
mot "référence" signifie la même chose  
qu'un pointeur partagé en C++, PAS une  
référence en C++

## Java

- `T f=new T(a,b)`

Les objets en Java se comportent  
comme un C++ shared\_ptr

- `int f=7`

Les types intégrés se comportent  
comme des variables normales C++

# Typage

## C++

- Fortement typé
- `auto` pour les types que le compilateur peut déduire.

## Python

- Typé dynamiquement
- Les variables n'ont pas de type fixe
- Les fonctions sont comme n'importe quelle autre variable

```
def f(x,y):
 return x(y)

def plusone(x):
 return x+1

print(f(plusone,5))
```

## Java

- Fortement typé
- `var` pour les types que le compilateur peut déduire

# Définir des classes

## C++

- Tu sais ça
- `this` est une variable spéciale qui pointe vers l'instance actuelle.

## Python

- Les méthodes spéciales sont toutes `__quelquechose__`
- ```
class C :  
    def f(self,x)
```
- ```
x=C(...)
x.f(24)
```

`z` est `self`  
`24` est `x`

## Java

- Comme C++
- `this` comme en C++
- aussi `super()`

# Constructeurs et destructeurs

## C++

- Classname
- ~Classname

## Python

- `__init__`
- `__del__`

Appelé exactement comme dans un `shared_ptr` C++, lorsqu'il n'y a plus de références à l'objet

## Java

- Classname
- Pas de destructeur

# Surcharge des opérateurs

## C++

- Oui
- `T operator +(other)`

## Python

- Oui
- `def __add__(self,other)`

## Java

- Non

# Itérateurs

## C++

begin(x) appelle x.begin()

- `begin()` et `end()` obtiennent des objets itérateurs.
- `++` et `*` incrémenter et récupérer les données
- Riche hiérarchie d'itérateurs avec diverses fonctionnalités
- Boucle `for` basée sur la plage  
`for (auto &x:C)` utilise les iterators

## Python

iter(x) appelle x.\_\_iter\_\_()

- `__iter__` obtient l'itérateur
- `__next__` renvoie un objet et avance
- Boucle `for` basée sur la plage :  
`for x in Y`
- Générateurs  

```
def squares(L):
 for x in L:
 yield x*x
```
- Compréhension de liste  
`[x*x for x in range(10)]`

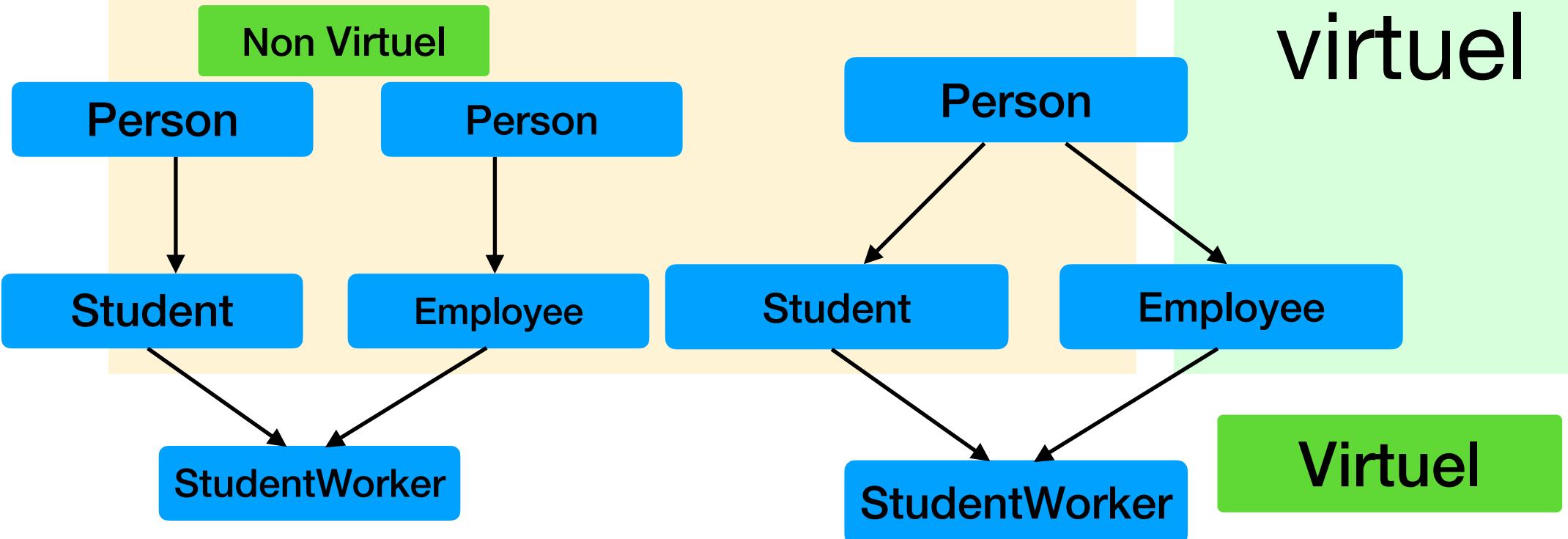
## Java

- `iterator()` obtient un itérateur
- L'itérateur a `hasNext()` et `next()` (et `remove()`)
- Boucle `for` basée sur la plage:  
`for (T x:Y)`

# Héritage multiple

## C++

- Oui
- Conflits non autorisés, s'il y a un conflit, vous devez le spécifier à l'aide de `classname::method`
- Virtuel vs non virtuel



## Python

- Oui
- Conflits résolus conformément à la « Method Resolution Order »
- Uniquement ce que C++ appelle l'héritage virtuel

## Java

- Non, en général
- Interfaces : classes abstraites (aucune méthode n'a d'implémentation, aucune donnée)
- Les classes peuvent hériter d'une autre classe réelle et « implémenter » plusieurs interfaces
- Pas de problème de diamant avec les classes abstraites

# Transtypage (Typecasing)

## C++

- Système complexe
- `dynamic_cast<NewType>(x)`
- `(NewType)x`

C-style Cast

Échec via un `nullptr` ou une exception

## Python

- Pas nécessaire mais peut se renseigner sur le type ou l'existence de méthodes particulières
- `hasattr()`, `isinstance()`, `vars()`, `__dict__`, ...

## Java

- `x instanceof NewType`
- `(NewType)x`

# Programmation générique

## C++

- Modèles

```
template <typename T>
class SomeClass
```

- Peut avoir des spécialisations, les règles complètes peuvent être complexes

## Python

- Not needed

## Java

- Génériques

```
class public SomeClass<T>
```

- Ne fonctionne pas avec les types intégrés

# Functions comme objets

C++

Surchargez operator ()

- Functors
- Expressions lambda
- Pointeurs de fonction

Python

- Ils fonctionnent juste sans avoir à faire quoi que ce soit

Java

- Expressions lambda
- Interfaces fonctionnelles

L'idée est d'avoir une interface (classe abstraite) avec une seule méthode qui représente la fonction, et vous créez une classe qui hérite de cette interface et implémente la fonction.

`java.util.function` possède un certain nombre de ces interfaces, qui diffèrent en fonction des paramètres et de la valeur de retour de la fonction

# Vitesse

## C++

- Très vite
- Chaque objet est aussi petit que possible
- Compromis exacts possibles entre vitesse/ espace (méthodes virtuelles vs non virtuelles et héritage)
- Des tableaux de tout type d'objet sont possibles

## Python

- Ralentir
- Chaque objet est énorme
- Vous ne pouvez pas avoir de tableaux d'objets, ce sont des tableaux de pointeurs (références en javaspeak) vers des objets.
- De nombreuses bibliothèques courantes comme NumPy sont rapides

## Java

- Milieu
- Vous ne pouvez pas avoir de tableaux d'objets, ce sont des tableaux de pointeurs (références en javaspeak) vers des objets.

NumPy est écrit en C/C++ !

# Programmation système et bas niveau

## C++

- Possible
- Les pointeurs bruts peuvent pointer vers n'importe quelle adresse mémoire
- Accès aux opcodes/GPU/etc de bas niveau de la machine

## Python

- Impossible

## Java

- Impossible

# Summary

- C++ est très riche
- Java est plus simple et Python encore plus
- Le choix d'un langage de programmation se fait souvent non seulement en fonction du langage lui-même mais aussi des bibliothèques.
- Par exemple:

Pour la programmation sur iOS j'utiliserais Swift

Pour le développement d'applications multi-plateformes avec la bibliothèque Flutter, j'utiliserais le langage Dart

Pour écrire une interface Web à l'aide de node.js, il faudrait utiliser Javascript

Pour faire du calcul scientifique interactif en utilisant NumPy et Pandas, on utiliserait Python

- Ne vous attendez pas à une explication détaillée d'autres langues dans les prochains cours. Vous avez maintenant les compétences nécessaires pour apprendre par vous-même. Vous devriez être capable, par exemple, d'apprendre les bases de Dart et de commencer à coder une application en une journée.
- C++ a été choisi car il est facile de passer du C++ à la plupart des autres langages orientés objet, et il a résisté à l'épreuve du temps.