

Programmation C++

2^{ème} année Informatique

J. Allali

julien.allali@labri.fr

IPB/ENSEIRB-MATMECA

Prog. C++

Plan

1 Historique

Plan

1 Historique

2 Allocation

Plan

1 Historique

2 Allocation

3 Classes

Plan

- 1 Historique
- 2 Allocation
- 3 Classes
- 4 Operateurs

Plan

- 1 Historique
- 2 Allocation
- 3 Classes
- 4 Operateurs
- 5 Héritage, polymorphisme

Plan

- 1 Historique
- 2 Allocation
- 3 Classes
- 4 Operateurs
- 5 Héritage, polymorphisme
- 6 Exceptions

Plan

- 1 Historique
- 2 Allocation
- 3 Classes
- 4 Operateurs
- 5 Héritage, polymorphisme
- 6 Exceptions
- 7 Template

Plan

- 1 Historique
- 2 Allocation
- 3 Classes
- 4 Operateurs
- 5 Héritage, polymorphisme
- 6 Exceptions
- 7 Template

Plan

1 Historique

Historique (wikipedia)

- 198x: Bjarne Stroustrup (AT&T Bell): “C with classes”:
 - Classes
 - fonctions virtuelles
 - surcharge d’opérateurs
 - héritage multiple
 - ...

Historique (wikipedia)

- 198x: Bjarne Stroustrup (AT&T Bell): “C with classes”:
 - Classes
 - fonctions virtuelles
 - surcharge d’opérateurs
 - héritage multiple
 - ...

Normes:

- 1998: Normalisation du C++ par l’ISO (International Organization for Standardization) et ANSI ISO/CEI 14882:1998
- 2003: Dernière version 14882:2003
- 2011: Dernière version 14882:2011

C (++)

Le C++ est un language impératif, *orienté objets*:

C (++)

Le C++ est un language impératif, *orienté objets*:

L'ajout de fonctionnalités permettant la mise en oeuvre de concepts objets dans le language C:

- l'objet: attributs(données internes) + méthodes (comportements), encapsulation.
- Typage des objets.
- Polymorphisme: un objet peut avoir plus d'un type.
- redéfinition.
- classe: description et génération des objets.

C (++)

Le C++ est un language impératif, *orienté objets*:

L'ajout de fonctionnalités permettant la mise en oeuvre de concepts objets dans le language C:

- l'objet: attributs(données internes) + méthodes (comportements), encapsulation.
- Typage des objets.
- Polymorphisme: un objet peut avoir plus d'un type.
- redéfinition.
- classe: description et génération des objets.

Il permet aussi la *programmation générique*: écriture de fonctions (et d'objets) indépendantes du type de ces arguments. C'est l'idée du “*code à trous*”.

Incompatibilité entre le C et le C++

Source C qui ne compile (invalidité syntaxique) pas en C++:

- Si le source contient un des nouveaux mots clés du C++
- Le C autorise la conversion implicite de void * en n'importe quel autre type de pointeur:

exemple

```
int *i=malloc(4);
```

IEEE Bjarne Stroustrup C++ Interview

On the 1st of January, 1998, Bjarne Stroustrup gave an interview to the IEEE's 'Computer' magazine. Naturally, the editors thought he would be giving a retrospective view of seven years of object-oriented design, using the language he created (C++).

By the end of the interview, the interviewer got more than he had bargained for and, subsequently, the editor decided to suppress its contents, 'for the good of the industry' but, as with many of these things, there was a leak.

Here is a complete transcript of what was said, unedited, and unrehearsed, so it isn't as neat as planned interviews. You will find it interesting...

Interviewer: Well, it's been a few years since you changed the world of software design, how does it feel, looking back?

Stroustrup: Actually, I was thinking about those days, just before you arrived. Do you remember? Everyone was writing 'C' and, the trouble was, they were pretty damn good at it. Universities got pretty good at teaching it, too. They were turning out competent - I stress the word 'competent' - graduates at a phenomenal rate. That's what caused the problem.

Interviewer: Problem?

Stroustrup: Yes, problem. Remember when everyone wrote Cobol?

Interviewer: Of course, I did too

Stroustrup: Well, in the beginning, these guys were like demi-gods. Their salaries were high, and they were treated like royalty.

Interviewer: Those were the days, eh?

Stroustrup: Right. So what happened? IBM got sick of it, and invested millions in training programmers, till they were a dime a dozen.

Interviewer: That's why I got out. Salaries dropped within a year, to the point where being a journalist actually paid better.

Stroustrup: Exactly. Well, the same happened with 'C' programmers.

Interviewer: I see, but what's the point?

Stroustrup: Well, one day, when I was sitting in my office, I thought of this little scheme, which would redress the balance a little. I thought 'I wonder what would happen, if there were a language so complicated, so difficult to learn, that nobody would ever be able to swamp the market with programmers? Actually, I got some of the ideas from X10, you know, X windows. That was such a bitch of a graphics system, that it only just ran on those Sun 3/60 things. They had all the ingredients for what I wanted. A really ridiculously complex syntax, obscure functions, and pseudo-OO structure. Even now, nobody writes raw X-windows code. Motif is the only way to go if you want to retain your sanity.

Interviewer: You're kidding...?

Stroustrup: Not a bit of it. In fact, there was another problem. Unix was written in 'C', which meant that any 'C' programmer could very easily become a systems programmer. Remember what a mainframe systems programmer used to earn?

Interviewer: You bet I do, that's what I used to do.

Stroustrup: OK, so this new language had to divorce itself from Unix, by hiding all the system calls that bound the two together so nicely. This would enable guys who only knew about DOS to earn a decent living too.

Interviewer: I don't believe you said that...

Stroustrup: Well, it's been long enough, now, and I believe most people have figured out for themselves that C++ is a waste of time but, I must say, it's taken them a lot longer than I thought it would.

Interviewer: So how exactly did you do it?

Stroustrup: It was only supposed to be a joke, I never thought people would take the book seriously. Anyone with half a brain can see that object-oriented programming is counter-intuitive, illogical and inefficient.

Interviewer: What?

Stroustrup: And as for 're-useable code' - when did you ever hear of a company re-using its code?

Interviewer: Well, never, actually, but...

Stroustrup: There you are then. Mind you, a few tried, in the early days. There was this Oregon company - Mentor Graphics, I think they were called - really caught a cold trying to rewrite everything in C++ in about '90 or '91. I felt sorry for them really, but I thought people would learn from their mistakes.

Interviewer: Obviously, they didn't?

Stroustrup: Not in the slightest. Trouble is, most companies hush-up all their major blunders, and explaining a \$30 million loss to the shareholders would have been difficult. Give them their due, though, they made it work in the end.

Interviewer: They did? Well, there you are then, it proves O-O works.

Stroustrup: Well, almost. The executable was so huge, it took five minutes to load, on an HP workstation, with 128MB of RAM. Then it ran like treacle. Actually, I thought this would be a major stumbling-block, and I'd get found out within a week, but nobody cared. Sun and HP were only too glad to sell enormously powerful boxes, with huge resources just to run trivial programs. You know, when we had our first C++ compiler, at AT&T, I compiled 'Hello World', and couldn't believe the size of the executable. 2.1MB

Interviewer: What? Well, compilers have come a long way, since then.
Stroustrup: They have? Try it on the latest version of g++ - you won't get much change out of half a megabyte. Also, there are several quite recent examples for you, from all over the world. British Telecom had a major disaster on their hands but, luckily, managed to scrap the whole thing and start again. They were luckier than Australian Telecom. Now I hear that Siemens is building a dinosaur, and getting more and more worried as the size of the hardware gets bigger, to accommodate the executables. Isn't multiple inheritance a joy?

Interviewer: Yes, but C++ is basically a sound language.

Stroustrup: You really believe that, don't you? Have you ever sat down and worked on a C++ project? Here's what happens: First, I've put in enough pitfalls to make sure that only the most trivial projects will work first time. Take operator overloading. At the end of the project, almost every module has it, usually, because guys feel they really should do it, as it was in their training course. The same operator then means something totally different in every module. Try pulling that lot together, when you have a hundred or so modules. And as for data hiding. God, I sometimes can't help laughing when I hear about the problems companies have making their modules talk to each other. I think the word 'synergistic' was specially invented to twist the knife in a project manager's ribs.

Interviewer: I have to say, I'm beginning to be quite appalled at all this. You say you did it to raise programmers' salaries? That's obscene.

Stroustrup: Not really. Everyone has a choice. I didn't expect the thing to get so much out of hand. Anyway, I basically succeeded. C++ is dying off now, but programmers still get high salaries - especially those poor devils who have to maintain all this crap. You do realise, it's impossible to maintain a large C++ software module if you didn't actually write it?

Interviewer: How come?

Stroustrup: You are out of touch, aren't you? Remember the `typedef`?

Interviewer: Yes, of course.

Stroustrup: Remember how long it took to grope through the header files only to find that 'RoofRaised' was a double precision number? Well, imagine how long it takes to find all the implicit `typedefs` in all the Classes in a major project.

Interviewer: So how do you reckon you've succeeded?

Stroustrup: Remember the length of the average-sized 'C' project? About 6 months. Not nearly long enough for a guy with a wife and kids to earn enough to have a decent standard of living. Take the same project, design it in C++ and what do you get? I'll tell you. One to two years. Isn't that great? All that job security, just through one mistake of judgement. And another thing. The universities haven't been teaching 'C' for such a long time, there's now a shortage of decent 'C' programmers. Especially those who know anything about Unix systems programming. How many guys would know what to do with 'malloc', when they've used 'new' all these years - and never bothered to check the return code. In fact, most C++ programmers throw away their return codes. Whatever happened to good ol' '-1'? At least you knew you had an error, without bogging the thing down in all that 'throw' 'catch' 'try' stuff.

Interviewer: But, surely, inheritance does save a lot of time?

Stroustrup: Does it? Have you ever noticed the difference between a 'C' project plan, and a C++ project plan? The planning stage for a C++ project is three times as long. Precisely to make sure that everything which should be inherited is, and what shouldn't isn't. Then, they still get it wrong. Whoever heard of memory leaks in a 'C' program? Now finding them is a major industry. Most companies give up, and send the product out, knowing it leaks like a sieve, simply to avoid the expense of tracking them all down.

Interviewer: There are tools...

Stroustrup: Most of which were written in C++.

Interviewer: If we publish this, you'll probably get lynched, you do realise that?

Stroustrup: I doubt it. As I said, C++ is way past its peak now, and no company in its right mind would start a C++ project without a pilot trial. That should convince them that it's the road to disaster. If not, they deserve all they get. You know, I tried to convince Dennis Ritchie to rewrite Unix in C++.

Interviewer: Oh my God. What did he say?

Stroustrup: Well, luckily, he has a good sense of humor. I think both he and Brian figured out what I was doing, in the early days, but never let on. He said he'd help me write a C++ version of DOS, if I was interested.

Interviewer: Were you?

Stroustrup: Actually, I did write DOS in C++, I'll give you a demo when we're through. I have it running on a Sparc 20 in the computer room. Goes like a rocket on 4 CPU's, and only takes up 70 megs of disk.

Interviewer: What's it like on a PC?

Stroustrup: Now you're kidding. Haven't you ever seen Windows '95? I think of that as my biggest success. Nearly blew the game before I was ready, though.

Interviewer: You know, that idea of a Unix++ has really got me thinking. Somewhere out there, there's a guy going to try it.

Stroustrup: Not after they read this interview.

Interviewer: I'm sorry, but I don't see us being able to publish any of this.

Stroustrup: But it's the story of the century. I only want to be remembered by my fellow programmers, for what I've done for them. You know how much a C++ guy can get these days?

Interviewer: Last I heard, a really top guy is worth \$70 - \$80 an hour.

Stroustrup: See? And I bet he earns it. Keeping track of all the gotchas I put into C++ is no easy job. And, as I said before, every C++ programmer feels bound by some mystic promise to use every damn element of the language on every project. Actually, that really annoys me sometimes, even though it serves my original purpose. I almost like the language after all this time.

Interviewer: You mean you didn't before?

Stroustrup: Hated it. It even looks clumsy, don't you agree? But when the book royalties started to come in... well, you get the picture.

Interviewer: Just a minute. What about references? You must admit, you improved on 'C' pointers.

Stroustrup: Hmm. I've always wondered about that. Originally, I thought I had. Then, one day I was discussing this with a guy who'd written C++ from the beginning. He said he could never remember whether his variables were referenced or dereferenced, so he always used pointers. He said the little asterisk always reminded him.

Interviewer: Well, at this point, I usually say 'thank you very much' but it hardly seems adequate.

Stroustrup: Promise me you'll publish this. My conscience is getting the better of me these days.

Interviewer: I'll let you know, but I think I know what my editor will say.

Stroustrup: Who'd believe it anyway? Although, can you send me a copy of that tape?

Interviewer: I can do that.

Plan

② Allocation

Allocation automatique

L'allocation automatique se fait dans la pile.

Allocation automatique

L'allocation automatique se fait dans la pile.

Exemple

```
int i;
```

La variable i est allouée dans la pile automatiquement.

Allocation automatique

L'allocation automatique se fait dans la pile.

Exemple

```
int i;
```

La variable i est allouée dans la pile automatiquement.

Ainsi, &i correspond à une adresse dans la pile à laquelle sizeof (int) octets sont réservés.

Allocation automatique

L'allocation automatique se fait dans la pile.

Exemple

```
int i;
```

La variable i est allouée dans la pile automatiquement.

Ainsi, &i correspond à une adresse dans la pile à laquelle sizeof (int) octets sont réservés.

À la sortie du bloc dans lequel est déclarée i, il y a dépilage et donc l'adresse &i correspond à une zone mémoire qui n'est plus réservée.

Allocation dynamique

L'allocation dynamique en C++ se fait à l'aide de l'opérateur **new**.

On distingue deux types d'allocation dynamique: l'allocation d'un objet ou l'allocation d'un tableau d'objets.

Exemple

new int

new int [10]

Dans le premier cas, on réserve sizeof(int) octets dans le tas, l'opérateur **new** retourne l'adresse de début de cette zone.

Dans le deuxième cas, on réserve sizeof(int)*10 octets dans le tas, l'opérateur **new** retourne l'adresse de début de cette zone.

Ces zones mémoires sont réservées tant que l'on n'a pas indiqué explicitement que l'on souhaitait les libérer avec l'opérateur **delete**.

Allocation dynamique

L'allocation dynamique en C++ se fait à l'aide de l'opérateur **new**.

On distingue deux types d'allocation dynamique: l'allocation d'un objet ou l'allocation d'un tableau d'objets.

Exemple

new int

new int [10]

Dans le premier cas, on réserve sizeof(int) octets dans le tas, l'opérateur **new** retourne l'adresse de début de cette zone.

Dans le deuxième cas, on réserve sizeof(int)*10 octets dans le tas, l'opérateur **new** retourne l'adresse de début de cette zone.

Ces zones mémoires sont réservées tant que l'on n'a pas indiqué explicitement que l'on souhaitait les libérer avec l'opérateur **delete**.

Allocation dynamique

L'allocation dynamique en C++ se fait à l'aide de l'opérateur **new**.

On distingue deux types d'allocation dynamique: l'allocation d'un objet ou l'allocation d'un tableau d'objets.

Exemple

new int

new int [10]

Dans le premier cas, on réserve sizeof(int) octets dans le tas, l'opérateur **new** retourne l'adresse de début de cette zone.

Dans le deuxième cas, on réserve sizeof(int)*10 octets dans le tas, l'opérateur **new** retourne l'adresse de début de cette zone.

Ces zones mémoires sont réservées tant que l'on n'a pas indiqué explicitement que l'on souhaitait les libérer avec l'opérateur **delete**.

Allocation dynamique

L'allocation dynamique en C++ se fait à l'aide de l'opérateur **new**.

On distingue deux types d'allocation dynamique: l'allocation d'un objet ou l'allocation d'un tableau d'objets.

Exemple

new int

new int [10]

Dans le premier cas, on réserve sizeof(int) octets dans le tas, l'opérateur **new** retourne l'adresse de début de cette zone.

Dans le deuxième cas, on réserve sizeof(int)*10 octets dans le tas, l'opérateur **new** retourne l'adresse de début de cette zone.

Ces zones mémoires sont réservées tant que l'on n'a pas indiqué explicitement que l'on souhaitait les libérer avec l'opérateur **delete**.

Allocation dynamique

L'allocation dynamique en C++ se fait à l'aide de l'opérateur **new**.

On distingue deux types d'allocation dynamique: l'allocation d'un objet ou l'allocation d'un tableau d'objets.

Exemple

new int

new int [10]

Dans le premier cas, on réserve sizeof(int) octets dans le tas, l'opérateur **new** retourne l'adresse de début de cette zone.

Dans le deuxième cas, on réserve sizeof(int)*10 octets dans le tas, l'opérateur **new** retourne l'adresse de début de cette zone.

Ces zones mémoires sont réservées tant que l'on n'a pas indiqué explicitement que l'on souhaitait les libérer avec l'opérateur **delete**.

Allocation dynamique: libération

Il faut libérer la mémoire allouée avec **new** en utilisant **delete**.

Il faut libérer la mémoire allouée avec **new type[]** en utilisant **delete[]**.

exemple

```
delete (new int);  
delete [] (new int [10]);
```

Une fois l'opérateur **delete** appelé, la mémoire qui était réservée à cette adresse ne l'est plus.

Allocation dynamique: libération

Il faut libérer la mémoire allouée avec **new** en utilisant **delete**.

Il faut libérer la mémoire allouée avec **new type[]** en utilisant **delete[]**.

exemple

```
delete (new int);  
delete [] (new int [10]);
```

Une fois l'opérateur **delete** appelé, la mémoire qui était réservée à cette adresse ne l'est plus.

Allocation dynamique: libération

Il faut libérer la mémoire allouée avec **new** en utilisant **delete**.

Il faut libérer la mémoire allouée avec **new type[]** en utilisant **delete[]**.

exemple

```
delete (new int);  
delete [] (new int [10]);
```

Une fois l'opérateur **delete** appelé, la mémoire qui était réservée à cette adresse ne l'est plus.

Allocation dynamique: libération

Il faut libérer la mémoire allouée avec **new** en utilisant **delete**.

Il faut libérer la mémoire allouée avec **new type[]** en utilisant **delete[]**.

exemple

```
delete (new int);  
delete [] (new int [10]);
```

Une fois l'opérateur **delete** appelé, la mémoire qui était réservée à cette adresse ne l'est plus.

Allocation: exemple complet

exemple complet

```
int *i = new int;  
int *t = new int[10];  
delete i;  
delete [] t;
```

Allocation: exemple complet

exemple complet

```
int *i = new int;  
int *t = new int[10];  
delete i;  
delete [] t;
```

Question: combien y-a-t-il d'allocations effectuées dans cet exemple?

Allocation: exemple complet

exemple complet

```
int *i = new int;  
int *t = new int[10];  
delete i;  
delete [] t;
```

Question: combien y-a-t-il d'allocations effectuées dans cet exemple?

Réponse: 4, 2 automatiques et 2 dynamiques.

Allocation: exemple complet

exemple complet

```
int *i = new int;  
int *t = new int[10];  
delete i;  
delete [] t;
```

Question: combien y-a-t-il d'allocations effectuées dans cet exemple?

Réponse: 4, 2 automatiques et 2 dynamiques.

En effet, il y a deux allocations automatiques de sizeof(int *) pour les variables i et t

Allocation: exemple complet

exemple complet

```
int *i = new int;  
int *t = new int[10];  
delete i;  
delete [] t;
```

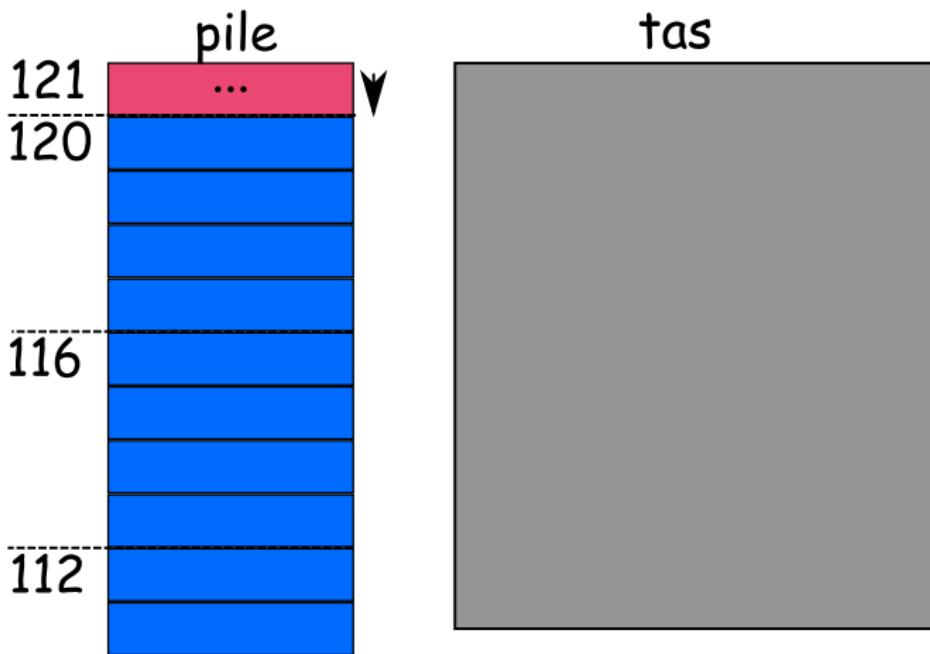
Question: combien y-a-t-il d'allocations effectuées dans cet exemple?

Réponse: 4, 2 automatiques et 2 dynamiques.

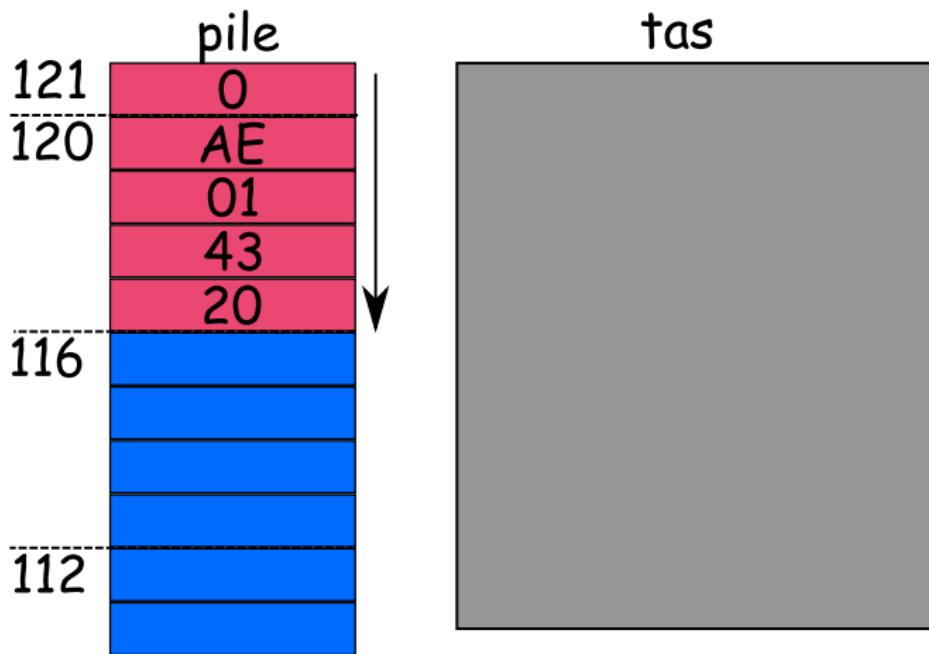
En effet, il y a deux allocations automatiques de sizeof(int *) pour les variables i et t

et deux allocations dynamiques effectuées avec **new** et dont les adresses de début de zones mémoires sont conservées dans i et t.

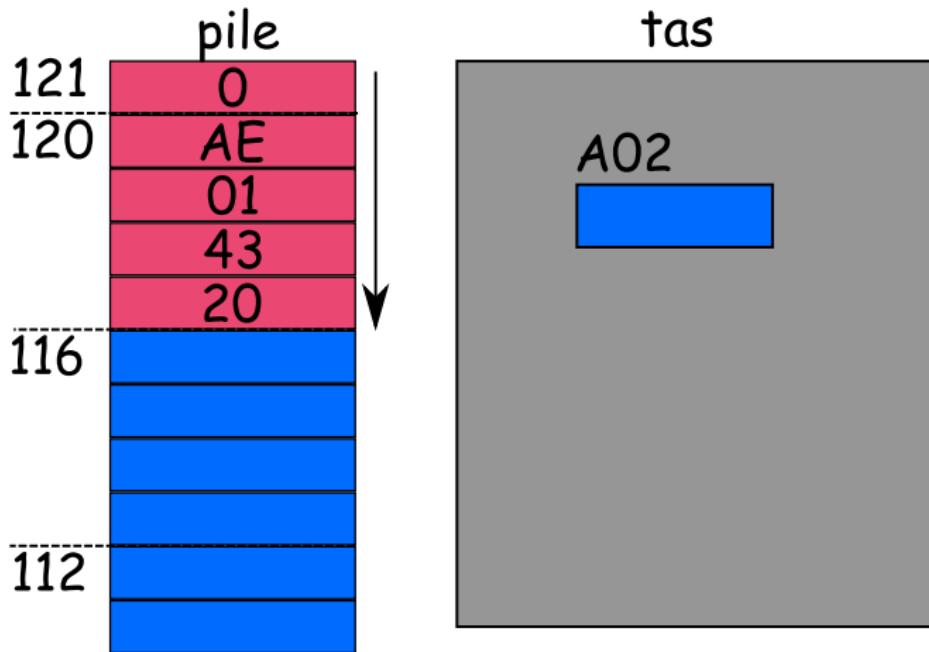
Allocation: exemple complet



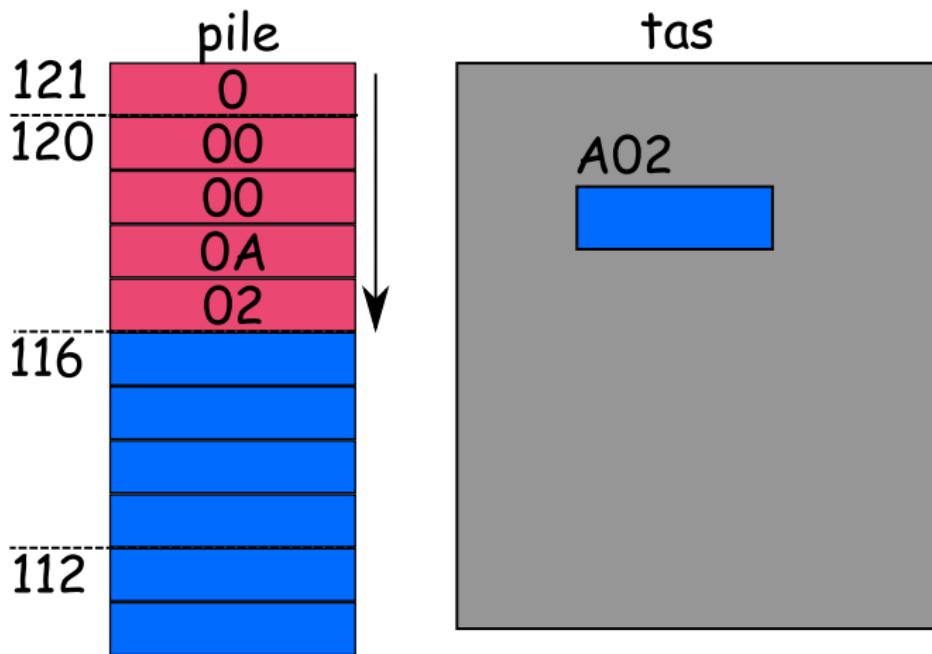
Allocation: exemple complet



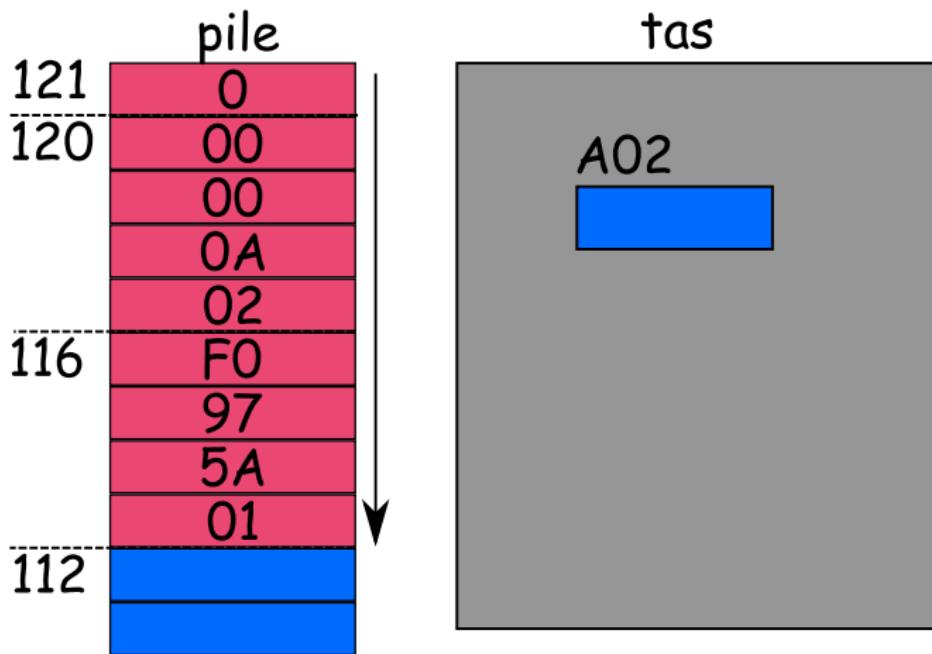
Allocation: exemple complet



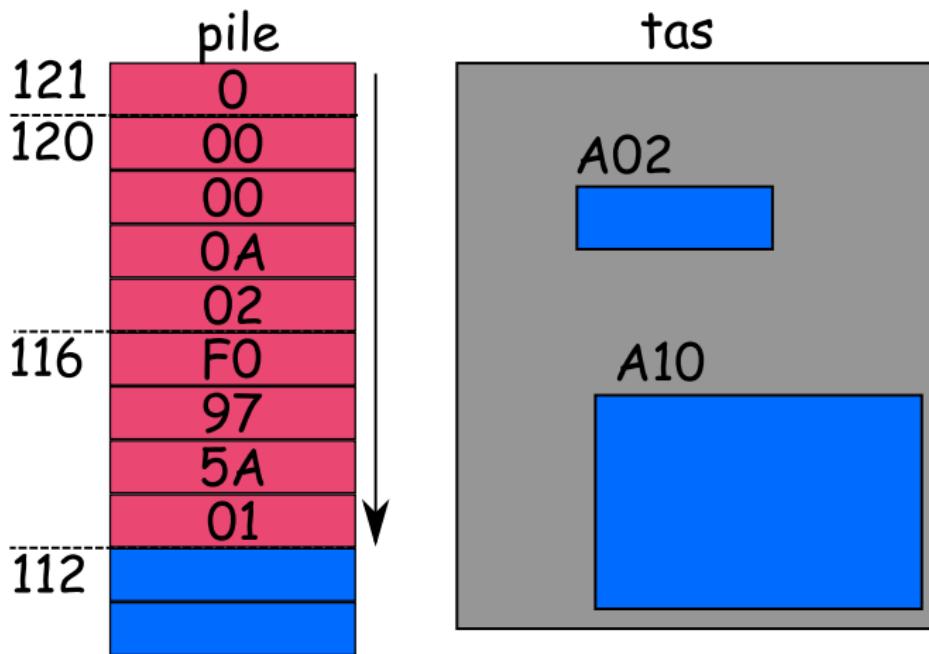
Allocation: exemple complet



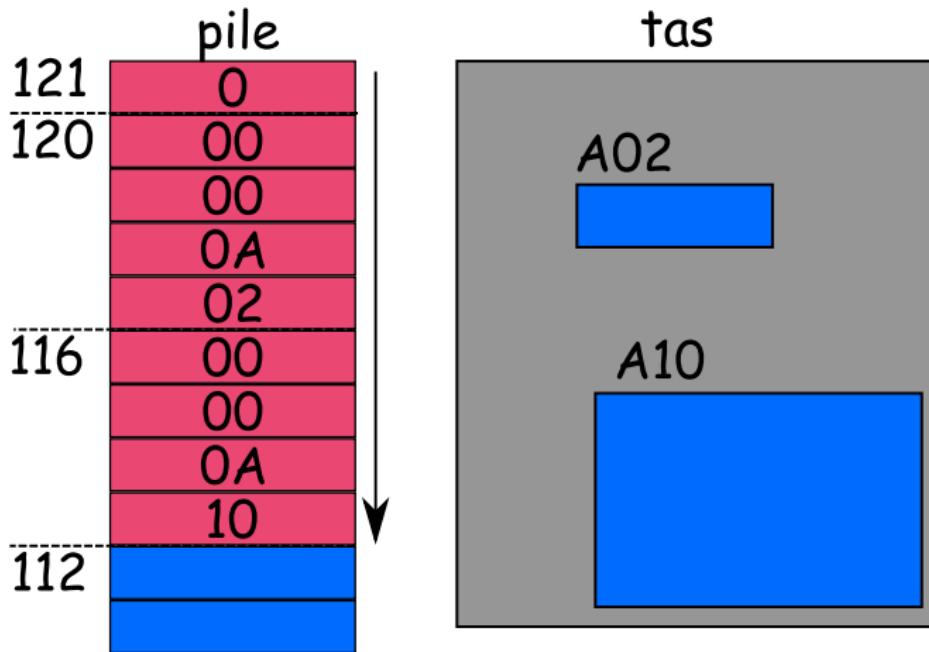
Allocation: exemple complet



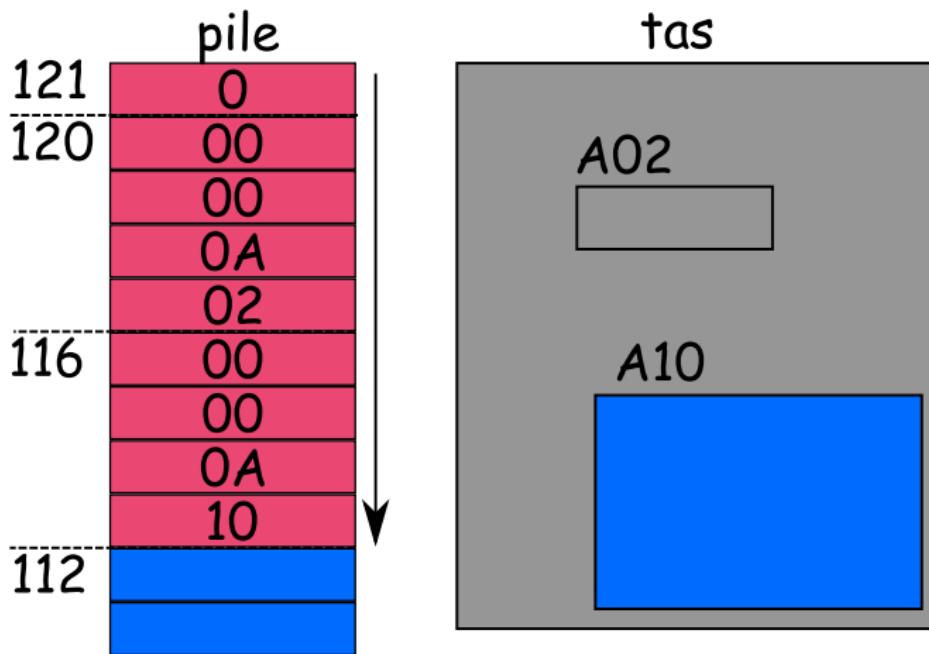
Allocation: exemple complet



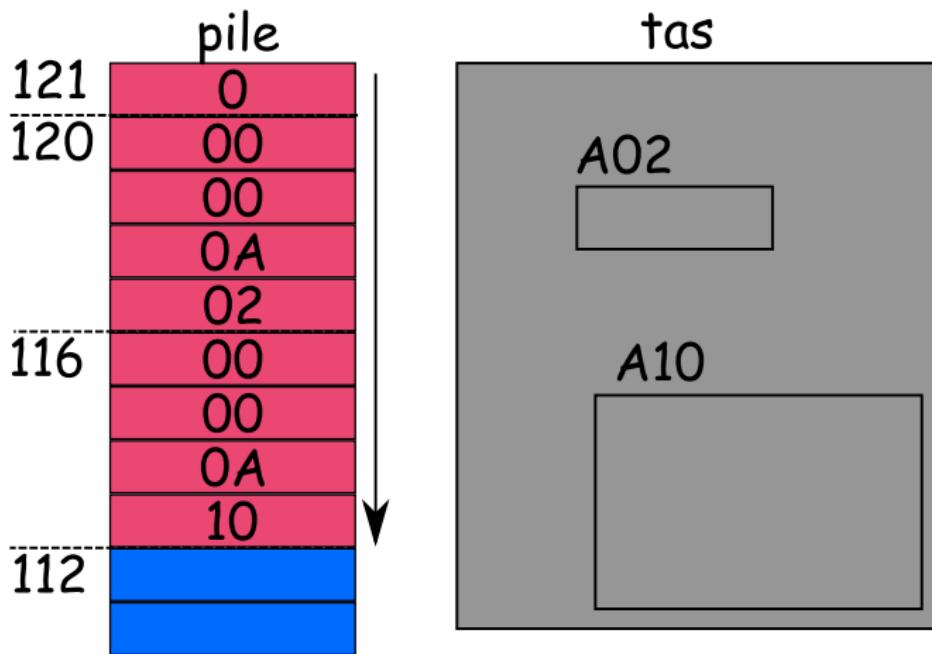
Allocation: exemple complet



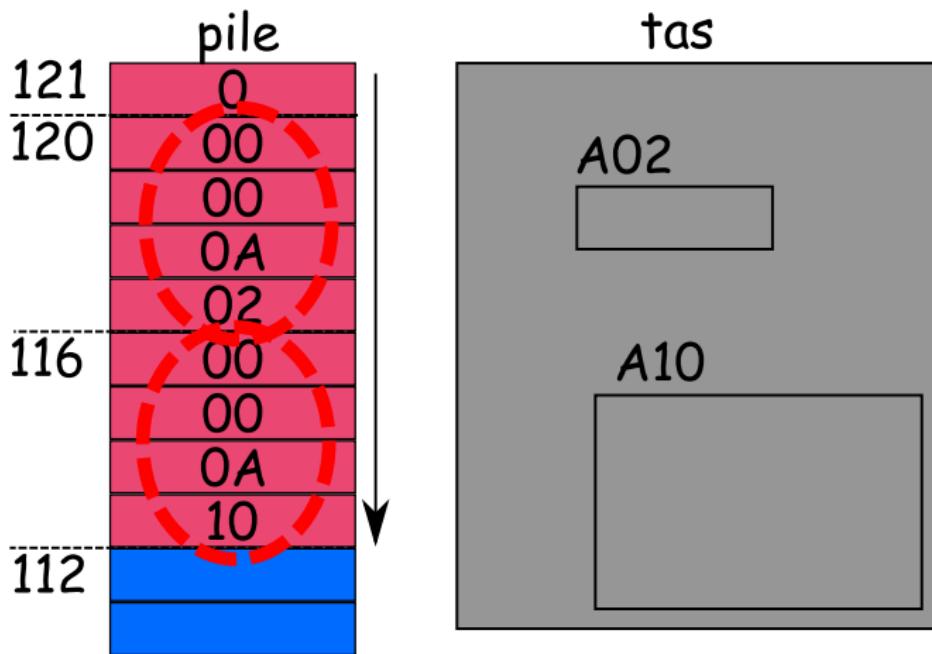
Allocation: exemple complet



Allocation: exemple complet



Allocation: exemple complet



Allocation: exemple complet

exemple complet

```
int *i = new int;
int *t = new int [10];
delete i;
i=NULL;
delete [] t;
t=NULL;
```

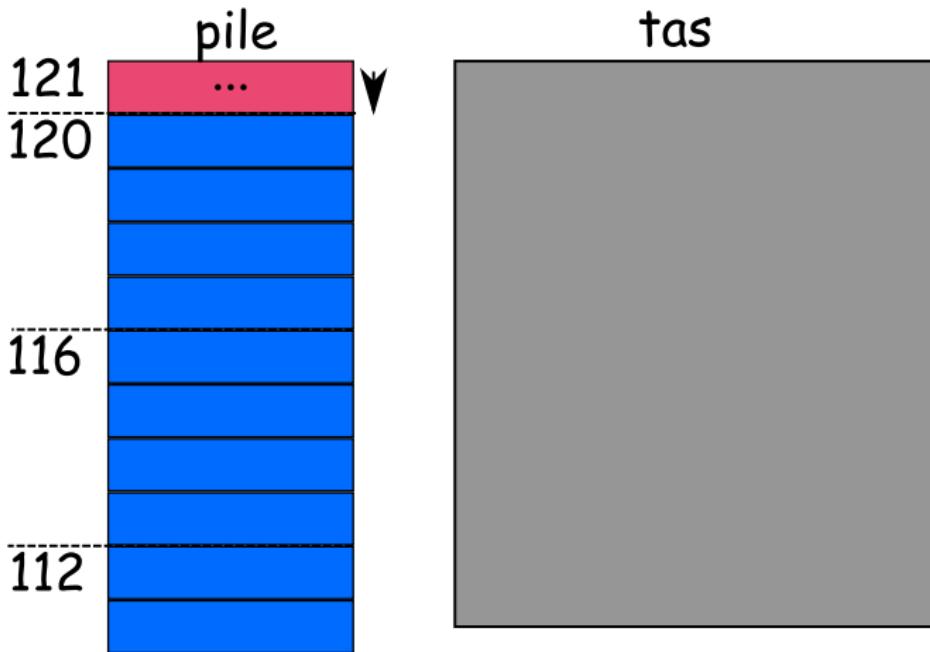
Allocation: exemple complet

exemple complet

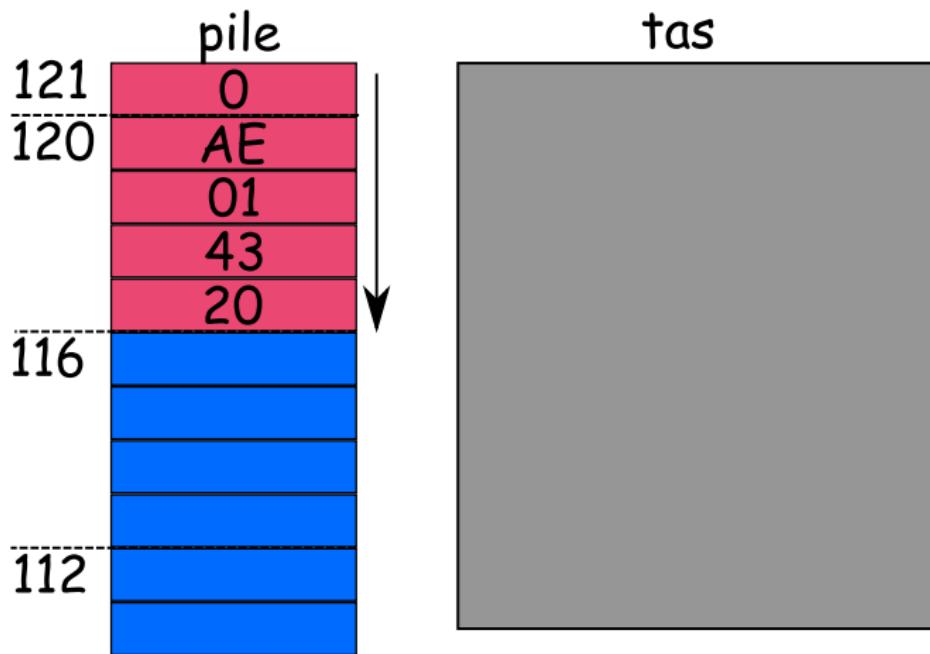
```
int *i = new int;
int *t = new int [10];
delete i;
i=NULL;
delete [] t;
t=NULL;
```

On réinitialise toujours un pointeur après avoir libéré la mémoire à l'adresse contenue dans ce pointeur.

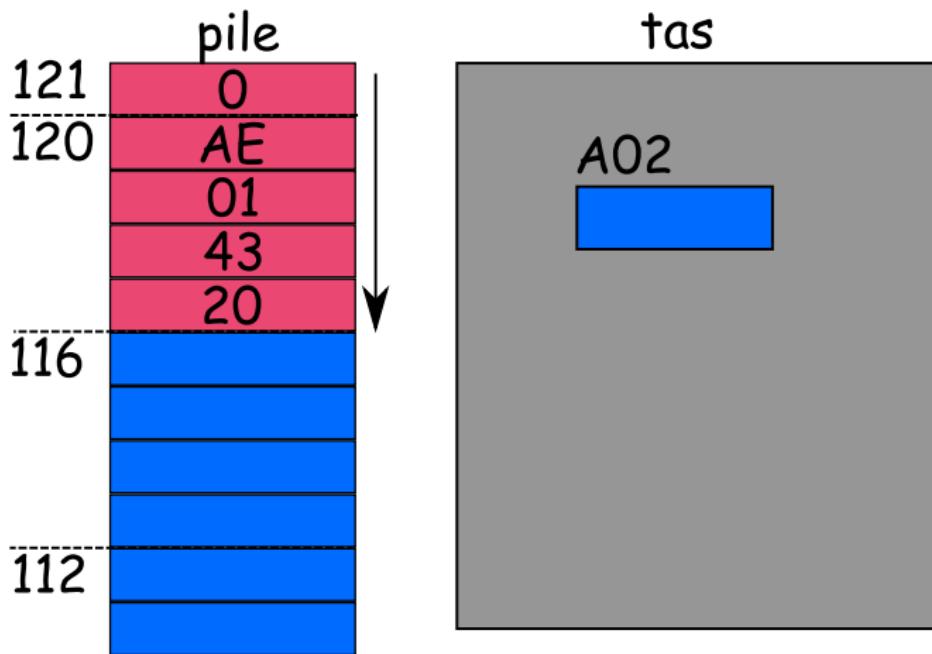
Allocation: exemple complet



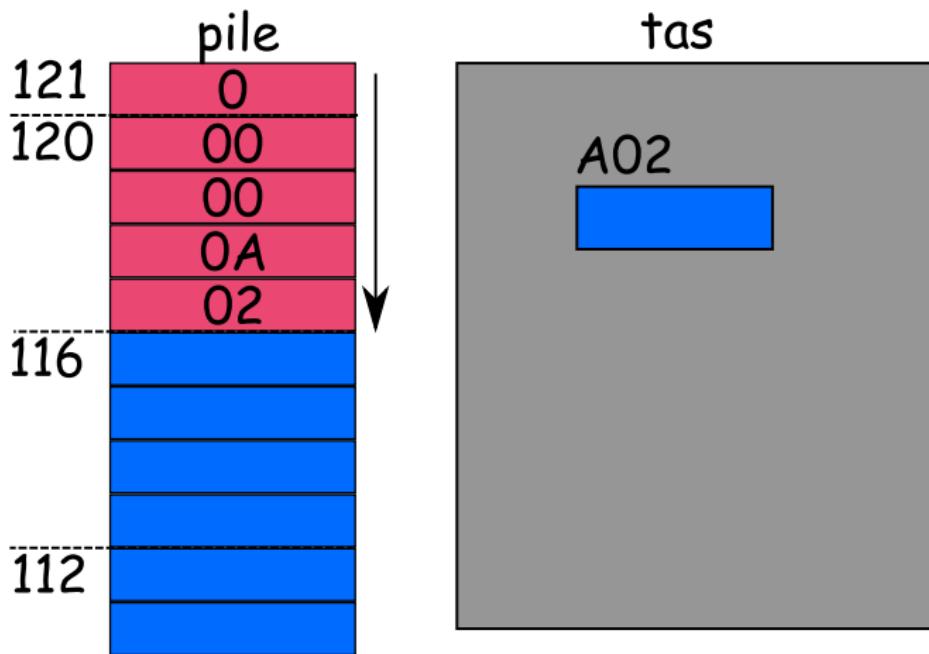
Allocation: exemple complet



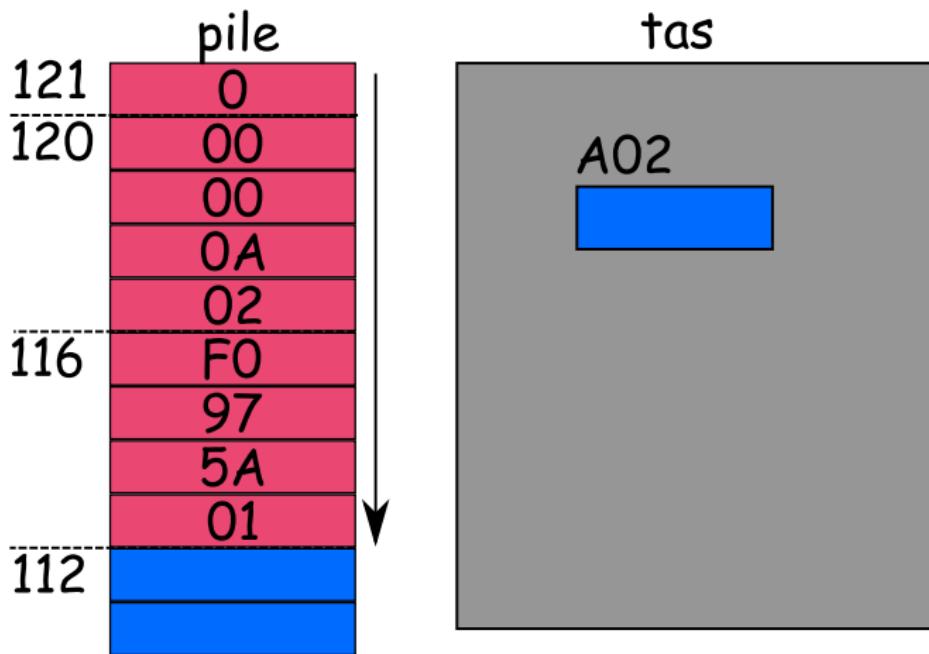
Allocation: exemple complet



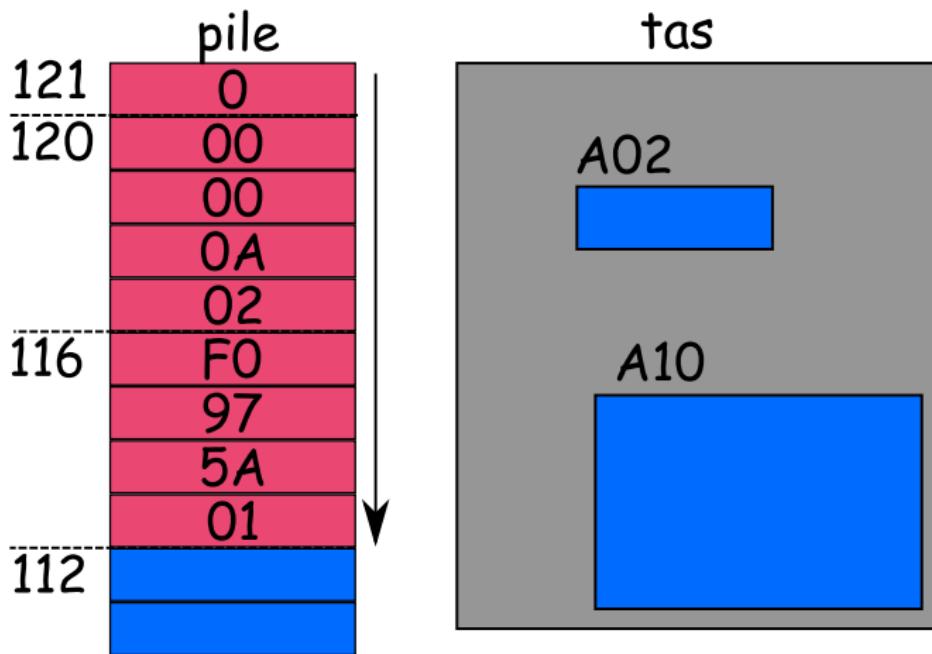
Allocation: exemple complet



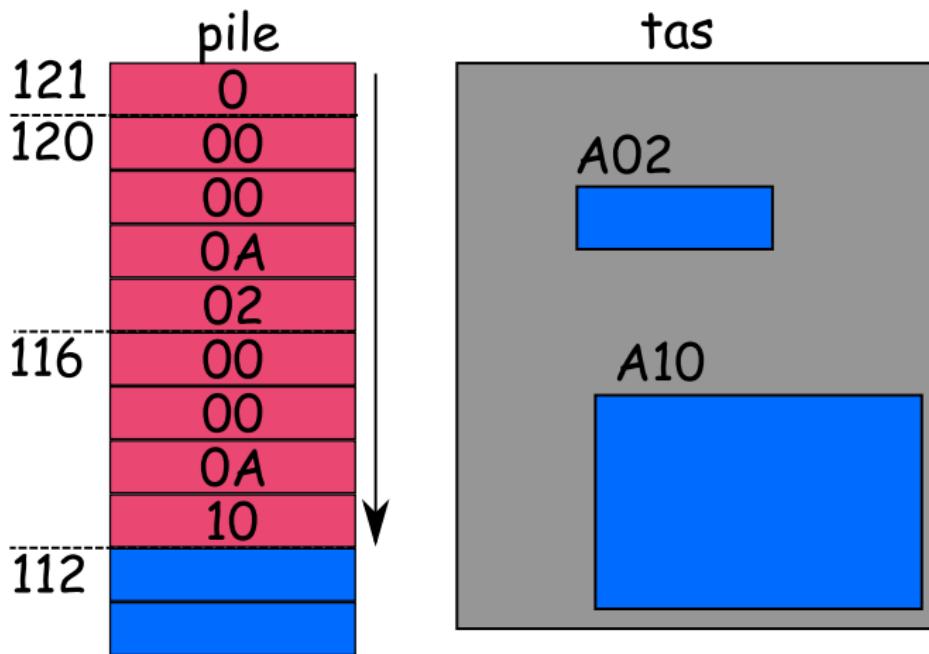
Allocation: exemple complet



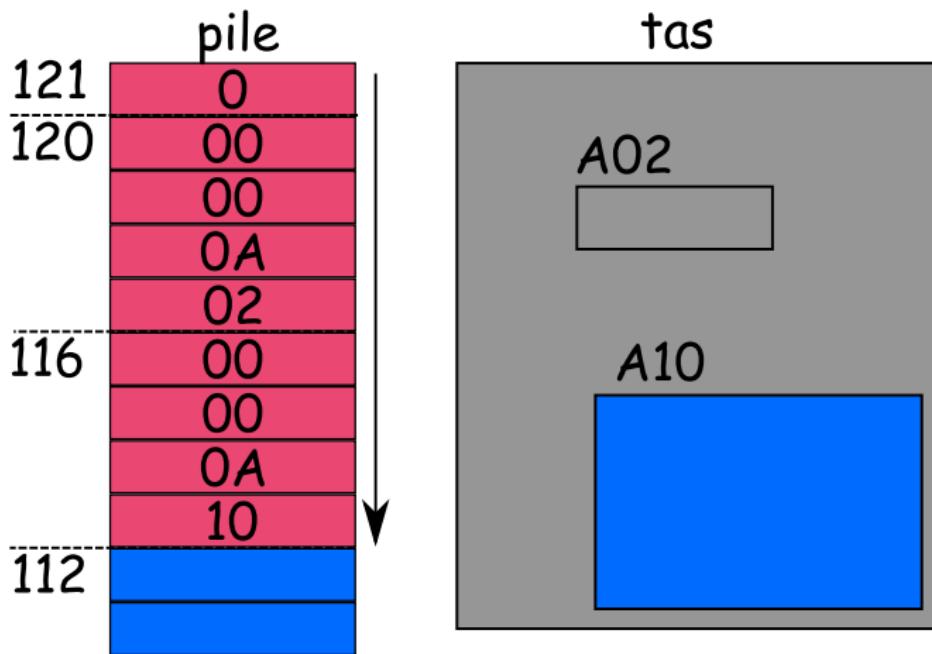
Allocation: exemple complet



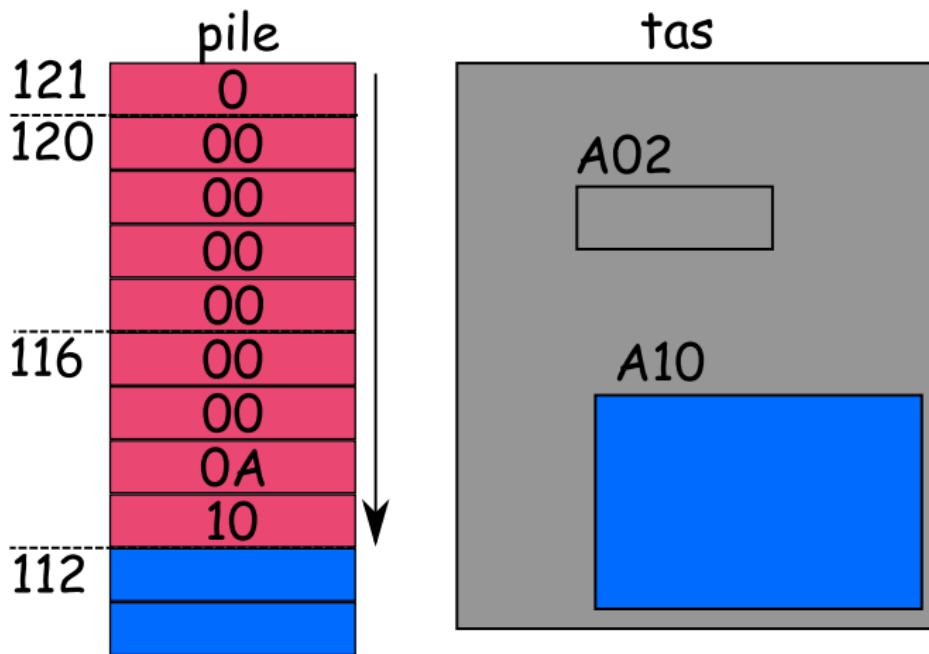
Allocation: exemple complet



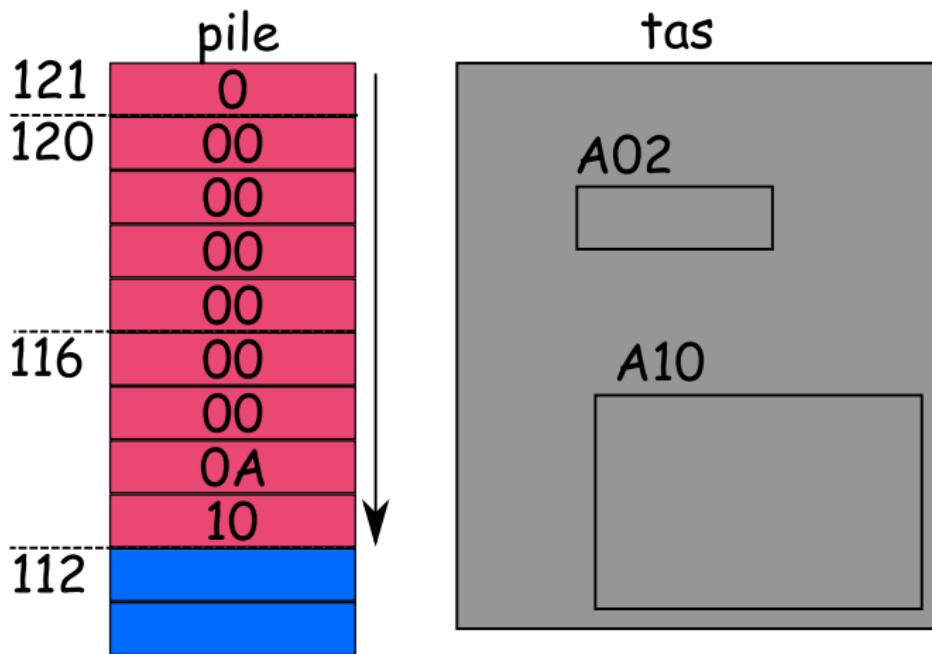
Allocation: exemple complet



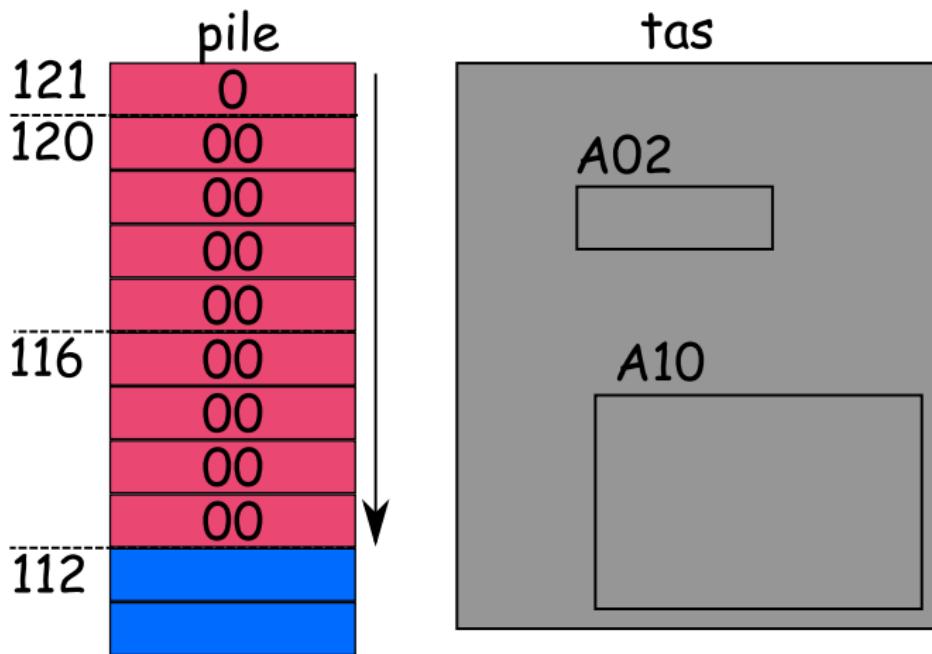
Allocation: exemple complet



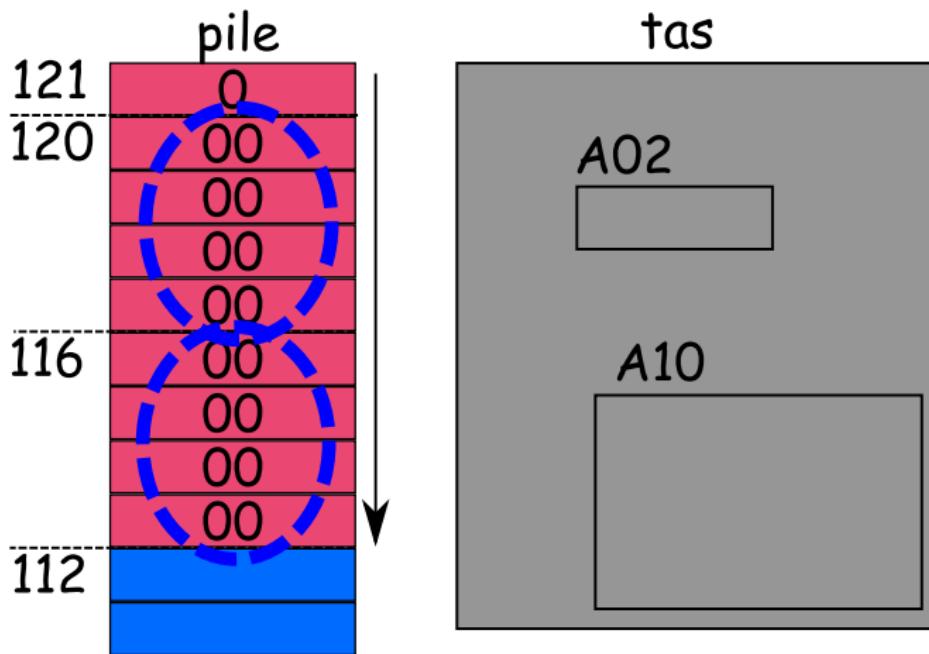
Allocation: exemple complet



Allocation: exemple complet



Allocation: exemple complet



Plan

3 Classes

Plan

3 Classes

- Définition et déclaration
- Visibilité, friend, struct
- Attributs et méthodes
- this
- espace de nom
- constructeur
- destructeur
- static
- exemple

Les classes: déclaration

Une classe consiste en un regroupement de méthodes et d'attributs.

Exemple

```
class NomClasse {  
    attributes  
    ...  
    methodes  
    ...  
};
```

L'ordre des déclarations ne compte pas.

La classe est une description des données internes et comportements qu'aura une instance générée par cette classe.

Les classes: instantiation

L'instanciation (ou réification), c'est à dire la création d'un objet instance (ressource) à partir de l'objet classe (description/générateur) peut se faire de façon dynamique:

allocation dynamique

```
A *a=new A();
```

ou automatique:

allocation automatique

```
A a;
```

Dans les deux cas, une ressource (adresse mémoire) est associée à l'instance. Dans l'allocation dynamique, cette ressource est située dans le tas, dans le cas automatique elle est située dans la pile.

Plan

3 Classes

- Définition et déclaration
- **Visibilité, friend, struct**
- Attributs et méthodes
- this
- espace de nom
- constructeur
- destructeur
- static
- exemple

Visibilité: définition

On contrôle l'accès aux méthodes et attributs pour une sous-classe ou un objet extérieur avec `public`, `protected` et `private`:

	la classe	sous-classe	exterieur
public	oui	oui	oui
protected	oui	oui	non
private	oui	non	non

Remarque: Les membres `protected` et `private` ne peuvent être accédés qu'à partir d'une fonction membre de l'instance

Visibilité: déclaration

La visibilité par défaut dans une classe est **private**. On modifie la visibilité de la façon suivante

Source

```
class NomClasse {  
attributs et méthodes privés  
public:  
attributs et méthodes publiques  
protected:  
attributs et méthodes protégés  
...  
};
```

On peut à tout moment changer la visibilité, celle-ci sera appliquée à toutes les déclarations suivantes jusqu'au prochain changement de visibilité.

Les structures: déclaration

Les structures se déclarent comme en C:

Exemple

```
struct Nom {  
    ...  
    attributs et méthodes publiques  
    ...  
};
```

En C++, les structures sont des classes dont la visibilité par défaut est **public**.

Par conséquent, elle peuvent contenir des attributs et des méthodes.

Deux noms de type sont associés à la structure: struct Nom et Nom

on voit que les structures C sont alors un cas particulier des structures C++

Les structures: instantiation

L'instanciation des structures se fait comme pour les classes.

allocation dynamique:

```
struct A { ... };
A *a = new A(); // erreur si fonction A
A *a = new struct A();
struct A *a=new struct A();
```

allocation automatique:

```
struct A { ... };
A a;
struct A a;
```

Friend: définition

Dans une classe A, le mot clé **friend** permet de donner à une fonction ou une autre classe les même droits qu'une méthode de A.

	la classe	friend	sous-classe	exterieur
public	oui	oui	oui	oui
protected	oui	oui	oui	non
private	oui	oui	non	non

une fonction famie d'une classe A pourra accéder aux attributs privés de A ainsi qu'aux attributs protected d'une des classes parents de A

Friend: déclaration

La déclaration se fait dans la classe en indiquant soit le prototype de la fonction amie soit le nom de la classe amie:

Classe amie

```
class NomClasse {  
...; friend class NomClasseAmie ; ...  
};
```

Fonction amie

```
class NomClasse {  
...; friend void fonctionAmie(int, char, NomClasse) ; ...  
};
```

la visibilité courante n'importe pas pour déclarer une classe ou fonction amie.

Plan

3 Classes

- Définition et déclaration
- Visibilité, friend, struct
- Attributs et méthodes**
- this
- espace de nom
- constructeur
- destructeur
- static
- exemple

Attributs et méthodes: définition

Une classe est un regroupement d'attributs, propriétés interne qu'aura une instance de la classe, et de méthodes, comportements qu'aura une instance.

Ces attributs sont déclarés à l'intérieur de la classe.

L'implémentation des méthodes peut se faire au moment de la déclaration ou à l'extérieur de la classe. On l'écrira dans la classe pour:

- les classes template
- les classes à usage locale au fichier
- l'inlining

Sinon on écrit un fichier entête “NomClasse.hpp” et un fichier source “NomClasse.cpp”.

Exemple de classe: 1

Un seul fichier

```
class Test {  
    int i; // attribut privé  
public:  
    void print(){ /* méthode publique */  
        printf("Test\n");  
    }  
};
```

Exemple de classe: 1

Un seul fichier

```
class Test {  
    int i; // attribut privé  
public:  
    void print() { /* méthode publique */  
        printf("Test\n");  
    }  
};  
  
Test t;  
t.print();
```

Exemple de classe: 1

Un seul fichier

```
class Test {  
    int i; // attribut privé  
public:  
    void print(){ /* méthode publique */  
        printf("Test\n");  
    }  
};  
  
Test *t = new Test;  
t->print();
```

Exemple de classe: 1

Un seul fichier

```
class Test {  
    int i; // attribut privé  
public:  
    void print(){ /* méthode publique */  
        printf("Test\n");  
    }  
};  
  
Test *t = new Test;  
t->print();
```

La compilation se fait avec la commande:

```
g++ -Wall Test.cpp -o test
```

Exemple de classe: 2

Deux fichiers

Test.hpp:

```
class Test {  
    int i; // attribut privé  
public:  
/* méthode publique */  
    void print();  
};
```

Exemple de classe: 2

Deux fichiers

Test.hpp:

```
class Test {  
    int i; // attribut privé  
public:  
/* méthode publique */  
    void print();  
};
```

Test.cpp:

```
#include "Test.hpp"  
void Test::print() {  
    printf("Test\n");  
}
```

Exemple de classe: 2

Deux fichiers

Test.hpp:

```
class Test {  
    int i; // attribut privé  
public:  
/* méthode publique */  
    void print();  
};
```

Test.cpp:

```
#include "Test.hpp"  
void Test::print() {  
    printf("Test\n");  
}
```

Exemple.cpp:

```
Test t;  
t.print();
```

Exemple de classe: 2

Deux fichiers

Test.hpp:

```
class Test {  
    int i; // attribut privé  
public:  
/* méthode publique */  
    void print();  
};
```

Test.cpp:

```
#include "Test.hpp"  
void Test::print() {  
    printf("Test\n");  
}
```

Exemple.cpp:

```
Test *t = new Test;  
t->print();
```

Exemple de classe: 2

Deux fichiers

Test.hpp:

```
class Test {  
    int i; // attribut privé  
public:  
/* méthode publique */  
    void print();  
};
```

Test.cpp:

```
#include "Test.hpp"  
void Test::print() {  
    printf("Test\n");  
}
```

Exemple.cpp:

```
Test *t = new Test;  
t->print();
```

La compilation se fait en deux temps:

```
g++ -Wall -c Test.cpp -o Test.o  
g++ -Wall -c Exemple.cpp -o Exemple.o  
g++ Test.o Exemple.o -o Exemple
```

Accès aux attributs

L'accès aux champs d'une classe se fait à l'aide de . ou -> selon que l'on dispose d'une instance de classe ou d'un pointeur sur une instance de classe.

Accès aux attributs

L'accès aux champs d'une classe se fait à l'aide de `.` ou `->` selon que l'on dispose d'une instance de classe ou d'un pointeur sur une instance de classe.

Tout membre de classe a pour nom NomClasse::membre. Lorsqu'il n'y a pas d'ambiguité, NomClasse:: peut être omis.

Accès aux attributs

L'accès aux champs d'une classe se fait à l'aide de . ou -> selon que l'on dispose d'une instance de classe ou d'un pointeur sur une instance de classe.

Tout membre de classe a pour nom NomClasse::membre. Lorsqu'il n'y a pas d'ambiguité, NomClasse:: peut être omis.

Cas automatique

```
ClasseA a;  
a.i = 1 ; // i est un attribut public de ClasseA  
a.ClasseA::i = 1 ; // idem  
a.methode(); // appel de méthode  
a.ClasseA::methode(); // idem
```

Dans ce cas l'instance se trouve dans la pile.

Accès aux attributs

L'accès aux champs d'une classe se fait à l'aide de `.` ou `->` selon que l'on dispose d'une instance de classe ou d'un pointeur sur une instance de classe.

Tout membre de classe à pour nom NomClasse::membre. Lorsqu'il n'y a pas d'ambiguité, NomClasse:: peut être omis.

Cas dynamique

```
ClasseA *a = new ClasseA();
a->i = 1 ; // i est un attribut public de ClasseA
a->ClasseA::i = 1 ; // idem
a->methode(); // appel de méthode
a->ClasseA::methode(); // idem
```

Dans ce cas l'instance est allouée dynamiquement dans le tas.

Plan

3 Classes

- Définition et déclaration
- Visibilité, friend, struct
- Attributs et méthodes
- **this**
- espace de nom
- constructeur
- destructeur
- static
- exemple

Le mot clé **this**

Le mot clé **this** permet, lors d'un appel de méthode, de faire référence à l'instance source (site d'appel).

Le mot clé **this**

Le mot clé **this** permet, lors d'un appel de méthode, de faire référence à l'instance source (site d'appel).

Le type de la variable **this** est NomDeClasse const *: c'est un pointeur. (*Nous verrons plus tard le sens de **const**.*)

Le mot clé **this**

Le mot clé **this** permet, lors d'un appel de méthode, de faire référence à l'instance source (site d'appel).

Le type de la variable **this** est NomDeClasse const *: c'est un pointeur. (*Nous verrons plus tard le sens de **const**.*)

Exemple

```
class A {  
    int attributPrive;  
public:  
    void setAttribut(int valeur) {  
        this->attributPrive = valeur ;  
    }  
};
```

Le mot clé **this**

Le mot clé **this** permet, lors d'un appel de méthode, de faire référence à l'instance source (site d'appel).

Le type de la variable **this** est NomDeClasse const *: c'est un pointeur. (*Nous verrons plus tard le sens de **const**.*)

Exemple

```
class A {  
    int attributPrive;  
public:  
    void setAttribut(int valeur) {  
        attributPrive = valeur ;  
    }  
};
```

Lorsqu'il n'y a pas de variable locale de même nom que l'attribut, on peut omettre **this**.

Opérateur de portée

L'opérateur de portée `::` peut-être utilisé pour indiquer précisément la variable que l'on souhaite manipuler.

Opérateur de portée

L'opérateur de portée `::` peut-être utilisé pour indiquer précisément la variable que l'on souhaite manipuler.

Exemple:

```
int lunatique; // variable globale , beurk!
class A {
    char lunatique; // attribut privé de la classe
public:
    void printAllVariable(double lunatique){ // variable locale
        // à la méthode
        lunatique; // fait référence à la variable locale
        A::lunatique; // fait référence à l'attribut d'instance
        this->A::lunatique; // fait référence à l'attribut d'instance
        this->lunatique; // fait référence à l'attribut d'instance
        ::lunatique; // fait référence à la variable globale
    }
};
```

Plan

3 Classes

- Définition et déclaration
- Visibilité, friend, struct
- Attributs et méthodes
- this
- **espace de nom**
- constructeur
- destructeur
- static
- exemple

Les espaces de nom

Les espaces de nom permettent de regrouper un ensemble d'éléments (classes, variables globales, fonctions, ...).

Les espaces de nom

Les espaces de nom permettent de regrouper un ensemble d'éléments (classes, variables globales, fonctions, ...). Les espaces de nom sont utilisés pour structurer le code et pour éviter les problèmes de collisions.

Les espaces de nom

Les espaces de nom permettent de regrouper un ensemble d'éléments (classes, variables globales, fonctions, ...). Les espaces de nom sont utilisés pour structurer le code et pour éviter les problèmes de collisions. Pour déclarer un espace de nom on utilise le mot clé **namespace**:

Les espaces de nom

Les espaces de nom permettent de regrouper un ensemble d'éléments (classes, variables globales, fonctions, ...). Les espaces de nom sont utilisés pour structurer le code et pour éviter les problèmes de collisions. Pour déclarer un espace de nom on utilise le mot clé **namespace**:

Espace de nom:

```
namespace Nom { // début de l'espace de nom
    class A {
        ...
    };
    int i;
    int fonction();
} //fin de l'espace de nom
```

Espace de noms

Lors de l'écriture du code des fonctions ou des méthodes des classes dans le fichier .cpp on peut soit ré-ouvrir l'espace de nom, soit indiquer explicitement le nom de l'objet avec l'opérateur de portée.

Espace de noms

Lors de l'écriture du code des fonctions ou des méthodes des classes dans le fichier .cpp on peut soit ré-ouvrir l'espace de nom, soit indiquer explicitement le nom de l'objet avec l'opérateur de portée.

A.hpp

```
namespace tec {  
    class A {  
        public:  
            void m();  
    };  
}
```

Espace de noms

Lors de l'écriture du code des fonctions ou des méthodes des classes dans le fichier .cpp on peut soit ré-ouvrir l'espace de nom, soit indiquer explicitement le nom de l'objet avec l'opérateur de portée.

A.hpp

```
namespace tec {  
    class A {  
        public:  
            void m();  
    };  
}
```

A.cpp

```
#include "A.hpp"  
namespace tec {  
    void A::m(){  
        ...  
    }  
}
```

Espace de noms

Lors de l'écriture du code des fonctions ou des méthodes des classes dans le fichier .cpp on peut soit ré-ouvrir l'espace de nom, soit indiquer explicitement le nom de l'objet avec l'opérateur de portée.

A.hpp

```
namespace tec {  
    class A {  
        public:  
            void m();  
    };  
}
```

A.cpp

```
#include "A.hpp"  
void tec::A::m(){  
    ...  
}
```

Espace de noms

Lors de l'écriture du code des fonctions ou des méthodes des classes dans le fichier .cpp on peut soit ré-ouvrir l'espace de nom, soit indiquer explicitement le nom de l'objet avec l'opérateur de portée.

A.hpp

```
namespace tec {  
    class A {  
        public:  
            void m();  
    };  
}
```

A.cpp

```
#include "A.hpp"  
void tec::A::m(){  
    ...  
}
```

Il n'y pas pas de limite sur le nombre d'espaces de nom différents pouvant être définis dans un fichier ni sur le niveau d'imbrication de ces espaces. Cependant la taille des noms de fonctions est limitée et les espaces de nom font partis du nom de la fonction.

Espace de noms

Lors de l'écriture du code des fonctions ou des méthodes des classes dans le fichier .cpp on peut soit ré-ouvrir l'espace de nom, soit indiquer explicitement le nom de l'objet avec l'opérateur de portée.

A.hpp

```
namespace tec {  
    class A {  
        public:  
            void m();  
    };  
}
```

A.cpp

```
#include "A.hpp"  
void tec::A::m(){  
    ...  
}
```

D'une certaine manière, une classe peut-être vue comme un espace de nom particulier.

Espace de noms: **using**

Le mot clé **using** permet de rendre accessible tout ou partie d'un espace de nom dans une fonction, un espace de nom ou tout le code:

```
namespace spatial {  
    class Navette {};  
}  
namespace missions {  
    Navette n; //Erreur!  
}  
Navette n; //Erreur!  
void starWars(){  
    Navette n; //Erreur!  
}
```

Espace de noms: **using**

Le mot clé **using** permet de rendre accessible tout ou partie d'un espace de nom dans une fonction, un espace de nom ou tout le code:

```
namespace spatial {  
    class Navette {};  
}  
  
namespace missions {  
    spatial::Navette n; //ok  
}  
  
spatial::Navette n; //ok  
void starWars(){  
    spatial::Navette n; //ok!  
}
```

Utilisation de l'opérateur de portée

Espace de noms: **using**

Le mot clé **using** permet de rendre accessible tout ou partie d'un espace de nom dans une fonction, un espace de nom ou tout le code:

```
namespace spatial {  
    class Navette {};  
}  
using spatial::Navette;  
namespace missions {  
    Navette n; //ok  
}  
Navette n; //ok  
void starWars(){  
    Navette n; //ok!  
}
```

using sur un objet

Espace de noms: **using**

Le mot clé **using** permet de rendre accessible tout ou partie d'un espace de nom dans une fonction, un espace de nom ou tout le code:

```
namespace spatial {  
    class Navette {};  
}  
  
namespace missions {  
    Navette n; //Erreur!  
}  
  
using spatial::Navette;  
Navette n; //ok  
void starWars(){  
    Navette n; //ok!  
}
```

Le **using** n'est actif que sur les instructions qui suivent...

Espace de noms: **using**

Le mot clé **using** permet de rendre accessible tout ou partie d'un espace de nom dans une fonction, un espace de nom ou tout le code:

```
namespace spatial {  
    class Navette {};  
}  
  
namespace missions {  
    using spatial::Navette;  
    Navette n; //ok  
}  
  
Navette n; //Erreur!  
void starWars(){  
    Navette n; //Erreur!  
}
```

Le **using** dans un espace de nom est local à cet espace

Espace de noms: **using**

Le mot clé **using** permet de rendre accessible tout ou partie d'un espace de nom dans une fonction, un espace de nom ou tout le code:

```
namespace spatial {  
    class Navette {};  
}  
  
namespace missions {  
    Navette n; //Erreur!  
}  
  
Navette n; //Erreur!  
  
void starWars(){  
    using spatial::Navette;  
    Navette n; //ok  
}
```

Le **using** dans une fonction est local à cette fonction

Espace de noms: **using**

Le mot clé **using** permet de rendre accessible tout ou partie d'un espace de nom dans une fonction, un espace de nom ou tout le code:

```
namespace spatial {  
    class Navette {};  
}  
  
namespace missions {  
    Navette n; //Erreur!  
}  
  
Navette n; //Erreur!  
  
void starWars(){  
    using namespace spatial;  
    Navette n; //ok  
}
```

Le **using** peut se faire sur l'ensemble des objets d'un espace

Using

Le **using** crée des synonymes locaux entre les espaces de nom:

Fichier exemple.hpp:

```
namespace math{
    const double pi=3.14;
}

namespace cercle{
    using math::pi;
    double surf(double);
}
```

Fichier exemple.cpp:

```
#include 'exemple.hpp'

double cercle::surf(double r)
{
    return pi*r*r;
}
```

Using

Le **using** crée des synonymes locaux entre les espaces de nom:

Fichier exemple.hpp:

```
namespace math{
    const double pi=3.14;
}

namespace cercle{
    using math::pi;
    double surf(double);
}
```

Fichier exemple.cpp:

```
#include 'exemple.hpp'

double cercle::surf(double r)
{
    return pi*r*r;
}
```

- Permet de remplacer facilement une référence externe par une autre.

Using

Le **using** crée des synonymes locaux entre les espaces de nom:

Fichier exemple.hpp:

```
namespace math{
    const double pi=3.14;
}

namespace cercle{
    using math::pi;
    double surf(double);
}
```

Fichier exemple.cpp:

```
#include 'exemple.hpp'

double cercle::surf(double r)
{
    return pi*r*r;
}
```

- Permet de remplacer facilement une référence externe par une autre.
- Permet aussi l'introduction de modularité sans modification profonde du code.

“using namespace nom;” permet de créer des synonymes pour tout les éléments de nom

Plan

3 Classes

- Définition et déclaration
- Visibilité, friend, struct
- Attributs et méthodes
- this
- espace de nom
- **constructeur**
- destructeur
- static
- exemple

Constructeur: définition

Un constructeur est une méthode de classe qui est appelée après la réservation des ressources nécessaires à la création d'une instance de la classe et qui a pour objectif d'initialiser les attributs de cette instance.

Constructeur: définition

Un constructeur est une méthode de classe qui est appelée après la réservation des ressources nécessaires à la création d'une instance de la classe et qui a pour objectif d'initialiser les attributs de cette instance.

En C++, le constructeur est une méthode sans valeur de retour de même nom que la classe.

Constructeur: définition

Un constructeur est une méthode de classe qui est appelée après la réservation des ressources nécessaires à la création d'une instance de la classe et qui a pour objectif d'initialiser les attributs de cette instance.

En C++, le constructeur est une méthode sans valeur de retour de même nom que la classe.

On appelle constructeur par défaut le constructeur ne prenant aucun argument.

Constructeur: définition

Un constructeur est une méthode de classe qui est appelée après la réservation des ressources nécessaires à la création d'une instance de la classe et qui a pour objectif d'initialiser les attributs de cette instance.

En C++, le constructeur est une méthode sans valeur de retour de même nom que la classe.

On appelle constructeur par défaut le constructeur ne prenant aucun argument.

Dans le cas où la classe ne comporte aucun constructeur, le compilateur ajoute un constructeur par défaut. Celui-ci n'est plus présent à partir du moment où l'on a écrit au moins un constructeur.

Constructeur: exemple 1 fichier

Exemple.cpp

```
class Exemple {  
    double pop;  
public:  
    Exemple() { // Constructeur par défaut  
        pop=0;  
    }  
    Exemple(double d) { // Constructeur  
        pop=d;  
    }  
};
```

Constructeur: exemple 2 fichiers

Exemple.hpp

```
class Exemple{  
    double pop;  
public:  
    Exemple();  
    Exemple(double);  
};
```

Exemple.cpp

```
#include "Exemple.hpp"  
  
Exemple::Exemple () {  
    pop=0;  
}  
Exemple::Exemple (double d) {  
    pop=d;  
}
```

Constructeur: initialisation des attributs

Les attributs de l'instance peuvent être initialisés de la façon suivante:

Initialisation:

```
class A{
    int pop;
public:
    A(int value):pop(value){
    }
};
```

Constructeur: initialisation des attributs

Les attributs de l'instance peuvent être initialisés de la façon suivante:

Initialisation:

```
class A{
    int pop;
public:
    A(int pop):pop(pop){
    }
};
```

Il n'y a pas d'ambiguïté sur les noms de variables.

Constructeur: initialisation des attributs

Les attributs de l'instance peuvent être initialisés de la façon suivante:

Initialisation:

```
class A{
    int pop;
public:
    A(int pop):pop(pop*2){
    }
};
```

On peut effectuer des opérations

Constructeur: initialisation des attributs

Les attributs de l'instance peuvent être initialisés de la façon suivante:

Initialisation:

```
intdecremente(int i){  
    return i-1;  
}  
class A{  
    int pop;  
    public:  
        A(int pop):pop(decremente(pop)*2){  
        }  
};
```

On peut appeler des fonctions

Constructeur: initialisation des attributs

Les attributs de l'instance peuvent être initialisés de la façon suivante:

Initialisation:

```
class A{  
    int pop;  
    int m(){  
        return 1;  
    }  
    public:  
        A(int pop):pop((m()+pop)*2){  
        }  
};
```

On peut appeler des méthodes!

Constructeur: initialisation des attributs

On peut initialiser plusieurs attributs de cette façon. Cependant l'ordre des initialisations se fait toujours selon celui des déclarations des attributs dans la classe:

Source:

```
class A {  
    int a;  
    double d;  
    char c;  
  
public:  
    A();  
};  
  
A::A() :c('a'), a(0) {} //Warning
```

Constructeur: initialisation des attributs

On peut initialiser plusieurs attributs de cette façon. Cependant l'ordre des initialisations se fait toujours selon celui des déclarations des attributs dans la classe:

Source:

```
class A {  
    int a;  
    double d;  
    char c;  
  
public:  
    A();  
};  
  
A::A():a(0),c('a'){} //ok
```

Constructeur: et instantiation

Dans le cas de l'allocation automatique, il existe deux façons d'indiquer le constructeur à appeler:

Constructeur: et instantiation

Dans le cas de l'allocation automatique, il existe deux façons d'indiquer le constructeur à appeler:

Parenthésée

```
A a;  
A a(1);  
A a('c');  
A a(); // Erreur:  
       // pas d'instanciation!
```

Affectation

```
A a;  
A a=1;  
A a='c';
```

Constructeur: et instantiation

Dans le cas de l'allocation automatique, il existe deux façons d'indiquer le constructeur à appeler:

Parenthésée

```
A a;  
A a(1);  
A a('c');  
A a(); // Erreur:  
       // pas d'instanciation!
```

Affectation

```
A a;  
A a=1;  
A a='c';
```

L'utilisation du '=' au moment de l'instanciation fait nécessairement référence à un constructeur.

Constructeur: et instantiation

Pour le cas dynamique, seule la version parenthésée est légale.

Constructeur: et instantiation

Pour le cas dynamique, seule la version parenthésée est légale.

Exemple

```
new A; // Construction par defaut
new A(3); // Utilisation du constructeur
           // prenant un entier
new A(); //Construction par defaut
new A=5; // Erreur !
```

Constructeur: et instantiation

Cette syntaxe est valable pour les types primitifs:

Constructeur: et instantiation

Cette syntaxe est valable pour les types primitifs:

Affectation

```
int i=0;  
char c='a';  
char *p=NULL;
```

Parenthésée

```
int i(0);  
char c('a');  
char *p(NULL);
```

Constructeur: et instantiation

Il est possible d'instancier des classes de façon anonyme (aucun nom de variable n'est associé à l'instance):

Constructeur: et instantiation

Il est possible d'instancier des classes de façon anonyme (aucun nom de variable n'est associé à l'instance):

Source:

```
A(); // Instanciation utilisant le  
      // constructeur par défaut  
A(1); // Avec le constructeur A(int );  
A; // Erreur ! syntaxe invalide.
```

Constructeur: et instantiation

Il est possible d'instancier des classes de façon anonyme (aucun nom de variable n'est associé à l'instance):

Source:

```
A(); // Instanciation utilisant le  
      // constructeur par défaut  
A(1); // Avec le constructeur A(int );  
A; // Erreur ! syntaxe invalide.
```

Ce type d'instanciation peut-être utilisé lors du passage d'un argument à une fonction ou lors d'une levée d'exception.

Constructeur: et instantiation

Source:

```
void f(A a){...}
```

```
...
```

```
f(A(1)); //1. Appel de fonction
```

```
f(1); //2. Utilisation de l'instanciation implicite
```

```
...
```

```
throw A(); //3. Levée d'exception
```

Constructeur: et instantiation

Source:

```
void f(A a){...}
```

```
...
```

```
f(A(1)); //1. Appel de fonction
```

```
f(1); //2. Utilisation de l'instanciation implicite
```

```
...
```

```
throw A(); //3. Levée d'exception
```

L'appel au constructeur dans le cas 2 se fait de façon implicite. Le compilateur cherche une fonction `f(int)`, il liste l'ensemble des fonctions disponibles: `f(A)` et cherche une façon de construire `A` à partir d'un entier.

Constructeur: et tableaux

Lors de l'allocation de tableau, les instances du tableaux doivent être créées à l'aide du constructeur par défaut.

Source:

```
A *t=new A[10]; // dynamique  
A tableau2[10]; //automatique
```

Constructeur: et tableaux

Lors de l'allocation de tableau, les instances du tableaux doivent être créées à l'aide du constructeur par défaut.

Source:

```
A *t=new A[10]; // dynamique  
A tableau2[10]; //automatique
```

⇒ Une classe ne comportant pas de constructeur par défaut ne peut pas être utilisée avec les tableaux.

Constructeur: et tableaux

Lors de l'allocation de tableau, les instances du tableaux doivent être créées à l'aide du constructeur par défaut.

Source:

```
A *t=new A[10]; // dynamique  
A tableau2[10]; //automatique
```

⇒ Une classe ne comportant pas de constructeur par défaut ne peut pas être utilisée avec les tableaux.

⇒ On utilisera un tableau de pointeurs, chaque élément du tableau sera instancié séparément avec le bon constructeur.

Constructeur: et tableaux

Tableaux:

```
class A{  
public:  
    A(int);  
};  
...  
A t[10]; // Erreur !  
A *t[10]; // Tableau de pointeurs  
for(int i=0;i<10;i++)  
    t[i]=new A(3); // utilisation du constructeur  
                                // allocation dynamique  
...  
for(int i=0;i<10;i++)  
    delete t[i];
```

Constructeur: conversion

Comme nous l'avons vu, l'appel au constructeur peut se faire de façon implicite.

Exemple

```
int f (A) ;  
...  
f (1) ;
```

Le conversion se fait sur au plus un niveau

Constructeur: conversion

```
class A{  
public:  
A(int );  
};  
  
class B{  
public:  
B(A );  
};  
  
void f(B);  
  
f(1); //Ne marche pas!  
f(A(1)); // ok
```

Constructeur: conversion

```
class A{  
public:  
A(int );  
};  
  
class B{  
public:  
B(A );  
};  
  
void f(B);  
  
f(1); //Ne marche pas!  
f(A(1)); // ok
```

La notation `A a=1;` fait appel au système de conversion.

Constructeur: explicit

Un constructeur déclaré **explicit** ne sera pas utilisé pour effectuer une conversion:

```
class A{
public:
    explicit A(int );
};

A::A(int i){} // explicit n'est pas reporté ici
...
void f(A);
...
f(1); // Erreur: ne trouve pas la fonction f(int)
A a=1; // Erreur!
A b(1); // ok
```

Constructeur: par recopie

Le constructeur par recopie est le constructeur qui permet d'instancier une classe à partir d'une autre instance de cette classe.

Constructeur: par recopie

Le constructeur par recopie est le constructeur qui permet d'instancier une classe à partir d'une autre instance de cette classe.

Ce constructeur est utilisé entre autre lors de la transmission de paramètres à une fonction et lors d'une valeur de retour de fonction.

Constructeur: par recopie

Le constructeur par recopie est le constructeur qui permet d'instancier une classe à partir d'une autre instance de cette classe.

Ce constructeur est utilisé entre autre lors de la transmission de paramètres à une fonction et lors d'une valeur de retour de fonction.

```
A f(A p) { return p; }  
....  
A a;  
f(a)
```

- La variable `p`, locale à la fonction `f`, est allouée automatiquement et instanciée avec le constructeur par recopie à partir de l'instance `a`.
- La valeur de retour de `f` est instanciée par recopie de la valeur locale.

Constructeur: par recopie

L'idée est donc d'avoir un constructeur dans la classe A qui prend une instance de type A en argument:

Constructeur: par recopie

L'idée est donc d'avoir un constructeur dans la classe A qui prend une instance de type A en argument:

```
class A{  
public:  
    A(A a);  
};
```

Constructeur: par recopie

L'idée est donc d'avoir un constructeur dans la classe A qui prend une instance de type A en argument:

```
class A{  
public:  
    A(A a);  
};
```

Le problème est que pour instancier la variable locale au constructeur il nous faut le constructeur par recopie!

Constructeur: par recopie

L'idée est donc d'avoir un constructeur dans la classe A qui prend une instance de type A en argument:

```
class A{  
public:  
    A(A a);  
};
```

Le problème est que pour instancier la variable locale au constructeur il nous faut le constructeur par recopie!

La solution repose sur l'utilisation des références:

```
class A{  
public:  
    A(const A&);  
    A(A &);  
};
```

Références

Une référence peut être vue comme un “alias” sur une instance.

Références

Une référence peut être vue comme un “alias” sur une instance.

Une référence doit obligatoirement être initialisée au moment de sa création et ne peut référencer une autre instance par la suite.

```
int i;  
int &r=i;
```

Après l'initialisation:

- les variables i et r représentent la même donnée
- l'adresse de r (&r) est égale à l'adresse de i (&i).

Références

Une référence peut être vue comme un “alias” sur une instance.

Une référence doit obligatoirement être initialisée au moment de sa création et ne peut référencer une autre instance par la suite.

```
int i;  
int &r=i;
```

Après l'initialisation:

- les variables i et r représentent la même donnée
- l'adresse de r (&r) est égale à l'adresse de i (&i).

⇒ contrairement à un pointeur, une référence “pointe” toujours sur une instance.

Références

Exemple:

```
void swap(int &i, int &j) {  
    int k=i;  
    i=j;  
    j=k;  
}
```

```
int a=3,b=5;  
swap(a,b);  
int t[5];  
swap(t[1],t[2]);
```

Références

Exemple:

```
void swap(int &i, int &j) {  
    int k=i;  
    i=j;  
    j=k;  
}  
  
int a=3, b=5;  
swap(a, b);  
int t[5];  
swap(t[1], t[2]);
```

On peut aussi utiliser les références comme valeur de retour:

```
int &element(int *t, int i) {  
    return t[i];  
}  
  
element(t, 5)=10; // modifie t[5]
```

Références

Exemple:

```
void swap(int &i, int &j) {  
    int k=i;  
    i=j;  
    j=k;  
}  
  
int a=3, b=5;  
swap(a, b);  
int t[5];  
swap(t[1], t[2]);
```

On peut aussi utiliser les références comme valeur de retour:

```
int &element(int *t, int i) {  
    return t[i];  
}  
  
element(t, 5)=10; // modifie t[5]
```

Règle: Sauf pour les types primitifs, on ne prendra plus les instances par copie mais par référence.

Références

Parmis les avantages des références, on pourra noter qu'une référence ne peut pas être nulle:

```
void swap(int *i, int *j) {  
    assert(i!=NULL);  
    assert(j!=NULL);  
    int k=*i;  
    *i=*j;  
    *j=k;  
}
```

```
void swap(int &i,int &j)  
{  
    int k=i;  
    i=j;  
    j=k;  
}
```

Références

Parmis les avantages des références, on pourra noter qu'une référence ne peut pas être nulle:

```
void swap(int *i, int *j) {  
    assert(i!=NULL);  
    assert(j!=NULL);  
    int k=*i;  
    *i=*j;  
    *j=k;  
}
```

```
void swap(int &i,int &j)  
{  
    int k=i;  
    i=j;  
    j=k;  
}
```

Sauf si l'on souhaite laisser la possibilité de passer le pointeur NULL ou bien de pouvoir modifier la valeur du pointeur, on remplace le pointeur par une référence.

Références

Pour les valeurs de retour, on ne peut retourner une référence que si l'objet renvoyé existe en dehors de la fonction (comme pour les pointeurs):

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

```
int &p(int *i) {  
    return *i;  
}  
  
f()=0;  
int j;  
g(j)=0;  
h(j)=0;  
p(&j)=0;  
p(NULL)=0;
```

Références

```
void print(int &i) {  
    // 1 entier  
}
```

```
void print(Matrix &m) {  
    // 1 instance  
}
```

```
int i;  
print(i);  
Matrix m(10000);  
print(m);
```

```
void print(int i) {  
    // 2 entiers  
}
```

```
void print(Matrix m) {  
    // 2 instances  
}
```

Références

```
void print(int &i) {  
    // 1 entier  
}
```

```
void print(int i) {  
    // 2 entiers  
}
```

```
void print(Matrix &m) {  
    // 1 instance  
}
```

```
void print(Matrix m) {  
    // 2 instances  
}
```

```
int i;  
print(i);  
Matrix m(10000);  
print(m);
```

Problème de l'intégrité des données : comment être sûre que la fonction ne va pas modifier l'instance passée en argument ?

const: définition

Le mot clé **const** sert à indiquer qu'une donnée ne peut être modifiée:

```
const int i=5;
int j=10;
const int &r=j;
r=4; // erreur!
j=4; // ok
i=1; // erreur!
```

const: définition

Le mot clé **const** sert à indiquer qu'une donnée ne peut être modifiée:

const int i=5;	const int *i=0; // int const * i=0;
int j=10;	i= new int [10];
const int &r=j;	i[0]=5; // erreur!
r=4; // erreur!	int *const p= new int [10];
j=4; // ok	p=NULL; // erreur!
i=1; // erreur!	p[0]=5; // ok

const: définition

Le mot clé **const** sert à indiquer qu'une donnée ne peut être modifiée:

const int i=5;	const int *i=0; // int const * i=0;
int j=10;	i= new int [10];
const int &r=j;	i[0]=5; // erreur!
r=4; // erreur!	int *const p= new int [10];
j=4; // ok	p=NULL; // erreur!
i=1; // erreur!	p[0]=5; // ok

Une variable déclarée **const** doit être initialisée lors de sa déclaration.

const: définition

Le mot clé **const** sert à indiquer qu'une donnée ne peut être modifiée:

const int i=5;	const int *i=0; // int const * i=0;
int j=10;	i= new int [10];
const int &r=j;	i[0]=5; // erreur!
r=4; // erreur!	int *const p= new int [10];
j=4; // ok	p=NULL; // erreur!
i=1; // erreur!	p[0]=5; // ok

Une variable déclarée **const** doit être initialisée lors de sa déclaration.

déclaration:	const	type	const	*	const	*	const	p
lecture seule:	**p=		**p=		*p=		p=	

const: définition

Le mot clé **const** sert à indiquer qu'une donnée ne peut être modifiée:

const int i=5;	const int *i=0; // int const * i=0;
int j=10;	i= new int [10];
const int &r=j;	i[0]=5; // erreur!
r=4; // erreur!	int *const p= new int [10];
j=4; // ok	p=NULL; // erreur!
i=1; // erreur!	p[0]=5; // ok

Une variable déclarée **const** doit être initialisée lors de sa déclaration.

déclaration:	const	type	const	*	const	*	const	p
lecture seule:	**p=		**p=		*p=		p=	

Lorsque l'on dispose d'une référence constante sur une instance de classe, comment garantir que l'appel à une méthode de cette instance ne va pas modifier l'instance ?

const: et instances

Illustration du problème:

```
class A{  
    int i;  
public:  
    int get(){ return i; }  
    void set(int i){  
        this->i=i;  
    }  
};  
  
void f(const A &a) {  
    a.get();  
    a.set(0); // !  
}  
  
A b;  
f(b);
```

La référence est constante (l'instance ne doit pas être modifiée) mais l'appel de méthode modifie l'objet !

const: et instances

Illustration du problème:

```
class A{  
    int i;  
public:  
    int get(){ return i; }  
    void set(int i){  
        this->i=i;  
    }  
};  
  
void f(const A &a) {  
    a.get();  
    a.set(0); // !  
}  
  
A b;  
f(b);
```

La référence est constante (l'instance ne doit pas être modifiée) mais l'appel de méthode modifie l'objet !

⇒ On distingue deux types de méthodes: celles qui sont susceptibles de modifier l'objet et celles qui ne modifient pas l'objet.

const: méthodes

Une méthode de classe peut être typée **const**. Cela indique que dans cette méthode le type de **this** est const NomClasse const *

const: méthodes

Une méthode de classe peut être typée **const**. Cela indique que dans cette méthode le type de **this** est const NomClasse const *

L'ensemble des attributs de la classe deviennent alors constants eux aussi:

```
class A{
    int i;
    int *d;
public:
    void m() const;
    void p();
};

void A::p() {
    // this: A *const
}

void A::m() const{
    // this: const A * const
    // this: A const * const
    // this->i: const int
    // this->d: int * const
}
```

const

Lorsque l'on dispose d'une variable constante sur une instance, seules les méthodes indiquées comme constantes peuvent être appelées sur cette instance:

```
A a;  
const A & b=a;
```

```
a.m(); // ok  
a.p(); // ok  
b.p(); // erreur!  
b.m(); //ok
```

const

Lorsque l'on dispose d'une variable constante sur une instance, seules les méthodes indiquées comme constantes peuvent être appelées sur cette instance:

```
A a;  
const A & b=a;
```

```
a.m(); // ok  
a.p(); // ok  
b.p(); // erreur!  
b.m(); //ok
```

Une méthode **const** peut être appelée sur tout type de variable tandis qu'une méthode non **const** ne peut être appelée que sur une variable non constante.

const

Lorsque l'on dispose d'une variable constante sur une instance, seules les méthodes indiquées comme constantes peuvent être appelées sur cette instance:

```
A a;  
const A & b=a;
```

```
a.m(); // ok  
a.p(); // ok  
b.p(); // erreur!  
b.m(); //ok
```

Une méthode **const** peut être appelée sur tout type de variable tandis qu'une méthode non **const** ne peut être appelée que sur une variable non constante.
Règle: lors de l'écriture d'une méthode, on ne doit pas se demander si la méthode doit être **const** mais plutôt si on a besoin qu'elle ne soit pas **const**.

const

Lorsque l'on dispose d'une variable constante sur une instance, seules les méthodes indiquées comme constantes peuvent être appelées sur cette instance:

```
A a;  
const A & b=a;
```

```
a.m(); // ok  
a.p(); // ok  
b.p(); // erreur!  
b.m(); //ok
```

Une méthode **const** peut être appelée sur tout type de variable tandis qu'une méthode non **const** ne peut être appelée que sur une variable non constante.

Règle: lors de l'écriture d'une méthode, on ne doit pas se demander si la méthode doit être **const** mais plutôt si on a besoin qu'elle ne soit pas **const**.

Règle: Les arguments de fonction (non primitifs) seront des références constantes sauf si l'on veut modifier l'argument.

const: surcharge

Il est possible de surcharger une méthode en utilisant le **const**:

```
class A{
public:
void m() {
    printf("le site d'appel
n'est pas constant");
}
void m() const{
    printf("le site d'appel
est constant")
}
};
```

```
A a;
const A &b;
A const *p;
a.m(); //?
b.m(); //?
p->m(); //?
```

const: surcharge

Il est possible de surcharger une méthode en utilisant le **const**:

```
class A{
public:
void m() {
    printf("le site d'appel
n'est pas constant");
}
void m() const{
    printf("le site d'appel
est constant");
}
};
```

```
A a;
const A &b;
A const *p;
a.m(); //?
b.m(); //?
p->m(); //?
```

Le **const** fait partie de la signature de la méthode, il doit être présent dans le .cpp et le .hpp.

const: argument de fonction

Exemple d'utilisation du **const** pour les arguments d'une fonction.

```
class A{
    int i;
public:
    int get() { return i; }
};

void f(A a) { // copie
    printf("%d\n", a.get());
}

void g(A &a) {
    printf("%d\n", a.get());
}

void h(const A &a) {
    g(a); //!
    f(a);
    a.get();
}
```

Constructeur: recopie (suite)

Ainsi, le constructeur par recopie peut s'écrire de deux façons:

```
class A{  
public:  
    A(const A &); // 1  
    A(A &); // 2  
};
```

Constructeur: recopie (suite)

Ainsi, le constructeur par recopie peut s'écrire de deux façons:

```
class A{  
public:  
    A(const A &); // 1  
    A(A &); // 2  
};
```

Si l'on n'écrit pas de constructeur par recopie, le compilateur en fournit un. Ce constructeur fera une copie attribut par attribut en utilisant les constructeurs par recopie des types des attributs.

Constructeur: recopie (suite)

Ainsi, le constructeur par recopie peut s'écrire de deux façons:

```
class A{  
public:  
    A(const A &); // 1  
    A(A &); // 2  
};
```

Si l'on n'écrit pas de constructeur par recopie, le compilateur en fournit un. Ce constructeur fera un recopie attribut par attribut en utilisant les constructeurs par recopie des types des attributs.

Le constructeur fournit sera de type 1, si tous les constructeurs par recopie des attributs sont de type 1, sinon il sera de type 2.

Constructeur: par recopie

On préférera toujours écrire le constructeur par recopie sous la forme 1.

Constructeur: par recopie

On préférera toujours écrire le constructeur par recopie sous la forme 1.

Règle: Toute classe nécessitant une copie profonde (allocation dynamique des attributs) devra écrire un constructeur par recopie.

Constructeur: par recopie

```
class A{  
    int *i;  
public:  
    A(): i(new int(0)) {}  
    ~A() { delete i; }  
};  
  
A f() {  
    A a;  
    return a;  
}  
f();
```

Ici, il y a une erreur car on libère deux fois le même espace mémoire.

Constructeur: par recopie

```
class A{  
    int *i;  
public:  
    A():i(new int(0)){}  
    ~A(){ delete i; }  
    A(const A &a):i(new int(a.i)){}    f();  
};  
  
A f(){  
    A a;  
    return a;  
}
```

Constructeur: protected et private

Une classe n'ayant que des constructeurs protected ne pourra être instanciée que par

- Une sous-classe
- Une classe ou fonction amie
- Une classe ou fonction amie d'une sous-classe
- Une méthode statique

Constructeur: protected et private

Une classe n'ayant que des constructeurs protected ne pourra être instanciée que par

- Une sous-classe
- Une classe ou fonction amie
- Une classe ou fonction amie d'une sous-classe
- Une méthode statique

Une classe n'ayant que des constructeurs private ne pourra être instanciée que par

- Une classe ou fonction amie
- Une méthode statique

Plan

3 Classes

- Définition et déclaration
- Visibilité, friend, struct
- Attributs et méthodes
- this
- espace de nom
- constructeur
- destructeur**
- static
- exemple

Destructeur: définition

Le destructeur est une méthode d'instance n'ayant pas de type de retour et ayant pour nom: ~NomClasse

```
class Nom{  
public:  
    ~Nom() {}  
};  
void f () {  
    Nom n;  
    Nom *p=new Nom;  
    delete p; // Appel de p->~Nom()  
} // Appel de n.~Nom()
```

Destructeur: définition

Le destructeur est une méthode d'instance n'ayant pas de type de retour et ayant pour nom: ~NomClasse

```
class Nom{  
public:  
    ~Nom() {}  
};  
void f () {  
    Nom n;  
    Nom *p=new Nom;  
    delete p; // Appel de p->~Nom()  
} // Appel de n.~Nom()
```

Règle: toute classe ayant allouée dynamiquement ces attributs devra écrire un destructeur.

Si l'instance a été allouée dynamiquement alors l'appel au destructeur est déclenché par l'opérateur delete.

Si l'instance a été allouée dynamiquement alors l'appel au destructeur est déclenché par l'opérateur delete.

Si l'instance a été allouée automatiquement alors l'appel au destructeur se fait à la sortie du bloc d'instructions qui contient cette instance.

Si l'instance a été allouée dynamiquement alors l'appel au destructeur est déclenché par l'opérateur delete.

Si l'instance a été allouée automatiquement alors l'appel au destructeur se fait à la sortie du bloc d'instructions qui contient cette instance.

Les ressources mémoires associées à l'instance sont libérées après l'appel au destructeur.

Destructeur: exemple

```
class Tab{  
    int *data;  
public:  
    Tab(int s){  
        data= new int [s];  
    }  
    ~Tab(){  
        delete [] data;  
    }  
};
```

```
int main(){  
    Tab t(10); //allocation  
} //liberation
```

Destructeur: exemple

```
class Tab{  
    int *data;  
public:  
    Tab(int s){  
        data= new int [s];  
    }  
    ~Tab(){  
        delete [] data;  
    }  
};  
  
int main(){  
    Tab t(10); //allocation  
} //liberation
```

Il est ainsi possible de "simuler" un tableau dans la pile avec une désallocation automatique.

Destructeur: auto

```
class AutoInt{                                // Utilisation :  
    int *t;  
public:  
    AutoInt(int *t):t(t);  
    int get(int i){  
        return t[i];  
    }  
    void set(int i,int v){  
        t[i]=v;  
    }  
    ~AutoInt(){  
        if (t!=NULL)  
            delete [] t;  
    }  
};  
  
int main(){  
    AutoInt t(new int[100]);  
    t.get(0);  
    t.set(0,0);  
} // libération auto.
```

Destructeur: auto

```
class AutoInt{                                // Utilisation :  
    int *t;  
public:  
    AutoInt(int *t):t(t);  
    int get(int i){  
        return t[i];  
    }  
    void set(int i,int v){  
        t[i]=v;  
    }  
    ~AutoInt(){  
        if (t!=NULL)  
            delete [] t;  
    }  
};
```

```
int main(){  
    AutoInt t(new int [100]);  
    t.get(0);  
    t.set(0,0);  
} // libération auto.
```

Nous verrons comment améliorer ce code avec les opérateurs, cependant attention à la recopie!

Destructeur: auto

```
class AutoInt {                                // Utilisation :  
    int *t;  
public:  
    AutoInt(int *t):t(t);  
    int get(int i);  
    void set(int i, int v);  
    ~AutoInt(){  
        if (t!=NULL)  
            delete [] t;  
    }  
    AutoInt(AutoInt &p){  
        t=p.t;  
        p.t=NULL;  
    }  
};
```

```
void f(AutoInt ai){  
} // libération auto.  
  
AutoInt tab(){  
    AutoInt t(new int[100]);  
    return t;  
}  
int main(){  
    AutoInt t(new int[100]);  
    f(t);  
  
    AutoInt q(tab());  
} // libération auto.
```

Une solution consiste à transferer le pointeur...

Plan

3 Classes

- Définition et déclaration
- Visibilité, friend, struct
- Attributs et méthodes
- this
- espace de nom
- constructeur
- destructeur
- **static**
- exemple

Static: définition

Une variable statique est une variable dont la durée de vie est égale à celle du programme et dont la portée peut être réduite à une fichier, une classe, une fonction ou méthode selon l'endroit où elle est déclarée.

Fichier.hpp

```
void f();  
  
class A{  
    static int i;  
public:  
    static void m();  
};
```

Fichier.cpp

```
#include "Fichier.hpp"  
  
static double v;  
  
int A::i=0;  
  
void A::m() {  
    this; // erreur!  
}  
void f() {  
    static int compteur=0;  
}
```

Static: et classes

Dans le contexte des classes, une méthode publique statique est équivalente à une fonction amie (sauf pour la portée).

Static: et classes

Dans le contexte des classes, une méthode publique statique est équivalente à une fonction amie (sauf pour la portée).

On pourra utiliser les méthodes statiques en jonction avec un constructeur privé.

Static: et classes

Dans le contexte des classes, une méthode publique statique est équivalente à une fonction amie (sauf pour la portée).

On pourra utiliser les méthodes statiques en jonction avec un constructeur privé.

On peut utiliser le mot clé **static** en jonction avec **const** afin de définir des propriétés constantes de classes:

```
class Chaine {  
public:  
    static const char END='\\0';  
};
```

Les variables **static** et **const** sont les seules à pouvoir être initialisées lors de la déclaration dans le fichier .hpp.

Static: et classes

Dans le contexte des classes, une méthode publique statique est équivalente à une fonction amie (sauf pour la portée).

On pourra utiliser les méthodes statiques en jonction avec un constructeur privé.

On peut utiliser le mot clé **static** en jonction avec **const** afin de définir des propriétés constantes de classes:

```
class Chaine {  
public:  
    static const char END;  
};  
const char Chaine::END= '\0';
```

L'initialisation peut se faire à la déclaration dans ce cas. Chaine::END n'a pas d'existence réelle durant l'exécution (équivalent à une macro).

Plan

3 Classes

- Définition et déclaration
- Visibilité, friend, struct
- Attributs et méthodes
- this
- espace de nom
- constructeur
- destructeur
- static
- exemple

Une classe compteur

L'objectif est d'écrire une classe `Compteur` telle que plusieurs instances de cette classe puissent partager un même compteur qui sera ici implanté par un entier.

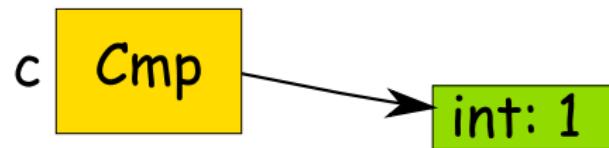
Une classe compteur

L'objectif est d'écrire une classe `Compteur` telle que plusieurs instances de cette classe puissent partager un même compteur qui sera ici implanté par un entier.

Ceci implique que le compteur soit alloué dynamiquement. En effet, on souhaite qu'il puisse être transmis entre des instances.

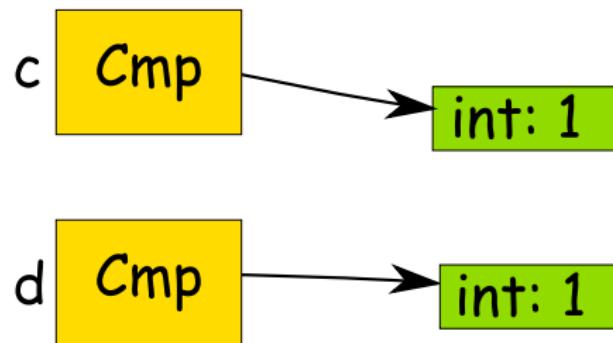
Compteur: Exemple1

Cmp c;



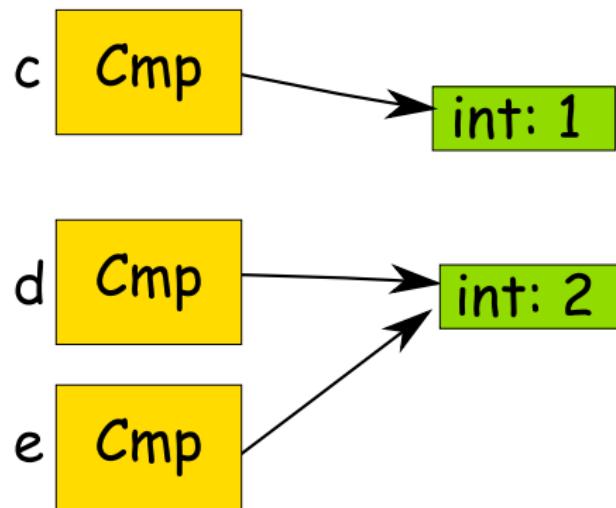
Compteur: Exemple1

```
Cmp c;  
Cmp d;
```



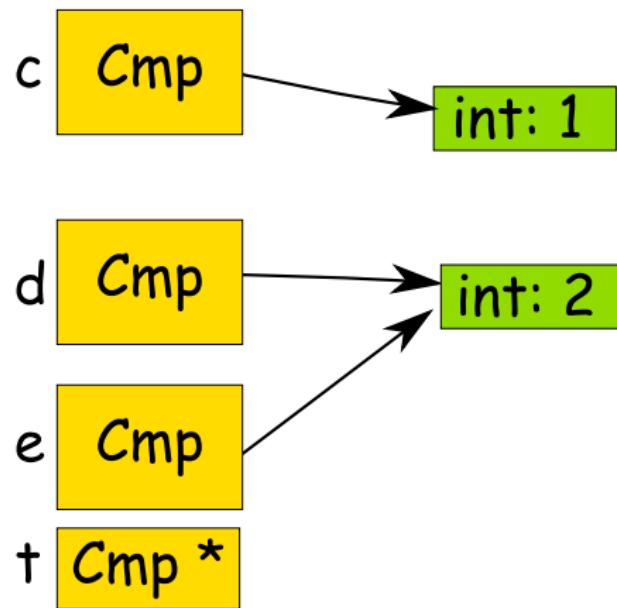
Compteur: Exemple1

```
Cmp c;  
Cmp d;  
Cmp e=d;
```



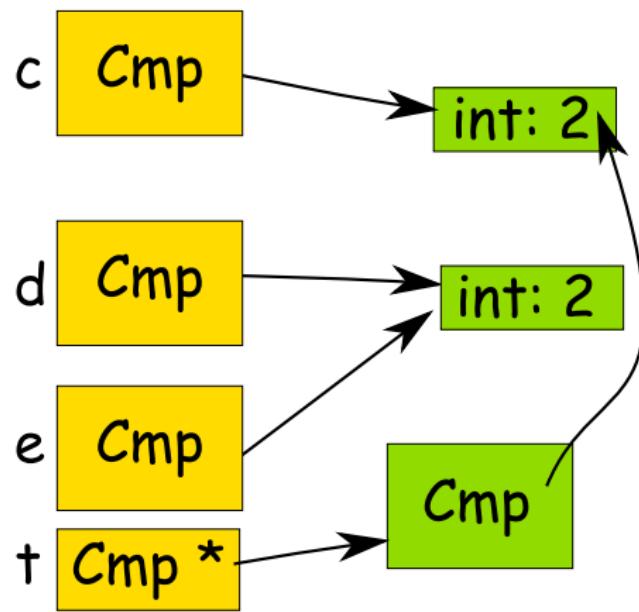
Compteur: Exemple1

```
Cmp c;  
Cmp d;  
Cmp e=d;  
Cmp *t;
```



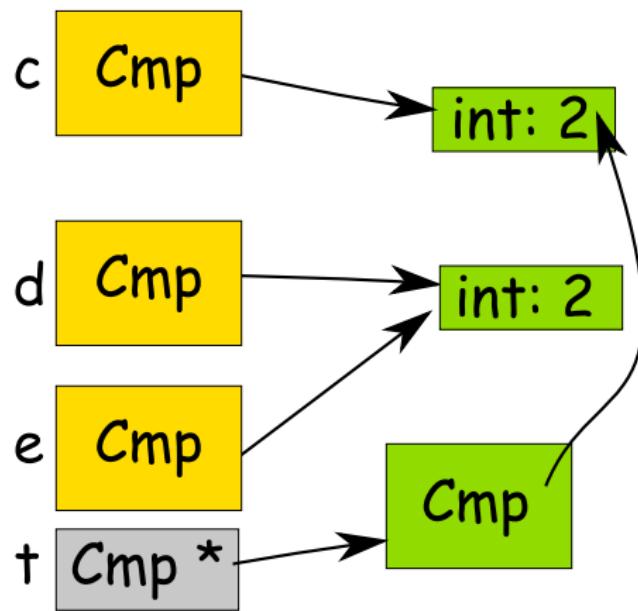
Compteur: Exemple1

```
Cmp c;  
Cmp d;  
Cmp e=d;  
Cmp *t;  
t=new Cmp(c); d
```



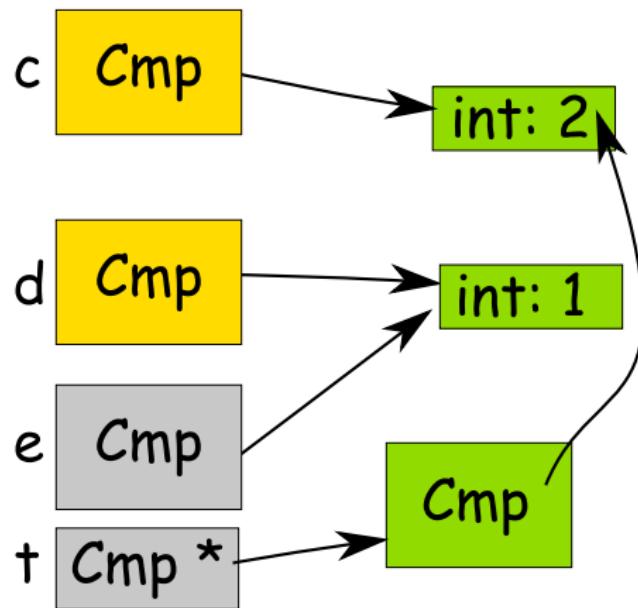
Compteur: Exemple1

```
Cmp c;  
Cmp d;  
Cmp e=d;  
Cmp *t;  
t=new Cmp(c); d  
...  
}
```



Compteur: Exemple1

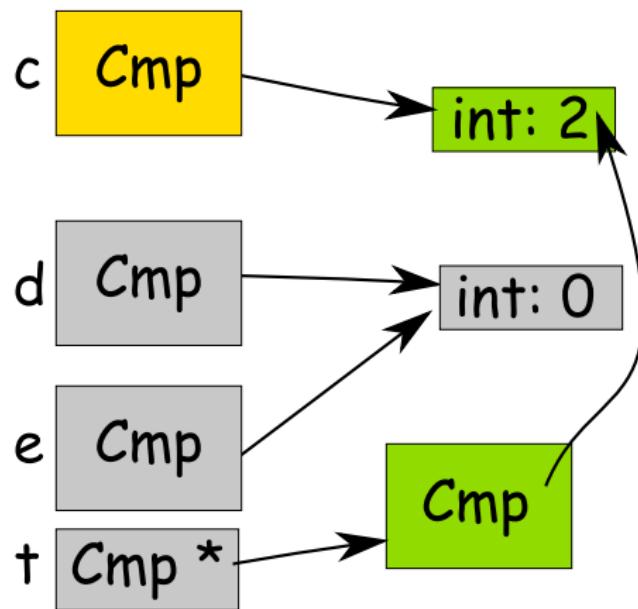
```
Cmp c;  
Cmp d;  
Cmp e=d;  
Cmp *t;  
t=new Cmp(c); d  
...  
}
```



Compteur: Exemple1

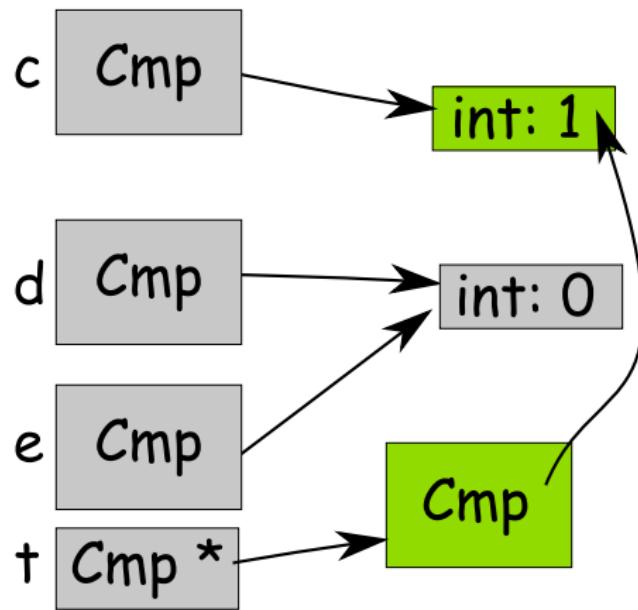
```
Cmp c;  
Cmp d;  
Cmp e=d;  
Cmp *t;  
t=new Cmp(c); d
```

```
...  
}
```



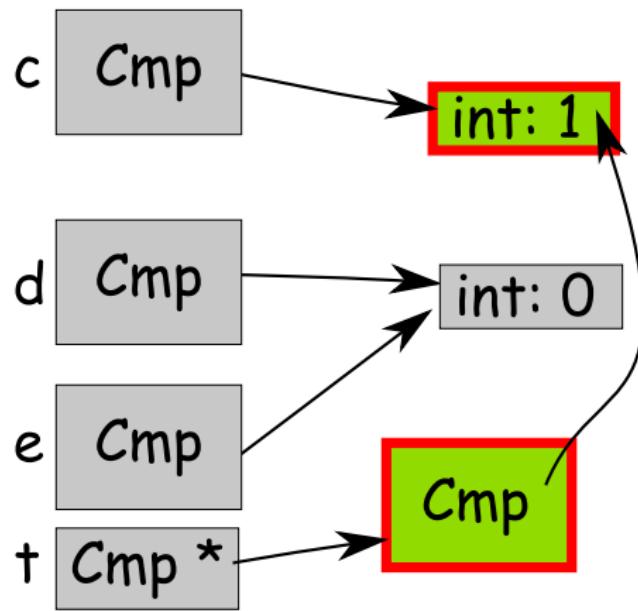
Compteur: Exemple1

```
Cmp c;  
Cmp d;  
Cmp e=d;  
Cmp *t;  
t=new Cmp(c); d  
...  
}
```



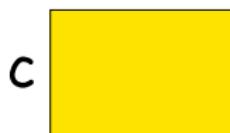
Compteur: Exemple1

```
Cmp c;  
Cmp d;  
Cmp e=d;  
Cmp *t;  
t=new Cmp(c); d  
...  
}
```



Compteur: Exemple2

```
Cmpt f(){  
Cmpt l;  
return l;  
}
```

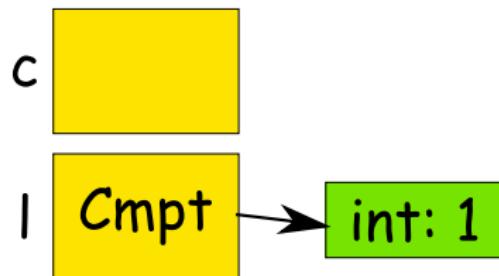


```
Cmpt c=f();
```



Compteur: Exemple2

```
Cmpt f(){  
Cmpt l;  
return l;  
}
```



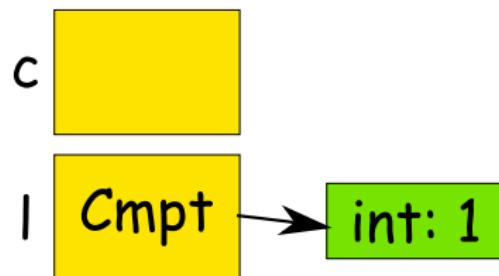
```
Cmpt c=f();
```

Compteur: Exemple2

```
Cmpt f(){  
Cmpt l;  
return l;  
}
```

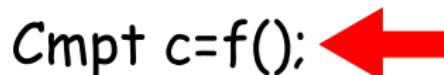


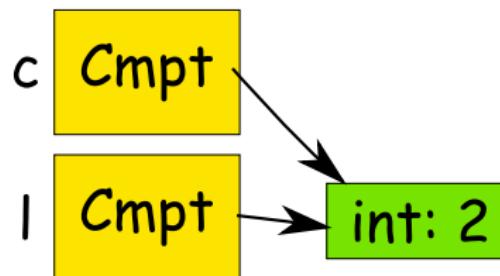
```
Cmpt c=f();
```



Compteur: Exemple2

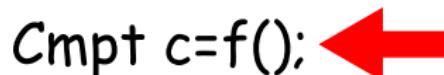
```
Cmpt f(){  
Cmpt l;  
return l;  
}
```

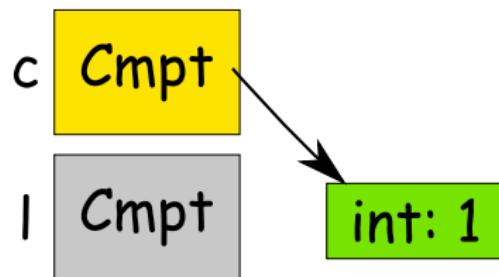
Cmpt c=f(); 



Compteur: Exemple2

```
Cmpt f(){  
Cmpt l;  
return l;  
}
```

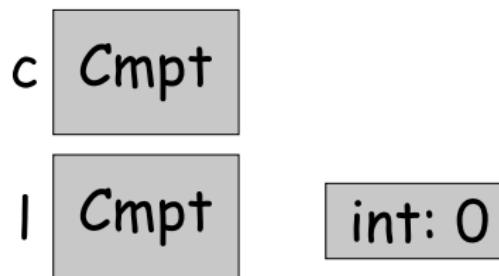
Cmpt c=f(); 



Compteur: Exemple2

```
Cmpt f(){  
Cmpt l;  
return l;  
}
```

Cmpt c=f();



La classe Compteur

La classe Compteur doit avoir les méthodes suivantes:

- **void** incremente() : **private**
- **void** decremente() : **private**
- **bool** dernier() : **public**

Ainsi que les constructeurs par défaut, recopie et destructeur.

- Compteur();
- Compteur(**const** Compteur &);
- ~Compteur();

implémentation de cette classe...

Problème:

Examinons le source suivant:

```
int main() {
    Compteur c;
    Compteur d;

    d=c; // Recopie de c dans d
}
```

Problème:

Examinons le source suivant:

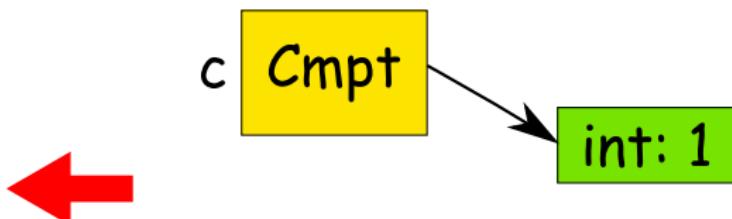
```
int main() {  
    Compteur c;  
    Compteur d;  
  
    d=c; // Recopie de c dans d  
}
```

La recopie se fait de la même façon que la construction par recopie. Le compilateur ajoute ici une recopie champs par champs!

Problème:

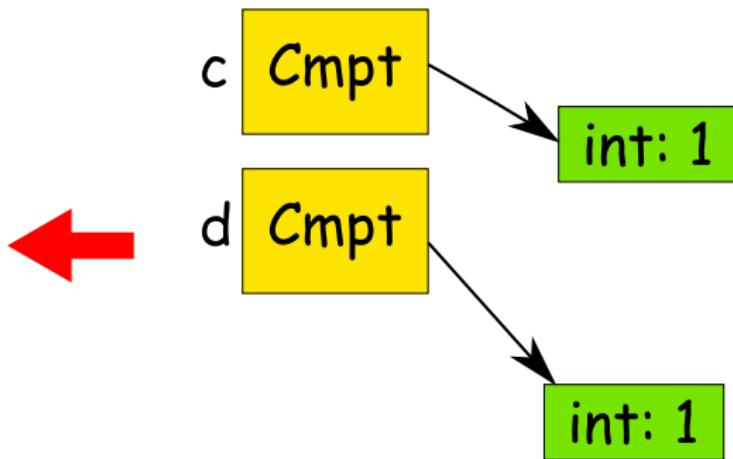
```
int main(){  
    Cmpt c;  
    Cmpt d;
```

```
}
```



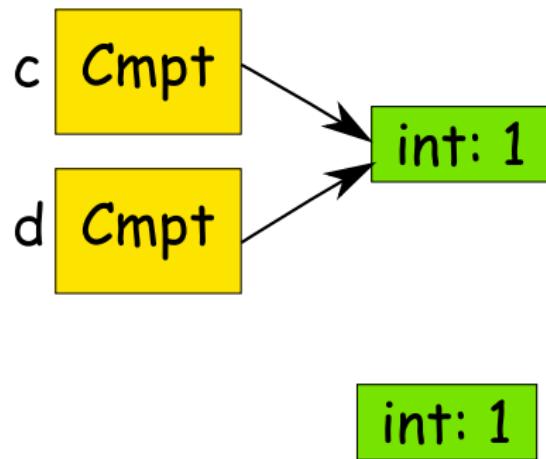
Problème:

```
int main(){  
    Cmpt c;  
    Cmpt d;  
}  
d=c;
```



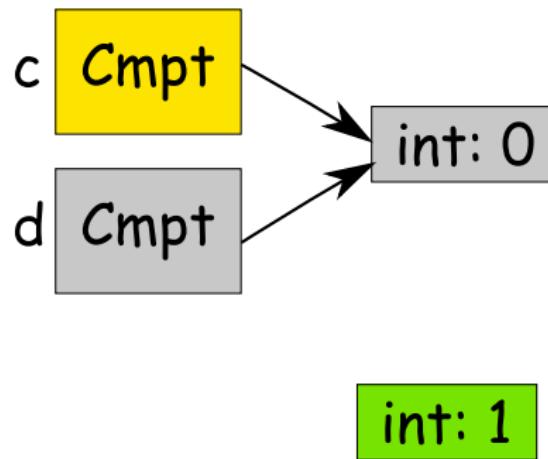
Problème:

```
int main(){  
    Cmpt c;  
    Cmpt d;  
}  
d=c;
```



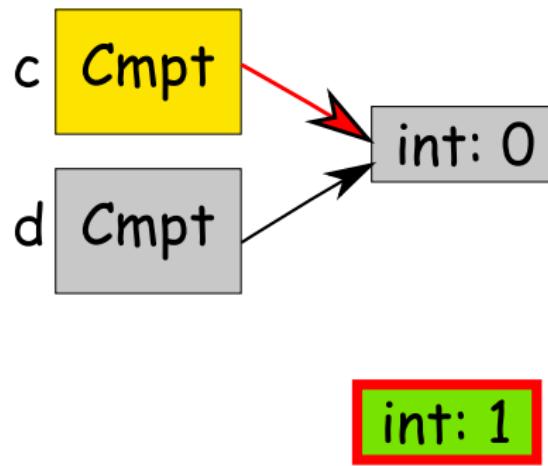
Problème:

```
int main(){  
    Cmpt c;  
    Cmpt d;  
  
    d=c;  
}
```



Problème:

```
int main(){  
    Cmpt c;  
    Cmpt d;  
  
    d=c;  
}
```



Plan

4 Operateurs

Opérateurs: définition

Un opérateur est une notation raccourcie permettant de représenter une opération fréquemment utilisée, par exemple:

Opérateurs: définition

Un opérateur est une notation raccourcie permettant de représenter une opération fréquemment utilisée, par exemple:

a+b*c;

Opérateurs: définition

Un opérateur est une notation raccourcie permettant de représenter une opération fréquemment utilisée, par exemple:

`a+b*c;`

On ajoute à `a` le résultat de la multiplication de `b` par `c`

Opérateurs: définition

Un opérateur est une notation raccourcie permettant de représenter une opération fréquemment utilisée, par exemple:

`a+b*c;`

On ajoute à `a` le résultat de la multiplication de `b` par `c`

Si `a` est un pointeur, le sens change: on ajoute à `a`, $b*c*\text{sizeof}(T)$ octets (si `T` est le type pointé par `a`).

Opérateurs: définition

Un opérateur est une notation raccourcie permettant de représenter une opération fréquemment utilisée, par exemple:

`a+b*c;`

On ajoute à `a` le résultat de la multiplication de `b` par `c`

Si `a` est un pointeur, le sens change: on ajoute à `a`, `b*c*sizeof(T)` octets (si `T` est le type pointé par `a`).

Dans le cas où `a` n'est pas un type primitif, cette notation est invalide sauf si l'on fournit la fonction permettant d'effectuer cette opération.

Opérateurs: liste

Voici la liste des opérateurs pouvant être définis en C++:

Unaire:

! & *

+ ++

- --

~

Opérateurs: liste

Voici la liste des opérateurs pouvant être définis en C++:

Unaire:

! & *
+ ++
- --
~

Binaire:

% %=
- -= + +=
/ /=
< <= > >=
!= == && ||
& &= | |=
^ ^= << <<=
>> >>=
-> ->*

Opérateurs: liste

Voici la liste des opérateurs pouvant être définis en C++:

Unaire:

- ! & *
- + ++
- --
- ~

spéciaux:

- new
- delete
- []
- ()
- conversion

Binaire:

%	%=	*	*=
-	-=	+	+=
/	/=		
<	<=	>	>=
!=	==	&&	
&	&=		=
^	^=	<<	<<=
>>	>>=		
		->	->*

Opérateurs: conditions

Pour pouvoir écrire un opérateur, au moins une des opérandes de cet opérateur doit correspondre à un type utilisateur (pas primitif). Par exemple, on ne peut pas modifier l'addition sur les entiers.

Opérateurs: conditions

Pour pouvoir écrire un opérateur, au moins une des opérandes de cet opérateur doit correspondre à un type utilisateur (pas primitif). Par exemple, on ne peut pas modifier l'addition sur les entiers.

Les opérateurs ne sont pas considérés comme commutatifs par le compilateur. C'est à dire que si l'on a écrit un opérateur permettant de faire l'addition d'un A avec un B,

B () +A () ;

ne sera pas résolu pas cet opérateur.

Opérateurs: conditions

Pour pouvoir écrire un opérateur, au moins une des opérandes de cet opérateur doit correspondre à un type utilisateur (pas primitif). Par exemple, on ne peut pas modifier l'addition sur les entiers.

Les opérateurs ne sont pas considérés comme commutatifs par le compilateur. C'est à dire que si l'on a écrit un opérateur permettant de faire l'addition d'un A avec un B,

B () +A () ;

ne sera pas résolu pas cet opérateur.

La précédence des opérateurs ne peut être modifiée.

Opérateurs: syntaxe

Un opérateur α s'écrit sous le nom d'une fonction ou méthode non statique de nom `operator α` .

Opérateurs: syntaxe

Un opérateur α s'écrit sous le nom d'une fonction ou méthode non statique de nom `operator α` .

Le type de retour de cette fonction est libre.

Opérateurs: syntaxe

Un opérateur α s'écrira sous le nom d'une fonction ou méthode non statique de nom `operator α` .

Le type de retour de cette fonction est libre.

C'est le premier paramètre qui est prioritaire pour la résolution. Dans le cas d'une méthode, le premier paramètre (correspondant à **this** dans la méthode) sera une référence du type de la classe.

Opérateurs: syntaxe

Un opérateur α s'écrira sous le nom d'une fonction ou méthode non statique de nom `operator α` .

Le type de retour de cette fonction est libre.

C'est le premier paramètre qui est prioritaire pour la résolution. Dans le cas d'une méthode, le premier paramètre (correspondant à **this** dans la méthode) sera une référence du type de la classe.

```
class A{
public:
    int operator+(int i);
};
```

Il faut voir cette méthode comme définissant l'opération + entre une instance de A non constante et un entier.

Opérateurs: syntaxe

On peut appeler les opérateurs comme des fonctions où des méthodes:
fonction:

```
class Dix{ };

int operator+(const Dix &d, int i) {
    return i+10;
}

int main() {
    Dix d;
    int douze=d+2;
    douze=operator+(d, 2);
    operator+(2, d); // Erreur!
}
```

Opérateurs: syntaxe

On peut appeler les opérateurs comme des fonctions où des méthodes:
méthode:

```
class Dix{
public:
    int operator+(int i) const{
        return i+10;
    }
};

int main() {
    Dix d;
    int douze=d+2;
    douze=d.operator+(2);
    2.operator+(d); // Erreur!
}
```

Opérateurs: précédence

La précédence des opérateurs ne peut être modifiée, l'évaluation se fait donc obligatoirement selon le respect des précédences standards.

exemple

```
class A{};  
class B{};  
  
A operator+(const A&, const B&);  
B operator*(const A&, const A&);  
  
int main() {  
    A a;  
    B b;  
    a+b*a; // Erreur !  
    (a+b)*a; // ok  
}
```

Opérateurs: unaires

On écrira les opérateurs unaires dans les classes:

```
class Entier{
    int v;
public:
    Entier operator-() const {
        return Entier(-v);
    }
};
```

Opérateurs: unaires

On écrira les opérateurs unaires dans les classes:

```
class Entier{
    int v;
public:
    Entier operator-() const {
        return Entier(-v);
    }
};
```

On écrira l'opérateur comme fonction si l'on n'a pas accès à la classe.

Opérateurs: binaires

On écrira les opérateurs binaires dans la classe si les deux opérandes sont de même type et que l'on a accès à la classe.

```
class Entier{
    int v;
public:
    Entier operator+(const Entier &e) const {
        return Entier(v+e.v);
    }
};
```

Opérateurs: binaires

Sinon on écrira l'opérateur à l'extérieur de la classe:

```
class Entier{
    int v;
    ...
};

Double operator+(const Entier &e, const Double &d) {
    return Double(e.value() + d.value());
}

Double operator+(const Double &d, const Entier &e) {
    return e+d;
}
```

Opérateurs: binaires

Sinon on écrira l'opérateur à l'extérieur de la classe:

```
class Entier{
    int v;
    ...
};

Double operator+(const Entier &e, const Double &d) {
    return Double(e.value() + d.value());
}

Double operator+(const Double &d, const Entier &e) {
    return e+d;
}
```

Eventuellement, on pourra déclarer l'opérateur comme fonction amie s'il est nécessaire d'accéder à des attributs privés (???).

Opérateurs: conception

Lorsque l'on écrit un nouvel opérateur, il faut faire attention à ce que l'introduction de ce "raccourci" n'apporte pas d'ambiguité: le but de l'opérateur est de simplifier, pas de compliquer.

Opérateurs: conception

Lorsque l'on écrit un nouvel opérateur, il faut faire attention à ce que l'introduction de ce "raccourci" n'apporte pas d'ambiguité: le but de l'opérateur est de simplifier, pas de compliquer.

En cas de doute, préférer toujours l'utilisation de fonctions ou méthodes classiques.

Opérateurs: conception

Lorsque l'on écrit un nouvel opérateur, il faut faire attention à ce que l'introduction de ce "raccourci" n'apporte pas d'ambiguité: le but de l'opérateur est de simplifier, pas de compliquer.

En cas de doute, préférer toujours l'utilisation de fonctions ou méthodes classiques.

Un exemple de mauvaise utilisation des opérateurs: la bibliothèque standard!

Opérateurs: iostream

La bibliothèque standard `iostream` fournit un modèle orienté-objet des entrées sorties, elle se décompose en

Opérateurs: iostream

La bibliothèque standard `iostream` fournit un modèle orienté-objet des entrées sorties, elle se décompose en

- `ios` : bases pour les entrées/sorties

Opérateurs: iostream

La bibliothèque standard `iostream` fournit un modèle orienté-objet des entrées sorties, elle se décompose en

- `ios` : bases pour les entrées/sorties
- `istream` : entrées

Opérateurs: iostream

La bibliothèque standard `iostream` fournit un modèle orienté-objet des entrées sorties, elle se décompose en

- `ios` : bases pour les entrées/sorties
- `istream` : entrées
- `ostream` : sorties

Opérateurs: iostream

La bibliothèque standard `iostream` fournit un modèle orienté-objet des entrées sorties, elle se décompose en

- `ios` : bases pour les entrées/sorties
- `istream` : entrées
- `ostream` : sorties
- `streambuf` : flux bufferisés

Opérateurs: iostream

La bibliothèque standard `iostream` fournit un modèle orienté-objet des entrées sorties, elle se décompose en

- `ios` : bases pour les entrées/sorties
- `istream` : entrées
- `ostream` : sorties
- `streambuf` : flux bufferisés
- `iostream` : entrées et sorties

Opérateurs: iostream

La bibliothèque standard `iostream` fournit un modèle orienté-objet des entrées sorties, elle se décompose en

- `ios` : bases pour les entrées/sorties
- `istream` : entrées
- `ostream` : sorties
- `streambuf` : flux bufferisés
- `iostream` : entrées et sorties
- `fstream` : fichiers

Opérateurs: iostream

La bibliothèque standard `iostream` fournit un modèle orienté-objet des entrées sorties, elle se décompose en

- `ios` : bases pour les entrées/sorties
- `istream` : entrées
- `ostream` : sorties
- `streambuf` : flux bufferisés
- `iostream` : entrées et sorties
- `fstream` : fichiers
- `sstream` : chaînes de caractères

Opérateurs: iostream

La bibliothèque standard `iostream` fournit un modèle orienté-objet des entrées sorties, elle se décompose en

- `ios` : bases pour les entrées/sorties
- `istream` : entrées
- `ostream` : sorties
- `streambuf` : flux bufferisés
- `iostream` : entrées et sorties
- `fstream` : fichiers
- `sstream` : chaînes de caractères

L'ensemble des classes font parties de l'espace de nom `std`

Opérateurs: iostream

La bibliothèque standard `iostream` fournit un modèle orienté-objet des entrées sorties, elle se décompose en

- `ios` : bases pour les entrées/sorties
- `istream` : entrées
- `ostream` : sorties
- `streambuf` : flux bufferisés
- `iostream` : entrées et sorties
- `fstream` : fichiers
- `sstream` : chaînes de caractères

L'ensemble des classes font parties de l'espace de nom `std`

Outre un ensemble de fonctionnalités permettant la gestion des flux, il a été ajouté un support pour la lecture et l'écriture basé sur les opérateurs

Opérateurs: iostream

L'opérateur << est utilisé pour écrire dans un flux de type std::ostream:

```
int main() {
    int i;
    std::cout<<"un entier:"<<i<<std::endl; // 1
    ...
    f<<1; // 2
    f<<i<<1; // 3
```

Opérateurs: iostream

L'opérateur `<<` est utilisé pour écrire dans un flux de type `std::ostream`:

```
int main() {
    int i;
    std::cout<<"un entier:"<<i<<std::endl; // 1
    ...
    f<<1; // 2
    f<<i<<1; // 3
```

La ligne 1 doit être interprétée comme ceci:

```
operator<<(
    operator<<(
        operator<<(std::cout, "un entier")
        , i)
    , std::endl)
```

Opérateurs: iostream

L'opérateur `<<` est utilisé pour écrire dans un flux de type `std::ostream`:

```
int main() {
    int i;
    std::cout<<"un entier:"<<i<<std::endl; // 1
    ...
    f<<1; // 2
    f<<i<<1; // 3
```

La ligne 1 doit être interprétée comme ceci:

```
operator<<(
    operator<<(
        operator<<(std::cout, "un entier")
        , i)
    , std::endl)
```

Qu'en déduire sur le type de retour de l'opérateur `<<` ?

Opérateurs: iostream

```
int main() {  
    int i;  
    std::cout<<"un entier :"<<i<<std::endl; // 1  
    ...  
    f<<1; // 2  
    f<<i<<1; // 3
```

Qu'en déduire sur le type de retour de l'opérateur << ?

Opérateurs: iostream

```
int main() {  
    int i;  
    std::cout<<"un entier :"<<i<<std::endl; // 1  
    ...  
    f<<1; // 2  
    f<<i<<1; // 3
```

Qu'en déduire sur le type de retour de l'opérateur << ?

le type de retour de l'opérateur << dans ce cas doit être compatible avec le premier argument.

Opérateurs: iostream

```
std::cout<<"un entier :"<<i<<std::endl; // 1  
f<<1; // 2  
f<<i<<1; // 3
```

Dans le cas 2, est-ce que l'on est en train de décaler un entier ou bien d'afficher 1 dans un flux ?

Dans le cas 3, est-ce:

- f décalé de i puis de 1 ?
- i décalé de 1 affiché dans f ?

Opérateurs: iostream

```
std::cout<<"un entier :"<<i<<std::endl; // 1  
f<<1; // 2  
f<<i<<1; // 3
```

Dans le cas 2, est-ce que l'on est en train de décaler un entier ou bien d'afficher 1 dans un flux ?

Dans le cas 3, est-ce:

- f décalé de i puis de 1 ?
- i décalé de 1 affiché dans f ?

⇒ d'une façon générale on essayera de ne pas “détourner” les opérateurs pour une utilisation qui n'est pas en rapport avec le sens premier de cette opérateur

Implanter le décalage à gauche pour une classe Entier qui multiplie l'entier par 2^i a du sens.

Opérateurs: iostream

La lecture depuis un flux se fait de la façon suivante

```
int i,j,k;  
char texte[10];  
std::cin>>i; // lecture d'un entier  
std::cin>>texte; // lecture d'une chaine !!  
std::cin>>i>>j>>k; // lecture de 3 entiers
```

Pour la lecture de chaîne de caractère on utilisera un type qui encapsule des mécanismes de réallocation telle que la classe std::string

Opérateurs: iostream

La lecture depuis un flux se fait de la façon suivante

```
int i,j,k;  
char texte[10];  
std::cin>>i; // lecture d'un entier  
std::cin>>texte; // lecture d'une chaine !!  
std::cin>>i>>j>>k; // lecture de 3 entiers
```

Pour la lecture de chaîne de caractère on utilisera un type qui encapsule des mécanismes de réallocation telle que la classe std::string

Quel est le type de retour de cet opérateur?

Opérateurs: iostream

La lecture depuis un flux se fait de la façon suivante

```
int i,j,k;  
char texte[10];  
std::cin>>i; // lecture d'un entier  
std::cin>>texte; // lecture d'une chaine !!  
std::cin>>i>>j>>k; // lecture de 3 entiers
```

Pour la lecture de chaîne de caractère on utilisera un type qui encapsule des mécanismes de réallocation telle que la classe std::string

Quel est le type de retour de cet opérateur?

Le type de retour doit compatible avec le premier argument:

```
std::istream &operator>>(std::istream &, ... )
```

Opérateurs: iostream

La lecture depuis un flux se fait de la façon suivante

```
int i,j,k;  
char texte[10];  
std::cin>>i; // lecture d'un entier  
std::cin>>texte; // lecture d'une chaine !!  
std::cin>>i>>j>>k; // lecture de 3 entiers
```

Pour la lecture de chaîne de caractère on utilisera un type qui encapsule des mécanismes de réallocation telle que la classe std::string

Quel est le type de retour de cet opérateur?

Le type de retour doit compatible avec le premier argument:

```
std::istream &operator>>(std::istream &, ... )
```

Pourquoi les références ne sont pas constantes ?

classe et iostream

Comment adapter votre classe pour pouvoir utiliser l'affichage de la bibliothèque standard:

```
class Entier{ int v; };
int main(){
    Entier e;
    std::cerr<<"debug"<<e;
}
```

classe et iostream

Comment adapter votre classe pour pouvoir utiliser l'affichage de la bibliothèque standard:

```
class Entier{ int v; };
int main(){
    Entier e;
    std::cerr<<"debug"<<e;
}
```

Erreur à la compilation:

```
A.cpp: In function 'int main()':
A.cpp:42: error: no match for 'operator<<' in
  'std::operator<< [with _Traits = std::char_traits<char>]
  ((std::basic_ostream<char, std::char_traits<char> >&) (&
  ((const char*)"debug")) << e'
```

suivi d'une bonne cinquantaine de lignes !

Solution

nous devons écrire l'opérateur << pour la classe std::ostream et Entier:

```
class Entier{  
    int v;  
};  
  
std::ostream &operator<<(std::ostream &stream, Entier e) {  
    stream<<e.v;  
    return stream;  
}
```

Combien d'erreurs dans ce code ?

Solution

nous devons écrire l'opérateur << pour la classe std::ostream et Entier:

```
class Entier{  
    int v;  
};  
  
std::ostream &operator<<(std::ostream &stream, Entier e) {  
    stream<<e.v;  
    return stream;  
}
```

Combien d'erreurs dans ce code ? Au moins 2:

- e doit être une référence constante
- e.v n'est pas accessible

Solution

nous devons écrire l'opérateur << pour la classe std::ostream et Entier:

```
class Entier{  
    int v;  
};
```

```
std::ostream &operator<<(std::ostream &stream, Entier e) {  
    stream<<e.v;  
    return stream;  
}
```

Combien d'erreurs dans ce code ? Au moins 2:

- e doit être une référence constante
- e.v n'est pas accessible

solutions:

- Ajout d'un accesseur dans le code.
- Déclarer l'opérateur comme amie de la classe

Solution

```
class Entier{
    int v;
public:
    int value() const { return v; }
};

std::ostream &operator<<(std::ostream &stream,
                           const Entier &e) {
    stream<<e.value();
    return stream;
}
```

Solution

```
class Entier{
    int v;
public:
    friend std::ostream &operator<<(
        std::ostream &stream,
        const Entier &e);
};

std::ostream &operator<<(std::ostream &stream,
                           const Entier &e) {
    stream<<e.v;
    return stream;
}
```

Solution

```
class Entier{
    int v;
public:
    friend std::ostream &operator<<(
        std::ostream &stream,
        const Entier &e);
};

std::ostream &operator<<(std::ostream &stream,
                           const Entier &e) {
    stream<<e.v;
    return stream;
}
```

Si l'implémentation change (l'attribut privé est renommé), on doit aussi changer le code de l'opérateur :(

Solution

```
class Entier{
    int v;
public:
    friend std::ostream &operator<<(
        std::ostream &stream,
        const Entier &e);
};

std::ostream &operator<<(std::ostream &stream,
                           const Entier &e) {
    stream<<e.v;
    return stream;
}
```

Si l'implémentation change (l'attribut privé est renommé), on doit aussi changer le code de l'opérateur :(

⇒ les fonctions amies ne doivent pas être utilisées à la place des accesseurs!

Opérateur: d'affectation

L'opérateur d'affectation correspond au '=' et doit nécessairement être écrit sous la forme d'un membre de classe:

```
class Entier{
    int v;
public:
    const Entier &operator=(int i) { v=i ; }
};

...
Entier e;
e=1;
Entier f=1; // Erreur !
```

Opérateur: d'affectation

Le compilateur fournit toujours un opérateur d'affectation permettant de copier une instance dans une instance de même type:

```
struct _S a,b;  
a=b;
```

L'opérateur d'affectation fournit est de la forme:

```
X &X::operator=( const X&); // 1  
X &X::operator=(X&); // 2
```

Dans les deux cas, la copie est effectuée attribut par attribut dans l'ordre de leur déclaration.

La forme 1 est utilisée si chaque attribut possède un opérateur d'affectation en forme 1.

Si l'on déclare explicitement un opérateur d'affectation en forme 1 ou 2 alors le compilateur n'ajoute plus l'opérateur.

Si la classe possède un attribut constant ou de type référence, le compilateur ne peut pas définir l'opérateur d'affectation.

Compteur

Revenons sur notre exemple de classe Compteur et ajoutons un opérateur d'affectation...

À partir du moment où une classe effectue de l'allocation dynamique, on écrira systématiquement un constructeur par recopie, un destructeur et un opérateur d'affectation

Opérateur: foncteur

L'opérateur de fonction () est le seul opérateur d'instance (i.e. à part new et delete) qui peut prendre un nombre variable d'arguments. Pour les autres, le nombre d'arguments est fixé par l'arité de l'opérateur.

Opérateur: foncteur

L'opérateur de fonction () est le seul opérateur d'instance (i.e. à part new et delete) qui peut prendre un nombre variable d'arguments. Pour les autres, le nombre d'arguments est fixé par l'arité de l'opérateur.

Une classe qui implémente cet opérateur est appelée foncteur (ou fonctor) car l'on peut alors utiliser une instance comme une fonction:

Opérateur: foncteur

L'opérateur de fonction () est le seul opérateur d'instance (i.e. à part new et delete) qui peut prendre un nombre variable d'arguments. Pour les autres, le nombre d'arguments est fixé par l'arité de l'opérateur.

Une classe qui implémente cet opérateur est appelée foncteur (ou fonctor) car l'on peut alors utiliser une instance comme une fonction:

```
class Greater{  
public:  
    bool operator () (int a, int b){ return a>b; }  
    bool operator () (double a, double b) {  
        return a>b;  
    }  
    bool operator () (const char *a, const char *b) {  
        return strcmp(a,b)>0;  
    }  
};
```

Opérateur: foncteur

```
class Greater{
public:
    bool operator()(int a, int b);
    bool operator()(double a, double b);
    bool operator()(const char *a, const char *b);
};

...
Greater greater;
if (greater(1,0)){ ... }
if (greater("aba","abb")){... }
```

Opérateur: foncteur

```
class Greater{  
public:  
    bool operator()(int a, int b);  
    bool operator()(double a, double b);  
    bool operator()(const char *a, const char *b);  
};  
...  
Greater greater;  
if (greater(1,0)){ ... }  
if (greater("aba","abb")){... }
```

On aurait aussi bien pu écrire plusieurs fonctions greater en utilisant la surcharge.

Opérateur: foncteur

```
class Greater{
public:
    bool operator()(int a, int b);
    bool operator()(double a, double b);
    bool operator()(const char *a, const char *b);
};

...
Greater greater;
if (greater(1,0)){ ... }
if (greater("aba","abb")){... }
```

On aurait aussi bien pu écrire plusieurs fonctions greater en utilisant la surcharge.

Un autre exemple plus utile (et complexe) est l'implémentation d'une fonction de hachage paramétrable (attributs).

Opérateur: de conversion

Il est possible d'écrire des opérateurs prenant en charge la conversion d'une instance vers un autre type:

```
class Entier{
    int _value;
public:
operator int () const {
    return _value;
}
};
```

Opérateur: de conversion

Il est possible d'écrire des opérateurs prenant en charge la conversion d'une instance vers un autre type:

```
class Entier{
    int _value;
public:
operator int () const {
    return _value;
}
};
```

Le prototype de l'opérateur de conversion est **operator type() const**, ne possède pas de type de retour (puisque c'est type) et ne prend pas d'argument.

Opérateur: de conversion

Il est possible d'écrire des opérateurs prenant en charge la conversion d'une instance vers un autre type:

```
class Entier{
    int _value;
public:
operator int () const {
    return _value;
}
};
```

Le prototype de l'opérateur de conversion est **operator type() const**, ne possède pas de type de retour (puisque c'est type) et ne prend pas d'argument.

L'opérateur de conversion s'écrit obligatoirement sous la forme d'une fonction membre.

Opérateurs: de conversion

L'opérateur de conversion peut être appelé explicitement avec un cast ou bien implicitement:

```
void f(int);  
...  
Entier e;  
int i=e; // conversion implicite  
(int)e; // conversion explicite  
f(e); // conversion implicite
```

Les opérateurs de conversions sont pris en compte par l'algorithme de résolution.

Opérateur: de conversion

Une utilisation intéressante de l'opérateur de conversion est la conversion en std::string:

```
#include <string>
class Entier{
public:
    operator int() const{ return _value; }
    operator std::string () const{
        std::stringstream s;
        s<<_value;
        return s.str();
    }
};

...
Entier e;
cout<<(std::string )e;
```

Cette approche est plus logique (que l'implémentation de <<) et permet plus de possibilités.

Opérateurs: autres

L'opérateur [] est souvent utilisé. Il prend un seul argument et doit obligatoirement être écrit comme fonction membre.

Opérateurs: autres

L'opérateur `[]` est souvent utilisé. Il prend un seul argument et doit obligatoirement être écrit comme fonction membre.

L'opérateur **new** est utilisé pour faire du placement d'objet en mémoire. Il permet d'écrire ces propres allocateurs ou encore de faire du “debug-age”. Le nombre d'argument est variable mais le premier est obligatoirement de type `size_t`, le type de retour est obligatoirement **void***.

Opérateur: new

Il existe 2 versions de l'opérateur new:

```
void *operator new(size_t) throw(bad_alloc);  
void *operator new(size_t, const nothrow_t &) throw();
```

Opérateur: new

Il existe 2 versions de l'opérateur new:

```
void *operator new(size_t) throw(bad_alloc);  
void *operator new(size_t, const nothrow_t &) throw();
```

2 versions de l'opérateur new[]:

```
void *operator new[](size_t) throw(bad_alloc);  
void *operator new[](size_t, const nothrow_t &) throw();
```

Opérateur: new

Il existe 2 versions de l'opérateur new:

```
void *operator new(size_t) throw(bad_alloc);  
void *operator new(size_t, constnothrow_t &) throw();
```

2 versions de l'opérateur new[]:

```
void *operator new[](size_t) throw(bad_alloc);  
void *operator new[](size_t, constnothrow_t &) throw();
```

La première version lève une exception dans le cas où l'allocation demandée pose problème (plus assez de mémoire).

Opérateur: new

Il existe 2 versions de l'opérateur new:

```
void *operator new(size_t) throw(bad_alloc);  
void *operator new(size_t, constnothrow_t &) throw();
```

2 versions de l'opérateur new[]:

```
void *operator new[](size_t) throw(bad_alloc);  
void *operator new[](size_t, constnothrow_t &) throw();
```

La première version lève une exception dans le cas où l'allocation demandée pose problème (plus assez de mémoire).

La deuxième garantie de ne pas lever d'exception et renvoie 0 en cas de problème.

Opérateur: new

Il existe 2 versions de l'opérateur new:

```
void *operator new(size_t) throw(bad_alloc);  
void *operator new(size_t, constnothrow_t &) throw();
```

2 versions de l'opérateur new[]:

```
void *operator new[](size_t) throw(bad_alloc);  
void *operator new[](size_t, constnothrow_t &) throw();
```

La première version lève une exception dans le cas où l'allocation demandée pose problème (plus assez de mémoire).

La deuxième garantie de ne pas lever d'exception et renvoie 0 en cas de problème.

```
int *i=new int;  
int *j=new (nothrow) int;
```

Opérateur: new

Ces versions de l'opérateur new sont redefinissable:

```
void *operator new(size_t s){  
    printf("allocation_de_%d_octets");  
    return new(s,nothrow); // version sans exception  
}  
...  
int *i=new int; // affiche: allocation de 4 octets
```

Ici, on utilise la version sans exception de l'opérateur new fourni par la bibliothèque standard pour effectuer l'allocation mémoire.

Opérateur: new

Ces versions de l'opérateur new sont redefinissable:

```
void *operator new(size_t s){  
    printf("allocation_de_%d_octets");  
    return new(s,nothrow); // version sans exception  
}  
...  
int *i=new int; // affiche: allocation de 4 octets
```

Ici, on utilise la version sans exception de l'opérateur new fourni par la bibliothèque standard pour effectuer l'allocation mémoire.

Il faut noter que toutes les allocations dynamiques passeront par notre fonction. Peut-être utile pour du debug...

Opérateur: new

Il existe aussi une version de placement de l'opérateur new

```
void *operator new(size_t, void *) throw();  
void *operator new[](size_t, void *) throw();
```

Ces versions, dites de placement, ne sont pas redéfinissables et retourne l'adresse passée en argument. Elles permettent de créer une instance à un emplacement mémoire précis:

Opérateur: new

Il existe aussi une version de placement de l'opérateur new

```
void *operator new(size_t, void *) throw();  
void *operator new[](size_t, void *) throw();
```

Ces versions, dites de placement, ne sont pas redéfinissables et retourne l'adresse passée en argument. Elles permettent de créer une instance à un emplacement mémoire précis:

```
char data[1000];  
int *i=new(data) int;
```

Opérateur: new

On peut aussi créer son propre allocateur, pour cela il faut écrire une fonction
void *operator new (size_t, suivi d'un nombre quelconque d'arguments.

Opérateur: new

On peut aussi créer son propre allocateur, pour cela il faut écrire une fonction
void *operator new (size_t, suivi d'un nombre quelconque d'arguments.

Cette fonction ne peut pas faire partie d'un espace de nom

Opérateur: new

On peut aussi créer son propre allocateur, pour cela il faut écrire une fonction
void *operator new (size_t, suivi d'un nombre quelconque d'arguments.

Cette fonction ne peut pas faire partie d'un espace de nom

```
class IntTab{
    int *data;
    int current,max;
public:
    IntTab(int max):max(max),current(0),data(new int [max]) {}
    int *next () { return data+current++; }
};

void *operator new (size_t t,IntTab &ia) {
    return ia.next();
}
```

Opérateur: new

Utilisation:

Sans

```
for (i=0;i<100000000;i++)
    new int;
```

Avec

```
IntTab alloc(100000000);
for (i=0;i<100000000;i++)
    new (alloc) int;
```

performances:

- sans: 1,5Gb en 4,45s
- avec: 384Mb en 1,25s

il faut aussi ajouter l'opérateur **void delete (void *, IntTab &)**.

Plan

5 Héritage, polymorphisme

Héritage: définition

L'héritage est un mécanisme permettant de construire un type T à partir d'un autre type Base . Le type T se retrouve doté des comportements (méthodes) et propriétés (attributs) du type Base . Une relation de typage relie ces deux types: T est un sous-type de Base (ou classe dérivée de Base) tandis que Base est un super-type de T .

Héritage: définition

L'héritage est un mécanisme permettant de construire un type T à partir d'un autre type Base . Le type T se retrouve doté des comportements (méthodes) et propriétés (attributs) du type Base . Une relation de typage relie ces deux types: T est un sous-type de Base (ou classe dérivée de Base) tandis que Base est un super-type de T .

En C++, les membres hérités et la relation de typage qu'il existe entre T et Base ne sont pas nécessairement visible de l'extérieur.

Héritage: définition

L'héritage est un mécanisme permettant de construire un type *T* à partir d'un autre type *Base*. Le type *T* se retrouve doté des comportements (méthodes) et propriétés (attributs) du type de *Base*. Une relation de typage relie ces deux types: *T* est un sous-type de *Base* (ou classe dérivée de *Base*) tandis que *Base* est un super-type de *T*.

En C++, les membres hérités et la relation de typage qu'il existe entre *T* et *Base* ne sont pas nécessairement visible de l'extérieur.

La relation est indiquée lors de la déclaration de *T*:

```
class T: public Base{ ...}
```

Nous reviendrons sur le sens du **public** plus tard.

Héritage: exemple

Voici un exemple classique d'héritage:

```
class Tuyau{
    double _diametre;
public:
    double debit() const;
    double diametre() const;
};
```

```
class TuyauPer : public Tuyau{
    double _densite;
public:
    double densite() const;
};
```

Héritage: exemple

Voici un exemple classique d'héritage:

```
class Tuyau{  
    double _diametre;  
public:  
    double debit() const;  
    double diametre() const;  
};
```

```
class TuyauPer : public Tuyau{  
    double _densite;  
public:  
    double densite() const;  
};
```

```
TuyauPer tper;  
Tuyau &tuyau=tper; // relation de typage  
tper.densite(); // ok  
tuyau.densite(); // erreur!  
tper.debit(); //ok
```

Héritage: pointeurs et référence

L'utilisation de la relation de typage ne peut se faire qu'avec les pointeurs ou les références.

Héritage: pointeurs et référence

L'utilisation de la relation de typage ne peut se faire qu'avec les pointeurs ou les références.

En effet, pour faire jouer la relation de sous-typage on souhaite manipuler une même instance à travers différentes variables de types différents.

Héritage: pointeurs et référence

L'utilisation de la relation de typage ne peut se faire qu'avec les pointeurs ou les références.

En effet, pour faire jouer la relation de sous-typage on souhaite manipuler une même instance à travers différentes variables de types différents.

```
TuyauPer tper;
Tuyau t=tper; // Erreur !
Tuyau *p=&tper; // ok
Tuyau &r=tper; // ok
TuyauPer &r2=r; // Erreur !
```

Héritage: appel de méthodes

En C++ l'appel de méthode doit être vu comme un appel de fonction avec passage d'un premier paramètre caché (**this** dans la méthode).

Héritage: appel de méthodes

En C++ l'appel de méthode doit être vu comme un appel de fonction avec passage d'un premier paramètre caché (**this** dans la méthode).

Lors d'un appel, la méthode choisie est celle du type de la variable. Si la méthode n'est pas présente directement dans le type, c'est la méthode compatible de son parent le plus proche qui est choisie.

Héritage: appel de méthodes

En C++ l'appel de méthode doit être vu comme un appel de fonction avec passage d'un premier paramètre caché (**this** dans la méthode).

Lors d'un appel, la méthode choisie est celle du type de la variable. Si la méthode n'est pas présente directement dans le type, c'est la méthode compatible de son parent le plus proche qui est choisie.

```
class A{
public:
    void f(char) { puts("A\n"); }
};

class B:public A{
public:
    void f(int) { puts("B\n"); }
};

class C:public B{};
```

```
C c;
c.f(1); // => B
c.f('a'); // => B
c.A::f('A'); // => A
c.B::f('a'); // => B
B &b=c;
b.f('a'); // => B
```

On utilise l'opérateur de portée pour spécifier la méthode à appeler.

Héritage: appel

```
class A{  
public:  
    void m() {}  
};
```

```
B b;  
A a;  
A &r=b;  
b.m(); // b est de type B => B::m()  
a.m(); // a est de type A => A::m()  
r.m(); // r est de type A => A::m()  
A *p=&b;  
p->m(); // p est de type A* => A::m()
```

```
class B : public A{  
public:  
    void m() {}  
};
```

Héritage: et mangling

Le “mangling” désigne la façon de générer un nom de symbole à partir d'une fonction ou méthode:

```
$g++ -c /tmp/mang.c  
$nm mang.o  
U __gxx_personality_v0  
00000000 T _ZN1A1mEv  
00000006 T _ZN1B1mEv
```

La première fonction correspond à `A::m()`, la deuxième à `B::m()`

Héritage: et mangling

Le “mangling” désigne la façon de générer un nom de symbole à partir d’une fonction ou méthode:

```
$g++ -c /tmp/mang.c  
$nm mang.o  
U __gxx_personality_v0  
00000000 T _ZN1A1mEv  
00000006 T _ZN1B1mEv
```

La première fonction correspond à `A::m()`, la deuxième à `B::m()`

Il faut savoir que cette étape d’encodage n’est pas normalisée et est propre à chaque compilateur. Ainsi, les objets compilés avec un compilateur ne seront sans doute pas exploitable avec un autre compilateur!

Héritage: et mangling

Le “mangling” désigne la façon de générer un nom de symbole à partir d’une fonction ou méthode:

```
$g++ -c /tmp/mang.c  
$nm mang.o  
U __gxx_personality_v0  
00000000 T _ZN1A1mEv  
00000006 T _ZN1B1mEv
```

La première fonction correspond à `A::m()`, la deuxième à `B::m()`

Il faut savoir que cette étape d’encodage n’est pas normalisée et est propre à chaque compilateur. Ainsi, les objets compilés avec un compilateur ne seront sans doute pas exploitable avec un autre compilateur!

En C, il n’y a pas de “mangling”, une fonction génère un symbole de même nom que cette fonction.

Héritage: et mangling

Le “mangling” désigne la façon de générer un nom de symbole à partir d’une fonction ou méthode:

```
$g++ -c /tmp/mang.c  
$nm mang.o  
U __gxx_personality_v0  
00000000 T _ZN1A1mEv  
00000006 T _ZN1B1mEv
```

La première fonction correspond à `A::m()`, la deuxième à `B::m()`

Il faut savoir que cette étape d’encodage n’est pas normalisée et est propre à chaque compilateur. Ainsi, les objets compilés avec un compilateur ne seront sans doute pas exploitable avec un autre compilateur!

En C, il n’y a pas de “mangling”, une fonction génère un symbole de même nom que cette fonction.

Le “mangling” permet la mise en oeuvre de la surcharge, des espaces de nom, des méthodes de classes...

Héritage: et constructeur

Lors de la construction d'une instance de la classe `Fille`, sa classe de base, la classe `Mere` doit elle aussi être initialisée.

Pour initialiser la partie `Mere`, le compilateur ajoute un appel au constructeur par défaut:

```
class Mere{  
public:  
    Mere() {puts("Mere");}  
};  
class Fille: public Mere{  
public:  
    Fille() {puts("Fille");}  
};
```

```
Fille f;  
/*  
Affiche :  
  
Mere  
Fille  
*/
```

Héritage: et constructeur

Lors de la construction d'une instance de la classe `Fille`, sa classe de base, la classe `Mere` doit elle aussi être initialisée.

Pour initialiser la partie `Mere`, le compilateur ajoute un appel au constructeur par défaut:

```
class Mere{
public:
    Mere() {puts("Mere"); }
};

class Fille: public Mere{
public:
    Fille() {puts("Fille"); }
};
```

Fille f;
/*
Affiche :

Mere
Fille
*/

Comment faire si la classe `Mere` n'a pas de constructeur par défaut?

Héritage: et constructeur

On indique le constructeur à appeler comme ceci:

```
class Mere{  
    public :  
        Mere(int i) {puts("Mere");}  
};  
  
class Fille: public Mere{  
    public :  
        Fille(): Mere(1)  
        {puts("Fille");}  
};
```

Fille f;
/*
Affiche :

Mere
Fille
*/

L'appel au constructeur de la classe `Mere` doit se faire avant l'initialisation des attributs de la classe.

Dans le cas contraire le compilateur inverse les initialisations.

Héritage: et constructeur

```
class Mere{  
public:  
Mere(int i){  
    printf("Mere: %d\n", i);  
}  
};  
  
class Fille: public Mere{  
int v;  
public:  
Fille():v(1),Mere(v)  
{puts("Fille");}  
};
```

Fille f;
/*
Affiche :

Mere: -1479451508
Fille
*/

Si l'on compile avec `-Wall`, le compilateur nous signal l'inversion, sinon rien n'est dit.

Héritage: et destructeur

De même que pour le constructeur, les appels au destructeur se font en “cascade”. C'est à dire qu'à la fin du destructeur de la classe `Fille`, un appel au destructeur de la classe `Mere` est ajouté:

```
class Mere{
public:
Mere() {puts ("Mere"); }
~Mere() {puts ("~Mere"); }
};

class Fille: public Mere{
public:
Fille() {puts ("Fille"); }
~Fille() {puts ("~Fille"); }
};
```

```
{
Fille f;
}
/*
Affiche :
Mere
Fille
~Fille
~Mere
*/
```

Héritage: et opérateur d'affectation

Nous avons vu que le compilateur ajoute dans les classes un opérateur d'affectation si celui-ci n'est pas écrit.

Héritage: et opérateur d'affectation

Nous avons vu que le compilateur ajoute dans les classes un opérateur d'affectation si celui-ci n'est pas écrit.

L'opérateur ajouté par le compilateur fait appel à l'opérateur de la classe Mere:

```

class Mere{
    int _v;
    public:
        Mere(int v=0) :_v(v) {}
        const Mere & operator=(const Mere &) {
            printf("Mere:%d\n", _v);
            return *this;
        }
    };
    class Fille: public Mere{
        Mere m;
        public:
            Fille(): Mere(10) {}
    };

```

Mere:10
Mere:0
*/**

Héritage: et opérateur d'affectation

Voici à quoi ressemble l'opérateur d'affectation ajouté par le compilateur:

```
class Fille: public Mere{  
    ...  
public:  
    const Fille &operator=(const Fille &f) {  
        Mere::operator=(f);  
        this->attribut1=f.attribut1;  
        this->attribut2=f.attribut2;  
        ...  
    }  
};
```

Dans le cas où nous voulons écrire notre propre opérateur, c'est à nous d'appeler explicitement l'opérateur d'affectation de la classe `Mere`.

Héritage: et opérateur d'affectation

```
class A{
public:
const A& operator=(const A&) {
    puts("A=A");
    return *this;
}
class M{
    A a;
};
class F: public M{
public:
const F& operator=(const F&f) {
    return *this;
}
};
```

F f, g;
f=g;
// Pas d'affichage !

Héritage: et opérateur d'affectation

```
class A{
public:
const A& operator=(const A&) {
    puts("A=A");
    return *this;
};

class M{
    A a;
};

class F: public M{
public:
const F& operator=(const F&f) {
    M::operator=(f);
    return *this;
};
};
```

F f, g;
f=g;
// Affiche A=A

Héritage: et opérateur d'affectation

```
class A{
public:
const A& operator=(const A&) {
    puts("A=A");
    return *this;
};

class M{
    A a;
};

class F: public M{
public:
const F& operator=(const F&f) {
    M::operator=(f);
    return *this;
};
};
```

F f, g;
f=g;
// Affiche A=A

On peut faire appel à l'opérateur ajouté par le compilateur!

Héritage: public, protected et private

Il existe trois possibilités pour l'héritage, public: **class T: public Base**,
protected: **class T: protected Base** et private: **class T: private Base**.
L'héritage est privé par défaut dans les classes et publique pour les structures.

Héritage: public, protected et private

Il existe trois possibilités pour l'héritage, public: **class T: public Base**, protected: **class T: protected Base** et private: **class T: private Base**. L'héritage est privé par défaut dans les classes et publique pour les structures.

Dans le cas de l'héritage **public**, la relation entre T et Base est visible de tous. Les méthodes **public** de Base sont **public** dans T et les méthodes **protected** sont **protected** dans T.

Héritage: public, protected et private

Il existe trois possibilités pour l'héritage, public: **class T: public Base**, protected: **class T: protected Base** et private: **class T: private Base**. L'héritage est privé par défaut dans les classes et publique pour les structures.

Dans le cas de l'héritage **public**, la relation entre T et Base est visible de tous. Les méthodes **public** de Base sont **public** dans T et les méthodes **protected** sont **protected** dans T.

Dans le cas de l'héritage **protected**, les méthodes **public** et **protected** de Base sont **protected** dans T.

Héritage: public, protected et private

Il existe trois possibilités pour l'héritage, public: **class T: public Base**, protected: **class T: protected Base** et private: **class T: private Base**. L'héritage est privé par défaut dans les classes et publique pour les structures.

Dans le cas de l'héritage **public**, la relation entre T et Base est visible de tous. Les méthodes **public** de Base sont **public** dans T et les méthodes **protected** sont **protected** dans T.

Dans le cas de l'héritage **protected**, les méthodes **public** et **protected** de Base sont **protected** dans T.

Dans le cas de l'héritage **private**, les méthodes **public** et **protected** de Base sont **private** dans T.

Héritage: public, protected et private

On ne peut utiliser un relation type/sous-type que si l'on peut avoir accès à un membre public de la classe Base à travers T.

Héritage: public, protected et private

On ne peut utiliser un relation type/sous-type que si l'on peut avoir accès à un membre public de la classe Base à travers T.

Si l'héritage est public, tout le monde voit les méthodes publiques de Base à travers T. Ainsi on pourra toujours voir un T comme un Base.

Héritage: public, protected et private

On ne peut utiliser un relation type/sous-type que si l'on peut avoir accès à un membre public de la classe Base à travers T.

Si l'héritage est public, tout le monde voit les méthodes publiques de Base à travers T. Ainsi on pourra toujours voir un T comme un Base.

Si l'héritage est protégé alors seules les fonctions/classes amies, sous-classes et amies des sous-classes pourront voir un T comme un Base.

Héritage: public, protected et private

On ne peut utiliser un relation type/sous-type que si l'on peut avoir accès à un membre public de la classe Base à travers T.

Si l'héritage est publique, tout le monde voit les méthodes publiques de Base à travers T. Ainsi on pourra toujours voir un T comme un Base.

Si l'héritage est protégé alors seules les fonctions/classes amies, sous-classes et amies des sous-classes pourront voir un T comme un Base.

Si l'héritage est privé alors seules les fonctions/classes amies pourront voir un T comme un Base.

Héritage: public, protected et private

```

class A{
    ...
};

class B: public A{
    ...
};

class C: protected A{
    ...
};

class D: public C{
    ...
};

class E: private A{
    ...
};

class F: public E{
    ...
};

        B b;
        A&a=b; // ok

        C c;
        A *p=&c; // erreur!

void D::f() {
    A &r=*this; // ok
}

void F::f() {
    A &r=*this; // erreur
}

void E::f() {
    A &r=*this; // ok
}

```

Héritage: public, protected et private

L'héritage **protected** ou **private** permet de récupérer une implémentation.

Héritage: public, protected et private

L'héritage **protected** ou **private** permet de récupérer une implémentation.

C'est une alternative efficace à la délégation (lien a-un).

Héritage: public, protected et private

L'héritage **protected** ou **private** permet de récupérer une implémentation.

C'est une alternative efficace à la délégation (lien a-un).

Un exemple: on veut implémenter une classe `Pile`. Pour cela on souhaite utiliser la classe `Vector`.

Or, une pile n'est pas un vecteur particulier, on ne souhaite donc pas que la classe `Pile` présente toutes les méthodes de la classe `Vector`.

Héritage: public, protected et private

L'héritage **protected** ou **private** permet de récupérer une implémentation.

C'est une alternative efficace à la délégation (lien a-un).

Un exemple: on veut implémenter une classe `Pile`. Pour cela on souhaite utiliser la classe `Vector`.

Or, une pile n'est pas un vecteur particulier, on ne souhaite donc pas que la classe `Pile` présente toutes les méthodes de la classe `Vector`.

La solution sans héritage privé consiste à utiliser la délégation: on dispose d'un attribut de type `Vector` que l'on utilise pour implanter la pile.

Héritage: public, protected et private

L'héritage **protected** ou **private** permet de récupérer une implémentation.

C'est une alternative efficace à la délégation (lien a-un).

Un exemple: on veut implémenter une classe `Pile`. Pour cela on souhaite utiliser la classe `Vector`.

Or, une pile n'est pas un vecteur particulier, on ne souhaite donc pas que la classe `Pile` présente toutes les méthodes de la classe `Vector`.

La solution sans héritage privé consiste à utiliser la délégation: on dispose d'un attribut de type `Vector` que l'on utilise pour implanter la pile.

Si l'on souhaite que cette "délégation" soit accessible aux sous classes alors on utilise l'héritage protégé sinon on utilise l'héritage privé.

Héritage: et polymorphisme

Jusqu'à présent, l'héritage permet le factorisation de code mais ne permet pas de mettre en oeuvre le polymorphisme.

En effet, pour qu'il y ai polymorphisme il faut que la méthode appelée sur une instance soit celle correspondant au type réel de l'instance et non au type de la variable manipulant cette instance.

Héritage: et polymorphisme

Jusqu'à présent, l'héritage permet le factorisation de code mais ne permet pas de mettre en oeuvre le polymorphisme.

En effet, pour qu'il y ai polymorphisme il faut que la méthode appelée sur une instance soit celle correspondant au type réel de l'instance et non au type de la variable manipulant cette instance.

Pour mettre en oeuvre le polymorphisme il faut donc utiliser l'instance elle-même pour connaître son type.

Héritage: et polymorphisme

Jusqu'à présent, l'héritage permet le factorisation de code mais ne permet pas de mettre en oeuvre le polymorphisme.

En effet, pour qu'il y ai polymorphisme il faut que la méthode appelée sur une instance soit celle correspondant au type réel de l'instance et non au type de la variable manipulant cette instance.

Pour mettre en oeuvre le polymorphisme il faut donc utiliser l'instance elle-même pour connaître son type.

Ceci est implanté grâce à un attribut caché appelé `vtable`. C'est un pointeur vers une table référençant toutes les méthodes pour lesquelles le polymorphisme doit être mis en oeuvre.

Héritage: et virtual

Tout d'abord il nous faut indiquer quelles sont les méthodes “virtuelles”, c'est à dire pour lesquelles on veut du polymorphisme.

Héritage: et virtual

Tout d'abord il nous faut indiquer quelles sont les méthodes “virtuelles”, c'est à dire pour lesquelles on veut du polymorphisme.

```
class Base {  
  
public:  
    virtual std::string toString() const {  
        return "Base";  
    }  
};
```

Héritage: et virtual

Tout d'abord il nous faut indiquer quelles sont les méthodes “virtuelles”, c'est à dire pour lesquelles on veut du polymorphisme.

```
class Base {  
  
public:  
    virtual std::string toString() const {  
        return "Base";  
    }  
};
```

On notera que la présence de la `vtable` fait “grossir” les instances de la classe: on passe ici de 1 octet à 4 octets (sur un machine 32 bits).

Héritage: et virtual

Tout d'abord il nous faut indiquer quelles sont les méthodes “virtuelles”, c'est à dire pour lesquelles on veut du polymorphisme.

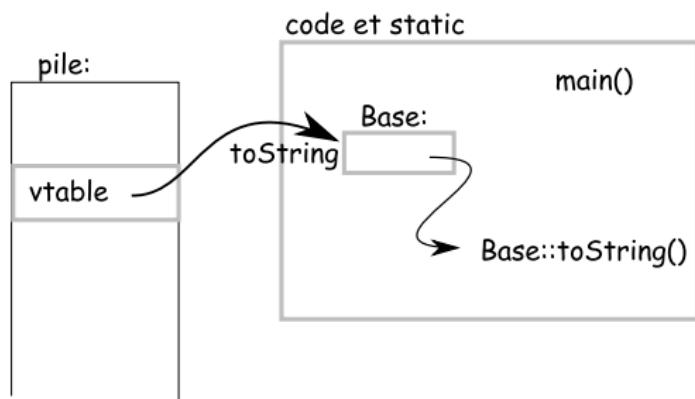
```
class Base {  
  
public :  
    virtual std::string toString() const {  
        return "Base";  
    }  
};
```

On notera que la présence de la `vtable` fait “grossir” les instances de la classe: on passe ici de 1 octet à 4 octets (sur un machine 32 bits).

Ceci crée une table associée à la classe Base. Cette table est une table de pointeurs de fonctions: les méthodes virtuelles

Héritage: vtable

```
Base b;
```



Héritage: vtable

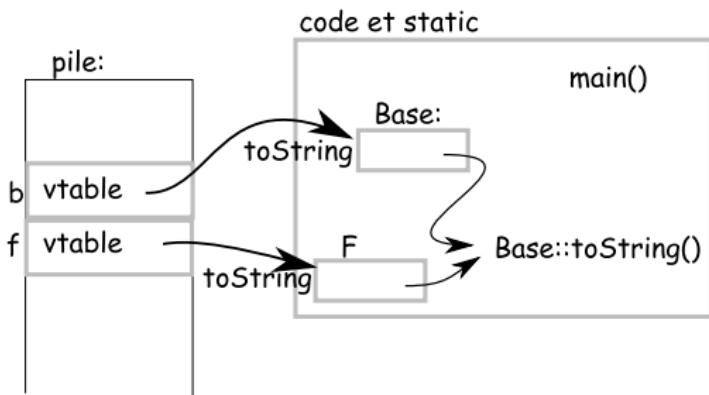
Ajoutons une classe dérivée de Base:

```
class F : public Base{  
    public:  
};
```

Si l'on ne redéfini pas la méthode `toString` alors on obtient le schéma suivant

Héritage: vtable

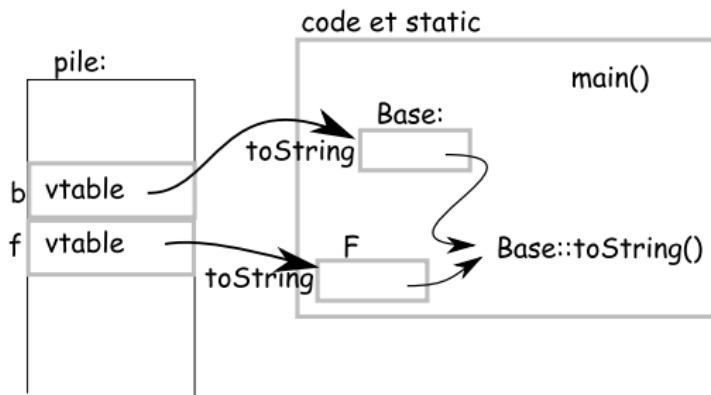
```
Base b;  
F f;
```



Une table des fonctions virtuelles est créée pour la classe F. Celle-ci est initialement construite par “recopie” de la table de la classe de base.

Héritage: vtable

```
Base b;
F f;
```



Une table des fonctions virtuelles est créée pour la classe F. Celle-ci est initialement construite par “recopie” de la table de la classe de base.

Dans les constructeurs, le compilateur ajoute une instruction permettant d’initialiser l’attribut caché `vtable` pour qu’il pointe vers la table correspondant au type de l’instance.

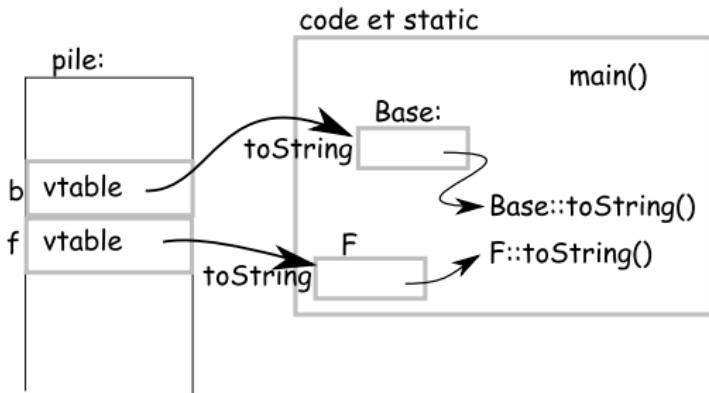
Héritage: vtable

Cas où l'on redéfini la méthode dans la classe F:

```
class F : public Base{
public:
    virtual std::string toString() const {
        return "Fille";
    }
};
```

Héritage: vtable

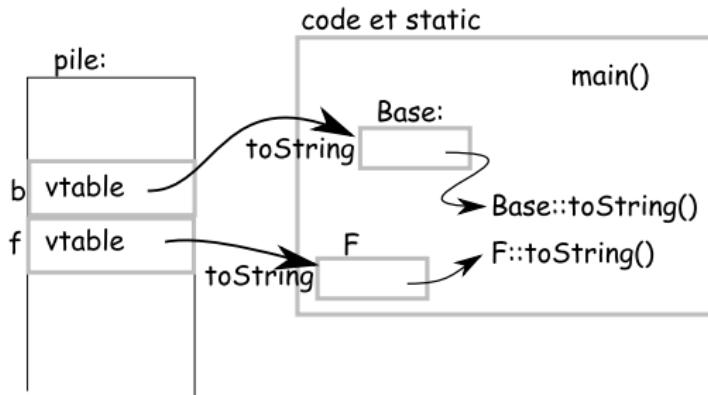
```
Base b;
F f;
```



Si une méthode est redéfinie: même nom, même arguments (même type) alors l'entrée dans la table correspondante à cette méthode est modifiée.

Héritage: vtable

```
Base b;
F f;
```



Si une méthode est redéfinie: même nom, même arguments (même type) alors l'entrée dans la table correspondante à cette méthode est modifiée.
 On voit que ce mécanisme ne modifie en rien le code de la méthode, c'est la façon dont la méthode va être appelée qui va être changé.

Héritage: appel d'une méthode virtuelle

Lors de l'appel à une méthode sur une instance, le compilateur commence par chercher cette méthode dans la classe correspondante au type de la variable sur laquelle la méthode est appelée.

```
Fille f;  
Base &r=f;  
  
r.toString();
```

Héritage: appel d'une méthode virtuelle

Lors de l'appel à une méthode sur une instance, le compilateur commence par chercher cette méthode dans la classe correspondante au type de la variable sur laquelle la méthode est appelée.

```
Fille f;  
Base &r=f;  
  
r.toString();
```

Une fois cette méthode trouvée, il y a deux possibilités: soit la méthode est virtuelle, soit elle ne l'est pas.

Héritage: appel d'une méthode virtuelle

Lors de l'appel à une méthode sur une instance, le compilateur commence par chercher cette méthode dans la classe correspondante au type de la variable sur laquelle la méthode est appelée.

```
Fille f;  
Base &r=f;  
  
r.toString();
```

Une fois cette méthode trouvée, il y a deux possibilités: soit la méthode est virtuelle, soit elle ne l'est pas.

Si elle n'est pas virtuelle, alors c'est la méthode du type de la variable qui est appelée, dans l'exemple `Base::toString`.

Héritage: appel d'une méthode virtuelle

Lors de l'appel à une méthode sur une instance, le compilateur commence par chercher cette méthode dans la classe correspondante au type de la variable sur laquelle la méthode est appelée.

```
Fille f;  
Base &r=f;  
  
r.toString();
```

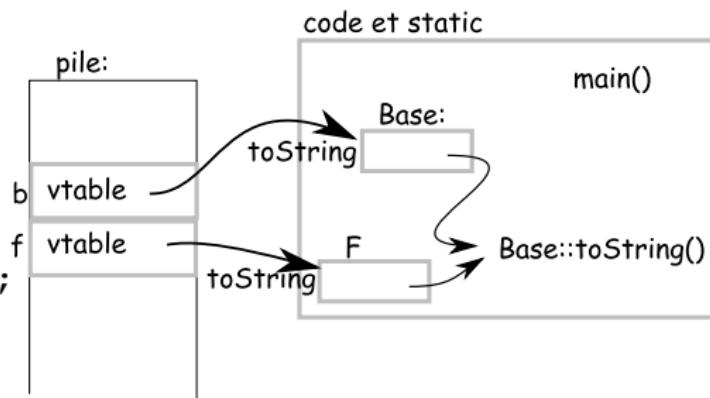
Une fois cette méthode trouvée, il y a deux possibilités: soit la méthode est virtuelle, soit elle ne l'est pas.

Si elle n'est pas virtuelle, alors c'est la méthode du type de la variable qui est appelée, dans l'exemple `Base::toString`.

Si elle est virtuelle, alors c'est la méthode dont l'adresse est dans la `vtable` qui est appelée, dans l'exemple cela dépend si elle a été redéfinie.

Héritage: vtable

```
Fille f;
Base &r=f;
r.toString();
```



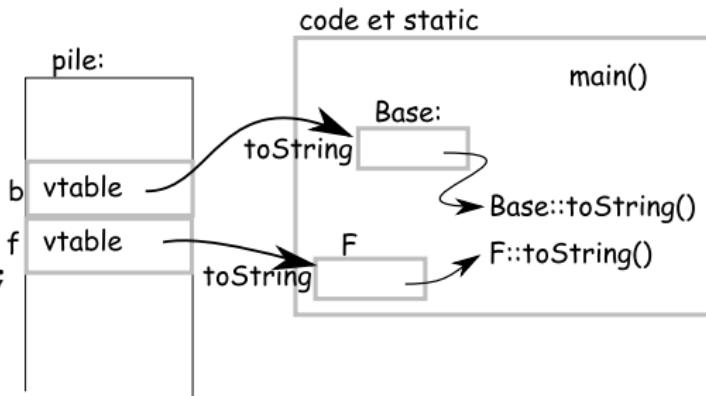
L'appel à la méthode peut être vu comme

```
r.vtable["toString"]();
```

Dans cet exemple, cela correspond à l'appel de `Base::toString()` car la méthode n'a pas été redéfinie.

Héritage: vtable

```
Fille f;
Base &r=f;
r.toString();
```



L'appel à la méthode peut être vu comme

```
r.vtable["toString"]();
```

Dans cet exemple, cela correspond à l'appel de `Fille::toString()` car la méthode a été redéfinie.

Héritage: virtuelle pure

Une méthode est dite “virtuelle pure” si elle n'est pas implémentée dans la classe où elle est déclarée.

```
classe Base{  
    public:  
        virtual std::string toString() const =0;  
};
```

Le =0 est bien compréhensible: cela correspond à mettre à 0 le pointeur dans la table des fonctions virtuelles!

Héritage: virtuelle pure

Une méthode est dite “virtuelle pure” si elle n'est pas implémentée dans la classe où elle est déclarée.

```
classe Base{  
    public:  
        virtual std::string toString() const =0;  
};
```

Le =0 est bien compréhensible: cela correspond à mettre à 0 le pointeur dans la table des fonctions virtuelles!

Une classe n'est instanciable que si sa table des fonctions virtuelles ne comporte pas de 0.

Héritage: virtuelle pure

Une méthode est dite “virtuelle pure” si elle n'est pas implémentée dans la classe où elle est déclarée.

```
classe Base{  
    public:  
        virtual std::string toString() const =0;  
};
```

Le =0 est bien compréhensible: cela correspond à mettre à 0 le pointeur dans la table des fonctions virtuelles!

Une classe n'est instanciable que si sa table des fonctions virtuelles ne comporte pas de 0.

Dans cette exemple, `Base` n'est donc pas instanciable et toute sous classe de `Base` ne sera instanciable que si elle redéfinie la méthode `toString`.

Héritage: virtuelle pure

Soit le code suivant:

```
class Base{
public:
    virtual void m(const Base &)=0;
};

class Fille : public Base{
public:
    virtual void m(const Fille &) {}
}
```

Question:

- est-ce que `Base` est instanciable?
- est-ce que `Fille` est instanciable?

Héritage: virtuelle pure

Soit le code suivant:

```
class Base{  
public:  
    virtual void m(const Base &)=0;  
};  
  
class Fille : public Base{  
public:  
    virtual void m(const Fille &) {}  
}
```

Question:

- est-ce que `Base` est instanciable?
- est-ce que `Fille` est instanciable?

Les deux classes ne sont pas instanciables car la méthode `Fille::m` est une surcharge et non une redéfinition.

Héritage: et redéfinition

La redéfinition d'une méthode consiste à écrire dans une classe dérivée une méthode qui

- porte le même nom
- a le même nombre et type d'arguments
- a comme type de retour un sous-type du type de retour de la méthode redéfinie ou le même type.

Héritage: et valeur par défaut

En C++, les arguments des fonctions et méthodes peuvent prendre des valeurs par défaut pour les arguments:

```
void f (int =0) { }
```

```
f(); // Appel f(0);
```

Les valeurs par défaut doivent être précisées dans la déclaration, en effet elle indique au compilateur comment compléter l'appel avec ces valeurs si nécessaire.

Héritage: et valeur par défaut

En C++, les arguments des fonctions et méthodes peuvent prendre des valeurs par défaut pour les arguments:

```
void f (int =0) { }
```

```
f(); // Appel f(0);
```

Les valeurs par défaut doivent être précisées dans la déclaration, en effet elle indique au compilateur comment compléter l'appel avec ces valeurs si nécessaire.

Les valeurs par défaut doivent être données du dernier argument vers le premier argument afin d'éviter toute ambiguïté.

Héritage: et valeur par défaut

```
class M{  
public:  
    virtual ~M() {}  
    virtual void m(int =0)=0;  
};  
class F: public M{  
public:  
    virtual void m(int i=1) {}  
};  
  
int main()  
{  
    F f;  
    M &m=f;  
  
    m.m(); // appel de F::m(0);  
    f.m(); // appel de F::m(1);  
}
```

Bien que l'appel soit virtuel, c'est le type de la variable qui est utilisé pour connaître les valeurs par défaut des arguments!

Héritage: et valeur par défaut

```
class M{                                int main()
public:                                {
    virtual ~M() {}                     F f;
    virtual void m(int =0)=0;           M &m=f;
};                                         m.m(); // appel de F::m(0);
class F: public M{                      f.m(); // Erreur !
public:                                }
    virtual void m(int i){}             }
```

D'une manière générale on evitera de mélanger redéfinition, surcharge et valeur par défaut: il faut toujours (c'est aussi valable pour les opérateurs, exceptions...) que le comportement attendu soit "limpide".

Héritage: et using

Lorsque l'on mélange redéfinition et surcharge il peut arriver des choses “bizarre”:

```
class M{
public:
void m(const M&);
```

F f, g;
M &m=f, &n=g;

```
} ;
```

f.m(g); // 1. ok

```
class F: public M{
public:
void m(const F&);
```

m.m(n); // 2. ok

```
} ;
```

m.m(f); // 3. ok

f.m(m); // 4. erreur!

Héritage: et using

Lorsque l'on mélange redéfinition et surcharge il peut arriver des choses “bizarre”:

```
class M{
public:
void m(const M&);
```

F f, g;
M &m=f, &n=g;

```
} ;
```

f.m(g); // 1. ok

```
class F: public M{
public:
void m(const F&);
```

m.m(n); // 2. ok

```
} ;
```

m.m(f); // 3. ok

f.m(m); // 4. erreur!

Quelles méthodes sont appelées en 1,2 et 3 ?

Héritage: et using

Lorsque l'on mélange redéfinition et surcharge il peut arriver des choses “bizarre”:

```
class M{
public:
void m(const M&);
```

F f, g;
M &m=f, &n=g;

```
} ;
```

f.m(g); // 1. ok

```
class F: public M{
public:
void m(const F&);
```

m.m(n); // 2. ok

```
} ;
```

m.m(f); // 3. ok

f.m(m); // 4. erreur!

Quelles méthodes sont appelées en 1,2 et 3 ?

Pour le cas 4, comme F hérite de M on pourrait s'attendre à ce que la méthode M::m soit appelée.

Le problème ici est lié à l'algorithme de résolution des appels et à la surcharge.

Algorithme de résolution

Lors de l'appel d'une méthode m sur une variable v de type V pointant sur une instance de type I , la méthode est cherchée de la manière suivante:

- Existe-t-il des méthodes de nom m dans la classe V ?

Algorithme de résolution

Lors de l'appel d'une méthode m sur une variable v de type V pointant sur une instance de type I , la méthode est cherchée de la manière suivante:

- Existe-t-il des méthodes de nom m dans la classe V ?
- Oui (partie 1):
 - Si une méthode est compatible: elle est choisie
 - Si plusieurs méthodes sont compatibles: ambiguïté
 - Si aucune méthode n'est compatible: erreur.

Algorithme de résolution

Lors de l'appel d'une méthode m sur une variable v de type V pointant sur une instance de type I , la méthode est cherchée de la manière suivante:

- Existe-t-il des méthodes de nom m dans la classe V ?
- Oui (partie 1):
 - Si une méthode est compatible: elle est choisie
 - Si plusieurs méthodes sont compatibles: ambiguïté
 - Si aucune méthode n'est compatible: erreur.
- Non (partie 2):
 - On cherche m dans la hiérarchie de V . Lorsqu'une méthode est trouvée, on applique la partie 1.

Algorithme de résolution

Lors de l'appel d'une méthode m sur une variable v de type V pointant sur une instance de type I , la méthode est cherchée de la manière suivante:

- Existe-t-il des méthodes de nom m dans la classe V ?
- Oui (partie 1):
 - Si une méthode est compatible: elle est choisie
 - Si plusieurs méthodes sont compatibles: ambiguïté
 - Si aucune méthode n'est compatible: erreur.
- Non (partie 2):
 - On cherche m dans la hiérarchie de V . Lorsqu'une méthode est trouvée, on applique la partie 1.
- Une méthode a été trouvée (sinon erreur).

Algorithme de résolution

Lors de l'appel d'une méthode `m` sur une variable `v` de type `V` pointant sur une instance de type `I`, la méthode est cherchée de la manière suivante:

- Existe-t-il des méthodes de nom `m` dans la classe `V`?
- Oui (partie 1):
 - Si une méthode est compatible: elle est choisie
 - Si plusieurs méthodes sont compatibles: ambiguïté
 - Si aucune méthode n'est compatible: erreur.
- Non (partie 2):
 - On cherche `m` dans la hiérarchie de `V`. Lorsqu'une méthode est trouvée, on applique la partie 1.
- Une méthode a été trouvée (sinon erreur).
- Si elle est virtuelle:
 - On effectue l'appel en utilisant la `vtable`
 - Si la méthode a été redéfinie dans `I` alors c'est `m : I` qui est appelée.
 - Sinon c'est la méthode du plus proche parent.

Algorithme de résolution

Lors de l'appel d'une méthode m sur une variable v de type V pointant sur une instance de type I , la méthode est cherchée de la manière suivante:

- Existe-t-il des méthodes de nom m dans la classe V ?
- Oui (partie 1):
 - Si une méthode est compatible: elle est choisie
 - Si plusieurs méthodes sont compatibles: ambiguïté
 - Si aucune méthode n'est compatible: erreur.
- Non (partie 2):
 - On cherche m dans la hiérarchie de V . Lorsqu'une méthode est trouvée, on applique la partie 1.
- Une méthode a été trouvée (sinon erreur).
- Si elle est virtuelle:
 - On effectue l'appel en utilisant la vtable
 - Si la méthode a été redéfinie dans I alors c'est $m : : I$ qui est appelée.
 - Sinon c'est la méthode du plus proche parent.
- Sinon on appelle $m : : V$

Algorithme de résolution

Lors de l'appel d'une méthode `m` sur une variable `v` de type `V` pointant sur une instance de type `I`, la méthode est cherchée de la manière suivante:

- Existe-t-il des méthodes de nom `m` dans la classe `V`? \Rightarrow **using**
- Oui (partie 1):
 - Si une méthode est compatible: elle est choisie
 - Si plusieurs méthodes sont compatibles: ambiguïté
 - Si aucune méthode n'est compatible: erreur.
- Non (partie 2):
 - On cherche `m` dans la hiérarchie de `V`. Lorsqu'une méthode est trouvée, on applique la partie 1.
- Une méthode a été trouvée (sinon erreur).
- Si elle est virtuelle:
 - On effectue l'appel en utilisant la `vtable`
 - Si la méthode a été redéfinie dans `I` alors c'est `m : I` qui est appelée.
 - Sinon c'est la méthode du plus proche parent.
- Sinon on appelle `m : V`

Appel avec opérateur de portée

Lorsque l'on utilise l'opérateur de portée, l'appel de méthode se fait en inspectant la classe indiquée dans l'appel et sans tenir compte du **virtual**.

```
struct A{  
    virtual void m() {}  
};  
  
struct B: A{  
};  
  
struct C: B{  
    virtual void m() {}  
};  
  
C c;  
A &a=c;  
B &b=c;  
a.m();  
c.B::m();  
b.m();  
b.B::m();
```

Les appels utilisant l'opérateur de portée ne passent pas par la `vtable`, la résolution se faisant à la compilation.

Héritage: et using

On peut utiliser le mot clé **using** pour inclure des méthodes parent dans les méthodes à inspecter:

```
class M{
public:
    void m(const M&);
```

} ;

```
class F: public M{
public:
    using M::m;
    void m(const F&);
```

} ;

F f, g;
f.m(g); // 1. ok
M &m=f, &n=g;
m.m(n); // 2. ok
m.m(f); // 3. ok
f.m(m); // 4. ok

Le **using M::m;** dans la classe F indique au compilateur d'inclure la ou les méthodes de nom m dans la classe M dans l'algorithme de résolution.

Héritage: classe abstraite

On appelle "classe abstraite" une classe ayant au moins une méthode virtuelle pure.

Héritage: classe abstraite

On appelle "classe abstraite" une classe ayant au moins une méthode virtuelle pure.

En C++, il n'existe pas de définition d'interface (dans la norme), cependant on pourra appeler interface une classe ne disposant que de méthodes virtuelles pures (et un destructeur virtuel).

Héritage: classe abstraite

On appelle "classe abstraite" une classe ayant au moins une méthode virtuelle pure.

En C++, il n'existe pas de définition d'interface (dans la norme), cependant on pourra appeler interface une classe ne disposant que de méthodes virtuelles pures (et un destructeur virtuel).

Les classes abstraites et les interfaces ne sont pas instanciables.

Une classe abstraite ou interface ne peut pas être utilisée comme type de retour par recopie ou comme argument par valeur (recopie).

On ne peut utiliser que des pointeurs et références sur ces types.

Héritage: et interface

Exemple d'une interface:

```
class Usager{
public:
    virtual ~Usager() { }
    virtual std::string nom() const=0;
    virtual void monterDans(Transport &)=0;
};
```

Héritage: et classe abstraite

Exemple d'une classe abstraite:

```
class PassagerAbstrait{
protected:
    std::string _nom;
    int _etat;
    int _destination;
    ...
    virtual choixPlaceMontee(Bus &b)=0;
    ...
public:
    virtual ~PassagerAbstrait() {}
    void estDebout() const {
        return _etat==DEBOUT;
    }
};
```

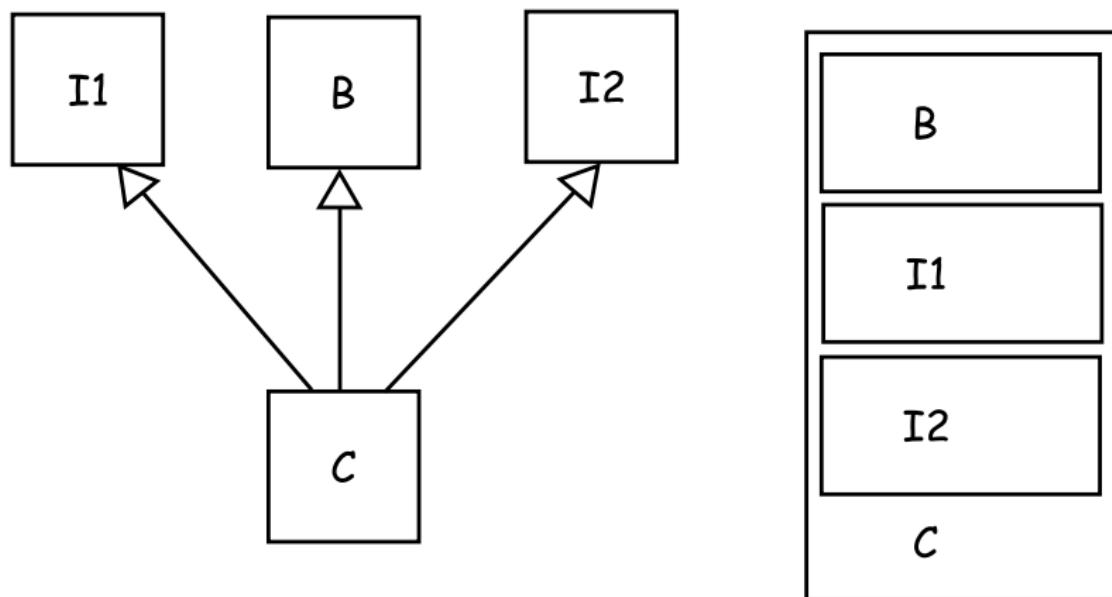
Héritage multiple

En C++, une classe peut avoir plusieurs classes de base, que celle-ci soient abstraites ou non:

```
class I1{ ... };
class I2{ ... };
class B{ ... };
class C: public B, public I1, public I2 {
    // implémentation de toutes les méthodes
    // virtuelles pures.
};
```

La classe C hérite ici de trois classes. Ainsi, on pourra se représenter C en mémoire de la façon suivante:

Hérite multiple: exemple



Héritage multiple

L'opérateur de portée et le **using** sont très utiles pour lever les ambiguïtés:

```
class I1{
    public:
        void m();
};

class I2{
    public:
        void m();
};

class F: public I1, public I2{
    ...
    m(); // Erreur!
    I1::m(); // ok
    ...
};
```

Héritage multiple

L'opérateur de portée et le **using** sont très utiles pour lever les ambiguïtés:

```
class I1{
    public:
        void m();
};

class I2{
    public:
        void m();
};

class F: public I1, public I2{
    ...
    using I2::m;
    m(); // ok
    I1::m(); // ok
    ...
};
```

Héritage multiple virtuel

Il peut arriver qu'une classe possède plusieurs fois une même classe comme classe de base:

```
class I{ public: int _value; };

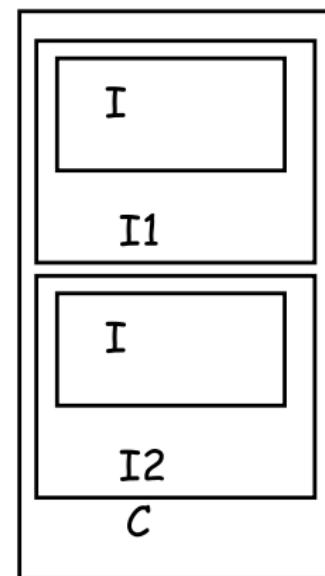
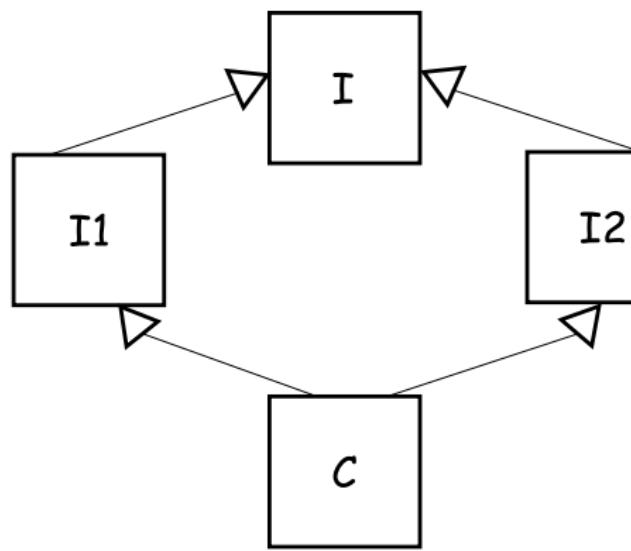
class I1: public I{};

class I2: public I{};

class C: public I1, public I2{};
```

Dans ce cas, C dispose de deux instances de la classe I:

Hérite multiple: exemple



Héritage multiple: ambiguïté

Ainsi, une instance de la classe C à deux attributs `_value`:

```
C c;  
c._value ; // Erreur!
```

L'opérateur de portée permet de lever cette ambiguïté:

```
C c;  
c.I1::_value ; // Ok  
c.I1::I::_value ; // Ok  
c.I2::_value ; // Ok
```

Héritage multiple: virtuel

Dans le cas d'un héritage en losange comme celui que nous venons de voir, les classes `I1` et `I2` pourraient vouloir partager leur classe de base `I`.

Héritage multiple: virtuel

Dans le cas d'un héritage en losange comme celui que nous venons de voir, les classes `I1` et `I2` pourraient vouloir partager leur classe de base `I`.

Pour cela, on utilise l'héritage virtuel.

Héritage multiple: virtuel

Dans le cas d'un héritage en losange comme celui que nous venons de voir, les classes `I1` et `I2` pourraient vouloir partager leur classe de base `I`.

Pour cela, on utilise l'héritage virtuel.

L'héritage virtuel permet d'indiquer qu'une classe est prête à partager une de ces classes de base avec une autre classe qui aurait fait de même:

Héritage multiple: virtuel

Dans le cas d'un héritage en losange comme celui que nous venons de voir, les classes I1 et I2 pourraient vouloir partager leur classe de base I.

Pour cela, on utilise l'héritage virtuel.

L'héritage virtuel permet d'indiquer qu'une classe est prête à partager une de ces classes de base avec une autre classe qui aurait fait de même:

```
class I{};  
class I1 : public virtual I{};  
class I2 : public virtual I{};  
class I3 : public I {};  
class C : public I1, public I2{};  
class D : public I1, public I3{};
```

Héritage multiple: virtuel

Dans le cas d'un héritage en losange comme celui que nous venons de voir, les classes `I1` et `I2` pourraient vouloir partager leur classe de base `I`.

Pour cela, on utilise l'héritage virtuel.

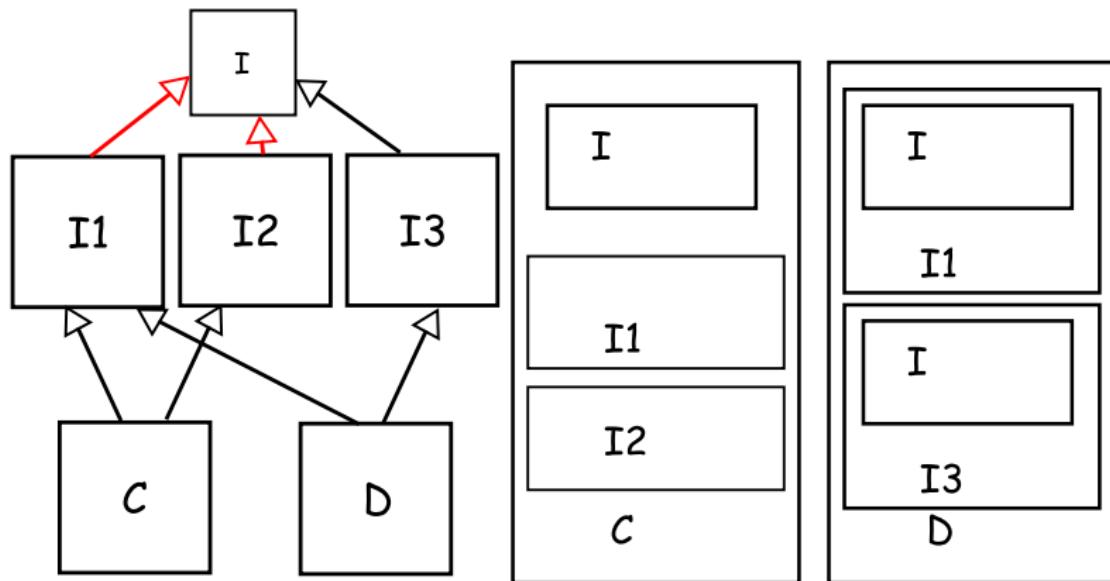
L'héritage virtuel permet d'indiquer qu'une classe est prête à partager une de ces classes de base avec une autre classe qui aurait fait de même:

```
class I{};  
class I1 : public virtual I{};  
class I2 : public virtual I{};  
class I3 : public I {};  
class C : public I1, public I2{};  
class D : public I1, public I3{};
```

Une instance de la classe `C` ne contient qu'un seul `I`

Une instance de la classe `D` en possède deux

Hérite multiple: virtuel



Hérite multiple: virtuel

L'héritage virtuel permet de lever des ambiguïtés:

```
class I{ public: virtual void m()=0; };
class I1 : public virtual I{};
class I2 : public virtual I{};
class I3 : public I {};
class C : public I1, public I2{};
class D : public I1, public I3{};
```

```
C c;
c.m(); // ok
D d;
```

```
d.m(); // Erreur!
```

On utilisera systématiquement l'héritage virtuel lors de l'héritage d'une interface.

Hérite multiple: virtuel

L'héritage virtuel permet de lever des ambiguïtés:

```
class I{ public: virtual void m()=0; };
class I1 : public virtual I{};
class I2 : public virtual I{};
class I3 : public I {};
class C : public I1, public I2{};
class D : public I1, public I3{};
```

```
C c;
c.m(); // ok
D d;
```

```
d.m(); // Erreur!
```

On utilisera systématiquement l'héritage virtuel lors de l'héritage d'une interface.

Dans le cas ci-dessus, `I` doit bien entendu posséder un destructeur virtuel.

Hérite multiple: virtuel

D'une façon générale, on n'utilisera pas l'héritage virtuel si la classe de base possède des propriétés (attributs) pouvant changer.

On pourra s'autoriser l'héritage virtuel si la classe de base est une interface, ou bien une classe ne possédant que des méthodes virtuelles pures, un constructeur par défaut et des attributs dont la valeur est fixée lors de la construction.

Plan

6 Exceptions

Exceptions : définition

Une exception est un mécanisme permettant d'interrompre l'exécution "normale" du programme.

Exceptions : définition

Une exception est un mécanisme permettant d'interrompre l'exécution "normale" du programme.

Lorsqu'une exception se produit, l'exécution va remonter la pile jusqu'à trouver un gestionnaire d'exception.

Exceptions : définition

Une exception est un mécanisme permettant d'interrompre l'exécution "normale" du programme.

Lorsqu'une exception se produit, l'exécution va remonter la pile jusqu'à trouver un gestionnaire d'exception.

Si aucun gestionnaire n'est trouvé, alors le programme se termine.

Exceptions : définition

Une exception est un mécanisme permettant d'interrompre l'exécution "normale" du programme.

Lorsqu'une exception se produit, l'exécution va remonter la pile jusqu'à trouver un gestionnaire d'exception.

Si aucun gestionnaire n'est trouvé, alors le programme se termine.

Les exceptions sont typées : une instance est associée au déclenchement de l'exception et différents gestionnaires doivent être écrits selon le type de l'exception.

Exceptions : la levée

La levée (ou déclenchement) d'exception se fait à l'aide de l'instruction **throw**:

```
void affiche_text(char *s) {  
    if (s==NULL)  
        throw "Argument_Invalidé";  
    ...  
}
```

Dans cet exemple, l'exception levée est de type **const char ***. Pour pouvoir gérer cette exception, le gestionnaire devra gérer ce type d'exception.

Exceptions : la capture

On appelle "capture d'exception" le fait de stopper la remonté de pile due à une levée.

Exceptions : la capture

On appelle "capture d'exception" le fait de stopper la remonté de pile due à une levée.

Pour cela deux conditions doivent être remplies:

- ① l'exception doit avoir été levée par des instructions exécutée dans un bloc **try { ... }.**
- ② le type du gestionnaire doit être compatible avec le type de la levée.

Exceptions : la capture

On appelle "capture d'exception" le fait de stopper la remonté de pile due à une levée.

Pour cela deux conditions doivent être remplies:

- ① l'exception doit avoir été levée par des instructions exécutée dans un bloc **try { ... }.**
- ② le type du gestionnaire doit être compatible avec le type de la levée.

```
try {
    affiche_text(NULL);
}
catch (const char *s) {
    ... //gestion de l'exception
}
```

Exceptions : les gestionnaires

Pour un même bloc **try**, plusieurs gestionnaires peuvent être écrits:

```
try {  
    ...  
}  
catch (int i){ ... }  
catch (const char *){ ... }  
catch (...) { ... }
```

Le gestionnaire **catch (...)** permet de capturer tout type d'exceptions : il doit être en dernier.

Exceptions : les gestionnaires

Pour un même bloc **try**, plusieurs gestionnaires peuvent être écrits:

```
try {  
    ...  
}  
catch (int i) { ... }  
catch (const char *) { ... }  
catch (...) { ... }
```

Le gestionnaire **catch (...)** permet de capturer tout type d'exceptions : il doit être en dernier.

Les gestionnaires sont "testés" dans l'ordre d'apparition.

Exceptions : polymorphisme

```
class Erreur{};  
struct ArgErreur : Erreur {};  
  
try{...}  
catch (const ArgErreur &e){...}  
catch (const Erreur &e){...}  
  
try{...}  
catch (const Erreur &e){...}  
catch (const ArgErreur &e){...}// Jamais atteint;  
}
```

Exceptions : propagation

Une exception peut être capturée puis re-levée à l'aide de l'instruction **throw**:

```
try {  
    ...  
}  
catch (const char *s) {  
    throw; // re-levée  
}
```

Exceptions : mémoire

Une gestion correcte des exceptions est assez difficile en C++ à cause de problèmes mémoires:

Pour l'allocation automatique le compilateur s'assure que toutes les variables automatiques sont correctement libérées au fur et à mesure de la remonté:

```
void g() { throw 1; }
int f() {
    A a;
    g(); // le destructeur de "a" est appelé
}
```

Dans le cas dynamique:

```
void g() { throw 1; }
int f() {
    A *a=new A;
    g(); // l'instance est "perdue"
}
```

Exceptions : mémoire

Il en résulte que toute allocation dynamique doit se faire dans un bloc **try** et que la libération doit être prévue dans le(s) gestionnaire(s):

```
void g() { throw 1; }
int f() {
    A *a;
    try {
        a=new A;
        g(); // l'instance est "perdue"
    }
    catch (...) {
        delete a;
        throw;
    }
}
```

Ce code est correct sauf si l'exception levée est `bad_alloc`!

Exceptions : mémoire

```
void g() { throw 1; }
int f() {
    A *a;
    try {
        a=new A;
        g(); // l'instance est "perdue"
    }
    catch (const bad_alloc &b) {
        throw;
    }
    catch (...) {
        delete a;
        throw;
    }
}
```

Le problème dans ce cas est qu'on ne sait pas si c'est le **new** ou bien **g** qui est à l'origine de l'exception.

Exceptions : mémoire

Il faut être méticuleux:

```
void g() { throw 1; }
int f() {
    A *a=NULL; // ICI
    try {
        a=new A;
        g();
    }
    catch (...) {
        if (a!=NULL) // libération
            delete a;
        throw; // propagation
    }
}
```

On peut aussi utiliser la version de l'opérateur **new** qui ne lève pas d'exception:

```
a = new (nothrow) A;
```

Exceptions : prototypes

Il est possible d'indiquer la liste des exceptions qu'une fonction est susceptible de lever ou bien de propager:

Exceptions : prototypes

Il est possible d'indiquer la liste des exceptions qu'une fonction est susceptible de lever ou bien de propager:

```
void f();  
void g() throw(int);  
void h() throw(const char *, int, bad_alloc);  
void k() throw();
```

- f peut lever tout type d'exception
- g ne peut lever qu'une exception de type int
- h ne peut lever qu'une exception de type const char*, int ou bien bad_alloc
- k ne peut pas lever d'exception.

Si g, h ou k doit lever ou propager une exception qui n'est pas dans la liste alors la fonction unexpected est appelée.

La fonction `unexpected` permet de lever une exception autorisée par la fonction. Le gestionnaire associé est modifié à l'aide de:

```
handler set_unexpected(handler h) throw() ;
```

Le gestionnaire par défaut appelle la fonction `terminate`.

La fonction `unexpected` permet de lever une exception autorisée par la fonction. Le gestionnaire associé est modifié à l'aide de:

```
handler set_unexpected(handler h) throw() ;
```

Le gestionnaire par défaut appelle la fonction `terminate`.

Cette fonction est appelée lorsqu'une exception n'est pas gérée et par défaut mène à la fin du programme (appel la fonction `abort()`).

Ce comportement peut être modifié:

```
handler set_terminate(handler h) throw() ;
```

La fonction `unexpected` permet de lever une exception autorisée par la fonction. Le gestionnaire associé est modifié à l'aide de:

```
handler set_unexpected(handler h) throw();
```

Le gestionnaire par défaut appelle la fonction `terminate`.

Cette fonction est appelée lorsqu'une exception n'est pas gérée et par défaut mène à la fin du programme (appel la fonction `abort()`).

Ce comportement peut être modifié:

```
handler set_terminate(handler h) throw();
```

En fait `throw()` revient à écrire:

```
void f() throw() {  
    try {}  
    catch(...) {  
        try { unexpected(); }  
        catch(...) { terminate(); }  
    }  
}
```

ou encore:

```
void f() throw(int) {
    try {} 
    catch(int) {
        throw; // ok
    }
    catch(...) {
        try { unexpected(); }
        catch(int) {
            throw; // ok
        }
        catch(...) {
            terminate();
        }
    }
}
```

Exceptions : standard

La bibliothèque standard propose une hiérarchie d'exceptions avec comme classe de base la classe `std::exception`:

```
#include <exception>

namespace std {
    class exception {
public:
    exception() throw();
    exception(const exception&) throw();
    exception& operator=(const exception&) throw();
    virtual ~exception() throw();
    virtual const char* what() const throw();
};
}
```

Exceptions : standard

```
#include <stdexcept>

namespace std {
    class logic_error : public exception { ... };
    class domain_error : public logic_error { ... };
    class invalid_argument : public logic_error { ... };
    class length_error : public logic_error { ... };
    class out_of_range : public logic_error { ... };

    class runtime_error : public exception { ... };
    class range_error : public runtime_error { ... };
    class overflow_error : public runtime_error { ... };
    class underflow_error : public runtime_error { ... };
}
```

Deux types d'exceptions, les `logic_error` et les `runtime_error`.

Exceptions : conclusion

En conclusion, les exceptions sont un outil puissant pour la gestion d'erreurs dans un programme.

Exceptions : conclusion

En conclusion, les exceptions sont un outil puissant pour la gestion d'erreurs dans un programme.

Elles permettent de mettre en place des systèmes d'erreurs beaucoup plus complet qu'en C car on peut "porter" beaucoup d'informations via l'exception.

Exceptions : conclusion

En conclusion, les exceptions sont un outil puissant pour la gestion d'erreurs dans un programme.

Elles permettent de mettre en place des systèmes d'erreurs beaucoup plus complet qu'en C car on peut "porter" beaucoup d'informations via l'exception.

Peut poser des problèmes avec l'allocation dynamique. Pour cela on peut:

- accepter une fuite mémoire (!)
- utiliser des outils qui libère la mémoire correctement (type smart pointeurs)

Plan

7

Template

Template: présentation

En plus du support de la programmation objet, le C++ ajoute au C la programmation générique.

Template: présentation

En plus du support de la programmation objet, le C++ ajoute au C la programmation générique.

La programmation générique consiste à écrire du code (fonction ou classe) indépendamment de certains types. Ceux-ci seront fixés plus tard lors de l'utilisation de ce code.

Template: présentation

En plus du support de la programmation objet, le C++ ajoute au C la programmation générique.

La programmation générique consiste à écrire du code (fonction ou classe) indépendamment de certains types. Ceux-ci seront fixés plus tard lors de l'utilisation de ce code.

En C++ c'est le mot clé **template** qui permet la définition de la générnicité

Template: présentation

En plus du support de la programmation objet, le C++ ajoute au C la programmation générique.

La programmation générique consiste à écrire du code (fonction ou classe) indépendamment de certains types. Ceux-ci seront fixés plus tard lors de l'utilisation de ce code.

En C++ c'est le mot clé **template** qui permet la définition de la générnicité

On préfixera le code générique d'une instruction **template<class T>** où T devient un type variable, paramètre du code.

Template: présentation

On pourra écrire deux types de code générique: des classes génériques et des fonctions génériques.

Voici un exemple de fonction générique:

```
template <class T>
void swap(T &a, T&b) {
    T t=a;
    a=b;
    b=t;
}
```

Cette fonction effectue l'échange de valeur entre `a` et `b` quel que soit leur type.
On voit cependant que `a` et `b` doivent être de même type

L'utilisation de cette fonction swap se fait comme suit:

```
template<class T>
void swap(T &a, T&b) { ... }

int a=0;
int b=1;
char c='a';

swap<int>(a,b); // 1
swap(a,b); // 2
swap(a,c); // 3
```

L'utilisation de cette fonction swap se fait comme suit:

```
template<class T>
void swap(T &a, T&b) { ... }

int a=0;
int b=1;
char c='a';

swap<int>(a,b); // 1
swap(a,b); // 2
swap(a,c); // 3
```

Dans le cas 1, nous indiquons explicitement le type pour T, le compilateur remplace alors T par int et trouve la fonction swap<int> (int &, int &) qui convient à l'appel. Par la même occasion, il compile cette fonction.

L'utilisation de cette fonction `swap` se fait comme suit:

```
template<class T>
void swap(T &a, T&b) { ... }

int a=0;
int b=1;
char c='a';

swap<int>(a,b); // 1
swap(a,b); // 2
swap(a,c); // 3
```

Dans le cas 1, nous indiquons explicitement le type pour `T`, le compilateur remplace alors `T` par `int` et trouve la fonction `swap<int>(int &, int &)` qui convient à l'appel. Par la même occasion, il compile cette fonction.

Dans le cas 2, le compilateur cherche la fonction `swap` et essaye de déduire un type convenable pour `T` à partir des arguments (ceci n'est pas toujours possible). Il construit `swap<int>` et compile cette fonction.

L'utilisation de cette fonction `swap` se fait comme suit:

```
template <class T>
void swap(T &a, T&b) { ... }

int a=0;
int b=1;
char c='a';

swap<int>(a,b); // 1
swap(a,b); // 2
swap(a,c); // 3
```

Dans le cas 1, nous indiquons explicitement le type pour `T`, le compilateur remplace alors `T` par `int` et trouve la fonction `swap<int>(int &, int &)` qui convient à l'appel. Par la même occasion, il compile cette fonction.

Dans le cas 2, le compilateur cherche la fonction `swap` et essaye de déduire un type convenable pour `T` à partir des arguments (ceci n'est pas toujours possible). Il construit `swap<int>` et compile cette fonction.

Dans le cas 3, il n'arrive pas à trouver une fonction `swap` avec le bon prototype et génère une erreur.

Template: et compilation

Un problème se pose pour la compilation d'un code template.

En effet, celui-ci étant à trous, il ne peut être compilé qu'une fois les types de paramètres connus.

Pour cela, on ne peut plus mettre seulement le prototype dans le .hpp, nous mettrons aussi le code de la fonction:

en deux fois:

en une fois:

```
template <class T>
void swap(T &a, T& b) {
    T t=a;
    a=b;
    b=t;
}
```

```
template <class T>
void swap(T &, T& );
```

...

```
template <class T>
void swap(T &a, T& b) {
    T t=a;a=b;b=t;
}
```

Template: classe

Voici un exemple de classe paramétrée (ou générique):

```
template<class T>
class Entier{
    T _value;
public:
    Entier(const T &t) :_value(t) {}
};
```

Template: classe

Voici un exemple de classe paramétrée (ou générique):

```
template<class T>
class Entier{
    T _value;
public:
    Entier(const T &t) :_value(t) {}
};
```

Dans le cas d'une classe, on écrira donc le code directement dans le fichier .hpp

Template: classe

Voici un exemple de classe paramétrée (ou générique):

```
template<class T>
class Entier{
    T _value;
public:
    Entier(const T &t) :_value(t) {}
};
```

Dans le cas d'une classe, on écrira donc le code directement dans le fichier .hpp

On peut tout à fait séparer les déclarations de l'implémentation:

```
template<class T>
class Entier{
    T _value;
public:
    Entier(const T &t);
};

template<class T>
Entier<T>::Entier(const T &t) :_value(t)
```

Template: classe

Lors de l'utilisation d'une classe générique, nous devons indiquer explicitement le type à utiliser pour les paramètres:

```
Entier a; // Erreur!  
Entier<int> b; // ok  
Entier<long> c; // ok
```

On parlera de type complet pour désigner une classe générique dont l'ensemble des paramètres sont spécifiés.

Template: classe

Lors de l'utilisation d'une classe générique, nous devons indiquer explicitement le type à utiliser pour les paramètres:

```
Entier a; // Erreur!  
Entier<int> b; // ok  
Entier<long> c; // ok
```

On parlera de type complet pour désigner une classe générique dont l'ensemble des paramètres sont spécifiés.

Il est important de retenir qu'il n'existe aucune relation de typage entre deux types complets correspondant à une même classe générique.

Template: valeur par défaut

On peut spécifier des valeurs par défaut pour les paramètres d'une classe générique (pas des fonctions). Celles-ci doivent être fournies de la droite vers la gauche:

```
template<class T=int>
class Entier{...}
```

```
template<class D, class H=int>
class HashTable{...}
```

```
template<class T=int, class C> // Erreur!
int compare(T *, T *)
```

```
Entier e; // ok :)
```

Template: et type primitif

Un code peut aussi être paramétré par des valeurs (constantes) de types primitifs:

```
template<int taille>
class Matrix{
    int _data[taille][taille];
public:
};
```

Template: et type primitif

Un code peut aussi être paramétré par des valeurs (constantes) de types primitifs:

```
template<int taille>
class Matrix{
    int _data[taille][taille];
    public:
};
```

Dans ce cas, `Matrix<4>` et `Matrix<3>` sont deux types différents. Cet exemple pourrait être utilisé en infographie.

Template: spécialisation

Il est possible de modifier l'implémentation pour certaines valeurs de paramètre:

```
template<class T>
void affiche(const T &t) {
    std::cout<<t<<std::endl;
}
```

Dans le cas des entiers, on utilise printf:

```
template<>
void affiche<int>(const int &t) {
    printf("%d\n", t);
}
```

Template: spécialisation

Dans le cas où les paramètres sont des types primifs, on peut spécialiser selon les valeurs:

```
template<int n>
void affiche() {
    printf("%d\n", n);
}

template<>
void affiche<0>() {
    printf("zero\n");
}

...
affiche(); // erreur !
affiche<1>(); // affiche : 1
affiche<0>(); // affiche : zero
```

Template: spécialisation

Dans le cas des classes, la spécialisation peut aussi se faire sur les pointeurs:

```
template<class T>
class Tableau{
    T *_data;
    ...
};

template<class T>
class Tableau<T *>{
    T **_data; // Attention ici !
public:
    Tableau(int i) {
        ...
        _data[i]=NULL;
        ...
    }
};
```

Template: spécialisation partielle

Il est possible de ne spécialiser que certains paramètres d'un code générique:

```
template<class T, class H>
class HashTable{
};

template<class T, class H>
class HashTable<T *, H>{
};

template<class T>
class HashTable<T *, int>{
};
```

Important: il n'y a aucune contrainte entre ces différentes implémentations car elle ne correspondent pas au même type

Template: contraintes

Lorsque l'on utilise un type dans un code générique, il faut faire très attention aux propriétés que l'on impose sur ce type:

Template: contraintes

Lorsque l'on utilise un type dans un code générique, il faut faire très attention aux propriétés que l'on impose sur ce type:

```
template<class T>
void swap(T &a, T&b) {
    T t=a;
    a=b;
    b=t;
}
```

```
template<class T>
void swap(T &a, T&b) {
    T t;
    t=a;
    a=b;
    b=t;
}
```

Template: contraintes

Lorsque l'on utilise un type dans un code générique, il faut faire très attention aux propriétés que l'on impose sur ce type:

```
template<class T>
void swap(T &a, T&b) {
    T t=a;
    a=b;
    b=t;
}
```

```
template<class T>
void swap(T &a, T&b) {
    T t;
    t=a;
    a=b;
    b=t;
}
```

On veillera à toujours indiquer clairement les méthodes attendues pour une type paramètre.

Template: STL et BOOST

Il existe deux grandes bibliothèques utilisant les templates, la Standard Template Library et BOOST.

La STL contient:

- des conteneurs (tableaux, map, ...),
- des iterateurs (parcours),
- des algorithmes (comparaison, recopie, recherche, tris...),
- des allocateurs mémoire,
- des foncteurs

La bibliothèque BOOST contient beaucoup, beaucoup de choses (conteneurs, graphes, maths, entrées/sorties, fichiers...).

Template: STL

Pour comprendre et utiliser la STL, il faut savoir qu'il est possible de déclarer un alias de type dans une classe:

```
template<class T>
class Entier{
    public:
        typedef T type;
};

Entier<int>::type i;
```

Ceci fait que la variable `i` est de type `int`

Template: STL

Parmis les conteneurs les plus connus, il y a les vecteurs, ceux-ci répondent au concept de “Conteneur à Accès Quelconque” et “Sequence avec insertion en fin”. Ces concepts définissent un certain nombre des propriétés: compléxité, nom de méthodes, type interne...

```
#include <vector>

std::vector<int> v;
v.push_back(0);
v.push_back(1);
v[0]=2;
for (std::vector<int>::iterator i=v.begin();
     i!=v.end();
     i++)
    std::cout<<"_ "<<*i;
```

Cet exemple utilise le type interne `iterator` qui correspond au type implémentant le concept d’itérateur pour la classe vecteur.

Template: STL

Les itérateurs ont été conçus pour fonctionner comme les pointeurs: on utilise `++` et `--` pour se déplacer, `*` et `->` pour utiliser la valeur représentée par l'itérateur.

Template: STL

Les itérateurs ont été conçus pour fonctionner comme les pointeurs: on utilise `++` et `--` pour se déplacer, `*` et `->` pour utiliser la valeur représentée par l'itérateur.

Voici un exemple permettant de trier un vecteur:

```
#include <vector>
```

```
vector<int> v;
v.push_back(0); v.push_back(5); v.push_back(2);
std::sort(v.begin(), v.end());
```

L'algorithme `std::sort` effectue un tri des données contenues par la séquence des deux itérateurs.

Ainsi, cet algorithme pourra être utiliser avec de nombreux autre conteneurs de la STL qui disposent des itérateurs et même des tableaux:

```
int t[3] = {0, 5, 2};
std::sort(t, t+3);
```

STL

La STL repose sur la notion de concept.

STL

La STL repose sur la notion de concept.

Un concept est un peu comme une "interface" mais qui n'a d'existence que sur le papier.

STL

La STL repose sur la notion de concept.

Un concept est un peu comme une "interface" mais qui n'a d'existence que sur le papier.

Une concept est un ensemble de méthodes, types internes, propriétés que doit posséder une classe répondant au concept.

STL

La STL repose sur la notion de concept.

Un concept est un peu comme une "interface" mais qui n'a d'existence que sur le papier.

Une concept est un ensemble de méthodes, types internes, propriétés que doit posséder une classe répondant au concept.

Exemple de concept: "container"

- `X::value_type` type contenu
- `X::iterator` type à utiliser pour parcourir ("input iterator")
- `X::const_iterator` type pour parcourir sans modifier
- `X::reference` type de la référence vers le type contenu
- `X::const_reference` référence constante
- `X::pointer` pointeur vers le type contenu
- `X::difference_type` différence entre deux iterateurs
- `X::size_type`

STL

La STL propose un ensemble de classes paramétrées répondant à un ou plusieurs concepts :

- Séquences : vector, deque, list, slist, bit_vector
- Conteneurs associatifs : set, map, multiset, multimap, hash_set, hash_map, hash_multiset, hash_multimap, hash
- Les chaînes de caractères : Character Traits, char_traits, basic_string

Ainsi qu'un ensemble d'algorithmes :

- for_each, find, find_if, adjacent_find, find_first_of, count, count_if, mismatch, equal, search, search_n, find_end
- remplacement dans une séquence ...
- tris, mélange, rotation, renversement, ...
- recherche d'élément, min, max ...

Template: et metaprogrammation

La metaprogrammation consiste à faire s'exécuter un programme par le compilateur au moment de la compilation et donc de façon statique.

Template: et metaprogrammation

La metaprogrammation consiste à faire s'exécuter un programme par le compilateur au moment de la compilation et donc de façon statique.

Un exemple classique de metaprogrammation est un calcul mathématique simple:

```
template<int a, int b>
struct Addition{
    static const int resultat=a+b;
};

std::cout<< Addition<2,3>::resultat <<std::endl;
```

Template: et metaprogrammation

et un peu plus compliqué:

```
template<int a, int b>
struct Puissance{
    static const int resultat=a*Puissance<a,b-1>::resultat;
};

template<int a>
struct Puissance<a,0>{
    static const int resultat=1;
};

std::cout<< Puissance<2,3>::resultat << std::endl;
```

Template: et metaprogrammation

La metaprogrammation permet de faire des choses très puissantes telles que la composition de types. Cependant, elle reste souvent réservée à l'implémentation de bibliothèques complexe et est difficile à maîtriser.

Elle repose sur les concepts suivant:

- les templates,
- la spécialisation,
- les types internes,
- les attributs statiques et constants

Templates: et metaprogrammation

On peut ainsi écrire un “language” sur les types, avec des branchements conditionnels:

```
template<bool C, class T, class E>
struct IfThenElse{
    typedef T result;
};

template<class T, class E>
struct IfThenElse<false , T, E>{
    typedef E result;
};

IfThenElse<true , int , char>::result i;
```

Template: et metaprogrammation

Comparaison de types:

```
template<class T, class U>
struct Equal{
    static bool result=false;
};

template<class T>
struct Equal<T, T>{
    static bool result=true;
}

bool b=Equal<int, size_t>::result;
```

```
IfThenElse<Equal<int, ssize_t>::result,
           int,
           long>::result i;
```

Template: et metaprogrammation

La métaprogrammation permet de faire des API simplifiées:

```
struct RandomContainer{};  
struct ReversibleContainer{};  
  
template<class T, class Type>  
struct Container{  
    typedef vector<T> result;  
}  
  
template<class T>  
struct Container<T, ReversibleContainer>{  
    typedef list<T> result;  
}  
  
Container<int, RandomContainer>::result container;
```

Plan

8 Plus

Plugin: introduction

Un plugin est une fonctionnalité que l'on peut ajouter à un programme sans avoir à le recompiler.

Plugin: introduction

Un plugin est une fonctionnalité que l'on peut ajouter à un programme sans avoir à le recompiler.

Celle-ci peut-être prise en compte au cours de l'execution du programme ou bien au chargement de celui-ci (relancer l'application).

Plugin: introduction

Un plugin est une fonctionnalité que l'on peut ajouter à un programme sans avoir à le recompiler.

Celle-ci peut-être prise en compte au cours de l'execution du programme ou bien au chargement de celui-ci (relancer l'application).

Afin que l'intéraction entre le plugin et l'application se passe bien, l'interface de dialogue est normalisée.

Plugin: introduction

Un plugin est une fonctionnalité que l'on peut ajouter à un programme sans avoir à le recompiler.

Celle-ci peut-être prise en compte au cours de l'execution du programme ou bien au chargement de celui-ci (relancer l'application).

Afin que l'intéraction entre le plugin et l'application se passe bien, l'interface de dialogue est normalisée.

En C++, l'application fournie une classe abstraite (interface) contenant l'ensemble des méthodes devant être implantées par le plugin (permet l'interaction). Un plugin est donc une sous-classe de cette interface.

Plugin : mise en oeuvre

Ainsi, le créateur de plugin va :

- Écrire une classe héritant de l'interface fournie par l'application
- Compiler cette classe sous la forme d'une bibliothèque
- Copier le fichier .so dans un répertoire désigné par l'application

L'application quand à elle doit :

- Trouver le plugin
- Ouvrir le plugin
- Instancier la classe écrite dans le plugin

Plugin : mise en oeuvre

Ainsi, le créateur de plugin va :

- Écrire une classe héritant de l'interface fournie par l'application
- Compiler cette classe sous la forme d'une bibliothèque
- Copier le fichier .so dans un répertoire désigné par l'application

L'application quand à elle doit :

- Trouver le plugin
- Ouvrir le plugin
- Instancier la classe écrite dans le plugin

Problèmes:

- la notion de classe n'existe pas dans un bibliothèque qui n'est qu'un regroupement de fonctions, code et variables statiques.
- comment instancier une classe dont on ne connaît pas le nom!

Plugin : mise en oeuvre

Comment manipuler des bibliothèques C++ dynamiquement? :

```
// rary.c:  
void f () {  
}  
// compilation:  
// g++ -shared rary.c -o library.so
```

Plugin : mise en oeuvre

Comment manipuler des bibliothèques C++ dynamiquement? :

```
// rary.c:  
void f () {  
}  
// compilation:  
// g++ -shared rary.c -o library.so
```

```
void *h=dlopen ("library.so", RTLD_LAZY);  
void (*f) (void) = (void (*) (void)) dlsym(h, "f");  
f();
```

Ceci ne marche pas car le symbole "f" n'est pas trouvé! Pourquoi?

Plugin : mise en oeuvre

Comment manipuler des bibliothèques C++ dynamiquement? :

```
// rary.c:  
void f () {  
}  
// compilation:  
// g++ -shared rary.c -o library.so
```

```
void *h=dlopen ("library.so", RTLD_LAZY);  
void (*f) (void) = (void (*) (void)) dlsym(h, "f");  
f();
```

Ceci ne marche pas car le symbole "f" n'est pas trouvé! Pourquoi?

Réponse : le mangling, la même bibliothèque compilée avec gcc (C) marche très bien.

Plugin : extern

Deux solutions possibles :

- connaître l'algorithme de mangling et rechercher le symbole
- faire en sorte qu'il n'y ai pas de mangling

Plugin : extern

Deux solutions possibles :

- connaître l'algorithme de mangling et rechercher le symbole
- faire en sorte qu'il n'y ai pas de mangling

```
// rary.c:  
extern "C" {  
    void f () {  
    }  
}  
// compilation:  
// g++ -shared rary.c -o library.so
```

On peut alors voir que

```
allali@troll:/tmp$ nm library.so  
U __gxx_personality_v0  
00000000 T f  
allali@troll:/tmp$
```

Plugin : instantiation

Le deuxième problème est celui de l'instanciation du plugin : L'application ne peut pas instancier le plugin car elle n'a accès qu'à des fonctions de la bibliothèque.

Plugin : instantiation

Le deuxième problème est celui de l'instanciation du plugin : L'application ne peut pas instancier le plugin car elle n'a accès qu'à des fonctions de la bibliothèque.

Pour résoudre ce problème, l'utilisateur va devoir écrire une fonction qui instanciera le plugin. Cette fonction aura un nom fixé à l'avance et devra avoir un symbole de même nom que la fonction:

```
class MonPlugin : public Plugin{...}
extern "C"{
    Plugin *fabrique() {
        returnne new MonPlugin();
    }
}
```

Plugin : chargement

Coté application, le chargement du plugin se fait de la façon suivante:

```
void *h=dlopen("library.so", RTLD_LAZY);
Plugin (*f) (void) = (Plugin (*) (void))dlsym(h, "fabrique");
Plugin *p=f();
```

Pour compléter, l'application pourra ainsi définir plusieurs fonctions devant être implantée par le plugin (`release`, ...).