Polymorphisme, Algorithmique

- copies d'objets
- rappel: polymorphisme

- algorithmique et calculabilité
- algorithmique et récursion

- structures dynamiques, basées sur les objets

May 02, 2001 12:43

Copie d'Objets

Il y a une initialisation d'un objet dans les situations suivantes:

- déclaration d'un objet avec initialisateur
  comme par une ***affectation***

```
class A ...
A a1 = a2;
```

- objet transmis par valeur ***comme paramètre d'appel***
- objet transmis par valeur ***comme résultat d'appel***

```
class A ... class B ...
B f (A a);
```

Dans chaque cas, *le constructeur par copie est utilisé* ...

Exemple: Substitutivité (I)

```
class A {
public:
   int va;
};
```

```
class B : public A {
public:
   int vb;
};
```

```
void affiche(A o) {
   cout<< o.va;
   cout<< endl;
};
```

```
void bffiche(B o) {
   cout<< o.vb;
   cout<< endl;
};
```

```
int main() {
   B b; b.va=4; b.vb=3;
   A a; a=b;
   affiche(a);
   affiche(b);
   bffiche(b);
};
```

Exemple: Substitutivité (II)

```
class A {
public:
  int va;
  void dump(){
    cout<< va;}
};
```

```
class B : public A {
public:
  int vb;
  void dump(){
    A::dump();
    cout<< vb;}
};
```

```
void affiche(A o) {
  o.dump();
  cout<< endl;
};
```

```
void bffiche(B o) {
  o.dump();
  cout<< endl;
};
```

```
int main() {
  B b; b.va=4; b.vb=3;
  A a; a=b;
  affiche(a);
  affiche(b);
  bffiche(b);
};
```

Exemple: Substitutivité (III)

```
class A {
public:
   int va;
   void dump(){
      cout<< va;}
};
```

```
class B : public A {
public:
   int vb;
   void dump(){
      A::dump();
      cout<< vb;}
};
```

```
void affiche(A &o) {
   o.dump();
   cout<< endl;
};
```

```
void bffiche(B o) {
   o.dump();
   cout<< endl;
};
```

```
main() {
   B b; b.va=4; b.vb=3;
   A a; a=b;
   affiche(a);
   affiche(b);
   bffiche(b);
};
```

Liaisons statiques et dynamiques

statique (static binding)

la signature de la fonction
connue au moment de la compilation
détermine la version de la fonction à exécuter
$\Rightarrow$ *surcharge*

dynamique (late binding)

la version de la fonction à exécuter
n'est connue qu'au moment de l'exécution

Méthodes Virtuelles

Le mot–clef `virtual` indique au compilateur qu'une méthode peut faire l'objet d'une résolution dynamique.

```
virtual <type_retour> <nom>(<args>);
```

Lorsqu'on invoque une méthode virtuelle *à partir d'une référence vers une instance*, c'est la définition de la méthode *associée au type réel* de l'instance.

- Les redéfinitions sont aussi considérées comme virtuelles.
- Les constructeurs ne peuvent pas être virtuels.
- Les destructeurs devraient être définis comme virtuels.

Exemple: Méthodes Virtuelles

```
class A {
public:
  int va;
  virtual void dump(){
    cout<< va;}
};
```

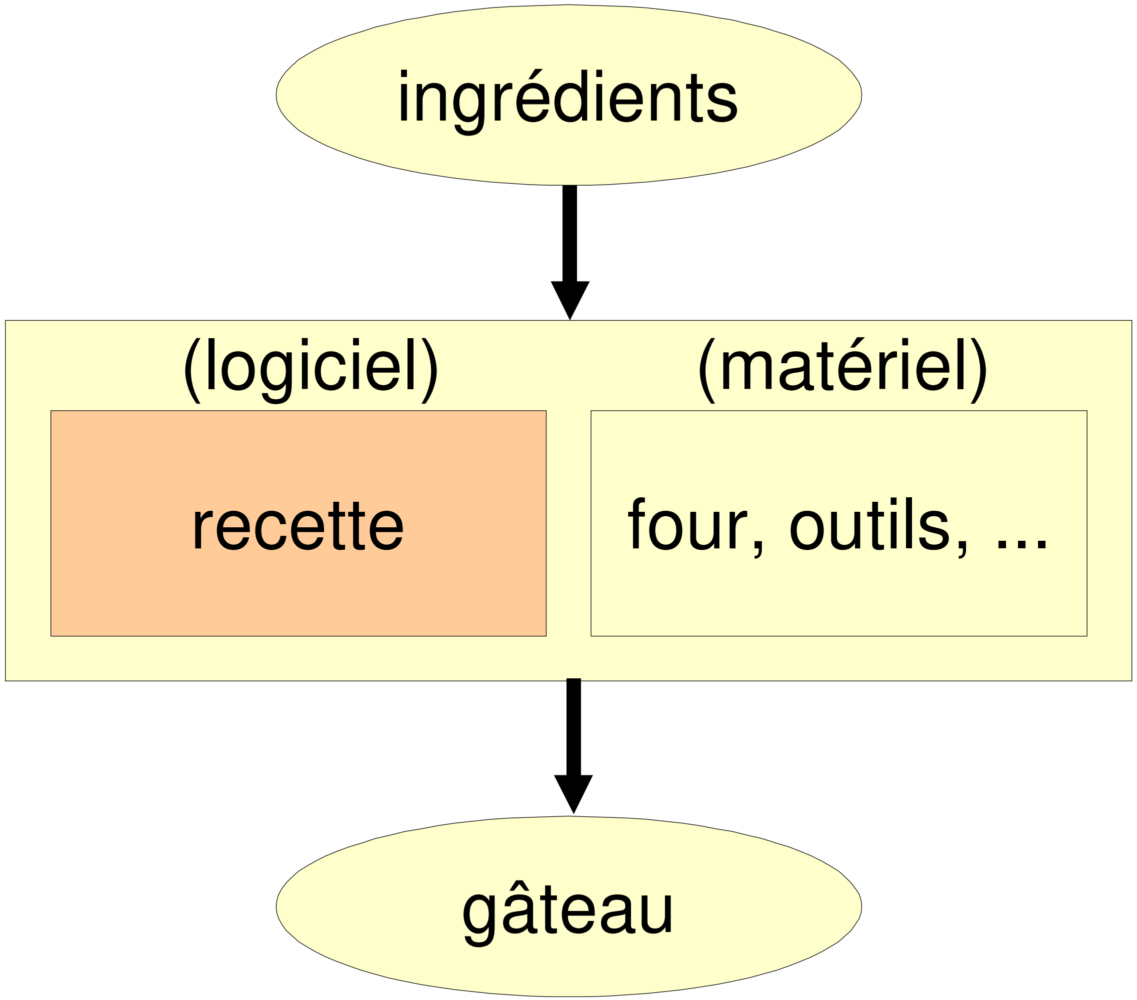
```
class B : public A {
public:
  int vb;
  void dump(){
    A::dump();
    cout<< vb;}
};
```

```
void affiche(A &o) {
  o.dump();
  cout<< endl;
};
```

```
void bffiche(B o) {
  o.dump();
  cout<< endl;
};
```

```
int main() {
  B b; b.va=4; b.vb=3;
  A a; a=b;
  affiche(a);
  affiche(b);
  bffiche(b);
};
```

Algorithmique (I)



Algorithmique (II)

Programmer, c'est ...
décomposer une tache à automatiser sous la forme d'(une suite d')instructions et de données, formulée en utilisant un *langage de programmation*.

Un ***algorithme***, c'est ...
une déscription finie dans un *language univoque quelquonque* d'une procédure composée d'étapes élémentaires.

Algorithmique, c'est ...
l'étude, et l'expertise, la science concernant les algorithmes.

(800 A.D.: Muhammed ibn Musa abu Djafar alChoresmi, *arithmos*)

Algorithmique (III)

Une science?

- méthode pour développer des algorithmes
- algorithmes standards et réutilisables
- analyse de validité
- analyse d'efficacité

<u>limitations des problèmes / solutions:</u>

- complexité
- décidabilité

David Harel: « *Algorithmics – The Spirit of Computing* »
(Addison–Wesley)

Calculabilité

Dans l'informatique théorique, pour démontrer des resultats généraux sur, par exemple, la *décidabilité* ou la *complexité* des algorithmes, on utilise des *modèles de calculs minimalistes* – ne possédant que quelques instructions elementaires – qui représentent des automates programmables.

- ***machines de Turing***
- URM (universal register machines)
- 2–counter machines
- λ***–calcul***
- programmes μ–recursives ...

Etonnamment, ils possèdent *tous la même expressivité* !

Machines de Turing (I)

- Un alphabet (ensemble fini de caractères)

- Une bande infinie, décomposée en cellules contenant des caractères.

ε	ε	ε	A	B	3	X	ε	U	ε	ε

- Une tête de lecture/écriture pouvant
  lire et modifier le caractère courant
  se déplacer vers la gauche (–) ou la droite (+)

- Un ensemble fini d'états internes de la machine

- Une table de transitions (une fonction):
  arguments: état, caractère
  résultat: état, caractère, mouvement

Machines de Turing (II)

- Codage des entrées et des sorties sur la bande
- Initialisation finie
  - partie finie de la bande avec des caractères
  - positionnement de la tête
- Suivre les règles de la table de transitions
  en lisant le caractère courant et
  en utilisant l'état interne courant
- La machine s'arrête quand on arrive a une combinaison d'état / caractère qui n'est pas traitée par la table.

Machines de Turing (III)

Exemple: Définir une machine de Turing qui décide si un numéro entier (codage binaire) est pair ou impair.

Initialisation:
signifie l'entier « 9 ».

ε	ε	ε	1	0	0	1	ε	ε	ε	ε

Algorithme:
1: cherche la fin en bougeant vers la droite
2: trouvé la fin, prêt à décider le resultat (alors: 3. ou 4.)
3: efface la bande en bougeant vers la droite
4: efface la bande en bougeant vers la droite
5: résultat écrit sur la band, bouge à droite, terminé.

Machines de Turing (IV)

Thèse de Church (1936):

Tout algorithme est Turing calculable.

C–à–d, tout algorithme peut être représenté sous la forme d'une table de transitions d'une machine de Turing.

Alors, « machine de Turing » = « programme ».

Théorème fondamentale: « Halting problem »

Il n'est pas décidable de savoir (pas possible de calculer) si une machine Turing s'arrêtera ou pas.

C–à–d: Il n'y a pas d'algorithme **H**

pour ce décider pour chaque programme P.

Machines de Turing (V)

Thèse: Il n’y a pas d’algorithme **H** pour décider pour chaque programme P, si P s’arrêtera ou pas.

Preuve (par contradiction): Soit P un programme.
Soit **H**(P) un programme avec:
« **H**(P)=*vrai* », si P s’arrête, sinon « **H**(P)=*faux* ».
Evidemment, **H**(P) s’arrête pour chaque P.

Soit **T**(P) un programme avec:
« Si **H**(P)=vrai, alors **T**(P) ne s’arrête pas, sinon **T**(P) s’arrête.
»
Observez le comportement du programme **T**(**T**) ...

Fonctions Récursives

Principe: Définir
le résultat d'une fonction appliquée à une valeur
par une opération qui utilise comme argument
le résultat de la fonction appliquée à *une valeur plus petite*.

Exemple: Puissance

$$\mathbf{b^{n} = b * b^{n-1}}$$

$$\mathbf{b^{0} = 1}$$

Exemple: « Fibonacci–numbers » 0 1 1 2 3 5 8 13 21 34 ...

```
f(n) = f(n-1) + f(n-2)
f(1) = 1
f(0) = 0
```

Algorithmes Récursifs

Principe: Ramener le problème à résoudre
à un *sous–problème* correspondant
à une *instance « réduite »* du problème lui–même.

Il est indispensable
de toujours « procéder » avec un *sous*–problème
Il est indispensable
d'avoir une *condition de terminaison*.

Cela se fait, par exemple, en utilisant les entiers
ou des tableaux / listes jusqu'à ce qu'on arrive à la fin.

Algorithmes Itératifs

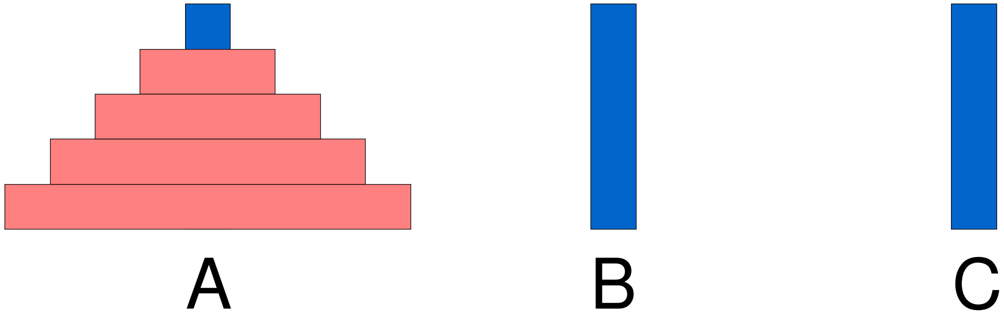
Principe: Résoudre un problème
par l'exécution d'une étape élémentaire:
soit pour un nombre de fois fixe (**`for`**),
soit jusqu'à ce qu'une condition soit satisfaite (**`while`**, **`until`**).

La difference compare avec la récursion ? Plutôt de style.

La récursion se fait par un appel à soi–même.
Normalement, elle utilise le résultat d'un sous–problème.

L'itération, ne traite pas forcément des sous–problèmes, mais
plutôt des problèmes de la même « taille » à chaque ronde.

Exemple: Les Tours de Hanoi



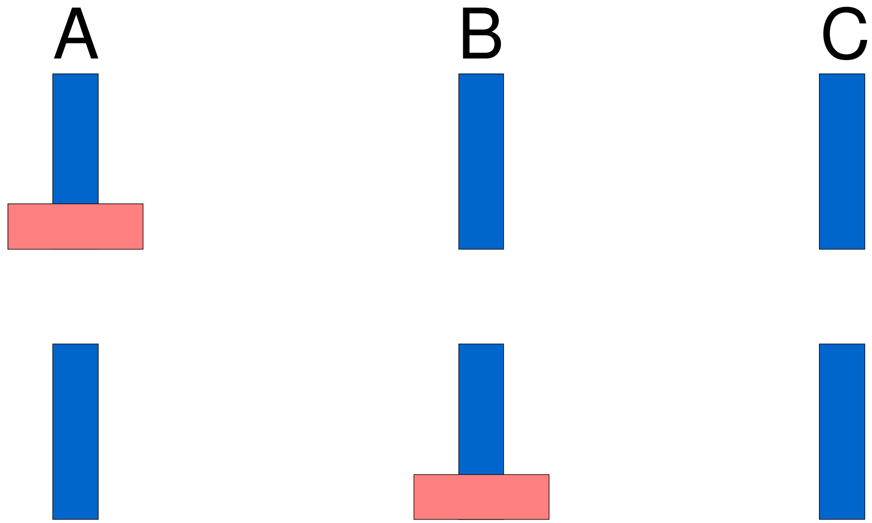
- transférer les anneaux de A à B
- un seul anneau peut être transféré à la fois
- jamais avoir un plus grand au–dessus d'un plus petit ...

- il y a une ***solution récursive pour chaque nombre*** **N** d'anneaux !

Les Tours de Hanoi: Recursion

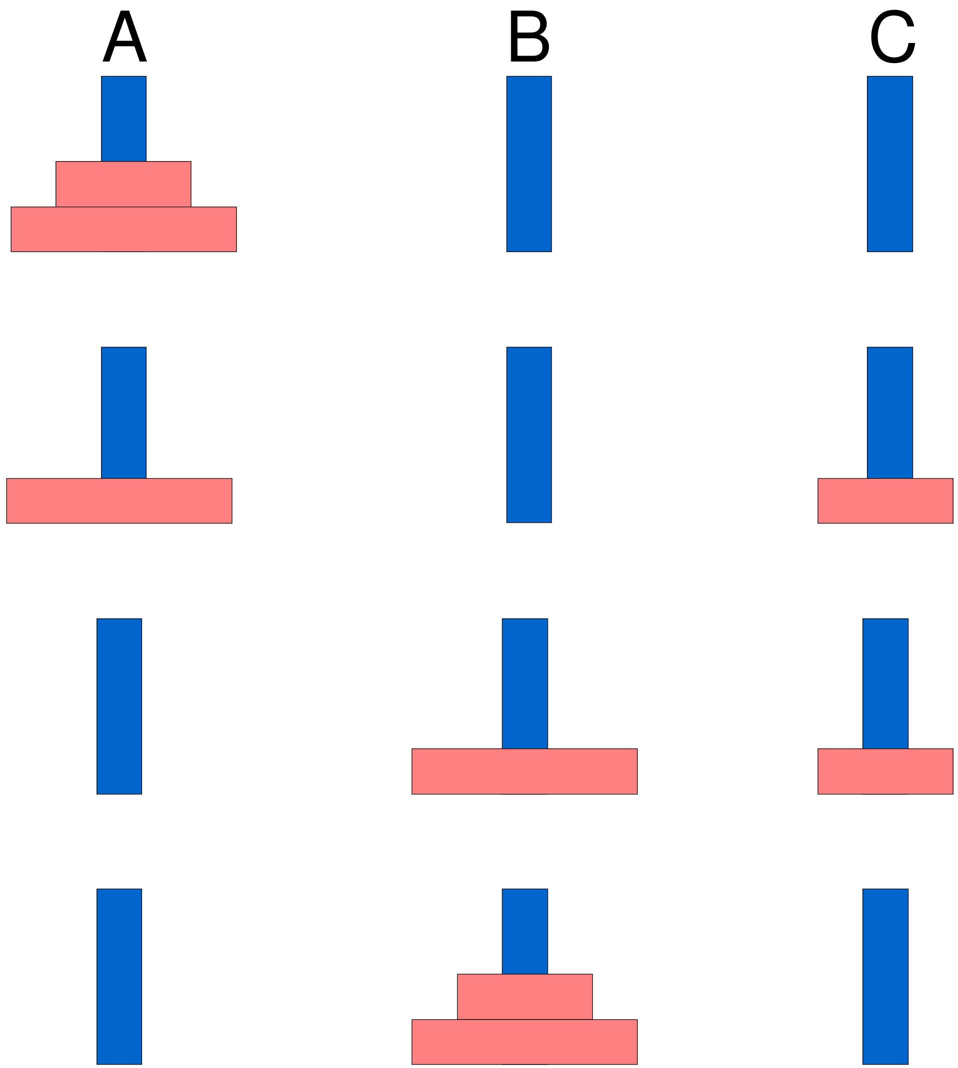
Algorithme h(N,X,Y,Z) pour:
« Transférer N anneaux de X à Y via Z »

h(1,A,B,C):



Dans le cas N=1, on n'utilise pas C !

h(2,A,B,C):

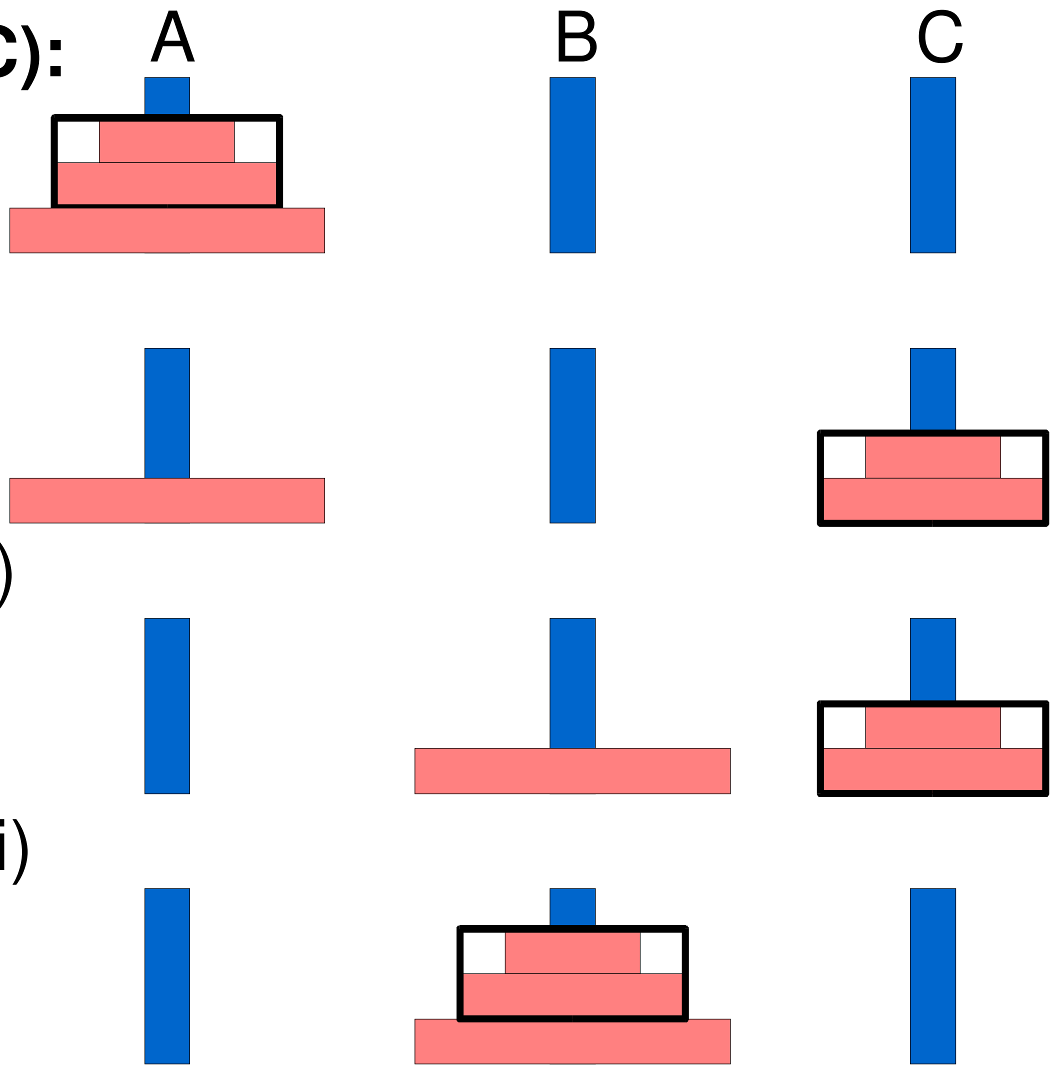


(i) h(1,A,C,B)

(ii) h(1,A,B,C)

(iii) h(1,C,B,A)

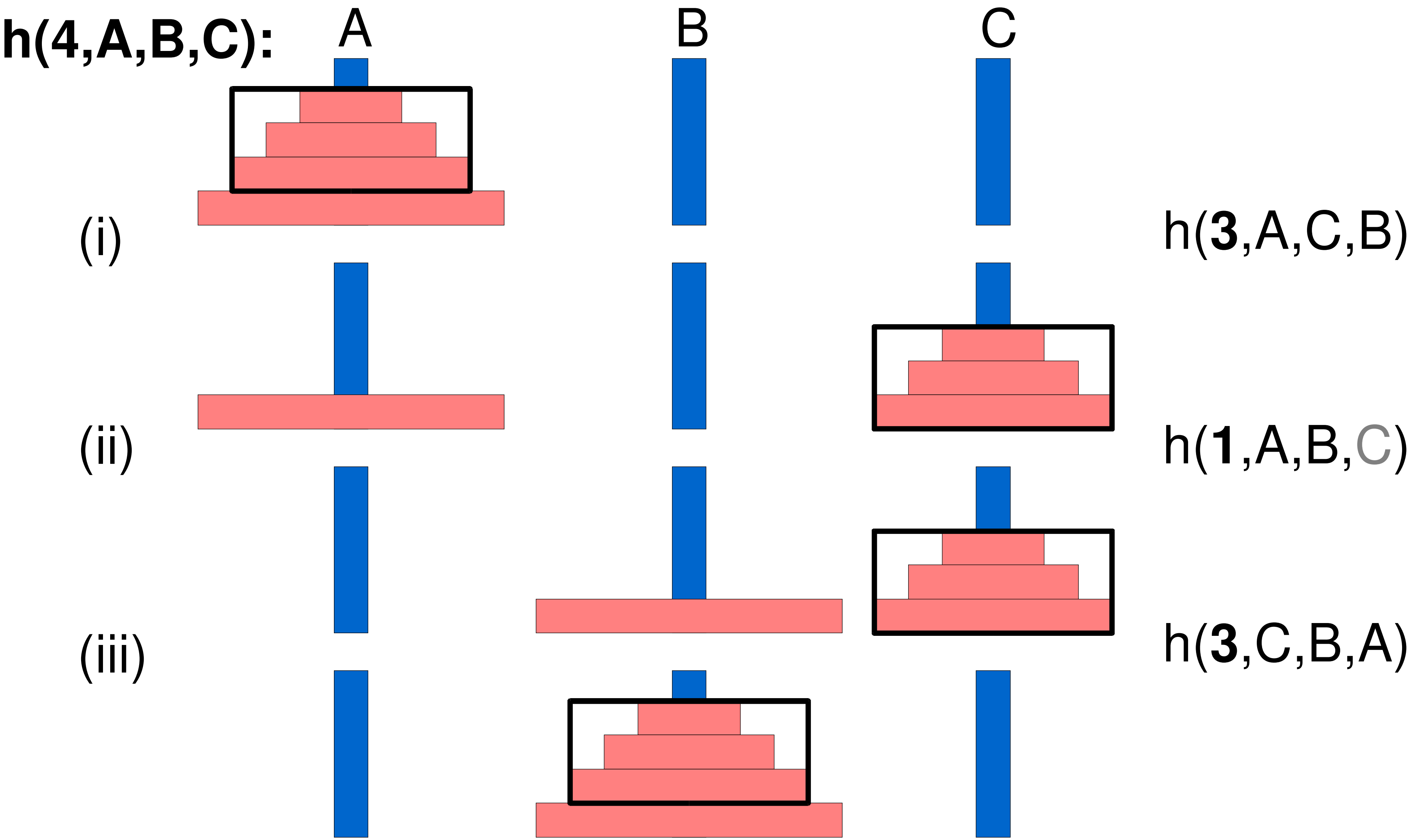
h(3,A,B,C):



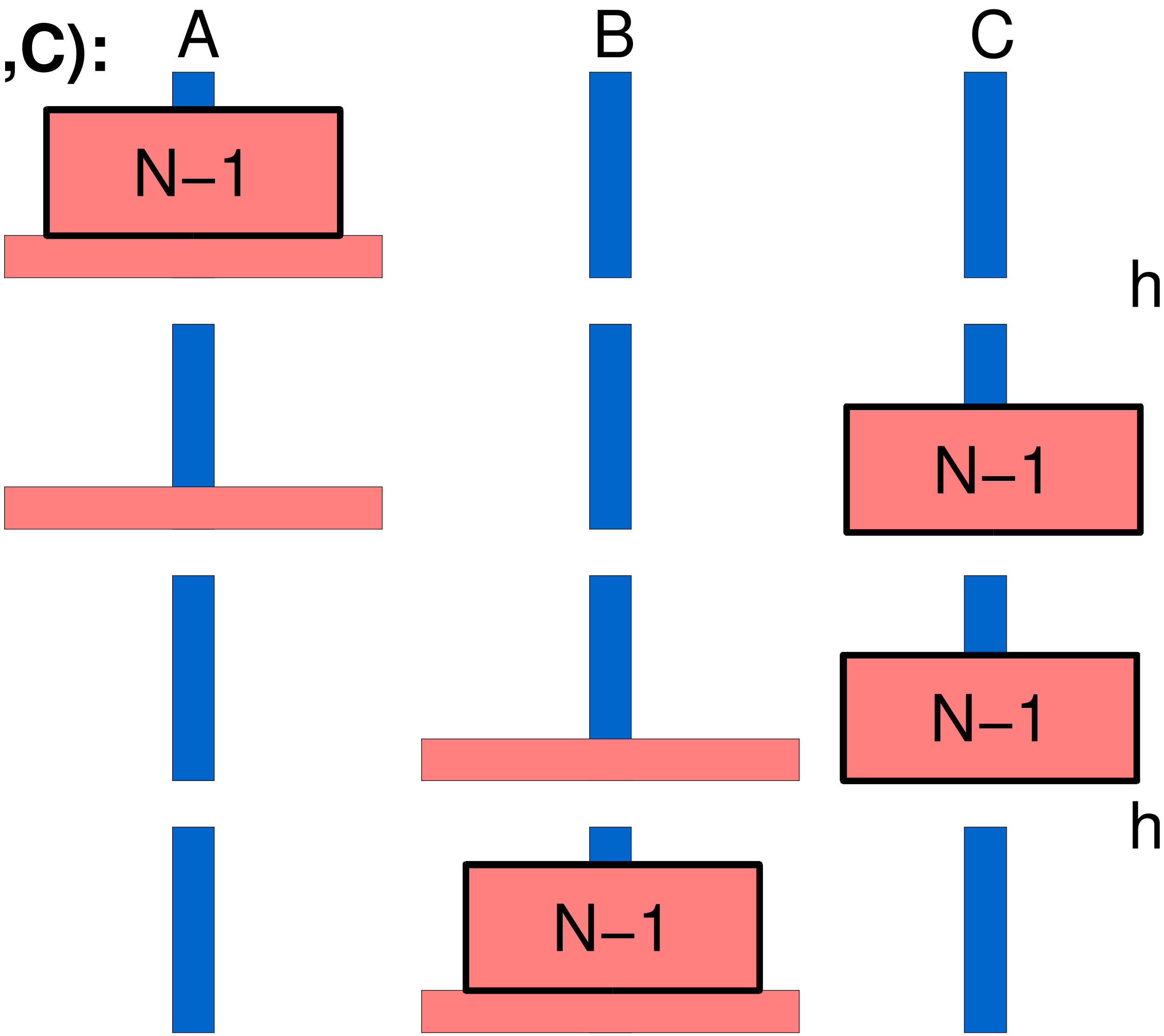
(i) h(**2**,A,C,B)

(ii) h(**1**,A,B,C)

(iii) h(**2**,C,B,A)


h(4,A,B,C):
A
B
C
(i)
h(3,A,C,B)
(ii)
h(1,A,B,C)
(iii)
h(3,C,B,A)

h(N,A,B,C):



(i) h(**N−1**,A,C,B)

(ii) h(**1**,A,B,C)

(iii) h(**N−1**,C,B,A)

Les Tours de Hanoi: Itération

Il y a aussi une ***solution itérative*** !

Arrangez les piquets dans une *cercle*.
Répétez jusqu'à ce que l'état final soit atteint:

1. Transférez *le plus petit anneau au piquet suivant* (dans le sens des aiguilles d'une montre)

2. Faites *le seul coup qui ne bouge pas le plus petit.*

Observez que *l'étape 2 est univoquement déterminée*.

Rappel: Structures Dynamiques

Comment ce faire avec les structures: QC++ §17

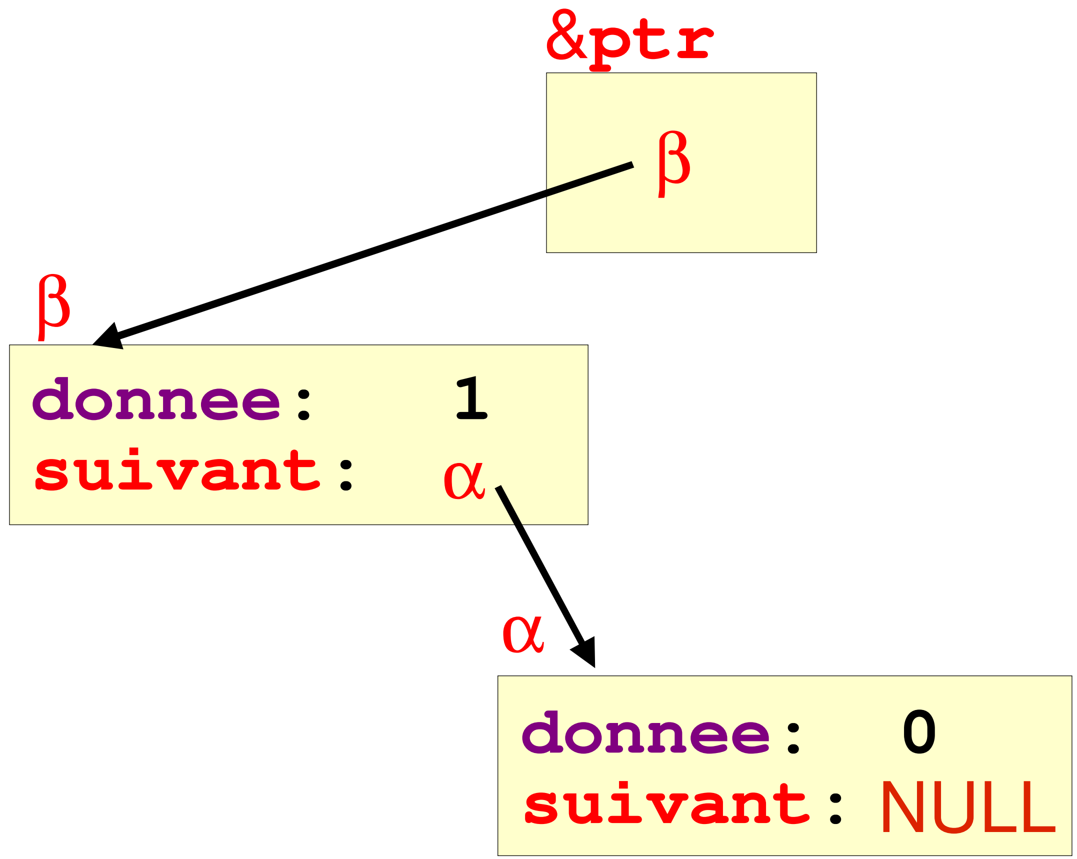
- listes
- arbres
- ...

Exploitation du concept des ***pointeurs*** pour la génération dynamique des "cellules" liées, définies par des ***structures récursive*** qui contiennent des données locales et l'information ou se trouve la suivantes.

Exemple

```
struct Cell {
  int donnee;
  Cell *suivant;
};

Cell c,*ptr;
```



Listes Basées Sur Les Objets

Il est avantageux d'*utiliser*
des classes à la place des structures.

Sécurité: protection des connections entres les cellules.
Modularité: ...
Lisibilité: ...
Réutilisabilité: ...
Extensibilité: ...
Séparation de l'implantation de l'utilisation !

$\rightarrow$ Comment parcourir une liste?

Listes avec Itérateurs

Représentation de tous les composants par des classes:

- les données
- les elements de la liste
  (c–à–d, les conteneur des données)
- la liste même

- ***le code qui fait parcourir la liste***