Notion d'indirection (1)[1]

Illustration 1:

Pourquoi garder une copie des fichiers postscript du cours dans mon répertoire, si je sais où se trouve l'original sur le web ?

☞ pour accéder à l'original au cas où je n'ai pas de connexion internet.

Il n'y a pas vraiment d'autre bonne raison pour conserver une telle copie: cela occupe inutilement de la place dans mon répertoire, et je ne verrai pas les éventuelles mises à jour ou correctifs apportés au cours.

Une meilleure solution est donc que je ne garde que **le *lien* vers** la page web du cours.

1. Adaptation du cours de M. Uwe Nestmann (LAMP-DI-EPFL)

Notion d'indirection (2)

Illustration 2:

Chacun d'entre vous a certainement déjà connu les joies d'un déménagement, et les difficultés pour informer du changement d'adresse toutes les personnes qui vous envoient du courrier.[2]

Une idée qui fait son chemin depuis quelques temps est de fournir à tout le monde une **adresse unique invariante**, attribuée par exemple par la Poste, cette dernière se chargant ensuite d'acheminer le courrier à votre domicile:

en cas de déménagement, il n'y a que la Poste à informer, et vous retrouver immédiatement tout votre courrier.

Les éléments communs à ces deux exemples sont les notions **d'adresses** (URL et de domicile) et **d'indirection**. La réalisation informatique de ces deux notions passe par le concept de ***pointeur***.

2. A part, bien sur, ceux qui envoient des factures !

Pointeurs (1)

Que se passe-t-il lorsque l'on déclare une variable ?

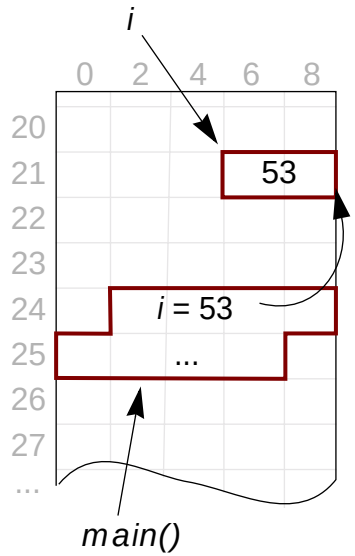
```
int main() {
   int i;
   i = 53;
   ...
}
```

Le petit programme ci-dessus consiste en une définition de fonction (`main`), dans laquelle une variable (`i`) est déclarée, puis se voit affectée la valeur 53.

Lors de la compilation, un espace est réservé dans la mémoire pour représenter la variable `i`; la taille de cet espace est imposée par le type de la variable, mais sa position dans la mémoire est à la discrétion du compilateur.

Il en va de même pour la fonction `main`, dont la transcription en une séquence d'instructions machines est également logée *quelque part* en mémoire.

Lors de l'exécution du programme, les instructions constituant la fonction `main` sont exécutées, et, en particulier, celle qui placera le littéral `53` dans l'espace mémoire représentant la variable `i`.

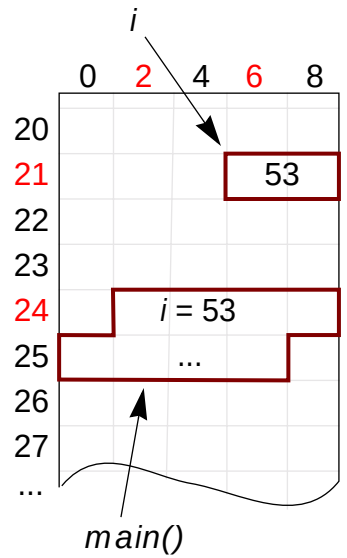


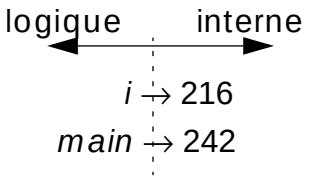
Pointeurs (2)

Cependant, le processeur ne manipule pas directement les identificateurs:

- les identificateurs «*i*» et «*main*» n'existent en fait que pour l'écriture du programme: ce sont des **identificateurs *logiques***, créés par et pour le programmeur.

- **l'identificateur *interne*** d'une élément (variable, fonction, ...) tel qu'il est manipulé par le processeur est tout simplement **l'adresse mémoire** de cet élément; cette adresse est générée par le compilateur[3] (lors de la compilation du programme).





On pourrait donc penser que les adresses mémoire sont des éléments internes sans véritable intérêt pour la programmation. C'est en partie vrai, mais dans certaines situations particulières (cas de collections hétérogène par exemple), il peut s'averer nécessaire de manipuler, au sein du programme, les adressesmémoires de certains éléments. Ceci se fait à l'aide de variables d'un type particulier:

les pointeurs.

3. Et l'éditeur de lien.

Pointeurs (3)

D'une façon générale, un pointeur est une variable permettant de stocker l'adresse mémoire d'un élément du programme

En C++ cependant, la définition d'un pointeur doit de plus comporter l'indication du type des éléments dont il va pouvoir stocker les adresses mémoires.

Ce type sera appelé le type de base du pointeur.

La syntaxe de la déclaration d'un pointeur sera alors:

[`const`*]* *<type de base>*`*` *<id. du pointeur>*`[(` *<valeur initiale>* `)];`

Exemple:

```
int main()
{
   int* ptr;
   ...
}
```

le type d'un pointeur vers des éléments de type `T` est donc de la forme **`T*`**

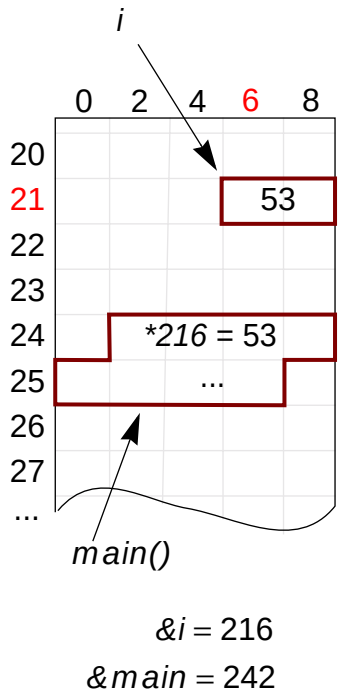
Adresses et référençage

Quelles sont alors les valeurs littérales possibles pour un pointeur ?

En d'autre termes, comment peut-on **représenter des adresses mémoires** ?

En C++, **l'adresse mémoire d'un élément** s'obtient en faisant précéder l'identificateur de cet l'élément par l'opérateur «&»:

ansi, l'expression: `&i`
correspond à l'adresse mémoire de la variable `i`,
et est évalué, dans notre exemple, à `216`.



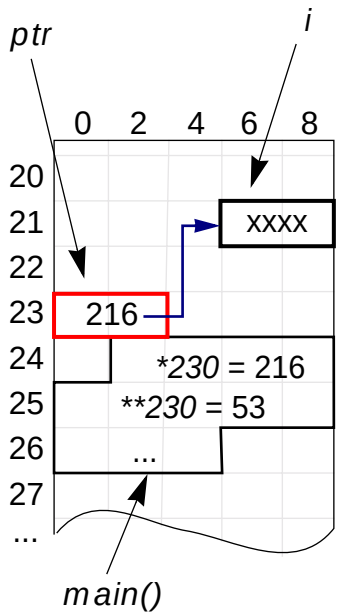
De plus, si `adr` est une adresse mémoire, l'expression **`*adr`** permet de **référencer le contenu de cette adresse**.

Ansi, l'expression: `*&i`
est toujours strictement équivalente à `i`.

Pointeurs (exemple)

Exemple:

```
int main() {
   int i;
   int* ptr(&i);
   *ptr = 53;
   ...
}
```



&i = 216
&ptr = 230
&main = 242

Utilisation des pointeurs (1)

☞ Au moyen de l'opérateur d'affectation «=»:

Un pointeur étant une variable ordinaire, on peut utiliser l'affectation pour changer l'adresse mémoire contenue dans le pointeur (i.e. changer ce vers quoi le pointeur «pointe»).

La valeur qui est affectée peut être une adresse littérale ou obtenue via un autre pointeur, mais dans les deux cas le **type de base doit correspondre à celui du pointeur**.

```
int i; float f;
int* ptr1; int* ptr2;
...
ptr1 = &i;
ptr2 = ptr1;
ptr1 = &f;
```

Comme pour les autres variables, un pointeur doit être initialisé avant d'être utilisé (en lecture), sans quoi il «pointera» vers une zone arbitraire de la mémoire, et accéder à cette zone risque de produire une erreur (détectée par le système d'exploitation).

Une valeur usuelle pour l'initialisation des pointeurs est la valeur `NULL` (équivalente à 0), qui représente un pointeur ne «pointant vers rien».

Utilisation des pointeurs (2)

- Au moyen des opérateurs d'indirection « * » et « -> »:

Le contenu d'un pointeur étant l'adresse d'un élément du programme, il est possible de manipuler cet élément **en passant par le pointeur**:

référencer l'élément désigné par un pointeur se fait en utilisant l'opérateur d'indirection « * » (= *contenu de*):

```
int i(5),j;
int* ptr(&j);
...
*ptr = i; // j = i;
i = (*ptr)+1;
```

On peut utiliser le contenu d'un pointeur indifféremment comme source ou comme cible d'une instruction d'affectation (i.e. à gauche ou à droite du signe « = »)

Un autre opérateur d'indirection pratique lorsque l'on manipule des pointeurs vers des objets (instances) est l'opérateur « -> », qui permet d'accéder directement aux propriétés de l'instance:

```
Cercle* cptr(&c);
cout << (*cptr).surface() << endl;
   // équivalent à
cout << cptr->surface() << endl;
```

Attention aux confusions

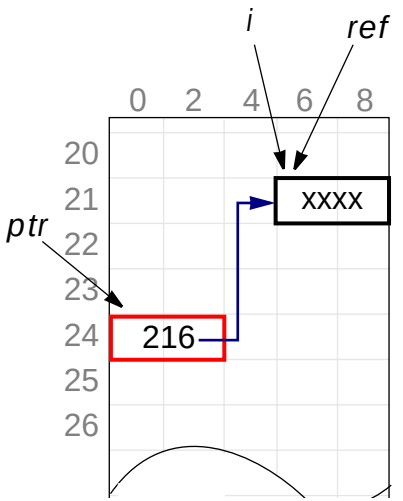
En C++, les symboles « * » et « & » admettent donc des significations différentes suivant le contexte:

- Lors d'une déclaration (de variable ou d'argument), les symboles apparaissant après un type participent au «typage» des éléments. «*» sert à déclarer un **pointeur** et «&» à déclarer une **référence**.

- Dans une instruction, ces symboles apparaîssent devant l'identificateur d'un élément du programme, et désignent alors des opérateurs. «*» permet d'accéder au contenu de l'élément, et «&» permet d'en **désigner l'adresse**.

Pointeurs et références sont deux choses bien différentes:

```
int i;          // la donnée «i» (une variable)
int& ref(i);    // une référence à la donnée «i»
int* ptr(&i);   // un pointeur vers la donnée «i»
```



Une référence est simplement un autre *nom logique* pour désigner une donnée, tandis qu'un pointeur est lui-même une donnée.

Pointeurs vers des pointeurs

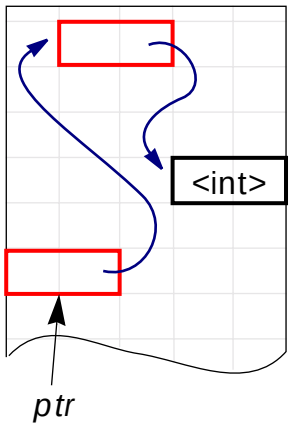
Les pointeurs étant des variables, ils possèdent eux aussi une adresse mémoire.

Par conséquent, un pointeur peut aussi être la «cible» d'un autre pointeur:

La déclaration suivante définit également un pointeur:

```
int** ptr;
```

et correspond à la déclaration du pointeur ptr, qui **contient l'adresse d'un autre pointeur**, **contenant lui l'adresse d'un entie**r.



Allocation de mémoire (1)

Il existe deux manières pour allouer[4] de la mémoire.

- La première consiste à **déclarer des variables**. Les besoins en réservation de mémoire liés aux déclarations de variables sont déterminés à la compilation; on parle alors d'**allocation *statique***

- La seconde manière consiste à allouer de la mémoire **pendant l'exécution du programme**, au moyen d'une instruction spécifique de réservation de mémoire; on parle alors d'**allocation *dynamique***.

Ce type d'allocation est particulièrement utile pour manipuler des données **en nombre variable** (non connu lors de l'écriture du programme), comme c'est le cas pour les tableaux de taille variable (`vector`) ou les chaînes de caractères de type `string`.

Les entités dont la mémoire est allouée de manière dynamique **n'ayant pas de nom logique** (au niveau du programme), elles ne pourront être manipulées qu'au travers de leur **adresse**, par le biais de pointeurs !

4. C'est-à-dire assigner un emplacement mémoire à un élément du programme, et à lui seul.

Allocation de mémoire (2)

C++ met à disposition deux opérateurs pour réaliser les opérations d'**allocation** et de **libération *dynamique*** de mémoire:

- **`new`** : allocation dynamique de mémoire.
  la syntaxe de cet opérateur est:

```
pointeur = new <type>[(<valeur initiale>)];
```

où <*type*> est un type *compatible*[5] avec le type de base du pointeur.

Une zone mémoire (contigüe) dont la taille est fonction du type précisé est réservée dans la mémoire, et la variable `pointeur` se voit affectée l'adresse de cette zone. Si une valeur initiale est précisée, la zone mémoire est initialisée avec cette valeur.

- **`delete`** : libération dynamique de mémoire.
  la syntaxe de cet opérateur est:

```
delete pointeur;
```

Si `pointeur` est différent de `NULL`, la zone mémoire pointée est libérée et `pointeur` est réinitialisé à `NULL`. Naturellement, une erreur se produit si `pointeur` référencie une zone mémoire qui n'a pas été allouée de manière dynamique.

5. En règle générale le même type, mais dans le cas de la POO, <*type*> peut être un sous-type du type de base du pointeur.

Allocation de mémoire (3)

Exemple:

```
int *ptr = new int;
*ptr = 20;
cout << *ptr << endl;
delete ptr;
```

Exemple d'initialisation de la zone mémoire allouée:

```
int* ptr = new int(20);
```

ou

```
int* ptr(new int(20));
```

Lorsque l'on travaille avec des pointeurs vers des classes (i.e. le type de base est une classe), l'allocation dynamique par `new` conduit à l'invocation [automatique] d'un constructeur pour la donnée allouée (le constructeur par défaut si aucune valeur d'initialisation n'est précisée), et symétriquement, la libération de la donnée par `delete` conduit à l'invocation du destructeur défini pour cette donnée.[6]

6. D'où l'intérêt à systématiquement déclarer *virtuel*s les destructeurs...

Allocation de mémoire (4)

Lors de l'utilisation du mécanisme d'allocation dynamique de mémoire dans des classes, reste à décider qui, du concepteur de la classe ou de son utilisateur, aura la responsabilité d'allouer et de libérer la mémoire pour les éléments créés dynamiquement.

Généralement, la création des instances sera laissée à la charge de l'utilisateur, et en particulier l'allocation de leur espace mémoire.[7]

En ce qui concerne la libération, les choses sont plus nuancées, et la politique adoptée par le concepteur de la classe (prendre ou non en charge la libération) devra obligatoirement être explicitée.

Cependant, une utilisation prudente de l'allocation dynamique plaide plutôt en faveur d'une libération à la charge du concepteur de la classe[8]

7. Cette politique permet d'éviter une copie (et donc une destruction) des instances.
8. Les arguments en faveur d'une libération effectuée par la classe contenant la collection sont d'une part le respect de la pratique de la programmation objet, qui veut que l'utilisateur connaisse le contexte dans lequel une instance est crée, mais pas celui dans lequel l'instance est détruite, et d'autre part un allègement des tâches à réaliser pour pouvoir utiliser la classe (ce qui permet également de limiter le risque de «fuites mémoire»).
   L'autre politique a également ses avantages, en particulier celui de laisser à l'utilisateur le choix pour effectivement réaliser une allocation dynamique (il se peut en effet que l'utilisation de variables statiques soit suffisante), et surtout permettre la réutilisation des instances dans un autre contexte (respect d'une pratique de programmation C qui veut que l'utilisateur ait la responsabilité de la gestion de la mémoire...).

Collection hétérogène (1)

Comme nous l'avons vu précedemment, une condition nécessaire pour que le polymorphisme [d'inclusion] puisse réellement être utilisé[34] est que les traitements génériques effectués portent sur des instances effectives, et non pas sur des copies de ces instances.

Pour cela, les fonctions de traitement génériques doivent admettre des arguments **passés par *référence***.

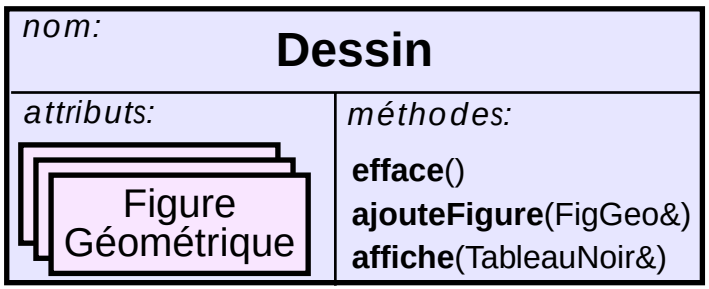
On peut alors se demander comment réaliser un traitement générique, portant non pas sur une instance à la fois, mais sur un **ensemble d'instances** ?

34. Plus précisément, pour que la résolution dynamique des liens puisse être faite.

Collection hétérogène (2)

Exemple:

On souhaite réaliser une classe `Dessin` qui permette de regrouper,
au sein d'une même entité, un ensemble quelconque de *figures géométriques*
(un *dessin* est donc une ***aggrégation*** de *figures géométriques*).



Se pose alors l'épineux problème suivant:

Comment définir dans la classe `Dessin`
une collection de figures géométriques ?

Collection hétérogène (3)

L'idée la plus naturelle est d'utiliser un `vector`.
Mais comme la classe `FigureGeometrique` est une classe abstraite (donc non instanciable), il n'est pas possible de créer un `vector` avec cette classe comme type de base.

De plus, même si cette classe n'était pas abstraite, une telle solution ne serait pas viable, puisqu'il serait impossible de réaliser correctement l'affichage des différentes figures (résolution dynamique des invocations de la méthode virtuelle de dessin des figures).

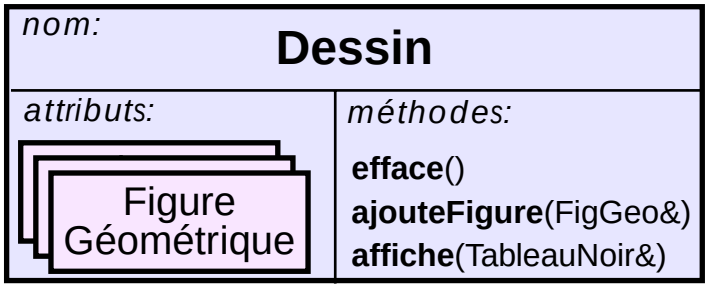
Une meilleure idée serait donc de créer un `vector` de ***références*** vers des instances de type `FigureGeometrique`...

... mais ce n'est pas possible non plus, puisqu'une *référence* ne peut être réassignée...

En C++, l'unique possibilité pour réaliser une telle collection hétérogène d'instances passe par l'utilisation d'un nouveau concept: le ***pointeur*** !

Collection hétérogène: solution (4)

La réalisation de la classe `Dessin` nécessite quelques adaptations préalables.



Il faut en premier lieu enrichir notre modèle de figures géométriques, de manière à pouvoir positionner dans l'espace (engendré par le dessin) chacune de ces figures.

```
class FigureGeometrique {
public:
 virtual void
 dessine(TableauNoir&) const = 0;
};
```

Un dessin étant destiné à être représenté de manière graphique (affiché sur l'écran, imprimé sur un feuille, etc), il faudra ensuite munir chaque classe particulière d'une fonction d'affichage, réalisant le tracé de la figure sur un support (supposé connu) passé en paramètre, par exemple une référence vers une instance de `TableauNoir`.

```
class Cercle:public FigureFermee{
public:
 ...
 void dessine(TableauNoir&) const;
protected:
 double rayon;
 double cx,cy; // les coord. du centre
};
```

Collection hétérogène: solution (5)

Finalement, la classe `Dessin` peut être définie, en implémentant la collection de figures au moyen d'un vecteur de pointeurs vers des instances de type `FigureGeometrique`:

```
vector<FigureGeometrique*> collection;
```

Seul le pointeur est copié dans la structure, et pas l'instance pointée; cependant, par le biais du pointeur, c'est bien l'instance originelle qui est manipulée,i.e l'instance pour laquelle la résolution dynamique des liens est possible.

```
typedef FigureGeometrique FG;


class Dessin {
public:
   void ajouteFigure(FG& fig);
   void affiche(Tableau& dsp) const;
   void efface();

protected:
   vector<FG*> figures;
};
```

```
void Dessin::ajouteFigure(FG& fig) {
   figures.push_back(&fig);
}
void Dessin::efface() {
   figures.clear();
}

void
Dessin::affiche(Tableau& dsp) const {
   for(int f(0);f<figures.size();++f)
     figures[f]->dessine(dsp);
}
```

Collection hétérogène (6)

Bien que fonctionnelle, notre classe Dessin
n'est pas encore tout à fait satisfaisante

De part sa conception, elle impose que les instances 'figures' constituant le dessin aient une durée de vie au moins égale à celle du dessin (ou jusqu'à ce qu'un effacement soit réalisé). La classe ne manipulant que des pointeurs, il est en effet primordial que **les éléments pointés existent au moins aussi lontemps que leurs pointeurs**.

Et ceci n'est pas garanti, en particulier, l'utilisation suivante conduit à une catastrophe:

```
void creeCercle(Dessin& img) {
   int r,cx,cy;
   // saisie/calcul des paramètres
   Cercle c(r,cx,cy);
   img.ajouteFigure(c);
}
...

int main() {
   Dessin d;
   creeCercle(d);
   d.affiche(tableauNoir); ...
```

La fonction `creeCercle` ajoute un nouveau cercle au dessin img, mais par le biais d'un pointeur vers une variable **locale** à la fonction.

En d'autre terme, une fois l'exécution de `creeCercle` terminée, la variable locale `c` est détruite (n'existe plus), et son emplacement mémoire peut alors être utilisé pour une autre variable locale... alors que le pointeur stocké dans le dessin existe toujours... l'affichage du dessin conduira donc au mieux à un arrêt du programme, et au pire à l'affichage d'un cercle qui n'est pas du tout celui défini par la fonction `creeCercle`.

Collection hétérogène (7)

Dès que l'on manipule des pointeurs, se pose en fait le problème général de **l'intégrité** des données d'un programme (en particulier celui de l'existence des instances référencées)

Malheureusement, il n'existe pas de moyen absolu, et il faut donc se contenter le plsu souvent de bonnes pratiques de programmation...

Plus spécifiquement, dans notre exemple, comment peut-on rendre utilisable la classe `Dessin`, sans obliger le programmeur à assurer l'existence de toutes les instances de figures en les déclarant globales, et en les stockant au sein de plusieurs vecteurs (un par type de figure) si leur nombre n'est pas connu à l'avance ?

En C++, une solution à ce problème est fournie par le concept ***d'allocation dynamique*** !

Collection hétérogène (8)

En adoptant la politique d'une allocation à la charge de l'utilisateur et d'une libération prise en charge par le concepteur, notre classe `Dessin` doit alors implémenter un destructeur,

chargé de libérer, en fin de vie du dessin, la mémoire occupée par les éléments de la collection. Cette libération mémoire devant également être réalisée lorsque le dessin est 'effacé', il est possible d'implémenter la classe ainsi:

```
...

class Dessin {
public:
   virtual ~Dessin();
   void ajouteFigure(FGPtr fig);[35]
   void affiche(Tableau& dsp) const;
   void efface();

protected:
   vector<FGPtr> figures;
};
```

```
Dessin::~Dessin() {
   erase();
}

void Dessin::ajouteFigure(FGPtr fig){
   figures.push_back(fig);
}

void Dessin::erase() {
   for(int i(0);i<figures.size();++i)
      delete figures[i];
   figures.clear();
```

35. Le changement de prototype n'est pas obligatoire, mais il permet d'indiquer clairement que les éléments manipulés par la classe sont des pointeurs.

Collection hétérogène (9)

Exemple d'utilisation:

```
void creeCercle(Dessin& img) {
   int r,cx,cy;
   // saisie/calcul des paramètres
   img.ajouteFigure(new Cercle(r,cx,cy));
}
void creeRectangle(...) {...}

int main() {
   Dessin d;
   creeCercle(d);
   d.affiche(tableauNoir); ...
}
```

création dynamique d'une instance de la classe Cercle

stockage d'un pointeur vers cette instance dans l'objet dessin

affichage du dessin

les instances le composant étant allouées dynamiquement, elles existent toujours dans la mémoire

destruction du dessin, et donc destruction de toutes les instances le composant, et libération de la mémoire y-relative

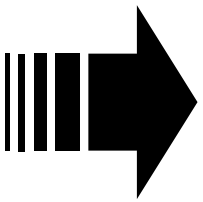
Collection hétérogène (10)

Notre classe `Dessin` semble être utilisable,
mais elle recèle en fait encore quelques surprises...

Que se passe-t-il si l'on exécute le programme suivant ?

```
void f(Dessin img) {
   img.ajouteFigure(new Carre(2,3,-5));
   img.affiche(unAutreTableauNoir);
}

int main() {
   Dessin d;
   d.ajouteFigure(new Cercle(5,0,0));
   f(d);
   d.affiche(tableauNoir);
};
```



- segmentation fault !
- ou bus error !
- ou memory access violation !
- ou ...

se produit après l'affichage du dessin
sur la fenêtre de «l'autre tableau noir»....

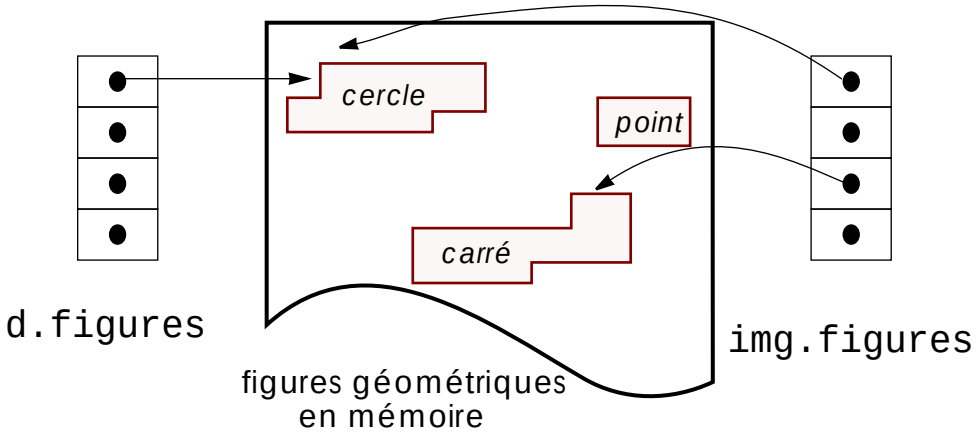
POURQUOI ?

Collection hétérogène (11)

La raison est simple:
l'appel à la fonction `f` provoque **la copie** de l'instance `d` !

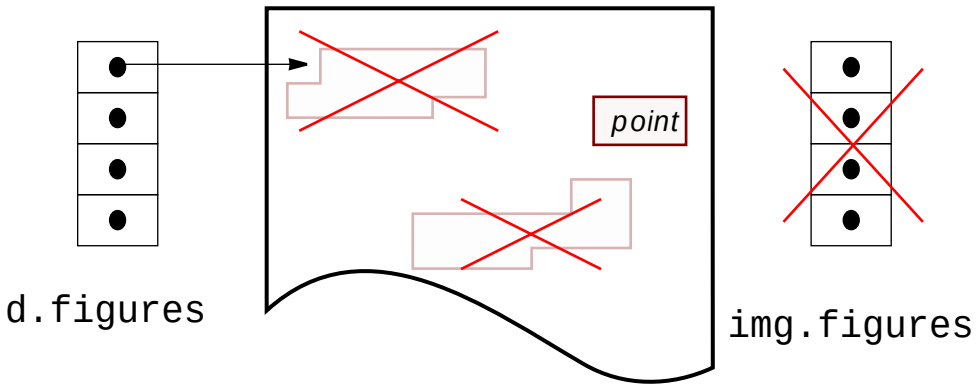
Pour réaliser cette copie, le compilateur utilise le constructeur de copie qu'il a automatiquement généré...

... et qui réalise une copie appelée **copie de surface**: le tableau de pointeurs est bien dupliqué, mais pas les éléments pointés.



A la fin de l'exécution de la fonction `f`, l'instance `img` est détruite...

... le destructeur de la classe `Dessin` est donc invoqué, et il s'appliquera du mieux qu'il peut pour détruire les figures de la collection, et restituer l'espace mémoire !



Collection hétérogène (12)

Pour être vraiment utilisable,
une classe qui mobilise une ressource (typiquement
une allocation de mémoire) ne peut se contenter des constructeurs et
destructeurs générés par le compilateur.

Le concepteur de la classe devra fournir ses propres version
du **constructeur de copie**, du **destructeur**, de **l'opérateur d'affectation**,
et parfois de **l'opérateur d'égalité** « == », afin de prendre en compte
la structure ***profonde*** de sa classe.

Réaliser une copie profonde (v.s. de surface) consiste à dupliquer les
éléments référencés par les pointeurs. Cette tâche ne peut toutefois
être réalisée par notre seule collection hétérogène, puisqu'elle
ne connait pas le type réel des instances qu'elle manipule.

Il faut donc définir une nouvelle méthode virtuelle au niveau
de la classe des `FiguresGeometrique`, réalisant une copie de l'instance
pour laquelle la méthode est invoquée, en utilisant l'allocation dynamique.

Collection hétérogène - final (13)

```
class FigureGeometrique {
public:
 virtual void
 dessine(TableauNoir&) const = 0;

 virtual FigureGeometrique*
 copy() const = 0;
};
typedef FigureGeometrique FG;

class FigureFermee: public FG
{ ... }

class Cercle: public FigureFermee {
public:
 void dessine(TableauNoir&) const;
 FG* copy() const {
  return (new Cercle(*this));
 }
 ...
};
```

```
class Dessin {
public:
   Dessin(const Dessin& d);
   virtual ~Dessin();
   Dessin& operator=(Dessin&);
   ...
};

Dessin::Dessin(const Dessin& d) {
   for (int i(0);
        i<d.figures.size(); ++i)
   {
      figures.push_back(
        d.figures[i]->copy());
   }
}

...
```