Plan de cette partie du cours

- Programmation dirigée par les types
- *Smart pointers*
- Retour sur les *templates*
- *Multi-dispatching*
- Constructeurs virtuels

Programmation dirigée par les types (un vecteur)

Le vecteur mémorise le descripteur de fichier, et l'indice maximum des éléments initialisés :

```
class Vecteur {
public:
 Vecteur() {
  fd = open (tempnam(0), …);
  max = -1;
 }
 …
private:
 int fd;  // n° du fichier
 int max; // max indice élément initialisé
};
```

Programmation dirigée par les types (exécution retardée)

On peut faire, t[i] = x, et aussi x = t[i]

- La méthode operator[] ne sait pas dans quel contexte elle est appelée
  - Donc elle ne sait pas s'il faut lire ou écrire le fichier
  - Elle doit différer son travail, ses opérandes doivent être mémorisés

```
class Vecteur {
 friend struct index {
  int x; Vecteur& v;
  index(int i, Vecteur& a) : x(i), v(a) {}
 } ;
 index operator[] (int x) {
  return index (x, *this);
 }
 …
```

Programmation dirigée par les types (exécution retardée)

Le contexte de l'opérateur [] décide de l'opération à appliquer au vecteur (et éventuellement au fichier)

- à droite d'un =

```
friend struct index {
operator int() {
 if (x > v.max) return 0;
 int vi;
 lseek (v.fd, x * sizeof(int), 0);
 read (v.fd, &vi, sizeof(int));
 return vi;
}
```

```
int k = v[i]

int = index
      ↓
      int
```

Programmation dirigée par les types (exécution retardée)

- à gauche d'un =

```
friend struct index {
int operator= (int k) {
 if (x > v.max) v.max = x;
 lseek (v.fd, x * sizeof(int), 0);
 write (v.fd, &k, sizeof(int));
 return k;
}
```

`v[i] = k`

index::operator=(int)

- et, puisque la sémantique de = sur des index n'est pas celle donnée par défaut

```
int operator=(const index& x) {
 return *this = int(x);
}
```

`v[i] = v[j]`

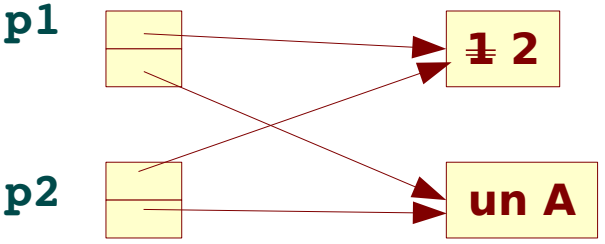
index::operator=(index)

Pointeurs *intelligents*
(classe et constructeur)

En fait, ce sont des pointeurs capables d'éviter (dans certains cas) un *garbage collector*

```
template <typename T>
class SmartPointer {
 int* refnb;
 T*   val;
public :
 SmartPointer (T* v) {
  refnb = new int(1);
  val = v;  // peut-être prévoir le cas v = 0
};
```

```
SmartPointer<A> p1 = new A();
SmartPointer<A> p2 = p1;
```



Pointeurs *intelligents* *(copie, affectation, destruction)*

```
template <typename T>
class SmartPointer {
 ...
 SmartPointer (const SmartPointer& p) {
  share(p);
 }
 const SmartPointer& operator=
                              (const SmartPointer& p) {
  if (val != p.val) {
   unshare(); share(p);
  }
  return *this;
 }
 ~SmartPointer() { unshare(); }
```

Pointeurs *intelligents*
(partage et oubli)

```
template <typename T>
class SmartPointer {
 ...
protected :
 void share (const SmartPointer& p) {
  refnb = p.refnb; val = p.val; *refnb++;
 }

 void unshare() {
  *refnb--;
  if (*refnb == 0) { delete val; delete refnb; }
 }
};
```

Retour sur les templates
(multiples instanciations identiques)

On a vu (pour simplifier) qu'une solution était d'inclure le .cc dans le .h

- Cette solution a un gros inconvénient : le code instancié est présent dans le code produit pour chaque unité de compilation :

Le code de A<string>::M se trouve dans b.o, **et aussi dans** c.o

b.cc
```
#include "a.h"
A<double> ad;
A<string> as;
```

c.cc
```
#include "a.h"
A<int> ai;
A<string> as;
```

a.h
```
template <typename T>
class A {
  void M (T);
  …
};
#include "a.cc"
```

```
template <typename T>
void A::M(T k) { … }
```
a.cc

ne peut pas être compilé sans être instancié

Retour sur les templates
(rendre uniques les instanciations identiques)

Plusieurs solutions sont possibles

- L'éditeur de liens est capable de reconnaître des portions identiques de code, et de les fusionner
- On utilise un *template repositery* : le compilateur range les templates et leurs instanciations dans ce dépot : à l'edition de lien, on commence pas compiler toutes les instanciations
- Dans les 2 cas, rien n'est demandé à l'utilisateur, mais besoin d'un éditeur de liens spécialisé

Retour sur les templates
(solution g++ pour instanciations uniques)

Plusieurs solutions sont offertes

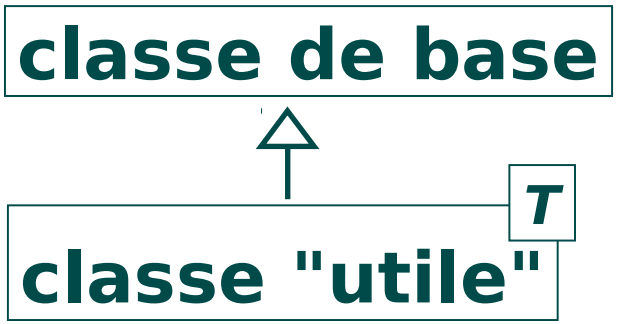
1. Compiler le code template (donc le a.cc précédent) avec l'option -frepo qui produit un fichier .rpo ; faire l'édition de lien avec les .o et les .rpo ... **du moins théoriquement**
2. Compiler b.cc et c.cc avec l'option -fno-implicit-templates, créer un fichier qui contient toutes les instanciations attendues, le compiler et relier tous les .o

ia.cc

```
#include "a.cc"
template class A<int>;
template class A<double>;
template class A<string>;
```

Limiter les coûts de la généricité

- Toutes les méthodes des classes génériques ne sont pas génériques par essence
  - exemple: se positionner sur le n-ième élément d'une liste est indépendant du type des éléments
  - principes d'architecture:

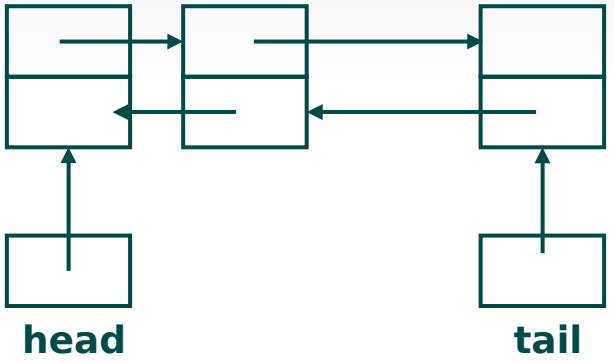


Limiter les coûts de la généricité : exemple avec une liste

- Liste de base: opérations sur la structure
  - cellule de base

```
class BaseList {
protected:
 struct BaseCell {
  virtual ~BaseCell() { }
  BaseCell *next, *prec;
 } *head, *tail;

 BaseList() : head(0), tail(0) {}
public:
 …
```



Limiter les coûts de la généricité : exemple avec une liste

Opérations de base

```
class BaseList {
protected:
 ~BaseList() { suppress_all(); }
 void prepend (BaseCell*);
 void append (BaseCell*);
 void insert_after (BaseCell*, BaseCell*);
 void suppress_head();
 void suppress_tail();
 void suppress (BaseCell*);
 BaseCell* get (int);
 void suppress_all ();
 …
};
```

Limiter les coûts de la généricité : exemple avec une liste

Implémentation des opérations de base

```
void BaseList::prepend (BaseCell* c) {
 if (tail == 0) tail = c;
 c->next = head; c->prec = 0; head = c;
}

void BaseList::suppress_tail () {
 BaseCell* c = tail;
 tail = tail->prec;
 tail->next = 0;
 delete c;
}
…
```

Ces méthodes peuvent être *inline*

Limiter les coûts de la généricité : exemple avec une liste

Implémentation des opérations de base

```
void BaseList::suppress_all() {
 while (head != 0) {
  BaseCell* c = head; head = head->next;
  delete c;
 }
}

BaseList::BaseCell* BaseList::get (int x) {
 BaseCell* c = head;
 while (x != 0 && c != 0) {--x; c = c->next;}
  if (c == 0) throw BadIndex();
  return c;
}
```

Ces méthodes ne peuvent pas raisonnablement être *inline*

Limiter les coûts de la généricité : exemple avec une liste

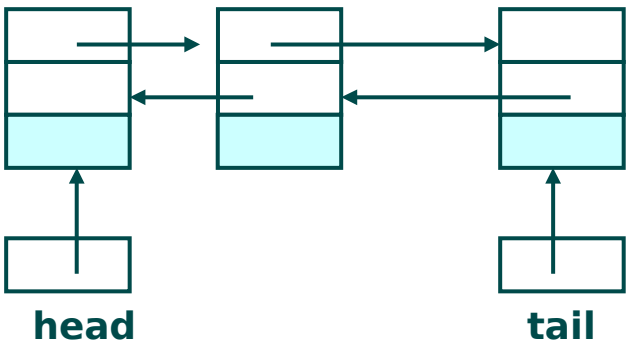
Liste générique

- cellule avec élément

```
template <typename T>
class List : protected BaseList {
protected:
 struct Cell : protected BaseCell {
  Cell(const T& e) : elem (e) {}
  T elem;
 };

public:
 …
};
```

On demande au type T uniquement un constructeur de copie



Limiter les coûts de la généricité : exemple avec une liste

Opérations de liste

```
template <typename T>
class List : protected BaseList {
 …
public:
 List<T>& prepend(const T& e) {
  BaseList::prepend (new Cell(e));
  return *this;
 }
 List<T>& append(const T& e) {
  BaseList::append (new Cell(e));
  return *this;
 }
 …
```

Pas nécessaire de définir de constructeur (autre que de copie), ni de destructeur

Limiter les coûts de la généricité : exemple avec une liste

Opérations de liste

```
template <typename T>
class List : protected BaseList {
 …
public:
 …
 const T& operator[] (int x) const {
  BaseCell* c = get(x);
  return (static_cast<Cell<T>*>(c)->elem;
 }

 void suppress (int x) {
  suppress_after (x > 0 ? get (x-1) : 0);
 }
```

Limiter les coûts de la généricité : exemple avec une liste

Copie et affectation

```
template <typename T>
class List : protected BaseList {
 …
public:
 …
 List (const List&);
 const List& operator= (const List&)
```

- Comment copier les éléments sans écrire de boucle dans la classe générique ?

Limiter les coûts de la généricité : exemple avec une liste

Retour sur la cellule de base

```
class BaseList {
protected:
 struct BaseCell {
  BaseCell *next, *prev;

  virtual BaseCell* clone () const = 0;

 }* head;

 void copy (const BaseList&);
 …
```

Limiter les coûts de la généricité : exemple avec une liste

Copie d'une liste de base

```
void BaseList::copy (const BaseList& l) {
 if (l.head == 0)
  head = 0;
 else {
  BaseCell* c1 = l.head;
  BaseCell* c0 = c1->clone();
  head = c0; c1 = c1->next;
  while (c1 != 0) {
   BaseCell* c = c1->clone();
   c0->next = c; c0 = c; c1 = c1->next;
  }
  c0->next = 0;
 }
```

Noter que l'utilisation de la liaison dynamique pour clone est coûteuse

Limiter les coûts de la généricité : exemple avec une liste

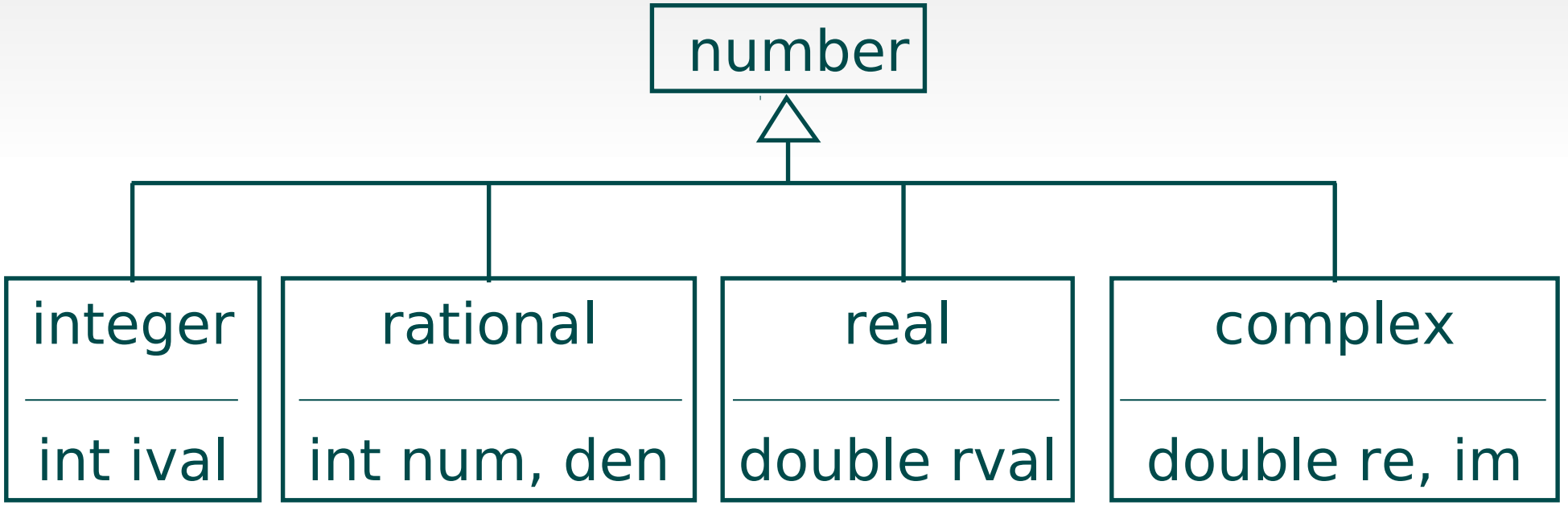
D'où copie et affectation

```
template <typename T>
class List : protected BaseList {
protected:
 struct Cell : protected BaseCell {
  …
  BaseCell* clone() const { return new Cell(elem); }
 };
public:
 List (const List& l) { copy (l);}
 const List& operator= (const List& l) {
  if (this!= &l) { suppress_all(); copy(l); }
  return *this;
 }
```

~~Éviter la suppression~~
(donc copier dans les cellules existantes)
impose que l'affectation soit possible sur T

Multi dispatching

Soit le schéma de classes



- comment réaliser
  `number +operator (number, number);`

Multi dispatching : classe gérante - classe corps

```
class number {
protected:
 class repr {
 public:
  virtual repr* clone() const = 0;
 …
 }* val;

 number (repr* v) : val(v) {}

public:
 …
```

Multi dispatching : classe gérante - classe corps

Le gérant construit le corps

```
class number {
 …
public:
 number (const number& n) {
  val = n.val->clone();
 }

 const number& operator= (const number& n) {
  if (val != n.val) {
   delete val;
   val = n.val->clone();
  }
  return *this;
 }
```

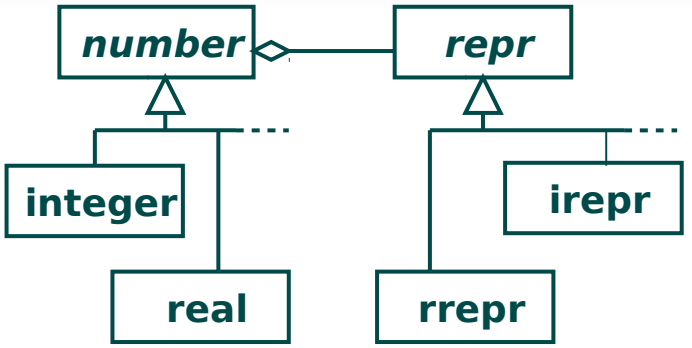
on pourrait de plus implémenter un compte de référence sur les représentations

Multi dispatching : classe gérante - classe corps

Spécialisation des gérants et des corps

```
class integer : public number {
protected:
 class irepr : public repr {
 public:
  irepr (int v) : ival(v) {}
  …
 protected:
  int ival;
 };
public:
 integer (int v = 0) : number (new irepr(v)) {}
 …
};
```

(bridge pattern)



Multi dispatching : classe gérante – 1er dispatching

Le gérant *dispatche* sur le corps

```
class number {
 …
 class repr {
  virtual repr* add_to (const repr*) const = 0;
  …
 };

 repr* add_to (const number& n) const {
  return val->add_to (n.val);
 }
};

number operator+ (const number& n1,const number& n2) {
 return number (n2.add_to(n1));
}
```

1er dispatching sur le type de l'opérande droit

Multi dispatching
classe corps – 1er dispatching

Spécialisation des opérations des corps

```
class integer : public number {
protected:
 class irepr : public repr {
 public:
  …
  repr* clone() const {
   return new irepr(ival);
  }
  repr* add_to(const repr* r) const {
   return r->add_with(ival);
  }
};
```

2ème dispatching sur le type de l'opérande gauche, en fixant le type de l'opérande droit (ici int)

Multi dispatching
classe corps – 2ème dispatching

Double spécialisation des opérations des corps

Le deuxième dispatching résulte de la résolution de la surcharge et de la liaison dynamique

```
class number {
protected:
 class repr {
  …
  virtual repr* add_with (int) const = 0;
  virtual repr* add_with (double) const = 0;
  virtual repr* add_with (int,int) const = 0;
  …
```

Multi dispatching
classe corps – 2ème dispatching

Double spécialisation des opérations des corps

```
class integer : public number {
protected:
 class irepr : public repr {
 public:
  repr* add_with (int iv) {
   return new integer::irepr (ival + iv) ;
  }
  repr* add_with (double rv) {
   return new real::rrepr (ival + rv) ;
  }
  …
```

Multi dispatching
améliorations

- La solution précédente est insatisfaisante
  - beaucoup de combinaisons possibles
  - toutes les classes corps doivent être modifiées si on ajoute un nouveau genre de nombre
- Pour éviter ces problèmes
  - définir les opérations sur des représentations homogènes (real + real par exemple)
  - convertir vers le type le plus général des opérandes

Multi dispatching
améliorations

Opérations sur des représentations homogènes

```
class number {
 class repr {
  virtual repr* add (const repr*) const = 0;
  …
 };
 …
};
```

convert permet de convertir dans une représentation identique (la plus ”large” des 2)

```
number operator+ (const number& a, const number& b) {
 number::repr* l = convert (a.val, b.val);
 number::repr* r = convert (b.val, a.val);
 return number (l.add(r));
}
```

Multi dispatching
améliorations

Spécialisation des opérations d'une représentation

```
class integer : public number {
protected:
 class irepr : public repr {
 public:
  …
  repr* add_to (const repr* r) const {
   const irepr* x =
                  static_cast<const irepr*>(r);
   return new irepr (ival + x->ival);
  }
 };
…
```

Maintenant, l'opérande droit est toujours un irepr

Multi dispatching
améliorations – fonctions de conversion

On définit les fonctions de conversion (la combinatoire est ici inévitable) :

```
repr* int_to_real (const repr* r) {
 const integer::irepr* i =
     static_cast<const integer::irepr*>(r);
 return new real::rrepr (double(i->val));
}

repr* int_to_rat (const repr* r) {
 const integer::irepr* i =
     static_cast<const integer::irepr*>(r);
 return new rational::rrepr (i->val, 1);
}
… // int_to_complex, rat_to_real, rat_to_complex,
     real_to_complex
```

Multi dispatching
améliorations – fonctions de conversion

- Ordre sur les représentations :
  entiers ⊂ rationnels ⊂ réels ⊂ complexes

```
type_info* order[] = {
 &typeid(integer::irepr),
 &typeid(rational::rrepr),
 &typeid(real::frepr),
 &typeid(complex::crepr),
 0
}
```

- Table des fonctions de conversion

```
typedef repr* (*cvt) (repr*);
cvt cfunc[][4] = {
 { 0, &int_to_rat, &int_to_real, &int_to_complex},
 { 0, 0,           &rat_to_real, &rat_to_complex},
 { 0, 0,         0,              &real_to_complex}
};
```

Multi dispatching
choix de la fonction de conversion

- Quelle est le type effectif de repr ?

```
int search (const repr* r) {
 for (int i = 0; order[i] != 0; ++i)
  if (&typeid(*r) == order[i]) return i;
 return -1; // ne devrait pas arriver
}
```

- Choix de la conversion

```
repr* convert (repr* from, repr* to) {
 int i1 = search (from); int i2 = search (to);
 return i1 < i2 ? (*cfunc[i1][i2])(from) : from;
}
```

Constructeurs virtuels

Que se passe-t-il pour la déclaration

```
nombre n1 = 3, n2 = 5.22
```
 ?

- On peut évidemment prévoir autant de constructeurs que nécessaire :

```
class number {
 …
 number (int v) { val = new integer::irepr(v); }
 number (double v) { val = new real::frepr(v); }
 …
} ;
```

- Mais il faut de nouveau modifier la classe **number** si on ajoute un nouveau genre ; comment l'éviter ?

Constructeurs virtuels

On utilisera un seul constructeur (générique)

```
class number {
 …
 template<typename T>
 number (T v);
 …
};
```

Déclarations

```
number i = 3;   // entier
number q = make_pair(3,2); // rationnel
…
```

Constructeurs virtuels : exemplaires

Chaque classe dérivée

- dit avec quel type d'argument elle se construit
- fournit un constructeur virtuel
- On définit un tableau d'exemplaires, c.a.d. un représentant de chaque dérivée

```
class number {
protected:
 virtual bool built_with(const type_info&)
                                    const = 0;
 virtual repr* make(void*) const = 0;
 static number* exemplaireDerivee[];
 …
```

Constructeurs virtuels : exemplaires

Par exemple

```
class integer : public number {
protected:
 bool built_with (const type_info& i) const {
  return i == typeid(int);
 }

 repr* make (void* v) const {
  int* p = static_cast<int*>(v);
  return new irepr (*p);
 }
 …
```

dans le .cc, on crée les exemplaires

```
integer iex; real rex; …
number* number::exemplaireDerivee[] = {&iex, &rex, …, 0 };
```

Constructeurs virtuels : exemplaires

Le constructeur générique

- regarde si une des classes dérivées peut se construire avec ce type de paramètre
- si oui, il construit une instance de cette classe

```
class number {
 …
 template<typename T>
 number (const T& v) {
  for (int i = 0; exemplaireDerivee[i] != 0; ++i)
   if (exemplaireDerivee[i]->built_with(typeid(T)) {
    val = exemplaireDerivee[i]->make(&v);
    return;
   }
  throw BadArgument();
 }
```