C++ : Un peu de méta-programmation

- Calcul de constante
- Calcul sur les types
- Vérification de concept
- Polymorphisme statique
- Liste de types
- Hiérarchie automatique
- Manipulation de vecteurs

C++ : Compter le nombre d'objets

```
template<typename T>
struct Compteur {
  static unsigned cp;
protected:
  Compteur() { ++cp; }
  Compteur(const Compteur& c) { ++cp; }
  ~Compteur() { --cp; }
};
template<typename T>
unsigned Compteur<T>::cp=0;

struct A : public Compteur<A> { /*..*/ };
```


C++ : Calcul de constantes

Grâce aux classes template, on peut faire des calculs récursifs sur les constantes.

```
template<unsigned N>
struct NbBits {
 static const unsigned res = 1+NbBits<N/2>::res;
};

template<>
struct NbBits<0> {
 static const unsigned res = 0;
};
```


C++ : Calcul de constantes

On peut vérifier que la constante est bien calculée au moment de la compilation.

```
template<unsigned N>
struct Aff {};

int main() { Aff< NbBits<273>::res >::print; }
```

Lorsqu'on compile avec g++ on obtient

```
error : 'print' is not a member of 'Aff<9u>'
```


C++ : Calcul de constantes

Attention, le code suivant n'est pas correct:

```
template<unsigned N>
struct NbBits {
 static const unsigned res =
      (N==0) ? 0 : (1+NbBits<N/2>::res);
};
```

En effet, pour pouvoir évaluer l'expression, le compilateur évalue toutes les sous-expressions, d'où un récurrence infinie.

C++ : Tester l'égalité de deux types

```
template<typename T, typename U>
struct AreEquals {  //  Définition
  static const bool val= false;
};
template<typename T>
struct AreEquals<T,T> {  //  Spécialisation
  static const bool val= true;
};

template<typename T, typename U>
void truc(const T& x, const U& y) {
  if(AreEquals<T,U>::val) //...
}
```

C++ : Vérification de concept

On peut écrire un type template pour vérifier un concept. Par exemple, pour s'assurer qu'un type possède

– une méthode `static void make(int);` et

– une méthode objet `int get() const`:

On écrit la classe suivante:

```
template<typename T,
         void (*Make)(int) = &T::make,
         int (T::*Get)() = &T::get>
struct CheckConcept{
   static const int check=0;
};
```


C++ : Vérification de concept

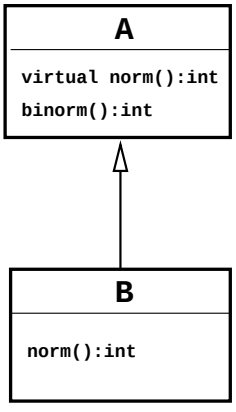
Pour vérifier si un type `truc_t` vérifie le concept, il suffit d'écrire `{CheckConcept<truc_t>::check;}`.

Si la classe `truc_t` ne possède pas les méthodes dont le type est celui des pointeurs requis en paramètres de `CheckConcept`, le compilateur affiche une erreur.

```
error: 'get' is not a member of 'truc_t'
error: template argument 3 is invalid
```


C++ : Rappel : Polymorphisme dynamique

Rappel: de manière dynamique, le polymorphisme s'obtient grâce à `virtual`.



C++ : Rappel : Polymorphisme dynamique

```
struct A {
  virtual int norm() { /***/ }
  int binorm() { return 2*norm(); }
};

struct B : public A {
  int norm() { /***/ }
};

B b;
//...
b.binorm(); // Calculé avec norm() de B
```


C++ : Polymorphisme statique

On voudrait un comportement similaire, avec des appels de méthodes résolus à la compilation.

Pour cela, on fera en sorte que chaque classe soit paramétrée par le vrai type de l'objet. Ainsi, la classe `B` est passée en paramètre de la classe qu'elle étend.

```
struct B : public A<B> {
  int norm() { /***/ }
};
```


C++ : Polymorphisme statique

```
template<typename T>
struct Self{
 public:
  typedef T type;
  type& self() {return static_cast<type&>(*this);}
  const type& self() const {
        return static_cast<const type&>(*this); }
};
template<typename T>
struct A : public Self<T> {
  int norm() { /***/ }
  int binorm() { return 2*this->self().norm(); }
};
```

C++ : Polymorphisme statique

La solution proposée fonctionne bien si on veut simuler une classe abstraite `A`. Mais sinon, on ne peut pas instancier d'objet de type `A`, puisqu'il s'agit d'une classe template...

C++ : Polymorphisme statique

Première solution:

```
template<typename T>
struct A_: public Self<T> {
  int norm() { /***/ }
  int binorm() { return 2*this->self().norm(); }
};

struct A : public A_<A> {};

struct B : public A_<B> {
  int norm() { /***/ }
};
```

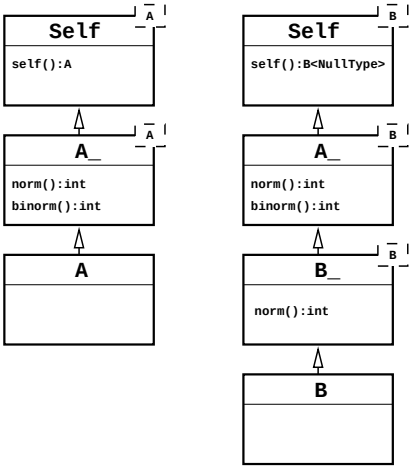

C++ : Polymorphisme statique

On peut même étendre ce principe à toutes les classes de la hiérarchie:

```
template<typename T>
struct B_ : public A_<T> {
  int norm() { /***/ }
};

struct B : public B_<B> {};
```


C++ : Polymorphisme statique



C++ : Polymorphisme statique

Seconde solution:

On crée un type `NullType` pour paramétrer un type qu'on souhaite instancier.

```
struct NullType {};
```

Le type `A<T>` représente alors un objet dont le *vrai* type est `T`, sauf si `T=NullType`, auquel cas le vrai type est `A<NullType>`.

C++ : Polymorphisme statique

On peut écrire une classe qui fait ce calcul:

```
template<template<typename> class A, typename T>
struct TrueType {
  typedef T type;
};

template<template<typename> class A>
struct TrueType<A,NullType> {
  typedef A<NullType> type;
};
```

C++ : Polymorphisme statique

La classe `A` s'écrit alors :

```
template<typename T=NullType>
struct A :
 public Self<typename TrueType<A,T>::type > {

  int norm() { /***/ }
  int binorm() { return 2*this->self().norm(); }
};
```

Et on peut instancier un objet de type `A` en écrivant `A<> a;`

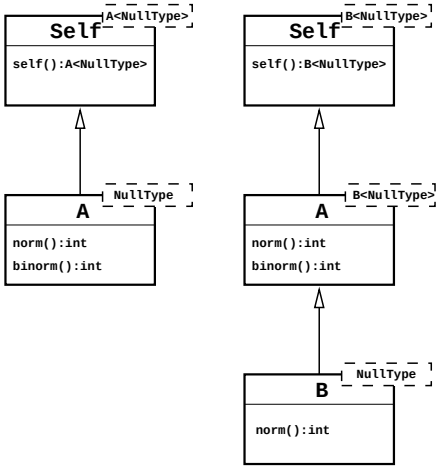
C++ : Polymorphisme statique

On peut étendre ce principe à toutes les classes de la hiérarchie:

```
template<typename T=NullType>
struct B :
  public A<typename TrueType<B,T>::type > {

   int binorm() { return 2*this->self().norm(); }
};
```


C++ : Polymorphisme statique



C++ : Liste de types

```
struct NullType{};

template<typename H, typename T>
struct TypeList{
       typedef H head;
       typedef T tail;
};

typedef
TypeList<int, TypeList<double, NullType> >
  liste_t;
```


C++ : Liste de types

```
template<typename List>
struct ListLenght{
   static const int value =
     1+ListLenght<typename List::tail>::value;
};

template<>
struct ListLenght<NullType>{
  static const int value = 0;
};
```


C++ : Liste de types

```
template<typename List, typename T>
struct ListAdd {
 typedef TypeList<
          typename List::head,
          typename ListAdd<typename List::tail, T>
         > type;
};

template<>
struct ListAdd<NullType, T>{
       typedef TypeList<T, NullType> type;
};
```

C++ : Créer des listes

On peut utiliser des fabriques pour faciliter la création des listes de types.

C++ : Créer des listes

```
template<typename T1, typename T2=NullType,
   typename T3=NullType, typename T4=NullType>
struct MakeList {
  typedef TypeList<T1,
     typename MakeList<T2,T3,T4,NullType>::type
    > type;};

template<>
struct MakeList<NullType,NullType,
                NullType,NullType> {
  typedef NullType type; };
```


C++ : Créer des listes

On peut aussi écrire des classes pour concaténer les listes, supprimer les doublons, etc.

C++ : Hiérarchie automatique

On peut grâce aux templates et aux listes de types engendrer automatiquement une hiérarchie.

Le but peut être

- de créer un type qui étend tous les types de la liste;
- d'avoir une méthode définie pour chaque type de la liste

C++ : Hiérarchie automatique

```
template<typename T>
struct MonHandler {
  bool ma_methode(const T& t) const { /****/ }
};
```


C++ : Hiérarchie automatique

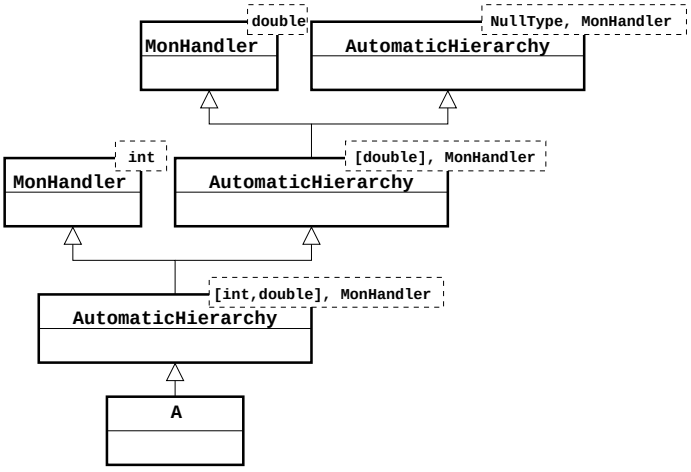
```
template< typename TList,
          template <typename> class Handler>
struct AutomaticHierarchy :
   public Handler<typename TList::head>,
   public AutomaticHierarchy<typename TList::tail,
                             Handler> {};

template<template <typename> struct Handler >
struct AutomaticHierarchy<NullType, Handler> {};
```

C++ : Hiérarchie automatique

```
struct A :
  AutomaticHierarchy<MakeList<int,double>::type,
  MonHandler> {
 /****/
};
```


C++ : Hiérarchie automatique



C++ : Hiérarchie automatique

Un autre type de hiérarchie automatique est celle qui permet d'hériter plusieurs fois de la même classe.

On veut par exemple implanter la composition d'un objet de type Voiture avec 4 objets de type Roue, ce qui se fait en C++ via l'héritage privé.

C++ : Hiérarchie automatique

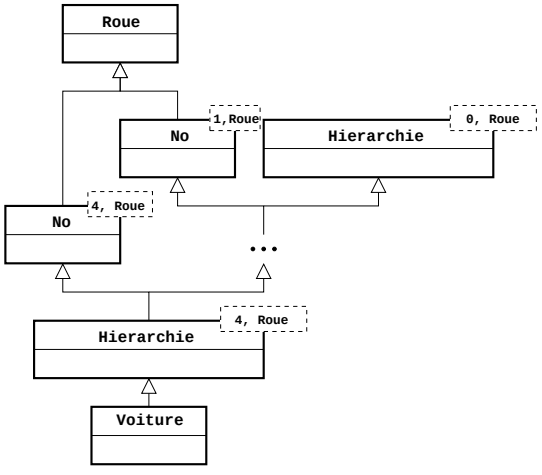
```
template<unsigned N, typename T>
struct No : T{};

template<unsigned N, typename T>
struct Hierarchy : No<N,T>, Hierarchy<N-1, T> {};

template<typename T>
struct Hierarchy<0,T>{};

struct Voiture :
  private Hierarchy<4,Roue> {
 /****/
};
```


C++ : Hiérarchie automatique



C++ : Hiérarchie automatique

On pourra ensuite appeler la méthode `truc` de la troisième roue `Roue` avec `this->No<3,Roue>::truc(...)`.

C++ : Expressions template

On veut écrire une bibliothèque `Vector` qui permette d'écrire, par exemple

```
Vector U,V,W,X;
//...
X = ( V+W )*5 - U;
```

Si on surcharge classiquement les opérateurs pour les vecteurs, chaque sous-expression nécessite un parcours linéaire et la création d'un objet temporaire.

Comment faire en sorte qu'un tel calcul se fasse avec un seul parcours et sans objet temporaire?

C++ : Expressions template

La solution consiste à faire en sorte que `( V+W )*5 - U` construise un arbre d'itérateurs;

chaque feuille est un itérateur sur le vecteur qui est un atome de l'expression,

chaque nœud interne correspond à un opérateur arithmétique, il fait avancer ses fils avec lui et son déréférencement consiste à appliquer l'opérateur au déréférencement de ses fils.

L'opérateur d'affectation appliqué à cet arbre provoque le parcours qui permet de renseigner le résultat.

C++ : Expressions template

```
struct Vector
{
  double Elements[3];
  typedef double* iterator;
  typedef const double* const_iterator;
  template<typename T>
     const Vector& operator=(T t);
  iterator begin() {return Elements;}
  iterator end()   {return Elements + 3;}
  const_iterator begin() const {return Elements;}
  const_iterator end() const {return Elements+3;}
};
```


C++ : Expressions template

```
//Ce qui est commun à tous les noeuds binaires
template <class T, class U> struct BinaryOpBase
{
  BinaryOpBase(T I1, U I2) : It1(I1), It2(I2) {}
  inline void operator ++() {++It1; ++It2;}

  T It1;
  U It2;
};
```


C++ : Expressions template

```
//Pour chaque noeud, on fixe le déréférencement
template <class T, class U>
struct AddOp : public BinaryOpBase<T, U>
{
  AddOp(T I1, U I2) :
       BinaryOpBase<T, U>(I1, I2) {}
  inline double operator *() {
   return *It1 + *It2;}
};
template <class T, class U>
inline AddOp<T, U> MakeAdd(const T& t, const U& u)
{  return AddOp<T, U>(t, u);  }
```


C++ : Expressions template

```
template <class T, class U>
AddOp<T, U> operator +(const T& v1, const U& v2)
{
    return MakeAdd(v1, v2);
}
AddOp<Vector::const_iterator,
      Vector::const_iterator>
   operator +(const Vector& v1, const Vector& v2)
{
    return MakeAdd(v1.begin(), v2.begin());
}
//et versions (const Vector& v1, const T& v2)
           et (const T& v1, const Vector& v2)
```

Il suffit alors de surcharger l'opérateur d'affectation.

```
template <class T>
const Vector& Vector::operator =(T Expr)
{
  for (iterator i = begin(); i != end();
                             ++i, ++Expr)
     *i = *Expr;

  return *this;
}
```

et voilà !