Université de Lyon – Université Claude Bernard Lyon 1
Master d'Informatique

Programmation Avancée
Les différents mécanismes des langages et de C++ pour la généricité

Norme ISO

Raphaëlle Chaine
raphaelle.chaine@liris.cnrs.fr
2023-2024

332

Meta-programmation avec les template

- Mécanisme des template :
  Génération de code par interprétation à la compilation
- Lorsque le compilateur rencontre du code correspondant à une instantiation d'un template :
  - génération de la spécialisation associée aux paramètres fournis.
- Réflexivité offerte par le langage C++ à la compilation

333

333

Spécialisation d'une classe template

```
template <typename T1, typename T2>
class maTemplate{
    T1 a;
    T2 b;
 };
```

- Pour donner une définition différente de certaines instantiations de maTemplate :

```
template< >
class maTemplate<int,double>{
    int a;
    double b[3];
};
```

334

334

Spécialisation d'une classe template

- La spécialisation (d'une classe uniquement!) peut n'être que partielle :

```
template<typename T2>
class maTemplate<char,T2>{
  int a[6];
  T2 b;
};
```

- La spécialisation peut avoir plus de paramètres template que la définition initiale

```
template <typename T1, typename T2, typename T3>
class maTemplate<T1, std::map<T2,T3> > {
    T1 a;
    std::map<T2,T3> b;
};
```

Puis

```
maTemplate<double,double> a1;
maTemplate<int,double> a2;
maTemplate<char,double> a3;
maTemplate<int,std::map<double, int> > a4;
```

335

335

- La métaprogrammation consiste à pousser plus loin les possibilités offertes par le compilateur, de manière à générer du code encore plus performant… au prix d'une compilation plus longue!
- Un métaprogramme génère et manipule du code correspondant à des programmes pendant leur compilation
- Sens littéral : « Un programme sur les programmes »

336

336

- Domaine récent :
  Naissance de nouvelles astuces template de métaprogrammation provoquant l'effervescence des experts
- Quelles en sont leurs applications?
- Comment acquérir les connaissances de base permettant de les utiliser avec discipline?

337

337

– Template à base numérique
  - Optimisation des performances requises par des calculs mathématiques
  - Exponentielle, sinus, factorielle, calculs matriciels
– Template à base de types
  - Typelists (liste de types)
  - Implantation de design patterns (fabrique, visiteurs, …) en conservant un typage fort
  - Analyseurs syntaxiques …

338

338

Compilateur C++ : un système Turing complet?

– Intuition confirmée par le programme d'Erwin Unruh qui calcule les nombres premiers à la compilation, et les affiche à travers les messages d'erreur du compilateur (1994)
– Rappel :
  - Un nombre premier n'est divisible que par 1 et par lui même
  - p est premier s'il est « premier avec les entiers de 2 à p-1 » (ie. pas divisible par)

339

339

```
template <int i> struct D {
    D(void*);
    operator int();
  };
template <int p, int i>
struct is_prime {
    enum { prim = (p==2) ||
               (p%i)&& is_prime<(i>2?p:0),i-1>::prim
    };
  };
template<> struct is_prime<0,0> { enum {prim=1}; };
template<> struct is_prime<0,1> { enum {prim=1}; };

template <int i>
struct Prime_print {
    Prime_print<i-1> a;
    enum { prim = is_prime<i,i-1>::prim };
    void f() { D<i> d = prim ? 1 : 0; a.f();}
  };
template<> struct Prime_print<1> {
    enum {prim=0};
    void f() { D<1> d = prim ? 1 : 0; };
  };
#ifndef LAST
#define LAST 18
#endif
main() {
    Prime_print<LAST> a;
    a.f();
  }
```

Permet d'afficher un entier i dans un message d'errreur

Métafonction calculant récursivement si p est premier avec les entiers inférieurs à i

La fonction membre f affiche i s'il est premier + appel récursif sur i-1 via a

340

340

- Rappel : Les énumérations permettent de donner des noms à des valeurs entières
  enum Jour { lundi=0, mardi=1, mercredi =2 };
  int var=lundi;
- On utilise les énumérations pour donner un nom à des valeurs qui seront « le retour de métafonctions »
- Principe :
  – utiliser des noms paramétrés par des valeurs numériques (pour obtenir une syntaxe du style META_FONCTION<N>)
  – mais on ne peut pas directement paramétrer une énumération par un paramètre template …
  – En revanche on peut l'encapsuler dans une classe template META_FONCTION!
    META_FONCTION<N>::MonEnum désigne alors la valeur de la métafonction appliquée à N

341

341

Autre exemple :
calcul à la compilation de la factorielle d'un nombre entier.

```
template <unsigned int N> struct Factorielle
{
  enum {valeur = N * Factorielle<N-1>::valeur};
};
```

- Cette construction récursive trouve un cas d'arrêt dans une spécialisation du template.

```
template <> struct Factorielle<0>
{
  enum {valeur =1};
}

unsigned int x =Factorielle<3>::valeur;
```

- Dès la compilation `Factorielle<3>::valeur` vaudra 6 (économie du calcul à l'exécution)

342

342

Que penser de :

```
unsigned int i=4;
unsigned int x
=Factorielle<i>::valeur;
```

343

343

- La valeur doit être connue dès la compilation :
```
const unsigned int i=4;
unsigned int x
=Factorielle<i>::valeur;
```
- Depuis C++11 :
`constexpr` permet de spécifier que ce qui suit est évaluable à la compilation

```
constexpr unsigned int i=4;
unsigned int x
=Factorielle<i>::valeur;
```

344

344

- Certaines fonctions peuvent désormais être constexpr

```
constexpr int factorial(int n) {
 return (n<=1)?1:(n*factorial(n-1));
}
```
- Avec `constexpr` la « métaprogrammation » ressemble parfois à de la « programmation » C++!!!!
- ATTENTION :
  - IL FAUT QUE LA FONCTION SOIT EVALUABLE A LA COMPILATION!!
  - `constexpr` n'est pas utilisable dans tous les contextes de métaprogrammation

345

345

- Autre exemple classique :
  Au lieu d'utiliser des énumérations, on peut aussi utiliser des données membres statiques constantes

```
template <unsigned int N>
struct Binaire
{
  static const unsigned int valeur
     =Binaire<N/10>::valeur *2 + N%10;
};
// avec N séquence de 0 et de 1
```

- Spécialisation pour 0 (la faire également pour 1):

```
template <>
struct Binaire<0>
{
  static const unsigned int valeur = 0;
}
```

346

346

- Utilisation :

```
const unsigned int
i=binaire<1001>::valeur;
```

  La valeur 9 de i est calculée à la compilation

- Attention :
  `binaire<54>::valeur` n'a pas de sens!
  Mais il existe des techniques pour s'assurer que le paramètre d'instanciation est bien composé de 0 et de 1 (Abrahams, Gurtovoy)

347

347

- Dans un méta programme, une classe template peut être considérée comme une fonction dans sa forme la plus simple : **prenant des paramètres numériques et retournant une (ou plusieurs!) valeurs**
- Attention les paramètres template ne peuvent être des flottants!
  (mais `3.0f/4.0f` calculé à la compilation ☺)
- Metaprogramme fabriquant un flottant :

```
template <int N> inline double Factorielle()
{return N*Factorielle<N-1>();}
template <> inline double Factorielle<0>()
{return 1.0;}
```

348

348

- Sont évalués à la compilation, les calculs impliquant
  - des valeurs entières ou flottantes
  - des variables entières constantes (ex : const int i=5) (mais pas des doubles)
  - des `constexpr` depuis C++11

```
constexpr double d=5.0;
constexpr double e=d/10.0;
```

  - L'avènement du qualificatif `constexpr` améliore les possibilités de métaprogrammation :
    - Possibilité de faire des fonctions `constexpr`, en utilisant la récursion (C++11)

```
constexpr double factorial(int n)
  { return n <= 1? 1 : (n * factorial(n - 1)); }
```

    - Possibilité de faire des fonctions `constexpr` utilisant les itérations (C++14)

```
for (int i = 0; i < 10; i++)
```

349

349

Récapitulatif sur l'exemple de la factorielle

- Comment mettre en oeuvre une métafonction Factorielle(N)?

- ```
  template <unsigned int N> struct Factorielle
  { enum {valeur = N * Factorielle<N-1>::valeur};};
  template <> struct Factorielle<0>
  { enum {valeur =1}; };
  ```
- ```
  template <int N> inline double Factorielle()
  {return N*Factorielle<N-1>();}
  template <> inline double Factorielle<0>()
  {return 1.0;}
  ```
- ```
  constexpr double factorielle(int n) //C++11
  { return n <= 1? 1 : (n * factorielle(n - 1)); }
  ```

350

350

- Structure de contrôle dans un meta programme:
  - Quel équivalent du `if/else` dans un metaprogramme?

351

351

- Utilisation d'une classe templatée par des booléens et dont les 2 instanciations correspondent à des comportements différents

```
template <bool Condition>
struct Test;
```

- Spécialisations :

```
template <> struct Test<true>
  {
     static void traitement()
        { traitement1();//si possible inline}
  };
template <> struct Test<false>
  {
     static void traitement()
        {traitement2(); //si possible inline}
  };
```

352

352

- Des instructions du type

```
if (Condition)
   traitement1();
else
    traitement2();
```

- Pourront ainsi être remplacées par :

```
Test<Condition>::traitement();
```

- A condition que `Condition` soit évaluable à la compilation!!!

353

353

- L'opérateur conditionnel ternaire peut être évalué à la compilation
  - Exemple d'utilisation :

```
template <unsigned int I> struct NumTest
{
 enum
 {
    Pair = (I%2? false : true),
    Zero = (I==0 ? true : false)
  };
};
```

  - On peut ensuite utiliser les valeurs booléenne `NumTest<I>::Pair` et `NumTest<I>::Zero` (si I évaluable à la compilation!)

354

354

- Boucles dans un meta programme:
  - Quel équivalent du `for` dans un metaprogramme?

355

355
```

- Utilisation d'une construction récursive utilisant une classe templatée par une valeur numérique marquant le démarrage et une valeur numérique marquant la fin de la boucle

```
template <unsigned int Debut, unsigned int Fin>
struct Boucle
{
  static void traitement()
  { MonTraitement();
    Boucle<Debut+1,Fin>::traitement();
  }
};
```

- Spécialisation partielle

```
template < unsigned int N>
struct Boucle<N,N>
{
  static void traitement() {}
};
```

356

356

Des instructions du type

```
for (int i=0;i<10;i++)
  MonTraitement();
```

Pourront ainsi être remplacées par :

```
Boucle<0,10>::traitement();
```

Danger si `Debut>Fin`
(peut être vérifié à la compilation)

357

357

- Le compilateur permet de réaliser des traitements fonctionnels, mais également de stocker des résultats dans des pseudo-variables temporaires ☺

```
template <int X, int Y, int Z>
struct MetaProg
  {
     enum
     {
       v1 = NumTest<Z>::Pair;
       v2= X*v1+Y;
       v3 = NumTest<v2>::Zero ? X : 2*Y;
       valeur = v1*v3;
      };
  };
```

358

358

Utilisation

- Réécriture optimisée de fonctions mathématiques :
- Fonction puissance :

```
template  <unsigned int N>
inline double Puissance(double x)
  {return x*Puissance<N-1>(x);}
template <>
inline double Puissance<0>(double x)
{return 1.0;}
```

- Ici, seul N est un paramètre template, mais pas x (qui peut donc être non évaluable à la compilation)

```
double x, y;
  …
Y=puissance<7>(x);
```

359

359

- Réécriture optimisée de fonctions mathématiques qui s'approximent à partir de leur développement en série entière
  - Si on tronque la série entière à partir d'un degré N, on obtient un polynome de degré N (N=précision numérique)
  - Ex : exponentielle, cos, sin, atan, etc…
- Réécriture de calculs vectoriels et matriciels en utilisant des expressions template (cf. boost::uBlas)
  - Idée : éviter la construction d'objets temporaires
  - Construction arborescente de template
  - Paramètres template = iterateurs sur les vecteurs

360

360

Les typelists

- Listes de types (présentes dans `boost`)
- Utilisées pour la génération automatique de code

```
template < typename T1, typename TL>
struct TypeList
{
 typedef T1 Head; //premier type
 typedef TL Tail; //liste des types restants
};
struct NullType {};
```

- Aucune donnée dans une `TypeList`

361

361

- A partir de cette structure template, on peut fabriquer des listes de types, selon une construction récursive.
- Exemple :

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

- Simplification de la construction à l'aide de macros :

```
#define TYPELIST_1(t1)
   TypeList<t1, NullType>
#define TYPELIST_2(t1,t2)
   TypeList<t1, TYPELIST_1(t2)>
#define TYPELIST_3(t1,t2,t3)
   TypeList<t1, TYPELIST_2(t2,t3)>
```

- Puis

```
typedef TYPELIST_2(int,double) l_int_double;
```

362

362

Opérations sur les typelists

- Méta fonctions:
  - Length<typename TL>
  - IndexOf<typename TL, typename T>
  - TypeAt<typename TL, int N>
  - Union<typename TL1, typename TL2>
  - …

363

363

Length<typename TL>

- Spécialisations :

```
template < typename H, typename TL>
```

Métafonction `Length< TypeList<H,TL> >`

```
… valeur = 1 + Length<TL>::valeur
```

- `template<>`

Métafonction `Length<NullType>`

```
… valeur = 0
```

- Renvoie une erreur si on n'utilise pas une TypeList

Comme vu précédemment, les métafonctions peuvent être mise en œuvre à l'aide de données membres statiques ou des types énumérés encapsulés dans des struct/class templates

364

364

Length<typename TL>

- Traduction en C++
  - Déclaration de la méta-fonction :

```
template < typename TL>
struct Length;
```

  - Spécialisations :

```
template<typename T1, typename TL>
struct Length<TypeList<T1,TL> >
{ enum{ valeur=1+Length<TL>::valeur };
};
template<>
struct Length<NullType>
{enum{ valeur=0 };
};
```

Puis utilisation : `Length<LILI>::valeur`

365

365

IndexOf<typename TL, typename A>

- Spécialisations

```
template <typename TL, typename A>
```

Métafonction `IndexOf< TypeList<`**`A`**`,TL>,` **`A`** `>`

```
… valeur = 0
```

- `template <typename A>`

Métafonction `IndexOf<NullType, A>`

```
… valeur = -1
```

- `template <typename H, typename TL, typename A>`

Métafonction `IndexOf< TypeList<H,TL>, A >`

Utilisation d'une « pseudo-variable » temporaire

```
temp = IndexOf<TL, A> //recherche sur la Queue
```

Si `(temp == -1)` alors `valeur = -1 // pas trouvé`

Sinon `valeur = 1 + temp // trouvé`

366

366

Utilisation

- Permet de définir des familles de type
  - `typedef TYPELIST_4(int, char, long, short) EntierSigné`
  - `typedef TYPELIST_4(uint, uchar, ulong, ushort) EntierNonSigné`
  - `typedef Union<EntierSigné, EntierNonSigné> Entier`
  - `typedef TYPELIST_2(float, double) Flottant`
  - `typedef Union<Entier, Flottant> Nombre`
- Permet d'explorer un ensemble de type (par exemple pour générer du code associé)
  - Les template variadiques de C++11 le permettent également

367

367

- Curiously Recurring Template Pattern (CRTP)
  - Schéma

```
template <class T>
class Base {
 // ...
};
class Derivee : public Base<Derivee> {
 // ...
};
```

  - Modèle d'une classe qui sera **dérivée une seule fois** et qui sait par quelle classe Derivee.
  - Les fonctions membres de la Base<Derivee> pourront accéder par downcast à celles de la Derivee
  - Possible car les fonctions membres de Base et de Dérivée, si elles sont instanciées, le seront bien après leur classes d'appartenance

368

368

- CRTP peut être utilisé à des fins de polymorphisme statique (sans surcout lié à virtual)
  - Schéma

```
template <class T>
class Base{
    void interfaceSpecialisee()
    { static_cast<T*>(this)->implementation(); }
    static void static_func()
    { T::static_sub_func(); }
}; //Un Base<Derivee>* est downcastable en Derivee*

struct Derivee : public Base<Derivee> {
    void implementation();
    static void static_sub_func();
};
```

  - Vous vous demandez à quoi ca peut servir?

369

369

- Génération automatique d'une fonction clone dans une hiérarchie de classe polymorphe

```
class Figure {
  public:
    virtual ~Figure() {}
    virtual Figure *clone() const = 0;
};
// La classe CRTP dérive ici de Figure …
template <typename Derivee>
class Figure_CRTP : public Figure {
public:
   virtual Derivee *clone() const
    {return new Derivee(
          *static_cast<Derivee const*>(this));}
};
```

- Définition d'une spécialisation de Figure déjà munie d'une fonction clone par simple dérivation du patron de méthode

```
class Triangle : public Figure_CRTP<Triangle> {
};
```

370

370

- Exemple d'utilisation :
  - Embarquement automatique d'un compteur d'instances dans une classe

```
template <typename T>
struct avec_compteur{
  protected:
       static int objets_vivants;
       avec_compteur(){++objets_vivants;}
       avec_compteur(const avec_compteur &)
                  {++objets_vivants;}
       ~avec_compteur(){--objets_vivants;}
};
template <typename T>
  int avec_compteur<T>::objets_vivants( 0 );
```

  - Définition d'une classe avec un compteur d'instance par simple dérivation du template précédent

```
class X : protected avec_compteur<X>{// …
};
```

371

371

Utilisation des listes de types

- Génération automatique de hiérarchies de classes
- Exemple :
  - Etant donné une hiérarchie de classes
    Ex : Hiérarchie de classe Figure
    - rectangle,
    - rond,
    - groupe de Figures [*pattern composite*]

```
class Figure {  …  };
class groupe : public Figure { …
  set<Figure *> composants;
  };
```

  - On veut mettre en place une deuxième hiérarchie de classe correspondant à un pattern Visiteur
    (ex: pour visualiser avec de nouveaux modes d'affichage)
  - Permet une séparation données/traitement d'affichage

372

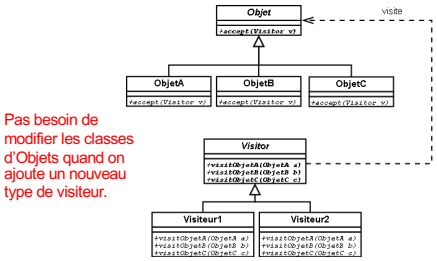
372

Utilisation

- Toutes les figures dispose de fonctions d'affichages élémentaires (ex: affichage1 et affichage2), à partir desquelles il est possible de composer de nouveaux modes d'affichage
- On ne souhaite pas revenir sur la définition des figures, chaque fois qu'on souhaite ajouter un nouveau mode d'affichage
- C'est de nouveaux visiteurs qui se chargeront d'afficher les figures selon ce nouveau mode.

373

373

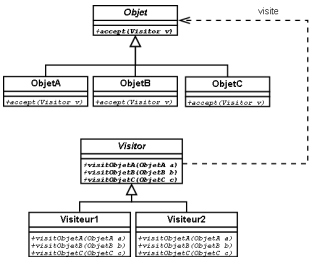


Pas besoin de modifier les classes d'Objets quand on ajoute un nouveau type de visiteur.

- Un visiteur dont on n'a pas besoin de connaître le type exact pourra visiter un objet (ici une figure) dont on ne connaît pas le type exact
- Ici : accepte = recoitVisite

374

374




```
class Figure
{public :
  virtual void
    accepte(Visiteur&)const =0;
};
```

375

375

Pour chaque classe d'objet visité :

- fonction membre **virtuelle** `accepte`:

```
void FigureA::accepte(Visiteur& v) const
{ v.visiteA(this) ; }
```

Pour chaque classe de visiteur :

- Autant de fonctions membres **virtuelle**s `visite` que de classes dans la hiérarchie de Figures

```
void MonVisiteur::visiteA(const FigureA& f)
  { // Traitement d'un objet de type A }
void MonVisiteur::visiteB(const FigureB& f)
  { // Traitement d'un objet de type B }
void MonVisiteur::visiteC(const FigureC& f)
  { // Traitement d'un objet de type C }
```

- On peut ici bénéficier du mécanisme de surcharge :
  Une fonction visite par type visité! Elles ont des paramètres de type différent ce qui permet de les distinguer.
  On peut tout appeler visite

376

376

Pour chaque classe d'objet visité :

- fonction membre **virtuelle** `accepte`:

```
void FigureA::accepte(Visiteur& v)const
{ v.visite(this) ; }
```

Pour chaque classe de visiteur :

- Autant de fonctions membres **virtuelle**s `visite` que de classes dans la hiérarchie de Figures

```
void MonVisiteur::visite(const FigureA& f)
  { // Traitement d'un objet de type A }
void MonVisiteur::visite(const FigureB& f)
  { // Traitement d'un objet de type B }
void MonVisiteur::visite(const FigureC& f)
  { // Traitement d'un objet de type C }
```

En effet les visiteurs sont des visiteurs multiples au sens où ils doivent pouvoir visiter différents types de Figure.

377

377

Utilité d'une génération de code automatique :

Dans une spécialisation MonVisiteur de la Hiérarchie Visiteur, toutes les surcharges de MonVisiteur::visite ont généralement le même code! (une variation est néanmoins envisageable)

Pour la classe `VisiteurAffichagePleinRouge` :

- Autant de fonctions membres **virtuelle**s `visite` que de classes dans la hiérarchie de Figures

```
void VisiteurAffichagePleinRouge::visite(
  const FigureA& f)
  { f.affichagePlein(); f.afficheRouge();}
void VisiteurAffichagePleinRouge::visite(
  const FigureB& f)
  {f.affichagePlein(); f.afficheRouge();}
void VisiteurAffichagePleinRouge::visite(
  const FigureC& f)
  {f.affichagePlein(); f.afficheRouge();}
```

378

378

- Etant donné une spécialisation de Visiteur :
  - N'écrire le code de `visite` qu'une seule fois dans un *template*, puis l'instancier pour chacun des types d'une `TypeList` des spécialisations de `Figure`

Remarque :

- Il est envisageable que le code de visite soit différent juste pour les `Ronds` par exemple (grâce à une spécialisation adhoc du template qui génère le code de visite)

379

379

- Ecriture d'une classe *templatée* par le type FF de `Figure`, permettant de générer une classe de Visiteur avec UNE fonction membre `visite(const FF&)` et son code.
- ```
  template <class FF>
  class PatronVisiteur1 :
                  public Visiteur
  { blablacode
    using Visiteur::visite;
    void visite(const FF& f)
     {f.affichageMode1();}
  }
  ```
- Attention affichageMode1 peut même être non virtuelle ☺

380

380

- La classe `VisiteurMultiple1` devra savoir visiter plusieurs type de `Figure`s, elle héritera donc des instanciation du *template* pour tous les types de `Figure`s.
- Pour résoudre le problème de l'héritage multiple de la classe de base `Figure`, on opte pour un **héritage** dit **virtuel** (cf. cours sur l'héritage virtuel)

381

381

- Code écrit une seule fois dans un *template*, puis instanciation pour chacun des types d'une TypeList de `Figure`s
- ```
  template <class FF>
  class PatronVisiteur1
      : virtual public Visiteur
  { blablacode
    using Visiteur::visite;
    void visitObjet(const FF *o)
     {o->affichageMode1();}
  };
  ```
- La classe `VisiteurMultiple1` finale dérivera de toutes ses instanciations!

382

382

- Création d'un template qui instancie le Patron de visiteur pour tous les types de `Figure` listées dans une TypeList (ou un template varyadique).
  - Utilisation d'un *template* dont un des paramètres *template* est un *template*!

  (ici le *template* du Patron de Visiteurs)
- On obtient ensuite un `VisiteurMultiple` qui sait visiter tous les types de `Figure` selon le mode unique défini par le *template*

383

383

```
template <class TList,
          template <class> class PVisiteur>
class VisiteurMultiple;
```

Spécialisations :

```
• template <class T1, class TL,
            template <class> class PVisiteur>
  class VisiteurMultiple<TypeList<T1,TL>, PVisiteur >
   : public PVisiteur<T1>,
     public VisiteurMultiple<TL,PVisiteur>
     { ...A vous de mettre les bons using
     }; // 2 héritages
• template <class T,
            template <class> class PVisiteur>
  class VisiteurMultiple<TypeList<T,NullType>,
                         PVisiteur >
   : public PVisiteur<T>
    {...A vous de mettre le bon using
    }; // 1 seul héritage dans le cas d'une liste
       // contenant un seul type
```

384

384

- Reste à définir la classe de base `Visiteur` qui doit elle même présenter tous les prototypes des surcharges de la fonction membre virtuelle visite
- On utilise un mécanisme similaire au précédent, pour construire `Visiteur` par héritage multiple de classes construites par un *template*

385

385

```
template <class OO>
class PatronBaseVisiteur
{
  void visitObjet(const OO *o) virtual=0;
};
```

La classe `Visiteur` finale dérivera de toutes les instanciations de ce *template* pour la liste des types d'objets traités!

Même mécanisme que précédemment en définissant un template

```
template <class TList >
class BaseVisiteurMultiple;
```

386

386

Au final, la classe Visiteur correspond à :

```
typedef BaseVisiteurMultiple<LTypesFig>
Visiteur;
```

où `LTypesFig` est une `TypeList` des types de figures possibles

387

387

Notions de concept et de modèle

- Lorsqu'on définit un template générique, certaines propriétés syntaxiques mais aussi sémantiques sont attendues de la part des paramètres *template*
- En cas de non respect, à l'instantiation :
  1. d'un prérequis syntaxique :
     erreur de compilation (pas très sibylline) à l'endroit où est réalisé l'appel à la propriété syntaxique défaillante!
     --- On aurait préféré être renseigné dès l'instanciation ---
  2. d'une prérequis sémantique :
     aucune erreur de compilation, mais risque de déroulement incorrect du programme

388

388

En JAVA

- Template de JAVA
  - On peut même imposer des contraintes sur le type template ☺

```
public class LeJavaTemplate<E extends LaJavaClasse
& LaJavaInterface1 & LaJavaInterface2>
extends Vector<E>{
 …
 public E methode1(){…}
 public <E2 extends E> E2 methode2(E2 t){…}
 …
}
```

389

389

En C++

- Pour tester qu'une classe hérite d'une autre
- Depuis C++11
  - Métafonction std::is_base_of

```
template< class Base, class Derived >
struct is_base_of;
```

- A partir de C++17

```
template< class Base, class Derived >
inline constexpr bool is_base_of_v
= is_base_of<Base,Derived>::value;
```

- Sinon, vous pouvez toujours faire la métafonction vous-même en vous inspirant du design de boost

390

390

Static_assert

- Initialement présent dans boost, maintenant dans C++11 sous la forme d'une constexpr
  `static_assert(`expression booléenne`)`
- L'assertion statique est un test à la compilation

```
template <bool Test> class StaticAssert;
template<> class StaticAssert<true> {};
```

Une seule instanciation possible !

Evaluation statique de Test

- `true` : le compilateur passe le test
- `false` : erreur `StaticAssert<false>` non défini

Exemple :

- `StaticAssert< sizeof(T) == 4>`
- vérifie que le type T occupe 4 octets

391

391

- Catégorisation des types abstraits de données en concepts décrivant des prérequis syntaxiques et sémantiques
- Concepts permettant de spécifier
  - à quels types un traitement générique est applicable
  - quels types peuvent être substitués aux paramètres template formels
- Un type satisfaisant à un concept C est dit modèle de ce concept C
- Un modèle qui ajoute des prérequis à un autre correspond à un raffinement de ce concept

392

392

- Un template doit fournir ses exigences en termes de concepts
- Concepts de la STL : `Assignable, Default Constructible, Less than Comparable, Equality Comparable, Container, Iterator, Algorithms, Fonctors, Adaptors,…`
- Dans un concept peuvent aussi apparaître des invariants (associativité, symétrie,…) ou des garanties en termes de complexité.

393

393

- Mais aucun contrôle n'est fait à l'instanciation!
- Historiquement, les concepts C++ ne jouait qu'un rôle de documentation
- La responsabilité de la vérification incombait à l'utilisateur!!
- Les concepts sont apparus dans la norme depuis C++20

394

394

- Notion de concept disponible depuis C++20
  - On peut imposer des contraintes sur les paramètres template :

```
template <typename T>
requires std::totally_ordered<T>
T myMin(T e1,T e2)
 {
  return (e1<e2)?e1:e2;
 }
```

  - Le compilateur refusera d'instancier myMin avec des classes ne satisfaisant pas le concept totally_ordered.
  - La bibliothèque standard offre un certain nombre de concepts de base et le programmeur peut en créer de nouveau (cf. cours ultérieur)

395

395

Possibilité de bâtir des nouveaux concepts

- Exemple :

```
template <typename T>
concept MonConcept = requires(T v)
{ {v.MaFonction()} ->
std::convertible_to<std::string>; };
```

`T` satisfait `MonConcept` si il offre une fonction `MaFonction` retournant un objet convertible en `string`

396

396

- Concepts offerts par la bibliothèque standard depuis C++20 (concept library)

same_as, derived_from, convertible_to, common_reference_with, common_with, integral, signed_integral, unsigned_integral, floating_point, assignable_from, swappable/swappable_with, destructible, constructible_from, default_initializable, move_constructible, copy_constructible

boolean-testable, equality_comparable/equality_comparable_with, totally_ordered/totally_ordered_with, three_way_comparable/three_way_comparable_with
Movable, copyable, semiregular, regular

invocable/regular_invocable
equivalence_relation
strict_weak_order

397

397