Patrons de conception statiques pour la programmation générique en C++

A. Duret-Lutz et T. Géraud

Rapport technique 9903
Septembre 1999

(version préliminaire)

Laboratoire de Recherche et Développement d'EPITA
14-16, rue Voltaire – 94276 Le Kremlin-Bicêtre cedex – France
Tél. +33 1 44 08 01 01 – Fax +33 1 44 08 01 99

Table des matières

1 Introduction

1.1 Remerciements

Merci à Gabriel Dos Reis pour ses commentaires sur la première moitié de ce rapport.

1.2 Conventions

Certains mots de ce texte sont utilisés dans des acceptions bien précises. Le mot « patron » est utilisé comme traduction de l'anglais *template* tandis que « modèle » est pris au sens STL comme « instance de concept » (ces termes seront définis précisement section 4.2). De même nous ferons la distinction entre les paramètres de génériques des paramètres ordinaires. Par exemple dans

```
template< typename T, int U = 3 >
void foo(int a, T b) {
  // ...
}
```

`T` et `U` sont les paramètres de la fonction générique `foo`, tandis que `a` et `b` sont les paramètres de la fonction `foo<T,U>`.

1.3 Polymorphisme statique

Dans ce rapport, nous ne nous intéressons qu'à un style de programmation appelé « polymorphisme statique ». Dans ce paradigme, les objets ne sont manipulés qu'à travers type le plus inférieur (*leaf type* en anglais) c'est à dire qu'à tout moment le type dynamique d'un objet est égal à son type statique. Par exemple si une classe `B` dérive d'une classe `A`, une instance de `B` ne sera jamais manipulée en tant qu'instance de `A`. Ceci permet d'une part au compilateur de résoudre l'appel des méthodes lors de la compilation et de ne pas avoir besoin de table de fonctions virtuelles. D'autre part cela permet d'utiliser convenablement les templates (ou seul le type statique est pris en compte).

2 Méthodes différées

2.1 Références

L'idée de paramétrer une classe de base par sa classe dérivée à été donnée pour la première fois par Barton et Nackman dans [1] où ils s'en servent pour définir des opérateurs automatiquement (différement de ce que nous présentons section 2.4). Ce patron que nous allons

présenter est donc souvent dénommé *The Barton and Nackman Trick* bien qu'en réalité l'idée de transtyper la classe de base vers la classe inférieure soit donnée par James Coplien dans [2] et [3] sous le nom de *Curiously Recurring Template Pattern*.

Dans [11] Todd VELDUIZEN utilise le nom *Curiously Recursive Template Pattern* qu'il attribue à *Geoff Furnish*.

2.2 Buts

Ce patron permet l'appel dans une classe de base de méthodes définies dans une classe dérivée. Contrairement à une approche classique par méthodes polymorphes (`virtual`), la résolution de l'appel est faite statiquement par le compilateur qui peut donc facilement continuer à appliquer des techniques d'optimisation telles que l'*inlining*.

Une utilisation est la factorisation dans une classe de base de méthodes communes à plusieurs sous-classes.

Ce patron permet aussi l'accès aux attributs des classes dérivées, par contre il ne permet pas d'en déduire des types (voir section 3.4).

2.3 Détails

On souhaite traduire statiquement l'exemple suivant qui utilise le polymorphisme dynamique pour permettre de substituer `f()` dans les sous-classes `B` et `C`.

```
struct A
{
  virtual void f() { cout << "A::f" << endl; }
  void g()         { f(); }
};

struct B : public A
{
  void f()         { cout << "B::f" << endl; }
};

struct C : public A
{};

int     main()
{
  B b;
  C c;
  b.g();  // prints "B::f"
  c.g();  // prints "A::f"
}
```

L'idée est de paramétrer la classe `A` par sa classe fille afin que tout `A` sache se convertir de façon complètement sûre en son type réel. La fonction `A::self()` est chargée d'effectuer cette conversion.

```
template< typename Infer >
struct A
{
  void f() { cout << "1" << endl; }
  void g() { self().f(); }

  Infer& self() { return static_cast<Infer&>(*this); }
};

struct B : public A< B >
{
  void f() { cout << "2" << endl; }
};

struct C : public A< C >
{};

int     main()
{
  B b;
  C c;
  b.g();  // prints "2"
  c.g();  // prints "1"
}
```

L'utilisation de `self()` force le **compilateur** à remonter l'arbre d'héritage à partir du type le plus inférieur (ici `B` ou `C`) pour résoudre l'appel d'une fonction ou l'accès à un attribut. Cette résolution est véritablement effectuée lors de la compilation et non pendant l'exécution d'où un gain de temps (par rapport à l'utilisation d'une table de fonctions virtuelles).

Ce type d'héritage supporte facilement plusieurs niveaux, il suffit que toutes les classes soient paramétrées par le type de la classe la plus inférieure et transmettent ce type lors de l'héritage. À titre d'exemple, et pour une utilisation plus pratique de ce patron il est facile de définir une classe dédiée.

```
template< class Infer >
class infer
{
protected:
        Infer &_self()       { return static_cast<Infer&>(*this); }
  const Infer &_self() const { return static_cast<const Infer&>(*this); }
};
```

La classe générique `A` se réécrirait alors de la façon suivante.

```
template< typename Infer > : infer< Infer >
struct A
{
  void f() { cout << "1" << endl; }
  void g() { _self().f(); }
};
```

Il aurait été possible de ne pas définir `A::f()`, mais à la différence d'une construction `virutal void f() = 0;` le compilateur ne produira de message d'erreur qu'en essayant d'appeler une implantation de `f()` inexistante au lien de le faire lors d'un instanciation d'objet. En d'autre terme nous ne pouvons pas forcer le programmeur que dérive une class de `A` à respecter une interface comme le permet l'écriture `virutal void f() = 0;`. Nous ne conservons que le coté *virtuel* de la déclatation pas le *virtuel pur*.

Cette façon d'écrire nécessite l'appel de `_self()` avant chaque appel de fonction « virtuelle ». Il s'agit d'un problème. En effet, à un n'importe quel niveau de la hiérarchie il devient imperatif de connaître les fonctions virtuelles afin de leur préfixer le `_self()` est nécessaire. Une solution

est de séparer les parties de déclarations et d'implémentation de telles fonctions.

```
template< typename Infer > : infer< Infer >
struct A
{
  void f() { _self().f_impl(); }
  void g() { f(); }

  void f_impl() { cout << "1" << endl; }
};
```

À tout niveau il est maintenant possible d'appeler `f()` comme s'il s'agissait d'une fonction normale, et il est possible de redéfinir `f()` en définissant `f_impl()`.

2.4 Exemple utile

Une utilisation pratique est la définition d'opérateur arithmétiques de façon automatique à partir d'un sous-ensemble d'opérateurs définis.

L'approche classique, utilisée dans STL, consiste à déclarer des opérateurs templates à un niveau global.

```
template< class T >
inline const T operator*( const T& lvalue, const T& rvalue )
{
  return T( lvalue ) *= rvalue;
}

template< class T >
inline const T operator+( const T& lvalue, const T& rvalue )
{
  return T( lvalue ) += rvalue;
}
```

Il est ensuite possible d'appliquer les opérateurs `+` et `*` à toute classe définissant `*=` et `+=`. Les deux inconvénients sont d'une part que ces définitions d'opérateurs ne s'appliquent pas à des classes particulières mais à toutes celles qui sont utilisées (STL[9] définit ses opérateurs dans l'espace de noms `std::rel_ops` ce qui ne fait que déplacer le problème), d'autre part que ces opérateurs ont priorité sur les opérateurs hérités comme le montre l'exemple suivant.

```
template< class T >
inline const int operator*( const T& lvalue, const T& rvalue )
{
  return 1;
}

struct A
{
  const int operator*( const A& rvalue )
    {
      return 2;
    }
};

struct B : public A
{};

int main()
{
  A a;
  B b;
  cout << a*a << endl;            // prints "2"
  cout << b*b << endl;            // prints "1"  (1)
  cout << b.operator*(b) << endl; // prints "2"
}
```

Ligne `(1)` l'opérateur `B::operator*()` n'est pas utilisé car son appel nécessite la conversion de `b` en `A` alors qu'en revanche l'opérateur template peut être instancié pour `B` et appliqué directement sur `b` sans conversion.

Ainsi, cette façon de faire limite la création d'opérateur particuliers et devient une source d'erreurs lorsque l'héritage est utilisé.

Une solution basée sur les *méthodes différées* peut-être :

```
template <typename T>
struct operable : public infer< T >
{
  const T operator*( const T& r ) const
    {
      T t(_self());
      t *= r;
      return t;
    }

  const T operator+( const T& r ) const
    {
      T t(_self());
      t += r;
      return t;
    }
};
```

Une classe `A` pour laquelle `operator+=()` ou `operator*=()` et définit bénéficiera automatiquement des opérateurs `+` et `*` associés si elle dérive de `operable<A>`.

Il faut noter que `g++ 2.95.1` ne parvient pas à optimiser les appels fait dans la classe générique `operable` aux opérateurs `*=` et `+=` lorsque constructeur de copie de `T` n'est pas défini à la main.

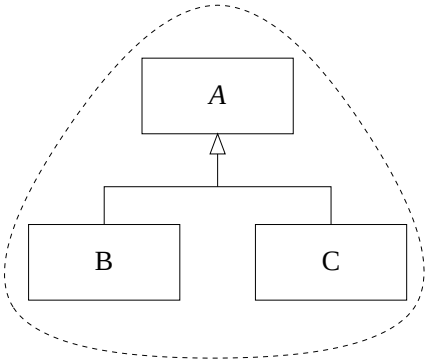

FIG. 1 – Héritage avec fonctions virtuelles

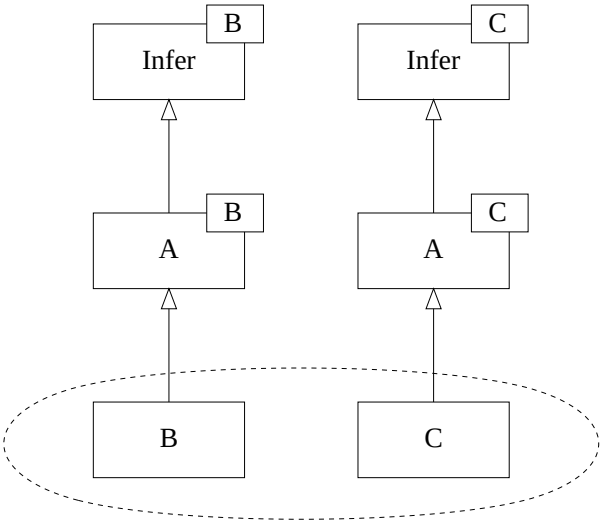

FIG. 2 – Héritage avec template récursif

2.5 Différentes formes d'héritage possibles

Comme tout patron de programmation basé sur le polymorphisme statique, cette méthode n'est utilisable que lorsque l'on manipule uniquement des objets dont le type statique égale le type dynamique. Dans les figures 1 et 2 seules les classes entourées sont utilisables (*i.e.* instanciables, ou référençables).

Il est possible de faire dériver une classe de `B`, mais pas d'y substituer des méthodes car `_self()` ne descendra pas plus bas que `Infer` qui vaut `B`.

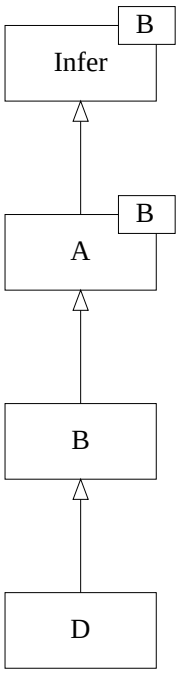

FIG. 3 – `D` hérite de `B`

```
template< typename Infer >
struct A : infer< Infer >
{
  void f() { _self().f_impl(); }
  void f_impl() { cout << "A::f_impl" << endl; }
};

struct B : public A< B >
{
  void f_impl() { cout << "B::f_impl" << endl; }
};

struct D : public B
{
  void f_impl() { cout << "D::f_impl" << endl; }
};

int main()
{
  D d;
  d.f(); // prints "B::f_impl"
}
```

Sur ce dernier exemple, l'instruction `d.f()` provoque l'appels de `d.A<B>::f()`, à son tour l'appel de `_self().f_impl()` dans la classe `A<B>` est traduit en `static_cast<B&>(*this).f_impl()`. C'est donc bien `d.B::f_impl()` qui est executé.

Une solution consiste à créer tout d'abord toute la hiérarchie correspondante de classes paramétrées, comme l'illustre la figure 4(a) et le code suivant.

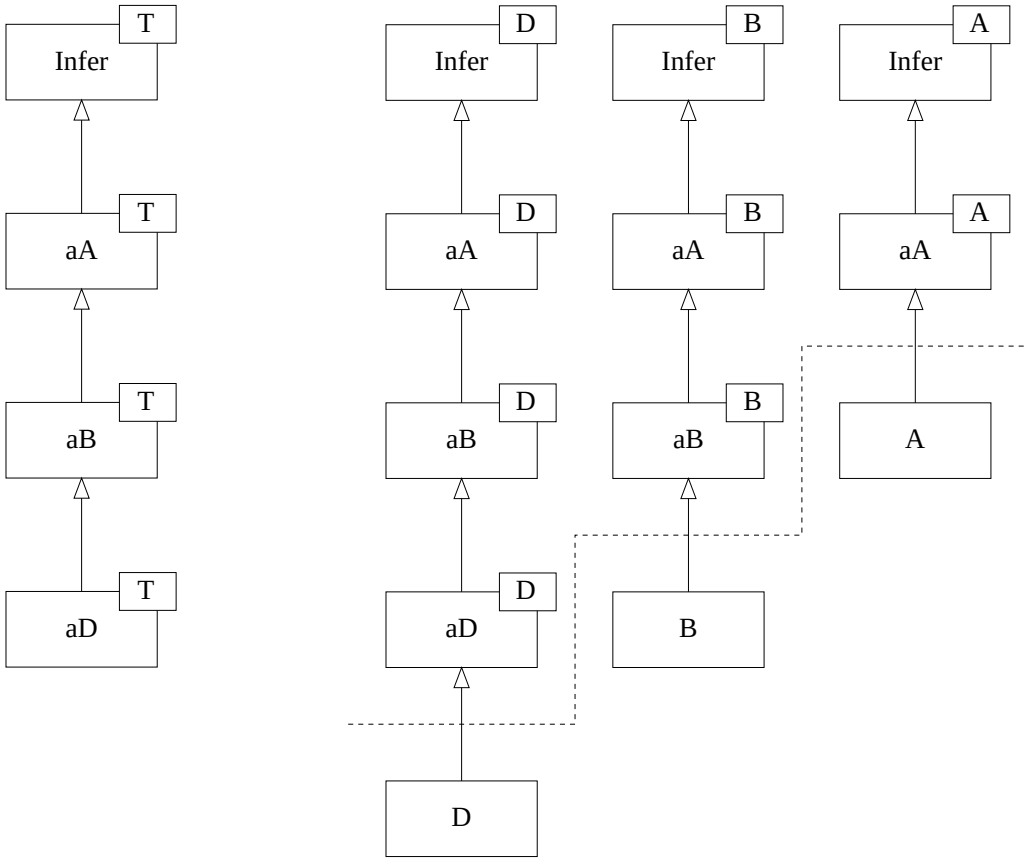

FIG. 4 – (a) hiérarchie « abstraite ». (b) classes concrètes. Conceptuellement `D` hérite de `B` qui hérite elle-même de `A`, mais pratiquement ce sont trois classes avec leurs propres hiérarchies.

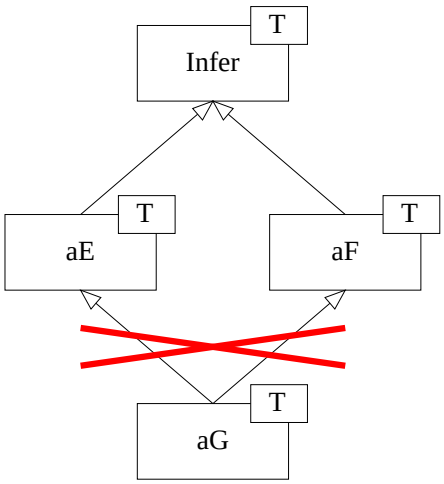

FIG. 5 – Héritage multiple impossible avec les classes paramétrées par `Infer`

```
template< typename Infer >
struct aA : infer< Infer >
{
  void f() { _self().f_impl(); }
  void f_impl() { cout << "aA::f()" << endl; }
};

template< typename Infer >
struct aB : public aA< Infer >
{
  void f_impl() { cout << "aB::f()" << endl; }
}

template< typename Infer >
struct aD : public aB< Infer >
{
  void f_impl() { cout << "aD::f()" << endl; }
};
```

Pour utiliser l'une de ces classe, il suffit d'en faire dériver une classe vide (figure 4(b)).

```
struct A: public aA< A > {};
struct B: public aB< B > {};
struct D: public aD< D > {};

int main()
{
  A a; B b; D d;
  a.f(); // prints "aA::f()"
  b.f(); // prints "aB::f()"
  d.f(); // prints "aD::f()"
}
```

Il n'est pas possible de pratiquer l'héritage multiple dans les classes paramétrées par `Infer` avec ce patron(figure 5).

En revanche l'héritage multiple reste possible dans les classes utilisables. Deux configurations sont alors envisageables : figure 6(a), `G` hérite de `E` et `F` (qui sont des classes vides, puisque toutes les méthodes se trouvent dans `aE` et `aF`) ; figure 6(b) `G` hérite directement des classes `aE` et `aF`. En fait, cette seconde forme n'est pas compilable, car le `static_cast` vers `G` ne peut-être résolu dans les classes `aE<G>` et `aF<G>` (un `dynamic_cast` et une table de méthodes

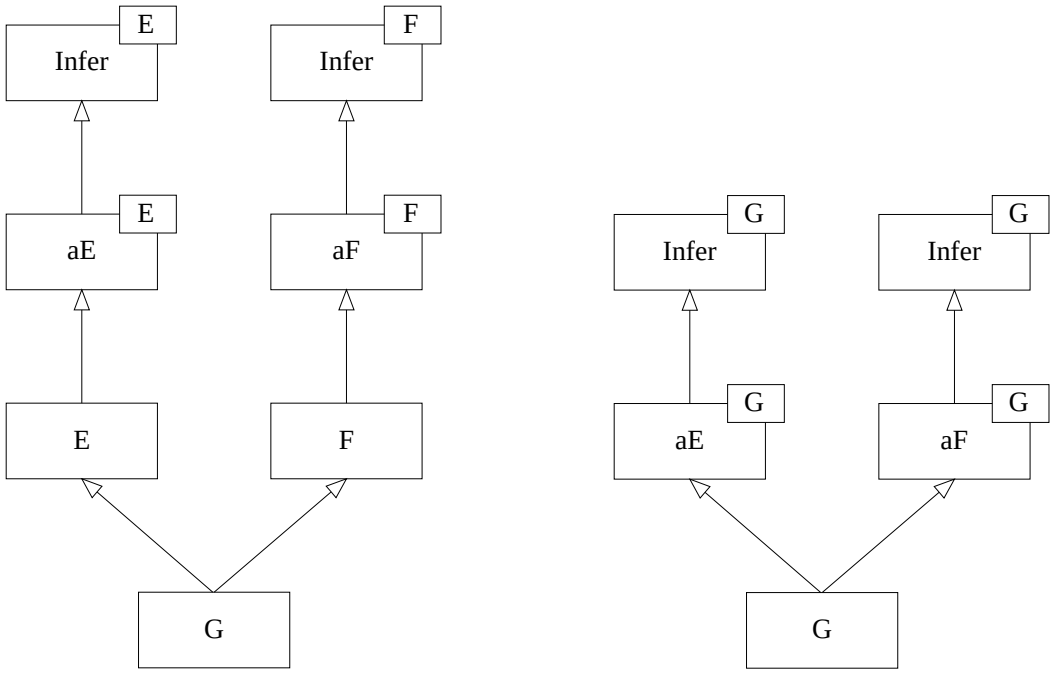

FIG. 6 – 6(a) Héritage multiple de classe concrètes. 6(b) Héritage multiple de classe abstraites (IMPOSSIBLE).

virtuelles contenant des informations sur la position des classes `aE<G>` et `aF<G>` dans `G` est nécessaire, mais il devient alors plus simple d'utiliser des méthodes polymorphes).

2.6 Limitations

Il n'est pas possible de déduire des types des classes dérivées (et exemple et une solution à ce problème sont présentés section 3.4).

Ce patron conduit à une forte duplication des classes comme le montrent les figures 2 et 4(b) et par conséquent une génération de code plus volumineux (*code bloat*).

Toute classe appliquant ce patron devient inutilisable seule. `aA<Infer>`, `aB<Infer>`, `aC<Infer>`, sont inutilisable sans `Infer`. Il faut alors en faire dériver une classe vide. La section 5.6 présente un facilité pour réaliser cela.

3 `typedefs` extrusifs

3.1 Références

L'utilisation des *traits* à été décrite en 1995 par Nathan Meyers dans [7].

3.2 Buts

Déclarer les typedefs relatifs à une classe à l'extérieur de la classe en question. L'utilité première est de dissocier les déductions de types associés à des classes des classes elle-même. On souhaite être en mesure de déduire un type d'une classe dont on connait le nom mais pas la définition ; on veut aussi pouvoir ajouter de nouvelles déductions de types sans avoir à recompiler la classe.

Il est alors possible d'utiliser la déduction de types des classes inférieures dans le patron précédent.

3.3 Détails

L'idée est de déclarer ces types en dehors de la classe. En utilisant un *trait* par type.

Supposons que nous disposions d'une classe `A` sur laquelle il soit possible d'itérer en utilisant une classe dédiée `Iterator`. On souhaite pouvoir déduire `Iterator` de `A`.

iterator.hh

```
template< typename T >
struct iterator
{};
```

A.hh

```
class A
{
  // ... no typedef
};
```

AIterator.hh

```
#include "iterator.hh"

namespace XyzInternal {
  class AIterator
  {
    // ...
  };
}

class A; // predeclaration

template<>
struct iterator<A> // specialization
{
  typedef XyzInternal::AIterator type;
};
```

main.cc

```
#include "AIterator.hh"

// ...
iterator< A >::type  i;
// ...
```

Il est possible d'utiliser `Iterator< A >` sans connaître `A`.

Mieux, avec cette écriture il est possible de définir des types associés à des types prédéfinis comme `int` (par exemple on pourrait définir un `bits_iterator<int>::type`).

Une limitation importante se constate lors de l'écriture d'une routine générique. Le code suivant ne compilera pas.

foo.h

```
template< typename T >
void foo(T& t)
{
   typename iterator< T >::type i
   // ...
};
```

main.cc

```
#include "A.hh"
#include "foo.hh"

int main()
{
  A a;
  foo(A);
}
```

En effet, `foo()` utilise `iterator< A >::type` qui n'est pas défini. Au moins deux solutions sont envisageables, soit l'utilisateur de `foo.hh` se charge d'inclure tous les fichiers d'en-tête qui lui sont nécessaires, soit `foo.h` charge un fichier `iterator.hh` contenant toutes les définitions des itérateurs existants.

Il existe d'autres façons d'agencer ces classes pour les déduire les unes des autres. Discutons rapidements d'écritures plus classiques pour la déduction de type sur ce même exemple. Il est possible d'écrire :

A.hh

```
class A
{
  public:
    class Iterator
    {
      // ...
    };
};
```

main.cc

```
#include "A.hh"

// ...
A::Iterator  i;
// ...
```

ou, ce qui revient au même :

A.hh

```
class A
{
  public:
    typedef AIterator Iterator;

  // ...

  private:
    class AIterator
    {
      // ...
    };
};
```

Dans ces deux écritures la classe `A` et sont itérateur sont très liées. Il est alors possible de définir l'itérateur en dehors de la classe `A` :

A.hh

```
namespace XyzInternal {
  class AIterator
  {
    // ...
  };
}

class A
{
  public:
    typedef XyzInternal::AIterator Iterator;
  // ...
};
```

`A` et `AIterator` sont un peu plus indépendants. Si `AIterator` n'est pas utilisé dans `A`, il est possible de séparer d'avantage ces deux classes, pour qu'une utilisation de `A` n'entraîne pas une déclaration (peut-être inutile) de `AIterator` et qu'une modification de `AIterator` n'entraîne pas une recompilation de `A`.

AIterator.hh

```
namespace XyzInternal {
  class AIterator
  {
    // ...
  };
}
```

A.hh

```
namespace XyzInternal {
  class XyzInternal::AIterator; // predeclaration
}

class A
{
  public:
    typedef XyzInternal::AIterator Iterator;
  // ...
};
```

main.cc

```
#include "A.hh"
#include "AIterator.hh"

// ...
A::Iterator  i;
// ...
```

Il est maintenant possible d'utiliser la classe `A` sans avoir chargé la définition de `AIterator`. Le découplage discuté par John Lakos dans [6] est meilleur. La contrepartie est qu'un utilisateur voulant utiliser `A` et `AIterator` doit inclure un fichier d'en-tête supplémentaire (à moins de faire inclure `A.hh` par `AIterator.hh'`). En revanche ce n'est pas possible dans l'autre sens : il est *impossible d'utiliser un type déduit de `A` sans connaître `A`*. Par exemple `A` pourrait définir un type `A::cumul` que l'utilisateur voudrait utiliser sans forcément connaître le reste de la définition de `A`.

L'utilisation des *traits* présentée en debut de section permet donc de répondre à ce problème.

3.4 Utilisation avec le patron précédent

Le patron précedent permet d'utiliser des méthodes et des attributs des classes dérivées mais la déduction de `typedef`s n'est pas possible car on se retrouve dans le cas de figure suivant.

```
template< class Infer >
class A
{
  public :
    typename Infer::data_type data;  // (2)
};

class B : public A< B >              // (1)
{
  public :
    typedef int data_type;
    // ...
};
```

Cet échantillon de code ne compile pas. La raison est un problème d'ordre des déclarations dans les classes qui utilisent les types de leurs classes dérivées. En effet le compilateur effectue les opérations suivantes :
– il lit la définition de A ;
– en arrivant en `(1)` il a besoin d'instantier `A<B>` ;
– en instanciant A il découvre `(2)` qui est un type non encore déclaré.
En utilisant une déclaration de type extrusive le problème ne se pose plus.

```
template< typename T >
struct data_type
{};

template< class Infer >
class A
{
  public :
    typename data_type< B >::type data;
};

class B; // predeclaration

template<>
struct data_type< B >
{
  typedef int type;
};

class B : public A< B >              // (1)
{
  public :
    // ...
};
```

Au moment d'instancier `A<B>` en `(1)` le type `data_type<B>::data` est connu.

3.5 Lien avec la déduction de type classique

Ce patron est compatible avec une définition intrusive des types. Il suffit que par défaut le *trait* du type soit défini par rapport au type déduit de son paramètre :

```
template< typename T >
struct data_type {
  typedef typename T::data_type type;
};
```

Si une classe définit déjà le type `data_type`, par exemple

```
srtuct C {
  typedef int data_type;
  // ...
};
```

l'écriture `data_type<C>::type` garde tout son sens. Cette façon d'écrire est donc utilisable dans un projet où certaines classes seraient écrites avec une déduction de types classique.

3.6 Limitations

La syntaxe est alourdie, mais peut-être allégée avec des macros.

```
#define PRETYPE(name) \
  template<T> struct name { typedef T::name name; };

#define TYPEDEF(class,value,name) \
  template<> struct name<class> { typedef value name; };
```

4 Vues d'un objet – concepts

4.1 But

Manipuler un objet à travers une interface différente selon les utilisations que l'on souhaite en faire. Implanter la notion de concept de façon à ce qu'elle puisse être contrôlée par le compilateur.

4.2 Détails

STL, *Standard Template Library* [10, 9], définit la notion de concept. Les concepts de STL définissent des interfaces, c'est-à-dire un ensemble de méthodes et de types que doit posséder une classe. Les objets qui réalisent l'interface d'un concept `A` sont dits *modèles* de `A`. Un objet peut-être modèle de plusieurs concepts, qui sont autant de façons de voir l'objet. L'utilisation des concepts est devenue nécessaire lorsqu'il a fallu définir quelles classes pouvaient être utilisées comme paramètres des algorithmes de STL. Ainsi un modèle de `container` doit définir des fonctions telles que `size()`, `empty()`, et définir des types comme `iterator`, etc.

Prenons les deux classes suivantes pour illustrer ce fonctionnement.

```
class list {
  public :
    int number_of_elements() const;
    typedef list_iterator iterator;
    // ...
};
```

```
class tree {
  public :
    int number_of_vertices() const;
    typedef prefixe_tree_iterator prefixe_iterator;
    typedef postfixe_tree_iterator postfixe_iterator;
    // ...
};
```

Une liste et un arbre peuvent tous deux être vus comme un conteneur. Dans un algorithme générique s'appuyant uniquement sur ce concept, on souhaiterait donc pouvoir appeler une méthode `cardinal()` sur chacune de ces deux classes.

Une première solution, mise en œuvre dans STL, consiste à déclarer toutes les méthodes des concepts dont une classe est modèle dans la classe elle-même. Cette approche est possible à condition que pour deux concepts donnés les méthodes de même nom aient la même sémantique.

```
class list {
  public :
    int number_of_elements() const;
    int cardinal() const { return number_of_elements(); }
    typedef list_iterator iterator;
    // ...
};
```

```
class tree {
  public :
    int number_of_vertices() const;
    int cardinal() const { return number_of_vertices(); }
    typedef prefixe_tree_iterator prefixe_iterator;
    typedef postfixe_tree_iterator postfixe_iterator;
    // ...
};
```

En réalité STL n'aurait pas déclaré de méthodes `number_of*()`, mais uniquement `cardinal()`. Cependant si les classes existaient avant leur adaptation aux concepts, leurs méthodes peuvent être déjà utilisées. Quoi qu'il en soit, une méthode utilisant le concept de conteneur s'écrit alors simplement.

```
template<class Container>
void foo(Container& c)
{
  int i = c.cardinal();
  // ...
}
```

Pour le programmeur, la signature de la méthode est claire : `foo()` travaille sur un conteneur. Il est donc possible d'utiliser tous les types définit par le concept `container`. Du point de vu du compilateur en revanche `Container` n'est qu'un type quelconque. Cela signifie que le type n'est contraint que lors des appels aux fonctions (ou lors des déductions de type). En d'autres termes la notion de concept ne fait pas partie de l'implantation mais uniquement de la documentation.[1] Dans l'exemple précédent il est possible d'appeler sur `c` une méthode qui ne fait pas partie du concept `Container`, disons `number_of_elements()` sans rencontrer d'erreur ; l'erreur ne sera détéctée que lorsque `foo()` sera appliqué à autre chose qu'une liste. Le patron que nous présentons permet d'intégrer véritablement les concepts à l'écriture du code, et de restreindre l'interface utilisable à celle du concept afin de provoquer l'impression de messages d'erreur si cette interface n'est pas respectée, ou si une fonction appelée n'y figure pas.

Un second inconvénient de cette dernière approche est la nécessité de modifier les classes lors de la création d'un nouveau concept pour créer les fonctions et types demandés par l'interface. Les classes doivent définir les méthodes correspondantes à tous les concepts selon lesquels elles peuvent être utilisées quand la plupart du temps il aurait suffit de renommer les fonctions pour la durée de l'utilisation sous le concept donné. Le patron présenté ici permet d'effectuer cette association « méthode de concept » – « méthode de classe » sans toucher à la classe.

```
template< class Model,
          int (Model::*Cardinal)() const = &Model::cardinal,
          typename Iterator = typename Model::iterator >
struct container_concept_def
{
  static int cardinal(const Model& model)
  {
    return (model.*Cardinal)();
  }
  typedef Iterator iterator;
};
```

[1] SGI a récemment mis au point un ensemble de macros permettant de vérifier que les classes passées en paramètre d'une méthode correspondent au concept associé. [8]

```
template< class Model >
struct container_concept
{
  typedef container_concept_def< Model > def;
};
```

Le code précédent définit le concept `container` comme contenant une méthode `cardinal()` et un type `iterator`. `cardinal()` appelle sur un objet passé en paramètre la méthode `Cardinal` (paramètre de `contenair_concept_def)`, ce qui permet de rediriger les méthodes simplement. Par défaut, `cardinal()` appellera la méthode `cardinal()` sur l'objet `model`. Pour exprimer le fait que `list` et `tree` peuvent être vus comme des concepts de `container` il suffit de spécialiser `container_concept` lorsque cela est nécessaire, c'est-à-dire lorsque des méthodes ou types doivent être renommés.

```
template<>
struct container_concept< list >
{
  typedef container_concept_def< list, &list::number_of_elements > def;
};
```

```
template<>
struct container_concept< tree >
{
  typedef container_concept_def< tree,
                                 &tree::number_of_vertices,
                                 tree::prefixe_iterator > def;
};
```

Lors de l'écriture d'une méthode utilisant un `tree` ou une `list` comme modèle de `container` il suffit d'accéder aux méthodes dans `container_concept_def` à travers le type `def` défini dans la spécialisation de `container_concept` pour le type manipulé :

```
template< class T >
void foo( T& t )
{
  typedef typename container_concept< T >::def as_a_container; // (1)

  int card = as_a_container::cardinal( t );

  typename as_a_container::iterator iter;

  // ...
}
```

La ligne `(1)` n'est pas innocente : elle provoque une instanciation de `container_concept<T>` donc une vérification de la part du compilateur que la classe `T` possède bien les méthodes et types nécessaires à son utilisation comme concept de `container`. Toutes les erreurs liées à un non-respect de l'interface de `container` seront reportées à cette ligne et non au fil des appels de méthodes.

Mieux encore, il est possible de rajouter des fonctionnalités au concept sans toucher à la classe. Par exemple un `container` pourrait posséder un fonction `empty()` retournant `true` si `cardinal()` retourne `0`.

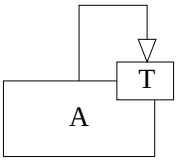

FIG. 7 – `A` hérite de son paramètre `T`

```
template< class Model,
          int (Model::*Cardinal)() const = &Model::cardinal,
          typename Iterator = typename Model::iterator >
struct container_concept_def
{
  static int cardinal(const Model& model)
  {
    return (model.*Cardinal)();
  }
  static bool empty(const Model& model)
  {
    return cardinal(model) == 0;
  }
  typedef Iterator iterator;
};
```

4.3 Limitations

quelles sont-elles ?

4.4 Références

Erich Gamma et *al.* présentent dans [4] un patron appelé « adapteur » permettant d'adapter l'interface d'une classe à ses besoins. Mais ce patron ne permet que de choisir l'interface de l'objet au moment de son instanciation, et non de changer l'interface d'un objet déjà instancié comme ce que nous venons de présenter permet.

5 Héritage paramétré

5.1 Buts

Modifier les comportement d'une classe tout en gardant cette modification générique vis-à-vis de la classe.

5.2 Détails

Le principe, très simple, est résumé par la figure 7.

Une implémentation en C++ se résume au code suivant.

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

5.3 Exemple

Définissons deux itérateurs, capables de parcourir un champ de données dont le type est passé en paramètre, l'un en avant, l'autre en arrière.

```
template< typename T >
class ForwardIterator {

public :
  ForwardIterator(T* str, int size) :_str(str), _size(size) { _pos=0;}

  void next() { ++_pos; }
  T& value() { return _str[_pos]; }
  bool end() { return _pos >= _size; }

private :
  int _size,_pos;
  T* _str;
};
```

```
template< typename T >
class BackwardIterator {

public :
  BackwardIterator(T* str, int size) :_str(str), _size(size) { _pos = size-1;}

  void next() { --_pos; }
  T& value() { return _str[_pos]; }
  bool end() { return _pos < 0; }

private :
  int _size,_pos;
  T* _str;
};
```

Nous souhaitons créer à partir de ces classes d'itérateurs, mais sans les modifier, des versions capables d'avancer de plusieurs positions à la fois. Deux possibilité s'offrent à nous : déléguer ou sous-classer.

Commençons pas la délégation. Une version générique de cette modification de comportement peut facilement être écrite. La classe `JumperIterator` est paramétrée par l'itérateur que l'on souhaite modifier, elle contient une copie (ou référence au choix) de l'itérateur modifié et lui délegue toutes ses un méthodes excepté `next()` qui est répétée `_jump` fois.

```
template< class Iter >
class JumperIterator {
public:
  JumperIterator( const Iter& iter, int jump ) : _iter(iter), _jump(jump) {}

  void next() { for (int i = _jump; i>0 ; --i) _iter.next(); }
  typename Iter::value_type & value() { return _iter.value(); }
  bool end() { return _iter.end(); }

private :
  int _jump;
  Iter _iter;
};
```

L'itérateur précédent s'utilise alors de la façon suivante.

main.cc

```
int main() {
  char s[] = "Hello world!";
  {
    ForwardIterator< char > i( s, sizeof(s) );
    JumperIterator< ForwardIterator< char > > j( i , 2 );

    for ( ; !i.end(); i.next())  std::cout << i.value();
    std::cout << std::endl;

    for ( ; !j.end(); j.next())  std::cout << j.value();
    std::cout << std::endl;
  }
  {
    BackwardIterator< char > i( s, sizeof(s) );
    JumperIterator< BackwardIterator< char > > j( i , 2 );

    for ( ; !i.end(); i.next())  std::cout << i.value();
    std::cout << std::endl;

    for ( ; !j.end(); j.next())  std::cout << j.value();
    std::cout << std::endl;
  }
}
```

affiche

```
Hello world!
Hlowrd
!dlrow olleH
drwolH
```

On remarque que pour écrire la délégation de `value()` il faut connaître son type de retour. En se rend donc compte qu'il faudrait ajouter un type `value_type` à chacune des classes d'itérateurs pour que cela fonctionne. Nous rompons donc notre contrat qui était de ne pas toucher à ces classes. À ce problème, aux moins deux solutions existent. L'une, non développée ici consiste à utiliser le patron des `typedef`s extrusifs. L'autre consiste à passer `value_type` en paramètre. L'écriture suivante est envisageable

```
template< typename T, class Iter >
class JumperIterator {
  // ...
};
```

Malheuresement, la syntaxe induite pour créer un tel itérateur force à répéter l'information `char`.

```
ForwardIterator< char > i( s, sizeof(s) );
JumperIterator< char, ForwardIterator< char > > j( i , 2 );
```

Pour palier à ce problème on paramètre `JumperIterator` par le type des valeur, et le patron de l'itérateur.

```
template< typename T, template <typename U> class Iter >
class JumperIterator {
public:
  JumperIterator( const Iter< T >& iter, int jump ) : _iter(iter), _jump(jump) {}

  void next() { for (int i = _jump; i>0 ; --i) _iter.next(); }
  T & value() { return _iter.value(); }
  bool end() { return _iter.end(); }

private :
  int _jump;
  Iter< T > _iter;
};
```

On écrira alors :

```
ForwardIterator< char > i( s, sizeof(s) );
JumperIterator< char, ForwardIterator > j( i , 2 );
```

Le problème de connaître le type des valeurs cache en fait l'un des inconvéniants serieux dont souffre l'extension par délégation : il est impératif de connaître les signatures de toutes les méthodes à déléguer. Finalement l'extension est trés liée à la classe étendue. Trop pour pouvoir être véritablement réutilisable, car en voulant ne changer que la fonction `next()` nous avons imposé la présence des fonctions `value()` et `end()` !

Voyons maintenant le cas de l'extension par sous-classage. Il est possible de dériver chaque classe itérateur pour créer deux classes `JumperForwardIterator` et `JumperBackwardIterator` redéfinissant chacune la fonction `next()` de la même façon. L'idée de ce patron est justement de regrouper ces deux classes en une seule, et de paramétrer ce qui change, c'est-à-dire l'héritage :

```
template< typename Iter >
class JumperIterator : public Iter {

public :
  JumperIterator( const Iter& iter, int jump ) : Iter(iter), _jump(jump) {}

  void next() { for (int i = _jump; i>0 ; --i) Iter::next(); }

private :
  int _jump;
};
```

Le code repéré main.cc reste completement valable, cet itérateur agit véritablement comme une délégation, seulement nous n'imposons rien d'autre sur à la classe étendue que de contenir la méthode `next()` qui est redéfinie.

5.4 Limitations

Dans l'exemple précédent la surcharge de la fonction `next()` n'est faite qu'au niveau de la classe `JumperIterator<T>`. Une fonction de `ForwardIterator` qui utiliserait `next()` appelerait `ForwardIterator::next()` et non celle de `JumperIterator` même si l'appel à cette fonction a été fait depuis `JumperIterator`. La délégation produit le même comportement. Tandis qu'avec du polymorphisme dynamique, les fonctions de `ForwardIterator` peuvent être rendues polymorphes et se faire substituer par celles de `JumperIterator`. Il serait tentant d'essayer d'appliquer les patron des méthodes différées ici. `ForwardIterator` deviendrait alors `ForwardIterator<Infer>` et serait à lui seul inutilisable (sans `Infer` comme indiqué

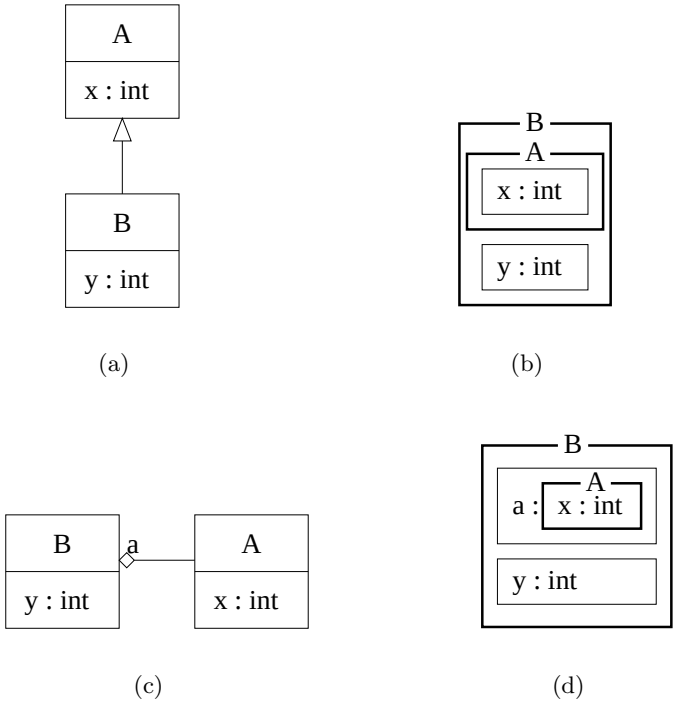

FIG. 8 – Différences internes entre la composition par héritage et la composistion par délégation. 8(a) composition par héritage ; 8(b) sa réalisation. 8(c) composition par délégation ; 8(d) sa réalisation.

section 2.6), de plus on ne peut pas utiliser une classe paramétrée par sa classe inférieure (`ForwardIterator<Infer>`) et un classe paramétrée par sa classe supérieure (`JumperIterator<Super>`) ensemble : les types seraient récurssifs.

5.5 Différences entre composition par héritage et composition délégation

Comme le montre la figure 5.5, une classe `B` qui hérite de `A` ou qui contient `A` contient les mêmes champs d'un point de vu réalisation en mémoire. Seul change la façon d'accéder aux données : lorsque `B` contient un champ `a` de type `A` (figure 8(c)) il est nécéssaire de traverser `a` pour accéder à ses champs. On peut voir cela comme un renommage de variables, `x` devient `a.x`.

Dans la composition par délégation, Comme `A` est un champ de `B` il est possible d'en changer, d'écrire des choses comme :

```
b1.a = b2.a;
```

où `b1` et `b2` sont des instances de `B`. De même, cela est tout à fait réalisable dans la version avec héritage ! Il suffit d'interpréter `B` comme un `A` :

```
static_cast<A&>b1 = static_cast<A&>b2;
```

La composition par héritage réalise une union (au sens ensembliste) des attributs et méthodes des classes participantes, il peut y avoir écrasement ou conflit lorsque ces ensembles ne sont pas disjoints. La composition par délégation réalise aussi une union mais après avoir renommé les attributs et méthodes d'une classe évitant tout conflit (dans le cadre d'une extension ce renommage force le programmeur à réécrire les appels aux fonctions déléguées).

La composition par délégation évite les éventuels conflits mais force à réécrire les appels à toutes les fonctions déléguées.

Bien entendu l'héritage possède d'autres propriétés, comme le fait de pouvoir voir `B` comme un `A`, qui ne nous intéressent pas ici car nous ne voulons manipuler que les types des feuilles. Dans ce contexte, les compositions par héritage ou délégation rendent des services équivalants.

5.6 Utilisation avec les méthodes différées

Nous avons montré section 2.5 qu'une façon d'utiliser l'héritage paramétré par la classe inférieure consistait à créer toute une hiérarchie paramétrée puis à dériver une classe vide à tous les niveaux que l'ont souhaite pouvoir instancier (figure 4(b)).

L'héritage paramétré offre une possibilité pour gérer ceci. Nous pouvons créer un classe générique vide qui dérive de sont paramètre (qui est une classe paramétrée par la classe inférieur) :

```
template< template< class U > class T >
struct concrete : public T< concrete< T > >
{
  // empty
};
```

Maintenant nous pouvons l'utiliser comme suit :

```
template< class Infer >
struct A : public infer< Infer > {
  void f() { _self().f_impl(); }
  void f_impl() { std::cout << "A::f()" << std::endl; }
};

template< class Infer >
struct B : public A< Infer > {
  void f_impl() { std::cout << "B::f()" << std::endl; }
};

template< class Infer >
struct D : public B< Infer > {
  void f_impl() { std::cout << "C::f()" << std::endl; }
};

int main()
{
  concrete<A> a;
  concrete<B> b;
  concrete<D> d;
  a.f();                // "A::f()"
  b.f();                // "B::f()"
  d.f();                // "D::f()"
}
```

5.7 Utilisation avec STL

Cette façon d'hériter pourrait-être utilisé avec STL à deux niveaux.

D'abord, comme dans l'exemple précédant, pour modifier le comportement des objets ; par exemple définir des accesseurs qui permettent aux itérateurs de n'accéder qu'à certains membres des éléments des containers, par exemple dans [5] nous montrons comment itérer sur le champ `Red` d'une liste `std::list< RGB< int > >`.

Ensuite cet héritage pourrait être utilisé à l'intérieur de STL. A titre d'exemple STL possède une classe `stack` qui n'est pas véritablement un conteneur mais plutôt une surcouche offrant une interface réduite à un conteneur sous-jacent (STL designe ceci par *adaptor*). Concrètement, la classe `stack` ressemble à ceci :

```
template< class T, class Sequence = deque< T > >
class stack
{
  public :
    typedef typename Sequence::value_type      value_type;
    typedef typename Sequence::size_type       size_type;
    typedef          Sequence                  container_type;

    typedef typename Sequence::reference       reference;
    typedef typename Sequence::const_reference const_reference;

    stack() : _c() {}
    explicit stack(const Sequence& s) : _c(s) {}

    bool empty() const             { return _c.empty(); }
    size_type size() const         { return _c.size(); }
    reference top()                { return _c.back(); }
    const_reference top() const    { return _c.back(); }
    void push(const value_type& x) { _c.push_back(x); }
    void pop()                     { _c.pop_back(); }

  protected:
    Sequence _c;
};
```

Voici comment cette classe pourrait-être réécrite en utilisant l'héritage paramétré :

```
template< class T, class Sequence = deque< T > >
class stack : protected Sequence
{
  public :
    using typename Sequence::value_type;
    using typename Sequence::size_type;
    typedef          Sequence                  container_type;

    using typename Sequence::reference;
    using typename Sequence::const_reference;

    mystack() : Sequence() {}
    explicit mystack(const Sequence& s) : Sequence(s) {}

    using Sequence::empty;
    using Sequence::size;
    reference top()                { return back(); }
    const_reference top() const    { return back(); }
    void push(const value_type& x) { push_back(x); }
    void pop()                     { pop_back(); }
};
```

L'héritage protégé est utilisé pour dissimuler les attributs et méthodes de la `Sequence` utilisée,

et avec le mot clef `using` nous rendons certains types et méthodes publiques (au moment ou ce rapport est écrit, '`using typename`' n'est pas encore supporté par `g++`, il faut donc utilise des `typedef`).

Références

[1] J. Barton and L. Nackman. Scientific and Engineering C++. Addison-Wesley, 1994.

[2] James O. Coplien. The columns without a name : A curiously recurring template pattern. C++ Report, February 1995.

[3] James O. Coplien. Curiously recurring template pattern. In Stanly B. Lippman, editor, C++ Gems. Cambridge University Press & Sigs Books, 1996. `http://people.we.mediaone.net/stanlipp/gems.html`.

[4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns : Elements of Reusable Object-Oriented Software. 1994.

[5] Thierry Géraud and Alexandre Duret-Lutz. Vers une plus grande réutilisabilité en programmation générique. Soumis à Technique et scences informatiques, 2000.

[6] John Lakos. Large-Scale C++ Software Design. Addison-Wesley, 1996.

[7] Nathan Myers. A new and useful template technique : "Traits". C++ Report, 7(5) :32–35, June 1995. `http://www.cantrip.org/traits.html`.

[8] Jeremy Siek. Template argument checking. OON mailing list, August 1999. `http://www.oonumerics.org/oon/oon-list/archive/0245.html`.

[9] Silicon Graphics Computer Systems, Inc. Standard Template Library Programmer's Guide. `http://www.sgi.com/Technology/STL/index.html`.

[10] Alex Stepanov and Meng Lee. The Standard Template Library. Hewlett Packard Laboratories, 1501 Page Mill Road, Palo Alto, CA 94304, February 1995.

[11] Todd Velduizen. Technique for Scientific C++, April 1998. `http://extreme.indiana.edu/~tveldhui/papers/`.