

Ch. 13 - Templates

Langage C / C++

R. Absil

Haute École Bruxelles-Brabant
École supérieure d'Informatique



6 octobre 2021

Table des matières

- 1 Généralités
- 2 Instanciation
- 3 Dédution d'arguments
- 4 Templates amis
 - Amitié et opérateurs
- 5 Templates variadiques
- 6 Spécialisation de templates

Table des matières

- 1 Généralités
- 2 Instanciation
- 3 Dédution d'arguments
- 4 Templates amis
 - Amitié et opérateurs
- 5 Templates variadiques
- 6 Spécialisation de templates

Table des matières

- 1 Généralités
- 2 Instanciation
- 3 Dédution d'arguments
- 4 Templates amis
 - Amitié et opérateurs
- 5 Templates variadiques
- 6 Spécialisation de templates

Table des matières

- 1 Généralités
- 2 Instanciation
- 3 Dédution d'arguments
- 4 Templates amis
 - Amitié et opérateurs
- 5 Templates variadiques
- 6 Spécialisation de templates

Table des matières

- 1 Généralités
- 2 Instanciation
- 3 Dédution d'arguments
- 4 Templates amis
 - Amitié et opérateurs
- 5 Templates variadiques
- 6 Spécialisation de templates

Table des matières

- 1 Généralités
- 2 Instanciation
- 3 Dédution d'arguments
- 4 Templates amis
 - Amitié et opérateurs
- 5 Templates variadiques
- 6 Spécialisation de templates

Table des matières

- 1 Généralités
- 2 Instanciation
- 3 Dédution d'arguments
- 4 Templates amis
 - Amitié et opérateurs
- 5 Templates variadiques
- 6 Spécialisation de templates

Généralités

Overview

- *Introduction* au mécanisme de templates en C++
 - Livres complets existants
- Très efficace
 - Génération à la compilation
 - Liaison statique des liens
- Permet
 - de la programmation modulaire
 - d'émuler des relations d'héritage
 - d'émuler le polymorphisme en « compile-time »
 - d'éviter des conversions implicites
- Mécanisme très différent des generics en Java

Overview

- *Introduction* au mécanisme de templates en C++
 - Livres complets existants
- Très efficace
 - Génération à la compilation
 - Liaison statique des liens
- Permet
 - de la programmation modulaire
 - d'émuler des relations d'héritage
 - d'émuler le polymorphisme en « compile-time »
 - d'éviter des conversions implicites
- Mécanisme très différent des generics en Java

Overview

- *Introduction* au mécanisme de templates en C++
 - Livres complets existants
- Très efficace
 - Génération à la compilation
 - Liaison statique des liens
- Permet
 - de la programmation modulaire
 - d'émuler des relations d'héritage
 - d'émuler le polymorphisme en « compile-time »
 - d'éviter des conversions implicites
- Mécanisme très différent des generics en Java

Overview

- *Introduction* au mécanisme de templates en C++
 - Livres complets existants
- Très efficace
 - Génération à la compilation
 - Liaison statique des liens
- Permet
 - de la programmation modulaire
 - d'émuler des relations d'héritage
 - d'émuler le polymorphisme en « compile-time »
 - d'éviter des conversions implicites
- Mécanisme très différent des generics en Java

Overview

- *Introduction* au mécanisme de templates en C++
 - Livres complets existants
- Très efficace
 - Génération à la compilation
 - Liaison statique des liens
- Permet
 - de la programmation modulaire
 - d'émuler des relations d'héritage
 - d'émuler le polymorphisme en « compile-time »
 - d'éviter des conversions implicites
- Mécanisme très différent des generics en Java

Overview

- *Introduction* au mécanisme de templates en C++
 - Livres complets existants
- Très efficace
 - Génération à la compilation
 - Liaison statique des liens
- Permet
 - de la programmation modulaire
 - d'émuler des relations d'héritage
 - d'émuler le polymorphisme en « compile-time »
 - d'éviter des conversions implicites
- Mécanisme très différent des generics en Java

Overview

- *Introduction* au mécanisme de templates en C++
 - Livres complets existants
- Très efficace
 - Génération à la compilation
 - Liaison statique des liens
- Permet
 - de la programmation modulaire
 - d'émuler des relations d'héritage
 - d'émuler le polymorphisme en « compile-time »
 - d'éviter des conversions implicites
- Mécanisme très différent des generics en Java

Overview

- *Introduction* au mécanisme de templates en C++
 - Livres complets existants
- Très efficace
 - Génération à la compilation
 - Liaison statique des liens
- Permet
 - de la programmation modulaire
 - d'émuler des relations d'héritage
 - d'émuler le polymorphisme en « compile-time »
 - d'éviter des conversions implicites
- Mécanisme très différent des generics en Java

Overview

- *Introduction* au mécanisme de templates en C++
 - Livres complets existants
- Très efficace
 - Génération à la compilation
 - Liaison statique des liens
- Permet
 - de la programmation modulaire
 - d'émuler des relations d'héritage
 - d'émuler le polymorphisme en « compile-time »
 - d'éviter des conversions implicites
- Mécanisme très différent des generics en Java

Overview

- *Introduction* au mécanisme de templates en C++
 - Livres complets existants
- Très efficace
 - Génération à la compilation
 - Liaison statique des liens
- Permet
 - de la programmation modulaire
 - d'émuler des relations d'héritage
 - d'émuler le polymorphisme en « compile-time »
 - d'éviter des conversions implicites
- Mécanisme très différent des generics en Java

Overview

- *Introduction* au mécanisme de templates en C++
 - Livres complets existants
- Très efficace
 - Génération à la compilation
 - Liaison statique des liens
- Permet
 - de la programmation modulaire
 - d'émuler des relations d'héritage
 - d'émuler le polymorphisme en « compile-time »
 - d'éviter des conversions implicites
- Mécanisme très différent des generics en Java

Syntaxe

■ `template<liste params> prototype`

■ Déclare le prototype

- d'une classe
- d'un membre de classe
- d'une fonction
- d'une variable

■ La liste des paramètres est une liste non-vide de « types »

Syntaxe

■ `template<liste params> prototype`

■ Déclare le prototype

- 1 d'une classe
- 2 d'un membre de classe
- 3 d'une fonction
- 4 d'une variable

■ La liste des paramètres est une liste non-vide de « types »

Syntaxe

■ `template`<liste params> prototype

■ Déclare le prototype

- 1 d'une classe
- 2 d'un membre de classe
- 3 d'une fonction
- 4 d'une variable

■ La liste des paramètres est une liste non-vide de « types »

Syntaxe

■ `template<liste params> prototype`

■ Déclare le prototype

- 1 d'une classe
- 2 d'un membre de classe
- 3 d'une fonction
- 4 d'une variable

■ La liste des paramètres est une liste non-vide de « types »

Syntaxe

■ `template<liste params> prototype`

■ Déclare le prototype

- 1 d'une classe
- 2 d'un membre de classe
- 3 d'une fonction
- 4 d'une variable

■ La liste des paramètres est une liste non-vide de « types »

Syntaxe

■ `template<liste params> prototype`

■ Déclare le prototype

- 1 d'une classe
- 2 d'un membre de classe
- 3 d'une fonction
- 4 d'une variable

■ La liste des paramètres est une liste non-vide de « types »

Syntaxe

- `template<liste params> prototype`
- Déclare le prototype
 - 1 d'une classe
 - 2 d'un membre de classe
 - 3 d'une fonction
 - 4 d'une variable
- La liste des paramètres est une liste non-vide de « types »

Exemples

```
1 template<class T> class A {};  
2  
3 A<int> a;
```

```
1 class B  
2 {  
3     template<class T> void brof(T t);  
4 };  
5  
6 B b;  
7 b.brof(2);
```

```
1 template<int N> class C {};  
2  
3 C<5> a;
```

```
1 template<class T> T gcd(T a, T b) {...};  
2  
3 int a = gcd(24, 36);
```

```
1 template<class T = double> constexpr T PI = T(atan(1) * 4);  
2  
3 double area = PI<> * r * r;
```

Implémentation (1/2)

Règle

- Le code « concret » d'un template est généré (et compilé) à la compilation, si besoin est.

Avantages

- Très efficace
- La taille d'un T est connue à la compilation
- Résolution statique des liens

Implémentation (1/2)

Règle

- Le code « concret » d'un template est généré (et compilé) à la compilation, si besoin est.

Avantages

- Très efficace
- La taille d'un T est connue à la compilation
 - son type « véritable » aussi
- Résolution statique des liens

Implémentation (1/2)

Règle

- Le code « concret » d'un template est généré (et compilé) à la compilation, si besoin est.

Avantages

- Très efficace
- La taille d'un T est connue à la compilation
 - son type « véritable » aussi
- Résolution statique des liens

Implémentation (1/2)

Règle

- Le code « concret » d'un template est généré (et compilé) à la compilation, si besoin est.

Avantages

- Très efficace
- La taille d'un T est connue à la compilation
 - son type « véritable » aussi
 - Résolution statique des liens

Implémentation (1/2)

Règle

- Le code « concret » d'un template est généré (et compilé) à la compilation, si besoin est.

Avantages

- Très efficace
- La taille d'un \mathbb{T} est connue à la compilation
 - son type « véritable » aussi
- Résolution statique des liens

Implémentation (1/2)

Règle

- Le code « concret » d'un template est généré (et compilé) à la compilation, si besoin est.

Avantages

- Très efficace
- La taille d'un T est connue à la compilation
 - son type « véritable » aussi
- Résolution statique des liens

Implémentation (2/2)

Inconvénients

■ Peut cacher des bugs

- Des bugs compilatoires ou sémantiques peuvent apparaître pour certaines spécialisations et pas d'autres
- Messages d'erreurs de compilation parfois obscurs
- Grossit la taille de l'exécutable
 - Grossit la taille du segment de code
- L'implémentation des prototypes se fait au sein du même fichier
 - Force l'open source (code au sein du fichier)
 - Évite les problèmes liés au langage C++

■ Peut poser des problèmes à la surdéfinition

- Constructeurs `Brol(T)` et `Brol(int)` empêche la création d'un `Brol<int>`

Implémentation (2/2)

Inconvénients

■ Peut cacher des bugs

- Des bugs compilatoires ou sémantiques peuvent apparaître pour certaines spécialisations et pas d'autres
- Messages d'erreurs de compilation parfois obscurs
- Grossit la taille de l'exécutable
 - Grossit la taille du segment de code
- L'implémentation des prototypes se fait au sein du même fichier
 - Force l'open source (code au sein du header)
 - Retarde les mises à jour du standard (lobbys)

■ Peut poser des problèmes à la surdéfinition

- Constructeurs `Brol(T)` et `Brol(int)` empêche la création d'un `Brol<int>`

Implémentation (2/2)

Inconvénients

- Peut cacher des bugs
 - Des bugs compilatoires ou sémantiques peuvent apparaître pour certaines spécialisations et pas d'autres
 - Messages d'erreurs de compilation parfois obscurs
 - Grossit la taille de l'exécutable
 - Grossit la taille du segment de code
 - L'implémentation des prototypes se fait au sein du même fichier
 - Force l'open source (code au sein du header)
 - Retarde les mises à jour du standard (lobbys)
- Peut poser des problèmes à la surdéfinition
 - Constructeurs `Brol(T)` et `Brol(int)` empêche la création d'un `Brol<int>`

Implémentation (2/2)

Inconvénients

- Peut cacher des bugs
 - Des bugs compilatoires ou sémantiques peuvent apparaître pour certaines spécialisations et pas d'autres
 - Messages d'erreurs de compilation parfois obscurs
 - Grossit la taille de l'exécutable
 - Grossit la taille du segment de code
 - L'implémentation des prototypes se fait au sein du même fichier
 - Force l'open source (code au sein du header)
 - Retarde les mises à jour du standard (lobbys)
- Peut poser des problèmes à la surdéfinition
 - Constructeurs `Brol(T)` et `Brol(int)` empêche la création d'un `Brol<int>`

Implémentation (2/2)

Inconvénients

- Peut cacher des bugs
 - Des bugs compilatoires ou sémantiques peuvent apparaître pour certaines spécialisations et pas d'autres
 - Messages d'erreurs de compilation parfois obscurs
 - Grossit la taille de l'exécutable
 - Grossit la taille du segment de code
 - L'implémentation des prototypes se fait au sein du même fichier
 - Force l'open source (code au sein du header)
 - Retarde les mises à jour du standard (lobbys)
- Peut poser des problèmes à la surdéfinition
 - Constructeurs `Brol(T)` et `Brol(int)` empêche la création d'un `Brol<int>`

Implémentation (2/2)

Inconvénients

- Peut cacher des bugs
 - Des bugs compilatoires ou sémantiques peuvent apparaître pour certaines spécialisations et pas d'autres
 - Messages d'erreurs de compilation parfois obscurs
 - Grossit la taille de l'exécutable
 - Grossit la taille du segment de code
 - L'implémentation des prototypes se fait au sein du même fichier
 - Force l'open source (code au sein du header)
 - Retarde les mises à jour du standard (lobbys)
- Peut poser des problèmes à la surdéfinition
 - Constructeurs `Brol(T)` et `Brol(int)` empêche la création d'un `Brol<int>`

Implémentation (2/2)

Inconvénients

- Peut cacher des bugs
 - Des bugs compilatoires ou sémantiques peuvent apparaître pour certaines spécialisations et pas d'autres
 - Messages d'erreurs de compilation parfois obscurs
 - Grossit la taille de l'exécutable
 - Grossit la taille du segment de code
 - L'implémentation des prototypes se fait au sein du même fichier
 - Force l'open source (code au sein du header)
 - Retarde les mises à jour du standard (lobbys)
- Peut poser des problèmes à la surdéfinition
 - Constructeurs `Brol(T)` et `Brol(int)` empêche la création d'un `Brol<int>`

Implémentation (2/2)

Inconvénients

- Peut cacher des bugs
 - Des bugs compilatoires ou sémantiques peuvent apparaître pour certaines spécialisations et pas d'autres
 - Messages d'erreurs de compilation parfois obscurs
 - Grossit la taille de l'exécutable
 - Grossit la taille du segment de code
 - L'implémentation des prototypes se fait au sein du même fichier
 - Force l'open source (code au sein du header)
 - Retarde les mises à jour du standard (lobbys)
- Peut poser des problèmes à la surdéfinition
 - Constructeurs `Brol(T)` et `Brol(int)` empêche la création d'un `Brol<int>`

Implémentation (2/2)

Inconvénients

- Peut cacher des bugs
 - Des bugs compilatoires ou sémantiques peuvent apparaître pour certaines spécialisations et pas d'autres
 - Messages d'erreurs de compilation parfois obscurs
 - Grossit la taille de l'exécutable
 - Grossit la taille du segment de code
 - L'implémentation des prototypes se fait au sein du même fichier
 - Force l'open source (code au sein du header)
 - Retarde les mises à jour du standard (lobbys)
- Peut poser des problèmes à la surdéfinition
 - Constructeurs `Brol(T)` et `Brol(int)` empêche la création d'un `Brol<int>`

Implémentation (2/2)

Inconvénients

- Peut cacher des bugs
 - Des bugs compilatoires ou sémantiques peuvent apparaître pour certaines spécialisations et pas d'autres
 - Messages d'erreurs de compilation parfois obscurs
 - Grossit la taille de l'exécutable
 - Grossit la taille du segment de code
 - L'implémentation des prototypes se fait au sein du même fichier
 - Force l'open source (code au sein du header)
 - Retarde les mises à jour du standard (lobbys)
- Peut poser des problèmes à la surdéfinition
 - Constructeurs `Brol(T)` et `Brol(int)` empêche la création d'un `Brol<int>`

Implémentation (2/2)

Inconvénients

- Peut cacher des bugs
 - Des bugs compilatoires ou sémantiques peuvent apparaître pour certaines spécialisations et pas d'autres
 - Messages d'erreurs de compilation parfois obscurs
 - Grossit la taille de l'exécutable
 - Grossit la taille du segment de code
 - L'implémentation des prototypes se fait au sein du même fichier
 - Force l'open source (code au sein du header)
 - Retarde les mises à jour du standard (lobbys)
- Peut poser des problèmes à la surdéfinition
 - Constructeurs `Brol(T)` et `Brol(int)` empêche la création d'un `Brol<int>`

Utilisation : programmation générique

■ Fichier Array.hpp

```
1  template<class T, int N> class Array
2  {
3      T * t;
4
5      public:
6          Array() : t(new T[N]) {}
7          Array(initializer_list<T> l) : Array<T,N>()
8          {
9              auto it = l.begin();
10             for(unsigned i = 0; i < N; i++)
11                 {
12                     t[i] = *it;
13                     it++;
14                 }
15             }
16
17     ~Array() { delete[] t; }
18
19     Array(const Array<T,N>& a) = delete;
20     Array<T,N>& operator=(const Array<T,N>& a) = delete;
21
22     T& operator[](unsigned i) { return t[i]; }
23     const T& operator[](unsigned i) const { return t[i]; }
24
25     unsigned size() const { return N; }
26 };
```

Utilisation : éviter des conversions

■ Fichier `arithmetic.hpp`

```
1  template<class T = double> constexpr T pi = T(atan(1) * 4);
2  constexpr double PI = pi<>;
3
4  template<class Integer>
5  constexpr Integer gcd(Integer a, Integer b)
6  {
7      if (b == 0)
8          return a;
9      else
10         return gcd(b, a % b);
11 }
12
13 template<class Integer>
14 constexpr Integer lcm(Integer a, Integer b)
15 {
16     return (a / gcd(a, b)) * b;
17 }
```

Utilisation : émulation de polymorphisme

■ Fichier `print_container.cpp`

```
1  template<class Container>
2  void print(const Container& c)
3  {
4      for(const auto & o : c)
5          cout << o << " ";
6  }
7
8  int main()
9  {
10     vector<int> v = {1,2,3,4,5};
11     list<double> l = {1.1, 2.2, 3.3, 4.4, 5.5};
12
13     print(v);
14     cout << endl;
15
16     print(l);
17     cout << endl;
18 }
```


Différences entre generics (Java) et templates

- En Java,
 - les `T` sont interprétés comme des `Object`
 - la résolution des liens et types est dynamique pour les `T`

Problème

- Comment effectuer des casts dégradants (`Object` \rightarrow Concret) ?
 - Manipulation non portable
- Comment allouer la mémoire lors d'un `new` / tableau ?
 - Opéré pour les directions

Différences entre generics (Java) et templates

- En Java,
 - les `T` sont interprétés comme des `Object`
 - la résolution des liens et types est dynamique pour les `T`

Problème

- Comment effectuer des casts dégradants (`Object` \rightarrow Concret) ?
 - Manipulation non portable
- Comment allouer la mémoire lors d'un `new` / tableau ?
 - Opéré pour les dimensions

Différences entre generics (Java) et templates

- En Java,
 - les `T` sont interprétés comme des `Object`
 - la résolution des liens et types est dynamique pour les `T`

Problème

- Comment effectuer des casts dégradants (`Object` \rightarrow Concret) ?
→ Manipulation des pointers
- Comment allouer la mémoire lors d'un `new` / tableau ?
→ Opérations sur les adresses

Différences entre generics (Java) et templates

- En Java,
 - les `T` sont interprétés comme des `Object`
 - la résolution des liens et types est dynamique pour les `T`

Problème

- Comment effectuer des casts dégradants (`Object` \rightarrow `Concret`) ?
 - Manuellement, avec perte possible
- Comment allouer la mémoire lors d'un `new` / tableau ?
 - On ne peut pas directement

Différences entre generics (Java) et templates

- En Java,
 - les `T` sont interprétés comme des `Object`
 - la résolution des liens et types est dynamique pour les `T`

Problème

- Comment effectuer des casts dégradants (`Object` \rightarrow Concret) ?
 - Manuellement, avec perte possible
- Comment allouer la mémoire lors d'un `new` / tableau ?
 - On ne peut pas directement

Différences entre generics (Java) et templates

- En Java,
 - les `T` sont interprétés comme des `Object`
 - la résolution des liens et types est dynamique pour les `T`

Problème

- Comment effectuer des casts dégradants (`Object` \rightarrow `Concret`) ?
 - Manuellement, avec perte possible
- Comment allouer la mémoire lors d'un `new` / tableau ?
 - On ne peut pas directement

Différences entre generics (Java) et templates

- En Java,
 - les `T` sont interprétés comme des `Object`
 - la résolution des liens et types est dynamique pour les `T`

Problème

- Comment effectuer des casts dégradants (`Object` \rightarrow `Concret`) ?
 - Manuellement, avec perte possible
- Comment allouer la mémoire lors d'un `new` / tableau ?
 - On ne peut pas directement

Différences entre generics (Java) et templates

- En Java,
 - les `T` sont interprétés comme des `Object`
 - la résolution des liens et types est dynamique pour les `T`

Problème

- Comment effectuer des casts dégradants (`Object` \rightarrow `Concret`) ?
 - Manuellement, avec perte possible
- Comment allouer la mémoire lors d'un `new` / tableau ?
 - On ne peut pas directement

En C++

- Les `T` sont interprétés sous leur type « véritable »
- La résolution des liens et types est statique pour les `T`
 - Sauf demande explicite du programmeur

Non-problème

- Le problème des casts dégradants ne se pose pas (car les types sont « véritables »)
- On peut instancier un `new T`
- On peut créer directement un tableau de `T`

En C++

- Les T sont interprétés sous leur type « véritable »
- La résolution des liens et types est statique pour les T
 - Sauf demande explicite du programmeur

Non-problème

- Le problème des casts dégradants ne se pose pas (car les types sont « véritables »)
- On peut instancier un `new T`
- On peut créer directement un tableau de T

En C++

- Les T sont interprétés sous leur type « véritable »
- La résolution des liens et types est statique pour les T
 - Sauf demande explicite du programmeur

Non-problème

- Le problème des casts dégradants ne se pose pas (car les types sont « véritables »)
- On peut instancier un `new T`
- On peut créer directement un tableau de T

En C++

- Les T sont interprétés sous leur type « véritable »
- La résolution des liens et types est statique pour les T
 - Sauf demande explicite du programmeur

Non-problème

- Le problème des casts dégradants ne se pose pas (car les types sont « véritables »)
 - Manuellement toujours possible avec opérateurs de cast
- On peut instancier un `new T`
- On peut créer directement un tableau de T
 - Car les types sont « véritables », on connaît leur taille

En C++

- Les T sont interprétés sous leur type « véritable »
- La résolution des liens et types est statique pour les T
 - Sauf demande explicite du programmeur

Non-problème

- Le problème des casts dégradants ne se pose pas (car les types sont « véritables »)
 - Manuellement toujours possible avec opérateurs de cast
- On peut instancier un `new T`
- On peut créer directement un tableau de T
 - Car les types sont « véritables », on connaît leur taille

En C++

- Les T sont interprétés sous leur type « véritable »
- La résolution des liens et types est statique pour les T
 - Sauf demande explicite du programmeur

Non-problème

- Le problème des casts dégradants ne se pose pas (car les types sont « véritables »)
 - Manuellement toujours possible avec opérateurs de cast
- On peut instancier un `new T`
- On peut créer directement un tableau de T
 - Car les types sont « véritables », on connaît leur taille

En C++

- Les T sont interprétés sous leur type « véritable »
- La résolution des liens et types est statique pour les T
 - Sauf demande explicite du programmeur

Non-problème

- Le problème des casts dégradants ne se pose pas (car les types sont « véritables »)
 - Manuellement toujours possible avec opérateurs de cast
- On peut instancier un `new T`
- On peut créer directement un tableau de T
 - Car les types sont « véritables », on connaît leur taille

En C++

- Les T sont interprétés sous leur type « véritable »
- La résolution des liens et types est statique pour les T
 - Sauf demande explicite du programmeur

Non-problème

- Le problème des casts dégradants ne se pose pas (car les types sont « véritables »)
 - Manuellement toujours possible avec opérateurs de cast
- On peut instancier un `new T`
- On peut créer directement un tableau de T
 - Car les types sont « véritables », on connaît leur taille

En C++

- Les T sont interprétés sous leur type « véritable »
- La résolution des liens et types est statique pour les T
 - Sauf demande explicite du programmeur

Non-problème

- Le problème des casts dégradants ne se pose pas (car les types sont « véritables »)
 - Manuellement toujours possible avec opérateurs de cast
- On peut instancier un `new T`
- On peut créer directement un tableau de T
 - Car les types sont « véritables », on connaît leur taille

Instanciation

Mécanisme

- À titre de compilation, il est nécessaire de différencier la *déclaration* d'un template et son *instanciation*
 - Quels sont les fonctions « concrètes » qui existent au sein de mon programme ?
 - Quelle est la fonction qui est appelée sur des paramètres donnés ?
- Un template est un « moule » à fonctions / classes
 - Pas un type, pas une fonction, ou « quoi que ce soit »
 - Cela n'existe pas au sein des fichiers objets
- L'utilisation d'un template va *l'instancier*
- Deux types d'instanciation
 - Explicite
 - Implicite

Mécanisme

- À titre de compilation, il est nécessaire de différencier la *déclaration* d'un template et son *instanciation*
 - Quels sont les fonctions « concrètes » qui existent au sein de mon programme ?
 - Quelle est la fonction qui est appelée sur des paramètres donnés ?
- Un template est un « moule » à fonctions / classes
 - Pas un type, pas une fonction, ou « quoi que ce soit »
 - Cela n'existe pas au sein des fichiers objets
- L'utilisation d'un template va *l'instancier*
- Deux types d'instanciation
 - Explicite
 - Implicite

Mécanisme

- À titre de compilation, il est nécessaire de différencier la *déclaration* d'un template et son *instanciation*
 - Quels sont les fonctions « concrètes » qui existent au sein de mon programme ?
 - Quelle est la fonction qui est appelée sur des paramètres donnés ?
- Un template est un « moule » à fonctions / classes
 - Pas un type, pas une fonction, ou « quoi que ce soit »
 - Cela n'existe pas au sein des fichiers objets
- L'utilisation d'un template va *l'instancier*
- Deux types d'instanciation
 - Explicite
 - Implicite

Mécanisme

- À titre de compilation, il est nécessaire de différencier la *déclaration* d'un template et son *instanciation*
 - Quels sont les fonctions « concrètes » qui existent au sein de mon programme ?
 - Quelle est la fonction qui est appelée sur des paramètres donnés ?
- Un template est un « moule » à fonctions / classes
 - Pas un type, pas une fonction, ou « quoi que ce soit »
 - Cela n'existe pas au sein des fichiers objets
- L'utilisation d'un template va *l'instancier*
- Deux types d'instanciation
 - Explicite
 - Implicite

Mécanisme

- À titre de compilation, il est nécessaire de différencier la *déclaration* d'un template et son *instanciation*
 - Quels sont les fonctions « concrètes » qui existent au sein de mon programme ?
 - Quelle est la fonction qui est appelée sur des paramètres donnés ?
- Un template est un « moule » à fonctions / classes
 - Pas un type, pas une fonction, ou « quoi que ce soit »
 - Cela n'existe pas au sein des fichiers objets
- L'utilisation d'un template va *l'instancier*
- Deux types d'instanciation
 - Explicite
 - Implicite

Mécanisme

- À titre de compilation, il est nécessaire de différencier la *déclaration* d'un template et son *instanciation*
 - Quels sont les fonctions « concrètes » qui existent au sein de mon programme ?
 - Quelle est la fonction qui est appelée sur des paramètres donnés ?
- Un template est un « moule » à fonctions / classes
 - Pas un type, pas une fonction, ou « quoi que ce soit »
 - Cela n'existe pas au sein des fichiers objets
- L'utilisation d'un template va *l'instancier*
- Deux types d'instanciation
 - Explicite
 - Implicite

Mécanisme

- À titre de compilation, il est nécessaire de différencier la *déclaration* d'un template et son *instanciation*
 - Quels sont les fonctions « concrètes » qui existent au sein de mon programme ?
 - Quelle est la fonction qui est appelée sur des paramètres donnés ?
- Un template est un « moule » à fonctions / classes
 - Pas un type, pas une fonction, ou « quoi que ce soit »
 - Cela n'existe pas au sein des fichiers objets
- L'utilisation d'un template va *l'instancier*
- Deux types d'instanciation
 - Explicite
 - Implicite

Mécanisme

- À titre de compilation, il est nécessaire de différencier la *déclaration* d'un template et son *instanciation*
 - Quels sont les fonctions « concrètes » qui existent au sein de mon programme ?
 - Quelle est la fonction qui est appelée sur des paramètres donnés ?
- Un template est un « moule » à fonctions / classes
 - Pas un type, pas une fonction, ou « quoi que ce soit »
 - Cela n'existe pas au sein des fichiers objets
- L'utilisation d'un template va *l'instancier*
- Deux types d'instanciation
 - 1 Explicite
 - 2 Implicite

Mécanisme

- À titre de compilation, il est nécessaire de différencier la *déclaration* d'un template et son *instanciation*
 - Quels sont les fonctions « concrètes » qui existent au sein de mon programme ?
 - Quelle est la fonction qui est appelée sur des paramètres donnés ?
- Un template est un « moule » à fonctions / classes
 - Pas un type, pas une fonction, ou « quoi que ce soit »
 - Cela n'existe pas au sein des fichiers objets
- L'utilisation d'un template va *l'instancier*
- Deux types d'instanciation
 - 1 Explicite
 - 2 Implicite

Mécanisme

- À titre de compilation, il est nécessaire de différencier la *déclaration* d'un template et son *instanciation*
 - Quels sont les fonctions « concrètes » qui existent au sein de mon programme ?
 - Quelle est la fonction qui est appelée sur des paramètres donnés ?
- Un template est un « moule » à fonctions / classes
 - Pas un type, pas une fonction, ou « quoi que ce soit »
 - Cela n'existe pas au sein des fichiers objets
- L'utilisation d'un template va *l'instancier*
- Deux types d'instanciation
 - 1 Explicite
 - 2 Implicite

Instanciation explicite

- Un template peut être instancié explicitement

Exemple

```
■ template<class T> T factorial(T t);  
■ template int factorial(int t);
```

- Avantages :
 - Évite des instanciations implicites non désirées
 - Évite l'ambiguïté / redéclaration
- Inconvénient : verbeux
- Remarque : une instanciation explicite d'un template avec des arguments par défaut n'utilise pas les arguments
 - Ne tente pas de les utiliser

Instanciation explicite

- Un template peut être instancié explicitement

Exemple

```
■ template<class T> T factorial(T t);  
■ template int factorial(int t);
```

- Avantages :
 - Évite des instanciations implicites non désirées
 - Évite l'ambiguïté / redéclaration
- Inconvénient : verbeux
- Remarque : une instanciation explicite d'un template avec des arguments par défaut n'utilise pas les arguments
 - Ne tente pas de les utiliser

Instanciation explicite

- Un template peut être instancié explicitement

Exemple

```
■ template<class T> T factorial(T t);  
■ template int factorial(int t);
```

- Avantages :
 - Évite des instanciations implicites non désirées
 - Évite l'ambiguïté / redéclaration
- Inconvénient : verbeux
- Remarque : une instanciation explicite d'un template avec des arguments par défaut n'utilise pas les arguments
 - Ne tente pas de les utiliser

Instanciation explicite

- Un template peut être instancié explicitement

Exemple

```
■ template<class T> T factorial(T t);  
■ template int factorial(int t);
```

- Avantages :
 - Évite des instanciations implicites non désirées
 - Évite l'ambiguïté / redéclaration
- Inconvénient : verbeux
- Remarque : une instanciation explicite d'un template avec des arguments par défaut n'utilise pas les arguments
 - Ne tente pas de les utiliser

Instanciation explicite

- Un template peut être instancié explicitement

Exemple

```
■ template<class T> T factorial(T t);  
■ template int factorial(int t);
```

- Avantages :

- Évite des instanciations implicites non désirées
- Évite l'ambiguïté / redéclaration

- Inconvénient : verbeux

- Remarque : une instanciation explicite d'un template avec des arguments par défaut n'utilise pas les arguments
 - Ne tente pas de les utiliser

Instanciation explicite

- Un template peut être instancié explicitement

Exemple

```
■ template<class T> T factorial(T t);  
■ template int factorial(int t);
```

- Avantages :
 - Évite des instanciations implicites non désirées
 - Évite l'ambiguïté / redéclaration
- Inconvénient : verbeux
- Remarque : une instanciation explicite d'un template avec des arguments par défaut n'utilise pas les arguments
 - Ne tente pas de les utiliser

Instanciation explicite

- Un template peut être instancié explicitement

Exemple

```
■ template<class T> T factorial(T t);  
■ template int factorial(int t);
```

- Avantages :
 - Évite des instanciations implicites non désirées
 - Évite l'ambiguïté / redéclaration
- Inconvénient : verbeux
- Remarque : une instanciation explicite d'un template avec des arguments par défaut n'utilise pas les arguments
 - Ne tente pas de les utiliser

Instanciation explicite

- Un template peut être instancié explicitement

Exemple

```
■ template<class T> T factorial(T t);  
■ template int factorial(int t);
```

- Avantages :
 - Évite des instanciations implicites non désirées
 - Évite l'ambiguïté / redéclaration
- Inconvénient : verbeux
- Remarque : une instanciation explicite d'un template avec des arguments par défaut n'utilise pas les arguments
 - Ne tente pas de les utiliser

Instanciation explicite

- Un template peut être instancié explicitement

Exemple

```
■ template<class T> T factorial(T t);  
■ template int factorial(int t);
```

- Avantages :
 - Évite des instanciations implicites non désirées
 - Évite l'ambiguïté / redéclaration
- Inconvénient : verbeux
- Remarque : une instanciation explicite d'un template avec des arguments par défaut n'utilise pas les arguments
 - Ne tente pas de les utiliser

Instanciation explicite

- Un template peut être instancié explicitement

Exemple

```
■ template<class T> T factorial(T t);  
■ template int factorial(int t);
```

- Avantages :
 - Évite des instanciations implicites non désirées
 - Évite l'ambiguïté / redéclaration
- Inconvénient : verbeux
- Remarque : une instanciation explicite d'un template avec des arguments par défaut n'utilise pas les arguments
 - Ne tente pas de les utiliser

Exemple

■ Fichier `explicit.cpp`

```
1  template<class T>
2  void f(T s)
3  {
4      cout << s << endl;
5  }
6
7  template void f<double>(double); // instantiates f<double>(double)
8  template void f<>(char); // instantiates f<char>(char), template argument deduced
9  template void f(int); // instantiates f<int>(int), template argument deduced
10
11 //not a use of default arguments
12 char* p = 0;
13 template<class T> T g(T x = &p) { return x; }
14 template int g<int>(int); // OK even though &p isn't an int.
```

■ Source : `cppreference`

Instanciation implicite

Pré-requis

- 1 Une instruction requiert que la définition d'une fonction template existe
 - 2 Ladite fonction n'est pas instanciée explicitement
- Par ex. : un appel requiert que la fonction existe
 - Déduction des arguments template possible
 - Dépend du contexte
 - Avantage : écriture concise
 - Inconvénient : il n'est pas toujours facile de savoir *a priori* si la déduction d'arguments va fonctionner
 - Mais si elle échoue, le code ne compile pas

Instanciation implicite

Pré-requis

- 1 Une instruction requiert que la définition d'une fonction template existe
 - 2 Ladite fonction n'est pas instanciée explicitement
- Par ex. : un appel requiert que la fonction existe
 - Déduction des arguments template possible
 - Dépend du contexte
 - Avantage : écriture concise
 - Inconvénient : il n'est pas toujours facile de savoir *a priori* si la déduction d'arguments va fonctionner
 - Mais si elle échoue, le code ne compile pas

Instanciation implicite

Pré-requis

- 1 Une instruction requiert que la définition d'une fonction template existe
 - 2 Ladite fonction n'est pas instanciée explicitement
- Par ex. : un appel requiert que la fonction existe
 - Déduction des arguments template possible
 - Dépend du contexte
 - Avantage : écriture concise
 - Inconvénient : il n'est pas toujours facile de savoir *a priori* si la déduction d'arguments va fonctionner
 - Mais si elle échoue, le code ne compile pas

Instanciation implicite

Pré-requis

- 1 Une instruction requiert que la définition d'une fonction template existe
 - 2 Ladite fonction n'est pas instanciée explicitement
- Par ex. : un appel requiert que la fonction existe
 - Déduction des arguments template possible
 - Dépend du contexte
 - Avantage : écriture concise
 - Inconvénient : il n'est pas toujours facile de savoir *a priori* si la déduction d'arguments va fonctionner
 - Mais si elle échoue, le code ne compile pas

Instanciation implicite

Pré-requis

- 1 Une instruction requiert que la définition d'une fonction template existe
 - 2 Ladite fonction n'est pas instanciée explicitement
- Par ex. : un appel requiert que la fonction existe
 - Déduction des arguments template possible
 - Dépend du contexte
 - Avantage : écriture concise
 - Inconvénient : il n'est pas toujours facile de savoir *a priori* si la déduction d'arguments va fonctionner
 - Mais si elle échoue, le code ne compile pas

Instanciation implicite

Pré-requis

- 1 Une instruction requiert que la définition d'une fonction template existe
 - 2 Ladite fonction n'est pas instanciée explicitement
- Par ex. : un appel requiert que la fonction existe
 - Déduction des arguments template possible
 - Dépend du contexte
 - Avantage : écriture concise
 - Inconvénient : il n'est pas toujours facile de savoir *a priori* si la déduction d'arguments va fonctionner
 - Mais si elle échoue, le code ne compile pas

Instanciation implicite

Pré-requis

- 1 Une instruction requiert que la définition d'une fonction template existe
 - 2 Ladite fonction n'est pas instanciée explicitement
- Par ex. : un appel requiert que la fonction existe
 - Déduction des arguments template possible
 - Dépend du contexte
 - Avantage : écriture concise
 - Inconvénient : il n'est pas toujours facile de savoir *a priori* si la déduction d'arguments va fonctionner
 - Mais si elle échoue, le code ne compile pas

Instanciation implicite

Pré-requis

- 1 Une instruction requiert que la définition d'une fonction template existe
 - 2 Ladite fonction n'est pas instanciée explicitement
- Par ex. : un appel requiert que la fonction existe
 - Déduction des arguments template possible
 - Dépend du contexte
 - Avantage : écriture concise
 - Inconvénient : il n'est pas toujours facile de savoir *a priori* si la déduction d'arguments va fonctionner
 - Mais si elle échoue, le code ne compile pas

Instanciation implicite

Pré-requis

- 1 Une instruction requiert que la définition d'une fonction template existe
 - 2 Ladite fonction n'est pas instanciée explicitement
- Par ex. : un appel requiert que la fonction existe
 - Déduction des arguments template possible
 - Dépend du contexte
 - Avantage : écriture concise
 - Inconvénient : il n'est pas toujours facile de savoir *a priori* si la déduction d'arguments va fonctionner
 - Mais si elle échoue, le code ne compile pas

Exemple

■ Fichier `implicit.cpp`

```
1  template<class T>
2  void f(T s)
3  {
4      cout << s << endl;
5  }
6
7  int main()
8  {
9      f<double>(1); // instantiates and calls f<double>(double)
10     f<>('a'); // instantiates and calls f<char>(char)
11     f(7); // instantiates and calls f<int>(int)
12     void (*ptr)(std::string) = f; // instantiates f<string>(string)
13 }
```

■ Source : `cppreference`

Template de classe

- Au même titre que les fonctions, les classes peuvent être des templates
- Différence entre déclaration et instanciation
 - Instanciation explicite ou implicite
- Similairement aux fonctions, un template est un « moule » à classes
 - N'existe pas s'il n'est pas instancié
- Similairement aux fonctions, certaines erreurs ne peuvent apparaître qu'à l'instanciation

Template de classe

- Au même titre que les fonctions, les classes peuvent être des templates
- Différence entre déclaration et instanciation
 - Instanciation explicite ou implicite
- Similairement aux fonctions, un template est un « moule » à classes
 - N'existe pas s'il n'est pas instancié
- Similairement aux fonctions, certaines erreurs ne peuvent apparaître qu'à l'instanciation

Template de classe

- Au même titre que les fonctions, les classes peuvent être des templates
- Différence entre déclaration et instanciation
 - Instanciation explicite ou implicite
- Similairement aux fonctions, un template est un « moule » à classes
 - N'existe pas s'il n'est pas instancié
- Similairement aux fonctions, certaines erreurs ne peuvent apparaître qu'à l'instanciation

Template de classe

- Au même titre que les fonctions, les classes peuvent être des templates
- Différence entre déclaration et instanciation
 - Instanciation explicite ou implicite
- Similairement aux fonctions, un template est un « moule » à classes
 - N'existe pas s'il n'est pas instancié
- Similairement aux fonctions, certaines erreurs ne peuvent apparaître qu'à l'instanciation

Template de classe

- Au même titre que les fonctions, les classes peuvent être des templates
- Différence entre déclaration et instanciation
 - Instanciation explicite ou implicite
- Similairement aux fonctions, un template est un « moule » à classes
 - N'existe pas s'il n'est pas instancié
- Similairement aux fonctions, certaines erreurs ne peuvent apparaître qu'à l'instanciation

Template de classe

- Au même titre que les fonctions, les classes peuvent être des templates
- Différence entre déclaration et instanciation
 - Instanciation explicite ou implicite
- Similairement aux fonctions, un template est un « moule » à classes
 - N'existe pas s'il n'est pas instancié
- Similairement aux fonctions, certaines erreurs ne peuvent apparaître qu'à l'instanciation

Exemple

■ Fichier inst-class.cpp

```
1 namespace N {
2     template<class T> class Y { void mf() { } }; // template definition
3 }
4 // template class Y<int>; // error: class template Y not visible in the global namespace
5 using N::Y;
6 // template class Y<int>; // error: explicit instantiation outside of the namespace
7 template class N::Y<char*>; // OK: explicit instantiation
8 template void N::Y<double>::mf(); // OK: explicit instantiation
9
10 template<class T> struct Z
11 {
12     void f() {}
13     void g(); // never defined
14 };
15 template struct Z<double>; // explicit instantiation of Z<double>
16
17 int main()
18 {
19     Z<int> a; // implicit instantiation of Z<int>
20     Z<char*> p; // nothing is instantiated here
21     p->f(); // implicit instantiation of Z<char*> and Z<char*>::f() occurs here.
22     // Z<char*>::g() is never needed and never instantiated: it does not have to be defined
23 }
```

■ Source : cppreference

Dédution d'arguments

Introduction

- À l'instanciation, tous les arguments templates doivent être connus
 - Mais ils ne doivent pas tous être spécifiés
- Si possible, le compilateur déduit les arguments manquants
- Ce processus déduction est mis en œuvre quand
 - un appel de fonction est effectué
 - l'adresse d'une fonction est acquise
- Les arguments sont déduits du contexte, si possible
- La déduction est effectuée

Introduction

- À l'instanciation, tous les arguments templates doivent être connus
 - Mais ils ne doivent pas tous être spécifiés
- Si possible, le compilateur déduit les arguments manquants
- Ce processus déduction est mis en œuvre quand
 - un appel de fonction est effectué
 - l'adresse d'une fonction est acquise
- Les arguments sont déduits du contexte, si possible
- La déduction est effectuée

Introduction

- À l'instanciation, tous les arguments templates doivent être connus
 - Mais ils ne doivent pas tous être spécifiés
- Si possible, le compilateur déduit les arguments manquants
- Ce processus déduction est mis en œuvre quand
 - un appel de fonction est effectué
 - l'adresse d'une fonction est acquise
- Les arguments sont déduits du contexte, si possible
- La déduction est effectuée

Introduction

- À l'instanciation, tous les arguments templates doivent être connus
 - Mais ils ne doivent pas tous être spécifiés
- Si possible, le compilateur déduit les arguments manquants
- Ce processus déduction est mis en œuvre quand
 - 1 un appel de fonction est effectué
 - 2 l'adresse d'une fonction est acquise
- Les arguments sont déduits du contexte, si possible
- La déduction est effectuée

Introduction

- À l'instanciation, tous les arguments templates doivent être connus
 - Mais ils ne doivent pas tous être spécifiés
- Si possible, le compilateur déduit les arguments manquants
- Ce processus déduction est mis en œuvre quand
 - 1 un appel de fonction est effectué
 - 2 l'adresse d'une fonction est acquise
- Les arguments sont déduits du contexte, si possible
- La déduction est effectuée

Introduction

- À l'instanciation, tous les arguments templates doivent être connus
 - Mais ils ne doivent pas tous être spécifiés
- Si possible, le compilateur déduit les arguments manquants
- Ce processus déduction est mis en œuvre quand
 - 1 un appel de fonction est effectué
 - 2 l'adresse d'une fonction est acquise
- Les arguments sont déduits du contexte, si possible
- La déduction est effectuée

Introduction

- À l'instanciation, tous les arguments templates doivent être connus
 - Mais ils ne doivent pas tous être spécifiés
- Si possible, le compilateur déduit les arguments manquants
- Ce processus déduction est mis en œuvre quand
 - 1 un appel de fonction est effectué
 - 2 l'adresse d'une fonction est acquise
- Les arguments sont déduits du contexte, si possible
- La déduction est effectuée

Introduction

- À l'instanciation, tous les arguments templates doivent être connus
 - Mais ils ne doivent pas tous être spécifiés
- Si possible, le compilateur déduit les arguments manquants
- Ce processus déduction est mis en œuvre quand
 - 1 un appel de fonction est effectué
 - 2 l'adresse d'une fonction est acquise
- Les arguments sont déduits du contexte, si possible
- La déduction est effectuée

Les différentes sortes de paramètres template

■ Les paramètres templates sont de trois sortes

- 1 Les paramètres templates de templates (TT)
- 2 Les paramètres templates de type (T)
- 3 Les paramètres templates qui ne sont pas de types (I)

Les différentes sortes de paramètres template

- Les paramètres templates sont de trois sortes
 - 1 Les paramètres templates de templates (TT)
 - 2 Les paramètres templates de type (T)
 - 3 Les paramètres templates qui ne sont pas de types (I)

Les différentes sortes de paramètres template

- Les paramètres templates sont de trois sortes
 - 1 Les paramètres templates de templates (TT)
 - 2 Les paramètres templates de type (T)
 - 3 Les paramètres templates qui ne sont pas de types (I)

Les différentes sortes de paramètres template

- Les paramètres templates sont de trois sortes
 - 1 Les paramètres templates de templates (TT)
 - 2 Les paramètres templates de type (T)
 - 3 Les paramètres templates qui ne sont pas de types (I)

Les différentes sortes de paramètres template

■ Les paramètres templates sont de trois sortes

- 1 Les paramètres templates de templates (TT)
- 2 Les paramètres templates de type (T)
- 3 Les paramètres templates qui ne sont pas de types (I)

Paramètre template non type

Paramètre template de template

Paramètre template de type

```
template<int i , template<class> class Collection , class Element>
const Element& get( Collection<Element>& c)
{
    return c[i];
}
```

```
Array<int> a = {1,2,3,4,5};
cout << (get<2>(a)) << endl;
```

Déduction des arguments templates d'un type

- Pour déduire les arguments template d'un type, on possède deux informations
 - 1 Le type-id « d'origine » P, combinaison de TTs, de Ts et de ls
 - 2 Le type-id « sortie » A, sans paramètres template
- Le compilateur tente de trouver
 - un template de classe pour TT
 - un type pour l
 - une valeur pour l
- But : obtenir $P = A$

Exemple

- Sur l'exemple précédent, on a
 - $P = \text{int}, \text{Collection}\langle \text{Element} \rangle$
 - $A = 2, \text{Array}\langle \text{int} \rangle$

Dédution des arguments templates d'un type

- Pour déduire les arguments template d'un type, on possède deux informations
 - 1 Le type-id « d'origine » P, combinaison de TTs, de Ts et de Is
 - 2 Le type-id « sortie » A, sans paramètres template
- Le compilateur tente de trouver
 - un template de classe pour TT
 - un type pour l
 - une valeur pour l
- But : obtenir $P = A$

Exemple

- Sur l'exemple précédent, on a
 - $P = \text{int}, \text{Collection}\langle \text{Element} \rangle$
 - $A = 2, \text{Array}\langle \text{int} \rangle$

Dédution des arguments templates d'un type

- Pour déduire les arguments template d'un type, on possède deux informations
 - 1 Le type-id « d'origine » P, combinaison de TTs, de Ts et de Is
 - 2 Le type-id « sortie » A, sans paramètres template
- Le compilateur tente de trouver
 - un template de classe pour TT
 - un type pour l
 - une valeur pour l
- But : obtenir $P = A$

Exemple

- Sur l'exemple précédent, on a
 - $P = \text{int}, \text{Collection}\langle \text{Element} \rangle$
 - $A = 2, \text{Array}\langle \text{int} \rangle$

Dédution des arguments templates d'un type

- Pour déduire les arguments template d'un type, on possède deux informations
 - 1 Le type-id « d'origine » P, combinaison de TTs, de Ts et de Is
 - 2 Le type-id « sortie » A, sans paramètres template
- Le compilateur tente de trouver
 - un template de classe pour TT
 - un type pour l
 - une valeur pour l
- But : obtenir $P = A$

Exemple

- Sur l'exemple précédent, on a
 - $P = \text{int}, \text{Collection}\langle \text{Element} \rangle$
 - $A = 2, \text{Array}\langle \text{int} \rangle$

Dédution des arguments templates d'un type

- Pour déduire les arguments template d'un type, on possède deux informations
 - 1 Le type-id « d'origine » P, combinaison de TTs, de Ts et de Is
 - 2 Le type-id « sortie » A, sans paramètres template
- Le compilateur tente de trouver
 - un template de classe pour TT
 - un type pour I
 - une valeur pour I
- But : obtenir $P = A$

Exemple

- Sur l'exemple précédent, on a
 - `P = int, Collection<Element>`
 - `A = 2, Array<int>`

Dédution des arguments templates d'un type

- Pour déduire les arguments template d'un type, on possède deux informations
 - 1 Le type-id « d'origine » P, combinaison de TTs, de Ts et de Is
 - 2 Le type-id « sortie » A, sans paramètres template
- Le compilateur tente de trouver
 - un template de classe pour TT
 - un type pour I
 - une valeur pour l
- But : obtenir $P = A$

Exemple

- Sur l'exemple précédent, on a
 - $P = \text{int}, \text{Collection}\langle \text{Element} \rangle$
 - $A = 2, \text{Array}\langle \text{int} \rangle$

Déduction des arguments templates d'un type

- Pour déduire les arguments template d'un type, on possède deux informations
 - 1 Le type-id « d'origine » P, combinaison de TTs, de Ts et de Is
 - 2 Le type-id « sortie » A, sans paramètres template
- Le compilateur tente de trouver
 - un template de classe pour TT
 - un type pour I
 - une valeur pour I
- But : obtenir $P = A$

Exemple

- Sur l'exemple précédent, on a
 - `P = int, Collection<Element>`
 - `A = 2, Array<int>`

Dédution des arguments templates d'un type

- Pour déduire les arguments template d'un type, on possède deux informations
 - 1 Le type-id « d'origine » P, combinaison de TTs, de Ts et de Is
 - 2 Le type-id « sortie » A, sans paramètres template
- Le compilateur tente de trouver
 - un template de classe pour TT
 - un type pour I
 - une valeur pour I
- But : obtenir $P = A$

Exemple

- Sur l'exemple précédent, on a
 - `P = int, Collection<Element>`
 - `A = 2, Array<int>`

Déduction des arguments templates d'un type

- Pour déduire les arguments template d'un type, on possède deux informations
 - 1 Le type-id « d'origine » P, combinaison de TTs, de Ts et de Is
 - 2 Le type-id « sortie » A, sans paramètres template
- Le compilateur tente de trouver
 - un template de classe pour TT
 - un type pour I
 - une valeur pour I
- But : obtenir $P = A$

Exemple

- Sur l'exemple précédent, on a
 - `P = int, Collection<Element>`
 - `TT = Array<int>, T = int, I = 2`
 - `A = 2, Array<int>`

Déduction des arguments templates d'un type

- Pour déduire les arguments template d'un type, on possède deux informations
 - 1 Le type-id « d'origine » P, combinaison de TTs, de Ts et de Is
 - 2 Le type-id « sortie » A, sans paramètres template
- Le compilateur tente de trouver
 - un template de classe pour TT
 - un type pour I
 - une valeur pour I
- But : obtenir $P = A$

Exemple

- Sur l'exemple précédent, on a
 - $P = \text{int}, \text{Collection}<\text{Element}>$
 - $\text{TT} = \text{Array}<\text{int}>, T = \text{int}, I = 2$
 - $A = 2, \text{Array}<\text{int}>$

Déduction des arguments templates d'un type

- Pour déduire les arguments template d'un type, on possède deux informations
 - 1 Le type-id « d'origine » P, combinaison de TTs, de Ts et de Is
 - 2 Le type-id « sortie » A, sans paramètres template
- Le compilateur tente de trouver
 - un template de classe pour TT
 - un type pour I
 - une valeur pour I
- But : obtenir $P = A$

Exemple

- Sur l'exemple précédent, on a
 - $P = \text{int}, \text{Collection}<\text{Element}>$
 - $\text{TT} = \text{Array}<\text{int}>, T = \text{int}, I = 2$
 - $A = 2, \text{Array}<\text{int}>$

Déduction des arguments templates d'un type

- Pour déduire les arguments template d'un type, on possède deux informations
 - 1 Le type-id « d'origine » P, combinaison de TTs, de Ts et de Is
 - 2 Le type-id « sortie » A, sans paramètres template
- Le compilateur tente de trouver
 - un template de classe pour TT
 - un type pour I
 - une valeur pour I
- But : obtenir $P = A$

Exemple

- Sur l'exemple précédent, on a
 - $P = \text{int}, \text{Collection}<\text{Element}>$
 - $\text{TT} = \text{Array}<\text{int}>, T = \text{int}, I = 2$
 - $A = 2, \text{Array}<\text{int}>$

Exemple

■ Fichier deduct.cpp

```
1  template<int i, template<class> class Collection, class Element>
2  Element& get(Collection<Element>& c)
3  {
4      return c[i];
5  }
6
7  template<class E>
8  class Array
9  {
10     E* a;
11     int _size;
12
13     public:
14         ...
15
16         E& operator[](int i) { return a[i]; }
17 };
18
19 int main()
20 {
21     Array<int> a = {1,2,3,4,5};
22     cout << (get<2>(a)) << endl;
23 }
```

Dédution d'un type

- En pratique, A peut être n'importe quel type C++
- P peut être un T, un T*, un T&, un TT<T>, etc.
 - Liste exhaustive pour P : http://en.cppreference.com/w/cpp/language/template_argument_deduction
- Sur l'exemple précédent, « à l'évidence », la valeur 2 ne peut pas être déduite
 - Mais `Array<int>` le peut
- La déduction de type est effectuée
 - à l'appel d'une fonction
 - à la prise d'adresse d'une fonction
 - à l'instanciation explicite d'un template
 - quand l'inférence de type est utilisée, etc.

Dédution d'un type

- En pratique, A peut être n'importe quel type C++
- P peut être un T, un T*, un T&, un TT<T>, etc.
 - Liste exhaustive pour P : http://en.cppreference.com/w/cpp/language/template_argument_deduction
- Sur l'exemple précédent, « à l'évidence », la valeur 2 ne peut pas être déduite
 - Mais `Array<int>` le peut
- La déduction de type est effectuée
 - à l'appel d'une fonction
 - à la prise d'adresse d'une fonction
 - à l'instanciation explicite d'un template
 - quand l'inférence de type est utilisée, etc.

Dédution d'un type

- En pratique, A peut être n'importe quel type C++
- P peut être un T, un T*, un T&, un TT<T>, etc.
 - Liste exhaustive pour P : http://en.cppreference.com/w/cpp/language/template_argument_deduction
- Sur l'exemple précédent, « à l'évidence », la valeur 2 ne peut pas être déduite
 - Mais `Array<int>` le peut
- La déduction de type est effectuée
 - à l'appel d'une fonction
 - à la prise d'adresse d'une fonction
 - à l'instanciation explicite d'un template
 - quand l'inférence de type est utilisée, etc.

Dédution d'un type

- En pratique, A peut être n'importe quel type C++
- P peut être un T, un T*, un T&, un TT<T>, etc.
 - Liste exhaustive pour P : http://en.cppreference.com/w/cpp/language/template_argument_deduction
- Sur l'exemple précédent, « à l'évidence », la valeur 2 ne peut pas être déduite
 - Mais `Array<int>` le peut
- La déduction de type est effectuée
 - à l'appel d'une fonction
 - à la prise d'adresse d'une fonction
 - à l'instanciation explicite d'un template
 - quand l'inférence de type est utilisée, etc.

Dédution d'un type

- En pratique, A peut être n'importe quel type C++
- P peut être un T, un T*, un T&, un TT<T>, etc.
 - Liste exhaustive pour P : http://en.cppreference.com/w/cpp/language/template_argument_deduction
- Sur l'exemple précédent, « à l'évidence », la valeur 2 ne peut pas être déduite
 - Mais `Array<int>` le peut
- La déduction de type est effectuée
 - à l'appel d'une fonction
 - à la prise d'adresse d'une fonction
 - à l'instanciation explicite d'un template
 - quand l'inférence de type est utilisée, etc.

Dédution d'un type

- En pratique, A peut être n'importe quel type C++
- P peut être un T, un T*, un T&, un TT<T>, etc.
 - Liste exhaustive pour P : http://en.cppreference.com/w/cpp/language/template_argument_deduction
- Sur l'exemple précédent, « à l'évidence », la valeur 2 ne peut pas être déduite
 - Mais `Array<int>` le peut
- La déduction de type est effectuée
 - à l'appel d'une fonction
 - à la prise d'adresse d'une fonction
 - à l'instanciation explicite d'un template
 - quand l'inférence de type est utilisée, etc.

Dédution d'un type

- En pratique, A peut être n'importe quel type C++
- P peut être un T, un T*, un T&, un TT<T>, etc.
 - Liste exhaustive pour P : http://en.cppreference.com/w/cpp/language/template_argument_deduction
- Sur l'exemple précédent, « à l'évidence », la valeur 2 ne peut pas être déduite
 - Mais `Array<int>` le peut
- La déduction de type est effectuée
 - à l'appel d'une fonction
 - à la prise d'adresse d'une fonction
 - à l'instanciation explicite d'un template
 - quand l'inférence de type est utilisée, etc.

Dédution d'un type

- En pratique, A peut être n'importe quel type C++
- P peut être un T, un T*, un T&, un TT<T>, etc.
 - Liste exhaustive pour P : http://en.cppreference.com/w/cpp/language/template_argument_deduction
- Sur l'exemple précédent, « à l'évidence », la valeur 2 ne peut pas être déduite
 - Mais `Array<int>` le peut
- La déduction de type est effectuée
 - à l'appel d'une fonction
 - à la prise d'adresse d'une fonction
 - à l'instanciation explicite d'un template
 - quand l'inférence de type est utilisée, etc.

Dédution d'un type

- En pratique, A peut être n'importe quel type C++
- P peut être un T, un T*, un T&, un TT<T>, etc.
 - Liste exhaustive pour P : http://en.cppreference.com/w/cpp/language/template_argument_deduction
- Sur l'exemple précédent, « à l'évidence », la valeur 2 ne peut pas être déduite
 - Mais `Array<int>` le peut
- La déduction de type est effectuée
 - à l'appel d'une fonction
 - à la prise d'adresse d'une fonction
 - à l'instanciation explicite d'un template
 - quand l'inférence de type est utilisée, etc.

Dédution d'un type

- En pratique, A peut être n'importe quel type C++
- P peut être un T, un T*, un T&, un TT<T>, etc.
 - Liste exhaustive pour P : http://en.cppreference.com/w/cpp/language/template_argument_deduction
- Sur l'exemple précédent, « à l'évidence », la valeur 2 ne peut pas être déduite
 - Mais `Array<int>` le peut
- La déduction de type est effectuée
 - à l'appel d'une fonction
 - à la prise d'adresse d'une fonction
 - à l'instanciation explicite d'un template
 - quand l'inférence de type est utilisée, etc.

Templates amis

Amitiés

- Les déclarations de templates de classe et de fonction peuvent être déclarées amies
- Si déclaration au sein d'un template de classe, toutes les spécialisations du template sont des amies
 - Le seul cas de « transitivité » de la relation d'amitié
- Les signatures doivent correspondre exactement pour l'amitié
- Les déclarations d'amitiés ne peuvent référencer des spécialisations partielles
- Lors de l'amitié d'une spécialisation complète, les arguments par défaut ne peuvent être utilisés

Amitiés

- Les déclarations de templates de classe et de fonction peuvent être déclarées amies
- Si déclaration au sein d'un template de classe, toutes les spécialisations du template sont des amies
 - Le seul cas de « transitivité » de la relation d'amitié
- Les signatures doivent correspondre exactement pour l'amitié
- Les déclarations d'amitiés ne peuvent référencer des spécialisations partielles
- Lors de l'amitié d'une spécialisation complète, les arguments par défaut ne peuvent être utilisés

Amitiés

- Les déclarations de templates de classe et de fonction peuvent être déclarées amies
- Si déclaration au sein d'un template de classe, toutes les spécialisations du template sont des amies
 - Le seul cas de « transitivité » de la relation d'amitié
- Les signatures doivent correspondre exactement pour l'amitié
- Les déclarations d'amitiés ne peuvent référencer des spécialisations partielles
- Lors de l'amitié d'une spécialisation complète, les arguments par défaut ne peuvent être utilisés

Amitiés

- Les déclarations de templates de classe et de fonction peuvent être déclarées amies
- Si déclaration au sein d'un template de classe, toutes les spécialisations du template sont des amies
 - Le seul cas de « transitivité » de la relation d'amitié
- Les signatures doivent correspondre exactement pour l'amitié
- Les déclarations d'amitiés ne peuvent référencer des spécialisations partielles
- Lors de l'amitié d'une spécialisation complète, les arguments par défaut ne peuvent être utilisés

Amitiés

- Les déclarations de templates de classe et de fonction peuvent être déclarées amies
- Si déclaration au sein d'un template de classe, toutes les spécialisations du template sont des amies
 - Le seul cas de « transitivité » de la relation d'amitié
- Les signatures doivent correspondre exactement pour l'amitié
- Les déclarations d'amitiés ne peuvent référencer des spécialisations partielles
- Lors de l'amitié d'une spécialisation complète, les arguments par défaut ne peuvent être utilisés

Amitiés

- Les déclarations de templates de classe et de fonction peuvent être déclarées amies
- Si déclaration au sein d'un template de classe, toutes les spécialisations du template sont des amies
 - Le seul cas de « transitivité » de la relation d'amitié
- Les signatures doivent correspondre exactement pour l'amitié
- Les déclarations d'amitiés ne peuvent référencer des spécialisations partielles
- Lors de l'amitié d'une spécialisation complète, les arguments par défaut ne peuvent être utilisés

Exemple

■ Fichier friend-1.cpp

```
1  template<class T> class B;  
2  template<class T> int add(T t);  
3  
4  class A  
5  {  
6      int i;  
7  
8      public:  
9          A(int i) : i(i) {}  
10  
11         template<class T>  
12             friend class B; // every B<T> is a friend of A  
13  
14         template<class T>  
15             friend int add(T t); // every f<T> is a friend of A  
16     };  
17  
18     template<class T> class B { ... };  
19  
20     template<class T> int add(T t) { ... }
```

Exemple

■ Fichier friend-2.cpp

```
1  template<class T> class A {}; // primary
2  template<class T> class A<T*> {}; // partial
3  template<> class A<int> {}; // full
4
5  class X
6  {
7      template<class T> friend class A<T*>; // error!
8      friend class A<int>; // OK
9  };
10
11 template<class T> void f(int);
12 template<> void f<int>(int);
13
14 class Y
15 {
16     friend void f<int>(int x = 1); // error: default args not allowed
17 };
```

■ Source : cppreference

Exemple

■ Fichier friend-3.cpp

```

1  template<class T> // primary
2  struct A
3  {
4      struct C {}; void f();
5      struct D { void g(); };
6  };
7  template<> // full
8  struct A<int>
9  {
10     struct C {}; int f(); //!<= signature
11     struct D { void g(); };
12 };
13 class B // non-template
14 {
15     template<class T>
16     friend struct A<T>::C; // A<int>::C is a friend , as well as all A<T>::C
17     template<class T>
18     friend void A<T>::f(); // A<int>::f() is not a friend , because the
19                          // signatures do not match, but A<char>::f() is
20     template<class T>
21     friend void A<T>::D::g(); // A<int>::D::g() is not a friend: it is not a member
22                          // of A, and A<int>::D is not a specialization of A<T>::D
23 };

```

■ Source : cppreference

Table des matières

- 1 Généralités
- 2 Instanciation
- 3 Dédution d'arguments
- 4 Templates amis**
 - Amitié et opérateurs**
- 5 Templates variadiques
- 6 Spécialisation de templates

Amitié et opérateurs

- Utilisation classique d'amitié de template : opérateurs
 - P. ex. : injection de flux
- Deux façons de procéder
 - Opérateur non-template
 - Opérateur template

Hygiène de programmation

- Utiliser des opérateurs template

Amitié et opérateurs

- Utilisation classique d'amitié de template : opérateurs
 - P. ex. : injection de flux
- Deux façons de procéder
 - Opérateur non-template
 - Opérateur template

Hygiène de programmation

- Utiliser des opérateurs template

Amitié et opérateurs

- Utilisation classique d'amitié de template : opérateurs
 - P. ex. : injection de flux
- Deux façons de procéder
 - 1 Opérateur non-template
 - Warning
 - 2 Opérateur template

Hygiène de programmation

- Utiliser des opérateurs template

Amitié et opérateurs

- Utilisation classique d'amitié de template : opérateurs
 - P. ex. : injection de flux
- Deux façons de procéder
 - 1 Opérateur non-template
 - Warning
 - 2 Opérateur template

Hygiène de programmation

- Utiliser des opérateurs template

Amitié et opérateurs

- Utilisation classique d'amitié de template : opérateurs
 - P. ex. : injection de flux
- Deux façons de procéder
 - 1 Opérateur non-template
 - Warning
 - 2 Opérateur template

Hygiène de programmation

- Utiliser des opérateurs template

Amitié et opérateurs

- Utilisation classique d'amitié de template : opérateurs
 - P. ex. : injection de flux
- Deux façons de procéder
 - 1 Opérateur non-template
 - Warning
 - 2 Opérateur template

Hygiène de programmation

- Utiliser des opérateurs template

Amitié et opérateurs

- Utilisation classique d'amitié de template : opérateurs
 - P. ex. : injection de flux
- Deux façons de procéder
 - 1 Opérateur non-template
 - Warning
 - 2 Opérateur template

Hygiène de programmation

- Utiliser des opérateurs template
 - Moins prompt aux erreurs

Amitié et opérateurs

- Utilisation classique d'amitié de template : opérateurs
 - P. ex. : injection de flux
- Deux façons de procéder
 - 1 Opérateur non-template
 - Warning
 - 2 Opérateur template

Hygiène de programmation

- Utiliser des opérateurs template
 - Moins prompt aux erreurs

Amitié et opérateurs

- Utilisation classique d'amitié de template : opérateurs
 - P. ex. : injection de flux
- Deux façons de procéder
 - 1 Opérateur non-template
 - Warning
 - 2 Opérateur template

Hygiène de programmation

- Utiliser des opérateurs template
 - Moins prompt aux erreurs

Exemple

■ Fichier friend-op1.cpp

```
1  template<class T>
2  class A
3  {
4      T t;
5      public:
6          A(T t) : t(t) {}
7          friend ostream& operator<<(ostream& out, const A& a)
8          {
9              return out << a.t;
10         }
11     };
12
13     int main()
14     {
15         A<int> a(2);
16         cout << a << endl;
17     }
```

Exemple

■ Fichier friend-op2.cpp

```
1  template<class T>
2  class A
3  {
4      T t;
5      public:
6          A(T t) : t(t) {}
7
8          friend ostream& operator<<(ostream& out, const A& a); //warning
9  };
10
11 template<class T>
12 ostream& operator<<(ostream& out, const A<T>& a)
13 {
14     return out << a.t;
15 }
16
17 int main()
18 {
19     A<int> a(2);
20     cout << a << endl; //error
21 }
```

Exemple

- Fichier friend-op3.cpp
- Similaire à friend-op1.cpp

```
1  template<class T>
2  class A
3  {
4      T t;
5      public:
6          A(T t) : t(t) {}
7
8          template<class U>
9              friend ostream& operator<<(ostream& out, const A<U>& a);
10 };
11
12 template<class T>
13 ostream& operator<<(ostream& out, const A<T>& a)
14 {
15     return out << a.t;
16 }
17
18 int main()
19 {
20     A<int> a(2);
21     cout << a << endl;
22 }
```

Exemple

- Fichier `friend-op4.cpp`
- Opérateur template

```
1  template<class T> class A; // forward declare to make function declaration possible
2  template<class T> ostream& operator<<(ostream&, const A<T>&); // declaration
3
4  template<class T>
5  class A
6  {
7      T t;
8      public:
9          A(T t) : t(t) {}
10         // refers to a full specialization for this particular T
11         friend std::ostream& operator<< <> (std::ostream&, const A&);
12     };
13
14     // definition
15     template<typename T>
16     ostream& operator<<(ostream& out, const A<T>& a) { return out << a.t; }
17
18     int main()
19     {
20         A<int> a(2);
21         cout << a << endl;
22     }
```

Exemple

■ Fichier friend-wtf.cpp

```

1  template<class T> class A;
2  template<class T> std::ostream& operator<<(std::ostream&, const A<T>&);
3  template<class T> A<T> operator+(const T&, const A<T>&);
4
5  template<class T>
6  class A
7  {
8      T i;
9      public:
10         explicit A(T t) : i(t) {}
11         friend A<T> operator+ <>(const T& t, const A<T>& a); //YOU HAVE TO DECLARE THIS HERE
12         friend std::ostream& operator<< <>(std::ostream& out, const A<T>& a);
13         A operator +(A<T> a) const { return A(a.i + i); }
14         A operator +(T t) const { return (*this) + A<T>(t); }
15     };
16
17     template<class T>
18     std::ostream& operator<<(std::ostream& out, const A<T>& a)
19     {
20         out << a.i; return out;
21     }
22
23     template<class T>
24     A<T> operator +(const T& t, const A<T>& a) { return A<T>(t) + a; }

```

Templates variadiques

Les arguments variables en nombre

- Jusqu'à présent, en C++, on n'a pas vu de manière « satisfaisante » comment passer un nombre d'arguments variables en nombre
 - Comme en C : utiliser `va_list`
 - `std::initializer_list`
- Grâce aux templates, on peut faire mieux
- Templates variadiques
- Syntaxe : `template<class Args> A f(Args ... args)`
 - Pour des templates de classe, les arguments variadiques doivent être les derniers
 - Pour des templates de fonction, les paramètres explicites doivent être les derniers
- Autre utilité : certains templates ont des paramètres « cachés »
 - `map`, `vector`, `list`, etc.
 - On ne pouvait pas les utiliser avec `get` (Slide 24, 26)
 - Erreur de substitution

Les arguments variables en nombre

- Jusqu'à présent, en C++, on n'a pas vu de manière « satisfaisante » comment passer un nombre d'arguments variables en nombre
 - Comme en C : utiliser `va_list`
 - `std::initializer_list`
- Grâce aux templates, on peut faire mieux
- Templates variadiques
- Syntaxe : `template<class Args> A f(Args ... args)`
 - Pour des templates de classe, les arguments variadiques doivent être les derniers
 - Pour des templates de fonction, les paramètres explicites doivent être les derniers
- Autre utilité : certains templates ont des paramètres « cachés »
 - `map`, `vector`, `list`, etc.
 - On ne pouvait pas les utiliser avec `get` (Slide 24, 26)
 - Erreur de substitution

Les arguments variables en nombre

- Jusqu'à présent, en C++, on n'a pas vu de manière « satisfaisante » comment passer un nombre d'arguments variables en nombre
 - Comme en C : utiliser `va_list`
 - `std::initializer_list`
- Grâce aux templates, on peut faire mieux
- Templates variadiques
- Syntaxe : `template<class Args> A f(Args ... args)`
 - Pour des templates de classe, les arguments variadiques doivent être les derniers
 - Pour des templates de fonction, les paramètres explicites doivent être les derniers
- Autre utilité : certains templates ont des paramètres « cachés »
 - `map`, `vector`, `list`, etc.
 - On ne pouvait pas les utiliser avec `get` (Slide 24, 26)
 - Erreur de substitution

Les arguments variables en nombre

- Jusqu'à présent, en C++, on n'a pas vu de manière « satisfaisante » comment passer un nombre d'arguments variables en nombre
 - Comme en C : utiliser `va_list`
 - `std::initializer_list`
- Grâce aux templates, on peut faire mieux
- Templates variadiques
- Syntaxe : `template<class Args> A f(Args ... args)`
 - Pour des templates de classe, les arguments variadiques doivent être les derniers
 - Pour des templates de fonction, les paramètres explicites doivent être les derniers
- Autre utilité : certains templates ont des paramètres « cachés »
 - `map`, `vector`, `list`, etc.
 - On ne pouvait pas les utiliser avec `get` (Slide 24, 26)
 - Erreur de substitution

Les arguments variables en nombre

- Jusqu'à présent, en C++, on n'a pas vu de manière « satisfaisante » comment passer un nombre d'arguments variables en nombre
 - Comme en C : utiliser `va_list`
 - `std::initializer_list`
- Grâce aux templates, on peut faire mieux
- Templates variadiques
- Syntaxe : `template<class Args> A f(Args ... args)`
 - Pour des templates de classe, les arguments variadiques doivent être les derniers
 - Pour des templates de fonction, les paramètres explicites doivent être les derniers
- Autre utilité : certains templates ont des paramètres « cachés »
 - `map`, `vector`, `list`, etc.
 - On ne pouvait pas les utiliser avec `get` (Slide 24, 26)
 - Erreur de substitution

Les arguments variables en nombre

- Jusqu'à présent, en C++, on n'a pas vu de manière « satisfaisante » comment passer un nombre d'arguments variables en nombre
 - Comme en C : utiliser `va_list`
 - `std::initializer_list`
- Grâce aux templates, on peut faire mieux
- Templates variadiques
- Syntaxe : `template<class Args> A f(Args ... args)`
 - Pour des templates de classe, les arguments variadiques doivent être les derniers
 - Pour des templates de fonction, les paramètres explicites doivent être les derniers
- Autre utilité : certains templates ont des paramètres « cachés »
 - `map`, `vector`, `list`, etc.
 - On ne pouvait pas les utiliser avec `get` (Slide 24, 26)
 - Erreur de substitution

Les arguments variables en nombre

- Jusqu'à présent, en C++, on n'a pas vu de manière « satisfaisante » comment passer un nombre d'arguments variables en nombre
 - Comme en C : utiliser `va_list`
 - `std::initializer_list`
- Grâce aux templates, on peut faire mieux
- Templates variadiques
- Syntaxe : `template<class Args> A f(Args ... args)`
 - Pour des templates de classe, les arguments variadiques doivent être les derniers
 - Pour des templates de fonction, les paramètres explicites doivent être les derniers
- Autre utilité : certains templates ont des paramètres « cachés »
 - `map`, `vector`, `list`, etc.
 - On ne pouvait pas les utiliser avec `get` (Slide 24, 26)
 - Erreur de substitution

Les arguments variables en nombre

- Jusqu'à présent, en C++, on n'a pas vu de manière « satisfaisante » comment passer un nombre d'arguments variables en nombre
 - Comme en C : utiliser `va_list`
 - `std::initializer_list`
- Grâce aux templates, on peut faire mieux
- Templates variadiques
- Syntaxe : `template<class Args> A f(Args ... args)`
 - Pour des templates de classe, les arguments variadiques doivent être les derniers
 - Pour des templates de fonction, les paramètres explicites doivent être les derniers
- Autre utilité : certains templates ont des paramètres « cachés »
 - `map`, `vector`, `list`, etc.
 - On ne pouvait pas les utiliser avec `get` (Slide 24, 26)
 - Erreur de substitution

Les arguments variables en nombre

- Jusqu'à présent, en C++, on n'a pas vu de manière « satisfaisante » comment passer un nombre d'arguments variables en nombre
 - Comme en C : utiliser `va_list`
 - `std::initializer_list`
- Grâce aux templates, on peut faire mieux
- Templates variadiques
- Syntaxe : `template<class Args> A f(Args ... args)`
 - Pour des templates de classe, les arguments variadiques doivent être les derniers
 - Pour des templates de fonction, les paramètres explicites doivent être les derniers
- Autre utilité : certains templates ont des paramètres « cachés »
 - `map, vector, list, etc.`
 - On ne pouvait pas les utiliser avec `get` (Slide 24, 26)
 - Erreur de substitution

Les arguments variables en nombre

- Jusqu'à présent, en C++, on n'a pas vu de manière « satisfaisante » comment passer un nombre d'arguments variables en nombre
 - Comme en C : utiliser `va_list`
 - `std::initializer_list`
- Grâce aux templates, on peut faire mieux
- Templates variadiques
- Syntaxe : `template<class Args> A f(Args ... args)`
 - Pour des templates de classe, les arguments variadiques doivent être les derniers
 - Pour des templates de fonction, les paramètres explicites doivent être les derniers
- Autre utilité : certains templates ont des paramètres « cachés »
 - `map`, `vector`, `list`, etc.
 - On ne pouvait pas les utiliser avec `get` (Slide 24, 26)
 - Erreur de substitution

Les arguments variables en nombre

- Jusqu'à présent, en C++, on n'a pas vu de manière « satisfaisante » comment passer un nombre d'arguments variables en nombre
 - Comme en C : utiliser `va_list`
 - `std::initializer_list`
- Grâce aux templates, on peut faire mieux
- Templates variadiques
- Syntaxe : `template<class Args> A f(Args ... args)`
 - Pour des templates de classe, les arguments variadiques doivent être les derniers
 - Pour des templates de fonction, les paramètres explicites doivent être les derniers
- Autre utilité : certains templates ont des paramètres « cachés »
 - `map`, `vector`, `list`, etc.
 - On ne pouvait pas les utiliser avec `get` (Slide 24, 26)
 - Erreur de substitution

Les arguments variables en nombre

- Jusqu'à présent, en C++, on n'a pas vu de manière « satisfaisante » comment passer un nombre d'arguments variables en nombre
 - Comme en C : utiliser `va_list`
 - `std::initializer_list`
- Grâce aux templates, on peut faire mieux
- Templates variadiques
- Syntaxe : `template<class Args> A f(Args ... args)`
 - Pour des templates de classe, les arguments variadiques doivent être les derniers
 - Pour des templates de fonction, les paramètres explicites doivent être les derniers
- Autre utilité : certains templates ont des paramètres « cachés »
 - `map`, `vector`, `list`, etc.
 - On ne pouvait pas les utiliser avec `get` (Slide 24, 26)
 - Erreur de substitution

Exemple

```
1 template<class ... Types> struct Tuple {};  
2 Tuple<> t0;           // Types contains no arguments  
3 Tuple<int> t1;         // Types contains one argument: int  
4 Tuple<int, float> t2;  // Types contains two arguments: int and float  
5 Tuple<0> error;        // 0 is not a type
```

```
1 template<class ... Types> void f(Types ... args);  
2 f();           // OK: args contains no arguments  
3 f(1);          // OK: args contains one argument: int  
4 f(2, 1.0);     // OK: args contains two arguments: int and double
```

```
1 template<typename... Ts, typename U> struct Invalid; // Error: Ts.. not at the end  
2  
3 template<typename ...Ts, typename U, typename=void>  
4 void valid(U, Ts...); // OK: can deduce U  
5 //void valid(Ts..., U); // error : cannot deduce  
6  
7 valid(1.0, 1, 2, 3); // OK: deduces U as double, Ts as {int,int,int}
```

Extension de pack

- Terminologie : `Args ... args` est un pack de paramètres
- On accède aux paramètres un par un en « étendant » le pack
 - `args...` est l'expansion de pack
- L'expansion est effectuée récursivement, paramètre par paramètre
- On peut « préfixer » l'extension
 - Exemple : `&args...`
- Si packs sont imbriqués ou étendus « ensemble », les règles sont assez complexes

Extension de pack

- Terminologie : `Args ... args` est un pack de paramètres
- On accède aux paramètres un par un en « étendant » le pack
 - `args...` est l'expansion de pack
- L'expansion est effectuée récursivement, paramètre par paramètre
- On peut « préfixer » l'extension
 - Exemple : `&args...`
- Si packs sont imbriqués ou étendus « ensemble », les règles sont assez complexes

Extension de pack

- Terminologie : `Args ... args` est un pack de paramètres
- On accède aux paramètres un par un en « étendant » le pack
 - `args...` est l'expansion de pack
- L'expansion est effectuée récursivement, paramètre par paramètre
- On peut « préfixer » l'extension
 - Exemple : `&args...`
- Si packs sont imbriqués ou étendus « ensemble », les règles sont assez complexes

Extension de pack

- Terminologie : `Args ... args` est un pack de paramètres
- On accède aux paramètres un par un en « étendant » le pack
 - `args...` est l'expansion de pack
- L'expansion est effectuée récursivement, paramètre par paramètre
- On peut « préfixer » l'extension
 - Exemple : `&args...`
- Si packs sont imbriqués ou étendus « ensemble », les règles sont assez complexes

Extension de pack

- Terminologie : `Args ... args` est un pack de paramètres
- On accède aux paramètres un par un en « étendant » le pack
 - `args...` est l'expansion de pack
- L'expansion est effectuée récursivement, paramètre par paramètre
- On peut « préfixer » l'extension
 - Exemple : `&args...`
- Si packs sont imbriqués ou étendus « ensemble », les règles sont assez complexes

Extension de pack

- Terminologie : `Args ... args` est un pack de paramètres
- On accède aux paramètres un par un en « étendant » le pack
 - `args...` est l'expansion de pack
- L'expansion est effectuée récursivement, paramètre par paramètre
- On peut « préfixer » l'extension
 - Exemple : `&args...`
- Si packs sont imbriqués ou étendus « ensemble », les règles sont assez complexes

Extension de pack

- Terminologie : `Args ... args` est un pack de paramètres
- On accède aux paramètres un par un en « étendant » le pack
 - `args...` est l'expansion de pack
- L'expansion est effectuée récursivement, paramètre par paramètre
- On peut « préfixer » l'extension
 - Exemple : `&args...`
- Si packs sont imbriqués ou étendus « ensemble », les règles sont assez complexes

Exemple

■ Fichier variadic.cpp

```
1  template<typename T>
2  T sum(T v)
3  {
4      return v;
5  }
6
7  template<class T, class ... Args>
8  T sum(T first, Args... args)
9  {
10     return first + sum(args...);
11 }
12
13 int main()
14 {
15     int s = sum(1,2,3,4);
16     cout << s << endl;
17
18     string s1 = "a", s2 = "b", s3 = "c", s4 = "d";
19     string str = sum(s1, s2, s3, s4);
20     cout << str << endl;
21 }
```

Le problème de forwarding

- Une fonction `void f(int& i, int& j, int& k)` ne peut pas être appelée avec des immédiats

- Ni aucune rvalue

- On ne peut pas convertir implicitement une rvalue en rvalue

- « Solution » :

```
void f(const int& i, const int& j, const int& k)
```

- Inconvénient : on ne peut plus modifier les paramètres

- Et c'est une bonne chose si on passe des immédiats

- Si on veut en modifier certains, il ne faut mettre que certains `const`

- Sur un template variadique, c'est probablement voulu

- Par exemple, si on veut écrire une fonction qui prend en paramètre une fonction `f`, et ses arguments

- `auto call_function(Function f, Args& ... args)`

Le problème de forwarding

- Une fonction `void f(int& i, int& j, int& k)` ne peut pas être appelée avec des immédiats

- Ni aucune rvalue

- On ne peut pas convertir implicitement une rvalue en lvalue

- « Solution » :

```
void f(const int& i, const int& j, const int& k)
```

- Inconvénient : on ne peut plus modifier les paramètres

- Et c'est une bonne chose si on passe des immédiats

- Si on veut en modifier certains, il ne faut mettre que certains `const`

- Sur un template variadique, c'est probablement voulu

- Par exemple, si on veut écrire une fonction qui prend en paramètre une fonction `f`, et ses arguments

- `auto call_function(Function f, Args& ... args)`

Le problème de forwarding

- Une fonction `void f(int& i, int& j, int& k)` ne peut pas être appelée avec des immédiats

- Ni aucune rvalue

- On ne peut pas convertir implicitement une rvalue en rvalue

- « Solution » :

```
void f(const int& i, const int& j, const int& k)
```

- Inconvénient : on ne peut plus modifier les paramètres

- Et c'est une bonne chose si on passe des immédiats

- Si on veut en modifier certains, il ne faut mettre que certains `const`

- Sur un template variadique, c'est probablement voulu

- Par exemple, si on veut écrire une fonction qui prend en paramètre une fonction `f`, et ses arguments

- `auto call_function(Function f, Args& ... args)`

Le problème de forwarding

- Une fonction `void f(int& i, int& j, int& k)` ne peut pas être appelée avec des immédiats
 - Ni aucune rvalue
- On ne peut pas convertir implicitement une rvalue en lvalue
- « Solution » :
`void f(const int& i, const int& j, const int& k)`
- Inconvénient : on ne peut plus modifier les paramètres
 - Et c'est une bonne chose si on passe des immédiats
- Si on veut en modifier certains, il ne faut mettre que certains `const`
- Sur un template variadique, c'est probablement voulu
 - Par exemple, si on veut écrire une fonction qui prend en paramètre une fonction `f`, et ses arguments
 - `auto call_function(Function f, Args& ... args)`

Le problème de forwarding

- Une fonction `void f(int& i, int& j, int& k)` ne peut pas être appelée avec des immédiats
 - Ni aucune rvalue
- On ne peut pas convertir implicitement une rvalue en lvalue
- « Solution » :
`void f(const int& i, const int& j, const int& k)`
- Inconvénient : on ne peut plus modifier les paramètres
 - Et c'est une bonne chose si on passe des immédiats
- Si on veut en modifier certains, il ne faut mettre que certains `const`
- Sur un template variadique, c'est probablement voulu
 - Par exemple, si on veut écrire une fonction qui prend en paramètre une fonction `f`, et ses arguments
 - `auto call_function(Function f, Args& ... args)`

Le problème de forwarding

- Une fonction `void f(int& i, int& j, int& k)` ne peut pas être appelée avec des immédiats
 - Ni aucune rvalue
- On ne peut pas convertir implicitement une rvalue en lvalue
- « Solution » :
`void f(const int& i, const int& j, const int& k)`
- Inconvénient : on ne peut plus modifier les paramètres
 - Et c'est une bonne chose si on passe des immédiats
- Si on veut en modifier certains, il ne faut mettre que certains `const`
- Sur un template variadique, c'est probablement voulu
 - Par exemple, si on veut écrire une fonction qui prend en paramètre une fonction `f`, et ses arguments
 - `auto call_function(Function f, Args&... args)`

Le problème de forwarding

- Une fonction `void f(int& i, int& j, int& k)` ne peut pas être appelée avec des immédiats
 - Ni aucune rvalue
- On ne peut pas convertir implicitement une rvalue en lvalue
- « Solution » :
`void f(const int& i, const int& j, const int& k)`
- Inconvénient : on ne peut plus modifier les paramètres
 - Et c'est une bonne chose si on passe des immédiats
- Si on veut en modifier certains, il ne faut mettre que certains `const`
- Sur un template variadique, c'est probablement voulu
 - Par exemple, si on veut écrire une fonction qui prend en paramètre une fonction `f`, et ses arguments
 - `auto call_function(Function f, Args&... args)`

Le problème de forwarding

- Une fonction `void f(int& i, int& j, int& k)` ne peut pas être appelée avec des immédiats
 - Ni aucune rvalue
- On ne peut pas convertir implicitement une rvalue en lvalue
- « Solution » :
`void f(const int& i, const int& j, const int& k)`
- Inconvénient : on ne peut plus modifier les paramètres
 - Et c'est une bonne chose si on passe des immédiats
- Si on veut en modifier certains, il ne faut mettre que certains `const`
- Sur un template variadique, c'est probablement voulu
 - Par exemple, si on veut écrire une fonction qui prend en paramètre une fonction `f`, et ses arguments
 - `auto call_function(Function f, Args& ... args)`

Le problème de forwarding

- Une fonction `void f(int& i, int& j, int& k)` ne peut pas être appelée avec des immédiats
 - Ni aucune rvalue
- On ne peut pas convertir implicitement une rvalue en lvalue
- « Solution » :

```
void f(const int& i, const int& j, const int& k)
```
- Inconvénient : on ne peut plus modifier les paramètres
 - Et c'est une bonne chose si on passe des immédiats
- Si on veut en modifier certains, il ne faut mettre que certains `const`
- Sur un template variadique, c'est probablement voulu
 - Par exemple, si on veut écrire une fonction qui prend en paramètre une fonction `f`, et ses arguments

```
auto call_function(Function f, Args& ... args)
```

Le problème de forwarding

- Une fonction `void f(int& i, int& j, int& k)` ne peut pas être appelée avec des immédiats
 - Ni aucune rvalue
- On ne peut pas convertir implicitement une rvalue en lvalue
- « Solution » :
`void f(const int& i, const int& j, const int& k)`
- Inconvénient : on ne peut plus modifier les paramètres
 - Et c'est une bonne chose si on passe des immédiats
- Si on veut en modifier certains, il ne faut mettre que certains `const`
- Sur un template variadique, c'est probablement voulu
 - Par exemple, si on veut écrire une fonction qui prend en paramètre une fonction `f`, et ses arguments
 - `auto call_function(Function f, Args& ... args)`

Ce que l'on ne veut pas faire

■ Écrire toutes les combinaisons de const et pas const

```
1  template <class A, class B, class C>
2  void f(A& a, B& b, C& c);
3
4  template <class A, class B, class C>
5  void f(const A& a, B& b, C& c);
6
7  template <class A, class B, class C>
8  void f(A& a, const B& b, C& c);
9
10 template <class A, class B, class C>
11 void f(A& a, B& b, const C& c);
12
13 template <class A, class B, class C>
14 void f(const A& a, const B& b, C& c);
15
16 template <class A, class B, class C>
17 void f(const A& a, B& b, const C& c);
18
19 template <class A, class B, class C>
20 void f(A& a, const B& b, const C& c);
21
22 template <class A, class B, class C>
23 void f(const A& a, const B& b, const C& c);
```

■ Avec des templates variadiques, c'est impossible

Astuce : utiliser les références de rvalue

```
1  template<class Function, class ... Args>
2  auto call_function(Function f, Args&& ... args) -> decltype(f(args...))
3  {
4      return f(args...);
5  }
6
7  void print(int i) { cout << i << endl; }
8
9  void increment(int & i) { i++; }
10
11 int main()
12 {
13     call_function(print, 2);
14
15     int i = 3;
16     call_function(increment, i);
17     print(i);
18
19     call_function(increment, 3); // wtf ?!
20 }
```

- Problème : on ne veut pas que `call_function(increment, 3)` compile

Solution

■ Fichier fwd.cpp

```
1  template<class Function, class ... Args>
2  auto call_function(Function f, Args&& ... args) -> decltype(f(args...))
3  {
4      return f(forward<Args>(args)...);
5  }
6
7  void print(int i) { cout << i << endl; }
8
9  void increment(int & i) { i++; }
10
11 int main()
12 {
13     call_function(print, 2);
14
15     int i = 3;
16     call_function(increment, i);
17     print(i);
18
19     // call_function(increment, 3); //doesn't compile anymore
20 }
```

Fonctionnement

- **Rappel : && crée des références de lvalue**
- `std::forward<T>(T && t)` est résolu en
 - `T&` si `t` est une référence de rvalue
 - `std::move` sinon
- Le risque de « modification d'immédiat » est éliminé par `std::move`
- Sucre syntaxique pour `static_cast<T&&>(t)`

Fonctionnement

- **Rappel : && crée des références de lvalue**
- `std::forward<T>(T && t)` est résolu en
 - `T&` si `t` est une référence de rvalue
 - `std::move` sinon
- Le risque de « modification d'immédiat » est éliminé par `std::move`
- Sucre syntaxique pour `static_cast<T&&>(t)`

Fonctionnement

- **Rappel : && crée des références de lvalue**
- `std::forward<T>(T && t)` est résolu en
 - `T&` si `t` est une référence de rvalue
 - `std::move` sinon
- Le risque de « modification d'immédiat » est éliminé par `std::move`
- Sucre syntaxique pour `static_cast<T&&>(t)`

Fonctionnement

- **Rappel : && crée des références de lvalue**
- `std::forward<T>(T && t)` est résolu en
 - `T&` si `t` est une référence de rvalue
 - `std::move` sinon
- Le risque de « modification d'immédiat » est éliminé par `std::move`
- Sucre syntaxique pour `static_cast<T&&>(t)`

Fonctionnement

- Rappel : `&&` crée des références de lvalue
- `std::forward<T>(T && t)` est résolu en
 - `T&` si `t` est une référence de rvalue
 - `std::move` sinon
- Le risque de « modification d'immédiat » est éliminé par `std::move`
- Sucre syntaxique pour `static_cast<T&&>(t)`

Fonctionnement

- Rappel : `&&` crée des références de lvalue
- `std::forward<T>(T && t)` est résolu en
 - `T&` si `t` est une référence de rvalue
 - `std::move` sinon
- Le risque de « modification d'immédiat » est éliminé par `std::move`
- Sucre syntaxique pour `static_cast<T&&>(t)`

Différence entre `move` et `forward`

■ Fichier `move-fwd.cpp`

```
1 void overloaded(const int & arg ) { cout << "by_lvalue\n"; }
2
3 void overloaded(int && arg ) { cout << "by_rvalue\n"; }
4
5 template<class T>
6 void forwarding( T && arg )
7 {
8     cout << "by_simple_passing:_"; overloaded(arg);
9     cout << "via_std::forward:_"; overloaded( forward<T>(arg) );
10    cout << "via_std::move:_"; overloaded( move(arg) ); //arg is now invalidated
11 }
12
13 int main()
14 {
15     cout << "initial_caller_passes_rvalue:\n";
16     forwarding(5);
17     cout << endl;
18
19     cout << "initial_caller_passes_lvalue:\n";
20     int x = 5;
21     forwarding(x);
22 }
```


Spécialisation de templates

Overview

- La plupart du temps, on écrit une classe template et elle est « valide » pour tous les types
 - Parfois, on veut particulariser la classe pour certains types
- Deux types de spécialisation
 - *complète* : tous les arguments templates sont fixés
 - *partielle* : certains arguments templates sont spécialisés
- Permet de personnaliser le comportement pour certains arguments template
 - Efficacité

Exemple

- `vector` est complètement spécialisé pour les vecteurs booléens
- Économie de taille

Overview

- La plupart du temps, on écrit une classe template et elle est « valide » pour tous les types
 - Parfois, on veut particulariser la classe pour certains types
- Deux types de spécialisation
 - *complète* : tous les arguments templates sont fixés
 - *partielle* : certains arguments templates sont spécialisés
- Permet de personnaliser le comportement pour certains arguments template
 - Efficacité

Exemple

- `vector` est complètement spécialisé pour les vecteurs booléens
- Économie de taille

Overview

- La plupart du temps, on écrit une classe template et elle est « valide » pour tous les types
 - Parfois, on veut particulariser la classe pour certains types
- Deux types de spécialisation
 - 1 *complète* : tous les arguments templates sont fixés
 - 2 *partielle* : certains arguments templates sont spécialisés
- Permet de personnaliser le comportement pour certains arguments template
 - Efficacité

Exemple

- `vector` est complètement spécialisé pour les vecteurs booléens
- Économie de taille

Overview

- La plupart du temps, on écrit une classe template et elle est « valide » pour tous les types
 - Parfois, on veut particulariser la classe pour certains types
- Deux types de spécialisation
 - 1 *complète* : tous les arguments templates sont fixés
 - 2 *partielle* : certains arguments templates sont spécialisés
- Permet de personnaliser le comportement pour certains arguments template
 - Efficacité

Exemple

- `vector` est complètement spécialisé pour les vecteurs booléens
- Économie de taille

Overview

- La plupart du temps, on écrit une classe template et elle est « valide » pour tous les types
 - Parfois, on veut particulariser la classe pour certains types
- Deux types de spécialisation
 - 1 *complète* : tous les arguments templates sont fixés
 - 2 *partielle* : certains arguments templates sont spécialisés
- Permet de personnaliser le comportement pour certains arguments template
 - Efficacité

Exemple

- `vector` est complètement spécialisé pour les vecteurs booléens
- Économie de taille

Overview

- La plupart du temps, on écrit une classe template et elle est « valide » pour tous les types
 - Parfois, on veut particulariser la classe pour certains types
- Deux types de spécialisation
 - 1 *complète* : tous les arguments templates sont fixés
 - 2 *partielle* : certains arguments templates sont spécialisés
- Permet de personnaliser le comportement pour certains arguments template
 - Efficacité

Exemple

- `vector` est complètement spécialisé pour les vecteurs booléens
- Économie de taille

Overview

- La plupart du temps, on écrit une classe template et elle est « valide » pour tous les types
 - Parfois, on veut particulariser la classe pour certains types
- Deux types de spécialisation
 - 1 *complète* : tous les arguments templates sont fixés
 - 2 *partielle* : certains arguments templates sont spécialisés
- Permet de personnaliser le comportement pour certains arguments template
 - Efficacité

Exemple

- `vector` est complètement spécialisé pour les vecteurs booléens
- Économie de taille

Overview

- La plupart du temps, on écrit une classe template et elle est « valide » pour tous les types
 - Parfois, on veut particulariser la classe pour certains types
- Deux types de spécialisation
 - 1 *complète* : tous les arguments templates sont fixés
 - 2 *partielle* : certains arguments templates sont spécialisés
- Permet de personnaliser le comportement pour certains arguments template
 - Efficacité

Exemple

- `vector` est complètement spécialisé pour les vecteurs booléens
- Économie de taille

Overview

- La plupart du temps, on écrit une classe template et elle est « valide » pour tous les types
 - Parfois, on veut particulariser la classe pour certains types
- Deux types de spécialisation
 - 1 *complète* : tous les arguments templates sont fixés
 - 2 *partielle* : certains arguments templates sont spécialisés
- Permet de personnaliser le comportement pour certains arguments template
 - Efficacité

Exemple

- `vector` est complètement spécialisé pour les vecteurs booléens
- Économie de taille

Overview

- La plupart du temps, on écrit une classe template et elle est « valide » pour tous les types
 - Parfois, on veut particulariser la classe pour certains types
- Deux types de spécialisation
 - 1 *complète* : tous les arguments templates sont fixés
 - 2 *partielle* : certains arguments templates sont spécialisés
- Permet de personnaliser le comportement pour certains arguments template
 - Efficacité

Exemple

- `vector` est complètement spécialisé pour les vecteurs booléens
- Économie de taille

Spécialisation complète de templates

- Parfois, on veut spécialiser l'implémentation d'un template pour certains types
 - Efficacité
 - Fonctionnalité
- Possibilité de fournir de nouvelles fonctionnalités
 - ... voire de changer complètement l'interface publique
- Divergence possible grâce à l'instanciation
 - Code « généré », `vector<int>` est un type complet, et différent de `vector<bool>`

Hygiène de programmation

- Éviter les ambiguïtés et le code obscur

Spécialisation complète de templates

- Parfois, on veut spécialiser l'implémentation d'un template pour certains types
 - Efficacité
 - Fonctionnalité
- Possibilité de fournir de nouvelles fonctionnalités
 - ... voire de changer complètement l'interface publique
- Divergence possible grâce à l'instanciation
 - Code « généré », `vector<int>` est un type complet, et différent de `vector<bool>`

Hygiène de programmation

- Éviter les ambiguïtés et le code obscur

Spécialisation complète de templates

- Parfois, on veut spécialiser l'implémentation d'un template pour certains types
 - Efficacité
 - Fonctionnalité
- Possibilité de fournir de nouvelles fonctionnalités
 - ... voire de changer complètement l'interface publique
- Divergence possible grâce à l'instanciation
 - Code « généré », `vector<int>` est un type complet, et différent de `vector<bool>`

Hygiène de programmation

- Éviter les ambiguïtés et le code obscur

Spécialisation complète de templates

- Parfois, on veut spécialiser l'implémentation d'un template pour certains types
 - Efficacité
 - Fonctionnalité
- Possibilité de fournir de nouvelles fonctionnalités
 - ... voire de changer complètement l'interface publique
- Divergence possible grâce à l'instanciation
 - Code « généré », `vector<int>` est un type complet, et différent de `vector<bool>`

Hygiène de programmation

- Éviter les ambiguïtés et le code obscur

Spécialisation complète de templates

- Parfois, on veut spécialiser l'implémentation d'un template pour certains types
 - Efficacité
 - Fonctionnalité
- Possibilité de fournir de nouvelles fonctionnalités
 - ... voire de changer complètement l'interface publique
- Divergence possible grâce à l'instanciation
 - Code « généré », `vector<int>` est un type complet, et différent de `vector<bool>`

Hygiène de programmation

- Éviter les ambiguïtés et le code obscur

Spécialisation complète de templates

- Parfois, on veut spécialiser l'implémentation d'un template pour certains types
 - Efficacité
 - Fonctionnalité
- Possibilité de fournir de nouvelles fonctionnalités
 - ... voire de changer complètement l'interface publique
- Divergence possible grâce à l'instanciation
 - Code « généré », `vector<int>` est un type complet, et différent de `vector<bool>`

Hygiène de programmation

- Éviter les ambiguïtés et le code obscur

Spécialisation complète de templates

- Parfois, on veut spécialiser l'implémentation d'un template pour certains types
 - Efficacité
 - Fonctionnalité
- Possibilité de fournir de nouvelles fonctionnalités
 - ... voire de changer complètement l'interface publique
- Divergence possible grâce à l'instanciation
 - Code « généré », `vector<int>` est un type complet, et différent de `vector<bool>`

Hygiène de programmation

- Éviter les ambiguïtés et le code obscur

Spécialisation complète de templates

- Parfois, on veut spécialiser l'implémentation d'un template pour certains types
 - Efficacité
 - Fonctionnalité
- Possibilité de fournir de nouvelles fonctionnalités
 - ... voire de changer complètement l'interface publique
- Divergence possible grâce à l'instanciation
 - Code « généré », `vector<int>` est un type complet, et différent de `vector<bool>`

Hygiène de programmation

- Éviter les ambiguïtés et le code obscur

Exemple

■ Fichier full.cpp

```
1  struct R {};  
2  struct S { S() { cout << "+S" << endl; } };  
3  
4  template<class T> struct A  
5  {  
6      int i; T t;  
7      A(int i, T t) : i(i), t(t) { cout << "+A<T>" << endl; }  
8  };  
9  
10 template<> struct A<bool>  
11 {  
12     S s; bool b;  
13     A(bool b) : b(b), s(S()) { cout << "+A<bool>" << endl; }  
14 };  
15  
16 int main()  
17 {  
18     A<R> a(2,R());  
19     cout << a.i << endl;  
20     A<bool> b(true);  
21     //cout << b.i << endl; //error : no A<bool>::i  
22     cout << b.b << endl;  
23 }
```

Spécialisation partielle de templates

- La spécialisation complète de templates ne permet plus de paramétrisation « générique » du template
 - On a spécifié que `vector<bool>` se comporte particulièrement
 - On ne sait pas dire « un vecteur de pointeurs se comporte différemment »
- La spécialisation partielle de templates permet d'effectuer cela

Exemple

- `class A<T,U>` : template primaire
- `class A<T*,U>` : spécialisation partielle où le premier argument est un pointeur
- `class A<T,T>` : spécialisation partielle où les deux arguments sont identiques

Spécialisation partielle de templates

- La spécialisation complète de templates ne permet plus de paramétrisation « générique » du template
 - On a spécifié que `vector<bool>` se comporte particulièrement
 - On ne sait pas dire « un vecteur de pointeurs se comporte différemment »
- La spécialisation partielle de templates permet d'effectuer cela

Exemple

- `class A<T,U>` : template primaire
- `class A<T*,U>` : spécialisation partielle où le premier argument est un pointeur
- `class A<T,T>` : spécialisation partielle où les deux arguments sont identiques

Spécialisation partielle de templates

- La spécialisation complète de templates ne permet plus de paramétrisation « générique » du template
 - On a spécifié que `vector<bool>` se comporte particulièrement
 - On ne sait pas dire « un vecteur de pointeurs se comporte différemment »
- La spécialisation partielle de templates permet d'effectuer cela

Exemple

- `class A<T,U>` : template primaire
- `class A<T*,U>` : spécialisation partielle où le premier argument est un pointeur
- `class A<T,T>` : spécialisation partielle où les deux arguments sont identiques

Spécialisation partielle de templates

- La spécialisation complète de templates ne permet plus de paramétrisation « générique » du template
 - On a spécifié que `vector<bool>` se comporte particulièrement
 - On ne sait pas dire « un vecteur de pointeurs se comporte différemment »
- La spécialisation partielle de templates permet d'effectuer cela

Exemple

- `class A<T,U>` : template primaire
- `class A<T*,U>` : spécialisation partielle où le premier argument est un pointeur
- `class A<T,T>` : spécialisation partielle où les deux arguments sont identiques

Spécialisation partielle de templates

- La spécialisation complète de templates ne permet plus de paramétrisation « générique » du template
 - On a spécifié que `vector<bool>` se comporte particulièrement
 - On ne sait pas dire « un vecteur de pointeurs se comporte différemment »
- La spécialisation partielle de templates permet d'effectuer cela

Exemple

- `class A<T,U>` : template primaire
- `class A<T*,U>` : spécialisation partielle où le premier argument est un pointeur
- `class A<T,T>` : spécialisation partielle où les deux arguments sont identiques

Spécialisation partielle de templates

- La spécialisation complète de templates ne permet plus de paramétrisation « générique » du template
 - On a spécifié que `vector<bool>` se comporte particulièrement
 - On ne sait pas dire « un vecteur de pointeurs se comporte différemment »
- La spécialisation partielle de templates permet d'effectuer cela

Exemple

- `class A<T, U>` : template primaire
- `class A<T*, U>` : spécialisation partielle où le premier argument est un pointeur
- `class A<T, T>` : spécialisation partielle où les deux arguments sont identiques

Spécialisation partielle de templates

- La spécialisation complète de templates ne permet plus de paramétrisation « générique » du template
 - On a spécifié que `vector<bool>` se comporte particulièrement
 - On ne sait pas dire « un vecteur de pointeurs se comporte différemment »
- La spécialisation partielle de templates permet d'effectuer cela

Exemple

- `class A<T, U>` : template primaire
- `class A<T*, U>` : spécialisation partielle où le premier argument est un pointeur
- `class A<T, T>` : spécialisation partielle où les deux arguments sont identiques

Spécialisation partielle de templates

- La spécialisation complète de templates ne permet plus de paramétrisation « générique » du template
 - On a spécifié que `vector<bool>` se comporte particulièrement
 - On ne sait pas dire « un vecteur de pointeurs se comporte différemment »
- La spécialisation partielle de templates permet d'effectuer cela

Exemple

- `class A<T, U>` : template primaire
- `class A<T*, U>` : spécialisation partielle où le premier argument est un pointeur
- `class A<T, T>` : spécialisation partielle où les deux arguments sont identiques

Exemple

■ Fichier `partial-1.cpp`

```
1  template<class T> struct A
2  {
3      T t;
4      A(T t) : t(t) {}
5
6      void print() { cout << t << endl; }
7  };
8
9  template<class T> struct A<T*>
10 {
11     T* t;
12     A(T* t) : t(t) {}
13
14     void print() { cout << *t << endl; }
15 };
16
17 int main()
18 {
19     int i = 2;
20     A<int> a1(i); a1.print();
21     A<int*> a2(&i); a2.print();
22 }
```

Ordre partiel

- Quand un template de classe est instancié, il peut exister plusieurs spécialisations partielles
- Le compilateur doit décider s'il utilise le template primaire ou une spécialisation

Règle

- Correspondance exacte
- Si plus d'une spécialisation correspond, « la plus spécialisée » est utilisée
- Template primaire

Ordre partiel

- Quand un template de classe est instancié, il peut exister plusieurs spécialisations partielles
- Le compilateur doit décider s'il utilise le template primaire ou une spécialisation

Règle

■ Correspondance exacte

■ Si plus d'une spécialisation correspond, « la plus spécialisée » est utilisée

■ Template primaire

Ordre partiel

- Quand un template de classe est instancié, il peut exister plusieurs spécialisations partielles
- Le compilateur doit décider s'il utilise le template primaire ou une spécialisation

Règle

- 1 Correspondance exacte
 - Les arguments template correspondent : utilisation de la spécialisation
- 2 Si plus d'une spécialisation correspond, « la plus spécialisée » est utilisée
 - Si aucune n'est plus spécialisée : erreur (ambiguïté)
- 3 Template primaire

Ordre partiel

- Quand un template de classe est instancié, il peut exister plusieurs spécialisations partielles
- Le compilateur doit décider s'il utilise le template primaire ou une spécialisation

Règle

1 Correspondance exacte

- Les arguments template correspondent : utilisation de la spécialisation

2 Si plus d'une spécialisation correspond, « la plus spécialisée » est utilisée

- Si aucune n'est plus spécialisée : erreur (ambiguïté)

3 Template primaire

Ordre partiel

- Quand un template de classe est instancié, il peut exister plusieurs spécialisations partielles
- Le compilateur doit décider s'il utilise le template primaire ou une spécialisation

Règle

- 1 Correspondance exacte
 - Les arguments template correspondent : utilisation de la spécialisation
- 2 Si plus d'une spécialisation correspond, « la plus spécialisée » est utilisée
 - Si aucune n'est plus spécialisée : erreur (ambiguïté)
- 3 Template primaire

Ordre partiel

- Quand un template de classe est instancié, il peut exister plusieurs spécialisations partielles
- Le compilateur doit décider s'il utilise le template primaire ou une spécialisation

Règle

- 1 Correspondance exacte
 - Les arguments template correspondent : utilisation de la spécialisation
- 2 Si plus d'une spécialisation correspond, « la plus spécialisée » est utilisée
 - Si aucune n'est plus spécialisée : erreur (ambiguïté)
- 3 Template primaire

Ordre partiel

- Quand un template de classe est instancié, il peut exister plusieurs spécialisations partielles
- Le compilateur doit décider s'il utilise le template primaire ou une spécialisation

Règle

- 1 Correspondance exacte
 - Les arguments template correspondent : utilisation de la spécialisation
- 2 Si plus d'une spécialisation correspond, « la plus spécialisée » est utilisée
 - Si aucune n'est plus spécialisée : erreur (ambiguïté)
- 3 Template primaire

Ordre partiel

- Quand un template de classe est instancié, il peut exister plusieurs spécialisations partielles
- Le compilateur doit décider s'il utilise le template primaire ou une spécialisation

Règle

- 1 Correspondance exacte
 - Les arguments template correspondent : utilisation de la spécialisation
- 2 Si plus d'une spécialisation correspond, « la plus spécialisée » est utilisée
 - Si aucune n'est plus spécialisée : erreur (ambiguïté)
- 3 Template primaire

Illustration

■ Fichier partial-2.cpp

```
1  template<class T1, class T2, int I>
2  class A {}; // primary template
3
4  template<class T, int I>
5  class A<T, T*, I> {}; // partial specialization where T2 is a pointer to T1
6
7  template<class T, class T2, int I>
8  class A<T*, T2, I> {}; // partial specialization where T1 is a pointer
9
10 template<class T>
11 class A<int, T*, 5> {}; // partial specialization where T1 is int, I is 5,
12                        // and T2 is a pointer
13 template<class X, class T, int I>
14 class A<X, T*, I> {}; // partial specialization where T2 is a pointer
15
16 int main()
17 {
18     A<int, int, 1> a1; // no specializations match, uses primary template
19     A<int, int*, 1> a2; // uses partial specialization #1 (T=int, I=1)
20     A<int, char*, 5> a3; // uses partial specialization #3, (T=char)
21     A<int, char*, 1> a4; // uses partial specialization #4, (X=int, T=char, I=1)
22     //A<int*, int*, 2> a5; // error: matches #2 (T=int, T2=int*, I=2)
23                        // matches #4 (X=int*, T=int, I=2)
24                        // neither one is more specialized than the other
25 }
```

Exemple

- On voudrait spécialiser `Array<T, N>` pour que
 - 1 si utilisé avec une adresse, on imprime les éléments déréférencés
 - 2 Si utilisé « comme un tableau 2D », qu'on l'affiche sous forme matricielle
- Il faut spécialiser le template dans `array.hpp`

```
1 Array<int*,3> a_add;  
2 int n[3];  
3 for(unsigned i = 0; i < 3; i++)  
4 {  
5     n[i] = i + 1;  
6     a_add[i] = &(n[i]);  
7 }  
8 cout << a_add << endl; //objective : 1 2 3
```

```
1 Array<Array<int,2>,3> a2d = {{1,2},{3,4},{5,6}};  
2 cout << a2d << endl; //objective : 1 2  
3                       //           3 4  
4                       //           5 6
```

Solution : premier cas

■ Fichier array-spec.hpp

```
1  template<class T, int N> class Array<T*,N>
2  {
3      T ** t;
4
5      public:
6          Array() : t(new T*[N])
7              {
8                  for(int i = 0; i < N; i++)
9                      t[i] = nullptr;
10             }
11
12             ...
13
14 };
15
16 template<class T, int N>
17 std::ostream& operator<<(std::ostream& out, const Array<T*,N>& a)
18 {
19     out << "{_}";
20     for(unsigned i = 0; i < a.size() - 1; i++)
21         out << *(a[i]) << "_,_";
22     out << *(a[a.size() - 1]) << "}_";
23
24     return out;
25 }
```


Solution : second cas

■ Fichier array-spec.hpp

```
1  template<class T, int N, int M> class Array<Array<T,M>, N>
2  {
3      Array<T,M> * t;
4
5      public:
6          Array() : t(new Array<T,M>[N]) {}
7
8          ...
9  };
10
11 template<class T, int N, int M>
12 std::ostream& operator<<(std::ostream& out, const Array<Array<T,M>,N>& a)
13 {
14     for(unsigned i = 0; i < N; i++)
15     {
16         for(unsigned j = 0; j < M; j++)
17             out << a[i][j] << " ";
18         out << std::endl;
19     }
20
21     return out;
22 }
```

Debriefing

■ Ça fonctionne

Problème

- Il y a énormément de copier / coller

Solution

- Utiliser SFINAE

Debriefing

■ Ça fonctionne

Problème

- Il y a énormément de copier / coller

Solution

- Utiliser SFINAE

Debriefing

- Ça fonctionne

Problème

- Il y a énormément de copier / coller

Solution

- Utiliser SFINAE

Debriefing

- Ça fonctionne

Problème

- Il y a énormément de copier / coller

Solution

- Utiliser SFINAE
 - En dur, avec `std::enable_if` ou avec les concepts
 - Cf. Ch. 14

Debriefing

- Ça fonctionne

Problème

- Il y a énormément de copier / coller

Solution

- Utiliser SFINAE
 - En dur, avec `std::enable_if` ou avec les concepts
 - Cf. Ch. 14

Debriefing

- Ça fonctionne

Problème

- Il y a énormément de copier / coller

Solution

- Utiliser SFINAE
 - En dur, avec `std::enable_if` ou avec les concepts
 - Cf. Ch. 14

Debriefing

- Ça fonctionne

Problème

- Il y a énormément de copier / coller

Solution

- Utiliser SFINAE
 - En dur, avec `std::enable_if` ou avec les concepts
 - Cf. Ch. 14