

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

Norme ISO

Raphaëlle Chaîne
raphaelle.chaine@iris.cnrs.fr
2024-2025

384

Depuis C++11

- Ajouts compatibles avec les versions précédentes
- Faire évoluer les techniques de programmation dans le bon sens
- Améliorer encore la protection des types
- Augmenter les possibilités de travailler directement avec le matériel

385

385

Depuis C++11

- Ajouts ne concernant pas que les template et corrigeant certaines limites
- Ajouts concernant les classes
 - Délégation et héritage du constructeur
 - Initialiseurs d'attributs
`Paire paire{5,25};`
 - Suppression ou mise à défaut des fonctions standards des objets
 - Opérateur de conversion explicite
 - Liste d'initialiseurs
 - Modification de la définition des POD (Plain Old Data)

386

386

Aspect multitâche

- Classe `std::thread`
- Notion de mémoire locale propre à chaque *thread*
 - Mot clé `thread_local`
 - Chaque *thread* embarque sa propre copie d'une variable déclarée de la sorte
 - Les modifications sont internes à chaque *thread*
 - La durée de vie des variables statiques locales à un *thread* est réduite à celle du *thread*
 - Modificateur `volatile`
 - Pour indiquer au compilateur qu'une variable peut-être modifiée par autre chose que le programme qu'il est en train de compiler (et prévenir toute optimisation fatale!)

387

387

Enrichissement des possibilités de *valueness*

Voir cours 01

- Notion de référence sur un temporaire (*rvalue* reference)


```
int && a
```

 - Permet de coder des fonctions spécifiques pour traiter avec des temporaires
- Attention toutefois au fait que lorsqu'on nomme un temporaire, il apparaîtra ensuite comme une lvalue
- Transformer/caster une lvalue en xvalue
`std::move`

388

388

Spécification de la *valueness*

```
void g(int && x)
{std::cout << "g sur temporaire \n";}
void g(int & x)
{std::cout << "g(int &) \n";}
void f(int && x) // x rvalue reference
{g(x);} -----> g(int &)
(Attention : dans la fonction x est
considéré comme lvalue! )
void f(int & x)
{g(x);} -----> g (int & )
```

389

389

Spécification de la *valueness*

```
void g(int && x)
{std::cout << "g sur temporaire \n";}
void g(int & x)
{std::cout << "g \n";}
void f(int && x)
{g(move(x));} -----> g sur temporaire
void f(int & x)
{g(move(x));} -----> g sur temporaire
```

390

390

Inférence de type

- Mot clé `auto`
 - Utilisé dans une définition de variable en lieu et place du type, si déductible automatiquement
 - Le type est déduit de celui de l'objet utilisé pour l'initialisation de la variable
 - `auto` seul permet de récupérer le type mais pas la *value-ness*
- ```
auto f = fonctionRenvoyantUnType(toto);
auto g=1; //int g=1;
```
- `auto &` permet de récupérer une *lvalue reference* si c'est possible (erreur de compilation sinon)
  - `auto &&` **Référence universelle** (contexte d'un type à déduire) permet de récupérer la *value-ness* d'une référence (**& ou &&**)

391

391

## Inférence de type

- Attention : `auto` seul ne permet pas de récupérer la *value-ness* de ce qui est à droite de l'affectation!

```
int fval();
int& fref();
const int& frefcst();
auto i1 = fval(); // i1 de type int
auto a2 = fref(); // i2 de type int
auto a3 = frefcst(); // i3 de type int
```

`i1` a le même type que la valeur de retour de `fval`.

Le compilateur se base sur le type déclaré de la valeur de retour

392

392

## Inférence de type

- A vous de spécifier si vous voulez une référence (*l-value*)  
**Si c'est possible!**

```
int & fref(); const int fvalcst();
int fval(); const int & frefcst();

auto & i1 = fref(); // int &
auto & i1 = fval(); // impossible
```

393

393

## Inférence de type

- En revanche `auto &&` ne vous permettra pas toujours de bâtir une référence sur un temporaire (*rvalue reference*). Tout dépend de la *valueness* de ce qui est à droite du =

- D'où le nom de **référence universelle**

```
int & fref(); const int fvalcst();
int fval(); const int & frefcst();

auto && i3 = fvalcst(); // const int &&
auto && i4 = fval(); // int &&
auto && i5 = fref(); // Attention! int &
auto && i6 = frefcst(); // const int &
auto && i7 = std::move(fref()); // int &&
```

394

394

## Inférence de type

- Moralité du transparent précédent :  
`auto &&i` ne signifie pas toujours *rvalue reference*
- ```
auto &&i=fct_returning_an_int(); //rv reference
int j; auto && ii = j; //lv reference
```

395

395

Inférence de type

- Le mot clé `decltype` permet de typer une variable à partir du type d'une autre variable, **y compris la value-ness!**

```
int i;
decltype(i) j = 5; j est donc de type int
```

396

396

Inférence de type

- `decltype` peut être utilisé pour récupérer la value-ness d'une expression

```
int fval();
int& fref();
const int& frefcst();

decltype(fval()) i1=fval(); //i1 type int
decltype(fref()) i2=fref(); //type int&
decltype(frefcst()) i3=frefcst();
//type const int&

i1 a le même type + valueness que la valeur de
retour de f1 : le compilateur regarde le type de la
valeur de retour
```

397

397

Inférence de type

- Utile pour éviter les répétitions
- ```
LaClasse *p=new LaClasse;
auto c2=new LaClasse;
```
- Utile pour éviter d'avoir à construire un nom compliqué lié à l'usage des template
- ```
vector<vector<int>> > v;
// C++98 (sans auto)
for (vector<vector<int>>>::iterator
    i=v.begin(); i!=v.end();++i)
// C++11
for (auto i=v.begin(); i!=v.end();++i)
```

398

398

Inférence de type

- Dans le contexte des templates
- ```
template <typename T>
```
- si `T` est un type déduit par le compilateur (lors d'une instantiation automatique), `T &&` ne signifie pas toujours référence sur un temporaire (*rvalue reference*)
  - **Référence universelle.**
  - Une référence universelle sera une *lvalue reference* ou une *rvalue reference* en fonction de son initialisation
    - Si l'expression provoquant l'instanciation de la référence universelle est une *lvalue*, la référence universelle devient une *lvalue reference*
    - Si l'expression l'instanciation la référence universelle est un *rvalue*, la référence universelle devient une *rvalue reference*

399

399

### Spécification de la *valueness*

- Notion de référence sur un temporaire (*rvalue reference*)
- ```
int && a
```
- Transformer/caster une *lvalue* en *xvalue*
- ```
std::move
```
- Dans les templates, spécification ou récupération de la *valueness* d'un argument
- ```
std::forward<T> (transfert de valueness)
std::ref (pour que l'instanciation implicite d'un
template se fasse avec une référence sur le type
considéré)
```

400

400

Universal reference dans les template

```
template<typename T>
void f(T&& uref);
```

Attention :

La substitution du type au moment de l'instanciation du template définira si c'est une *lvalue reference* ou une *rvalue reference*.

401

401

Spécification de la *valueness* dans les template

```
template<class T>
void f(T&& x) //universal reference
{ //x est considere comme lvalue ici
  g(std::forward<T>(x));
  //forward en lvalue ou rvalue suivant x
}
Permet de retrouver la valueness d'origine de x
void g(int && x)
{ std::cout << "g sur temporaire \n"; }
void g(int & x)
{ std::cout << "g \n"; }
```

402

402

Spécification de la *valueness* dans les template

```
int i;
f(i); // g
f(4); // g sur temporaire
```

403

403

Spécification de la *valueness* Wrapper reference

```
template<typename T>
void f(T x)
{ x++; }

int i=3;
f(i); //f<int>(i)
std::cout << i << std::cout; //affiche 3
f(std::ref(i)); //f<int &>(i)
std::cout << i << std::cout; //affiche 4
```

404

404

Wrapper reference

- A toute instantiation implicite de fonction template, il est désormais possible de spécifier qu'un paramètre est de type donnée-résultat

```
void fonc(int &r) { r++; }
template<class F, class P>
void g(F f, P t) { f(t); } // f foncteur
puis
int i=0;
g(fonc, i);
//instanciation de g<void(int &r),int>
//du coup i n'est pas modifie ☹
std::cout << i << std::endl; // affichage : 0
```

405

405

Wrapper reference

- A toute instantiation implicite de fonction template, il est désormais possible de spécifier qu'un paramètre est de type donnée-résultat

```
void fonc(int &r) { r++; }
template<class F, class P>
void g(F f, P t) { f(t); }
puis
int i=0;
g(fonc, std::ref(i));
// instanciation de
// g<void(int &r), reference_wrapper<int>>
// cette fois i est modifie ☺
std::cout << i << std::endl; // Affichage : 1
```

406

406

Itérer sur les éléments d'un conteneur

- Référence successive à chacun des éléments d'un tableau (foreach)

```
int mon_tableau[5] = {1, 2, 3, 4, 5};
for (int &x: mon_tableau) {
  x += 1;
}
```

- x référence successivement chacun des éléments du tableau
 - valable aussi pour les listes d'initialiseurs, ainsi que les conteneurs de la STL définissant les fonctions membres begin et end
- ```
for (int &x: v) { //vector<int> v={1, 2, 3};
 ...
}
```

407

407

## Le pointeur nul

- Proposition du mot-clé `nullptr` en lieu et place de l'entier 0
    - constante du langage assignable à tous les types de pointeurs
- ```
T* ptr = nullptr;
```
- la macro C `#define NULL ((void*)0)` était inutilisable (pas de conversion implicite d'un `void *` dans un pointeur)

408

408

Constructeurs plus souples...

- Délégation et héritage

```
struct LaClasse {
    LaClasse( int a1, int a2): _a1(a1), _a2(a2){}
    LaClasse( int a1): _a1(a1) {}
    LaClasse() : LaClasse(55,2) {}

    int _a1;
    int _a2=5; // initialisation C++11
};

struct AutreClasse: public LaClasse {
    using LaClasse::LaClasse;
    ...
};
```

409

409

Fonctions par défaut...

- Implantation par défaut par le compilateur

```
struct LaClasse {
    LaClasse()=default; //C++11
    virtual ~LaClasse()=default; //C++11
    ...
};
```

- Suppression de l'implantation par défaut par le compilateur

```
struct SansCopie{
    SansCopie & operator =
        (const SansCopie & ) = delete;
    SansCopie ( const SansCopie & ) = delete;
};
```

410

410

Template variadiques

- Plutôt que de templaté une fonction ou une classe par une liste de types
- On peut désormais utiliser les template variadiques

```
template<typename T, typename ... ST>
T Ma fonction(ST ... Args) {}
```

```
template<typename ... ST>
LaClasse{};
```

Puis plusieurs spécialisations de `LaClasse`...

411

411

Template variadiques

- Séquence de paramètres template

```
template<typename ... ST>
```

ST regroupe une séquence finie de types (possiblement vide)

- Séquence d'arguments de fonction

```
ST ... Args
```

Séquence de paramètres de types correspondant à ST

- Extension d'une séquence d'arguments

```
Args ...
```

- Peut-être passé en paramètre d'une fonction template variadique
- Utile pour l'aspect récursif de ces fonctions

412

412

- Printf vue en fonction template variadique

```
template<typename ... TArgs>
void printf;
```

413

413

– Printf vue en fonction template variadique

- printf avec un paramètre (mais aucun issu du template)

```
template<>
void printf(const char *s)
{
    while (*s) {
        if (*s == '%' && *++s != '%') {
            throw std::runtime_error( "nombre
insuffisant de parametres");
        }
        std::cout << *s++;
    }
}
```

414

414

– Printf vue en fonction template variadique

- Spécialisation printf avec plus de 2 paramètres

```
template<typename T, typename ... TArgs>
void printf(const char* s, const T& value,
const TArgs& ... args) {
    while (*s) {
        if (*s == '%' && *++s != '%') {
            std::cout << value;
            printf(++s, args...); //APPEL RECURSIF
            return;
        }
        std::cout << *s++;
    }
    throw std::runtime_error("Trop d'arguments dans
printf");
}
```

415

415

– La classe Tuple offerte par C++11 correspond à un autre exemple de template variadique

- Généralisation du concept de pair
- Séquence de dimension fixe d'objets de types différents

```
template<typename ... Types> struct tuple;
```

- Exemple

```
long li = 55 ;
tuple< int,long &,double > up(5,li,10.5);
li = get<0>(up); // Assigne à li la valeur 5
get<1>(up) = 10;
get<2>(up) = 55.5 ; // Modifie la 3ème
valeur du tuple
```

416

416

– Il est possible

- de créer un tuple sans définir son contenu si les éléments du tuple possèdent un constructeur par défaut
- d'assigner un tuple à un autre tuple si le type de chaque élément de l'opérande de droite est convertible dans le type correspondant dans l'opérande de gauche

– Expressions statiques disponibles sur les tuples (évaluées à la compilation)

```
tuple_size<TU>::value retourne le nombre
d'éléments des tuple de type TU,
tuple_element<I, TU>::type retourne le type
de l'objet placé en position I des tuple de
type TU.
```

417

417

Template variadiques

- Utile pour fabriquer une classe de Maker (fabrique) d'objets de type T
 - Adaptation au nombre de paramètres requis par le constructeur du type T

```
template<typename T>
class Maker{
public:
    template<typename... Args>
    std::shared_ptr<T> operator() (Args&&...args)
    {return std::shared_ptr<T>(new
T(std::forward<Args>(args ...)));};
};
```

418

418

```
template<typename T>
class Maker{
public:
    template<typename... Args>
    std::shared_ptr<T> operator() (Args&&...args)
    {return std::shared_ptr<T>(new
T(std::forward<Args>(args ...)));};
};
```

- `std::forward` sert à transférer les arguments au constructeur en préservant leur `value_ness`
- `auto p1=Maker<LaClasse>() (7, 5.5);`
`auto p2=Maker<LaClasse>() (9);`

419

419

Assertions statiques

– Pour effectuer un test à la compilation

- Exemple : Pour vérifier que l'on travaille bien sur un système 64 bits.

```
static_assert(sizeof(long) > sizeof(int),
    "La bibliothèque doit être compilée
    sous un système 64-BIT");
```

420

420

Nouvelle syntaxe de déclaration de fonctions

- `int f() { return 1; }`
- `auto f2() -> int { return 1; } // C++11`
- ...

421

421

Lambda expressions

– Pour définir des fonctions localement, sans leur donner un nom, avec accès au contexte local

– Syntaxe

```
[capture](parametre)->type_de_retour
{traitement}
```

– Exemple :

```
char s[]="Bonjour Mlif23!";
int majuscule = 0;
for_each(s, s+std::strlen(s),
    [&majuscule] (char c) {
        if (std::isupper(c))
            majuscule++;
    });
```

422

422

Lambda expressions

– Pour définir des fonctions localement, sans leur donner un nom, avec accès au contexte local

– Syntaxe

```
[capture](parameters)->return-type {body}
```

– Exemples :

```
auto f=[]{ return 3;};
auto f=[]() { return 3;};
auto f=[]()->int{ return 3;};
int f(){return 3;}
```

423

423

Lambda expressions

– Définition d'une fonction à l'intérieur du code d'une autre fonction!

– Les `[]` marquent le début de la lambda expression

– `&majuscule` signifie que la lambda embarque une référence sur la variable `majuscule`

– `[majuscule]` signifierait un passage par copie

– `[]` pas d'accès aux variables externes

– `&` accès à toutes les variables par référence.

– `=` accès à toutes les variables par copie

424

424

Syntaxe uniforme d'initialisation

- ```
class C {
 int a; int b;
public:
 C(int i, int j);
};
C c {0,0}; //C++11
C c(0,0);
int* a = new int[3]{5,9,0}; //C++11
...
```

425

425

## Meilleure clarté du polymorphisme dynamique

- Redéfinition plus explicite des fonctions membres
- ```
struct Base {
    virtual void fonc(int);
};
struct Derived : Base {
    virtual void fonc(int) override;
    // Le compilateur ramera si ce n'est
    // pas une vraie redéfinition ☺
};
```

426

426

Meilleure clarté du polymorphisme dynamique

- Impossibilité de dériver une classe (et donc possibilité d'éviter, sans complexes ☺, le recours à une table des fonctions virtuelles)
- ```
struct Base final { };
//struct Derivee : Base { };
// refuse par le compilateur
```

427

427

## Meilleure clarté du polymorphisme dynamique

- Impossibilité de redéfinir une fonction membre

```
struct Base {
 virtual void f() final;
};

struct Derivee : Base {
 //void f();
 // refuse par le compilateur
};
```

428

428

## Alias templates

- Avant C++11 : possibilité de faire des typedef sur des types, mais pas sur des templates ...

```
template <typename Prems, typename Deuz, int Troiz>
class UnType;
```

```
template <typename Deuz>
typedef UnType<int, Deuz, 5> TypedefTemplate;
// Illegal avec C++03 comme avec C++11
```

de même :

```
template <typename T>
class myAllocator
{
 template <typename U> typedef myAllocator<U> OtherType;
}; // Illegal avec C++03 (comme avec C++11)
```

GRRRRR!!!!!!

429

429

## Alias templates

- Avant C++11 : Utilisation d'une astuce de métaprogrammation fréquemment utilisée, le rebind :

```
template <typename T>
class myAllocator
{
 template <typename U>
 struct rebind {typedef myAllocator<U> OtherType;};
};
//Utilisation possible de
MyAllocator<int>::rebind<Cell>::OtherType

Permet à myAllocator d'allouer des T mais aussi d'autres types ☺
```

430

430

## Alias templates

- Avant C++11 : possibilité de faire des typedef sur des types, mais pas sur des templates

```
template <typename Prems, typename Deuz, int Troiz>
class UnType;
```

```
template <typename Deuz>
typedef UnType<int, Deuz, 5> TypedefTemplate;
// Illegal avec C++03
```

maintenant :

```
template <typename Prems, typename Deuz, int Troiz>
class UnType;
```

```
template <typename Deuz>
using TypedefTemplate = UnType<int, Deuz, 5>;

TypedefTemplate<int> correspond à UnType<int,int,5>
```

431

431



## C++14

- Littéraux binaires
  - `int nb = 0b11;`  
`std::cout << nb << std::endl; // sortie de l'application : 3`
- Séparateur de chiffres
  - `int un_milliard = 1'000'000'000;`
- L'attribut `[[deprecated]]`
  - `[[deprecated]] int f();`
  - `[[deprecated("La fonction g() est dépréciée. Utilisez plutôt la fonction h()")]] void g( int& x );`  
`void h( int& x );`
- Généricité et polymorphisme des fonctions lambda
  - `auto lambda = [](auto x, auto y) {return x + y;};` <sup>432</sup>

432