Chapitre VII

Fonctionnelles

Sommaire

1 Introduction

Cette leçon est destinée à décrire les outils fonctionnels et leurs applications disponibles en C++, à savoir :

- un rappel sur le mécanisme d'appel des fonctions,
- les pointeurs de fonction,
- les fonctors, les prédicats, les opérateurs et les fonctionoïds
- les lambda-expressions.
- les objets fonctionnels,

2 Appel des fonctions

2.1 Mécanisme d'appel des fonctions

Tout appel de fonction induit le mécanisme suivant :

1. allocation de l'environnement local (= réservation de la mémoire nécessaire).
2. construction d'une "stack frame" qui contient :
   - l'adresse de retour de la fonction (point du code où il faut retourner après l'appel de la fonction),
   - l'ensemble des arguments de la fonction initialisés avec les valeurs passées en paramètre,
   - la mémoire nécessaire pour l'ensemble des variables locales,
   - une sauvegarde de l'ensemble des registres qui seront modifiés par la fonction.
3. empilement du "stack frame" sur la pile d'exécution (call stack).
4. saut du pointeur d'exécution au début du code de la fonction, et exécution de la fonction,
5. au retour de la fonction,
   - rétablissement des registres sauvegardé dans le stack frame courant,
   - passage de la valeur de retour,
   - rétablissement du pointeur d'exécution à l'adresse de retour,
   - dépilement de la stack frame (libère la mémoire associée).
6. poursuite de l'exécution

Remarque : Pour toute fonction, la taille mémoire nécessaire à son exécution est **toujours** calculée à la compilation. Au moment de l'exécution, la mémoire à allouer dans le stack est déjà connue.

Pour toutes les méthodes fonctionnelles que nous allons voir :

- soit il n'y a pas d'appel (si le compilateur décide de remplacer localement l'appel par le code de la fonction),
- pour un appel de fonction ou de méthode classique, il y a un appel standard.
- pour un appel par un pointeur de fonction, il y a un appel après une indirection.

- pour un appel à une méthode virtuelle sur un objet , il y a appel après deux indirections (VPTR + pointeur de fonctions)
- pour une lambda-expression, l'appel crée un objet temporaire sur le stack associé au contexte d'exécution de l'expression, et appelle l'objet créé.
  Noter qu'une lambda expression peut être convertie en pointeur si elle ne capture pas le contexte d'exécution courant.

2.2 Conventions d'appel

Par convention d'appel, on entend la façon de lancer l'appel d'une fonction basée en particulier sur :

- les paramètres sont en général placés dans le stack, mais des registres peuvent être utilisés pour transférer les premiers paramètres. **conséquence :** placer en premier les paramètres les plus importants en convention `fastcall`.
- l'ordre dans lequel les arguments sont passés (exemple : droite à gauche),
- à qui revient la maintenance de la pile (nettoyage) : à l'appelant ou à la fonction elle-même,
- la décoration des noms (= nom binaire de la fonction)
  exemple : _ préfixés aux noms pour les fonctions C,

Il est important de savoir que ces conventions existent pour l'utilisation d'une fonction compilée : l'appel d'une fonction compilée suivant une convention particulière et appelée suivant une autre provoque des erreurs (en général au moment de l'édition de lien).

Les principales conventions d'appel sont les suivantes :

32bit	`cdecl`	défaut pour les applications C et les bibliothèques statiques
	`stdcall`	défaut pour les appels système
	`fastcall`	utilisation des registres afin d'accélérer le passage des paramètres
	`vectorcall`	(MS) idem `fastcall` en utilisant des registres vectoriels (= plus de paramètres)
	`thiscall`	défaut pour appel des méthodes (MS)
64bit	MS/GNU	une seule convention d'appel spécifique au système.

Exemple : de définition de la convention d'appel

```
// dans les prototype
int __cdecl system(const char *);
void __stdcall GetSystemTime(LPSYSTEMTIME lpSystemTime);

// utilisation de la convention d'appel fastcall
struct CMyClass { void __fastcall mymethod(); };
```

2.3 Décoration de noms

La décoration de nom (name mangling ou name decoration) est la manière en C (`cdecl` seulement) ou C++ de stocker et d'identifier une fonction dans un code objet.

Exemple : de décoration de noms en fonction de la convention d'appel

```
void __cdecl foo(void);             // _foo
void __cdecl foo(int a);            // _foo
void __cdecl foo(int a, int b);     // _foo
void __stdcall foo(void);           // _foo@0
void __stdcall foo(int a);          // _foo@4
void __stdcall foo(int a, int b);   // _foo@8
void __fastcall foo(void);          // @foo@0
void __fastcall foo(int a);         // @foo@4
void __fastcall foo(int a, int b);  // @foo@8
```

Ce sont les noms décorés que l'on observe en général lorsque l'on liste les fonctions/méthodes stockées dans une bibliothèque.

Noter que l'on retrouve que, en C++, une fonction est identifiée par son nom et le type de ses arguments.

3 Pointeurs de fonction

Rappel : lors de l'exécution d'un code sur un processeur, le pointeur d'instruction (instruction pointer ou program pointer) est un registre spécial qui contient l'adresse de la prochaine instruction à exécuter.

Pour l'appel d'une fonction standard, l'appel d'une fonction consiste (avec la création du stack frame) à affecter l'instruction pointer au point d'appel de la fonction.

L'idée des pointeurs de fonction est la suivante :

- créer un nouveau type de variable (un pointeur de fonction) destiné à contenir l'adresse du point d'appel de la fonction,
- par rapport à un appel classique, utiliser un pointeur vers une fonction (avec les arguments de cette fonction) provoque alors :
  - ◇ un déréférencement (= lire l'adresse stockée dans le pointeur, et se rendre à cette adresse),
  - ◇ l'appel de fonction pointée.

Note : le fonctionnement décrit ci-dessus n'entre pas dans les détails, et est juste destinée à faire comprendre la différence entre l'appel direct d'une fonction, et l'appel d'une fonction à travers un pointeur de fonction.

Définition : un pointeur de fonction est un pointeur qui pointe sur l'adresse du point d'entrée d'un code.

Afin de définit une variable pouvant contenir un pointeur de fonction, il faut définir un type associé à un pointeur sur une fonction.

Quel serait l'idée d'un tel pointeur de fonction ?

- paramétriser une fonction avec une autre fonction,
- intégrer un pointeur associé à une fonction particulière à un objet

afin de pouvoir choisir/modifier la fonction à appeler.

Or, si l'on peut remplacer une fonction par une autre, il faut que les fonctions interchangeables aient

les mêmes caractéristiques.

En conséquence, le **type d'un pointeur de fonction** est déterminé par :

- le type retourné par la fonction,
- le type qualifié de l'ensemble des paramètres,
- la convention d'appel utilisée (pour la suite de cette présentation et pour simplifier, on négligera ce dernier paramètre).

Cet ensemble d'informations est généralement donnée dans le prototype d'une fonction.

3.1 Déclaration

On donne alors les définitions suivantes :

- Le **type d'un pointeur de fonction** a donc la forme du prototype d'une fonction pour lequel `(*)` remplace le nom de la fonction.
  Exemple : pour une fonction `int fun(int a, int b)`, le type du pointeur de fonction pouvant pointer sur cette fonction est `int (*)(int,int)`.
- Une variable de type pointeur de fonction est définie en plaçant derrière l'étoile entre parenthèse le nom à utiliser pour ce pointeur.
  Exemple : pour pointeur de fonction nommé `ptrfun` et de type `int (*)(int,int)` est déclaré avec : `int (*ptrfun)(int,int);`
- Pour affecter une fonction `fun` à un pointeur de fonction `ptrfun`, il suffit d'affecter `ptrfun` à `fun` ou à `&fun` (les deux écritures sont autorisées).

Notes :

- il est également possible d'affecter un pointeur de fonction à `nullptr`/`NULL`.
- comme tout pointeur, un pointeur de fonction doit toujours être initialisé à sa création (à `nullptr` ou pointer vers une fonction).

3.2 Définition et utilisation

Utilisation :

- Affecter un pointeur de fonction `ptrfun` à une fonction `fun` de type compatible.
  Ceci se fait en écrivant `ptrfun=fun;` ou `ptrfun=&fun;`. L'écriture sans `&` est également acceptée (car pas d'ambiguïté).
- Afin de lancer l'exécution d'une fonction pointée par un pointeur de fontion `ptr`, il faut utiliser `(*ptr)(...)` où `(...)` représente les arguments attendus par la fonction.
  L'écriture sans `*` est également acceptée (car pas d'ambiguïté).

Exemple :

```
// fonctions
int add(int a, int b) { return a+b; }
int mul(int a, int b) { return a*b; }
// déclaration d'un pointeur de fonction
int (*ptFct)(int,int) = nullptr;
```

```
// dans un code (écriture 1)
pfFct = &add;
int z1 = (*pfFct)(4,3);
// dans un code (écriture 2)
pfFct = mul;
int z2 = pfFct(4,3);
```

Remarques :

- les objets pointés par un pointeur de fonctions pointent vers des zones mémoires déjà allouées (fonctions compilées dans le code, issues de bibliothèques externes, ...).
  en conséquence, pas de `new`/`delete` pour allouer/desallouer des fonctions.
  mais il est possible d'allouer de la mémoire, d'y charger un code compilé, et d'exécuter le code chargé à cette adresse.
- il est possible de créer des types particuliers associés à un pointeur de fonction :

  ```
  typedef int(*optype)(int x, int y);     // C/C++
  using OpType = int(*)(int x, int y);    // depuis C++11
  OpType ptr = &add;
  ```

  permet de définir plus facilement des variables ou des structures utilisant ce type.
- un type pointeur de fonction est un type comme un autre lorsqu'il s'agit d'un patron de classe.
  pas de problème à utiliser un pointeur de fonction comme type générique dans un template.

Exemple :

Supposons que nous voulions définir une fonction permettant de choisir l'opération à effectuer entre deux entiers. On pourrait écrire :

```
int SwitchOperator(int a, int b, char opCode) {
    switch(opCode){
    case '+' : return Plus (a, b);
    case '-' : return Minus (a, b);
    ... } }
// exemple d'appel
int v = SwitchOperator(4,5,'+');
```

Avec un pointeur de fonction, il est possible de définir :

```
int Plus (int a, int b) { return a+b; }
int Minus (int a, int b) { return a-b; }
using OperatorPt = int (*)(int,int);
int ApplyOperator(int a, int b, OperatorPt op) {
  return (*op)(a,b);  // ou op(a,b)
}
// exemple d'appel
int v = ApplyOperator(4,5,Plus);
```

L'intérêt de l'approche est que l'opérateur à utiliser est stocké dans une variable, donc peut être intégré à une structure de données et/ou être modifié suivant les besoins de l'application.

On a donc un moyen de transporter une fonctionnelle dans une variable.

Les pointeurs de fonction sont aussi des pointeurs, et en conséquence, ils peuvent être comparés.

- == pour comparer si un pointeur pointe vers une fonction particulière ou comparer si deux pointeurs pointent vers la même fonction.
- != idem avec la différence.

L'utilisation est similaire à celle des pointeurs standards.

Exemple :

```
int Plus(int a, int b) { return a+b; }
using OperatorPt = int (*)(int,int);
OperatorPt op = &Plus;
// exemple d'utilisation
if (op == nullptr) { /* op pointe sur aucune fonction */ }
else if (op == &Plus) { /* op pointe sur la fonction Plus */ }
```

Rappel : les pointeurs de fonction doivent toujours être initialisés lors de leurs créations (comme les pointeurs standards).

Exemple : utilisation d'un pointeur dans une structure

```
using namespace std;
using ptFun = int(*)(int, int);
int add(int x, int y) { return x + y; }
int mul(int x, int y) { return x * y; }

struct MyClass {
  ptFun fun;
  // avec définition du type associé au pointeur de fonction
  void setFun(ptFun f) { fun = f; }
  ptFun getFun(void) { return fun; }
  int call(int x, int y) { return fun(x, y); }
  int _call(int x, int y) { return this->fun(x, y); }
  // avec définition directe du pointeur de fonction
  void _setFun(int (*f)(int,int)) { fun = f; }
  int(*_getFun(void))(int, int) { return fun; }
};
// exemple d'utilisation
MyClass   Inst, *pInst = &Inst;
Inst.setFun(add);
int  v = Inst.call(4, 5);
ptFun       f = Inst.getFun();
```

Important :

- la définition du type de pointeur **doit** inclure la convention d'appel de la fonction,
- un type de pointeur définit pour une convention d'appel particulière est incompatible avec toute fonction n'ayant pas la même convention d'appel.
- rappel : si la convention d'appel n'est pas spécifiée, la convention d'appel par défaut est utilisée.

Exemple :

```
// avec cette déclaration
int funcallB(__cdecl    int(*fun)(int), int v);
```

```
// le pointeur de fonction en paramètre ne peut être
// que du type suivant:
__cdecl int funB(int x);
// sauf si __cdecl est la convention d'appel par défaut
```

3.3 callback

Un **callback** est une fonction passée en argument d'une autre fonction, dont elle peut faire usage comme une autre fonction, et qu'elle ne connait pas à l'avance.

Exemples de callback :

- dans une fonction de tri avec en paramètre un tableau d'éléments,
  le callback est un paramètre supplémentaire qui permet de comparer deux éléments.
  Exemple : `void SortIntArray(int ArraySize, int *Array, bool (*IntComp)(int,int))`
- dans une fonction de minimisation d'une fonction convexe,
  le callback est la fonction convexe à minimiser sur l'intervalle `[vMin,vMax]`.
  Exemple : `float ConvexMinimize(float vMin, float vMax, float (*fun)(float))`

Exemple : calcul d'une intégrale par la méthode des trapèzes

```
float TrapezoidalIntegral(float xMin, float xMax,
                          int n, float (*fun)(float)) {
    float eps = (xMax - xMin)/float(n), v = vmin;
    float area = 0.5f * (*fun)(vmin);
    for(int i=0;i<n;i++,v+=eps) area += (*fun)(v);
    area += 0.5f * (*fun)(vmax);
    return area * eps;
}
```

```
// exemple 1
float gauss(float x) {
    constexpr float norm = 1.f / sqrtf(2.f * float(M_PI));
    return norm * expf(- 0.5f*x*x );
}
float val1 = TrapezoidalIntegral(-6.f,+6.f,100,gauss);
```

```
// exemple 2
// logf (fonction standard de math.h)
float val2 = TrapezoidalIntegral(1.f,10.f,100,logf);
```

La bibliothèque standard (C et C++) fait aussi usage des callbacks.

La fonction `qsort` permet d'effectuer un tri d'une liste générique. Son prototype est le suivant :

```
void qsort( void *ptr, std::size_t count, std::size_t size,
        int(*comp)(const void *, const void *) )
```

où le callback retourne un nombre négatif si le premier argument est inférieur au second, positif si le premier est supérieur au second, et zéro si les arguments sont égaux.

Exemple :

```
struct V { int x,y; };
// tri suivant x, puis suivant y
int compV(void *p1, void *p2) {
  V   &a=*(V*)p1, &b=*(V*)p2;
  if (a.x == b.x) return a.y - b.y; else return a.x - b.x;
}
```

```
// utilisation de qsort
V   vArray[6] = { {4,1}, {1,6}, {8,3}, {6,5}, {4,9}, {1,2} };
qsort(vArray,6,sizeof(V),compV);
// ici vArray = { {1,2}, {1,6}, {4,1}, {4,9}, {6,5}, {8,3} };
```

3.4 Pointeur sur la méthode d'une classe

Le même principe peut également être utilisé dans une structure/classe pour fixer la fonction à utiliser pour réaliser certaines taches.

Rappel : une méthode statique est une méthode dont l'exécution ne dépend que des champs statiques et de ses paramètres.

Attention, il y a deux cas différents. Si la méthode à laquelle on veut faire référence est :

- une méthode statique, alors le pointeur de fonction est similaire à un pointeur de fonction standard,
- une méthode non statique, alors le pointeur de fonction inclus le nom de la classe dans la définition de son type (car alors il s'applique sur les objets instances de cette classe).
  Dans ce cas, la définition du type du pointeur de fonction **inclut le nom de la classe** (pointeur vers une fonction membre).
  Exemple : `float (Object::*)(int)` est le type d'un pointeur de fonction sur un membre de la classe `Object` prenant en paramètre un entier et renvoyant un flottant.

Remarque : ne jamais essayer de caster un pointeur de fonction membre non statique en un pointeur de fonction classique (le résultat est indéfini est génèrera très probablement une erreur d'exécution).

Note : pour une méthode `const` ajouter le mot-clé `const` à la fin de la définition du prototype.

Exemple 1 : pointeur de fonction sur une méthode statique

```
class ObjectStatic {
private:
    // pointeur de fonction standard
  using   DisplayFun = void(*)(int);
  int       x;
  DisplayFun  display;
public:
   void Set(DisplayFun f) { display = f; }
   // méthodes statiques (ne dépend pas de l'objet courant)
   static void TextDisplay(int v) { printf("sTXT: %d\n", v); }
   static void WindowDisplay(int v) { printf("sWIN: %d\n", v); };
   // appel du pointeur avec passage en paramètre de la valeur
   void Display() { display(x); };
   // constructeur
   ObjectStatic(int a) :
      x(a), display(&ObjectStatic::TextDisplay) {}
};
```

```
// exemple de code
ObjectStatic obj(2);
obj.Display();     // appel TextDisplay
obj.Set(&ObjectStatic::WindowDisplay);
obj.Display();     // appel WindowDisplay
// appel direct de la méthode statique
ObjectStatic::TextDisplay(5);
```

Exemple 2 : pointeur de fonction sur une méthode non statique

```
class Object {
private:
   // la définition du type inclut le nom de la classe
   using  DisplayFun = void (Object ::*)();
   int    x;
   DisplayFun        display;
public:
   void Set(DisplayFun f) { display = f; }
   // noter que ces deux méthodes s'appliquent sur un objet
   void TextDisplay() { printf("TXT: %d\n", x); }
   void WindowDisplay() { printf("WIN: %d\n", x); }
   // appel du pointeur sur l'instance courante
   void Display() { (this->*display)(); };
   // constructeur
   Object(int a) :
      x(a), display(&Object::TextDisplay) {}
};
```

```
// exemple de code
Object obj(4);
obj.Display();     // appel TextDiplay
obj.Set(&Object::WindowDisplay);
obj.Display();     // appel WindowDiplay
```

Avec un pointeur de fonction sur une méthode membre, il est également possible d'appeler la méthode

pointée sur un objet particulier.

Exemple :

```
using  ObjectFun = int (Object ::*)(int);
class Object {
private: int x;
public:  Object(int a) : x(a) {}
         int add(int y) { return x+y; }  };
```

```
// exemple de code
Object     obj(4);
ObjectFun  fun = & Object::add;
// comment executer fun sur obj?
```

Deux manières de procéder :

- **méthode 1 :** (obsolète sur certains compilateurs)
  `((object).*(ptrToMember))(MembersArguments)`
  Exemple : `int v = ((obj).*(fun))(3);`
- **méthode 2 :** (C_{17}^{++})
  utiliser `std::invoke(object,ptrToMember,MembersArguments)`
  Exemple : `int v = std::invoke(obj,fun,3);`

Permet d'avoir un pointeur sur une méthode capable de changer la méthode à exécuter (avec le même prototype).
Note : il est préférable d'avoir un pointeur de fonction interne à la classe et que la façon de changer le pointeur de fonction soit effectuée par une méthode occultant le détail pour l'utilisateur.

3.5 Tableau de pointeurs de fonction

Si l'on doit choisir parmi une fonction parmi un ensemble de pointeur de fonctions, il est plus facile d'indexer ces pointeurs dans un tableau.

Problème : comment déclarer un tableau de pointeurs de fonction ?

- définir un type associé au pointeur de fonction pour déclarer un tableau de ce type.
- utiliser `std::invoke` (où l'équivalent obsolète si non disponible) pour les appels sur un objet avec l'élément du tableau.

Exemple :

```
using ptFun = int (*)(int,int);
int add(int x, int y) { return x + y; }
int mul(int x, int y) { return x * y; }
ptFun tab[2] = { add, mul };
int v = tab[1](4, 5); // = mul(4,5)
```

```
class Object {
private: int x;
public:  Object(int a) : x(a) {}
         int add(int y) { return x+y; }
         int mul(int y) { return x*y; }
         using Fun = int (Object::*)(int);   };
Object::Fun tab[2] = { Object::add, Object::mul };
Object  obj(4);
int v = std::invoke(obj,tab[1],5);
```

3.6 Implémentation équivalente avec héritage

Une méthode virtuelle peut être utilisé pour implémenter un comportement similaire à un pointeur de fonction.

Exemple :

```
class A {
    protected: int x;
    public: A(int v) : x(v) {}
      virtual void Display() = null;
};
class B1 : public A { public: void Display() { /* impl.1 */ } };
class B2 : public A { public: void Display() { /* impl.2 */ } };
```

conséquence : toutes les instances appartenant à la même classe ont la même implémentation pour toutes leurs méthodes.

L'utilisation de méthode virtuelle oblige à composer les classes à partir d'autres classes afin d'obtenir l'implémentation souhaitée (spécialisation) pour chaque méthode.

Mais cette approche s'avère considérablement moins puissante que l'utilisation de membres pointeurs de fonction.

3.7 Comparaison des approches

l'utilisation de pointeur de fonction permet :

- pour chaque pointeur de fonction statique, d'affecter une méthode particulière **par classe**,
- pour chaque pointeur de fonction non statique, d'affecter une méthode particulière **par objet**.

Noter qu'un pointeur de fonction **peut être réaffecté en cours d'exécution**.

En conséquence,

- s'il existe une organisation des méthodes, la structuration en classe permet de la faire apparaitre naturelle,
- si cette structuration est faible, les pointeurs de fonction permettent de maximiser la flexibilité tout en minimisant le nombre de classes.

L'approche utilisant des pointeurs de fonction est par conséquent beaucoup plus flexible et puissante.

Elle est également plus rapide si l'instance de l'objet est manipulée sous une forme polymorphe (en raison de l'accès à la `VTABLE` à travers le `VPTR`, voir le cours sur le polymorphisme).

4 Référence d'une fonction

De la même façon qu'il est possible d'utiliser une référence à la place d'un pointeur pour un objet, il est possible d'utiliser des références de fonction.

La syntaxe est exactement la même que pour un pointeur de fonction en remplaçant l'étoile * par l'esperluette &.

Avantages :

- pas de déréférencement (gain de temps),
- pour une variable ou un champ, comme toute référence, nécessite d'être initialisée à sa création,
- en paramètre d'une fonction, fait toujours référence à une fonction existante (*i.e.* ne peut pas être nul, comme un pointeur),
- utilisable dans un patron de type stratégie (stratégie à utiliser), ou comme paramètre de templates

Inconvénient : impossibilité de faire des tableaux de références de fonction.

Sur le fond, l'utilisation de `std::function` permet une approche beaucoup plus flexible (voir à la fin de ce chapitre).

Exemples :

```
int fun1(int x) { return x+1; }
int fun2(int x) { return 2*x; }
int fun3(int x) { return x*x; }
int fun4(int x) { return x/2; }
using rFun = int (&)(int);
```

```
int main() {
    int (&func)() = fun2;
    int x = func(4); // appel à ::fun2(4);
}
```

```
struct A {
    rFun x;
    A(rFun v): x(v) {}
    int do_it(int v) { return x(v); }
};

int main() {
    A  a[4] = { fun1, fun2, fun3, fun4 };
    for(int i=0;i<4;i++)
        cout << a[i].do_it(i*i) << endl;
}
```

5 Fonctor

Une fonction-objet (ou fonctor) est un objet qui peut être appelé comme s'il était une fonction.

Principe :

- on crée une classe `C` qui surcharge `operator()` avec le nombre et le type d'arguments que l'on souhaite (possiblement avec des surcharges), par exemple : `int operator()(int a, int b)`
- avec cette surcharge, pour tout objet `c` de la classe `C`, l'écriture `c(x,y)` fait appel à cette surcharge.
- la classe peut comporter des états internes permettant de paramétrer la fonction. Ces états internes sont configurés avec les constructeurs ou reparamétré avec des setters.

Exemple :

```
struct LessThan {
public: bool operator()(int u, int v) { return (u < v); }
};
```

```
LessThan  lessthan;
bool t = lessthan(4,12);
```

Dans le cadre de la STL, les fonctors jouent le rôle de :

- **prédicat :** le fonctor est de la forme `bool pred(const T &a)` et permet de tester si un élément a vérifie une certaine propriété.
- **clef :** le fonctor est de la forme `bool cmp(const T &a, const T &b)` et permet de comparer deux éléments a et b.

Exemple :

```
struct A { int i; ... };

// predicat vérifiant une condition sur un objet de type A
struct Valid {
    int v;    // valeur avec laquelle comparer
    Valid(int V) : v(V) {}
    bool operator()(const A &a) { return (v==a.i); }
};

// clef de comparaison entre deux objets de type A
struct CompEqualA {
    bool operator()(const A &a1, const A &a2) {
        return (a1.i==a2.i);
    }
};
```

Autres types de fonctor :

- **modificateur d'éléments :** modifie l'élément sur lequel il est appelé.
  Exemple :

```
struct Trans {
  int v;    // valeur de translation
  Trans(int V) : v(V) {}
  void operator()(A &a) { a.i += v; }
};
```

```
A a(6);
Trans trans(4);
trans(a);
```

- **traitement d'éléments :** effectue un calcul sur un ensemble d'éléments.
  Exemple :

```
struct Sum {
    int s{};  // somme
    void operator()(A &a) { s+= a.i; }
    void reset() { s=0; }
};

Sum sum;
for(int i=0;i<n;i++) sum(a[i]);
// sum.s contient le résultat
```

Remarque : Les fonctors étant souvent associés à un type particulier, ne pas hésiter à les inclure dans les parties publiques des classes associées.

Exemple :

```
class A {
protected: int a{};
public:
    A(int v) : a(v) {};
    // internal fonctor
    struct less {
        bool operator()(const A& left, const A& right) const
            { return (left.a < right.a); }
    };
};

A v[5] = {5, 7, 4, 2, 8};
std::sort(v,v+5, A::less());
```

Pourquoi utiliser des fonctors plutôt que des pointeurs de fonction ?

- **les fonctors de la STL sont des templates**, à savoir du code générique qui peut être compilé pour le type souhaité,
  Exemple : le fonctor `std::greater<T>` ne compilera que si l'opérateur > est implémenté pour le type `T`.
- **les pointeurs de fonctions ne sont pas génériques**
  il faudrait créer les fonctions templates associées, forcer leurs instanciations et prendre leurs pointeurs.
- **un fonctor est plus flexible qu'une fonction**
  notamment, un fonctor peut avoir des états internes, permettant de la configurer avant utilisation, et conserver des résultats après.

6 Prédicats et opérateurs

Deux types de fonctions ou méthodes jouent un rôle fréquent dans le cadre de la conception de méthode/fonction prenant en paramètre des méthodes/fonctions :

1. les **prédicat**s : un prédicat est une fonction ou une méthode permettant de décider une propriété sur les arguments qui renvoie une valeur booléenne (vraie si la propriété est vérifiée et fausse sinon).

   Exemple : compter (resp. copier, supprimer) les éléments d'une liste qui vérifient le prédicat.

2. les **opérateurs** : un opérateur est une fonction ou une méthode permettant d'appliquer une opération unaire ou binaire sur des objets.

   Exemple : trier les éléments d'une liste qui utilise un opérateur de comparaison pour ordonner deux éléments.

Nous examinons maintenant le cadre d'utilisation des ces fonctions.

6.1 Prédicats

Cette section sera uniquement illustrée par un exemple.

Supposons que nous voulions écrire une fonction qui compte le nombre d'éléments d'un tableau qui vérifient une certaine propriété (= un prédicat).

Cette fonction étant générique, écrivons-la sous la forme d'un template permettant de fonctionner pour des tableaux de tout type, et pour toute propriété.

Exemple : template de comptage

```
template <class T> size_t ArrayPredCounter(
  size_t ArraySize,      // taille du tableau
  const T* Array,        // tableau
  bool(*Pred)(const T&) // prédicat
) {
  size_t count = 0;
  for (size_t i = 0; i < ArraySize; i++)
    if ((*Pred)(Array[i])) count++;
  return count;
}
```

On souhaite utiliser cette fonction pour compter le nombre d'éléments pairs dans un tableau d'entiers.

Exemple : nombre d'entiers pairs

```
// prédicat sur un entier (vrai si pair)
bool IsEven(const int &x) { return (x % 2 == 0); }
```

```
int     t[6] = { 1, 4, 3, 7, 8, 5 };
size_t   c1 = ArrayPredCounter(6, t, IsEven);
// ici c1 = 2
```

On souhaite utiliser cette même fonction pour compter le nombre de chaines de caractères d'un tableau qui sont des palindromes.

Exemple : nombre de palindromes

```
// prédicat sur une chaine (vrai si polindrome)
bool IsPalindrom(const string &s) {
  size_t End = s.size() - 1, HalfLen = s.size() / 2;
  for (size_t i = 0; i < HalfLen; i++)
    if (s[i] != s[End - i]) return false;
  return true;
}
```

```
string  s[4] = { "toto", "radar", "zoé", "papou" };
size_t   c2 = ArrayPredCounter(4, s, IsPalindrom);
// ici c2 = 1
```

6.2 Opérateur

On reste dans l'esprit d'écrire des fonctions génériques.

Si la fonction générique utilise un opérateur, alors pour le type sur lequel cet opérateur est utilisé, l'opérateur doit être défini, soit implicitement (par exemple, sur les entiers), soit explicitement (par

redéfinition d'opérateur).

Problèmes :

- si l'opérateur n'est pas défini, la compilation du template échouera.
- si l'opérateur est défini d'une manière inadéquate par rapport à son rôle attendu dans la fonction, la fonction n'est pas utilisable.

Conséquence :

Il préférable d'encapsuler l'opérateur dans un fonctor template représentant ce même opérateur.

On peut ainsi utiliser soit l'opérateur réel, soit tout opérateur alternatif.

A cet effet, la STL prédéfinit un ensemble de fonctors (templates) correspondant aux opérateurs les plus courants (`constexpr` à partir de C_{15}^{++}) :

Opérations arithmétiques

`plus`	x+y
`minus`	x-y
`multiplies`	x*y
`divides`	x/y
`modulus`	x%y
`negate`	-x

Opérations de comparaison

`equal_to`	$x = y$
`not_equal_to`	$x \neq y$
`greater`	$x > y$
`less`	$x < y$
`greater_equal`	$x \geq y$
`less_equal`	$x \leq x$

Opérations binaires

`bit_and`	`x & y`
`bit_or`	`x | y`
`bit_xor`	`x ^ y`

Opérations logiques

`logical_and`	`x && y`
`logical_or`	`x || y`
`logical_not`	`!x`

Négateur

- `unary_negate<T>` : négation de l'opération unaire `T::operator(x)`
- `binary_negate<T>` :négation de l'opération unaire `T::operator(x,y)`

Utilisation :

- inclure `<functional>`,
- surcharger si besoin l'opérateur associé dans le template.

Exemple : template de recherche d'un élément

```
template <class T> size_t Find(size_t Size, const T* Array,
    const T& Val) {
   size_t Pos = 0;
   while(Pos<Size) if (Array[Pos] == Val) break; else Pos++;
   return Pos;
}
```

Ce template retourne la position du premier élément ayant la valeur recherchée dans un tableau. Il est basé sur l'hypothèse que l'opérateur == est défini pour le type `T`.

Si on remplace l'opérateur == par `equal_to`, on ne permet pas de fournir son propre opérateur, il faut donc le passer en paramètre du template sous forme d'un fonctor [1].

1. Il serait possible de passer un pointeur de fonction, mais il y a un risque d'incompatibilité du pointeur (convention d'appel).

```
template <class T, class Equal = std::equal_to<T>>
  size_t Find(size_t Size, const T* Array, const T& Val) {
  size_t Pos = 0;
  Equal  equal;
  while (Pos < Size)
    if (equal(Array[Pos],Val)) break; else Pos++;
  return Pos;
}
```

Dans ce cas, l'écriture `Equal()(Array[Pos],Val)` est aussi possible.

Commentaires sur ce code :

- le fonctor est un argument du template dont la valeur par défaut est l'implémentation de l'opérateur == sur un objet de type `T`.
- la déclaration de `equal` permet de créer une instance d'un fonctor de type `Equal`.
- le `if` permet l'appel de ce fonctor avec les paramètres appropriés.

On peut ainsi effectuer les appels suivants :

```
int     t[6] = { 1, 4, 3, 7, 8, 5 };
// trouve l'indice de 7 dans le tableau t
size_t i1 = Find(6, t, 7);
// trouve l'indice du premier élément différent de 4 dans t
size_t i2 = Find<int, std::not_equal_to<int>> (6, t, 4);
```

ou encore :

```
struct divide {
    bool operator()(int x,int y) { return (x % y == 0); }
};
```

```
// trouve l'indice du premier élément divisible par 4
size_t i3 = Find<int,divide> (6, t, 4);
```

Ce qui démontre encore la flexibilité des fonctors sur les pointeurs de fonctions.

Inconvénient de l'approche précédente, le fonctor ne peut pas avoir d'états internes (le constructeur par défaut doit être utilisé : seul le type est passé).

Une approche alternative est de passer l'objet fonctor instanciée. Le code devient alors :

```
template <typename T, typename Equal = std::equal_to<T>>
  size_t Find(size_t Size, const T* Array, const T& Val,
  Equal &equal = Equal()) {
  size_t Pos = 0;
  while (Pos < Size)
    if (equal(Array[Pos],Val)) break; else Pos++;
  return Pos;
}
```

L'ensemble des appels suivants deviennent alors possibles :

```
size_t c1 = Find(6, t, 7);
// comme avant
size_t c2 = Find<int, std::not_equal_to<int>>(6, t, 4);
// remarque: le dernier argument est un constructeur
size_t c3 = Find(6, t, 4, std::not_equal_to<int>());
// remarque: passage d'un fonctor avec état interne
pgcd    pgcd3(3);  // pgcd3(x,y) vrai si pgcd(x,y)=3
size_t c4 = Find(6, t, 9, db3);
```

Deuxième exemple : fonction `sort` de la STL

La fonction `std::sort` est l'évolution C++ de la fonction `qsort` du C.

Exemple :

```
const size_t  len = 6;
int t[len] = { 18, 1, 4, 3, 7, 5 };
// utilise operator< (s'il est défini, échoue sinon)
sort(t, t + len);
// utilise le fonctor greater (= inverse l'ordre de tri)
sort(t, t + len, std::greater<int>() );
```

```
bool evencmp(const int &a, const int &b) {
  if (a%2 == 0) return (b%2 == 0 ? a < b : true);
  else          return (b%2 == 0 ? false : a < b);
};
```

```
// tri pair puis impair (evencmp = pointeur fonction)
sort(t, t + len, evencmp);
```

Analyse : pour un tableau de type `T`

- par défaut, la surcharge de l'opérateur < sur le type `T` est utilisé pour trier le tableau (si elle n'existe pas, la fonction `sort` échoue).
- un fonctor permettant de comparer deux éléments de type `T` peut être passé soit pour changer la méthode de comparaison par défaut (si `operator<` est défini), soit pour fournir une méthode de comparaison.

Pour résumer, le type des objets qu'il est possible de passer peut être :

- soit aucun (s'il y a un fonctor par défaut)

```
std::sort(v,v+5);
```

- soit un fonctor temporaire

```
std::sort(v,v+5, greater<int>());
```

- soit un objet fonctor

```
greater<int>    igreater;
std::sort(v,v+5, igreater);
```

- soit un fonctor de type non nommé

```
struct { bool operator()(const int &a, const int &b)
                            { return a<b; } } iless;
std::sort(v,v+5, iless);
```

- soit une fonction

```
bool iless(const int &a, const int &b) { return a<b; }
```

```
std::sort(v,v+5, iless);
```

- soit une lambda expression (voir ci-après)

```
std::sort(v,v+5, bool [](const int &a, const int &b)
    { return a<b; });
```

On obtient ainsi une flexibilité maximale.

7 Fonctionoïd

Une fonctionoïd (pour fonction sous stéroïd) consiste à gérer un pointeur de fonction ou un tableau de pointeurs de fonction en utilisant des fonctors.

Le principe est le suivant :

1. créer une classe virtuelle qui définit une méthode virtuelle dont le prototype possède les arguments et le type de retour souhaité pour l'appel de la méthode.
   ne surtout pas oublier le destructeur virtuel (comme pour toute classe virtuelle).
2. pour chaque implémentation différente souhaitée de la méthode, créer une classe particulière qui hérite de la classe virtuelle et qui implémente la méthode virtuelle.
   Noter que ces classes virtuelles peuvent chacune avoir des états internes différents (initialisés par un constructeur, voir modifiés en cours d'exécution par des setters).
3. au lieu d'un pointeur de fonction, on utilise un pointeur sur la classe virtuelle (de base) que l'on peut ainsi faire pointer vers n'importe quelle instance d'une des classes dérivées.

Noter que dans le cas du fonctionoïd, il n'y a pas de contrainte sur le nom de la méthode à surcharger (ce n'est pas nécessairement `operator()`). On peut donc y stocker plusieurs états ET méthodes différentes.

Exemple :

```
// créer une classe mère contenant une méthode virtuelle
// (par exemple do_it).
class Funct {
public: virtual int do_it(int x) = 0;
        virtual ~Funct() = 0;
};
```

```
inline Funct::~Funct() {} // important
// créer un ensemble de classes qui hérite de la classe mère
// et implémente la méthode do_it
class Funct0 : public Funct {
   public: virtual int do_it(int x) { /* impl0 */ } ... };
class Funct1 : public Funct {
   public: virtual int do_it(int x) { /* impl1 */ } ... };
...
```

```
// utilisation comme pointeur seul
Funct0    f0( /* arguments constructeur Funct0 */ );
Funct1    f1( /* arguments constructeur Funct1 */ );
Funct     *ptFunc = &f0;
ptFunc->do_it(2);
```

```
// utilisation comme tableau de pointeur
Funct *tab[2];
tab[0] = new Funct0( /* arguments constructeur Funct0 */ );
tab[1] = new Funct1( /* arguments constructeur Funct1 */ );
tab[0].do_it(4);
```

Remarquer que cette méthode peut être utilisée pour uniformiser l'accès à des fonctions/méthodes dont les prototypes sont différents.

Supposons que l'on souhaite avoir un fonctionoïd `int Funct::do_it(int x)` à partir des fonctions suivantes :

- `int funct1(int x, float y)` où `y` est un état connu
- `int funct2(int x, int n, const vector<double>& y)` où `n` et `y` sont des états connus,
- `int A::funct3(int y, int x)` où l'instance de l'objet `A` et l'état de `y` sont connus.

La solution consiste à :

- avoir une classe mère contenant une méthode virtuelle représentant l'accès commun souhaité.
- créer une classe Fonctionoïd par fonction, qui hérite de la classe mère et qui stocke dans ses états internes les valeurs des paramètres supplémentaires nécessaires.

Tous les fonctionoïds ont le même prototype d'appel (hérité), l'hétérogénéité de chaque appel et les états nécessaires à cet appel étant occultés dans le fonctionoïd (démarche généralisée avec les objets fonctionnels).
Exemple avec la méthode décrite ci-dessus :

```
class Funct1 : public Funct {
private: float y_;
public:
    Funct1(float y) : y_(y) { }
    virtual int do_it(int x) { /* code funct1(x,y_) */ }
};

class Funct2 : public Funct {
private: int n_;
         const vector<double>&   y_;
public:
    Funct2(int n, const vector<double>& y) : n_(n), y_(y) {}
    virtual int do_it(int x) { /* code funct2(x,n_,y_) */ }
};

class Funct3 : public Funct {
private: A   &a_;
         int y_;
public:
    Funct3(A& a, int y) : a_(a), y_(y) {}
    virtual int do_it(int x) { a_.funct3(y_,x); }
};
```

8 Lambda expression

8.1 Introduction

Un objet ou une expression `f` est appelable (callable) si on peut écrire `e(args)` où `args` est liste d'arguments séparé par des virgules.

Il existe 4 types d'objets ou expressions appelable : les fonctions, les pointeurs de fonction, les fonctors, et les lambda-expressions.

Caractéristiques : une lambda-expression (notée *λ-exp*)
- représente une unité de code appelable,
- a un type de retour, une liste de paramètres et un corps,
- peut être définie à l'intérieur d'une fonction,

peut être vu comme une fonction inline sans nom.

Intérêt d'une lambda-expression : définir au vol une fonctionnelle qui doit être passée en paramètre à une autre fonction.
- Il existe de nombreuses fonctions définies dans la stl permettant d'appliquer une fonctionnelle sur des containers.
- En général, une bonne idée d'avoir un ou plusieurs paramètres fonctionnels afin d'appliquer le même algorithme sur des objets complexes
  Déjà dans l'air avec la surcharge d'opérateur + template.

8.2 Définition d'une λ-expression

Une lambda-expression est de la forme :
```
[liste-capture] (liste-paramètres) -> type-retour { corps }
```

- `corps` : corps de la lambda-expression.
- `liste-capture` : liste des variables locales définie dans le corps de la *λ-exp*.
- `liste-paramètres` : liste des paramètres passés à la *λ-exp*.
- `-> type-retour` : type de retour de la *λ-exp* (peut être omis, et dans ce cas, le type est déduit de l'argument du `return`).

Exemple : d'une lambda-expression (hors contexte)
```
[](int val) { return (0 < val) && (val < 10); }
```

Liste de capture :
- `[a,&b]` : capture `a` par valeur, capture `b` par référence.
- `[this]` : capture le pointeur `this` par valeur (dans une méthode).
- `[&]` : capture toutes les variables locales par référence.
- `[=]` : capture toutes les variables locales par valeur.
- `[]` : ne capture rien.
- `[=,&a]` : capture tous les variables locales par valeur, sauf `a` qui est capturé par référence.
- `[&,=a]` : capture tous les variables locales par référence, sauf `a` qui est capturé par valeur.

Attention :
- on ne capture que les variables locales non statiques.

- la liste de capture ne doit pas capturer plusieurs fois le même élément.
  exemple : `[=]` contient `this`, `[&]` contient `&a`, ...
- la capture par un argument d'une entité ne prolonge pas sa durée de vie (ce point est précisé ci-après).
- pour capturer les membres de l'objet courant dans une de ses méthodes, capturer `this`.

Exemple :

```
size_t v1 = 42; // local variable
auto fun1 = [v1] { return v1; };   // passage par copie
auto fun2 = [&v1] { return v1; };  // passage par référence
v1 = 0;
// appels
auto x1 = fun1(); // x1 est 42 (copie de v1)
auto x2 = fun2(); // x2 est 0 (référence vers v1 )
```

8.3 Exemples

Exemple d'utilisation avec la STL :

`std::transform` applique un opérateur sur chaque élément d'un intervalle d'un container `[first,last]`, et écrit le résultat à l'emplacement `result` indiqué.

L'implémentation est typiquement :

```
template < class InputIterator , class OutputIterator ,
           class UnaryOperator >
   OutputIterator transform(InputIterator first,
             InputIterator last, OutputIterator result,
             UnaryOperator op) {
    while (first != last) {
       *result = op(*first);
       ++result;
       ++first;
    }
    return result;
}
```

Utilisation avec une *λ-exp* pour appliquer en place le modulo 2 sur chacun des éléments d'un container :

```
int m = 2;
std::vector<int> v{5, 32, 7, 12, 28};
std::transform(v.begin(), v.end(), v.begin(),
                   [m](int x){ return x % m;});
```

Exemples : d'utilisation d'une lambda expression

- **Pas de capture :**

```
vector<string> words;
// initialisation de words ici
// tri de ces mots par taille croissante
stable_sort(words.begin(), words.end(),
   [](const string &a, const string &b)
     { return a.size() < b.size(); });
```

- **Capture d'un entier (par copie) :**

```
size_t   sz = 6;
// retourne un itérateur sur le premier mot de taille
// supérieure où égale à 6
auto wc = find_if(words.begin(), words.end(),
       [sz](const string &a){ return a.size() >= sz; });
```

- **Calcule le produit des éléments d'un container :**

```
std::vector< int > v{2, 3, 4};
int prod = 1;
std::for_each(v.begin(), v.end(),
     [&prod]( int x)-> void {prod *= x;});
// prod contient le résultat
```

9 Objets fonctionnels

9.1 Introduction

Avec la multiplication des formes fonctionnelles, nous avons donc de nombreuse manière différente de créer des objets fonctionnelles :

- dont les formes d'appel sont toutes similaires,
- mais dont tous les types sont différents (en raison du typage fort du C++)

Exemple :

```
int fonction(int x) { return x + 1; }
int(*ptrfonction)(int x) = fonction;
struct Fonctor {
    int operator()(int x) { return x + 1; } };
auto lambfonction = [](int x) { return x + 2; };
// -> int : implicite
```

```
Fonctor fonctor;
int x1 = fonction(1);
int x2 = ptrfonction(1);
int x3 = fonctor(1);
int x4 = lambfonction(1);
```

Il serait donc approprié de disposer d'un moyen d'unifier l'ensemble des objets fonctionnels dont la forme de l'appel et le type de retour est le même.

9.2 Conteneur fonctionnel typé

La classe template `std::function` du C_{11}^{++} permet d'encapsuler tout type de fonctionnelles ayant le même type de paramètres en entrée et en sortie dans un objet de même type.

Le type du template (obligatoire) permet de définir les types d'entrées et le type de sortie de manière identique à un pointeur de fonction.

Exemple 1 : pour une fonctionnelle prenant en paramètre un entier et un flottant (dans cet ordre) et retournant un booléen, le type est : `bool(int,float)`.

Exemple 2 :

```
// en reprenant les définitions de l'exemple précédent
std::function<int(int)>   f1 = fonction;
std::function<int(int)>   f2 = lambfonction;
std::function<int(int)>   f3 = fonctor(3);
// constructeur du fonctor
std::function<int(int)>   f4 = ptrfonction;
// on aurait aussi pu créer un tableau
std::function<int(int)>  fun[4] = { f1,f2,f3,f4 };
// utilisation
int v = f1(1) + f2(1) + f3(1) + f4(1);
for (int i = 0; i < 4; i++) v += fun[i](5);
```

Applications : permet de définir une fonctionnelle qui doit être appelée à partir de n'importe quelle source fonctionnelle.

Exemple 3 : appels de plus ou moins niveau associé à une fonction

- **handler** (référence vers un objet associé à un comportement) : dans le handler, le comportement a avoir est associé à une fonction,
- **trigger** (comportement en réponse à un stimulus) : dans le trigger, fonction que déclenche le stimulus,
- **callback** (référence à une fonction exécuté par un tier) : l'objet fonctionnel propre.
- l'ensemble de ces objets peuvent être appelé dans des listes ou files d'attente d'évènements (pour les handlers), de stimuli (pour les triggers), de tâches à exécuter (pour les callbacks).

On accroit ainsi considérablement la flexibilité des fonctionnelles utilisables.

Néanmoins, cette approche a encore les limitations suivantes :

- lors d'une approche objet, les fonctionnelles deviennent plus rares, et celles qui sont intéressantes pourraient être des méthodes de classe
  donc, non utilisable comme fonctionnelle.
- il n'est pas possible d'utiliser des fonctions dont certains paramètres seraient connus,

En première étape, l'utilisation de membres d'une classe/structure d'un objet peuvent être transformés

en fonctionnelles.

Pour les méthodes, il est également possible d'utiliser `function`.

Exemple 4 :

```
// objets fonctionnels
std::function<int(fonctor&, int)>
f1 = &fonctor::incr;
std::function<int(const member&, int)>
f2 = &member::val;
// déclaration des objets
fonctor  obj1(4);
member   obj2(7);
// appel des fonctionnelles
int v = f1(obj1, 1) + f2(obj2, 1);
```

A noter que :

- pour une méthode, l'utilisation de & est obligatoire.
- transforme l'appel d'une méthode sur un objet en une fonctionnelle dont le premier paramètre est l'objet sur lequel la méthode doit être appelée.
- le type de premier paramètre est donc le type de l'objet auquel la méthode appartient. S'il est `const`, cela signifie que la fonction membre est `const`.

9.3 Conteneur fonctionnel pour membre

Une alternative `mem_fn` qui permet un accès aux méthodes ou aux champs :

Exemple :

```
// définition de la classe
class member {
public: int a;
   member(int x) : a(x) {}
   int val(int x) const { return a + x; }
};
// objets fonctionnels
auto p1 = std::mem_fn(&member::val);   // méthode
auto p2 = std::mem_fn(&member::a);     // champs
// utilisation
member   obj(7);      // définition de objets
int v = p1(obj, 4);   // équivalent appel obj.val(4)
v += p2(obj);         // équivalent de obj.a
```

Remarques :

- l'accès à tout membre (champ ou donnée) d'un objet peut être transformé en fonctionnelle.
- ce qui est private/protected reste private/protected.
- retour en type `auto` car inutilisable tel quel.

9.4 Binding fonctionnel

`bind` permet de construire des fonctionnelles sans argument en précisant les valeurs par défaut devant être prise en paramètres.

Exemple 1 :

```
int fun(int x, int y, int z) { return x + 2 * y + 4 * z; }
// création d'un objet fonctionnel avec binding
int  a=4,x;
auto p = std::bind(fun,a,1,1);
x = p();  // équivalent de l'appel fun(4,1,1) -> 10
a=0;
x = p();  // équivalent de l'appel fun(4,1,1) -> 10
```

Remarques : par défaut,

- La totalité des paramètres de la fonctionnelle doit être fournie.
- Tous les paramètres passés à un `bind` sont stockés localement dans le `bind` avec les valeurs des paramètres au moment de l'appel au constructeur de l'objet `bind` (i.e. constructeur par copie si le paramètre est une variable, constructeur par déplacement si c'est un temporaire).

conséquence : changer les paramètres passés au constructeur d'un `bind` ne change pas les paramètres avec lesquels l'objet fonctionnel créé par `bind` s'exécute.

Évidemment, il est parfois souhaitable de créer un véritable lien entre le paramètre passé au `bind` est le paramètre utilisé au moment de l'appel.

Ceci peut être réalisé avec :

- `std::ref` : la variable est stockée dans `bind` sous forme d'une référence.
- `std::cref` : la variable est stockée dans `bind` sous forme d'une référence constante.

Dans ce cas, il n'y a plus de copie dans `bind`, et l'appel à l'objet fonctionnel utilise la valeur de l'objet au moment de l'appel de l'objet fonctionnel.

Exemple 2 :

```
int fun(int x, int &y, const int &z) { y+=z; return y+x; }
// création d'un objet fonctionnel avec binding
int         a = 1;
const int   b = 1;
auto        p = bind(fun2, 1, ref(a), cref(b));
x = p();  // équivalent de l'appel fun(1,a,b) -> x=3, a=2
x = p();  // équivalent de l'appel fun(1,a,b) -> x=4, a=3
```

Remarque : attention à faire en sorte que la référence passée existe toujours lors de tous les appels à l'objet fonctionnel `bind`.

Enfin, il reste à permettre un passage partiel de paramètres (`bind` partiel), et une réorganisation des paramètres.

Ceci est réalisé avec `std::placeholders`. Alors `_1`, `_2`, `_3`

- représente symboliquement le $i^{\text{ème}}$ paramètre du `bind`.
- dans l'appel de la fonctionnelle, indique où le paramètre est utilisé.

Exemple 3 :

```
int fun2(const A &x, const A &y, const A &z)
    { return x.a + y.a + z.a + 1;   }
// création d'un objet fonctionnel paramétré
using namespace std::placeholders;
int a1 = 4;
auto p = std::bind(fun2, _2, 4, _1);
// appel de fun2(7,4,a1)
x = p(a1,7);     // ici _1 = a1 et _2 = 7
```

Remarques :

- le placeholder passe le paramètre exactement de la façon dont il serait passé avec un appel direct à la fonctionnelle,
- les paramètres peuvent être réordonnés comme souhaité.

Exemple 4 :

```
// formes fonctionnelles avec multiples paramètres
struct fonctor {  int a;
  fonctor(int x) : a(x) {}
  int operator()(int x, int y, int z) { return x*y + z*a; } };
struct member {   int a;
  member(int x) : a(x) {}
  int val(int x, int y) const { return a + x + y; } };
int fonction(int x, int y) { return x + y + 1; }
auto lambfonction = [](int x, int y, int z) { return x+y+z; };
int(*ptrfonction)(int,int) = fonction;
```

```
// unification en une seule forme fonctionnelle int(int)
int   a = 4, b = 6;
member  c(2);
using namespace std::placeholders;
function<int(int)>   f1 = bind(fonction,_1,a);
function<int(int)>   f2 = bind(lambfonction,ref(b),_1,a);
function<int(int)>   f3 = bind(fonctor(3),a,a,_1);
function<int(int)>   f4 = bind(ptrfonction,a,_1);
function<int(int)>   f5 = bind(&member::val, ref(c), _1);
// utilisation
int x = f1(4) + f2(2) + f3(5) + f4(8) + f5(3);
```

10 Complément sur les λ-expressions

10.1 Implémentation

Vocabulaire associé :

- une **lambda expression** est juste une expression (partie du code).
- une **fermeture** (closure) est l'objet créé à l'exécution par une λ-*exp*. Cet objet contient les copies ou les références des variables capturées.
- le **classe de la fermeture** (closure class) est la classe (unique pour chaque λ-*exp*) avec laquelle la fermeture est instanciée.

Une λ-*exp* est implémentée sous forme d'un fonctor avec `operator()` toujours `inline`.

Exemple : Implémentation équivalente d'une λ-expression avec un fonctor

```
class cum_prod {
private: int & prod;
public: cum_prod( int & prod_) : prod(prod_) {}
        inline void operator ()(int x) const {prod *= x;}
};
```

```
std::vector< int > v{2, 3, 4};
int prod = 1;
std::for_each(v.begin(), v.end(), cum_prod(prod));
// le fonctor appelé est cum_prod(prod)(*it)
```

Conversion vers les autres types d'objets fonctionnels :

- Une λ-*exp* peut être stockée dans un objet `std::function` mais elle ne sera probablement pas `inline` :
  Exemple :
  ```
  std::function<int(int,int)> fun2
        = [=](int a, int b) { return a+2*b; };
  ```
- Une λ-*exp* peut être convertie en un pointeur de fonction.
  Exemple :
  ```
  int& (*fpi)(int*) = [](int* a)->int& { return *a; };
  ```
- Une λ-*exp* ne peut pas être convertie en une référence de fonction.
  Exemple :
  ```
  int& (&rpi)(int*) = [](int* a)->int& { return *a; };   // invalide
  ```
  En revanche, ceci peut être réalisé depuis un pointeur de fonction :
  Exemple :
  ```
  int& (&fpi)(int*) = [](int* a)->int& { return *a; };
  int& (&rpi)(int*) = *fpi;                              // valide
  ```

Remarques :

- Le type d'une fermeture ne peut pas être nommé, mais il peut être déduit avec auto.
  Exemple :
  ```
  auto fun1=[](int i) { return i + 4; };
  ```
- La qualification `mutable` d'une λ-*exp* permet au corps de modifier les paramètres capturés par copie, ou appeler leurs méthodes non const.
  Exemple :
  ```
  int count = 5;
  auto get_count = [count]() mutable
       -> int { return count++; };
  int c;
  while ((c = get_count()) < 10) {
           std::cout << c << "\n"; }
  ```

10.2 Captures et références

Attention : une λ-*exp* a été conçue pour s'exécuter localement dans l'environnement dans lequel elle est définie :

- dans la mesure du possible, essayer d'utiliser une lambda-expression localement,
- sinon, faire **très attention** aux références invalides (dangling references).

Exemple :

```
auto GetLambda() {
    int Z = 4;
    return [&](int x) { return Z + x; };
    // la lambda fait référence à la variable locale Z
}   // détruite ici
```

```
auto fun = GetLambda();
// ici, Z n'existe plus
int y = fun(4); // utilise une variable locale détruite
```

Conséquence : quand une *λ-exp* n'est pas utilisée localement, il faut :

- éviter de faire de capture par référence,
- éviter d'utiliser des pointeurs (capture par valeur), y compris `this`,
- éviter d'utiliser de champs dans la méthode d'un objet (car leurs résolutions utilise `this`),
- éviter d'utiliser de variable `static const` (pas capturée et équivalent à une référence à la variable `static const`).

Sinon, il faut :

- s'assurer que tous les objets capturés par la *λ-exp* (et qu'elle utilise) existent encore au moment de l'appel,
- **et** que la valeur souhaitée soit celles des objets au moment de l'appel et non de leurs définitions.

Solution partielle : à partir de C^{++}_{15}, il est possible de capturer par valeur avec la syntaxe `[val=exp]` où :

- `val` est une variable définie dans la portée de *λ-exp*,
- `exp` est une expression définie dans la portée de la classe de fermeture (i.e. = le contexte de la définition de la *λ-exp*).

Cette méthode permet de conserver la valeur d'une expression ou d'une valeur pointée telle qu'elle était à la création de la *λ-exp* (*λ*-capture généralisée).

Exemple :

```
int *p = new int(10);
auto fun1 = [=]() { return *p; };
auto fun2 = [v=*p]() { return v; };
*p = 5;
// ici: l'appel fun1() retourne 5
// ici: l'appel fun2() retourne 10
```

Peut aussi être utilisée pour déplacer un objet dans une *λ-exp*.

Exemple : `auto func = [pw = std::move(pw)] { return pw->isValid(); }`

Note : fonctionnalité implémentée sous VS2015.

Alternative pour C^{++}_{11} :

`std::bind` crée un wrapper contenant un appel d'une fonction `f` associé à certains de ses argu-

ments (défini dans <functional>).

Exemple :
```
auto fun = [](int x) { return x*x; };
auto bindedfun = std::bind(fun,4);
bindedfun();    // appel fun(4)
```

L'ensemble des arguments de la fonction doit être renseigné, mais l'utiliser de `placeholders` permet de lever cette contrainte (où `_i` = $i^{\text{ème}}$ argument du wrapper).

Exemple :
```
using namespace std::placeholders;
auto fun = [](int a, int b) { return a*b; };
auto bindedfun = std::bind(fun,_1,4);
bindedfun(2);    // appel fun(2,4)
```

Les arguments de `bind` sont copiés (lvalue) ou déplacé (rvalue), mais jamais passé par référence, sauf si l'argument est wrappé avec `std::ref` (ou `std::cref` pour référence constante).

En conséquence, une λ-capture généralisée peut s'écrire en C_{11}^{++} (en reprenant l'exemple précédent) :

```
auto fun2 = bind([](int v) { return v; },*p);
```

Conclusion

Nous avons vu dans cette leçon que :

- la convention d'appel des fonctions est un élément dont il faut tenir compte afin de déclarer des objets fonctionnels compatibles.
- les pointeurs de fonction sont l'outil bas-niveau permettant de passer une fonctionnelle à une fonction ou de stocker son pointeur dans un objet.
- les fonctors sont des objets fonctionnels permettant de stocker des états, plus flexibles que les fonctions.
- Les fonctionoïds consistent à permettre reprendre compatible des fonctors hétérogènes en les faisant hériter d'une classe mère commune exposant les fonctionnalités communes.
- Les lambda-expressions permettent de créer des fonctions au vol dans un code, et de les passer en lieu et place de toute fonction.
- La classe `function` permet de fournir l'encapsulation de n'importe quel objet fonctionnel du C^{++}. Combiné avec `bind` et les `placeholders`, elle offre un moyen d'uniformiser les appels à des fonctions hétérogènes.

Ce chapitre n'est pas à négliger, car la maitrise des différents objets fonctionnels permettent une expressivité et une flexibilité plus grandes.

Depuis le C_{11}^{++}, la maîtrise des lambda-expressions est devenu obligatoire, en particulier pour l'utilisation des algorithmes fournis par la STL.