

# LANGUAGE OBJ. AV.( C++ ) MASTER 1

Yan Jurski

U.F.R. d'Informatique  
Université de Paris Cité

Aujourd'hui, 2 chapitres distincts :

- Les Modèles

(du genre `vector<T>`)

- Les Interfaces Fonctionnelles

(foncteurs, lambda)

# LES MODÈLES

- une pratique déjà connue, exemple illustratif
- la compilation d'un template
  - objectif-difficultés
  - convention d'écriture séparée
- Spécialisation
- Template pour les classes, exemples plus difficiles

# LES MODÈLES

- **une pratique déjà connue, exemple illustratif**
- la compilation d'un template
  - objectif-difficultés
  - convention d'écriture séparée
- Spécialisation
- Template pour les classes, exemples

# LES MODÈLES

vous savez déjà de quoi il s'agit :

la possibilité d'écrire, à partir d'une

définition unique de vector et de list :

- `vector<int>`

- `list<vector<A*>>`

# LES MODÈLES

Nous n'avons pas encore :

- fait un tour explicatif pour c++,
- étudié la syntaxe
- écrit nos propres applications

Comme toujours en c++ il y a beaucoup de choses.

Nous allons essayer de voir l'essentiel.

# LES MODÈLES

Un modèle s'appelle aussi :

"template", "gabarit", "pattern", "patron", "définition générique"

Cette notion s'applique aux fonctions et aux classes.

```
int plusPetit (int a, int b) {  
    if (a<b) return a; else return b;  
}
```

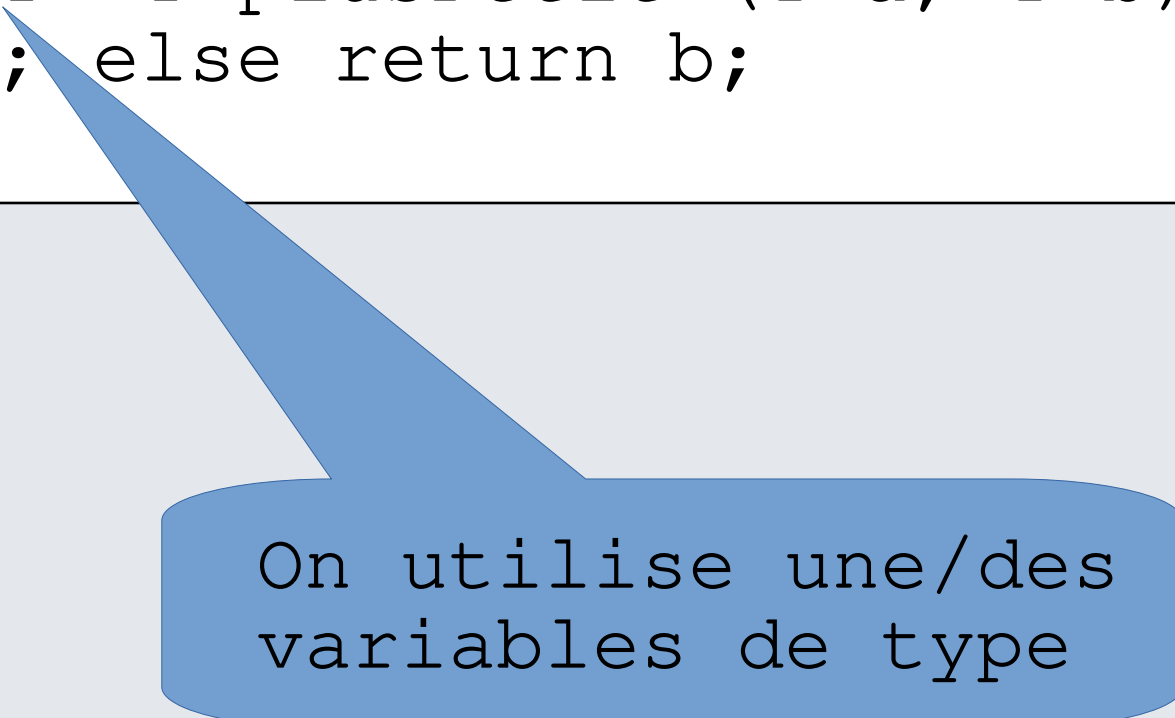
cette fonction pourrait tout aussi bien être définie pour des float, des char, des string, etc ... pour n'importe quel nom de type **homogène**



```
int plusPetit (int a, int b) {  
    if (a<b) return a; else return b;  
}
```

cette fonction pourrait tout aussi bien être définie pour des float, des char, des string, etc ... pour n'importe quel nom de type **homogène**

```
template <typename T> T plusPetit (T a, T b) {  
    if (a<b) return a; else return b;  
}
```



On utilise une/des variables de type

```
int plusPetit (int a, int b) {  
    if (a<b) return a; else return b;  
}
```

cette fonction pourrait tout aussi bien être définie pour des float, des char, des string, etc ... pour n'importe quel nom de type **homogène**

```
template <typename T> T plusPetit (T a, T b) {  
    if (a<b) return a; else return b;  
}
```

```
int i1{1}, i2{2};  
float f1{1.5}, f2{1.9};  
string s1{"toto"}, s2{"machin"};  
cout << plusPetit(i1,i2);  
cout << plusPetit(f1,f2);  
cout << plusPetit(s1,s2);  
cout << plusPetit(i1,f2); // non
```

```
int plusPetit (int a, int b) {  
    if (a<b) return a; else return b;  
}
```

cette fonction pourrait tout aussi bien être définie pour des float, des char, des string, etc ... pour n'importe quel nom de type **homogène**

```
template <typename T> T plusPetit (T a, T b) {  
    if (a<b) return a; else return b;  
}
```

```
int i1{1}, i2{2};  
float f1{1.5}, f2{1.9};  
string s1{"toto"}, s2{"machin"}  
cout << plusPetit(i1,i2);  
cout << plusPetit(f1,f2);  
cout << plusPetit(s1,s2);  
cout << plusPetit<int>(i2,f2);  
cout << plusPetit<float>(i2,f2);
```

On peut forcer le type. Alors le mécanisme de conversion implicite se met en oeuvre sur les arguments

```
int plusPetit (int a, int b) {  
    if (a<b) return a; else return b;  
}
```

cette fonction pourrait tout aussi bien être définie pour des float, des char, des string, etc ... pour n'importe quel nom de type **homogène**

```
template <typename T> T plusPetit (T a, T b) {  
    if (a<b) return a; else return b;  
}
```

```
int i1{1}, i2{2};  
float f1{1.5}, f2{1.9};  
string s1{"toto"}, s2{"machin"}  
cout << plusPetit(i1,i2);  
cout << plusPetit(f1,f2);  
cout << plusPetit(s1,s2);  
cout << plusPetit<int>(i2,f2);  
cout << plusPetit<float>(i2,f2);
```

On peut forcer le type. Alors le mécanisme de conversion implicite se met en oeuvre sur les arguments

1  
1.9

d'ailleurs, notez le résultat 1 qui est (int)1.9

## Remarque 1 :

avec des références il n'y aurait pas de construction/conversion implicite

```
template <typename T> T& plusPetit (T &a, T &b) {  
    if (a<b) return a; else return b;  
}
```

```
int i1{1}, i2{2};  
float f1{1.5}, f2{1.9};  
string s1{"toto"}, s2{"machin"};  
cout << plusPetit(i1,i2);  
cout << plusPetit(f1,f2);  
cout << plusPetit(s1,s2);  
cout << plusPetit<int>(i1,f2);
```

cannot bind non-const lvalue reference of type 'int&' to a value of type 'float' in : plusPetit<int>(i1,f2)

Remarque 2 : pour accepter plusPetit(int ,float)  
sans utiliser le mécanisme de spécialisation de T,  
on aurait pu tenter :

```
template <typename T1, typename T2>  
    ??? plusPetit (T1 a, T2 b) {  
        if (a<b) return a; else return b;  
    }
```

Mais on aurait rencontré un pb de conception pour  
définir correctement le type retour : T1 ou T2 ?

Essayons T1 ...


Remarque 2 : pour accepter plusPetit(int ,float) sans utiliser le mécanisme de spécialisation de T, on aurait pu tenter :

```
template <typename T1, typename T2>
    T1 plusPetit (T1 a, T2 b) {
        if (a<b) return a; else return b;
    }
```

Mais on aurait rencontré un pb de conception pour définir correctement le type retour : T1 ou T2 ?

Essayons T1 ... (ce serait pareil pour T2)

```
int main() {
    int i{2};
    float f{1.9};
    cout << plusPetit(i,f);
    cout << plusPetit(f,i);
}
```



1  
1.9

Le problème : la fonction ne serait pas symétrique.

```
template <typename T> T plusPetit (T a, T b) {  
    if (a<b) return a; else return b;  
}
```

## Petit résumé :

- `typename` exprime un type de base ou de classe (à préférer à `class` qui est cependant toléré)



```
template <typename T> T plusPetit (T a, T b) {  
    if (a<b) return a; else return b;  
}
```

## Petit résumé :

- `typename` exprime un type de base ou de classe
- on peut écrire `template <typename T1, typename T2>`

```
template <typename T> T plusPetit (T a, T b) {  
    if (a<b) return a; else return b;  
}
```

## Petit résumé :

- typename exprime un type de base ou de classe
- on peut écrire `template <typename T1, typename T2>`  
les types T1, T2 sont utilisables partout, pas  
uniquement en argument. Ex : `f(T1 x) { T2 y; ... }`

```
template <typename T> T plusPetit (T a, T b) {  
    if (a<b) return a; else return b;  
}
```

## Petit résumé :

- typename exprime un type de base ou de classe
- on peut écrire `template <typename T1, typename T2>`
- il faut, ici, que `operator<` soit défini sur `T`

```
template <typename T> T plusPetit (T a, T b) {  
    if (a<b) return a; else return b;  
}
```

## Petit résumé :

- `typename` exprime un type de base ou de classe
  - on peut écrire `template <typename T1, typename T2>`
  - il faut, ici, que `operator<` soit défini sur `T`
- pourtant rien ne l'annonce au niveau de `typename`  
(contrairement à java)

# LES MODÈLES

- une pratique déjà connue, exemple illustratif
- **la compilation d'un template**
  - **objectif-difficultés**
  - convention d'écriture séparée
- Spécialisation
- Template pour les classes, exemples

# Fonctionnement :

```
template <typename T> T plusPetit (T a, T b) {  
    if (a<b) return a; else return b;  
}
```

quand le compilateur rencontre ce code, une analyse purement syntaxique est effectuée (sans vérification des méthodes de T)

# Fonctionnement :

```
template <typename T> T plusPetit (T a, T b) {  
    if (a<b) return a; else return b;  
}
```

quand le compilateur rencontre ce code, une analyse purement syntaxique est effectuée (sans vérification des méthodes de T)

```
int main() {  
    int i{2};  
    float f{1.9};  
    cout << plusPetit(i,i);  
    cout << plusPetit(f,f);  
}
```

à la compilation de ce code, l'instanciation de T est faite pour int (puis pour float).

Deux fonctions différentes sont générées à partir du patron, et compilées normalement.

C'est là que l'existence d'operator< est vérifiée.

## Exemple de limites / difficultés possible

imaginez avoir écrit :

```
template <typename T> void imbrique (T a, int n) {  
    if (n == 0) return;  
    vector<T> va;  
    va.push_back(a);  
    imbrique(va, n-1);  
}  
  
int main() {  
    string s{"bottom"};  
    imbrique(s, 1);  
}
```



## Exemple de limites / difficultés possible

imaginez avoir écrit :

```
template <typename T> void imbrique (T a, int n) {  
    if (n == 0) return;  
    vector<T> va;  
    va.push_back(a);  
    imbrique(va, n-1);  
}  
  
int main() {  
    string s{"bottom"};  
    imbrique(s, 1);  
}
```

Le principe d'un modèle, est que le compilateur produise une version optimisée pour les types réellement utilisés.

## Exemple de limites / difficultés possible

imaginez avoir écrit :

```
template <typename T> void imbrique (T a, int n) {  
    if (n == 0) return;  
    vector<T> va;  
    va.push_back(a);  
    imbrique(va, n-1);  
}  
  
int main() {  
    string s{"bottom"};  
    imbrique(s, 1);  
}
```

**Ici la compilation ne termine pas ...**

Le compilateur passe par dessus la condition d'arrêt sans l'évaluer (ce n'est pas son rôle d'exécuter le code).

Son cycle d'instantiation de type est infini.

## Exemple de limites / difficultés possible

imaginez avoir écrit :

```
template <typename T> void imbrique (T a, int n) {  
    if (n == 0) return;  
    vector<T> va;  
    va.push_back(a);  
    imbrique(va, n-1);  
}  
  
int main() {  
    string s{"bottom"};  
    imbrique(s, 1);  
}
```

**Ici la compilation ne termine pas ...**

Remarquez la nature du problème : le compilateur devrait analyser le code et en déduire l'arrêt des imbrications.

On reconnaît un pb du type « arrêt d'une machine de Turing ». Ces situations sont donc intraitables en général.

## **Exemple de limites / difficultés possible**

Moralité :

- Les erreurs de compilation peuvent être difficiles à comprendre, les messages d'erreur peuvent être absents, ou peu clairs.
- Les templates doivent être conçus avec soin, et testés.

# LES MODÈLES

- une pratique déjà connue, exemple illustratif
- **la compilation d'un template**
  - objectif-difficultés
  - **convention d'écriture séparée**
- Spécialisation
- Template pour les classes, exemples

# Contexte de compilation séparée :

## (rappels de principe)

- le code de chaque module est indépendamment transformé en code machine avec des liens ouverts vers le code encore inconnu des méthodes déclarées dans les includes.

(c'est une compilation avec des trous)

- lors de la synthèse finale du programme complet, l'édition de liens termine le couplage de ce qui a été compilé séparément, en redirigeant les appels vers le code machine obtenu précédemment.

# Contexte de compilation séparée :

## (rappels de principe)

```
// f.hpp
#ifndef _F
#define _F
    string f();
#endif
```

```
// f.cpp
#include "f.hpp"
string f() {return "f";}
```

```
// main.cpp
#include "f.hpp"
int main() { cout << f(); }
```

```
// Makefile
CC= g++ -Wall -std=c++11
CCO= $(CC) -c $<
ALL= main.o f.o

all : $(ALL)
    $(CC) -o go $(ALL)

main.o : main.cpp f.hpp
    $(CCO)
f.o : f.cpp f.hpp
    $(CCO)
```

# Contexte de compilation séparée : (rappels de principe)

```
// f.hpp
#ifndef _F
#define _F
    string f();
#endif
```

```
// f.cpp
#include "f.hpp"
string f() {return "f";}
```

```
// main.cpp
#include "f.hpp"
int main() { cout << f(); }
```

```
// Makefile
CC= g++ -Wall -std=c++11
CCO= $(CC) -c $<
ALL= main.o f.o
```

```
all : $(ALL)
    $(CC) -o go $(ALL)

main.o : main.cpp f.hpp
    $(CCO)
f.o : f.cpp f.hpp
    $(CCO)
```

On liste les règles de  
chaque compilation  
séparée



# Contexte de compilation séparée : (rappels de principe)

```
// f.hpp
#ifndef _F
#define _F
    string f();
#endif
```

```
// f.cpp
#include "f.hpp"
string f() {return "f";}
```

```
// main.cpp
#include "f.hpp"
int main() { cout << f(); }
```

```
// Makefile
CC= g++ -Wall -std=c++11
CCO= $(CC) -c $<
ALL= main.o f.o

all : $(ALL)
    $(CC) -o go $(ALL)

main.o : main.cpp f.hpp
    $(CCO)
f.o : f.cpp f.hpp
    $(CCO)
```



La compilation avec  
ses options

# Contexte de compilation séparée : (rappels de principe)

```
// f.hpp
#ifndef _F
#define _F
    string f();
#endif
```

```
// f.cpp
#include "f.hpp"
string f() {return "f";}
```

```
// main.cpp
#include "f.hpp"
int main() { cout << f(); }
```

```
// Makefile
CC= g++ -Wall -std=c++11
CCO= $(CC) -c $<
ALL= main.o f.o
```

```
all : $(ALL)
    $(CC) -o go $(ALL)
```

```
main.o : main.cpp f.hpp
    $(CCO)
f.o : f.cpp f.hpp
    $(CCO)
```

la compilation séparée  
se fait avec d'autres  
options ...

# Contexte de compilation séparée : (rappels de principe)

```
// f.hpp
#ifndef _F
#define _F
    string f();
#endif
```

```
// f.cpp
#include "f.hpp"
string f() {return "f";}
```

```
// main.cpp
#include "f.hpp"
int main() { cout << f(); }
```

```
// Makefile
CC= g++ -Wall -std=c++11
CCO= $(CC) -c $<
ALL= main.o f.o

all : $(ALL)
    $(CC) -o go $(ALL)

main.o : main.cpp f.hpp
    $(CCO)
f.o : f.cpp f.hpp
    $(CCO)
```

on précise les  
dépendances dans les  
règles

# Contexte de compilation séparée : (rappels de principe)

```
// f.hpp
#ifndef _F
#define _F
    string f();
#endif
```

```
// f.cpp
#include "f.hpp"
string f() {return "f";}
```

```
// main.cpp
#include "f.hpp"
int main() { cout << f(); }
```

```
// Makefile
CC= g++ -Wall -std=c++11
CCO= $(CC) -c $<
ALL= main.o f.o
```

```
all : $(ALL)
    $(CC) -o go $(ALL)
```

```
main.o : main.cpp f.hpp
    $(CCO)
f.o : f.cpp f.hpp
    $(CCO)
```

n'oubliez pas les  
dépendances liées aux  
includes

# Contexte de compilation séparée : (rappels de principe)

```
// f.hpp
#ifndef _F
#define _F
    string f();
#endif
```

```
// f.cpp
#include "f.hpp"
string f() {return "f";}
```

```
// main.cpp
#include "f.hpp"
int main() { cout << f(); }
```

```
// Makefile
CC= g++ -Wall -std=c++11
CCO= $(CC) -c $<
ALL= main.o f.o
```

```
all : $(ALL)
    $(CC) -o go $(ALL)
```

```
main.o : main.cpp f.hpp
    $(CCO)
f.o : f.cpp f.hpp
    $(CCO)
```

est remplacé par la  
variable globale

# Contexte de compilation séparée : (rappels de principe)

```
// f.hpp
#ifndef _F
#define _F
    string f();
#endif
```

```
// f.cpp
#include "f.hpp"
string f() {return "f";}
```

```
// main.cpp
#include "f.hpp"
int main() { cout << f(); }
```

```
// Makefile
CC= g++ -Wall -std=c++11
CCO= $(CC) -c $<
ALL= main.o f.o
```

```
all : $(ALL)
    $(CC) -o go $(ALL)

main.o : main.cpp f.hpp
    $(CC) -c $<
f.o : f.cpp f.hpp
    $(CCO)
```

idem

# Contexte de compilation séparée : (rappels de principe)

```
// f.hpp
#ifndef _F
#define _F
    string f();
#endif
```

```
// f.cpp
#include "f.hpp"
string f() {return "f";}
```

```
// main.cpp
#include "f.hpp"
int main() { cout << f(); }
```

```
// Makefile
CC= g++ -Wall -std=c++11
CCO= $(CC) -c $<
ALL= main.o f.o

all : $(ALL)
    $(CC) -o go $(ALL)

main.o : main.cpp f.hpp
    g++ -Wall -std=c++11 -c $<
f.o : f.cpp f.hpp
    $(CCO)
```

ce \$< fait référence à  
la première dépendance  
de la règle courante

# Contexte de compilation séparée : (rappels de principe)

```
// f.hpp
#ifndef _F
#define _F
    string f();
#endif
```

```
// f.cpp
#include "f.hpp"
string f() {return "f";}
```

```
// main.cpp
#include "f.hpp"
int main() { cout << f(); }
```

```
// Makefile
CC= g++ -Wall -std=c++11
CCO= $(CC) -c $<
ALL= main.o f.o
```

```
all : $(ALL)
    $(CC) -o go $(ALL)
```

```
main.o : main.cpp f.hpp
    g++ -Wall -std=c++11 -c main.cpp
f.o : f.cpp f.hpp
    $(CCO)
```

c'est à dire ...



# Contexte de compilation séparée : (rappels de principe)

```
// f.hpp
#ifndef _F
#define _F
    string f();
#endif
```

```
// f.cpp
#include "f.hpp"
string f() {return "f";}
```

```
// main.cpp
#include "f.hpp"
int main() { cout << f(); }
```

```
// Makefile
CC= g++ -Wall -std=c++11
CCO= $(CC) -c $<
ALL= main.o f.o
```

```
all : $(ALL)
    $(CC) -o go $(ALL)
```

```
main.o : main.cpp f.hpp
    g++ -Wall -std=c++11 -c main.cpp
f.o : f.cpp f.hpp
    $(CCO)
```

notez que cette  
compilation est faite  
sans le code de f.cpp

# Contexte de compilation séparée : (rappels de principe)

```
// f.hpp
#ifndef _F
#define _F
    string f();
#endif
```

```
// f.cpp
#include "f.hpp"
string f() {return "f";}
```

```
// main.cpp
#include "f.hpp"
int main() { cout << f(); }
```

```
// Makefile
CC= g++ -Wall -std=c++11
CCO= $(CC) -c $<
ALL= main.o f.o
```

```
all : $(ALL)
    $(CC) -o go $(ALL)
```

```
main.o : main.cpp f.hpp
    g++ -Wall -std=c++11 -c main.cpp
f.o : f.cpp f.hpp
    g++ -Wall -std=c++11 -c f.cpp
```

idem, sauf que dans ce code tout est connu

# Contexte de compilation séparée : (rappels de principe)

```
// f.hpp
#ifndef _F
#define _F
    string f();
#endif
```

```
// f.cpp
#include "f.hpp"
string f() {return "f";}
```

```
// main.cpp
#include "f.hpp"
int main() { cout << f(); }
```

```
// Makefile
CC= g++ -Wall -std=c++11
CCO= $(CC) -c $<
ALL= main.o f.o
```

```
all : $(ALL)
    $(CC) -o go $(ALL)
```

```
main.o : main.cpp f.hpp
    g++ -Wall -std=c++11 -c main.cpp
f.o : f.cpp f.hpp
    g++ -Wall -std=c++11 -c f.cpp
```

les deux règles sont à jours, on passe à l'édition des liens. L'exécutable est produit

# Compilation séparée / cas des modèles :

```
// fichier_1.cpp
#include "pluspetit.hpp"
...
int i{2};
cout << plusPetit(i,i);
```

```
// fichier_2.cpp
#include "pluspetit.hpp"
...
float f{2};
cout << plusPetit(f,f);
```

c'est à la compilation séparée (donc isolée) que l'instanciation de T est faite pour int (pour float).

# Compilation séparée / cas des modèles :

```
// fichier_1.cpp
#include "pluspetit.hpp"
...
int i{2};
cout << plusPetit(i,i);
```

```
// fichier_2.cpp
#include "pluspetit.hpp"
...
float f{2};
cout << plusPetit(f,f);
```

c'est à la compilation séparée (donc isolée) que l'instanciation de T est faite pour int (pour float).

Quid de «pluspetit.cpp» ?

A sa compilation, isolée, on ne peut pas connaître tous les usages de T dans tout le projet...

Et pourtant la création du code machine doit se faire (c++ est non interprété), et être optimisé/spécialisé pour chaque T utilisé.

On comprend que cela **ne peut pas être résolu au niveau de la compilation du fichier** «pluspetit.cpp»

# Compilation séparée / cas des modèles :

```
// fichier_1.cpp
#include "pluspetit.hpp"
...
int i{2};
cout << plusPetit(i,i);
```

```
// fichier_2.cpp
#include "pluspetit.hpp"
...
float f{2};
cout << plusPetit(f,f);
```

On peut imaginer deux approches :

- les instantiations de type sont remontées dans les données du .o, et la compilation des templates est retardée jusqu'au moment de l'édition des liens.
- la compilation est faite exactement au moment de l'instantiation de type.

# Compilation séparée / cas des modèles :

```
// fichier_1.cpp
#include "pluspetit.hpp"
...
int i{2};
cout << plusPetit(i,i);
```

```
// fichier_2.cpp
#include "pluspetit.hpp"
...
float f{2};
cout << plusPetit(f,f);
```

ca voudrait dire « stocker » aussi le code source des modèles dans les .o (pour pouvoir le compiler plus tard) ... bof ...

- les instantiations de type sont remontés dans les données du .o, et la compilation des templates est retardée jusqu'au moment de l'édition des liens.
- la compilation est faite exactement au moment de l'instantiation de type.

# Compilation séparée / cas des modèles :

```
// fichier_1.cpp
#include "pluspetit.hpp"
...
int i{2};
cout << plusPetit(i,i);
```

```
// fichier_2.cpp
#include "pluspetit.hpp"
...
float f{2};
cout << plusPetit(f,f);
```

on ne se contente pas du .hpp du template, on a immédiatement besoin de son .cpp

rq : risque de doublon de code machine, dans fichier\_1.o et dans fichier\_2.o si chacun utilise la même instantiation (ménage à faire)

- les instantiations de type sont dans les données du .o, et la compilation des templates est retardée jusqu'au moment de l'édition des liens.

- la compilation est faite exactement au moment de l'instantiation de type.



# Compilation séparée / cas des modèles :

Solution retenue en c++ :

- la compilation est faite exactement au moment de l'instantiation de type.

# Compilation séparée / cas des modèles :

En pratique :

- déclaration habituelle dans un `template.hpp`
- pas de `template.cpp`
- pas de règles dans Makefile pour sa compilation
- écrire le code dans un `template.tpp`
- include du `.tpp` en fin du `.hpp`

# Compilation séparée / cas des modèles :

```
// tools.hpp
#ifndef _Tool
#define _Tool
    template <typename T> T plusPetit (T a, T b);
    #include "tools.tpp"
#endif
```

```
// tools.tpp
template <typename T> T plusPetit (T a, T b) {
    if (a<b) return a; else return b;
}
```

```
// fichier_1.cpp
#include "tools.hpp"
...
int i{2};
cout << plusPetit(i,i);
```

```
// fichier_2.cpp
#include "tools.hpp"
...
float f{2};
cout << plusPetit(f,f);
```

# Compilation séparée / cas des modèles :

dans les cas simples, on peut tolérer d'écrire le code dans le .hpp

```
// tools.hpp
#ifndef _Tool
#define _Tool
template <typename T> T plusPetit (T a, T b) {
    if (a<b) return a; else return b;
}
#endif
```

```
#include "tools.hpp"
int main() {
    int i{2};
    float f{1.9};
    cout << plusPetit(i,i);
    cout << plusPetit(f,f);
}
```

# LES MODÈLES

- une pratique déjà connue, exemple illustratif
- la compilation d'un template
  - objectif-difficultés
  - convention d'écriture séparée
- **Spécialisation**
- Template pour les classes, exemples

On a déjà vu qu'on pouvait préciser le type utilisé par un template :

```
template <typename T> T plusPetit (T a, T b) {  
    if (a<b) return a; else return b;  
}
```

```
int i1{1}, i2{2};  
float f1{1.5}, f2{1.9};  
string s1{"toto"}, s2{"machin"};  
cout << plusPetit(i1,i2);  
cout << plusPetit(f1,f2);  
cout << plusPetit(s1,s2);  
cout << plusPetit<int>(i2,f2);  
cout << plusPetit<float>(i2,f2);
```

On peut forcer le type. Alors le mécanisme de conversion implicite se met en oeuvre sur les arguments

Du point de vue du vocabulaire, on parle ici d'instanciation de type.

On a déjà vu qu'on pouvait préciser le type utilisé par un template :

```
template <typename T> T plusPetit (T a, T b) {  
    if (a<b) return a; else return b;  
}
```

```
int i1{1}, i2{2};  
float f1{1.5}, f2{1.9};  
string s1{"toto"}, s2{"machin"};  
cout << plusPetit(i1,i2);  
cout << plusPetit(f1,f2);  
cout << plusPetit(s1,s2);  
cout << plusPetit<int>(i2,f2);  
cout << plusPetit<float>(i2,f2);
```

On peut forcer le type. Alors le mécanisme de conversion implicite se met en oeuvre sur les arguments

La spécialisation, c'est l'écriture par le programmeur d'un code pour ce modèle qui sera dédié à un type particulier.

# Exemple :

```
template <typename T> T plusPetit (T a, T b) {  
    if (a<b) return a; else return b;  
}
```

est le patron universel, mais on peut vouloir l'adapter a un cas particulier entre strings.

Ici : `operator<` décide selon l'ordre lexico.

Et si, pour les string, on comparait plutôt les tailles ?

Il faut alors **spécialiser le template** :

```
template<>  
    string plusPetit (string a, string b) {  
        if (a.length()<b.length())  
            return a;  
        else return b;  
    }
```

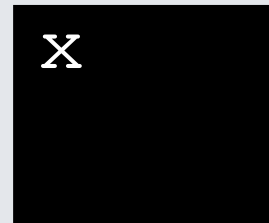


# On vérifie :

```
template <typename T> T plusPetit (T a, T b) {  
    if (a<b) return a; else return b;  
}
```

```
template<> // une spécialisation  
    string plusPetit (string a, string b) {  
        if (a.length()<b.length())  
            return a;  
        else return b;  
}
```

```
int main() {  
    string a{"ab"};  
    string b{"x"};  
    cout << plusPetit(a,b);  
}
```



# sans spécialisation on avait :

```
template <typename T> T plusPetit (T a, T b) {  
    if (a<b) return a; else return b;  
}
```

```
int main() {  
    string a{"ab"};  
    string b{"x"};  
    cout << plusPetit(a,b);  
}
```

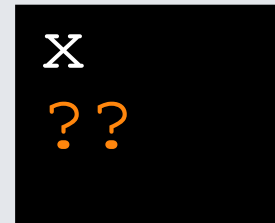
A black square containing the text "ab" in white.

# Question :

```
template <typename T> T plusPetit (T a, T b) {  
    if (a<b) return a; else return b;  
}
```

```
template<>  
    string plusPetit (string a, string b) {  
        if (a.length()<b.length())  
            return a;  
        else return b;  
}
```

```
int main() {  
    string a{"ab"};  
    string b{"x"};  
    cout << plusPetit (a,b);  
    cout << plusPetit ("ab", "x");  
}
```



x  
??

# Question :

```
template <typename T> T plusPetit (T a, T b) {  
    if (a<b) return a; else return b;  
}
```

```
template<>  
    string plusPetit (string a, string b) {  
        if (a.length()<b.length())  
            return a;  
        else return b;  
}
```

```
int main() {  
    string a{"ab"};  
    string b{"x"};  
    cout << plusPetit (a,b);  
    cout << plusPetit ("ab", "x");  
}
```



x  
ab

# Question :

```
template <typename T> T plusPetit (T a, T b) {  
    if (a<b) return a; else return b;  
}
```

```
template<>  
    string plusPetit (string a, string b) {  
        if (a.length()<b.length())  
            return a;  
        else return b;  
    }
```

```
int main() {  
    string a{"ab"};  
    string b{"x"};  
    cout << plusPetit(a,b);  
    cout << plusPetit("ab","x");  
}
```

Le patron est adapté  
pour char\*

x  
ab

# Question :

```
template <typename T> T plusPetit (T a, T b) {  
    if (a<b) return a; else return b;  
}
```

```
template<>  
    string plusPetit (string a, string b) {  
        if (a.length()<b.length())  
            return a;  
        else return b;  
}
```

```
int main() {  
    string a{"ab"};  
    string b{"x"};  
    cout << plusPetit (a,b);  
    cout << plusPetit ("ab", "x");  
}
```

L'opérateur choisi,  
(non membre) est celui  
qui upgrade char\* vers  
string

x  
ab

# Question :

```
template <typename T> T plusPetit (T a, T b) {  
    if (a<b) return a; else return b;  
}
```

```
template<>  
    string plusPetit (string a, string b) {  
        if (a.length()<b.length())  
            return a;  
        else return b;  
    }
```

```
int main() {  
    string a{"ab"};  
    string b{"x"};  
    cout << plusPetit(a,b);  
    cout << plusPetit("ab", "x");  
}
```

on retrouve l'ordre  
lexico



x  
ab

# Et s'il y a spécialisation + surcharge ?

```
template <typename T> T plusPetit (T a, T b) {  
    if (a<b) return a; else return b;  
}
```

```
template<>  
    string plusPetit (string a, string b) {  
        if (a.length()<b.length()) return a;  
        else return b;  
}
```

```
string plusPetit (string a, string b) {  
    cout << "prioritaire";  
    if (a.length()<b.length()) return a;  
    else return b;  
}
```

```
int main() {  
    string a{"x"};  
    plusPetit(a,a);  
}
```

prioritaire



L'existence des templates introduit donc un niveau supplémentaire dans la résolution de la méthode appelée.

L'idée (pour nous) est d'éviter toute spécialisation non vraiment justifiée, pour ne pas trop rentrer dans les détails de cette résolution.

L'algo de résolution cherche :

a) une fonction non template ayant **exactement** le type des arguments fournis

b) s'il n'y en a pas, cherche **exactement** parmi les templates. Où il peut y avoir des spécialisations...

c) si rien n'est trouvé, revient aux fonctions non template, et cherche à upcaster les arguments

# Exemple de bizarreries avec des spécialisations :

```
template<typename U, typename V> void f (U a, V b) {  
    cout << "f général" ;  
}  
template <> void f(int a,int b) {  
    cout << "f pour int et int";  
}  
int main() {  
    f(2,3);  
}
```

f pour int et int

( normal )

# Exemple de bizarreries avec des spécialisations :

```
template<typename U, typename V> void f (U a, V b) {  
    cout << "f général" ;  
}  
template<typename U> void f (U a, U b) {  
    cout << "f pour U et U";  
}  
int main() {  
    f(2,3);  
}
```

f pour U et U

( normal )

# Exemple de bizarreries avec des spécialisations :

```
template<typename U, typename V> void f (U a, V b) {  
    cout << "f général" ;  
}  
template <> void f(int a,int b) {  
    cout << "f pour int et int";  
}  
template<typename U> void f (U a, U b) {  
    cout << "f pour U et U";  
}  
int main() {  
    f(2,3);  
}
```

???

spoil : pas "évident"

# Exemple de bizarreries avec des spécialisations :

```
template<typename U, typename V> void f (U a, V b) {  
    cout << "f général" ;  
}  
template <> void f(int a,int b) {  
    cout << "f pour int et int";  
}  
template<typename U> void f (U a, U b) {  
    cout << "f pour U et U";  
}  
int main() {  
    f(2,3);  
}
```

f pour U et U

spoil : pas "évident"  
mais le compilateur (lui) sait pourquoi ...

# Exemple de bizarreries avec des spécialisations :

```
template<typename U, typename V> void f (U a, V b) {  
    cout << "f général" ;  
}  
template <> void f(int a,int b) {  
    cout << "f pour int et int";  
}  
template<typename U> void f (U a, U b) {  
    cout << "f pour U et U";  
}  
int main() {  
    f(2,3);  
    f<int>(2,3);  
    f<int,int>(2,3);  
}
```

```
f pour U et U  
??  
??
```

spoil : pas "évident"  
mais le compilateur (lui) sait pourquoi ...

# Exemple de bizarreries avec des spécialisations :

```
template<typename U, typename V> void f (U a, V b) {  
    cout << "f général" ;  
}  
template <> void f(int a,int b) {  
    cout << "f pour int et int";  
}  
template<typename U> void f (U a, U b) {  
    cout << "f pour U et U";  
}  
int main() {  
    f(2,3);  
    f<int>(2,3);  
    f<int,int>(2,3);  
}
```

```
f pour U et U  
f pour U et U  
f pour int et int
```

spoil : pas "évident"

mais le compilateur (lui) sait pourquoi ...



A nouveau : l'idée (pour nous) est d'éviter toute spécialisation non vraiment justifiée, pour ne pas rentrer dans les détails de cette résolution.

D'ailleurs, vous pouvez continuer à décliner l'exemple précédent :

- inversez les définitions de  $f(\text{int}, \text{int})$  et  $f(U, U)$
- dupliquez la spécialisation  $f(\text{int}, \text{int})$  avant et après celle de  $f(U, U)$
- constatez les effets sur  $f\langle \text{int} \rangle(2, 3)$

## Exercice (vocabulaire) :

quelle est la nature de la double définition :

```
template<typename U> void f(U a){cout<<"variable";}  
template<typename U> void f(U* a){cout<<"pointeur";}  
int main(){  
    int x{5};  
    f(5);  
    f(&x);  
}
```

f variable  
f pointeur

```
template<typename U> void f(U a){cout<<"variable";}  
// template<typename U> void f(U* a){cout<<"pointeur";}  
int main(){  
    int x{5};  
    f(5);  
    f(&x);  
}
```

f variable  
f variable

# Exercice (vocabulaire) :

## quelle est la nature de la double définition :

```
template<typename U> void f(U a){cout<<"variable";}  
template<typename U> void f(U* a){cout<<"pointeur";}  
int main(){  
    int x{5};  
    f(5);  
    f(&x);  
}
```

f variable  
f pointeur

Le compilateur choisira la plus spécifique qui correspond aux arguments passés.

Ce ne sont pas des spécialisations l'une de l'autre.

C'est une surcharge de fonction : même nom avec des signatures différentes.

Ce sont des templates indépendants.

# LES MODÈLES

- une pratique déjà connue, exemple illustratif
- la compilation d'un template
  - objectif-difficultés
  - convention d'écriture séparée
- Spécialisation
- **Template pour les classes**, exemples

# Template pour les classes

```
template < typename T> class A {  
    public :  
        void f();  
};  
template < typename T> void A<T>::f() {}
```

exemple simple de définition post déclaration  
d'une fonction membre d'une classe template

# Template pour les classes

```
template < typename T> class A {  
    public :  
        static int n;  
};  
template < typename T> int A<T>::n{0}
```

```
int main() {  
    cout << A<int>::n++;  
    cout << A<bool>::n;  
}
```



0  
0

On montre ici que ce sont bien des classes différentes :  
chacune possède son propre attribut statique.

# Template pour les classes

```
template <typename T> class A {  
    public :  
        template <typename U> void f(U u);  
};  
template <typename X>  
template <typename Y>  
void A<X>::f(Y u) {}
```

exemple avec en plus une fonction membre template

puis illustration de l'indépendance des noms utilisés en variables de type

# Template pour les classes

```
template <typename T> class A {  
    public :  
        template <typename U> void f(U u);  
};  
template <typename X,typename Y>  
void A<X>::f(Y u) {}
```

on aurait pu penser à regrouper les variables de type, mais **NON** l'un des templates concerne la signature de f, l'autre celui de la classe.

La bonne forme est vraiment :

```
template <typename X>  
template <typename Y>  
void A<X>::f(Y u) {};
```



# Template pour les classes - visibilité

```
template <typename T> class A {  
    private :  
        T* x;  
    public :  
        template <typename U> void f(U u);  
};  
template <typename T>  
template <typename U>  
void A<T>::f(U u) {cout << u.x; }
```

```
int main() {  
    A<int> a;  
    A<bool> b;  
    a.f(b);  
}
```

'x' is private within this context

normal, à priori T et U sont des types différents

# Template pour les classes - visibilité

```
template <typename T> class A {  
    private :  
        T* x;  
    public :  
        template <typename U> void f(U u);  
};  
template <typename T>  
template <typename U>  
void A<T>::f(U u) {cout << u.x; }
```

```
int main() {  
    A<int> a;  
  
    a.f(a);  
}
```

???

# Template pour les classes - visibilité

```
template <typename T> class A {  
    private :  
        T* x;  
    public :  
        template <typename U> void f(U u);  
};  
template <typename T>  
template <typename U>  
void A<T>::f(U u) {cout << u.x; }
```

```
int main() {  
    A<int> a;  
  
    a.f(a);  
}
```

ok !

à la compilation la fonction  
générée à partir du modèle est :  
void A<int>::f(A<int> u) {u.x; }  
aucun pb de droit

# Template pour les classes - operator<<

```
template <typename T> class A {  
public :  
    T* x;  
};  
template <typename T>  
    ostream& operator<<(ostream &o, const A<T> &a) {  
    o << a.x;  
    return o;  
}
```

```
int main() {  
    A<int> a;  
    cout << a;  
}
```

les fonctions operator<< doivent être redéfinies (et compilées) pour chaque instance de type T utilisée.

Ce sont bien des fonctions templates

# Template pour les classes - operator<<

```
template <typename T> class A {  
private :  
    T* x;  
};  
template <typename T>  
ostream& operator<<(ostream &o, const A<T> &a) {  
    o << a.x;  
    return o;  
}
```

l'introduction de private va compliquer un peu, beaucoup...

```
int main() {  
    A<int> a;  
    cout << a;  
}
```

# Template pour les classes - operator<<

```
template <typename T> class A {  
private :  
    T* x;  
};  
template <typename T>  
ostream& operator<<(ostream &o, const A<T> &a) {  
    o << a.x;  
    return o;  
}
```

cet accès est interdit  
il faut un friend

```
int main() {  
    A<int> a;  
    cout << a;  
}
```

# Template pour les classes - operator<< ?

```
template <typename T> class A {  
private :  
    T* x;  
    friend ostream &operator<< (ostream &, const A<T> &);  
};  
template <typename T>  
    ostream& operator<< (ostream &o, const A<T> &a) {  
    o << a.x;  
    return o;  
}
```

```
int main() {  
    A<int> a;  
    cout << a;  
}
```

compris simplement comme `::operator<<`  
puisque sa version template n'est  
définie qu'ensuite.  
Or on veut que la génération de code  
liée à `T=int` se fasse pour `operator<<`

# Template pour les classes

ce friend est accepté, mais potentiellement trop permissif.

```
template <typename T> class A {
private :
    T* x;
    template <typename X> friend
        ostream &operator<<(ostream &, const A<X> &);
};
template <typename T>
    ostream& operator<<(ostream &o, const A<T> &a) {
        o << a.x;
        return o;
    }
```

```
int main() {
    A<int> a;
    cout << a;
}
```



# Template pour les classes

celui là est interdit :  
il masque le T précédent

```
template <typename T> class A {  
private :  
    T* x;  
    template <typename T> friend  
        ostream &operator<<(ostream &, const A<T> &);  
};  
template <typename T>  
    ostream& operator<<(ostream &o, const A<T> &a) {  
        o << a.x;  
        return o;  
}
```

```
int main() {  
    A<int> a;  
    cout << a;  
}
```

# Template pour les classes - operator<< ?

```
template <typename T> class A {  
private :  
    T* x;  
    friend ostream &operator<< (ostream &, const A<T> &);  
};  
template <typename T>  
ostream& operator<<(ostream &o, const A<T> &a) {  
    o << a.x;  
    return o;  
}
```

```
int main() {  
    A<int> a;  
    cout << a;  
}
```

finalement c'était lui  
le moins pire. Il suffit  
d'essayer de régler le  
pb de la définition  
tardive du template

# Template pour les classes - operator<<

```
template <typename X>
ostream &operator<<(ostream &, const A<X> &);

template <typename T> class A {
private :
    T* x;
    friend ostream &operator<< (ostream &, const A<T> &);
};

template <typename T>
ostream& operator<<(ostream &o, const A<T> &a) {
    o << a.x;
    return o;
}
```

```
int main() {
    A<int> a;
    cout << a;
}
```

Par une déclaration préalable  
Elle dépend de A inconnu

# Template pour les classes - operator<<

```
template <typename X> class A;
template <typename X>
    ostream &operator<<(ostream &, const A<X> &);
template <typename T> class A {
private :
    T* x;
    friend ostream &operator<< (ostream &, const A<T> &);
};
template <typename T>
    ostream& operator<<(ostream &o, const A<T> &a) {
        o << a.x;
        return o;
    }
```

déclaration de A

```
int main() {
    A<int> a;
    cout << a;
}
```

n'est toujours pas reconnu  
comme une fonction template  
(pour des raisons syntaxiques)

# Template pour les classes - operator<<

```
template <typename X> class A;
template <typename X>
    ostream &operator<<(ostream &, const A<X> &);
template <typename T> class A {
private :
    T* x;
    friend ostream &operator<< <T>(ostream &, const A<T> &);
};
template <typename T>
    ostream& operator<<(ostream
        o << a.x;
        return o;
}
```

on clarifie : pour la  
"spécialisation" T

```
int main() {
    A<int> a;
    cout << a;
}
```

Ici, pour operator<< on a finalement 2 solutions :

- Une qui semble trop permissive
  - Une qui nécessite un tuning fin, dans un environnement difficile à maîtriser.
- (le mécanisme de résolution autour des template)

Personnellement, je trouve qu'il y a des choses difficiles à réaliser, et que cela rend raisonnable d'envisager d'accepter la première solution.

## Exercice :

Illustrons que la première solution est trop permissive.

```
template <typename T> class A {  
    private :  
        T* x;  
        template <typename X> friend  
            ostream &operator<<(ostream &, const A<X> &);  
};  
template <typename T>  
    ostream& operator<<(ostream &o, const A<T> &a) {  
        o << a.x;  
        return o;  
}
```

## Exercice :

Illustrons que la première solution est trop permissive.

```
template <typename T> class A {  
    private :  
        T* x;  
    template <typename X> friend  
        ostream &operator<<(ostream &, const A<X> &);  
};  
template <typename T>  
    ostream& operator<<(ostream &o, const A<T> &a) {  
        o << a.x;  
        return o;  
}
```



si ici on avait un A<U>  
on pourrait montrer que l'on  
peut accéder à son x



## Exercice :

Illustrons que la première solution est trop permissive.

```
template <typename T> class A {
private :
    T* x;
    A<bool> * gen() const { return new A<bool>(); }
    template <typename X> friend
        ostream &operator<<(ostream &, const A<X> &);
};
template <typename T>
    ostream& operator<<(ostream &o, const A<T> &a) {
        o << a.x << a.gen() ->x;
        return o;
    }
```

```
int main() {
    A<int> a;
    cout << a;
}
```

on a montré ici que le même  
operator<< est friend avec  
A<int> et A<bool>

## Exercice (défi)

transformez en template la méthode `gen` afin de pouvoir retourner n'importe quel type via, par exemple, l'appel à `gen<string>(); gen<bool>();` etc ...

puis faites le même test que précédemment dans `operator<<`

C'est un « défi », car on se heurte au mécanisme de génération de code des templates qui a parfois du mal à constater les instanciations de type, et à un débog pauvre. Finalement il faudra lui donner un coup de main avec une expression du genre : `a.template gen<string>()` que nous n'avons pas vu.

# LES INTERFACES FONCTIONNELLES

(FONCTEURS + LAMBDA)

On rappelle qu'une interface fonctionnelle est une classe abstraite pure déclarant implémenter une méthode particulière.

```
class F {  
    public :  
        virtual bool f(int x)=0;  
};
```

On rappelle qu'une interface fonctionnelle est une classe abstraite pure déclarant implémenter une méthode particulière.

```
class F {  
    public :  
        virtual bool f(int x)=0;  
};
```

```
class Pair : public F {  
    bool f(int x) {  
        return x%2==0;  
    }  
};
```

On rappelle qu'une interface fonctionnelle est une classe abstraite pure déclarant implémenter une méthode particulière.

```
class F {  
    public :  
        virtual bool f(int x)=0;  
};
```

```
class Pair : public F {  
    bool f(int x) {  
        return x%2==0;  
    }  
};
```

```
void parse(vector<int> v, F &f) {  
    for (int i:v)  
        if (f.f(i)) cout << i;  
}
```

On rappelle qu'une interface fonctionnelle est une classe abstraite pure déclarant implémenter une méthode particulière.

```
class F {  
    public :  
        virtual bool f(int x)=0;  
};
```

```
class Pair : public F {  
    bool f(int x) {  
        return x%2==0;  
    }  
};
```

```
void parse(vector<int> v, F &f) {  
    for (int i:v)  
        if (f.f(i)) cout << i;  
}
```

```
int main() {  
    vector<int> v{1,2,-3,-4,5,6};  
    Pair f;  
    parse(v, f);  
}
```

2 -4 6

c'est une première façon  
de "passer" une fonction  
en argument

On rappelle qu'une interface fonctionnelle est une classe abstraite pure déclarant implémenter une méthode particulière.

```
class F {  
    public :  
        virtual bool f(int x)=0;  
};
```

```
class Pair : public F {  
    bool f(int x) {  
        return x%2==0;  
    }  
};
```

```
void parse(vector<int> v, F &f) {  
    for (int i:v)  
        if (f.f(i)) cout << i,  
}
```

on peut trouver un peu lourd d'avoir à rappeler le nom de la méthode unique

```
int main() {  
    vector<int> v{1,2,-3,-4,5,6};  
    Pair f;  
    parse(v, f);  
}
```

2 -4 6

c'est une première façon de "passer" une fonction en argument



On a vu la semaine dernière la redéfinition d'operator()

```
class F {  
public :  
    virtual bool operator() (int x)=0;  
};
```

```
class Pair : public F {  
    bool operator() (int x) { return x%2==0; }  
};
```

```
void parse(vector<int> v, F &f) {  
    for (int i:v)  
        if (f(i)) cout << i;  
}
```

```
int main() {  
    vector<int> v{1,2,-3,-4,5,6};  
    Pair f;  
    parse(v, f);  
}
```

2 -4 6

# On a vu la semaine dernière la redéfinition d'operator()

```
class F {  
public :  
    virtual bool operator() (int x)=0;  
};
```

est un nom très naturel pour la méthode d'une interface fonctionnelle

```
class Pair : public F {  
    bool operator() (int x) { return x%2==0; }  
};
```

```
void parse(vector<int> v, F &f) {  
    for (int i:v)  
        if (f(i)) cout << i;  
}
```

```
int main() {  
    vector<int> v{1,2,-3,-4,5,6};  
    Pair f;  
    parse(v, f);  
}
```

2 -4 6

Pour aller plus loin, on aimerait pouvoir faire :

```
class F {  
public :  
    virtual bool operator() (int x)=0;  
};
```

```
bool positif(int x) {  
    return x>0;  
}
```

```
void parse(vector<int> v, F &f) {  
    for (int i:v)  
        if (f(i)) cout << i;  
}
```

```
int main() {  
    vector<int> v{1,2,-3,-4,5,6};  
    parse(v,positif); // refusé  
}
```

pouvoir passer  
directement une  
fonction ...

après tout c'est  
la même notation  
...

mais pas de type F  
...

le langage C++ propose des 'pointeurs de fonctions'

```
void parse(vector<int> v, bool (*f) (int)) {  
    for (int i:v)  
        if (f(i)) cout << i;  
}
```

```
int main() {  
    vector<int> v{1,2,-3,-4,5,6};  
    parse(v, positif);  
}
```

```
bool positif(int x) {  
    return x>0;  
}
```

# le langage C++ propose des 'pointeurs de fonctions'

```
class F {  
public :  
    virtual bool operator() (int x)=0;  
};
```

```
void parse(vector<int> v, F &f) {  
    for (int i:v)  
        if (f(i)) cout << i;  
}
```

on peut surcharger  
parse et garder  
les 2 possibilités

```
void parse(vector<int> v, bool (*f) (int)) {  
    for (int i:v)  
        if (f(i)) cout << i;  
}
```

```
int main() {  
    vector<int> v{1,2,-3,-4,5,6};  
    parse(v, positif);  
}
```

```
bool positif(int x) {  
    return x>0;  
}
```

# le langage C++ propose des 'pointeurs de fonctions'

```
class F {  
public :  
    virtual bool operator() (int x)=0;  
};
```

```
void parse(vector<int> v, F &f) {  
    for (int i:v)  
        if (f(i)) cout << i;  
}
```

qu'il faudrait  
améliorer pour  
éviter la  
redondance de code

```
void parse(vector<int> v, bool (*f) (int)) {  
    for (int i:v)  
        if (f(i)) cout << i;  
}
```

```
int main() {  
    vector<int> v{1,2,-3,-4,5,6};  
    parse(v, positif);  
}
```

```
bool positif(int x) {  
    return x>0;  
}
```

# le langage C++ propose des 'pointeurs de fonctions'

```
class F {  
public :  
    virtual bool operator() (int x)=0;  
};
```

```
void parse(vector<int> v, F &f) {  
    for (int i:v)  
        if (f(i)) cout << i;  
}
```

```
void parse(vector<int> v, bool (*f) (int)) {  
    class Loc : public F {  
        bool operator() (int x) { return f(x); }  
    };  
    Loc l;  
    parse(v, l);  
}
```

# le langage C propose des 'pointeurs de fonctions'

```
class F {  
public :  
    virtual bool operator() (int x)  
};
```

```
void parse(vector<int> v, F &f)  
    for (int i:v)  
        if (f(i)) cout << i;  
}
```

```
void parse(vector<int> v, bool (*f)(int)) {  
    class Loc : public F {  
        bool operator() (int x) { return f(x); }  
    };  
    Loc l;  
    parse(v, l);  
}
```

pb syntaxique : une classe locale ne peut dépendre ainsi d'une variable externe (la durée de vie de f est celle de son bloc, à priori différente de celle des objets Loc)



le langage C propose des 'pointeurs de fonctions'

```
class F {  
public :  
    virtual bool operator() (int x)=0;  
};
```

```
void parse(vector<int> v, F &f) {  
    for (int i:v)  
        if (f(i)) cout << i;  
}
```

```
void parse(vector<int> v, bool (*f) (int)) {  
    class Loc : public F {  
        bool (*store) (int) ;  
    public :  
        Loc(bool (*f) (int)) : store{f}{}  
        bool operator() (int x) { return store(x); }  
    };  
    Loc l{f};  
    parse(v,l);  
}
```

on tient une première solution

# on peut éviter l'usage des pointeurs de fonction grâce aux templates

```
template <typename T> void parse(vector<int> v, T f) {  
    for (int i:v) if (f(i)) cout << i ;  
}
```

```
bool positif(int x) { return x>0; }
```

```
class Pair {  
    public :  
        bool operator() (int x) { return x%2==0; }  
};
```

```
int main() {  
    vector<int> v{1,2,-3,-4,5,6};  
    parse(v,positif);  
    Pair f;  
    parse(v,f);  
}
```

# on peut éviter l'usage des pointeurs de fonction grâce aux templates

```
template <typename T> void parse(vector<int> v, T f) {  
    for (int i:v) if (f(i)) cout << i ;  
}
```

```
bool positif(int x) { return x>0; }  
  
class Pair {  
    public :  
        bool operator() (int x) { return x%2==0; }  
};
```

```
int main() {  
    vector<int> v{1,2,-3,-4,5,6};  
    parse(v,positif);  
    Pair f;  
    parse(v,f);  
}
```

la notation fonctionnelle  
est commune aux 2 types  
différents Pair et  
fonction

# on peut éviter l'usage des pointeurs de fonction grâce aux templates

```
template <typename T> void parse(vector<int> v, T f) {  
    for (int i:v) if (f(i)) cout << i ;  
}
```

```
bool positif(int x) { return x>0;}  
  
class Pair {  
    public :  
        bool operator()(int x) { return x%2==0;}  
};
```

```
int main() {  
    vector<int> v{1,2,-3,-4,5,6};  
    parse(v,positif);  
    Pair f;  
    parse(v,f);  
}
```

Les deux instantiations produiront le code compilé pour deux parse : un pour Pair, un pour int (\*) int

# on peut éviter l'usage des pointeurs de fonction grâce aux templates

```
template <typename T> void parse(vector<int> v, T f) {  
    for (int i:v) if (f(i)) cout << i ;  
}
```

C'est ce genre de solution qui est retenu dans la STL  
Standard **Template** Library

Elle ouvre en particulier la voie au traitement homogène des lambda-expressions, qui sont encore un autre "type d'interface fonctionnelle".

Elles peuvent être capturées par la solution "template" puisque le type exact importe peu pour les template.

# on peut éviter l'usage des pointeurs de fonction grâce aux templates

```
template <typename T> void parse(vector<int> v, T f) {  
    for (int i:v) if (f(i)) cout << i ;  
}
```

C'est ce genre de solution qui est retenu dans la STL  
Standard **Template** Library

Elle ouvre en particulier la voie au traitement homogène des lambda-expressions, qui sont encore un autre "type d'interface fonctionnelle".

Elles peuvent être capturées par la solution "template" puisque le type exact importe peu pour les template.

```
int main() {  
    vector<int> v{1,2,-3,-4,5,6};  
    parse(v, [](int a) { return (a%2==1); });  
}
```

1 -3 5

# Syntaxe des lambda-expressions

```
[ ] (int a) { return (a%2==1); }
```

- [ ] est un bloc de capture des variables d'environnement  
ex : [x, &y ]

- (int a, string &b) contient une liste de paramètres

- un corps d'instructions qui produisent un résultat

Rq : lorsque le return est assez clair, le type retour est déduit de l'expression. Sinon il faut le préciser :

```
[ ] (int a) -> bool { return (a%2==1); }
```

Pour mieux comprendre le rôle de ces parties, prenons un exemple :

# Compréhension des lambda-expressions

```
void print_divisible( const vector<int >& v, ostream &o, int m) {  
    for_each( v.begin(), v.end(),  
              [&o, m] (int x) {if (x%m==0) o<< x; }  
    );  
}
```

L'idée est d'afficher les éléments de `v` qui sont divisibles par `m`

- `v.begin()` et `v.end()` sont des itérateurs sur le vecteur `v`
- `for_each()` est définie dans `<algorithm>` de la STL :  
elle applique, à chaque élément situé entre les deux itérateurs, une fonction, donnée ici sous la forme d'une lambda-expression
- les variables `o` et `m` font partie de l'environnement
- ici `o` est capturée par référence, et `m` par valeur
- le paramètre `x` est associé par valeur au contenu de l'itérateur



# Compréhension des lambda-expressions

```
[&o, m] (int x) {if (x%m==0) o<< x; }
```

Peut être vue comme :

```
class MyLambda{  
private :  
    ostream & _o;  
    int _m;  
public :  
    MyLambda(ostream &o, int m) : _o{o}, _m{m} {}  
    void operator() (int x) const {  
        if (x%m==0) o<< x;  
    }  
};
```

# Compréhension des lambda-expressions

```
[&o, m] (int x) {if (x%m==0) o<< x; }
```

Peut être vue comme :

```
class MyLambda{  
private :  
    ostream & _o;  
    int _m;  
public :  
    MyLambda(ostream &o, int m) : _o{o}, _m{m} {}  
    void operator() (int x) const {  
        if (x%m==0) o<< x;  
    }  
};
```

les variables capturées sont  
stockées dans l'objet associé à  
la lambda.  
Initialisées à la construction

# Compréhension des lambda-expressions

```
[&o, m] (int x) {if (x%m==0) o<< x; }
```

Peut être vue comme :

```
class MyLambda{  
private :  
    ostream & _o;  
    int _m;  
public :  
    MyLambda(ostream &o, int m) : _o{o}, _m{m} {}  
    void operator() (int x) const {  
        if (x%m==0) o<< x;  
    }  
};
```

les variables capturées sont stockées dans l'objet associé à la lambda.  
Initialisées à la construction

un operator() const est associé.  
Il correspond à la nature des paramètres

# Compréhension des lambda-expressions

```
[&o, m] (int x) {if (x%m==0) o<<< x; }
```

Peut être vue comme :

```
class MyLambda{  
private :  
    ostream & _o;  
    int _m;  
public :  
    MyLambda(ostream &o, int m) : _o{o}, _m{m} {}  
    void operator() (int x) const {  
        if (x%m==0) o<<< x;  
    }  
};
```

les variables capturées sont stockées dans l'objet associé à la lambda.  
Initialisées à la construction

un operator() const est associé.  
Il correspond à la nature des paramètres

son corps est donné par le bloc de la lambda exp.

# Compréhension des lambda-expressions

```
[&o, m] (int x) -> void {if (x%m==0) o<< x; }
```

Peut être vue comme :

```
class MyLambda{  
    private :  
        ostream & _o;  
        int _m;  
    public :  
        MyLambda(ostream &o, int m) : _o{o}, _m{m} {}  
        void operator() (int x) const {  
            if (x%m==0) o<< x;  
        }  
};
```

Rq : lorsque c'est assez clair, le type retour est déduit de l'expression.  
Sinon il faut le préciser (ici c'était facultatif)

# Compréhension des lambda-expressions

```
[m] () mutable -> void {m++;}
```

Peut être vue comme :

```
class MyLambda{  
    private :  
        int _m;  
    public :  
        MyLambda(int m) : _m{m} {}  
        void operator() () const {  
            _m++;  
        }  
};
```

Rq : si on souhaite que operator() puisse modifier l'objet lambda, on peut le déclarer mutable. (Exemple trivial ici)

# Compréhension des lambda-expressions

Le type exact des lambda est non-spécifié.

Chaque expression a son propre type.

Si on veut stocker une lambda avec son type réel on doit utiliser `auto` (et c'est **le seul endroit où on tolère vraiment `auto`** dans ce cours)

```
auto pair = [] (int x) {y++; return x%2==0;};  
auto impair = [] (int x) {return x%2==1;};  
auto pos = [] (int x) {return x>0;};
```

# Compréhension des lambda-expressions

Le type exact des lambda est non-spécifié.

Chaque expression a son propre type.

Si on veut stocker une lambda avec son type réel on doit utiliser `auto` (et c'est **le seul endroit où on tolère vraiment `auto`** dans ce cours)

```
auto pair = [] (int x) {y++; return x%2==0;};  
auto impair = [] (int x) {return x%2==1;};  
auto pos = [] (int x) {return x>0;};
```

```
pair=pos; // interdit !
```

rq : `auto` n'est pas un type uniforme pour les lambdas.  
Il y a une inférence de type



# Compréhension des lambda-expressions

Le type exact des lambda est non-spécifié.

Chaque expression a son propre type.

un type compatible uniforme aux lambda est `function` :

```
#include <functional>
...
auto pair = [] (int x) {y++; return x%2==0;};
auto impair = [] (int x) {return x%2==1;};
auto pos = [] (int x) {return x>0;};

function<bool(int)> f {pair};
f=impair;
```

# Compréhension des lambda-expressions

Le type exact des lambda est non-spécifié.

Chaque expression a son propre type.

un type compatible uniforme aux **lambda et foncteurs** est `function` :

```
#include <functional>
...
auto pair = [] (int x) {y++; return x%2==0;};
auto impair = [] (int x) {return x%2==1;};
auto pos = [] (int x) {return x>0;};

function<bool(int)> f {pair};
class Neg {
public :
    bool operator()(int x) { return x < 0;}
};
f = Neg(); // convient aussi pour les foncteurs
```

# Compréhension des lambda-expressions

Le type exact des lambda est non-spécifié.

Chaque expression a son propre type.

un type compatible uniforme aux **lambda et foncteurs** est `function` :

```
#include <functional>
...
auto pair = [] (int x) {y++; return x%2==0;};
auto impair = [] (int x) {return x%2==1;};
auto pos = [] (int x) {return x>0;};

function<bool(int)> f {pair};
class Neg {
public :
    bool operator()(int x) { return x < 0;}
};
f = Neg(); // convient aussi pour les foncteurs

vector <function<bool(int)>> tab_f{pair, impair, pos, f};
```

# Compréhension des lambda-expressions

Définir des lambda-templates n'était pas d'actualité pour c++11

```
auto print = [] (auto x) {cout << x}; // à partir de c++14  
[]<typename T>(T a,T b) {return a + b;} // à partir de c++20
```

(pas au programme pour nous)

Exercice : qu'affiche le code suivant ?

```
int main() {  
    auto f = [] (int x) {  
        return [] (int y) { return y * 2; } (x) + 3;  
    };  
    cout << f(5) << endl;  
}
```

Exercice : qu'affiche le code suivant ?

```
int main() {  
    auto f = [] (int x) {  
        return [] (int y) { return y * 2; } (x) + 3;  
    };  
    cout << f(5) << endl;  
}
```

13

On reconnaît d'abord l'appel `f(5)` qui rentre dans le corps de la première lambda.

Le `return` définit une lambda, et l'applique aussitôt à `x` (qui vaut 5). Cette seconde lambda retourne 10, auquel on ajoute 3

Exercice : qu'affiche le code suivant ?

```
int main() {
    auto f = [] (int x) {
        return [x] (int y) { return x + y; };
    };
    auto g = [] (const function<int (int)>& f, int z) {
        return f(z) * 2;
    };
    cout << g(f(7), 8) << endl;
}
```

Exercice : qu'affiche le code suivant ?

```
int main() {  
    auto f = [] (int x) {  
        return [x] (int y) { return x + y; };  
    };  
    auto g = [] (const function<int (int)>& f, int z) {  
        return f(z) * 2;  
    };  
    cout << g(f(7), 8) << endl;  
}
```

30

f retourne une lambda et f(7) est la fonction  $y \rightarrow y+7$   
(remarquez la capture de x)

g applique cette fonction à 8, ce qui donne 15, puis multiplie par 2



Exercice :

Ecrivez une fonction template `print_all`, qui s'applique à un vector générique. Elle utilise ensuite `for_each` pour le parcourir en appliquant une lambda à tous ces éléments. Cette lambda capture un entier préalablement déclaré dans la fonction, initialisé à 0. Elle affiche chaque élément et les compte en même temps en utilisant cet entier.

Avant de sortir de la fonction `print_all`, elle affiche la taille calculée par la lambda.

## Exercice :

Ecrivez une fonction template `print_all`, qui s'applique à un vector générique. Elle utilise ensuite `for_each` pour le parcourir en appliquant une lambda à tous ces éléments. Cette lambda capture un entier préalablement déclaré dans la fonction, initialisé à 0. Elle affiche chaque élément et les compte en même temps en utilisant cet entier.

Avant de sortir de la fonction `print_all`, elle affiche la taille calculée par la lambda.

```
vector<int> vi={1,2,3,4,5};  
vector<string> vs={"un","deux"};  
print_all(vi);  
print_all(vs);
```

```
1 2 3 4 5  
5 elements  
un deux  
2 elements
```

## Exercice :

Ecrivez une fonction template `print_all`, qui s'applique à un vector générique. Elle utilise ensuite `for_each` pour le parcourir en appliquant une lambda à tous ces éléments. Cette lambda capture un entier préalablement déclaré dans la fonction, initialisé à 0. Elle affiche chaque élément et les compte en même temps en utilisant cet entier.

Avant de sortir de la fonction `print_all`, elle affiche la taille calculée par la lambda.

```
template <typename T>
void print_all(const vector<T>& v) {
    int nb=0;
    for_each(v.begin(), v.end(),
             [&nb](const T& n) {nb++; cout << n;});
    cout << nb << " elements" ;
}
```

```
vector<int> vi={1,2,3,4,5};
vector<string> vs={"un","deux"};
print_all(vi);
print_all(vs);
```

```
1 2 3 4 5
5 elements
un deux
2 elements
```