ECE PARIS
ÉCOLE D'INGÉNIEURS

Robin FERCOQ
2018-2019

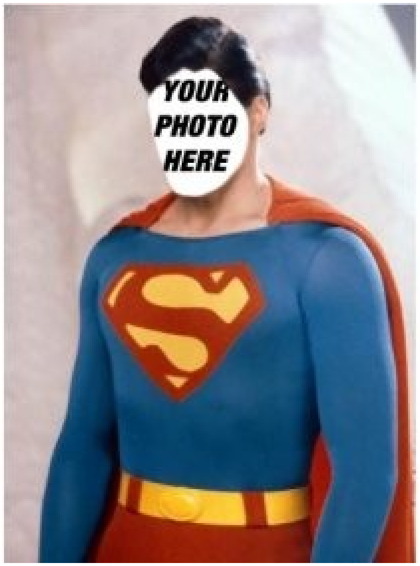
Conception et Programmation Orientée Objet C++

POO - C++

Sommaire général du semestre

COURS	Semaine suivante →	**TPs**
1. *Intro, concepts, 1 exemple*		1. *Organisation objet des données*
2. *Modélisation objet / UML*		2. *Diagrammes de classe UML*
3. *C++ pratique 1*		3. *C++ pratique, E/S, string, vector*
4. *C++ pratique 2*		4. *C++ pratique, type &, surcharge*
5. *Classes & C++ : bases*		5. *Date : une classe simple en C++*
6. *Classes & C++ : compléments*		6. *UML et C++, associations*
7. *Conteneurs & C++ : la STL*		7. *Gestion de collections complexes*
8. *Héritage / polymorphisme*		8. *Collections polymorphes*
9. *Abstraction / design patterns*		9. *Modèle composite et graphismes*
10. *Exceptions, flots, fichiers ...*		10. *Persistance / fichiers / except.*
11. Templates côté développeur		11. *Développement de templates*
12. *Gestion mémoire / smarts ptrs*		12. *Soutenance de **projet** ...*

Templates côté développeur



COURS 11

A) Fonctions/méthodes inline
B) Programmation générique
C) Templates de fonctions
D) Templates de classes
E) Spécialisation de templates
F) Paramètres fonctions

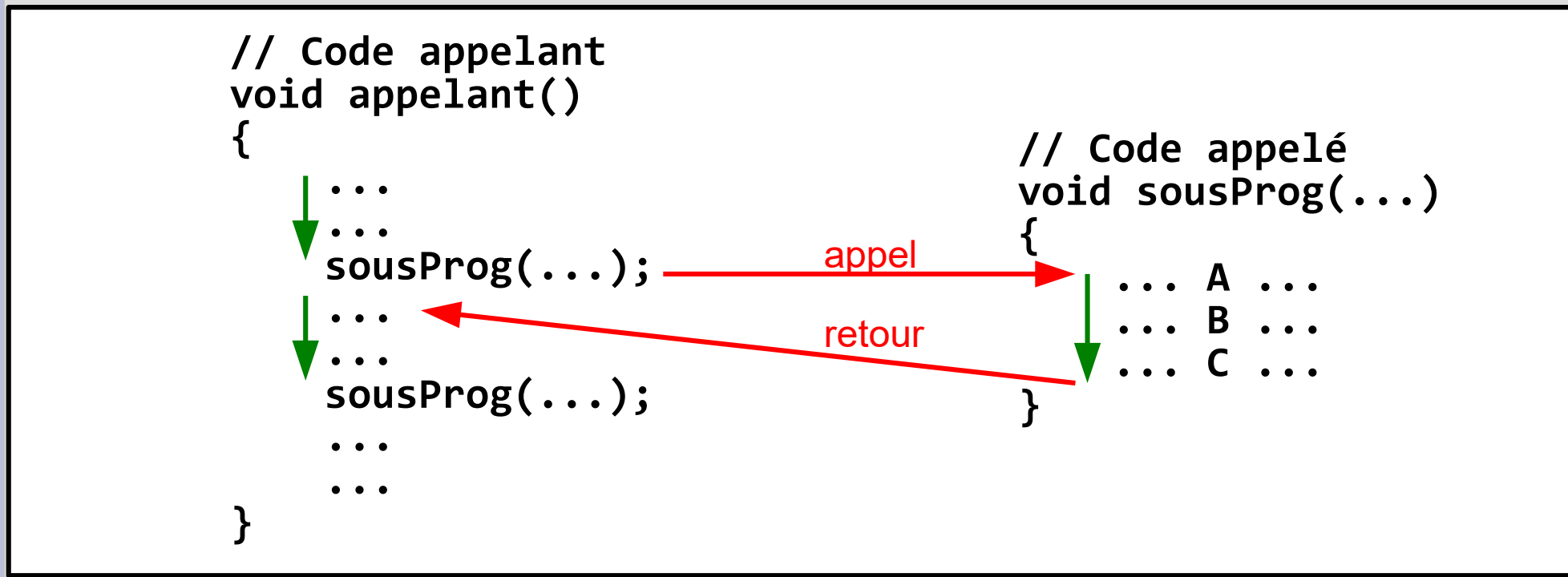
COURS 11

A) Fonctions/méthodes inline
B) Programmation générique
C) Templates de fonctions
D) Templates de classes
E) Spécialisation de templates
F) Paramètres fonctions

Fonctions/méthodes inline

Fonctions/méthodes inline

- *appel de sous-programme = mécanisme complexe :*
  - *le processeur « met en attente » la séquence actuelle*
  - *le processeur exécute la séquence sous-programme*
  - *le processeur poursuit la séquence après l'appel*

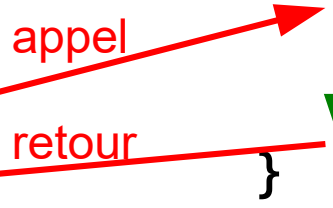


Fonctions/méthodes inline

- *appel de sous-programme = mécanisme complexe :*
  - *le processeur « met en attente » la séquence actuelle*
  - *le processeur exécute la séquence sous-programme*
  - *le processeur poursuit la séquence après l'appel*
  - *un 2ème appel implique le même travail du processeur*

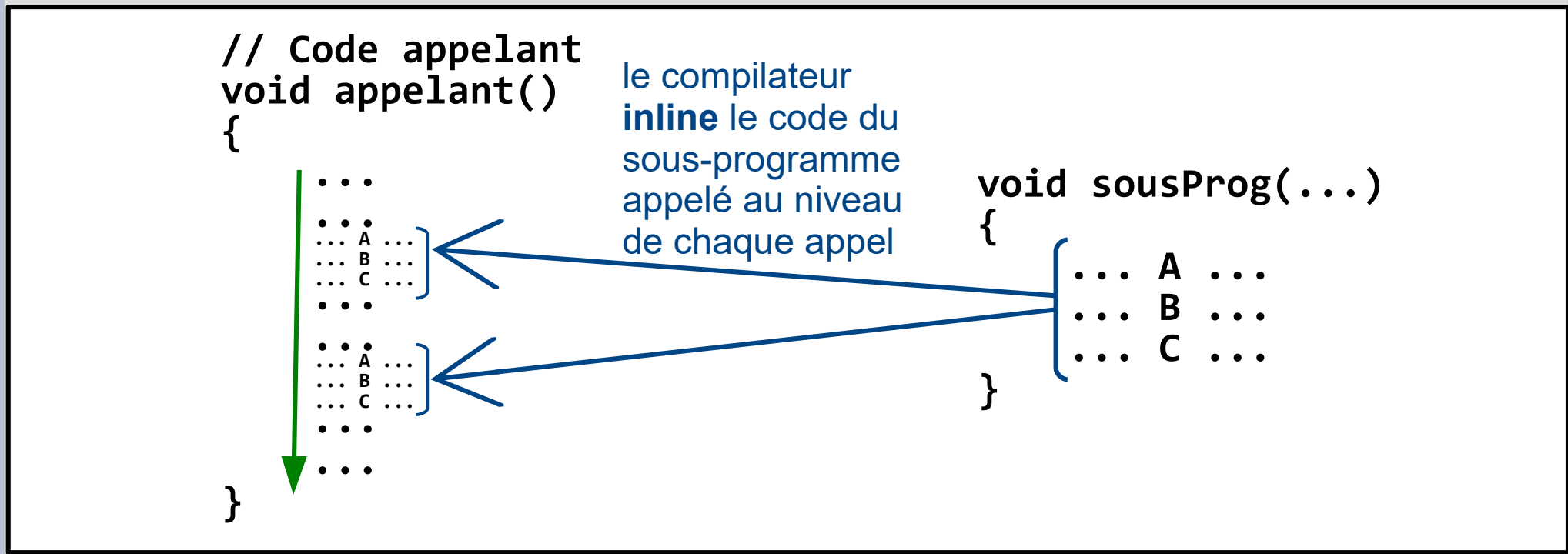
```
// Code appelant
void appelant()
{
    ...
    ...
    sousProg(...);
    ...
    ...
    sousProg(...);
    ...
    ...
}
```

```
// Code appelé
void sousProg(...)
{
    ... A ...
    ... B ...
    ... C ...
}
```



Fonctions/méthodes inline

- *Pour optimiser le compilateur peut décider de mettre **inline** le sous-programme*
- *C'est comme si le code du sous-programme était écrit directement au niveau de l'appel*
- *Le processeur économise les temps des allers-retours mais l'exécutable devient plus lourd (code bloat)*



Fonctions/méthodes inline

- *C'est donc un **compromis** entre la vitesse d'exécution et la taille de l'exécutable*
- *Traditionnellement (il y a 15 ans ou plus) le choix d'inliner ou pas une fonction au niveau de ses appels était indiqué par le développeur en la déclarant* **`inline`**
- *Désormais c'est le compilateur qui décide d'inliner ou pas un appel selon des critères d'optimisation du code machine généré (il est mieux placé que nous pour savoir!)*
- *La déclaration* `inline` *continue d'être utile car elle permet de définir des fonctions en même temps que leur déclaration : dans un en-tête* **`.h`** *ou* **`.hpp`**
- *On n'a alors plus besoin d'implémenter séparément la fonction dans un .cpp*

Fonctions/méthodes inline

- *Une fonction déclarée inline est implémentée dans le .h*

utile.h

```
double carre(double x);
```

Déclaration

utile.h

```
inline double carre(double x)
{
    return x*x;
}
```

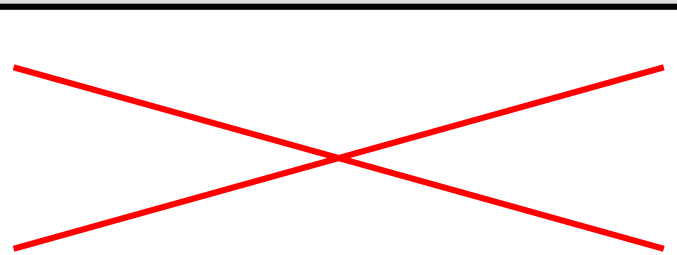
Déclaration & implémentation

utile.cpp

```
double carre(double x)
{
    return x*x;
}
```

Implémentation

utile.cpp



ici.cpp

```
#include "utile.h"
void utiliserIci()
{
    std::cout<<carre(3)<<std::endl;
}
```

ici.cpp

```
#include "utile.h"
void utiliserIci()
{
    std::cout<<carre(3)<<std::endl;
}
```

ailleurs.cpp

```
#include "utile.h"
void utiliserAilleurs()
{
    std::cout<<carre(4)<<std::endl;
}
```

ailleurs.cpp

```
#include "utile.h"
void utiliserAilleurs()
{
    std::cout<<carre(4)<<std::endl;
}
```

Fonctions/méthodes inline

- *Une fonction déclarée inline est implémentée dans le .h*

```
inline double carre(double x)
{
    return x*x;
}
```

utile.h

Déclaration & implémentation

- *Ceci rend possible des bibliothèques sans .cpp donc sans fichiers objets à linker : header-only libraries*
- *Beaucoup plus simples à **utiliser**, il suffit de copier les fichiers .h en local dans les répertoires include du compilateur et de faire #include <bibliotheque.h>*
- *Inconvénient principal : temps de compilation plus long*

Fonctions/méthodes inline

- *Une fonction déclarée inline est implémentée dans le .h*

utile.h

```
inline double carre(double x)
{
    return x*x;
}
```

Déclaration & implémentation

- *On peut être tenté d'écrire inline nos applications pour **se débarrasser des .cpp** et n'avoir que des .h*
- ***C'est déconseillé en pratique***
  - ➔ *Mauvaise séparation interface / implémentation*
  - ➔ *Compilation longue (mauvais pour développer !)*
  - ➔ *Pas adapté aux grosses fonctions...*

Fonctions/méthodes inline

hero.h

```
class Hero
{
    public :
        Hero(std::string realName, std::string heroName);
        std::string getHeroName() const;
        std::string getMission() const;
        void setMission(std::string mission);

    private :
        std::string m_realName;
        std::string m_heroName;
        std::string m_mission;
};

inline Hero::Hero(std::string realName, std::string heroName)
    : m_realName{realName}, m_heroName{heroName}
{ }

inline std::string Hero::getHeroName() const
{
    return m_heroName;
}

inline std::string Hero::getMission() const
{
    return m_mission;
}

inline void Hero::setMission(std::string mission)
{
    m_mission = mission;
}
```

méthodes déclarées normalement

*méthodes **inline** implémentées dans le .h*

Fonctions/méthodes inline

- *Quand on définit directement une méthode dans la déclaration de classe elle est automatiquement inline !*

hero.h

```
class Hero
{
    public :
        Hero(std::string realName, std::string heroName)
            : m_realName{realName}, m_heroName{heroName} { }

        std::string getHeroName() const
        {   return m_heroName;  }

        std::string getMission() const
        {   return m_mission;  }

        void setMission(std::string mission)
        {   m_mission = mission;   }

    private :
        std::string m_realName;
        std::string m_heroName;
        std::string m_mission;
};
```

méthodes inline implicites !

Fonctions/méthodes inline

- *Non obligatoire...* ***A réserver aux méthodes courtes et simples : pas plus de 10 lignes*** *(Google style guide)*

hero.h

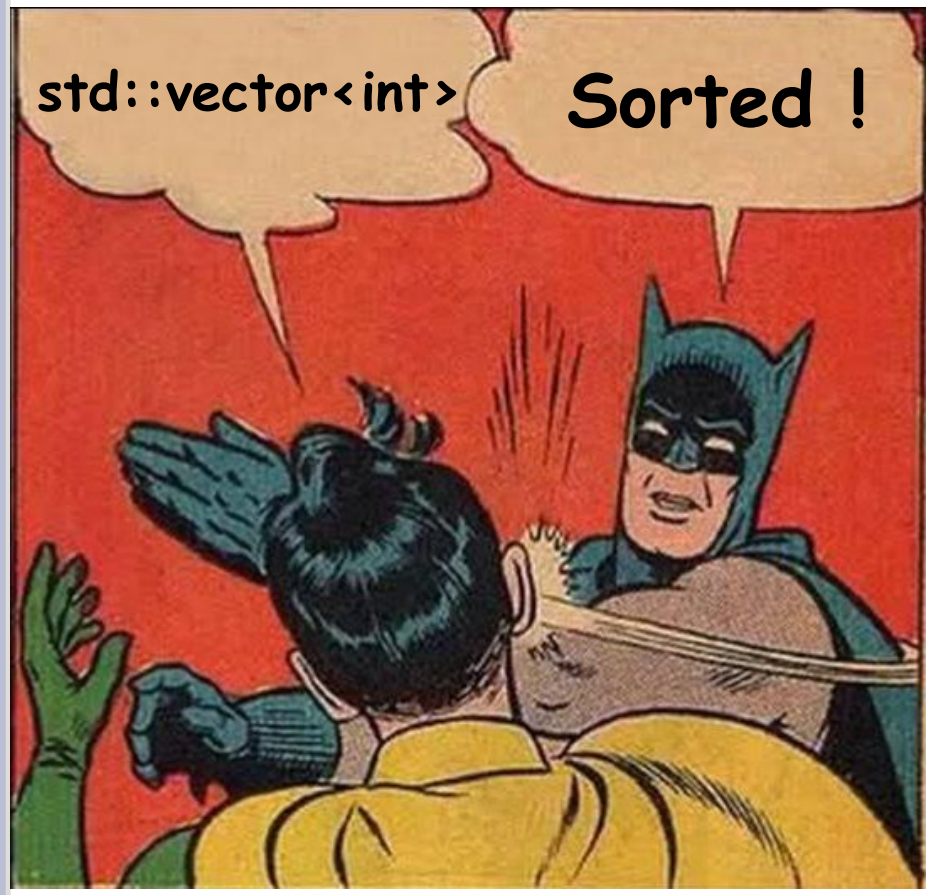
```
class Hero
{
    public :
        Hero(std::string realName, std::string heroName)
            : m_realName{realName}, m_heroName{heroName} { }

        std::string getHeroName() const
        {   return m_heroName;  }

        std::string getMission() const
        {   return m_mission;  }

        void setMission(std::string mission)
        {   m_mission = mission;   }

    private :
        std::string m_realName;
        std::string m_heroName;
        std::string m_mission;
};
```

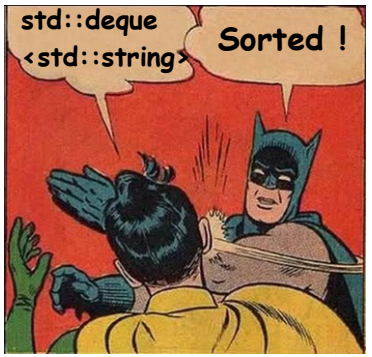
méthodes inline implicites !

COURS 11

A) Fonctions/méthodes inline
B) Programmation générique
C) Templates de fonctions
D) Templates de classes
E) Spécialisation de templates
F) Paramètres fonctions

Programmation générique





Quelles que soient les idées confuses de Robin,
Batman a toujours le même algorithme de tri !

Programmation générique

- *Un **algorithme** est un ensemble de traitements qui à partir de données initiales fourni des données résultats*
- *Une **structure de données** est une façon d'organiser des données en mémoire pour les stocker / retrouver...*
- *Très souvent l'algorithme ou la structure de données ne dépendent que d'hypothèses très limitées sur les opérations possibles avec les données*
- *Par exemple il suffit que les données définissent a < b*

  ➔*pour pouvoir appliquer un algorithme de tri*

  ➔*pour pouvoir les stocker dans un arbre binaire de recherche (comme le conteneur set)*

Programmation générique

- *Exemple simple : trier 2 données*

  Algorithme

  trier(a par référence, b par référence)

  ```
  |   Si pas(a < b) Alors
  |   |   tmp ← a
  |   |   a ← b
  |   |   b ← tmp
  |
  ```

- *Cet algorithme pourrait aussi bien traiter des entiers que des flottants que des caractères que des chaînes (le type string défini bien l'opérateur <) ...*
- ***Problème** : le C++ est un langage typé, il va falloir dupliquer un même code pour chaque type !*

Programmation générique

- ***Problème : le C++ est un langage typé, il va falloir dupliquer un « même code » pour chaque type !***

```
void trier(char& a, char& b)
{
    if ( !(a<b) )
    {
        char tmp = a;
        a = b;
        b = tmp;
    }
}

void trier(int& a, int& b)
{
    if ( !(a<b) )
    {
        int tmp = a;
        a = b;
        b = tmp;
    }
}

void trier(std::string& a, std::string& b)
{
    if ( !(a<b) )
    {
        std::string tmp = a;
        a = b;
        b = tmp;
    }
}
```

presque pareil

presque pareil

presque pareil

Le problème se pose principalement pour du code de niveau bibliothèque

```
char
unsigned char
short int
unsigned short int
int
unsigned int
float
double
long double
std::string
...

+ AnyPossibleCustomType...
  ( impossible )
```

Programmation générique

- ***Solution 0 : le préprocesseur, les macros***
- *En **C** la façon de faire consiste à demander au préprocesseur (1ère passe du compilateur) de **substituer littéralement** le code source « appelant » par un bloc de code avec paramètres (macro)*
- *Très limité, très artisanal...*
- *Incompatible en général avec des types objets (string)*
- *Hors sujet en C++ moderne*

```
#define SWAP(a, b) do { a ^= b; b ^= a; a ^= b; } while ( 0 )

#define SORT(a, b) do { if ((a) > (b)) SWAP((a), (b)); } while (0)
```

source

Programmation générique

- ***Solution 1 : renoncer aux types***
- *Certains langages ont un typage « dynamique » permissif ou pas de typage explicite du tout...*

➔ *Exemple en **JavaScript** (≠ Java)*

code javascript

```
function trier(a, b)
{
    if ( a < b )
        return [a, b];
    else
        return [b, a];
}

console.log( trier( "world", "hello" ) );
console.log( trier( 7, 5 ) );
console.log( trier( "world", 5 ) );?
```

On verra les types au moment de l'appel !

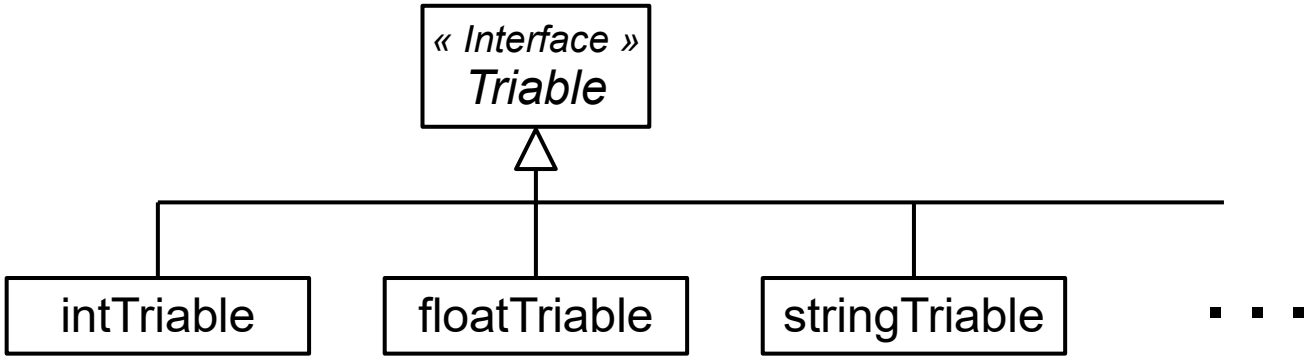
```
[ "hello", "world" ]
[ 5, 7 ]
[ 5, "world" ]???
```

➔ *Souple et simple : trop cool !* *Oui mais, **trop** cool...*

➔ *Pas de compilation => bugs au runtime + perfs ↘*

Programmation générique

- ***Solution 2 : le polymorphisme dynamique***




```
void trier(Triable& a, Triable& b)
{
    if ( !a.lessThan(b) )
    {
        Triable tmp = a.clone();
        a = b;
        b = tmp;
    }
}
```

Ici ça ne marche même pas !

- *Pas assez en commun pour partager logiquement une hiérarchie*
- *Gros problèmes de cohérence : 2 fois même type concret ?*
- *Variable tmp par valeur => slicing (perte du polymorphisme)*
- *Pas adapté aux types primitifs int / float / char ...*

Programmation générique

- ***Solution 3 : le polymorphisme statique***
- *Le langage C++ va proposer un mécanisme de* ***programmation générique*** *par* ***typage paramétrable***
- *Un code qui est paramétrable en type sera appelé* ***un template*** *(en français : patron)*
- *Peut s'appliquer aux fonctions (sous-programmes)*
- *Peut s'appliquer aux classes*
- *On a déjà rencontré des classes templates :*
  les conteneurs de la STL, Standard Template Library
- *Ainsi dans* `std::vector<int>` *le* `int` *entre chevrons est le paramètre de type d'une classe template* `std::vector< >`

Programmation générique

- ***Solution 3 : le polymorphisme statique***
- *On dit que c'est du polymorphisme parce que un même traitement (même code) va s'appliquer à des types concrets distincts (opérateurs spécifiques)*
- *On dit que c'est statique parce que la cohérence du type est déterminée à la compilation et non pas au runtime, d'où les avantages suivants :*
  - *Les erreurs sont détectées/signalées à la compilation*
  - *Pas de RTTI qui coûte des octets à chaque objet*
  - *Optimisable par le compilo pour chaque type concret*
  - *Compatible avec les types élémentaires (int, float...)*
  - *Pas besoin de classe de base en commun*

Programmation générique

- ***Solution 3 : le polymorphisme statique***

  Inconvénients :

  - *Par rapport au polymorphisme dynamique on ne peut pas mélanger des types distincts => homogénéité*
  - *On doit connaître à l'avance « en dur » les types*
  - *Pour chaque type concret utilisé le compilateur génère un ensemble de code dédié => code bloat (gros execs)*
  - *La beauté syntaxique des déclarations est... discutable*
  - *Utilise de la déduction automatique de type qui marche bien en général, mais pas toujours => surprises*
  - *À haut niveau la méta-programmation C++ générique par templates est notoirement illisible et compliquée*

Programmation générique

- *La définition d'un bloc de code templaté (définition d'une fonction ou classe template) commence par une déclaration de template :*

```
template<typename T>
/// Dans le code qui suit on utilise T comme un type
/// Définition d'une fonction ou d'une classe...
...
...
...
/// Fin de la fonction ou de la classe
après la fermeture de la définition on n'est plus dans le template
```

- ***IMPORTANT** : un code templaté est automatiquement (implicitement) **inline***
- *Il doit toujours être inclus dans le fichier où on veut l'utiliser : à moins de faire du test directement devant le main (possible) il va toujours dans un en-tête* **.h**

Programmation générique

- *La définition d'un bloc de code templaté (définition d'une fonction ou classe template) commence par une déclaration de template :*

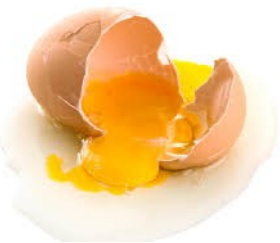
```
template<typename T, typename U, typename V ...>
/// Dans le code qui suit on utilise T, U, V... comme des types
/// Définition d'une fonction ou d'une classe...
...
...
...
/// Fin de la fonction ou de la classe
après la fermeture de la définition on n'est plus dans le template
```

- ***IMPORTANT** : un code templaté est automatiquement (implicitement) **inline***
- *Il doit toujours être inclus dans le fichier où on veut l'utiliser : à moins de faire du test directement devant le main (possible) **il va toujours dans un en-tête** `.h`*

COURS 11

A) Fonctions/méthodes inline
B) Programmation générique
C) Templates de fonctions
D) Templates de classes
E) Spécialisation de templates
F) Paramètres fonctions

Templates de fonctions



Templates de fonctions

- *Voici finalement la fonction générique de tri*

utile.h

```
template<typename T>
void trier(T& a, T& b)
{
    if ( !(a<b) )
    {
        T tmp = a;
        a = b;
        b = tmp;
    }
}
```

- Le type concret T est (si possible) déduit de l'appel à partir du type des paramètres utilisés
- A chaque appel un nouveau type T est déduit
- Le compilateur génère autant de versions concrètes (instances du template) qu'il y a de types utilisés

Templates de fonctions

- *Exemple instanciation pour un type char*

utile.h

```
template<typename T>
void trier(T& a, T& b)
{
    if ( !(a<b) )
    {
        T tmp = a;
        a = b;
        b = tmp;
    }
}
```

code concret automatiquement généré par le compilateur

```
void trier(char& a, char& b)
{
    if ( !(a<b) )
    {
        char tmp = a;
        a = b;
        b = tmp;
    }
}
```

main.cpp

```
int main()
{
    char x='e', y='d';
    trier(x, y);
}
```

type T déduit : char

Templates de fonctions

- *Utilisable avec tout type compatible opérations < et =*

utile.h

```
template<typename T>
void trier(T& a, T& b)
{
    if ( !(a<b) )
    {
        T tmp = a;
        a = b;
        b = tmp;
    }
}
```

main.cpp

```
int main()
{
    char x='e', y='d';
    trier(x, y);
    std::cout << x << " " << y << std::endl;

    double i=5.47, j=3.14;
    trier(i, j);
    std::cout << i << " " << j << std::endl;

    std::string m="world", n="hello";
    trier(m, n);
    std::cout << m << " " << n << std::endl;
```

```
d e
3.14 5.47
hello world
```

Templates de fonctions

- *Utilisable avec tout type compatible opérations < et =*

date.h

```
struct Date {
    int jour, mois, annee;
};

bool operator<(const Date& d1, const Date& d2) {
    if ( d1.annee < d2.annee ) return true;
    if ( d2.annee < d1.annee ) return false;
    if ( d1.mois < d2.mois ) return true;
    if ( d2.mois < d1.mois ) return false;
    if ( d1.jour < d2.jour ) return true;
    return false;
}

std::ostream& operator<<(std::ostream& os, const Date& date) {
    os << date.jour << '/' << date.mois << '/' << date.annee;
    return os;
}
```

main.cpp

```
int main()
{
    Date d1{27, 7, 2018};
    Date d2{15, 7, 2018};
    trier(d1, d2);
    std::cout << d1 << " " << d2 << std::endl;
```

```
15/7/2018 27/7/2018
```

Templates de fonctions

- *Appeler un template depuis un template...*

utile.h

```
template<typename T>
void trier(T& a, T& b)
{
    if ( !(a<b) )
    {
        T tmp = a;
        a = b;
        b = tmp;
    }
}

template<typename T>
void trierEtAfficher(T a, T b)
{
    trier(a, b);
    std::cout << a << " " << b << std::endl;
}
```

main.cpp

```
int main()
{
    trierEtAfficher('e', 'd');
    trierEtAfficher(5.47, 3.14);
    trierEtAfficher("world", "hello");
```

```
d e
3.14 5.47
world hello
```

Templates de fonctions

- ***Attention aux pièges de la déduction automatique !***

utile.h

```
template<typename T>
void trier(T& a, T& b)
{
    if ( !(a<b) )
    {
        T tmp = a;
        a = b;
        b = tmp;
    }
}

template<typename T>
void trierEtAfficher(T a, T b)
{
    trier(a, b);
    std::cout << a << " " << b << std::endl;
}
```

main.cpp

```
int main()
{
    trierEtAfficher('e', 'd');
    trierEtAfficher(5.47, 3.14);
    trierEtAfficher("world", "hello");
```

```
d e
3.14 5.47
world hello
```

???

Templates de fonctions

- ***Attention aux pièges de la déduction automatique !***

utile.h

```
template<typename T>
void trier(T& a, T& b)
{
    if ( !(a<b) )
    {
        T tmp = a;
        a = b;
        b = tmp;
    }
}

template<typename T>
void trierEtAfficher(T a, T b)
{
    trier(a, b);
    std::cout << a << " " << b << std::endl;
}
```

***type T déduit : const char ***
ce sont les adresses des chaînes
littérales qui sont triées, pas les chaînes !

main.cpp

```
int main()
{


    trierEtAfficher("world", "hello");
```

world hello **!!!**

Templates de fonctions

- *On peut forcer une version du template à l'appel ...*

utile.h

```
template<typename T>
void trier(T& a, T& b)
{
    if ( !(a<b) )
    {
        T tmp = a;
        a = b;
        b = tmp;
    }
}

template<typename T>
void trierEtAfficher(T a, T b)
{
    trier(a, b);
    std::cout << a << " " << b << std::endl;
}
```

on a une alternative plus satisfaisante pour le code client au chapitre spécialisation de templates

type T explicite: std::string
les paramètres sont convertis
(si il existe une conversion)

main.cpp

```
int main()
{


    trierEtAfficher<std::string>("world", "hello");
```

Chaîne littérales convertie en std::string

```
hello world
```

Templates de fonctions

- *Avec un « type perso » (classe utilisateur) ...*

utile.h

```
template<typename T>
void trier(T& a, T& b)
{
    if ( !(a<b) )
    {
        T tmp = a;
        a = b;
        b = tmp;
    }
}

template<typename T>
void trierEtAfficher(T a, T b)
{
    trier(a, b);
    std::cout << a << " " << b << std::endl;
}
```

main.cpp

```
int main()
{

    trierEtAfficher( Date{27, 7, 2018},
                     Date{15, 7, 2018} );
```

```
15/7/2018 27/7/2018
```

Templates de fonctions

- *Il faut que toutes les opérations faites par le code templaté sur les paramètres du type concret utilisé (copies, comparaisons, affichages...) soient possibles*
- *Sinon le compilateur se manifeste → error*
- *La syntaxe explicite d'utilisation d'un type est toujours possible au niveau de l'appelant, même quand elle n'est pas indispensable (utile pour confirmer l'intention)*

```
trierEtAfficher<char>('e', 'd');
```

- *Il existe d'autres types de paramètres de templates que* `typename`*, les règles sont complexes, on ne peut pas aborder tous les aspects en un seul cours, plutôt en 300 pages*

COURS 11

A) Fonctions/méthodes inline
B) Programmation générique
C) Templates de fonctions
D) Templates de classes
E) Spécialisation de templates
F) Paramètres fonctions

Templates de classes


```
template<typename T> class JusticeLeague { ... };
```

Templates de classes

- *Template de classe : un type paramètre intervient ! Ce type peut être utilisé comme paramètres, comme valeur de retour, comme attribut...*
- *Noter : les méthodes de la classe sont définies* ***inline***

intervalle.h

```
template<typename T>
class Intervalle
{
    public :

        Intervalle(T a, T b)
            : m_a{a}, m_b{b} { }

        bool contient(T x) {
            return m_a<=x && x<=m_b;
        }

    private :
        T m_a;
        T m_b;
};
```

Templates de classes

- *Contrairement aux fonctions il faut spécifier le type lors de l'utilisation*
- *Ensuite l'instance se « souvient », pas besoin de redire le type générique associé à chaque objet*

main.cpp

```cpp
int main()
{
    std::cout << std::boolalpha;

    Intervalle<char> minuscules{'a', 'z'};
    std::cout << minuscules.contient('m') << "\n";
    std::cout << minuscules.contient('3') << "\n";

    Intervalle<double> aigu{0.0, 90.0};
    std::cout << aigu.contient(20) << "\n";
    std::cout << aigu.contient(145) << "\n";

    Intervalle<std::string> contre{"antiatomique", "antivol"};
    std::cout << contre.contient("antilope") << "\n";
    std::cout << contre.contient("hantise") << "\n";

    return 0;
```

```
true
false
true
false
true
false
```

Templates de classes

- *Contrairement aux fonctions il faut spécifier le type lors de l'utilisation*
- *Ensuite l'instance se « souvient », pas besoin de redire le type générique associé à chaque objet*

main.cpp

```cpp
int main()
{
    std::cout << std::boolalpha;

    Intervalle<char> minuscules{'a', 'z'};
    std::cout << minuscules.contient('m') << "\n";
    std::cout << minuscules.contient('3') << "\n";

    Intervalle<double> aigu{0.0, 90.0};
    std::cout << aigu.contient(20) << "\n";
    std::cout << aigu.contient(145) << "\n";

    Intervalle<std::string> contre{"antiatomique", "antivol"};
    std::cout << contre.contient("antilope") << "\n";
    std::cout << contre.contient("hantise") << "\n";

    return 0;
```

```
true
false
true
false
true
false
```

Templates de classes

- *Pour information (→ futurs informaticiens purs et durs)*
- *Ça change en C++17*
  (C++17 : class template argument deduction)
- *Configurer le compilateur en c++17*
  Mais la version gcc Code::Blocks windows par défaut ne prend pas (et les versions plus récentes de gcc sur windows semblent incompatibles avec les timers de threads...)

  Ça permet de faire `std::vector vec{43, 57, 21};`

  au lieu de faire `std::vector<int> vec{43, 57, 21};`
- *Désormais les types des classes templates peuvent être déduits !* ***Mais à certaines conditions...***
- *Ça rajoute des pièges (char* ≠ std::string etc …)*

Templates de classes

- *Si les méthodes sont longues (plus de 4 ou 5 lignes) il peut être malcommode de les coder inline directement dans la définition de la classe template*
- *Dans ce cas une définition « déportée » est possible*

interface — ***intervalle.h***

```
template<typename T>
class Intervalle
{
    public :
        Intervalle(T a, T b);
        bool contient(T x) ;

    private :
        T m_a;
        T m_b;
};

#include "intervalle.tpp"
```

implémentation — ***intervalle.tpp***

```
template<typename T>
Intervalle<T>::Intervalle(T a, T b)
    : m_a{a}, m_b{b}
{ }

template<typename T>
bool Intervalle<T>::contient(T x)
{
    return m_a<=x && x<=m_b;
}
```

Templates de classes

- *Les autres mécanismes usuels du C++ s'appliquent*
- *En particulier on peut surcharger, hériter, redéfinir...*
- *Exemple d'héritage avec ajout de fonctionnalité*

```
template<typename T>
class IntervalleParcouru : public Intervalle<T>
{
    public :
        IntervalleParcouru(T a, T b, T pas)
            : Intervalle<T>{a, b}, m_pas{pas}, m_idx{a} { }

        bool fini() {
            return !Intervalle<T>::contient(m_idx);
        }

        T getNextStep() {
            T actuel = m_idx;
            m_idx += m_pas;
            return actuel;
        }

    private :
        T m_pas;
        T m_idx;
};
```

Templates de classes

- *Les autres mécanismes usuels du C++ s'appliquent*
- *En particulier on peut surcharger, hériter, redéfinir...*
- *Exemple d'héritage avec ajout de fonctionnalité*

```
template<typename T>
class IntervalleParcouru : public Intervalle<T>
{
    public :
        IntervalleParcouru(T a, T b, T pas)
            : Intervalle<T>{a, b}, m_pas{pas}, m_idx{a} { }

        bool fini() {
            return !Intervalle<T>::contient(m_idx);
        }

        T getNextStep() {
            T actuel = m_idx;
            m_idx += m_pas;
            return actuel;
        }

    private :
        T m_pas;
        T m_idx;
};
```

Ne pas hésiter à re-préciser le type paramètre utilisé

Templates de classes

```
int main()
{
    IntervalleParcouru<int> parkour1{10, 20, 2};
    while ( !parkour1.fini() )
        std::cout << parkour1.getNextStep() << " ";

    std::cout << std::endl;

    IntervalleParcouru<char> parkour2{'e', 'w', 3};
    while ( !parkour2.fini() )
        std::cout << parkour2.getNextStep() << " ";

    std::cout << std::endl;

    IntervalleParcouru<double> parkour3{2.0, 2.5, 0.1};
    while ( !parkour3.fini() )
        std::cout << parkour3.getNextStep() << " ";

    std::cout << std::endl;


    IntervalleParcouru<std::string> parkour4{"ba", "babababa", "ba"};
    while ( !parkour4.fini() )
        std::cout << parkour4.getNextStep() << " ";

    return 0;
}
```

```
10 12 14 16 18 20
e h k n q t w
2 2.1 2.2 2.3 2.4 2.5
ba baba bababa babababa
```

Templates de classes

- *Contrairement au polymorphisme dynamique on ne peut pas mélanger différents types templatés dans un même conteneur (lui même un template !)*
- *Par exemple on ne pourrait pas avoir*
```
std::vector<Intervalle> mix;
```
- *Ni faire*
```
std::vector<Intervalle*> mix;
```
- *Mais on peut avoir*
```
std::vector<Intervalle<double>> intervallesReels;
```

- *Et en supposant que ça fait sens, une classe templatée peut accueillir des objets de types dérivés du type T si ce type est utilisé par adresse (polymorphisme) : c'est précisément ce que font les conteneurs STL !*

Templates de classes

- *Enfin avec plusieurs paramètres de types on peut rendre génériques des structures complexes*

```
/// Différents types de sommets
class Sommet2D
{
    private :
        double m_x, m_y;
};

class Sommet3D
{
    private :
        double m_x, m_y, m_z;
};

/// Différents types de faces
template<typename Sommet>
class Triangle
{
    private :
        Sommet* m_sommets[3];
};

template<typename Sommet>
class Polygone
{
    private :
        std::vector<Sommet*> m_sommets;
};
```

Exemple très incomplet (structure générale)

Templates de classes

- *Enfin avec plusieurs paramètres de types on peut rendre génériques des structures complexes*

Exemple très incomplet (structure générale)

```
/// Un type maillage générique
template<typename Face, typename Sommet>
class Maillage
{
    private :
        std::vector<Sommet*> m_sommets;
        std::vector<Face*> m_faces;
};

int main()
{
    Maillage<Triangle<Sommet2D>, Sommet2D> maillageProjet;

    Maillage<Polygone<Sommet3D>, Sommet3D> maillageKillerApp;

    return 0;
}
```

COURS 11

A) Fonctions/méthodes inline
B) Programmation générique
C) Templates de fonctions
D) Templates de classes
E) Spécialisation de templates
F) Paramètres fonctions

Spécialisation de templates

Spécialisation de templates

- *Pour le confort du code client on peut vouloir préciser un comportement spécifique pour un type T connu*
- *Par exemple convertir les chaînes littérales en string*

```
template<typename T>
void trierEtAfficher(T a, T b)
{
    trier(a, b);
    std::cout << a << " " << b << std::endl;
}



template<>
void trierEtAfficher<const char*>(const char* a, const char* b)
{
    std::string sa{a};
    std::string sb{b};
    trierEtAfficher(sa, sb);
}
```

Spécialisation de templates

- *Pour le confort du code client on peut vouloir préciser un comportement spécifique pour un type T connu*
- *Par exemple convertir les chaînes littérales en string*

```
template<typename T>
void trierEtAfficher(T a, T b)
{
    trier(a, b);
    std::cout << a << " " << b << std::endl;
}
```

indique une spécialisation

précise pour quel type on spécialise

```
template<>
void trierEtAfficher<const char*>(const char* a, const char* b)
{
    std::string sa{a};
    std::string sb{b};
    trierEtAfficher(sa, sb);
}
```

Spécialisation de templates

- *Pour le confort du code client on peut vouloir préciser un comportement spécifique pour un type T connu*
- *Par exemple convertir les chaînes littérales en string*

```
template<typename T>
void trierEtAfficher(T a, T b)
{
    trier(a, b);
    std::cout << a << " " << b << std::endl;
}
```

indique une spécialisation

pour les cas simples la déduction de type marche

```
template<>
void trierEtAfficher(const char* a, const char* b)
{
    std::string sa{a};
    std::string sb{b};
    trierEtAfficher(sa, sb);
}
```

Spécialisation de templates

- *Pour le confort du code client on peut vouloir préciser un comportement spécifique pour un type T connu*
- *Par exemple convertir les chaînes littérales en string*

```
int main()
{

    trierEtAfficher<std::string>("world", "hello");

    trierEtAfficher("world", "hello");

    return 0;
}
```

Grâce à la spécialisation ces 2 appels côté client vont fonctionner de la même façon ce qui est préférable !

Spécialisation de templates

- *Une spécialisation bien connue (et mal aimée) de la STL est le* `std::vector<bool>`
- *En effet en le spécialisant la STL est capable de ne réserver en mémoire que 1 seul bit par case !*
- *C'est 8 fois plus efficace qu'une implémentation naïve*
- *Mais c'est une fausse bonne idée : comment le code suivant va fonctionner ?*

```
std::vector<bool> vec{true, false};
bool* ptr = &vec[1];
```

- *Ça ne compile pas ! Le vecteur de bits est donc un conteneur dont on ne peut pas prendre l'adresse d'un élément : c'est un cas particulier, et les cas particuliers compliquent la programmation générique.*

COURS 11

A) Fonctions/méthodes inline
B) Programmation générique
C) Templates de fonctions
D) Templates de classes
E) Spécialisation de templates
F) Paramètres fonctions

Paramètres fonctions

```
template<typename HeroicFunctionType>
beMyHero(HeroicFunctionType myHeroicFunction)
{ ... myHeroicFunction(supervillain) ... }
```

Paramètres fonctions

- Inversion de contrôle (*inversion of control*)
- Différentes techniques « objets » pour réaliser l'inversion de contrôle. Hériter d'une interface du framework est une des façons...

```
/// Classe interface (Abstraite pure)
class Fonction
{
  public :
    virtual double evaluer(double x)=0;
};

/// Intégration méthode du point milieu
double integrer(Fonction& f,
                double a, double b,
                double pas)
{
  double somme = 0;
  for (double x = a+pas/2; x<b; x+=pas)
    somme += f.evaluer(x) * pas;
  return somme;
}
```

```
/// Code utilisateur
class Fracrat : public Fonction
{
  public :
    double evaluer(double x);
};

double Fracrat::evaluer(double x)
{
  return 1/(1+x*x);
}

int main()
{
  Fracrat fr;
  std::cout<<4.0*integrer(fr,
                0, 1,
                0.001) << std::endl;
```

```
3.14159
```

Paramètres fonctions

- Inversion de contrôle (*inversion of control*)
- Différentes techniques « objets » pour réaliser l'inversion de contrôle. Hériter d'une interface du framework est une des façons...

```
/// Classe interface (Abstraite pure)
class Fonction
{
  public :
    virtual double evaluer(double x)=0;
};

/// Intégration méthode du point milieu
double integrer(Fonction& f,
                double a, double b,
                double pas)
{
  double somme = 0;
  for (double x = a+pas/2; x<b; x+=pas)
    somme += f.evaluer(x) * pas;
  return somme;
}
```

Polymorphisme...

Appel par l'interface

```
/// Code utilisateur
class Fracrat : public Fonction
{Classe concrète hérite interface
  public :
    double evaluer(double x);
};

double Fracrat::evaluer(double x)
{
  return 1/(1+x*x);
}

int main()
{
  Fracrat fr;
  std::cout<<4.0*integrer(fr,
                0, 1,
                0.001) << std::endl;
```

Implémentation !

3.14159

Paramètres fonctions

- *Quelle usine à gaz !*
- *Heureusement avec les paramètres templatés on a un mécanisme beaucoup plus simple pour passer une fonction en paramètre*

```
/// Fonction template avec type
/// "paramètre utilisé en fonction"

template<typename F>
double integrer(F f,
                double a, double b,
                double pas)
{
  double somme = 0;
  for (double x = a+pas/2; x<b; x+=pas)
    somme += f(x) * pas;
  return somme;
}
```

```
/// Code utilisateur
double fracrat(double x)
{
    return 1/(1+x*x);
}

int main()
{

std::cout<<4.0*integrer(fracrat,
                        0, 1, 0.001)
         << std::endl;
```

```
3.14159
```

Paramètres fonctions

- *Un paramètre templaté peut recevoir une fonction*
- *Il reçoit en fait l'**adresse** d'une fonction …*
- *On peut faire ça sans template mais les déclarations de pointeurs de fonctions ne sont pas sympathiques*

```
/// Fonction template avec type
/// "paramètre utilisé en fonction"

template<typename F>
double integrer(F f,
                double a, double b,
                double pas)
{
  double somme = 0;
  for (double x = a+pas/2; x<b; x+=pas)
    somme += f(x) * pas;
  return somme;
}
```

en appliquant f la fonction intégrer applique en fait la fonction reçue en paramètre

```
/// Code utilisateur
double fracrat(double x)
{
    return 1/(1+x*x);
}

int main()
{

std::cout<<4.0*integrer(fracrat,
                        0, 1, 0.001)
         << std::endl;
```

```
3.14159
```

Paramètres fonctions

- *Le gros avantage, on peut recevoir n'importe quelle entité qui se comporte comme une fonction...*
- *Par exemple une fonction anonyme (lambda) pas au programme, juste pour montrer la suite en C++*

```
/// Fonction template avec type
/// "paramètre utilisé en fonction"

template<typename F>
double integrer(F f,
                double a, double b,
                double pas)
{
  double somme = 0;
  for (double x = a+pas/2; x<b; x+=pas)
    somme += f(x) * pas;
  return somme;
}
```

en appliquant f la fonction intégrer applique en fait la fonction reçue en paramètre

```
/// Code utilisateur

int main()
{
  std::cout << 4.0*integrer(

[](double x){ return 1/(1+x*x);},

0, 1, 0.001) << std::endl;
```

```
3.14159
```

Paramètres fonctions

- *Le gros avantage, on peut recevoir n'importe quelle entité qui se comporte comme une fonction...*
- *Par exemple un objet fonction (foncteur) qui est une sorte de fonction paramétrable*

```
/// Classe foncteur : classe d'objets "fonctions paramétrables"
class SecondDegre {
    public :

        SecondDegre(double a, double b, double c)
            : m_a{a}, m_b{b}, m_c{c}   {  }

        double operator() (double x) {
            return m_a*x*x + m_b*x + m_c;
        }

    private :
        double m_a, m_b, m_c;
};

int main() {
    SecondDegre monPolynome{1.5, -1, 2};
    std::cout << integrer(monPolynome, 1, 2, 0.001) << std::endl;
```

$$f(x) = ax^2 + bx + c$$