# Fonctions de rappel en C et C++

## Fonctions de rappel en C

### Qu'est-ce qu'une fonction de rappel?

Une *fonction de rappel* (= *callback*) est une fonction passée en argument d'une autre fonction.

Est-ce possible en C ? Oui, par les *pointeurs de fonction*.

### Pointeur de fonction

Un pointeur de fonction pointe vers du code exécutable en mémoire au lieu de pointer vers des données. Il permet d'appeler la fonction pointée avec l'opérateur `()` comme lors d'un appel de fonction habituel.

```
double inches_to_cm(double inches) {
  return inches * 2.54;
}

int main() {
  double (*f)(double); //declaration
  f = inches_to_cm;    //initialisation
  printf("1.0 inch equals to %f cm \n", f(1.0));
  return 0;
}
```

### Pointeur de fonction: syntaxe

La déclaration d'un pointeur de fonction peut repousser au premier abord. Ce qu'il faut retenir, c'est qu'on utilise la signature d'une fonction (type de retour, nom, liste des paramètres) en faisant précéder le nom d'une étoile (pour indiquer que c'est un pointeur).

```
typeDeRetour (*nomPointeurFonction)(typeParametre1, typeParamètre2);
```

L'introduction d'un alias à l'aide de `typedef` permet de remplacer cette signature par une simple étiquette:

```
typedef typeDeRetour (*NouveauType)(typeParametre1, typeParamètre2);
NouveauType nomPointeurFonction;
```

### Ex.1. Exemple d'application (10 min)

- Copiez la fonction suivante après avoir défini le type `FonctionDeRDansR`.

```
double integration( double a, double b, int n,
                    FonctionDeRDansR f ) {
  double delta = (b - a) / n;
  double sum = 0.0;
  int i;
  for (i = 0; i < n; ++i)
    sum += f(a + i*delta);
  return sum * delta;
}
```

- Ecrivez un `main` qui intègre numériquement la fonction `cos` dans $[0;1]$. Comparez avec la valeur $sin(1)$.

### Intérêt et limite

On peut donc passer des fonctions en argument, ce qui permet de choisir à l'exécution la fonction à appeler. Dans l'exemple précédent, le même code peut intégrer numériquement n'importe quelle fonction qui prend un double en paramètre et retourne un double.

Cependant, on est limité aux fonctions, qu'on peut voir comme des objets sans état, non paramétrable. Comment faire pour manipuler des fonctions paramétrables (par exemple la fonction $s : \mathbb{R} \to \mathbb{R}$ telle que $s(x) = Acos(x) + Bsin(x)$) ?

## Introduction au C++

### Ce que vous savez déjà

Comme son nom l'indique, le C++ est un sur-ensemble du C. D'une certaine manière, vous savez donc déjà programmer en C++!

Mais c'est une illusion, car les fonctionnalités les plus intéressantes sont justement celles qui manquent au C.

Notez bien l'adresse suivante: http://www.cplusplus.com/.

### Exemple Hello World à la C

```
#include <cstdio>
int main() {
  printf("Hello World! \n");
  return 0;
}
```

Compilation et édition de liens:

```
g++ Hello1.cpp -o Hello1
```

### Caractéristiques

Aux caractéristiques du C, s'ajoutent:

- une couche *orientée objet* dès 1979 avec *C with classes*, rebaptisé *C++* en 1983: classes, héritage (multiple), méthodes virtuelles
- les références et les passages de paramètres par référence
- les arguments par défaut, la surcharge (= *overloading*), la redéfinition (= *overriding*),
- la gestion des erreurs et exceptions
- les patrons (= *template*)
- les espaces de noms (= *namespace*)

### Exemple Hello World en C++

```
#include <iostream>
int main() {
  std::cout << "Hello World!" << std::endl;
  return 0;
}
```

- Compilez et exécutez.
- Vous utilisez déjà sans le savoir un namespace (`std`), les templates (via `cout`), la surcharge d'opérateurs (<<) en utilisant la librairie standard ...

### La librairie standard C++

La librairie standard C++ est composée:

- de la librairie standard C (en-têtes commençant par la lettre 'c')
- de la librarie de flux d'entrée-sortie
- de la STL (= Standard Template Library) composée de
  - conteneurs
  - foncteurs
  - iterateurs
  - algorithmes

Nous allons nous concentrer sur le concept de *foncteur*, un objet qui généralise le concept de *fonction*.

## Programmation par classe

### Classe

Une classe définit l'implémentation de l'état et du comportement d'un objet par un regroupement de **champs** (variables) et de **méthodes** (fonctions).

```
#include <cmath>
class Sinusoide {
  private:
    double myA, myB;
  public:
    Sinusoide(double aA, double aB) : myA(aA), myB(aB) {}
    double eval(double x) {
      return myA*std::sin(x) + myB*std::cos(x);
    }
};
```

Attention à finir la déclaration de la classe par un point-virgule!

## Instanciation sur la pile

Comme n'importe quelle variable de type fondamental, un objet peut être instancié ainsi:

```
double x(5.0); //ou double x = 5.0;
Sinusoide s(2.0, 3.0); //ou Sinusoide s = Sinusoide(2.0, 3.0);
```

L'objet est alors créé sur la pile.

Une déclaration seule appelle le constructeur sans paramètre.

```
double x;
Sinusoide s; //ne compile pas, car il manque des arguments
```

## Instanciation sur le tas

La mémoire peut aussi être allouée sur le tas avec l'opérateur `new`:

```
double* xPtr = new double(5);
Sinusoide* sPtr = new Sinusoide(2.0, 3.0);
```

Dans ce cas, on ne manipule pas l'objet directement, mais via un *pointeur*. On accède donc à ces champs ou méthodes (publiques) par l'opérateur flèche (`->`) et non par l'opérateur point (`.`).

```
return sPtr->eval(x); //evalue la sinusoide en x
```

## Destruction

Quand un objet est instancié sur la pile, il est automatiquement détruit, comme n'importe quelle

variable.

En revanche, quand un objet est instancié sur le tas, c'est le développeur qui a la responsabilité de libérer la mémoire, avec l'opérateur `delete`:

```
delete sPtr; //libère la mémoire allouée
```

Il n'y a pas de *garbage collector* comme en Java, alors pour éviter les fuites mémoires, une règle simple:

- à tout `new` doit être associé un et un seul `delete`.

## Héritage

Comme en Java, l'**héritage** mêle deux mécanismes:

- l'**extension**: la classe dérivée hérite des services de la classe de base et les étend.
- le **polymorphisme**: un objet de la classe dérivée peut être utilisé comme un objet de la classe de base (transtypage ascendant).

NB: à la différence de Java, une classe peut hériter de *plusieurs* classes (héritage multiple).

## Exemple

```
class BaseClass {
  public:
    void method() {
      std::cout << "from Base" << std::endl;
    }
};
```

```
class DerivedClass : public BaseClass {
  public:
    void method() {
      std::cout << "from Derived" << std::endl;
    }
};
```

## Liaison dynamique

Que fait ce code ?

```
int main() {
  BaseClass* ptr = new DerivedClass();
  ptr->method();
}
```

Maintenant ajoutez le mot-clef `virtual` devant la signature de la méthode `method` de `BaseClass`. Cela permettra d'activer la **liaison dynamique** (= *dynamic binding*): c'est le mécanisme qui permet, à l'appel d'une méthode, d'exécuter non pas la méthode de la classe de base, mais sa version redéfinie dans une classe dérivée. Quelle différence observez-vous ?

## Classe abstraite

Bien sûr, une **méthode abstraite**, qui n'a pas de corps, doit être obligatoirement déclarée comme virtuelle. Une **classe abstraite** possède au moins une méthode abstraite et ne peut pas être instanciée.

```
class FonctionDeRDansR {
public:
  virtual double eval(double x) = 0;
};
```

NB. le C++ n'intègre pas la notion d'**interface** de Java, mais une interface peut être vue comme une **classe abstraite** (du fait de l'héritage multiple).

## Les foncteurs et l'opérateur `()`

Pour le moment, les objets de type `FonctionDeRDansR` ont une méthode appelée `eval` et ne ressemblent donc pas aux fonctions habituelles.

Heureusement, en C++, il est possible de les doter de l'opérateur `()` en définissant la méthode appelée `operator()`. Ils deviennent dès lors des *foncteurs*, c'est-à-dire des objets qui, dotés de l'opérateur `()`, se comportent comme des fonctions.

## Ex.2. Integration (15 min)

- Copiez les classes `FonctionDeRDansR` et `Sinusoide`, cette dernière dérivant de `FonctionDeRDansR`.
- Remplacez `eval` par `operator()` dans `FonctionDeRDansR` et `Sinusoide`.
- Dans du code client, testez l'intégration de la fonction cosinus sur $[0; 1]$. Comparez avec la valeur $sin(1)$.

## Fonctionnement

La fonction `integration` accepte comme dernier argument tout objet issu d'une classe dérivée de la classe abstraite `FonctionDeRDansR`.

A chaque appel de l'opérateur `()`, la table de fonctions virtuelles (= *vtable*) associée à la classe réelle de l'objet passé en argument (`Sinusoide` ici) est consultée pour déléguer l'éxécution à la méthode appropriée. Il y a donc le coût supplémentaire d'une redirection à chaque appel d'une fonction virtuelle.

Nous allons voir qu'en C++, il est possible d'opter pour un polymorphisme statique qui évite le coût de la liaison dynamique.

# Programmation template

## Template

En C++, les fonctions et les classes peuvent être rendues génériques en les paramétrant par des types.

Cela rappelle les *generics* du Java; sauf que la substitution entre les *paramètres template* et les arguments est effectuée à la compilation.

Prenons l'exemple d'une fonction qui calcule, entre deux valeurs, la plus petite. Quel que soit le type des valeurs en question, l'algorithme est le même:

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

## Ex.3. Min (5 min)

- Essayez la fonction `std::min`. Que se passe-t-il avec les appels suivants ?

```
#include <algorithm>
...
std::cout << std::min(5, 2) << std::endl;
std::cout << std::min(5.9, 2.1) << std::endl;
std::cout << std::min('a', 'b') << std::endl;
std::cout << std::min("bonjour", "au revoir") << std::endl;
std::cout << std::min<std::string>("bonjour", "au revoir") << std::endl;
std::cout << std::min(5, 2.1) << std::endl;
std::cout << std::min<double>(5, 2.1) << std::endl;
std::min( Sinusoide(1.0, 0.0), Sinusoide(0.0, 1.0) );
```

## Fonctionnement

A chaque appel d'une fonction template, le compilateur commence par déduire les *paramètres template* à partir des arguments. Dans l'exemple précédent, le paramètre template `T` de `std::min` sera forcément `int` si les arguments sont `5` et `2`. S'il y a ambiguité, les paramètres templates doivent être explicitement fournis entre chevrons après le nom de la fonction.

Pour chaque fonction template, seulement les versions déduites des appels sont compilés.

Dans l'exemple précédent, seulement les versions où le paramètre template `T` est `int`, `double`, `char` et `std::string` sont générées. Ainsi, une fonction template ne peut être compilée séparément.

## Ex.4. Integration/Template (10 min)

- Proposez une version *template* de la fonction `integration`.

## Polymorphisme statique vs dynamique

En C++, quand c'est possible, on préfère le **polymorphisme statique**, s'appuyant sur le mécanisme template, plutôt que le **polymorphisme dynamique**, basé sur l'héritage.

- plus fort **découplage** (pas besoin d'hériter d'une même classe).
- plus **performant** (aucun surcoût dû à la liaison dynamique).

Le code est cependant plus long à compiler, l'exécutable plus volumineux (puisqu'il contient plusieurs versions des mêmes méthodes).

## Conclusion

En C, il est possible de passer des fonctions et plus généralement en C++, des objets dotés de l'opérateur `()`, appelés *foncteurs*, comme arguments à d'autres fonctions.

Cela permet de rendre un bout de code paramétrable par une famille de fonctions de même signature (et même sémantique).