# Patrons, Modèles & Généricité

Rendre des fonctions, objets paramétrables par des types et des valeurs

# L'idée simple des « Template (Patrons) »

- Une simple fonction « identité »

```
int identity(int x) { return x; }
string identify(string x) { return x; }
float identity(float x) { return x; }
```

- Cette fonction est là même pour tous les types. Cependant, nous devons écrire une fonction pour chaque type !
- Pourquoi ne pas déléguer cette tâche au compilateur ?

# Principe des fonctions « Template »

- Idée : paramétrer la définition d'une fonction par un type :

```
template<class T>
T identity(T x) { return x; }
```

Ou

```
template<typename T>
T identity(T x) { return x; }
```

- La fonction `identity` devient une fonction qui prend un argument de type `T` et retourne une value de type `T`.

# Que fait le compilateur C++ ?

```
int x = identity(3);
```

- Le compilateur cherche la fonction « `identity` » dans la table des fonctions.
- Le compilateur trouve un patron pour la fonction « `identity` » qui prend un paramètre de type `T`.
- Le compilateur identifie le type `T` avec le type de la valeur `3`, soit `int`.
- Le compilateur instantie le code de la fonction pour `identity` le `T=int` et crée le code :

```
int identity(int x) { return x; }
```

- Le compilateur appelle la fonction qu'il vient de créer.

# Conséquences du processus de création des fonctions définies par des « templates »

- Le compilateur a besoin du code la fonction au moment où il rencontre l'appel à la fonction
  - Le code de la fonction est présent dans le fichier .hpp et non pas .cpp.
- Le compilateur génère plusieurs fois la même fonction
  - L'éditeur de liens doit supprimer les fonctions générées plusieurs fois.
- La vitesse de compilation est augmentée par la génération du code des fonctions templates

# La définition de classe paramétrée par des types

```
template<typename T>
class pair
{
   T x;
   T y;
   …
};
```

# Paramètrez par des valeurs

```
template<typename T, size_t size>
class array
{
   T array[size];
    …
};
```

Crée un tableau de `type T` et de taille `size`.

# TD --- Partie 1

- Création d‘une fonction de tri polymorphe.

# Problème des fonctions templates

Voici une fonction qui ajoute un chiffre à une suite de nombres :

```
template<class T1, class T2 >
T1 catDigit(T1 aValue, T2 aDigit)
{ if(aDigit < 0 || aDigit > 10)
    throw std::overflow_error(
      "Is not a digit");
  return aValue * 10 + aDigit;
}
```

# La fonction ne fonctionne pas pour tous les types !

```
std::string number = "123";
number = catDigit(number, 4);
```

- Le compilateur génère une erreur au moment de créer la fonction `catDigit` avec `T1=std::string` et `T2=int`.
  - Opérateur `*` n'est pas définir pour le type `std::string`.

# Solution « Fonction partiellement ou totalement Spécialisée » !

```
template<class T2 >
T1 catDigit<std::string, T2>(
            std::string aValue, T2 aDigit)
{
  char buffer[1];
  int digit = _itoa(aDigit, buffer, 10);
  if(digit < 0 || digit > 10)
    throw std::overflow_error(
      "Is not a digit");
  return aValue +
}
```

# Que fait le compilateur C++ ?

```
std::string number = "123";
number = catDigit(number, 4);
```

- Le compilateur cherche la fonction « `catDigit` » dans la table des fonctions.
- Le compilateur trouve un patron spécialisé pour le type `std::string`. Qui est prioritaire sur le patron générique.
- Le compilateur identifie le type `T` avec le type de la valeur `3`, soit `int`.
- Le compilateur instantie le code de la fonction pour `identity` le `T=int` et crée le code :

```
int identity(int x) { return x; }
```

- Le compilateur appelle la fonction qu'il vient de créer.

# La fonction ne fonctionne pas correctement pour tous les types !

```
float number = "123.4";
number = catDigit(number, 5);
std::cout << number << "\n";
// affiche 1235.4
```

- Le résultat attendu aurait été 123.45.
- Le problème : type `float` est un type à virgule flottante, type `int` est un type à virgule fixe.

# Définition d'une classe de caractérisant le type

```
template<class valueT>
struct numeric_traits
{
public:
  bool hasFractionalParts() = false;
  template<class T>
  static valueT append(valueT aValue,
    T aDigit)
    {
       return aValue * 10 + aDigit;
    }
};
```

# Définition d'une classe spécialisée caractérisant le type float

```
template<>
struct numeric_traits<float>
{
public:
  bool hasFractionalParts() = true;
  template<class T>
  static valueT append(valueT aValue,
    T aDigit)
    {
        char* buffer[20];
      _snprintf(buffer, 20, "%f%d",
         aValue, aDigit);
      return atof(buffer);
    }
};
```

- La classe spécialisée `numeric_traits<float>` définit une classe particulière pour le type **`float`**.

# Le code paramétrisé par la classe « traits »

```
template<typename T1, typename T2,
    typename traits = numeric_traits<T1>>
T1 catDigit(T1 aValue, T2 aDigit, traits* =
null)
{ if(aDigit < 0 || aDigit > 10)
     throw std::overflow_error(
       "Is not a digit");
  return traits::append (aValue,
      aDigit);
}
```

- La fonction `catDigit` est désormais paramétrée par la classe `traits` dont la valeur par défaut est `numeric_traits<T1>`.

# Que fait le compilateur C++ ?

```
float number = "123.4";
number = catDigit(number, 5);
std::cout << number << "\n";
// affiche 1234.5
```

- Le compilateur cherche la fonction `catDigit` pour le type `float` et `int`.

# Que fait le compilateur C++ ? (suite)

Le compilateur trouve

```
template<typename T1, typename
T2,typename traits =
numeric_traits<T1>>
T1 catDigit(T1 aValue, T2 aDigit)
```

Qu'il instancie avec :

```
T1 = float
T2 = int
traits = numeric_traits<float>
```

# Que fait le compilateur C++ ? (suite)

- Pour compiler

```
return traits::append
        (aValue, aDigit);
```

- Il appelle la fonction append de **template<>** **struct** numeric_traits<**float**>

Soit le bon code !

# TD --- Partie 2

- Création des fonctions de tri paramétrables par une classe compare_traits.