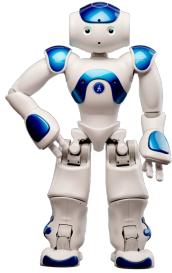


Classe et type génériques

1. Classes génériques (patrons de classes)



- Déclaration d'une classe générique

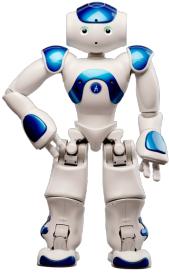
```
template < liste_de_paramètres > déclaration
```

```
class CVector {  
    double x;  
    double y;  
public:  
    CVector();  
    CVector(double a, double b)  
        : x(a), y(b) {}  
};
```



```
template <typename T>  
class CVector {  
    T x;  
    T y;  
public:  
    CVector();  
    CVector(T a, T b)  
        : x(a), y(b) {}  
};
```

Instanciation



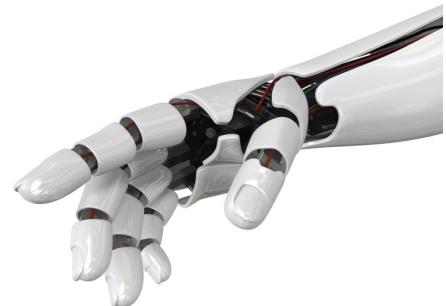
- Une classe générique doit être **instanciée** pour que le compilateur génère son code (comme pour les fonctions)

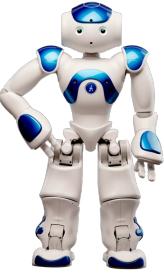
- on peut l'instancier **explicitement**

```
template class CVector<char>;
```

- ou **implicitement**, en l'utilisant dans le code dans l'appelant

```
CVector<int> v(1, 2);  
CVector<double> w(1.2, 3.4);
```

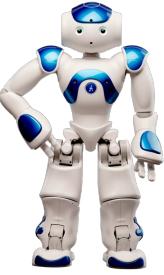




Instanciation

- Avant C++17,
 - toujours indiquer les paramètres génériques entre <>
 - pas de déduction d'arguments pour les classes, contrairement aux fonctions
- Depuis C++17,
 - **déduction possible** des arguments à partir des *paramètres passés au constructeur* selon les mêmes règles que pour les fonctions génériques
 - Le concepteur d'une classe peut guider cette déduction (hors sujet pour PRG1)
- Pour **éviter l'instanciation implicite** : mot-clé **extern**

```
extern template class CVector<int>;
```



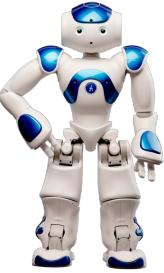
Fonctions membres de classes génériques

- Dans une classe générique, les **fonctions membres** peuvent être définies de deux façons, et utilisent les variables dans typename et le nom de la classe ainsi :

1. Définition en ligne

```
template <typename T>
class CVector {
    ...
    T produitScalaire(const CVector<T>& cv) const {
        return x * cv.x + y * cv.y;
    }
    ...
};
```

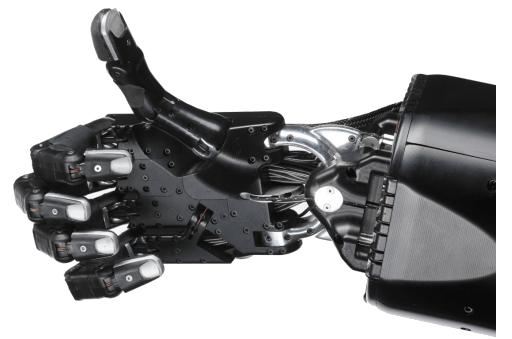
2. Séparation de la *déclaration* et *definition* pour les fonctions membres non triviales (slide suivante) = la bonne pratique



Déclaration et définition séparées

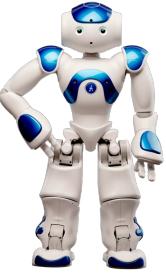
- On **déclare** la méthode dans la déclaration de la classe

```
template <typename T>
class CVector {
    ...
    T produitScalaire(const CVector<T>& cv) const;
    ...
};
```



- On la **définit** en dehors de cette déclaration, avec **obligation** d'indiquer la classe générique (patron) à laquelle elle appartient, comme suit :

```
template <typename T>
T CVector<T>::produitScalaire(const CVector<T>& cv) const {
    return x * cv.x + y * cv.y;
}
```

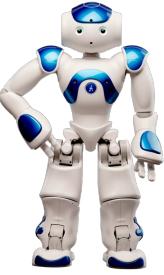


Opérateurs dans les patrons de classes

- Cela vaut également pour les opérateurs membres qui sont (sur ce point) des fonctions membres comme les autres

```
template <typename T>
class CVector {
    ...
    CVector<T> operator+ (const CVector<T>& cv) const;
    ...
};
```

```
template <typename T>
CVector<T> CVector<T>::operator+ (const CVector<T>& cv) const {
    CVector<T> temp;
    temp.x = x + cv.x;
    temp.y = y + cv.y;
    return temp;
}
```

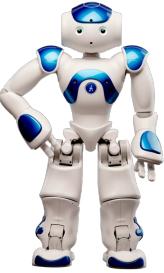


Foncteur – operator()

Nous avons vu la possibilité de créer

- une **fonction lambda** (chap. 10)
- un **foncteur** (chap. 11)

```
struct Entre {  
    int min;  
    int max;  
    bool operator() (int valeur) { return valeur >= min and valeur <= max; }  
};  
  
const vector v1 {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  
  
int min = 3, max = 8;  
cout << count_if(v1.begin(), v1.end(), [min, max] (int valeur) {  
    return valeur >= min and valeur <= max; });  
  
cout << count_if(v1.begin(), v1.end(), Entre{min, max});
```



Foncteur – operator()

Rendre un foncteur générique se fait naturellement

```
template <typename T>
struct Entre {
    const T& min;
    const T& max;
    bool operator() (const T& valeur) { return valeur >= min and valeur <= max; }
};

const vector v1 {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
const vector v2 {1., 2., 3., 4., 5., 6., 7., 8., 9., 10.};

int min = 2, max = 7;

cout << count_if(v1.begin(), v1.end(), Entre<int> {min, max}) << endl;
cout << count_if(v2.begin(), v2.end(), Entre<double>{min, max}) << endl;
```



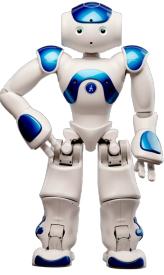
Foncteur – operator()

Une classe peut aussi être utilisée ...
mais ceci nécessite la création d'un constructeur ... un peu lourd

```
template <typename T>
class Entre {
public:
    Entre(const T& min, const T& max) : min(min), max(max) {}
    bool operator()(const T& valeur) { return valeur >= min and valeur <= max; }
private:
    const T& min;
    const T& max;
};

const vector v1{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
const vector v2{1., 2., 3., 4., 5., 6., 7., 8., 9., 10.};
int min = 2, max = 7;

cout << count_if(v1.begin(), v1.end(), Entre<int> (min, max)) << endl;
cout << count_if(v2.begin(), v2.end(), Entre<double>(min, max)) << endl;
```



Argument générique énumérable

- Les arguments génériques énumérables vus au chap 10 s'appliquent également aux classes génériques à l'image de la classe **array** déjà étudiée

```
template < class T, size_t N > class array;
```

- .. et peuvent avoir une valeur par défaut

```
template <typename T, int n=100> class Stack;
```



CVector.h

```
#ifndef CVECTOR_H
#define CVECTOR_H

template <typename T>
class CVector {
    T x, y;
public:
    CVector() {}
    CVector(T a, T b): x(a), y(b) {}
    T produitScalaire(const CVector<T>& cv) const;
    CVector<T> operator+(const CVector<T>& cv) const;
};

template <typename T>
T CVector<T>::produitScalaire(const CVector<T>& cv) const {
    return x * cv.x + y * cv.y;
}

template <typename T>
CVector<T> CVector<T>::operator+(const CVector<T>& cv) const {
    CVector<T> temp;
    temp.x = x + cv.x;
    temp.y = y + cv.y;
    return temp;
}

#endif
```

La définition d'une classe générique

1. Ne peut pas être compilée telle quel pour donner du code objet
 - doit être **instanciée** (comme pour les fonctions génériques)
2. Doit plutôt être considérée comme une déclaration
 - il faut l'inclure entièrement dans un fichier header
 - y compris les fonctions membres



Compilation séparée : solution courante

- couper le fichier header en deux : **déclarations + définitions**
- inclure le fichier avec les définitions après les déclarations

CVector.h

```
#ifndef CVECTOR_H
#define CVECTOR_H

template <typename T>
class CVector {
    T x, y;
public:
    CVector<T>() {}
    CVector<T>(T a, T b) : x(a), y(b) {}
    T produitScalaire(const Cvector<T>& cv) const;
    CVector<T> operator+(const CVector<T>& cv) const;
};

#include "CVectorImpl.h"

#endif
```

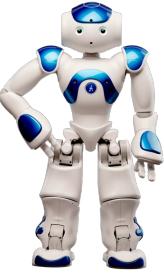
CVectorImpl.h

```
#ifndef CVECTORIMPL_H
#define CVECTORIMPL_H

template <typename T>
T CVector<T>::produitScalaire(const CVector<T>& cv) const {
    return x * cv.x + y * cv.y;
}

template <typename T>
CVector<T> CVector<T>::operator+(const CVector<T>& cv) const {
    CVector<T> temp;
    temp.x = x + cv.x;
    temp.y = y + cv.y;
    return temp;
}

#endif
```



Méthodes génériques

- Comme les autres fonctions, les fonctions membres d'une classe peuvent aussi être **génériques pour d'autres types**
- Attention à choisir des noms différents pour les **paramètres génériques** de la classe et de la fonction membre (ici, T et U)
- La définition s'écrit alors avec deux mots-clés **template**

```
template <typename T>
class CVector {
    ...
    template <typename U>
    CVector<U> convert() const;
    ...
};
```

```
template <typename T>
template <typename U>
CVector<U> CVector<T>::convert() const {
    return CVector<U>((U)x, (U)y);
}
```



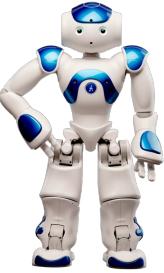
Fonctions amies génériques

- Pour pouvoir afficher les objets de type `CVector<T>`, il faut surcharger `operator<<` de manière générique aussi :

```
// déclaration avancée de CVector, pour pouvoir le
// mentionner dans la déclaration de operator<< juste après
template <typename T> class CVector;

// déclaration + définition de l'opérateur <<
template <typename T>
ostream& operator<<(ostream& os, const CVector<T>& cv) {
    return os << cv.x << ' ' << cv.y;
}

template <typename T> class CVector {
    // amitié entre CVector<T> et l'opérateur << générique avec
    // le paramètre générique effectif T
    friend ostream& operator<< <T>(ostream& os, const CVector<T>& cv);
    ...
};
```



Fonctions amies génériques

- Notons que les déclarations d'amitié peuvent s'écrire de plusieurs manières

```
friend ostream& operator<< <T>(ostream& os, const CVector<T>& cv);  
friend ostream& operator<< <T>(ostream& os, const CVector& cv);  
friend ostream& operator<< <>(ostream& os, const CVector<T>& cv);  
friend ostream& operator<< <>(ostream& os, const CVector& cv);
```

- Mais si l'on oublie <> ou <T> avant la parenthèse ouvrante, alors on aura une erreur de compilation



- Pour traiter différemment un type spécifique, on peut **spécialiser** :
 - **certaines méthodes**, ou
 - **toute la classe** générique
- On le fait en les **redéfinissant** pour ce type précis
- Par exemple, on peut spécialiser le produit scalaire pour le type `bool` ainsi :

```
template <typename T> // définition générale
T CVector<T>::produitScalaire(const CVector<T>& cv) const {
    return x * cv.x + y * cv.y;
}
```

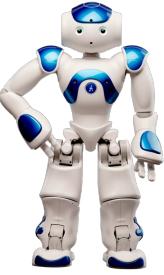
```
template<> // définition spécialisée
bool CVector<bool>::produitScalaire(const CVector& cv) const {
    return (x and cv.x) or (y and cv.y);
}
```



- Pour spécialiser toute une classe générique, on doit réécrire l'entièreté de la classe pour un argument (type) particulier

```
template<> class CVector<bool> {  
    // réécriture de toute la classe pour le type bool  
};
```





Spécialisation partielle

- Pour les classes à plusieurs paramètres génériques, il est possible de ne les **spécialiser que partiellement**, en gardant certains paramètres génériques

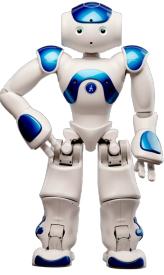
```
template<typename T1, typename T2, int I>
class A {};                                // template primaire

template<typename T, int I>
class A<T, T*, I> {};                      // #1: spécialisation partielle.
                                                // T2 est un pointeur vers T1

template<typename T, typename T2, int I>
class A<T*, T2, I> {};                      // #2: spécialisation partielle.
                                                // T1 est un pointeur

template<typename T>
class A<int, T*, 5> {};                    // #3: spécialisation partielle.
                                                // T1 est un int, I vaut 5,
                                                // et T2 est un pointeur

template<typename X, typename T, int I>
class A<X, T*, I> {};                      // #4: spécialisation partielle.
                                                // T2 est un pointeur
```



Paramètres template template

- Considérons le problème suivant.

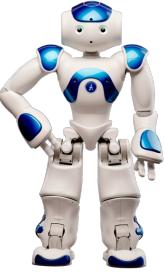
On dispose des conteneurs génériques Liste et Tableau :

```
template <typename T> class Liste { ... };
template <typename T> class Tableau { ... };
```

- On peut utiliser un Tableau pour créer une 3ème sorte de conteneur : Pile

```
template <typename T> class Pile {
    Tableau<T> data;
    ...
};
```

- Mais pourrait-on **laisser à l'utilisateur le choix d'utiliser Liste ou Tableau** pour créer une Pile ? Il faudrait passer le type du conteneur comme paramètre générique.



Paramètres template template

- Une solution (choisie par la STL) consiste à ajouter un paramètre générique pour indiquer le **type du conteneur** :

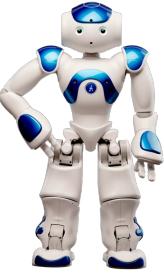
```
template <typename T, typename Conteneur>
class Pile {
    Conteneur data;
    ...
};
```

- Pour utiliser cette classe, on doit déclarer les piles ainsi :

```
Pile<int, Tableau<int>> p1;
Pile<double, Liste<double>> p2;
```

- Mais on préférerait passer directement le type au conteneur (pour déduire `Tableau<int>` du premier `int`) et écrire ceci :

```
Pile<int, Tableau> p1;
Pile<double, Liste> p2;
```



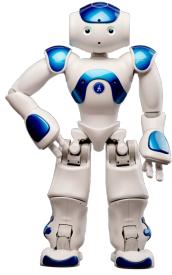
C'est possible !

- Modifier la déclaration de Pile en utilisant un nouveau type de paramètre générique : **template template** (!)

```
template <typename T, template <typename> class Conteneur>
class Pile {
    Conteneur<T> data;
    ...
};
```

- Au prix d'une déclaration plus complexe de Pile
 - son utilisation est simplifiée
 - le risque de se tromper en mélangeant les types disparaît (par exemple en écrivant `Pile<double, Tableau<int>> p1;`)
- Attention, c'est le seul cas où il faut utiliser le mot-clé **class** et pas **typename** (du moins avant la norme C++17)

2. Alias de types génériques (C++11) et variables génériques (C++14)



- C++11 introduit les alias de types génériques. Ils s'écrivent

```
template < template-parameter-list >
    using identifier = type-id ;
```

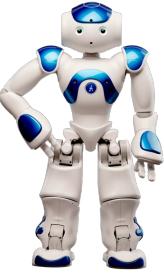
- Par exemple :

```
template <typename T> using Tab10 = array<T,10>;
template <typename T> using ptr = T*;
```

- Utilisation :

```
ptr<int> pi;           // au lieu de : int* pi;
Tab10<double> tab; // au lieu de : array<double,10> tab;
```





Exemple plus élaboré

- Pour simplifier des types complexes, par exemple :

```
template <typename T>
using rIter = vector<T>::reverse_iterator;
```

Error: Missing 'typename' prior to dependent type
name 'vector<T>::reverse_iterator'

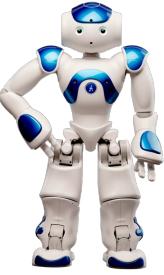
Message
explicite !

- Le compilateur ne peut être sûr que reverse_iterator est un type et non pas une variable, et vous suggère de le préciser avec le mot-clé **typename**

```
template <typename T>
using rIter = typename vector<T>::reverse_iterator;
```

- Attention à ne mettre **typename** qu'en cas d'ambiguïté, sinon erreur de compilation.





Variables génériques

- C++14 introduit la possibilité de déclarer des variables génériques.
L'exemple classique est :

```
template <typename T>
const T PI = T(3.1415926535897932385);
```

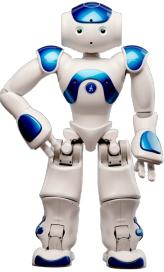
- On instancie et utilise par exemple cette variable ainsi :

```
template <typename T>
T circular_area(T r) {
    return PI<T> * r * r; // instantiation implicite de PI<T>
}
```

- Cela permet d'éviter des conversions de types inutiles
si nous avions défini PI de type **double** par exemple



3. Contrats et concepts dans la STL



Contrat implicite

- Ecrire une fonction générique telle que

```
template <typename T> T square(T a) {  
    return a * a;  
}
```

Invalid operands to binary expression
('const char *' and 'const char *')

implique un contrat implicite avec l'utilisateur de cette fonction :

il doit l'instancier avec un type pour lequel l'opérateur * existe, par exemple le type int

```
cout << square(5) << endl; // affiche 25
```

- Par contre, le code suivant ne compile pas.

```
cout << square("Hello") << endl;
```

L'erreur de compilation est indiquée dans la fonction générique, pas là où elle est : i.e. à l'instanciation de square avec un type non compatible.



Non-respect du contrat implicite

- Reprenons la version minimaliste de notre classe CVector

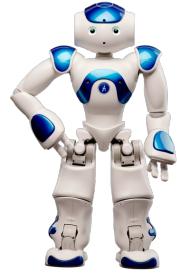
```
class CVector {  
    int x, y;  
public:  
    CVector(int x, int y) : x(x), y(y) {};  
};
```

- Et stockons des objets de ce type dans un std::vector.

```
vector<CVector> v;
```

- Jusqu'ici tout va bien...
- Essayons de redimensionner ce vector → Erreur !

```
v.resize(2);
```



Non-respect du contrat implicite

- Erreur de compilation à la ligne 1673 du fichier <memory> de la librairie standard !

The screenshot shows an IDE interface with a toolbar at the top. The left sidebar has 'By File' selected, showing a tree with 'clients 1 issue' and 'memory'. Under 'memory', there is a red exclamation mark icon and the message 'No matching constructor for initialization of 'CVector''. The main pane displays code for a constructor:

```
construct(_Up* __p, _Args&&... __args)
{
    ::new((void*)__p) _Up(_VSTD::forward<_Args>
        (__args)...);
}
```

A red box highlights the last line of the constructor body, containing the error message: 'No matching constructor for initialization of 'CVector''. The line numbers 1671, 1672, 1673, and 1674 are visible on the left.

- Pourquoi ?





Non-respect du contrat implicite

- Pour comprendre cette erreur, relisons le prototype de la méthode **resize**

public member function

std::vector::resize

C++98 C++11 ?

```
void resize (size_type n, value_type val = value_type());
```

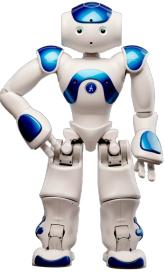
- Il y a un 2^{ème} paramètre de type `value_type`. C'est en fait un `typedef` du paramètre générique `T` de `vector`

Member types

C++98 C++11 ?

| member type | definition |
|-------------------------|---|
| <code>value_type</code> | The first template parameter (<code>T</code>) |

- `resize` avec le 2^{ème} paramètre non spécifié **utilise donc le constructeur par défaut de `T`.** Et ce constructeur n'existe pas pour notre classe `CVector`. Nous n'avons **pas respecté le contrat implicite** de `std::vector`



Contrat explicite : static_assert

- Depuis C++11, on peut expliciter ce contrat avec l'expression

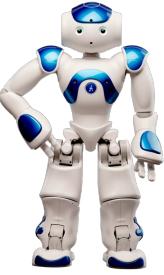
```
static_assert ( bool_constexpr , message )      (C++11)
static_assert ( bool_constexpr )                (C++17)
```

- qui génère une erreur de compilation si l'expression booléenne est fausse.
 - L'expression doit être évaluable à la compilation
 - Elle utilisera souvent les fonctions de la librairie <type_traits> (hors sujet PRG1)

```
template <class T> class C {
    public: static_assert(std::is_default_constructible<T>::value, "Bad T for C<T>");
};

class no_default {
    public: no_default () = delete;
};

int main() {
    C<no_default> c_error; // static assertion failed: Bad T for C<T>
}
```

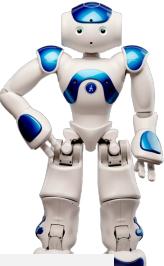


Contrat explicite : concept

- Via des extensions depuis C++14 et dans les standards depuis C++20, on peut maintenant formaliser explicitement ces contrats via la notion de **concept**.

(Wiki) : **Concepts** are an extension to the templates feature provided by the C++ programming language. Concepts are named Boolean predicates on template parameters, evaluated at compile time. A concept may be associated with a template (class template, function template, or member function of a class template), in which case it serves as a constraint: it limits the set of arguments that are accepted as template parameters.

- Cette notion sort cependant du cadre de PRG1 et ne sera donc pas approfondie ici.
- Un exemple de mise en œuvre de cette notion de concept est toutefois proposé sur les slides suivant.



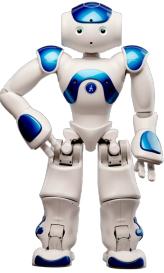
Concept utilisant <type_traits>

```
#include <type_traits>

template <typename T>
concept entier = is_integral<T>::value;

template <entier T> // le nom du concept remplace typename
int dernier_chiffre(T t) {
    return t % 10;
}

int main() {
    cout << dernier_chiffre(2024); // ok, affiche 4
    cout << dernier_chiffre('A'); // ok, affiche 5
    cout << dernier_chiffre(3.14); // Compilation: error: no matching
                                    // function for call to 'dernier_chiffre'
}
```



Concept utilisant <concepts>

- Mais le concept entier est déjà défini dans la librairie <concepts> sous le nom std::integral

```
#include <concepts>

template <integral T>      // std::integral est un concept défini dans concepts
int dernier_chiffre(T t) {
    return t % 10;
}

int main() {
    cout << dernier_chiffre(2024);    // ok, affiche 4
    cout << dernier_chiffre('A');    // ok, affiche 5
    cout << dernier_chiffre(3.14);   // Compilation: error: no matching
                                    // function for call to 'dernier_chiffre'
}
```



Concept utilisant requires

```
template <typename T>
concept Multipliable = requires (T a, T b) {
    a * b;
};

template <Multipliable T>
T square(T a) {
    return a * a;
}

int main() {
    cout << square(5);           // OK. Affiche 25
    cout << square('A');        // OK. Affiche 4225 (= 65 * 65)
    cout << square("Hello");    // Compilation: error: no matching function
                                // for call to 'square'
}
```