# PERFECTIONNEMENT EN C++

# Contenu

**Aspect Objet**

- Encapsulation, Polymorphisme
- RAII et Régle du Zero

**Aspect Générique**

- Fonctions et classes génériques
- Fonction lambda
- Manipulation des types et du code

# Contexte

# Modèle de programmation C++

**Un langage multi-paradigmes:**

- Support du style impératif
- Support du style objet
- Support du style générique
- Support du style fonctionnel

# Modèle de programmation C++

**Un langage multi-paradigmes:**

- Support du style **impératif**
  - Le code est une simple liste d'instructions
  - Ses éléments fondamentaux sont les fonctions et les structures de données
- Support du style objet
- Support du style générique
- Support du style fonctionnel

# Modèle de programmation C++

**Un langage multi-paradigmes:**

- Support du style impératif
  - Le code est une simple liste d'instructions
  - Ses éléments fondamentaux sont les fonctions et les structures de données
- Support du style **objet**
  - Le code s'articule autour de la définition d'objets modélisant les éléments du problème
  - Ses éléments fondamentaux sont les classes et les relations entre ces classes
- Support du style générique
- Support du style fonctionnel

# Modèle de programmation C++

**Un langage multi-paradigmes:**

- Support du style impératif
  - Le code est une simple liste d'instructions
  - Ses éléments fondamentaux sont les fonctions et les structures de données
- Support du style objet
  - Le code s'articule autour de la définition d'objets modélisant les éléments du problème
  - Ses éléments fondamentaux sont les classes et les relations entre ces classes
- Support du style **générique**
  - Le code s'écrit en toute indépendance des types concrets
  - Une partie des calculs sont résolus à la compilation
- Support du style fonctionnel

# Modèle de programmation C++

**Un langage multi-paradigmes:**

- Support du style impératif
  - Le code est une simple liste d'instructions
  - Ses éléments fondamentaux sont les fonctions et les structures de données
- Support du style objet
  - Le code s'articule autour de la définition d'objets modélisant les éléments du problème
  - Ses éléments fondamentaux sont les classes et les relations entre ces classes
- Support du style générique
  - Le code s'écrit en toute indépendance des types concrets
  - Une partie des calculs sont résolus à la compilation
- Support du style **fonctionnel**
  - Le code est basé sur l'utilisation de fonctions et de valeurs
  - Il n'y a pas de notion de mémoire ou de structure de contrôle

# Modèle de programmation C++

**Un langage multi-paradigmes:**

- Support du style impératif
- Support du style objet
- Support du style générique
- Support du style fonctionnel

> Certains problèmes ont des solutions plus simples en fonction du style

# Modèle de programmation C++

**Un langage multi-paradigmes:**

- Support du style impératif
- Support du style objet
- Support du style générique
- Support du style fonctionnel

> Certains problèmes ont des solutions plus simples en fonction du style

**Objectifs**

- Vision objet et générique du langage
- Notion de ressources
- Impact sur les choix de design
- Mise en avant des apports de la STD

# Programmation Objet en C++

# Types Structurés - Agrégats

## Forme générale

```
struct nom_type
{
  type_membre1 membre_1;
  type_membre2 membre_2;

  // etc ....

  type_memberN membre_N;
};
```

- Les valeurs contenues dans un agrégat sont des données **membres**
- Le nombre et le type de ces membres sont arbitraires
- Le nom de l'agrégat ou de ses membres sont arbitraires
- Un agrégat peut lui-même contenir un ou plusieurs agrégats

# Types Structurés - Agrégats

## Forme générale

```
// Agrégat simple
struct point3D
{
  double x; // x est un membre de type double de point3D
  double y;
  double z;
};

// Agrégat composite
struct point3D_pondere
{
  point3D point;  // Notez la réutilisation de point3D
  int     poids;  // poids est une donnée supplémentaire
};
```

## Construction

```cpp
#include <iostream>

struct point3D
{
  double x,y, z;
};

int main()
{
  // Initilisation de chacun des membres dans l'ordre de leur définition
  point3D p = { 1.5, -0.654, 2.7 };

  std::cout << p.x << " " << p.y << " " << p.z << "\n";
}
```

```
1.5 -0.654 2.7
```

## Construction

```cpp
#include <iostream>

struct point3D
{
  double x, y, z;
};

int main()
{
  // On peut 'oublier' des membres
  point3D p = { 9.99 };

  std::cout << p.x << " " << p.y << " " << p.z << "\n";
}
```

```
9.99 0 0
```

## Construction

```cpp
#include <iostream>

struct point3D
{
  double x, y, z;
};

int main()
{
  // On peut 'oublier' tout les membres
  point3D p = {};

  std::cout << p.x << " " << p.y << " " << p.z << "\n";
}
```

```
0 0 0
```

## Construction

```cpp
#include <iostream>

struct point3D
{
  double x, y, z;
};

int main()
{
  // Attention à ne pas utiliser de variable non initialisée !
  point3D p;

  std::cout << p.x << " " << p.y << " " << p.z << "\n";
}
```

```
2.07455e-317 2.07444e-317 6.95283e-310
```

## Construction

```cpp
#include <iostream>

// Reprise des définitions de point3D et point3D_pondere

int main()
{
  // L'initialisation des agrégats interne se fait via des {}
  point3D_pondere p{{1,1,1},9};

  std::cout << p.point.x << " " << p.point.y << " " << p.point.z << "\n";
  std::cout << p.poids << "\n";
}
```

```
1 1 1
9
```

## Valeurs par défaut

```cpp
#include <iostream>

struct point3D
{
  double x = 1.;
  double y = 2.;
  double z = 3.;
};

int main()
{
  point3D p;
  std::cout << p.x << " " << p.y << " " << p.z << "\n";
}
```

```
1 2 3
```

# De l'agrégat à la structure de données

## Limitations du modèle agrégat

- La mission d'un agrégat est d'être un ensemble de valeurs
- La cohésion de ses valeurs ne suffit pas à construire une sémantique claire
- **Un agrégat n'a pas d'invariants**

## Notion de classe

- La mission d'une classe est de fournir un service
- Ce service s'appuie potentiellement sur des valeurs
- **Une classe a des invariants qu'elle protège**

# De l'agrégat à la structure de données

**Encapsulation**

- Force le respect de la sémantique de la classe
- Notion de **visibilité**
- Notion de **membres spéciaux**

**Polymorphisme**

- Permet de raffiner le comportement d'une classe
- Sélection dynamique du bon comportement
- Notion d'**héritage**
- Notion de **fonction virtuelle**

# Principe de l'Encapsulation

### Notion d'invariant

- Un invariant est l'ensemble des propriétés mettant en œuvre les données membres d'une classe qui doivent rester vrai tout au long de la durée de vie d'une instance de cette classe.
- Une classe ne respectant plus ces invariants est dite **incohérente**

### Support niveau langage

- Modificateurs d'accès : `public`, `protected`, `private`
- Membres spéciaux : construction, affection, destruction
- Assertion

# Encapsulation - Modificateurs d'accès

### Membres publiques (`public`)

- Accessible depuis le code utilisateur
- Accessible depuis le code des sous-classes
- Accessible depuis le code interne à la classe

### Membres privés (`private`)

- Inaccessible depuis le code utilisateur
- Inaccessible depuis le code des sous-classes
- Accessible depuis le code interne à la classe

# Encapsulation - Modificateurs d'accès

### Cas des `struct`

- Visibilité par défaut: `public`
- Pas d'invariant car le contenu est accessible par tous
- L'important c'est les valeurs

### Cas des `class`

- Visibilité par défaut: `private`
- Des invariants existent et sont protégés par le masquage des membres associés
- L'important c'est le comportement

# Encapsulation - Que choisir ?

**Un membre est `private` si :**

- Il fait parti d'un invariant de classe
- Il ne participe qu'au fonctionnement interne de la classe

**Un membre est `public` si :**

- Il ne participe au maintien d'aucun invariant
- La sémantique de la classe l'impose

**Conclusion**

- Privilégiez au maximum les membres `public`
- Au moindre invariant, utilisez `private`

# Encapsulation - Exemple

## Le type `fraction`

- Une fraction de compose de deux entiers ; le numérateur, le dénumérateur
- Le numérateur est un entier relatif arbitraire
- Le dénumérateur est positif et non nul

```
1  struct fraction
2  {
3    int num, denum;
4  };
5
6  fraction f1{-9,10}; // OK
7  fraction f2{200,0}; // Erreur
```

# Encapsulation - Exemple

## Le type `fraction`

- Une fraction de compose de deux entiers ; le numérateur, le dénumérateur
- Le numérateur est un entier relatif arbitraire
- Le dénumérateur est positif et non nul

```
1  struct fraction
2  {
3    void num(int n)   { num = n; }
4    void denum(int d) { assert(d > 0); denum = d; }
5
6    private:
7    int num_, denum_;
8  };
9
10 fraction f1, f2;
11 f1.num(3); f1.denum(4);   // OK
12 f2.num(-2); f2.denum(0);  // Erreur au runtime
```

# Membres Spéciaux

**Problématique**

- Modifier manuellement un objet post-déclaration est peu pratique
- La durée de vie d'un objet en C++ est bornée
- Si cet objet contient des invariants, ils doivent être vérifiés à la création de l'objet
- Si cet objet contient des invariants, ils doivent être vérifiés à la destruction de l'objet

**Solution : Les Membres Spéciaux**

- Les constructeurs
- L'opérateur d'affectation
- Le destructeur

# Constructeurs

## Problématique

- La construction directe d'une structure en remplissant ses membres est limitée
- Il se peut qu'un type puisse être construit à partir de différentes valeurs
- Il se peut qu'un ou plusieurs membres doivent être initialisés en dépendant des autres
- Il peut être nécessaire de vérifier certaines propriétés sur les valeurs utilisées

## Notion de constructeur

- Un constructeur est un **membre spécial** qui gère l'initialisation d'une instance de structure
- Son identifiant est le nom de la structure
- Certains constructeurs ont un rôle particulier dans le cycle de vie d'une instance
- Si au moins un constructeur est défini, la structure perd sa propriété d'agrégat

# Constructeurs

## Constructeur arbitraire

```
1  struct fraction
2  {
3    private: int num, denum;
4
5    public:
6    //  Construit une fraction à partir d'un seul entier
7    //  Le dénumérateur vaut alors 1 par défaut.
8    fraction(int n) : num(n), denum(1) {}
9
10   //  Construit une fraction à partir d'un couple d'entier
11   fraction(int n, int d) : num{n}, denum{d} { assert(d > 0); }
12 };
```

# Constructeurs

## Constructeur arbitraire

```
1  struct fraction
2  {
3    private: int num, denum;
4
5    public:
6    //  Construit une fraction à partir d'un seul entier
7    //  Le dénumérateur vaut alors 1 par défaut.
8    //
9    //  Les membres de fraction sont construits dans
10   //  la liste d'initialisation définit ici
11   //              |
12   //              v
13   fraction(int n) : num(n), denum(1) {}
14
15   fraction(int n, int d) : num{n}, denum{d} { assert(d > 0); }
16 };
```

# Constructeurs

## Constructeur arbitraire

```
1  struct fraction
2  {
3    private: int num, denum;
4
5    public:
6    //  Construit une fraction à partir d'un seul entier
7    //  Le dénumérateur vaut alors 1 par défaut.
8    //
9    //  Chaque membre apparait dans l'ordre de leur définition
10   //  dans la liste des membres de la structure.
11   //                |       |
12   //                v       v
13   fraction(int n) : num(n), denum(1) {}
14
15   fraction(int n, int d) : num{n}, denum{d} { assert(d > 0); }
16 };
```

# Constructeurs

## Constructeur arbitraire

```
1  struct fraction
2  {
3    private: int num, denum;
4
5    public:
6    //  Construit une fraction à partir d'un seul entier
7    //  Le dénumérateur vaut alors 1 par défaut.
8    //
9    //  Chaque membre appelle son propre constructeur
10   //  via ( ) ou { } et en passant les valeurs nécessaires.
11   //                    |         |
12   //                    v         v
13   fraction(int n) : num(n), denum(1) {}
14
15   fraction(int n, int d) : num{n}, denum{d} { assert(d > 0); }
16 };
```

## Constructeur arbitraire

```
1  struct fraction
2  {
3    fraction(int n)        : num(n), denum(1) {}
4    fraction(int n, int d) : num{n}, denum{d} { assert(d > 0); }
5    int numerator()   const { return num;   }
6    int denumerator() const { return denum; }
7    private: int num, denum;
8  };
9
10 int main()
11 {
12   fraction f(7), g(7,4);
13   std::cout << f.numerator() << " / " << f.denumerator() << "\n";
14   std::cout << g.numerator() << " / " << g.denumerator() << "\n";
15 }
```

```
7 / 1
7 / 4
```

# Constructeurs

## Constructeur par défaut

- Il s'agit d'un constructeur ne prenant aucun paramètre de manière directe ou indirecte
- Il garantie la valeur d'une instance construite sans valeur d'initialisation
- En général, il est automatiquement fournit par le compilateur
- Il est nécessaire du moment ou il existe au moins un constructeur arbitraire

```
1  struct fraction
2  {
3    fraction()             : num(0), denum(1) {}  // Constructeur par défaut
4    fraction(int n)        : num(n), denum(1) {}
5    fraction(int n, int d) : num{n}, denum{d} { assert(d > 0); }
6
7    private: int num, denum;
8  };
9
10 fraction f; // f = {0,1}
```

# Constructeurs

## Constructeur par défaut

- Il s'agit d'un constructeur ne prenant aucun paramètre de manière directe ou indirecte
- Il garantie la valeur d'une instance construite sans valeur d'initialisation
- En général, il est automatiquement fournit par le compilateur
- Il est nécessaire du moment ou il existe au moins un constructeur arbitraire

```
1  struct fraction
2  {
3    // Ce constructeur n'est pas directement un constructeur par défaut
4    // mais il permet l'appel de fraction f;
5    fraction(int n = 0, int d = 1) : num{n}, denum{d} { assert(d > 0); }
6
7    private: int num, denum;
8  };
9
10 fraction f; // f = {0,1}
```

## Constructeur par défaut - Le piège du *Most Vexing Parse*

```
1  struct fraction
2  {
3    fraction(int n = 0, int d = 1) : num{n}, denum{d} { assert(d > 0); }
4    private: int num, denum;
5  };
6
7  fraction g();
8
9  std::cout << g.numerator() << "\n";
```

# Constructeurs

## Constructeur par défaut - Le piège du *Most Vexing Parse*

```
1  struct fraction
2  {
3    fraction(int n = 0, int d = 1) : num{n}, denum{d} { assert(d > 0); }
4    private: int num, denum;
5  };
6
7  fraction g();
8
9  // error: request for member 'numerator()' in 'g', which is of non-class type 'fraction()'
10 std::cout << g.numerator() << "\n";
```

# Constructeurs

## Constructeur par défaut - Le piège du *Most Vexing Parse*

```
1  struct fraction
2  {
3    fraction(int n = 0, int d = 1) : num{n}, denum{d} { assert(d > 0); }
4    private: int num, denum;
5  };
6
7  fraction g();
8
9  // error: request for member 'numerator()' in 'g', which is of non-class type 'fraction()'
10 std::cout << g.numerator() << "\n";
```

- Pour des raisons de rétro-compatibilité, `T f()` n'appelle pas le constructeur par défaut
- Cette notation définit un prototype de fonction locale
- Privilégiez donc la notation `T f{}` pour expliciter l'appel au constructeur par défaut

# Constructeurs

## Constructeur par défaut - Le piège du *Most Vexing Parse*

```
1  struct fraction
2  {
3    fraction(int n = 0, int d = 1) : num{n}, denum{d} { assert(d > 0); }
4    private: int num, denum;
5  };
6
7  fraction g{};
8
9  // OK
10 std::cout << g.numerator() << "\n";
```

- Pour des raisons de rétro-compatibilité, `T f()` n'appelle pas le constructeur par défaut
- Cette notation définit un prototype de fonction locale
- Privilégiez donc la notation `T f{}` pour expliciter l'appel au constructeur par défaut

# Constructeurs

## Constructeur de copie

- Il s'agit d'un constructeur recopiant un autre objet
- En général, il est automatiquement fournit par le compilateur
- Certains usages assez rares peuvent nécessiter sa définition

```
1  struct fraction
2  {
3    fraction(int n = 0, int d = 1) : num{n}, denum{d} { assert(d>0); }
4    fraction(fraction const& other) : num(other.num), denum(other.denum) {}
5
6    private: int num, denum;
7  };
8
9  fraction f{4,7};
10 fraction g{f};
```

# Construction par délégation

## Problématique

```
1  struct fraction
2  {
3    fraction(int n, int d) : num{n}, denum{d} { assert(d>0); }
4    fraction(int n)        : num(n), denum(1) {}
5    fraction()             : num(0), denum(1) {}
6    private: int num, denum;
7  };
```

- Certaines structures ont des constructeurs très proches en terme de code
- Il est envisageable de vouloir factoriser ces codes
- Nécessité de pouvoir appeler un constructeur depuis un autre constructeur

# Construction par délégation

## Problématique

```
1  struct fraction
2  {
3    fraction(int n, int d) : num{n}, denum{d} { assert(d>0); }  // Constructeur principal
4    fraction(int n)        : fraction(n,1)    {}  // Délégation
5    fraction()             : fraction(0,1)    {}  // Délégation
6    private: int num, denum;
7
8    // fraction(double n) : fraction{n*1000}, denum(1000) {} // Erreur
9  };
```

- Construire par délégation permet d'appeler un constructeur dans une liste d'initialisation
- Réduit la duplication de code entre les constructeurs
- On ne peut pas mélanger délégation et initialisation pour un même constructeur
- **Attention à ne pas créer de cycle entre les constructeurs**

# Constructeurs explicites

## Problématique

```
1  struct fraction
2  {
3    fraction(int n, int d) : num{n}, denum{d} { assert(d>0); }
4    fraction(int n)        : fraction(n,1)    {}
5    fraction()             : fraction(0,1)    {}
6    private: int num, denum;
7  };
8
9  void f(fraction f);
10
11 f( fraction{4,7} ); // OK
```

# Constructeurs explicites

## Problématique

```
1  struct fraction
2  {
3    fraction(int n, int d) : num{n}, denum{d} { assert(d>0); }
4    fraction(int n)        : fraction(n,1)    {}
5    fraction()             : fraction(0,1)    {}
6    private: int num, denum;
7  };
8
9  void f(fraction f);
10
11 f( fraction{4,7} ); // OK
12 f( 9 );             // Compile alors que f(int) n'existe pas ???
```

# Constructeurs explicites

## Problématique

```
1  struct fraction
2  {
3    fraction(int n, int d) : num{n}, denum{d} { assert(d>0); }
4    fraction(int n)        : fraction(n,1)    {}
5    fraction()             : fraction(0,1)    {}
6    private: int num, denum;
7  };
8
9  void f(fraction f);
10
11 f( fraction{4,7} ); // OK
12 f( 9 );             // Compile alors que f(int) n'existe pas ???
```

- L'appel à `f(9)` est compilé sans erreur contrairement à notre intuition
- Cela est du au fait que le compilateur déduit qu'il peut convertir 9 en `fraction`
- Cette conversion est possible car le constructeur `fraction(int n)` existe

# Constructeurs explicites

## Le mot-clé `explicit`

```
1  struct fraction
2  {
3              fraction(int n, int d) : num{n}, denum{d} { assert(d>0); }
4    explicit fraction(int n)        : fraction(n,1)    {}
5              fraction()            : fraction(0,1)    {}
6    private: int num, denum;
7  };
8
9  void f(fraction f);
10
11 f( fraction{4,7} ); // OK
12 f( 9 );             // OK - Ne compile plus
13 f( fraction{9} );   // OK
```

- Un constructeur marqué `explicit` ne peut etre utilisé dans une conversion implicite
- La conversion vers `fraction` doit être effectué consciemment

# Conclusion sur les Constructeurs

**Bonnes pratiques - Constructeurs spéciaux**

- Le plus souvent le constructeur de copie fourni par défaut est suffisant.
- Le plus souvent le constructeur par défaut fourni par défaut est suffisant.
- Évitez les comportements complexes pour ces dernier.

**Bonnes pratiques - Constructeurs arbitraires**

- Pas plus de constructeurs que nécessaires.
- La liste d'initialisation est à utiliser le plus possible.
- Les constructeurs à un seul paramètre doivent être `explicit` le plus souvent possible.
- Évitez les délégations de constructeurs complexes à comprendre

# Surcharge d'opérateurs

## Opérateur d'affectation

- `T& T::operator=(V v)` est un opérateur permettant d'affecter une valeur à une instance de structure

```
struct fraction
{
  // ...

  fraction& operator=(int v)
  {
    num = v;
    denum = 1;
    return *this;
  }
};
```

# Opérateur d'affection

## Définition

- `T& T::operator=(T const&)` est l'opérateur d'affection canonique
- Comme les constructeurs, il est considéré comme un membre spécial.

```
1  struct fraction
2  {
3    // ...
4
5    // Ce prototype est considéré comme l'opérateur d'affectation canonique
6    fraction& operator=(fraction const& v)
7    {
8      // Il est souvent inutile de le spécifier car l'operator=
9      // généré par le compilateur procéde à cette recopie automatiquement
10     num   = v.num;
11     denum = v.denum;
12     return *this;
13   }
14 };
```

# Destructeur

## Principe

```
struct fraction
{
  // ...

  // Ce prototype est le destructeur canonique
  ~fraction()
  {
    // Il est inutile de le déclarer 99.99% du temps
    // car le destructeur généré par le compilateur fait
    // souvent le travail.

    // Nous verrons plus tard quand ce membre DOIT etre
    // spécifié.
  }
};
```

# Principe du Polymorphisme

## Définition

- Le **Polymorphisme** est la capacité de modifier le comportement d'une fonction en fonction du ou des types de ces paramètres
- Il existe plusieurs types de **Polymorphisme** : statique, dynamique, ad hoc, universel.

## Le Polymorphisme en Programmation Objet

- Permet de raffiner le comportement d'une classe en la **sous-classant**
- La notion d'**héritage** permet d'expliciter ces relations
- Mise en avant d'une notion de **substituabilité**

# Le Polymorphisme en C++

## Relation d'héritage

- C++ supporte l'héritage public et privé
- Héritage public : mise en place du polymorphisme
- Héritage privé : factorisation de code

## Notion de virtualité

- C++ supporte la notion d'appel virtuel et d'appel abstrait
- Contrairement à d'autres langages, les fonctions membres en C++ ne sont pas polymorphe par défaut
- Le mot-clé `virtual` permet d'indiquer les membres participant à l'aspect polymorphe d'une classe

# Polymorphisme - Mise en œuvre

## Éléments de syntaxe pour l'héritage

```
1  class point
2  {
3    public:
4    point(int dx = 0, int dy = 0) :x(dx), y(dy) {}
5
6    void display() const
7    {
8      std::cout << "{" << x << ", " << y << "}";
9    }
10
11   int x,y;
12 };
```

# Polymorphisme - Mise en œuvre

## Éléments de syntaxe pour l'héritage

```
1  // class A : public B indique que A hérite de B publiquement
2  //                   v
3  class weighted_point : public point
4  {
5    public:
6    weighted_point ( int dx = 0, int dy = 0, float dm = 1.f) : point(dx,dy), m(dm)
7    // Utilisation du constructeur de la classe de base ------------^
8    {}
9
10   void display() const
11   {
12     point::display(); // Accès au membre de la classe de base
13     std::cout << "@[" << m << "]";
14   }
15
16   float m;
17 };
```

# Polymorphisme - Mise en oeuvre

## Éléments de syntaxe pour l'héritage

```
1  int main()
2  {
3    point           x{7, 9};
4    weighted_point  y{8,3,25.28f};
5
6    x.display();
7    std::cout << "\n";
8    y.display();
9  }
```

```
{7, 9}
{8, 3}@[25.28]
```

# Polymorphisme - Mise en oeuvre

## Activation du polymorphisme

```
1  void display( point const& p )
2  {
3    p.display();
4    std::cout << "\n";
5  }
6
7  int main()
8  {
9    point           x{7, 9};
10   weighted_point  y{8,3,25.28f};
11
12   display(x);
13   display(y);
14 }
```

```
{7, 9}
{8, 3}
```

# Polymorphisme - Mise en œuvre

## Activation du polymorphisme

- Par défaut, les fonctions membres ne sont pas polymorphes
- Le mot-clé `virtual` doit être utilisé pour activer se comportement
- Une classe avec au moins un fonction virtuelle est dite polymorphe.

## Contrôle fin du polymorphisme

- `override`
- `final`

# Polymorphisme - Mise en oeuvre

## Activation du polymorphisme

```
1  class point
2  {
3    public:
4    point(int dx = 0, int dy = 0) :x(dx), y(dy) {}
5
6    // display() const est désormais polymorphe
7    virtual void display() const
8    {
9      std::cout << "{" << x << ", " << y << "}";
10   }
11
12   int x,y;
13 };
```

# Polymorphisme - Mise en oeuvre

## Activation du polymorphisme

```
1  // weigthed_point hérite de point, elle est aussi polymorphe
2  class weighted_point : public point
3  {
4    public:
5    weighted_point ( int dx = 0, int dy = 0, float dm = 1.f) : point(dx,dy), m(dm)
6    {}
7
8    // override indique que l'on a conscience du caractère surchargée de display()
9    void display() const override
10   {
11     point::display(); // Acces au membre de la classe de base
12     std::cout << "@[" << m << "]";
13   }
14
15   float m;
16 };
```

# Polymorphisme - Mise en oeuvre

## Activation du polymorphisme

```
1  void display( point const& p )
2  {
3    p.display();
4    std::cout << "\n";
5  }
6
7  int main()
8  {
9    point           x{7, 9};
10   weighted_point  y{8,3,25.28f};
11
12   display(x);
13   display(y);
14 }
```

```
{7, 9}
{8, 3}@[25.28]
```

# Polymorphisme - Mise en œuvre

## Fonction abstraite

- Fonction virtuelle sans implémentation dans la classe de base
- Elle fournit un contrat sans comportement par défaut
- Force ses sous-classes à l'implémenter sous peine d'erreur
- Une classe possédant au moins une fonction abstraite est dite abstraite

## Conséquences

- Une classe abstraite n'est pas instantiable
- On nomme **interface** ces classes sans implémentation
- Elles ne font que définir un **contrat** avec l'utilisateur

# Polymorphisme - Mise en œuvre

## Activation du polymorphisme

```
1  struct displayable
2  {
3    virtual ~displayable() {}
4    virtual void display() const = 0;
5  };
6
7  class point          : public displayable { /* ... */ };
8  class weigthed_point : public point { /* ... */ };
9
10 void display( displayable const& p )
11 {
12   p.display();
13   std::cout << "\n";
14 }
```

# SOLID - Liskov Substitution Principle

> *Let q(x) be a property provable about objects of x of type T. Then q(y) should be provable for objects y of type S where S is a subtype of T.*
>
> — Barbara Liskov, 1999

**En pratique**

- Un code demandant explicitement le type concret d'une classe polymorphe pour décider de son implémentation est défectueux
- Les classes dérivées ne peuvent que relâcher les préconditions de leur fonctions
- Les classes dérivées ne peuvent que renforcer les post-conditions de leur fonctions
- Les invariants de la classe de base doivent être vérifiées par les classes dérivées

# SOLID - Liskov Substitution Principle

## Code original

```
struct rectangle
{
  virtual void set_h(double hh) { h = hh; }
  virtual void set_w(double ww) { w = ww; }
  double area() const { return h*w; }

  private:
  double h,w;
};

struct square : public rectangle
{
  void set_h(double hh) override { rectangle::set_h(hh); rectangle::set_w(hh); }
  void set_w(double ww) override { rectangle::set_h(ww); rectangle::set_w(ww); }
};
```

# SOLID - Liskov Substitution Principle

## Code original

```
1  void f(rectangle& r)
2  {
3    r.set_h(20);
4    r.set_w(10);
5
6    assert( r.area() == 200 );
7  }
8
9  int main()
10 {
11   rectangle r;
12   square    s;
13
14   f(r);
15   f(s);
16 }
```

# SOLID - Liskov Substitution Principle

## Code reformulé par LSP

```
struct compute_area { virtual double area() const = 0; };

struct rectangle : public compute_area
{
  rectangle(double hh, double ww) : h(hh), w(ww) {}
  double area() const override { return h*w; }
  private:
  double h,w;
};

struct square : public compute_area
{
  square(double ee) : edge(e) {}
  double area() const override { return edge*edge; }
  private:
  double edge;
};
```

# Gestion des Ressources

# lvalue vs rvalue

### Notion de catégorie de valeurs

- En plus de son type, une variable est classifiée selon sa catégorie de valeur
- La catégorie de valeur indique comment la durée de vie d'une valeur est gérée

### Notion de `lvalue`

- Une `lvalue` est une valeur avec un nom, un identifiant
- La durée de vie d'une `lvalue` est la portée courante

### Notion de `rvalue`

- Une `rvalue` est une valeur sans identifiant ou littérale
- La durée de vie d'une `rvalue` est l'expression courante

# lvalue vs rvalue

**Pourquoi discriminer `lvalues` et `rvalues` ?**

- Expliciter les possibilités d'optimisations
- Simplifier la définition de certaines fonctions
- Introduire au sein du langage la notion de durée de vie

**Notion de *rvalue reference***

- `int&` : référence vers une **lvalue** de type `int`
- `int const&` : référence vers une **lvalue** de type `int const`
- `int&&` : référence vers une **rvalue** de type `int`

# lvalue vs rvalue

## Détection et classification des rvalue/lvalue

```cpp
#include <iostream>

void f(int& )       { std::cout << "lvalue\n"; }
void f(int const& ) { std::cout << "lvalue constante\n"; }
void f(int&&)       { std::cout << "rvalue\n"; }
int g() { return 4; }

int main()
{
  int i = 1;
  int const j = 4;

  f(i);
  f(j);
  f(3);
  f(i+j);
  f(g());
}
```

# lvalue vs rvalue

## Détection et classification des rvalue/lvalue

```cpp
#include <iostream>

void f(int& )       { std::cout << "lvalue\n"; }
void f(int const& ) { std::cout << "lvalue constante\n"; }
void f(int&&)       { std::cout << "rvalue\n"; }
int g() { return 4; }

int main()
{
  int i = 1;
  int const j = 4;

  f(i);   //  Affiche `lvalue`
  f(j);   //  Affiche `lvalue constante`
  f(3);   //  Affiche `rvalue`
  f(i+j); //  Affiche `rvalue`
  f(g()); //  Affiche `rvalue`
}
```

# lvalue vs rvalue

## Détection et classification des rvalue/lvalue

```cpp
#include <iostream>

void f(int& )       { std::cout << "lvalue\n"; }
void f(int const& ) { std::cout << "lvalue constante\n"; }
void f(int&&)       { std::cout << "rvalue\n"; }
void h(int&& i)     { std::cout << "!!"; f(i); }

int main()
{
  h(4);
}
```

# lvalue vs rvalue

## Détection et classification des rvalue/lvalue

```cpp
#include <iostream>

void f(int& )       { std::cout << "lvalue\n"; }
void f(int const& ) { std::cout << "lvalue constante\n"; }
void f(int&&)       { std::cout << "rvalue\n"; }
void h(int&& i)     { std::cout << "!!"; f(i); }

int main()
{
  h(4); //  Affiche 'lvalue'
}
```

- `i` a pour type `int&&`
- Mais il a un identifiant, sa catégorie de valeur est donc **lvalue**

# lvalue vs rvalue

## Détection et classification des rvalue/lvalue

```cpp
#include <iostream>

void f(int& )       { std::cout << "lvalue\n"; }
void f(int const& ) { std::cout << "lvalue constante\n"; }
void f(int&&)       { std::cout << "rvalue\n"; }
void h(int&& i)     { std::cout << "!!"; f(std::move(i)); }

int main()
{
  h(4); //  Affiche 'rvalue'
}
```

- `i` a pour type `int&&`
- `std::move(i)` renvoie `i` sans copie
- `f` reçoit le temporaire valant i, c'est bien une **rvalue**

# lvalue vs rvalue

## Mise en oeuvre

- Copier un objet contenant des ressources est coûteux
- Copier un objet temporaire contenant des ressources est encore plus coûteux car il faut construire puis copier le temporaire
- Peut-on “recycler” ce temporaire ?

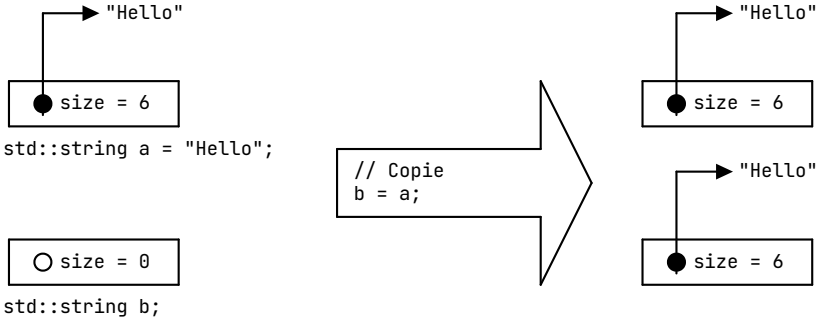
## Sémantique de transfert

- Un objet temporaire n'a pas d'identité, c'est donc une **rvalue**
- Comme sa durée de vie est limitée, on peut extraire sa ressource et l'affecter à une autre valeur
- Cette sémantique est supportée par tous les composants du standards

# Sémantique de transfert

## Cas de la copie

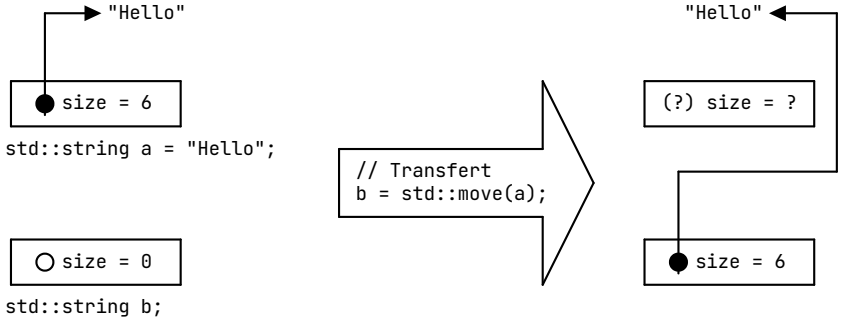
- Duplication des données et des ressources de la source vers la destination
- L'objet original reste inchangé



# Sémantique de transfert

## Cas du transfert

- Les données et ressources originales sont cédées à l'objet destination
- L'objet original passe dans un état incohérent



# Sémantique de Valeur

**Définition**

Une classe a une sémantique de valeur si deux instances de cette située à des différentes, mais au contenu identiques, sont considérés égales.

**Forme classique**

- Possède un opérateur d'affectation
- Est comparable avec == et, optionnellement, <
- Peut redéfinir d'autres opérateurs arithmétiques
- Ne peut pas être utilisé comme classe de base

# Sémantique d'Entité

**Définition**

Une classe a une sémantique d'entité si toutes les instances de cette classe sont nécessairement deux à deux distinctes, même si toutes leurs données membres sont égaux.

Elle modélise un concept d'identité : chaque objet représente un individu unique.

**Forme classique**

- Les copies se font via une fonction membre explicite
- Ne se compare ni avec `==` ni `<`
- Ne redéfinit que rarement d'autres opérateurs arithmétiques
- Est le candidat classique pour une classe de base

# Sémantique de Ressource

**Définition**

Une classe a une sémantique de ressource si sa seule responsabilité consiste à maintenir et interagir avec une ressource système limitée nécessitant une capture et/ou une libération explicite.

**Forme classique**

- Elle peut être copiable ou uniquement transférable
- Elle implémente la totalité de ses membres spéciaux
- Permet d'exploiter le principe de la RAII

# Principe de la RAII

## Objectifs

- Assurer la sûreté de la gestion des ressources
- Minimiser la gestion manuelle des ressources
- Simplifier les interactions avec les exceptions

## Principes

- Les ressources sont acquises dans le constructeur
- Les ressources sont libérées dans le destructeur
- La portée de l'objet borne la durée de vie de la ressource

# RAII : Mise en situation

## Un chronomètre RAII

```cpp
#include <chrono>

class stopwatch
{
  public:
  using time_t = std::chrono::duration<double, std::milli>;

  stopwatch(time_t& e)  : now( std::chrono::steady_clock::now()), elapsed(e) {}
  ~stopwatch()
  {
    auto then = std::chrono::steady_clock::now();
    elapsed = time_t(then - now);
  }

  private:
  std::chrono::time_point<std::chrono::steady_clock> now;
  std::chrono::duration<double, std::milli>& elapsed;
};
```

# RAII : Mise en situation

## Un chronomètre RAII

```cpp
#include <chrono>

class stopwatch { /* .. */ };

int main()
{
  stopwatch::time_t elapsed;

  {
    // Construction et capture de elapsed
    stopwatch here(elapsed);
    usleep(1337);
  } // destruction et calcul de la durée

  std::cout << elapsed.count() << "\n";
}
```

# RAII : Pointeurs à sémantique riche

## `std::shared_ptr`

```cpp
#include <memory>
#include <iostream>

int main()
{
  auto p1 = std::make_shared<std::string>(4,'-'); // Acquisition de la ressource
  std::cout << *p1 << "\n";

  {
    std::shared_ptr<std::string> p2 = p1; // Partage de la ressource
    p2→append(7,'!');
  }                                       //  Relâchement de la ressource par p2

  auto p3 = p1;   // Partage de la ressource
  p1.reset();     // Relâchement de la ressource par p1
  std::cout << *p3 << "\n";
}  //  Relâchement de la ressource par p3 et nettoyage
```

# RAII : Pointeurs à sémantique riche

## std::unique_ptr

```cpp
#include <memory>
#include <iostream>

int main()
{
  auto p1 = std::make_unique<std::string>(4,'-'); // Acquisition de la ressource
  std::cout << *p1 << "\n";

  // Code incorrect à la compilation - unique_ptr n'est pas copiable
  // std::unique_ptr<std::string> p2 = p1;

  auto p3 = std::move(p1);   // Transfert de la ressource
  p3->append(7,'!');
  std::cout << *p3 << "\n";
}  //  Relâchement de la ressource par p3 et nettoyage
```

# La règle des Trois, la règle des Cinq

## La Règle des Trois

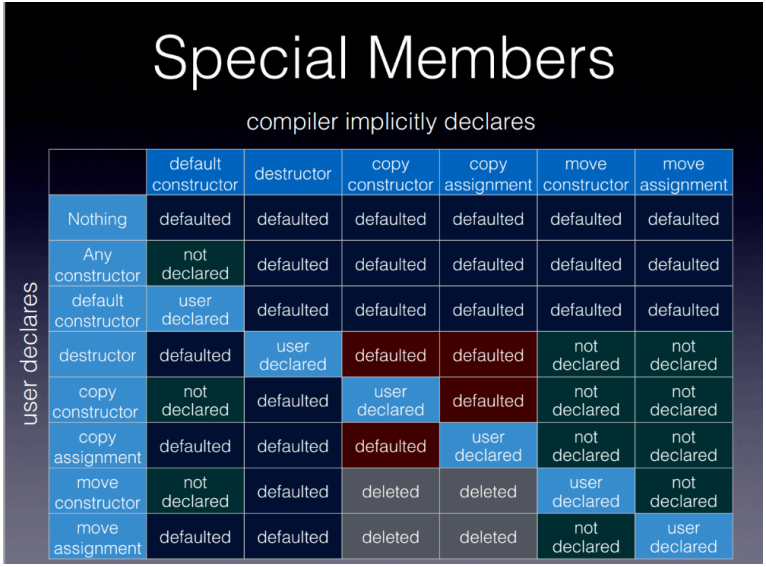
“If a class manually manages a ressource (like memory), the destructor, copy assignment operator and copy constructor have to be implemented. If either one of them is implemented, the other two should also be implemented”.

## La Règle des Cinq

“If a class has an user-defined destructor, user-defined copy constructor or user defined copy assignment operator, the move constructor and the move assignment operator have to be also implemented to realize the move semantics”.

La probabilité d'oublier un de ces éléments ou de le sur-spécifier est grand.

# La règle des Trois, la règle des Cinq




| user declares | default constructor | destructor | copy constructor | copy assignment | move constructor | move assignment |
|---|---|---|---|---|---|---|
| Nothing | defaulted | defaulted | defaulted | defaulted | defaulted | defaulted |
| Any constructor | not declared | defaulted | defaulted | defaulted | defaulted | defaulted |
| default constructor | user declared | defaulted | defaulted | defaulted | defaulted | defaulted |
| destructor | defaulted | user declared | defaulted | defaulted | not declared | not declared |
| copy constructor | not declared | defaulted | user declared | defaulted | not declared | not declared |
| copy assignment | defaulted | defaulted | defaulted | user declared | not declared | not declared |
| move constructor | not declared | defaulted | deleted | deleted | user declared | not declared |
| move assignment | defaulted | defaulted | deleted | deleted | not declared | user declared |

# La règle du Zéro

> *Classes that declare customs destructors, copy/move constructors, copy/move assignment operators should deals exclusively with ownership. Other classes should not declare custom special members functions.*

**Conséquence**

- Privilégiez l'utilisation des objets RAII du standard au lieu des constructions C
- `char*` → `std::string`
- `malloc`, `free` → `std::vector`
- N'utilisez des pointeurs nus que pour de l'observation
- Utilisez les pointeurs à sémantique riche pour clarifier la propriété des ressources

# Programmation Générique en C++

# Fonctions génériques

## Motivation

- Large quantité de surcharges pour une fonction donnée
- Code extrêmement répétitif

```
1  double minimum(double a, double b) { return a<b ? a : b; }
2  float  minimum(float  a, float  b) { return a<b ? a : b; }
3  int    minimum(int    a, int    b) { return a<b ? a : b; }
4  char   minimum(char   a, char   b) { return a<b ? a : b; }
5  short  minimum(short  a, short  b) { return a<b ? a : b; }
```

Ce type de code ne passe pas à l'échelle et est potentiellement une source d'erreur complexe à résoudre *a posteriori*.

# Fonctions génériques

## Objectifs

- Fournir une syntaxe permettant d'exprimer l'universalité d'une fonction surchargée
- Limiter l'impact sur le code écrit, maintenu et généré

## Solutions

- Notion de **patrons de fonctions**
- Syntaxe dite des *templates* de fonctions
- On parle ainsi de **Polymorphisme Statique**

```
1  template<typename Type>
2  Type  minimum(Type  a, Type  b)
3  {
4    return a<b ? a : b;
5  }
```

# Fonctions génériques

## Principes

- Remplacement des types par des types paramétriques
- L'appel de la fonction force le compilateur à déduire le type effectif de ses paramètres en analysant le type des valeurs passées en paramètres.
- Le code exact est régénéré à partir de cette déduction
- Si cette déduction est impossible, il y a erreur de compilation

```
1  template<typename Type>
2  Type  minimum(Type  a, Type  b)
3  {
4    return a<b ? a : b;
5  }
6
7  // Type est déduit comme `int`
8  auto x = minimum(4, 5);
9
10 // Type est déduit comme `float`
11 auto y = minimum(4.6f, -0.5f);
12
13 // Cas incorrect:
14 // error: no matching function
15 // for call to 'minimum(double, int)'
16
17 auto z = minimum(4.6, 5);
```

# Fonctions génériques

**Paramètre *template***

- Introduit par la notation `template< ... >`
- Contient au moins un paramètre de type
- Ces paramètres ont un identifiant unique introduit par le mot-clé `typename`

```
1  //       1er param. template
2  //                 |      2eme param. template
3  //                 |             |    3e param. template
4  //                 |             |             |
5  //                 v             v             v
6  template<typename Src, typename Dst, typename Size>
7  void copy(Src const& src, Dst& dst, Size qty)
8  {
9    // Utilisation de Size comme type de l'index i
10   for(Size i=0;i<qty;++i)
11     dst[i] = src[i];
12 }
```

# Inférence du type de retour

### Principes

- Le type de retour des fonctions génériques peut être complexe à exprimer voire impossible à déterminer par le développeur
- Le compilateur a toutes les informations nécessaires à sa détermination exacte
- On introduit une syntaxe pour laisser la main au compilateur pour cette tâche

### Eléments de syntaxe

- Mot-clé `auto`
- Mot-clé `decltype`

### Modes de fonctionnement

- Explicite : calcul retardé du type de retour
- Implicite : calcul entièrement délégué au compilateur

# Inférence du type de retour

**Exemple:**

```
template<typename T1, typename T2>
/* QUID ??? */ addition(T1 a, T2 b)
{
  return a + b;
}
```

**Cas explicite:**

```
  template<typename T1, typename T2>
// auto indique que le type va être calculé plus tard
// |
// v
  auto  addition(T1 a, T2 b)
        → decltype(a+b)  // <--- decltype évalue le type de son paramètre
  {
    return a + b;
  }
```

# Inférence du type de retour

**Exemple:**

```
template<typename T1, typename T2>
/* QUID ??? */ addition(T1 a, T2 b)
{
  return a + b;
}
```

**Cas implicite:**

```
  template<typename T1, typename T2>
// auto indique que le type va être calculé par le compilateur
// |
// v
  auto  addition(T1 a, T2 b)
  {
    // Le compilateur évalue le type de l'expression renvoyée par return
    return a + b;
  }
```

# Inférence du type de retour

## Cas du retour parfait:

```
template<typename T>
// auto renvoiet une valeur, le retour est donc une copie de a ou de b
auto opti_min(T const& a, T const& b)
{
  return (a<b) ? a : b;
}

template<typename T>
// decltype(auto) conserve les qualificateurs, on renvoi ici une référence vers a ou b
decltype(auto) opti_min(T const& a, T const& b)
{
  return (a<b) ? a : b;
}
```

# Fonctions anonymes

## Limitations

- Nécessité d'utiliser une fonction pour spécifier le comportement de l'algorithme
- Problématique de la multiplication de fonctions utilitaires
- Problématique des fonctions utilitaires templates

## Solutions

- Fonctions anonymes : équivalent fonctionnel des variables temporaires
- Augmente la localité du code et les performances
- Possibilité d'utiliser des fonctions anonymes comme retour de fonction
- Simplifie l'écriture des fonctions templates dites trampolines

# Fonctions anonymes

## Mise en place

```
bool process_buffer(std::vector<int> const& mems)
{
  return std::any_of( mems.begin(), mems.end()
                    , [](int i)
                      {
                        return i > 3 && i < 10;
                      }
                    );
}
```

# Fonctions anonymes

## Mise en place

```
1  bool process_buffer(std::vector<int> const& mems)
2  {
3    return std::any_of( mems.begin(), mems.end()
4                      , [](int i)
5                        // Corps de la fonction
6                        {
7                          return  i > 3 && i < 10;
8                        }
9                      );
10 }
```

# Fonctions anonymes

## Mise en place

```
bool process_buffer(std::vector<int> const& mems)
{
  return std::any_of( mems.begin(), mems.end()
  //                        Paramètre de la fonction anonyme
  //                        |
  //                        v
                    , [](int i)
                      // Corps de la fonction
                      {
                        return  i > 3 && i < 10;
                      }
                    );
}
```

# Fonctions anonymes

## Mise en place

```
bool process_buffer(std::vector<int> const& mems)
{
  return std::any_of( mems.begin(), mems.end()
  //  Environnement externe de la fonction anonyme
  //                  |     Paramètre de la fonction anonyme
  //                  |     |
  //                  v     v
                    , [](int i)
                      // Corps de la fonction
                      {
                        return  i > 3 && i < 10;
                      }
                    );
}
```

# Fonctions anonymes

## Paramètres templates

```
1
2  bool process_buffer(std::vector<int> const& mems)
3  {
4    return std::any_of( mems.begin(), mems.end()
5    //    Paramètre générique de la fonction anonyme
6    //                          |
7    //                          v
8                       , [](auto i)
9                         {
10                          return  i > 3 && i < 10;
11                        }
12                      );
13 }
```

# Fonctions anonymes

## Paramètres variadiques

```
// Le type d'une fonction anonyme n'est pas connu de l'utilisateur
auto f  = [](auto... args)
          {
            return std::make_tuple(args...);
          };

// t contient un tuple<int,const char*,float, std::pair<int,int>>
auto t = f(4, "4", 3.57f, std::make_pair(5,9));
```

# Fonctions anonymes

## Environnement de capture

```
std::vector<int> data{1,2,3,4,5,6,7};

int n;
std::cin >> n;

std::transform( data.begin(), data.end()
                // n est utilisé au sein de la fonction anonyme mais
                // n'apparait pas dans la liste des paramètres. Il est
                // nécessaire de le "capturer" par copie dans l'environnement
                // de la fonction anonyme
              , [n](auto d) { return n*d; }
              );
```

# Fonctions anonymes

## Environnement de capture

```
std::vector<int> data{1,2,3,4,5,6,7};

int n;
std::cin >> n;

std::transform( data.begin(), data.end()
                // n est utilisé au sein de la fonction anonyme mais
                // n'apparait pas dans la liste des paramètres. Il est
                // nécessaire de le "capturer" par référence dans l'environnement
                // de la fonction anonyme
              , [&n](auto d)
                {
                  n++;
                  return n*d;
                }
              );
```

# Fonctions anonymes

## Environnement de capture

```
void transform_via_ptr( std::vector<int>&       data
                      , std::unique_ptr<int>&&  ptr
                      )
{
  std::transform( data.begin(), data.end()
                // n est utilisé au sein de la fonction anonyme mais
                // n'apparait pas dans la liste des paramètres. Il est
                // nécessaire de le "capturer" dans l'environnement
                // met il est nécessaire de l'y transférer
              , [local = std::move(ptr)](auto d)
                {
                  return d + *local;
                }
              );
}
```

# Fonctions anonymes

## Fonction d'ordre supérieur

- Renvoyer une fonction créée de toute pièce depuis une autre fonction

```
template<typename F, typename G>
auto compose(F f , G g)
{
  return  [f,g](auto x)
          {
            return f(g(x));
          };
}

// f_g est la fonction cosf( sqrtf(x) ) et stockable dans une variable
auto f_g = compose( cosf, sqrtf );
```

# Structures Génériques

**Motivations**

- Certaines structures de données sont paramétrables
- Ex: un tableau contenant un type arbitraire
- Les structures génériques permettent de transmettre cette information

**Mise en place**

- Syntaxe identique à celle des fonctions génériques
- Guide de déduction
- Spécialisation et spécialisation partielle

### Conversion vers une structure générique

- Le type des données stockées pourrait être arbitraire
- La quantité de données stockées pourrait être arbitraire

```
1  struct array_of_5reals
2  {
3    float data[5];
4
5    std::size_t size() const { return 5; }
6
7    float  operator[](std::ptrdiff_t i) const { return data[i]; }
8    float& operator[](std::ptrdiff_t i)       { return data[i]; }
9  };
10
11 array_of_5reals x = {1.2f,0.3f,5f,7.9f,8.88f};
```

## Conversion vers une structure générique

- Introduction d'un paramètre *template* pour généraliser le type de donnée
- Le code dépendant de `Type` ets compilé seulement si il est effectivement utilisé

```
1  // array_of_5 est un patron de structure
2  template<typename Type> struct array_of_5
3  {
4    Type data[5];
5
6    std::size_t size() const { return 5; }
7
8    Type  operator[](std::ptrdiff_t i) const { return data[i]; }
9    Type& operator[](std::ptrdiff_t i)       { return data[i]; }
10 };
11
12 // array_of_5<int> est l'instantiation de ce patron pour Type = int
13 array_of_5<int> x = {12,3,57,79,888};
```

## Conversion vers une structure générique

- Introduction d'un paramètre *template* pour généraliser la taille du tableau
- Les entiers constants sont des paramètres *template* acceptables

```
1  // array est un patron de structure
2  template<std::size_t N, typename Type> struct array
3  {
4    Type data[N];   // N est utilisable comme une constante entière
5
6    std::size_t size() const { return N; }
7
8    Type  operator[](std::ptrdiff_t i) const { return data[i]; }
9    Type& operator[](std::ptrdiff_t i)       { return data[i]; }
10 };
11
12 // array<3,int> est l'instantiation de ce patron pour Type = int et N = 3
13 array<3,int> x = {12,3,57};
```

# Structures Génériques

## Conversion vers une structure générique

- Introduction d'un paramètre *template* pour généraliser la taille du tableau
- Les entiers constants sont des paramètres *template* acceptables

```
1  // array est un patron de structure
2  template<std::size_t N, typename Type> struct array
3  {
4    Type data[N];   // N est utilisable comme une constante entière
5
6    static std::size_t size() { return N; } // N ne dépend que du type de array
7
8    Type  operator[](std::ptrdiff_t i) const { return data[i]; }
9    Type& operator[](std::ptrdiff_t i)       { return data[i]; }
10 };
11
12 auto s = array<3,int>::size();
```

### Spécialisation de structures génériques

- Dans certains cas, le code contenu dans une structure générique peut vouloir varier en fonction d'un ou plusieurs paramètres templates afin de garantir la correction ou les performances du code en général.
- Il est possible de **spécialiser** un structure générique afin de s'assurer de la qualité du code généré.

### Spécialisations totales ou partielles

- On parle de **spécialisation totale** si l'on fige la totalité des paramètres templates
- On parle de **spécialisation partielle** si l'on fige une partie des paramètres templates
- Lors de l'instantiation d'une structure générique présentant des spécialisations, le compilateur favorise la version **la plus spécialisée**

# Structures Génériques

## Exemple de mise en œuvre

- `label` construit une chaîne de caractères contenant la valeur de son paramètre
- Ce code ne fonctionne pas en l'état pour les pointeurs
- Ce code ne fonctionne pas en l'état pour `void`

```
1  template<typename Type> struct label
2  {
3    label(Type v) : content_("valeur: " + std::to_string(v)) {}
4
5    std::string const& value() const { return content_; }
6
7    private:
8    std::string content_;
9  };
10
11 auto s = label(4).value();  // s contient "valeur: 4"
```

### Spécialisation de structures génériques

- Une spécialisation totale se distingue de son template primaire par la notation `template<>`
- Les types utilisés pour spécialiser la structure apparaissent accolés à sa définition

```
1  // Specialisation totale    Type utilisé pour spécialiser
2  //       v                       v
3  template< > struct label<std::nullptr_t>
4  {
5    label(std::nullptr_t) : content_("nulltpr") {}
6    std::string const& value() const { return content_; }
7
8    private:
9    std::string content_;
10 };
11
12 auto s = label(nullptr).value();  // s contient "nullptr"
```

### Spécialisation partielle de structures génériques

- Une spécialisation partielle se distingue de son template primaire par un spécificateur template ne contenant que les éléments nécessaire à définir le motif de spécialisation

```
1  // Specialisation partielle   Motif de type
2  //       v                          v
3  template<typename T> struct label<T*>
4  {
5    label(T* v)
6    {
7      std::ostringstream str;
8      str << "pointeur: " << (void*)(v);
9      content_ = str.str();
10   }
11   // ...
12 };
13
14 auto s = label(&f).value();  // s contient "pointeur: 0x7FFF4D88865A4200"
```

# Templates Variadiques

## Objectifs

- Spécifier un *template* de fonction ou de structure dont le nombre de paramètres est variable
- Introduire une notion de **pack de paramètre** manipulable à la compilation

## Applications

- Renforcer le typage de certaines constructions (tuple, etc...)
- Simplifier l'API niveau utilisateur

## Syntaxe de base

```
template<typename... Args> void f(Args... args);

template<typename... Types> struct tuple {};
template<int... Sizes>      struct hull {};
```

## Mise en oeuvre - Cas récursif

```
1  // Parametre template classique
2  //         |
3  //         |  typename... introduit un variadic pack
4  //         |  c'est à dire une liste de type à déterminer
5  //         |                    |
6  //         v                    v
7  template<typename First, typename... Others>
8  auto total_size(First const&, Others const&... args)
9  {
10   return sizeof(First) + total_size(args...);
11 }
```

## Mise en oeuvre - Cas récursif

```
template<typename First, typename... Others>
//       Argument de fonction classique
//              |
//              |  Other const&... args introduit un parameter pack
//              |  c'est à dire une liste d'argument à déterminer
//              |                   |
//              v                   v
auto total_size(First const&, Others const&... args)
{
  return sizeof(First) + total_size(args...);
}
```

## Mise en oeuvre - Cas récursif

```
template<typename First, typename... Others>
auto total_size(First const&, Others const&... args)
{
  // La notation ... déroule le contenu d'une expression
  // contenant un variadic pack ou un parameter pack pour
  // chacun des éléments qu'ils contiennent en les séparant
  // par une virgule syntaxique
  return sizeof(First) + total_size(args...);
}
```

## Mise en oeuvre - Cas récursif

```cpp
// L'aspect récursif du traitement de total_size implique
// qu'il est nécessaire de fournir un cas terminal
// sans template variadique
template<typename Type> auto total_size(Type const&)
{
  return sizeof(Type);
}

template<typename First, typename... Others>
auto total_size(First const&, Others const&... args)
{
  return sizeof(First) + total_size(args...);
}

auto sz = total_size(1, 'z', 3.); // sz = 4+1+8 = 13
```

## Mise en oeuvre - Cas non-récursif

```cpp
template<typename... Others>
auto total_size(Others const&...)
{
  // On utilise ici ... sur l'expression sizeof(args)... qui est donc déroulé
  // pour chaque Others, séparé par une virgule dans un contexte ou cette
  // syntaxe correspond à la création d'un tableau
  std::size_t size[] = { sizeof(Others)... };
  std::size_t r = 0;

  // Le calcul est ensuite exécuté de façon itérative
  for(auto s : size)
    r += s;

  return r;
}

auto sz = total_size(1, 'z', 3.); // sz = 4+1+8 = 13
```

## Mise en œuvre - Cas non-récursif C++17

```cpp
template<typename... Others>
auto total_size(Others const&...)
{
  // A partir de C++ 17, ... peut s'utiliser avec n'importe quel opérateurs
  // binaires du langage, déroulant ainsi les parameter pack ou les variadic pack
  // en les séparant par un opérateur arbitraire
  return (sizeof(Others) + ... + 0ULL);
}

auto sz = total_size(1, 'z', 3.); // sz = 4+1+8 = 13
```

# Le Perfect Forwarding

## Problématique

- Certaines fonctions sont utilisées comme trampoline vers une ou plusieurs autres fonctions
- Elles doivent pouvoir passer n'importe quel catégorie de paramètres aux fonctions sous-jacentes
- Comment gérer les $3^N$ cas du à la gestion de N paramètres pouvant être des lvalue, des lvalue immuables ou des rvalue ?

## Exemple : La fonction usine

```
1  // Ne fonctionne pas avec les rvalue et les const lvalue
2  template<typename Type, typename... Args>
3  auto instanciate(Args&... args) { return Type(args...); }
4
5  // Ne fonctionne pas avec les lvalue
6  template<typename Type, typename... Args>
7  auto instanciate(Args const&... args) { return Type(args...); }
8
9  auto k = instanciate<std::string>(4, '-');
```

# Le Perfect Forwarding

## Solution - Les Références Universelles

- Si `T` est un paramètre template, alors `T&&` est une **référence universelle**
- Ce type peut matcher des lvalue, des lvalue constante, des rvalue sans erreurs
- Une fonction `std::forward` permet de transférer proprement ces référence universelles à d'autres fonctions sans perte d'informations

## Exemple : La fonction usine

```
1  // Fonctionne pour toutes les catégories de valeurs
2  template<typename Type, typename... Args>
3  auto instanciate(Args &&... args) // /!\ Args&& n'est pas une rvalue-reference vers Args
4  {
5    return Type( std::forward<Args>(args)... );
6  }
7
8  auto k = instanciate<std::string>(4, '-');
```

# Les Traits

**Objectifs**

- Fournir un protocole standard pour classifier les types
- Fournir un protocole standard pour générer des types
- Permettre de généraliser du code template

**Exemple**

- Détection de qualificateur; `&`, `*`, `const`
- Classification par propriétés fondamentales
- Génération de types sécurisée

# Les Traits

## Principe général

- Un traits = une structure template fournissant un type interne ou valeur constante
- Ce membre interne renvoie un type ou une valeur constante répondant à une question précise
- Acessible via `#include <type_traits>`

## Exemples

```
template<typename T, typename U>
auto f(T t, U u)
{
  if( std::is_same<T,U>::value )  return t+u;
  else                            return typename std::common_type<T,U>::type{t};
}
```

# Les Traits

## Principe général

- Un traits = une structure template fournissant un type interne ou valeur constante
- Ce membre interne renvoie un type ou une valeur constante répondant à une question précise
- Acessible via `#include <type_traits>`

## Exemples

```
1  template<typename T> auto f(T t)
2  {
3    // Arrete la compilation si la condition est fausse
4    static_assert ( !std::is_pointer<T>::value
5                  , "Cette fonction ne fonctionne pas avec des pointeurs"
6                  );
7
8    return &t;
9  }
```

## Stratégie d'implémentation - Spécialisation partielle

```
template<typename T>
struct is_pointer : std::false_type {};

template<typename T>
struct is_pointer<T*> : std::true_type {};
```

## Stratégie d'implémentation - Spécialisation totale

```
template<typename T>
struct is_void : std::false_type {};

template<>
struct is_void<void> : std::true_type {};
```

### Stratégie d'implémentation - Surcharge de fonction

- Utilisation des propriétés de `decltype` pour forcer le choix entre deux surcharge de fonction
- `decltype` est ensuite réutilisé pour calculer le type de retour de la fonction sélectionnée

```
1  template<typename T> struct is_streamable
2  {
3    template<typename U>
4    static auto test(int) → decltype ( std::cout << std::declval<U>()
5                                     , std::true_type{}
6                                     );
7
8    template<typename> static std::false_type test(...);
9
10   static const bool value = decltype(test<T>(0))::value;
11 };
```

### Stratégie d'implémentation - Détection par `decltype`

- Utilisation des propriétés de `decltype` pour générer un type invalide si une expression est invalide
- `std::void_t` permet de détecter ces appels incorrects et de générer `void` sinon

```
1  template<typename T, typename Enable = void>
2  struct is_streamable : std::false_type
3  {};
4
5  template<typename T>
6  struct  is_streamable<T, std::void_t<decltype(std::cout << std::declval<T>())>
7        : std::true_type
8  {};
```

# Calcul à la compilation

**Problématiques**

- Les surcharges de templates permettent de calculer des valeurs ou des types à la compilation
- Ces calculs à la compilation permettent de simplifier des APIs, d'optimiser du code runtime
- Leur maintenance est complexe et peut mener à de la duplication de code

**Solution**

- Fusionner l'aspect runtime et compile-time des fonctions
- Définir des contextes où des calculs arbitraires peuvent être exécutés à la compilation
- On peut voir ces contextes comme une prise de contrôle du compilateur

# Calcul à la compilation

## Exemple - Calcul de factorielle à la compilation

- Implémentation traditionnelle à base de surcharge
- Beaucoup de bruit syntaxique

```
1  template<int N> struct factorial
2  {
3    static const int value = N * factorial<N-1>::value;
4  };
5
6  template<> struct factorial<0>
7  {
8    static const int value = 1;
9  };
10
11 std::array<int, factorial<7>::value> x; //  Tableau de taille 7!
```

# Calcul à la compilation

## Exemple - Calcul de factorielle à la compilation

- Introduction du mot-clé `constexpr`
- La fonction `constexpr` reste utilisable au runtime
- Code plus fluide

```cpp
constexpr int factorial(int n)
{
  int r = 1;
  for(int i=1;i<=n;i++) r *= i;
  return r;
}

std::array<int, factorial(7)> x; //  Tableau de taille 7!

std::cout << factorial(5) << "\n";
```

# Merci de votre attention