# BASE DU DÉV. LOGICIEL

Joel FALCOU

# Un peu d'histoire ...

**Au démarrage :**

- **1978** : Bjarne Stroustrup travaille sur Simula67 qui s'avère peu efficace
- **1980** : Démarrage du projet C with Classes, du C orienté objet compilé via CFront.
- **1983** : C with Classes devient C++
- **1985** : Parution de The C++ Programming Language
- **1998** : Première standardisation de C++ : ISO/IEC 14882 :1998

**La reprise :**

- **2005** : Parution du C++ Technical Report 1 qui deviendra C++0x
- **2011** : Ratification de C++0x sous le nom C++11
- **2014** : Ratification de C++14
- **2017** : Ratification de C++17
- **2020,23,26** : Prochaines milestones du langage

# Pourquoi C++ ?

***'Make simple things simple'***

- Les tâches simples doivent s’exprimer simplement
- Les tâches complexes ne doivent pas être insurmontables

**C++ : Un langage extensible**

- Les combinaisons d'éléments du langage doivent se comporter intuitivement
- Le niveau d'abstraction du code peut être arbitrairement élevé
- La réutilisation de composants est favorisée

**C++ : Un langage proche de la machine**

- *'Don't pay for what you don't use'*
- *'Zero Cost Abstraction'*

# Modèle de programmation C++

**Un langage multi-paradigmes**

- Support du style **impératif**
- Support du style **objet**
- 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 **fonctionnel**

# Modèle de programmation C++

**Un langage multi-paradigmes**

- Support du style **impératif**
- 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 **fonctionnel**

# Modèle de programmation C++

**Un langage multi-paradigmes**

- Support du style **impératif**
- Support du style **objet**
- 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
  - Ses éléments fondamentaux sont les fonctions

# Modèle de programmation C++

**Un langage multi-paradigmes**

- Support du style **impératif**
- Support du style **objet**
- Support du style **fonctionnel**

**Que choisir ?**

- Tout ces styles permettent de résoudre les problèmes de développement au quotidien
- Certains problèmes ont des solutions plus simple dans certain de ces styles
- Tous participent à faire de C++ un langage efficace

# Du code à l'exécutable

# Le Code source

**Définition du “code source”**

- Le code source est un ensemble de fichiers texte
- Ce(s) fichier(s) sont **lisibles par un humain**
- Il(s) exprime(nt) au mieux l'intention du développeur.

**Relation code/langage**

- Un **langage de programmation** (comme C++) permet de rédiger un code source via des constructions logiques formalisées.
- Le code source est traité par un **compilateur** qui l'analyse, le vérifie et le transforme en éléments exécutables par le processeur

# Le Code source - Exemple

**mon_programme.cpp**

- Un programme nécessite un **point d'entrée**
- Ce point d'entrée est exécuté lors de son démarrage.
- En C++, ce point d'entrée s'appelle `main`

```
1  main
```

# Le Code source - Exemple

**mon_programme.cpp**

- `main` est une **fonction**, une unité opérationnelle fondamentale en C++
- Une fonction reçoit des **arguments** en entrées
- Une fonction traite ses arguments et, potentiellement, renvoie un **résultat**
- `main`, sous sa forme simple, ne reçoit pas d'argument et renvoie un entier (`int`)

```
1  int main()
```

# Le Code source - Exemple

**mon_programme.cpp**

- Une fonction traite ses arguments en exécutant le code qu'elle contient
- Le code d'une fonction est contenu dans son **corps**, délimité par { }
- `main` renvoie un entier égal à 0 via le **mot-clé** `return`

```
1  int main()
2  {
3    return 0;
4  }
```

# Le Code source - Exemple

**mon_programme.cpp**

- C++ fournit des fonctions pré-définies : **la bibliothèque standard**
- Ces fonctions sont déclarées dans des **fichiers d'en-têtes**
- Les en-têtes sont utilisables via la directive `#include`
- Utilisons par exemple les fonctions d'affichages fournies par l'en-tête `iostream`

```cpp
#include <iostream>

int main()
{
  std::cout << "Exécution réussie !" << std::endl;
  return 0;
}
```

# La Compilation

## Principes généraux

- Analyse et vérifie la correction du code source
- Traduit le code en langage assembleur
- Transforme l'assembleur en code machine
- Agrège tous les éléments nécessaires pour produire un fichier binaire exécutable

## Les trois étapes de compilation

- Pre-processing
- Génération de code
- Édition des liens

# La Compilation

**Principes généraux**

- Analyse et vérifie la correction du code source
- Traduit le code en langage “processeur” (assembleur)
- Agrège tous les éléments nécessaires pour produire un fichier binaire exécutable

**Les trois étapes de compilation**

- Pre-processing
  - Traite le texte du source afin de le rendre compréhensible par le compilateur
- Génération de code
- Édition des liens

# La Compilation

**Principes généraux**

- Analyse et vérifie la correction du code source
- Traduit le code en langage “processeur” (assembleur)
- Agrège tous les éléments nécessaires pour produire un fichier binaire exécutable

**Les trois étapes de compilation**

- Pre-processing
- Génération de code
  - Analyse le code et génère du code binaire
- Édition des liens

# La Compilation

**Principes généraux**

- Analyse et vérifie la correction du code source
- Traduit le code en langage “processeur” (assembleur)
- Agrège tous les éléments nécessaires pour produire un fichier binaire exécutable

**Les trois étapes de compilation**

- Pre-processing
- Génération de code
- Édition des liens
  - Agrège les fragments de codes binaires dans un exécutable

# La Compilation

**Invoquer le compilateur**

```
g++ <source> -o <executable> <option>
```

- `<source>` : liste des fichiers sources à compiler
- `<executable>` : nom du programme exécutable à générer
- `<option>` : liste des options de compilations: optimisation, version du langage, vérification poussée des erreurs, gestion de l'architecture, etc...

**Exemples**

- Compilation optimisée avec vérifications poussées
  ```
  g++ mon_programme.cpp -o mon_programme.prog -Wall -O3 -std=c++17
  ```
- Compilation pour le débogage
  ```
  g++ mon_programme.cpp -o mon_programme.prog -g -std=c++17
  ```

# La Compilation

## Invoquer le compilateur

```
g++ <source> -o <executable> <option>
```

- `<source>` : liste des fichiers sources à compiler
- `<executable>` : nom du programme exécutable à générer
- `<option>` : liste des options de compilation : optimisation, version du langage, vérification poussée des erreurs, gestion de l'architecture, etc...

## Exemples

- Certains composants ont besoin de code externe pour fonctionner
- L'option `-l` permet à l'éditeur de lien d'accéder à ce code pré-compilé.

```
g++ mon_programme.cpp -o mon_programme.prog -Wall -O3 -std=c++17 -lpthread
```

# Types et Variables

# Contenu

**Notion de type**

- Types natifs
- Types avancés
- Qualificateurs
- Inférence de type

**Notion de structures de données**

- Types énumérés
- Types aggrégés

**Ce qu'il faudra retenir**

- Quel type pour quelle tâche ?
- Utilité des qualificateurs

# Notion de Types

# Types natifs

**Définition**

- Il s'agit des **types de données** gérés de manière directe par le langage et le processeur.
- Ils représentent les valeurs fondamentales manipulables par un programme C++.

**Sommaire**

- Entiers signés et non-signés
- Entiers de taille fixe
- Entiers de taille flexible
- Valeurs booléenes
- Nombres en virgule flottante

# Types natifs - Entiers signés

## Propriétés

- Représentation des entiers relatifs ($\mathbb{Z}$)
- **Leurs amplitudes numériques varient**
- Supportent les opérations arithmétiques classiques
- La manipulation les bits de ces types est à éviter

```
1  // Entiers signés
2  signed char      c; // valeurs comprises en général entre -128 et 127
3  signed short     s; // valeurs comprises en général entre -32768 et 32767
4  signed int       i; // valeurs comprises en général entre -2147483648 et 2147483647
5  signed long long l; // valeurs comprises en général entre -9223372036854775808 et 9223372036854775807
```

# Types natifs - Entiers signés

## Propriétés

- Représentation des entiers relatifs ($\mathbb{Z}$)
- Leurs amplitudes numériques varient
- **Supportent les opérations arithmétiques classiques**
- La manipulation les bits de ces types est à éviter

```
1  signed int    i = 4'000'000;
2  signed short  u = 40'000;
3  signed char   c = 60;
4
5  // Calculs arithmétiques
6  i = i + 3 * u;  // i = 4'120'000
7  u = -c;         // u = -60
8  c = 2*c;        // c = 120
9
10 // Réduction d'amplitude
11 c = 2*c;        // c = 112 (= 240 % 128)
```

# Types natifs - Entiers non-signés

## Propriétés

- Représentation d'un ensemble de bits
- **Leurs amplitudes numériques varient**
- Supportent les opérations sur les bits
- Les opérations arithmétiques sur ces types sont à éviter

```
1  // Entiers non-signés
2  unsigned char      c; // valeurs comprises en général entre 0 et 255
3  unsigned short     s; // valeurs comprises en général entre 0 et 65535
4  unsigned int       i; // valeurs comprises en général entre 0 et 4294967295
5  unsigned long long l; // valeurs comprises en général entre 0 et 18446744073709551615
```

# Types natifs - Entiers non-signés

## Propriétés

- Représentation d'un ensemble de bits
- Leur amplitudes numériques varient
- **Supportent les opérations sur les bits**
- Les opérations arithmétiques sur ces types sont à éviter

```
1  unsigned char some  = 0x3C;       // valeur 60 en hexadecimale
2  unsigned char other = 0b01010101; // valeur 85 en binaire
3
4  unsigned char r = some & other;   // r = 0b00010100 = 20
5  unsigned char s = some | other;   // s = 0b01111101 = 125
6
7  unsigned char d = r >> 2;         // d = 5
8  unsigned short l = (s << 8) | r;  // l = 0b0111110100010100 = 32020
```

# Types natifs - Entiers de taille fixe

**Problématique**

- La taille des entiers natifs dépend du système
- Comment s'assurer d'une taille minimale ?
- Cas d'utilisation: sauvegarde sur disque, envoi sur le réseau, etc...

**Solution**

- C++ fournit un jeu de types entiers de taille et signe garantis
- Permet de partager des valeurs entre systèmes et architectures
- **Ne remplacent pas les types natifs pour l'arithmétique**

# Types natifs - Entiers de taille fixe

## Usage

- Disponible dans `#include <cstdint>`
- Forme générale: `{u}int{8,16,32,64}_t`

```cpp
#include <cstdint>

// entier signé de 64 bits
std::int64_t s = 123456789098LL;

// entier non-signé de 8 bits
std::uint8_t mask = 0b11110000;
```

# Types natifs - Entiers de taille flexible

**Problématique**

- La valeur de certaines grandeurs dépend du système
  - taille d'un objet en mémoire
  - distance entre deux valeurs en mémoire
- Comment stocker ces valeurs dans une variable de type adéquat ?

**Solution**

- C++ fournit un jeu de types dont la taille et le signe varie automatiquement
- **A utiliser systématiquement pour stocker ces grandeurs**

# Types natifs - Entiers de taille flexible

## Usage

- Disponible dans `#include <cstddef>`

```
#include <cstddef>

long x,y;

// std::size_t représente la taille d'un objet en mémoire
std::size_t how_big = sizeof(x); // sizeof(x) renvoie la taille en octet de x

// std::ptrdiff_t représente la distance entre deux valeurs en mémoire
std::ptrdiff_t how_far = &y - &x;  // & indique la position d'une variable en mémoire
```

# Types natifs - Valeurs booléennes

## Objectifs

- Introduction du type `bool` et des valeurs `true` et `false`
- Représentent une valeur de type “oui” ou “non”
- **A utiliser systématiquement** pour représenter une valeur logique

```
1  bool prop1 = true;  // proposition logiquement vraie
2  bool prop2 = false;  // proposition logiquement fausse
3
4  int i = 9;
5  bool is_big = i > 15; // les operations de comparaisons produisent des bool
6
7  // Operations de combinaison logique
8  // !  : negation logique
9  // && : ET logique
10 // || : OU logique
11 bool truth = (!is_big && prop1) || prop2;
```

# Types natifs - Nombres à virgules flottantes

## Problématique

- Assurer un compromis entre précision et performance
- Utilisation du standard IEEE754
- Représentation « à trous » de ($\mathbb{R}$)

```
1  // valeur flottantes 32 bits - 7 chiffres significatifs
2  float x = 0.2541369f;
3
4  // valeur flottantes 64 bits - 16 chiffres significatifs
5  double x = 0.2541369780123548;
6
7  // Interaction avec les entiers
8  float quotient_incorrect = 1 / 5;   // = 0    car division entiere
9  float quotient_correct   = 1 / 5.f; // = 0.2f car division reelle
```

# Types avancés

**Définition**

- **Types de données** plus complexes
- Permettent de manipuler des abstractions de plus haut niveau
- Fournies par la bibliothèque standard

**Sommaire**

- Chaîne de caractères
- Tableau de taille fixe
- Tableau de taille dynamique
- Paire
- Tuple
- Ensemble
- Dictionnaire

# Types avancés - Chaîne de caractère

**Définition**

- C++ fournit un type représentant une chaîne de caractères
- Cette chaîne ne peut contenir que des caractères ASCII étendu.
- Définie dans l'en-tête `<string>`

**Interface**

- Accès aux éléments en lecture/écriture
- Copie, Concaténation, redimensionnement
- Recherche, extraction de sous-chaînes
- Gestion automatique de la mémoire associée

# Types avancés - Chaîne de caractère

## Construction par défaut

```cpp
#include <string>
#include <iostream>

int main()
{
  // Construction par défaut
  std::string s;

  // Accès à la taille
  std::cout << s.size() << "\n";
}
```

```
0
```

# Types avancés - Chaîne de caractère

## Constructions avancés

```cpp
#include <string>
#include <iostream>

int main()
{
  std::string sc = "ceci est du texte";
  std::string sp("ceci est du texte",4);
  std::string si( &sc[5], &sc[9]);
  std::string sr(10,'*');

  std::cout << sc << "\n" << sp << "\n";
  std::cout << si << "\n" << sr << "\n";
}
```

```
ceci est du texte
ceci
est du
**********
```

# Types avancés - Chaîne de caractère

## Manipulation de caractères

```cpp
#include <string>
#include <iostream>

int main()
{
  std::string sc = "ceci est du texte";
  std::cout << sc[3] << "\n";       // Lecture

  sc[0] = 'C';                      // Ecriture
  std::cout << sc << "\n";

  sc.push_back('!');                // Ajout en fin
  std::cout << sc << "\n";
}
```

```
i
Ceci est du texte
Ceci est du texte!
```

## Manipulation de Chaîne

```cpp
#include <string>
#include <iostream>

int main()
{
  std::string st = "top";
  std::string sb = "bingo!";

  auto stb = st + " " + sb;               // Concatenation
  std::cout << stb << "\n";

  stb.clear();                            // Vidange
  std::cout << "'" << stb << "'" << "\n";
}
```

```
top bingo!
''
```

# Types avancés - Chaîne de caractère

## Recherche/Extraction de sous-Chaîne

```cpp
#include <string>
#include <iostream>

int main()
{
  std::string sc = "ceci est du texte";

  auto sub = sc.substr(9,2);
  std::cout << sub << "\n";

  auto n = sc.find("est");
  if (n == std::string::npos) std::cout << "pas trouvé\n";
  else                        std::cout << "trouvé: " << sc.substr(n) << '\n';
}
```

```
du
trouvé: est du texte
```

# Types avancés - Tableau de taille fixe

## Définition

- C++ fournit un type représentant une succession de valeurs de taille fixe
- Ces éléments se trouvent rangés de manière contigue dans la mémoire
- Ce type représente un **tableau**
- Définie dans l'en-tête `<array>`

```
//    Type des valeurs du tableau
//          |
//          v
std::array< int, 42 >  p;
//                ^
//                |
//           Nombre d'éléments
```

# Types avancés - Tableau de taille fixe

## Usage

```cpp
#include <array>

int main()
{
  std::array<double, 3> origin = { 0,0,0 };
  std::array<double, 3> p1 = {1.5,-0.6,2.3 };
  std::array<double, 3> p2 = p1;

  std::cout << origin.size() << "\n";
  std::cout << p2[0] << " " << p2[1] << " " << p2[2] << "\n";
}
```

```
3
1.5 -0.6 2.3
```

# Types avancés - Tableau de taille dynamique

## Définition

- C++ fournit un type représentant une succession de valeurs de taille variable
- Ces éléments se trouvent rangés de manière contigue dans la mémoire
- Ce type représente un **tableau** dont le “volume” peut varier
- Définie dans l'en-tête `<vector>`

```
//      Type des valeurs du tableau
//           |
//           v
std::vector<float> v;
```

# Types avancés - Tableau de taille dynamique

## Construction par défaut

```cpp
#include <vector>
#include <iostream>

int main()
{
  // Construction par defaut
  std::vector<float> s;

  // Acces à la taille
  std::cout << s.size() << "\n";
}
```

```
0
```

# Types avancés - Tableau de taille dynamique

## Constructions avancés

```cpp
#include <vector>
#include <iostream>

int main()
{
  // 6 elements initialises par défaut
  std::vector<float> v(6);

  std::cout << v[0] << " " << v[1] << " " << v[2] << " ";
  std::cout << v[3] << " " << v[4] << " " << v[5] << "\n";
}
```

```
0 0 0 0 0 0
```

# Types avancés - Tableau de taille dynamique

## Constructions avancés

```cpp
#include <vector>
#include <iostream>

int main()
{
  // 6 valeurs spécifiques
  std::vector<float> v = {1.2f,3.4f,5.6f,7.8f,9.10f,11.12f};

  std::cout << v[0] << " " << v[1] << " " << v[2] << " ";
  std::cout << v[3] << " " << v[4] << " " << v[5] << "\n";
}
```

```
1.2 3.4 5.6 7.8 9.1 11.12
```

# Types avancés - Tableau de taille dynamique

## Constructions avancés

```cpp
#include <vector>
#include <iostream>

int main()
{
  // 6 valeurs répétitives
  std::vector<float> v(6,-42.69);

  std::cout << v[0] << " " << v[1] << " " << v[2] << " ";
  std::cout << v[3] << " " << v[4] << " " << v[5] << "\n";
}
```

```
-42.69 -42.69 -42.69 -42.69 -42.69 -42.69
```

# Types avancés - Tableau de taille dynamique

## Constructions avancés

```cpp
#include <vector>
#include <iostream>

int main()
{
  // 3 valeurs extraites d'une source annexe
  std::vector<float> u = {6,5,4,3,2,1};
  std::vector<float> v = { &u[1], &u[4] };

  std::cout << v[0] << " " << v[1] << " " << v[2] << "\n";
}
```

```
5 4 3
```

# Types avancés - Tableau de taille dynamique

## Manipulation des valeurs

```cpp
#include <vector>
#include <iostream>

int main()
{
  std::vector<float> v = {0,0.5,1.5,3,6,36};
  std::cout << v[3] << "\n";       // Lecture

  v[1] = 10*v[1];                  // Écriture
  std::cout << v[1] << "\n";

  v.push_back(99);                 // Ajout en fin
  std::cout << v[6] << "\n";
}
```

```
3
5
99
```

# Types avancés - Paire

## Définition

- La paire est un type de donnée qui contient deux éléments de types arbitraires
- Elle représente un aggrégat de deux valeurs qui sont appairées logiquement mais n'ont pas de sémantique particulière.
- Définie dans l'en-tête `<utility>` ou `<tuple>`
- Ces valeurs sont simplement nommées **`first`** et **`second`**

```
//    1er composant
//          |
//          v
std::pair< int, float >  p;
//                ^
//                |
//           2ème composant
```

# Types avancés - Paire

## Paire par défaut

```cpp
#include <utility>
#include <iostream>

int main()
{
  // Construction par défaut
  std::pair<int,float> p0;

  // Modification
  p0.second = 0.5f;

  // Accès
  std::cout << p0.first << " " << p0.second << "\n";
}
```

```
0 0.5
```

# Types avancés - Paire

## Initialisation d'une paire

```cpp
#include <utility>
#include <iostream>

int main()
{
  // Initialisation
  std::pair<int,float> p1(5,3.8f);

  // Accès
  std::cout << p1.first << " " << p1.second << "\n";
}
```

```
5 3.8
```

# Types avancés - Paire

## Initialisation d'une paire par conversion

```cpp
#include <utility>
#include <iostream>

int main()
{
  // Initialisation
  std::pair<int,float>    p1(65,3.8f);

  // Conversion implicite
  std::pair<char,double>  p2(p1);

  std::cout << p2.first << " " << p2.second << "\n";
}
```

```
A 3.8
```

# Types avancés - Paire

## Initialisation d'une paire par construction parcellaire

```cpp
#include <tuple>
#include <string>
#include <iostream>

int main()
{
  std::pair<std::string, int> p ( std::piecewise_construct
                                , std::forward_as_tuple(10, 'a')
                                , std::forward_as_tuple(42)
                                );

  std::cout << p.first << " " <<  p.second << "\n";
}
```

```
aaaaaaaaaa 42
```

# Types avancés - Tuple

## Définition

- Le tuple (ou N-uplet) est un type de donnée qui généralise la paire aux aggrégats de N valeurs de types arbitraires
- Comme pour `std::pair`, un tuple n'a pas de sémantique particulière
- Définie dans l'en-tête `<tuple>`
- Ces éléments sont accessibles via une indexation connue à la compilation

```
1  //    1er composant     3ème composant
2  //          |             |
3  //          v             v
4  std::tuple< int, float, char >  t;
5  //                ^
6  //                |
7  //           2ème composant
```

# Types avancés - Tuple

## Tuple par défaut

```cpp
#include <tuple>
#include <string>
#include <iostream>

int main()
{
  // Construction par défaut
  std::tuple<int,float, std::string> p0;

  // Modification
  std::get<2>(p0) = "Hello !";

  // Accès
  std::cout << std::get<0>(p0) << " " << std::get<1>(p0) << " " << std::get<2>(p0) << "\n";
}
```

```
0 0 Hello !
```

# Types avancés - Tuple

## Initialisation

```cpp
#include <tuple>
#include <string>
#include <iostream>

int main()
{
  // Initialisation manuelle
  std::tuple<int,float, std::string> p = { 3, 3.1415f, "pi" };

  // Initialisation par make_tuple
  auto e = std::make_tuple( 2, 2.71828f, "euler" );

  std::cout << std::get<2>(p) << " = " << std::get<1>(p) << "\n";
  std::cout << std::get<2>(e) << " = " << std::get<1>(e) << "\n";
}
```

```
pi = 3.1415
euler = 2.71828
```

# Types avancés - Tuple

## Liaison de variable

```cpp
#include <tuple>
#include <iostream>

int main()
{
  float       value;
  std::string name;

  auto v = std::tie(value,name);

  std::get<0>(v) = 1.61803f;
  std::get<1>(v) = "golden";

  std::cout << name << " = " << value;
}
```

```
golden = 1.61803
```

# Types avancés - Tuple

## Manipulation de tuple

```cpp
#include <tuple>
#include <iostream>

int main()
{
  auto t1 = std::make_tuple("Hell");
  auto t2 = std::make_tuple('o', " World", '!');

  auto t12 = std::tuple_cat(t1, t2);
  std::cout << std::get<0>(t12) << std::get<1>(t12)
            << std::get<2>(t12) << std::get<3>(t12) << "\n";
}
```

```
Hello World!
```

# Types avancés - Ensemble

## Définition

- Un ensemble est un type représentant une liste triée de valeurs uniques
- Il fournit des opérations d'insertion et de recherche adaptées
- Définie dans l'en-tête `<set>`

```
//    Type des valeurs
//          |
//          v
std::set< int >  s;
```

# Types avancés - Ensemble

## Construction et insertion

```cpp
#include <set>
#include <iostream>

int main()
{
  std::set<std::string> animals;
  animals.insert("horse");
  animals.insert("dog");
  animals.insert("zebra");
  animals.insert("cat");

  for(auto& str: animals) std::cout << str << "\n";
}
```

```
cat
dog
horse
zebra
```

# Types avancés - Ensemble

## Construction et recherche

```cpp
#include <set>
#include <iostream>

int main()
{
  std::set<std::string> numbers{"un", "deux", "trois", "sept", "onze"};

  auto n = numbers.find("trois");
    if (n != numbers.end()) std::cout << "Trouvé " << (*n) << '\n';
    else                    std::cout << "Pas trouvé :(\n";
}
```

```
Trouvé trois
```

# Types avancés - Dictionnaire

## Définition

- Un dictionnaire est un type représentant une liste d'associations entre **clés** et **valeurs**
- Il fournit des opérations d'insertion et de recherche adaptées
- Définie dans l'en-tête `<map>`

```
//    Type des clés
//          |
//          v
std::map< int, std::string >  m;
//                    ^
//                    |
//             Type des valeurs
```

# Types avancés - Dictionnaire

## Constructions

```cpp
#include <map>
#include <string>

int main()
{
  std::map<std::string, int> m; // Construction par défaut

  m["pierre"] = 27;             // Insertion paire {clé,valeur}
  m["paul"]   = 19;             // m["clé"] = valeur

  std::map<std::string, float> csts  // Construction par énumération
  {
 // { clé , valeur}
    {"pi"     , 3.14152f },
    {"euler"  , 2.71828f },
    {"golden" , 1.61803f }
  };
}
```

## Recherche

```cpp
#include <map>
#include <string>
#include <iostream>

int main()
{
  std::map<std::string, float> csts{ {"pi", 3.14152f }, {"euler", 2.71828f } };

  // Accès direct
  std::cout << csts["pi"] << "\n";

  // /!\ Incorrect si la clé n'existe pas
  std::cout << csts["golden"] << "\n";
}
```

```
3.14152
0
```

# Types avancés - Dictionnaire

## Recherche

```cpp
#include <map>
#include <string>
#include <iostream>

int main()
{
  std::map<std::string, float> csts{ {"pi", 3.14152f }, {"euler", 2.71828f } };

  // find a un comportement cohérent en cas de clé manquante
  auto v = csts.find("pi");

  // Le résultat donne accès à la clé et à la valeur
  if (v != csts.end()) std::cout << "Trouvé: " << v->first << " " << v->second << '\n';
  else                 std::cout << "Pas trouvé\n";
}
```

```
Trouvé: pi 3.14152
```

# Manipulation des types

# Qualificateurs de types

**Objectifs**

- Renforcer la sémantique des types
- Ajouter une propriété à un type
- Clarifier la relation de la variable avec l'environnement

**Sommaire**

- Immutabilité
- Référence
- Pointeur

# Qualificateurs de types - Immutabilité

## Définition

- Une variable **immutable** ne peut changer de valeur
- Elle doit être obligatoirement définie avant tout usage
- L'immutabilité s'exprime via le mot-clé `const`

```
int       x = 10; // variable
int const y = 42; // variable immutable

// Ce code est valide
x = 9;
x = 2*y;

// Ce code ne compile pas :
//  > error: assignment of read-only variable 'y'
y = 4;
```

# Qualificateurs de types - Référence

## Définition

- Une référence est un **alias** pour une variable existante.
- Elle doit être **obligatoirement** liée à une autre variable
- Manipuler une référence manipule directement la variable référencée
- Une référence peut rajouter de l'immutabilité

```
1  std::string   s = "Ex";     // variable originale
2  std::string const& r2 = s;  // r2 est un alias immutable de s
3
4  std::string& r1 = s;        // r1 est un alias de s
5  r1 += "emple";              // Modifier r1 modifie s
6
7  // Ce code ne compile pas :
8  std::string& r0;  //  > error: 'r0' declared as reference but not initialized
9  r2 += "!";        //  > error: cannot modify through reference to const
```

# Qualificateurs de types - Pointeur

## Définition

- Un pointeur représente la position d'une variable dans la mémoire via l'opérateur &
- Il fournit un Accès indirect à la valeur associée via l'opération de **déréférencement**
- Contrairement à une référence, un pointeur peut ne rien référencer
- Contrairement à une référence, un pointeur peut changer de cible
- L'immutabilité d'un pointeur est indépendante de celle de sa cible

```
1  std::string   s = "Ex";     // variable originale
2  std::string* p1 = &s;       // r1 est un pointeur vers une string
3
4  *p1 += "emple";              // Modifier *r1 modifie s
5
```

# Qualificateurs de types - Pointeur

## Définition

- Un pointeur représente la position d'une variable dans la mémoire via l'opérateur &
- Il fournit un Accès indirect à la valeur associée via l'opération de **déréférencement**
- Contrairement à une référence, un pointeur peut ne rien référencer
- Contrairement à une référence, un pointeur peut changer de cible
- L'immutabilité d'un pointeur est indépendante de celle de sa cible

```
1  std::string   s = "Ex";     // variable originale
2  std::string* p1 = &s;       // r1 est un pointeur vers une string
3
4  std::string* p0 = nullptr;  // p0 ne cible "rien"
5  p0 = p1;                    // p0 cible maintenant la même variable que p1
6
```

# Qualificateurs de types - Pointeur

## Définition

- Un pointeur représente la position d'une variable dans la mémoire via l'opérateur &
- Il fournit un Accès indirect à la valeur associée via l'opération de **déréférencement**
- Contrairement à une référence, un pointeur peut ne rien référencer
- Contrairement à une référence, un pointeur peut changer de cible
- **L'immutabilité d'un pointeur est indépendante de celle de sa cible**

```
1  int var = 5;  // variable originale
2
3  int* pi               = &var; // pointeur vers un int
4  int const* pci        = &var; // pointeur vers un int immutable
5  int* const cpi        = &var; // pointeur immutable vers un int
6  int const* const cpci = &var; // pointeur immutable vers un int immutable
```

# Alias de type

**Objectifs:**

- Certains types ont un nom très long et rendent la lecture du code difficile
- Certains types sont complexes à exprimer
- C++ propose un système de renommage léger de type

```cpp
#include <array>

// using nouveau_type   = ancien_type
using     mass          = float;
using     point3D       = std::array<double, 3>;

// utilisation directe
point3D p1 = { 0,1.5,3. };
mass ppl_weight = 51.5f;
```

# Inférence de type

## Problèmatiques

- Déclarer une variable nécessite d'exprimer son type
- Dans le cas des initialisations, le type est déjà connu par le compilateur
- Dans d'autres contextes, le type d'une variable n'est pas exprimable par un humain

```
1  #include<vector>
2
3  float const f = 3.f;  // 3.f indique déjà qu'il s'agit d'un float
4
5  std::size_t sz = sizeof(f); // il faut se rappeler le type de retour de sizeof
6
7  std::vector<std::uint8_t> bytes(sz);
8
9  // Ce fragment de code est trop long pour indiquer que b est le début de bytes
10 std::vector<std::uint8_t>::iterator b = bytes.begin();
```

# Inférence de type

**Solution : le mot clé `auto`**

- `auto` remplace le type d'une variable lors d'une initialisation
- Le type de la variable est alors déduit comme le type de la valeur de son initialiseur
- `auto` est utilisable avec des qualificateurs

```
1  #include<vector>
2
3  auto const f = 3.f;
4
5  auto sz = sizeof(f);
6
7  std::vector<std::uint8_t> bytes(sz);
8
9  auto b = bytes.begin();
```

# Entrées/Sorties

# Contenu

**Modèles de flux**

- Flux console
- Flux de chaîne
- Flux de fichiers

**Manipulations de flux**

- Gestion des erreurs
- Formatage pour structure de données
- Gestion des fichiers

**Ce qu'il faudra retenir**

- Quel flux pour quelle tâche ?
- Gérer les I/O de manière sûre

# Modèles de flux

# Flux en C++

**Principes**

- Homogénéisation des entrées/sorties
- Cas classiques : écran, clavier, fichier
- Cas moins classiques : chaînes de caractères
- Conçu pour être extensible

**Interface :**

- Insertion dans un flux:

```
flux << valeur
```

- Extraction depuis un flux:

```
flux >> valeur
```

# Flux console

## Principes

- Fournir des objets capable de gérer les flux d'entrée/sortie basiques
- cran, clavier, affichage d'erreur
- Accessible via `#include <iostream>`

## Flux fournis

- `std::cin` : entrée clavier
- `std::cout` : sortie console
- `std::cerr` : sortie console d'erreur

# Flux console

## Mise en œuvre

```cpp
#include <iostream>

int main()
{
  std::cout << "Donnez un réel positif:\n";

  float x;
  std::cin >> x;

  if(x < 0.f)
    std::cerr << "Erreur : " << x << " est négatif\n";
}
```

# Flux de chaîne

## Principes

- Adaptation des chaînes de caractères comme flux
- Une chaîne contient des données extractibles
- Une chaîne peut être construite par insertion dans un flux
- Accessible via `#include <sstream>`

## Flux fournis

- `std::ostringstream` : insertion dans une chaîne
- `std::istringstream` : extraction dans une chaîne

# Flux de chaîne

## Mise en œuvre

```cpp
#include <iostream>
#include <string>
#include <sstream>

int main()
{
  std::ostringstream data;
  data << "'test: " << 3.54f << ' ' << 15784 << "'\n";

  std::string s = data.str();
  std::cout << s;
}
```

```
'test: 3.54 15784'
```

# Flux de chaîne

## Mise en œuvre

```cpp
#include <iostream>
#include <string>
#include <sstream>

int main()
{
  int mn, mx;
  double v;
  std::string name;

  std::istringstream in("9 -8 7.5 camera");
  in >> mx >> mn >> v >> name;

  std::cout << name << " " << v << " " << mn << " " << mx << "\n";
}
```

```
camera 7.5 -8 9
```

# Flux de chaîne

## Mise en œuvre

```cpp
#include <iostream>
#include <sstream>
#include <string>
#include <map>

int main()
{
  std::map<std::string, int> histogram;
  std::istringstream ss("des papous papa ont des poux papa papous");

  std::string w;
  while(ss >> w)
    histogram[w]++;

  for(auto p : histogram)
    std::cout << p.first << " -> " << p.second << "\n";
}
```

# Structures de Contrôles

# Contenu

**Flots de contrôle répétitifs**

- Flot déterministe
- Flot conditionnel

**Flots de contrôle conditionels**

- Bloc conditionnel
- Bloc de sélection

**Ce qu'il faudra retenir**

- Sémantique des structures de contrôle
- Organisation gros grain d'un programme

# Flots de controles répétitifs

# Flots déterministes

**Objectif:**

- Répéter un bloc de code un certain nombre de fois
- Le nombre de répétitions est connu à l'avance
- Repose sur l'utilisation d'une ou plusieurs variables de décompte

**Forme générale:**

```
for( initialisation; condition d'arrêt; mise à jour)
{
  // code à répéter
}
```

# Répétitions déterministes

## Exemple : somme de 1 à N

```cpp
#include <iostream>

int main()
{
  int resultat = 0, N = 10;

  for (int i = 1; i ≤ N; ++i)
  {
    resultat = resultat + i;
  }

  std::cout << "Somme(1,10) = " << resultat << "\n";
}
```

```
Somme(1,10) = 55
```

# Répétitions déterministes

## Exemple : mise à jour multiplicative

```cpp
#include <iostream>

int main()
{
  // MAJ multiplicative
  for (int i = 1; i < 65536; i *= 16)
  {
    std::cout << i << "\n";
  }
}
```

```
1
16
256
4096
```

# Répétitions déterministes

## Parcours d'une structure de type tableau

```cpp
#include <iostream>
#include<array>

int main()
{
  int resultat = 0;
  std::array<int, 10> vs = { 1,2,3,4,5,6,7,8,9,10 };

  for (int i = 1; i ≤ N; i++)
  {
    resultat = resultat + vs[i];
  }

  std::cout << "Somme(1,10) = " << resultat << "\n";
}
```

```
Somme(1, 10) = 55
```

# Répétitions déterministes

## Parcours d'une structure de type tableau

```cpp
#include <iostream>
#include <array>

int main()
{
  int resultat = 0;
  std::array<int, 10> vs = { 1,2,3,4,5,6,7,8,9,10 };

  for (int v : vs)
  {
    resultat = resultat + v;
  }

  std::cout << "Somme(1,10) = " << resultat << "\n";
}
```

```
Somme(1, 10) = 55
```

# Répétitions déterministes

## Parcours d'une structure de type tableau

```cpp
#include <iostream>
#include <array>

int main()
{
  int resultat = 0;
  std::array<int, 10> vs = { 1,2,3,4,5,6,7,8,9,10 };

  for (auto v : vs)
  {
    resultat = resultat + v;
  }

  std::cout << "Somme(1,10) = " << resultat << "\n";
}
```

```
Somme(1, 10) = 55
```

# Répétitions déterministes

## Parcours d'une structure de type tableau

```cpp
#include <iostream>
#include <array>

int main()
{
  double resultat = 0;
  std::array<double, 10> vs = { 1,2,3,4,5,6,7,8,9,10 };

  for (auto v : vs)
  {
    resultat = resultat + v;
  }

  std::cout << "Somme(1,10) = " << resultat << "\n";
}
```

```
Somme(1, 10) = 55
```

# Répétitions déterministes

## Parcours d'une structure de type tableau

```cpp
#include <iostream>
#include <vector>

int main()
{
  double resultat = 0;
  std::vector<double> vs{ 1,2,3,4,5,6,7,8,9,10 };

  for (auto v : vs)
  {
    resultat = resultat + v;
  }

  std::cout << "Somme(1,10) = " << resultat << "\n";
}
```

```
Somme(1, 10) = 55
```

# Répétitions conditionelles

**Objectif:**

- Répéter un bloc de code en dépendant d'une condition externe
- Le nombre de répétition n'est pas forcement connu à l'avance
- Repose sur l'utilisation d'une condition mise à jour manuellement

**Formes générales - 0+ itérations:**

```
while( condition )
{
  // code à répéter tant que condition s'évalue à `true`
}
```

# Répétitions conditionelles

**Objectif:**

- Répéter un bloc de code en dépendant d'une condition externe
- Le nombre de répétition n'est pas forcement connu à l'avance
- Repose sur l'utilisation d'une condition mise à jour manuellement

**Formes générales - 1+ itérations:**

```
do
{
  // code à répéter tant que condition s'évalue à `true`
} while( condition );
```

## Exemple - Recherche de valeurs

```cpp
#include <iostream>
#include <vector>

int main()
{
  std::vector<float> x = { 3, 4, 5, 8, 7, 99, 32, 14, 5, 88, 6, 12, 1, 54, 7};

  std::size_t idx = 0;
  while (idx < x.size() && x[idx] != 1)
  {
    idx++;
  }

  if ( idx < x.size() )
    std::cout << "valeur 1 trouvée a l'index " << idx << "\n";
}
```

```
valeur 1 trouvée a l'index 12
```

# Répétitions conditionels

## Exemple - Somme interactive

```cpp
#include <iostream>

int main()
{
  float n, sum = 0.f;

  do
  {
    std::cin >> n;

    sum += n;
  } while (n != 0.f);

  std::cout << sum << "\n";
}
```

# Flots Conditionnels

# Structure conditionelle - `if ... else`

**Objectif:**

- **Permettre de modifier le flot du programme en fonction d'une condition arbitraire**
- La modification du flot peut être associée à une alternative
- Les conditions de choix du flot peuvent s'enchaîner

**Forme générale:**

```
1  // Code prologue exécuté systématiquement
2
3  if( condition )
4  {
5    // code exécuté si et seulement si `condition` s'évalue à `true`
6  }
7
8  // Code epilogue exécuté systématiquement
```

# Structure conditionelle - `if ... else`

**Objectif:**

- Permettre de modifier le flot du programme en fonction d'une condition arbitraire
- **La modification du flot peut être associée à une alternative**
- Les conditions de choix du flot peuvent s'enchaîner

**Forme générale:**

```
if( condition )
{
  // code exécuté si et seulement si `condition` s'évalue à `true`
}
else
{
  // code exécuté si et seulement si `condition` s'évalue à `false`
}
```

# Structure conditionelle - `if ... else`

**Objectif:**

- Permettre de modifier le flot du programme en fonction d'une condition arbitraire
- La modification du flot peut être associée à une alternative
- **Les conditions de choix du flot peuvent s'enchaîner**

**Forme générale:**

```
if( condition1 )      { /* code exécuté si `condition1` s'évalue à `true` */ }
else if( condition2 ) { /* sinon code exécuté si `condition2` s'évalue à `true`*/ }
else if( condition3 ) { /* sinon code exécuté si `condition3` s'évalue à `true`*/ }
else
{
  // code exécuté en dernier recours
}
```

# Structure conditionelle - `if ... else`

**Exemple:**

```cpp
#include <iostream>

int main()
{
  int n;

  std::cout << "Donner un entier: ";
  std::cin >> n;  // lecture d'un entier au clavier

  if(n % 2 == 0)  std::cout << n << " est pair.\n";
  else            std::cout << n << " est impair.\n";
}
```

# Structure conditionelle - `if ... else`

**Exemple:**

```
1  #include <iostream>
2
3  int main()
4  {
5    int n;
6
7    std::cout << "Donner un entier: ";
8    std::cin >> n;  // lecture d'un entier au clavier
9
10   if(n == 0)          std::cout << n << " est nul.\n";
11   else if(n % 2 == 0) std::cout << n << " est pair.\n";
12   else                std::cout << n << " est impair.\n";
13 }
```

# Structure conditionelle - `if ... else`

## Exemple - `if` imbriqués :

```
1  #include <iostream>
2
3  int main()
4  {
5    int n;
6
7    std::cout << "Donner un entier: ";
8    std::cin >> n;  // lecture d'un entier au clavier
9
10   if(n % 2 == 0)
11   {
12     if(n == 0)  std::cout << n << " est nul.\n";
13     else        std::cout << n << " est pair.\n";
14   }
15   else          std::cout << n << " est impair.\n";
16 }
```

# Structure sélective - `if ... else`

## Initialisation interne

- On peut initialiser une variable là où la condition est attendue
- La variable ne sera donc accessible que dans les diverses alternatives du `if`
- Sans code supplémentaire, la condition implicte est que la variable est différente de 0.

```
1  // i n'existe pas encore ici
2  if( int i = roll(0,10) )
3  {
4    std::cout << i << " est non-nul\n"; // i est accessible
5  }
6  else
7  {
8    std::cout << i << " est nul\n"; // i est accessible
9  }
10
11 // i n'existe plus là
```

# Structure sélective - `if ... else`

## Initialisation interne

- On peut initialiser une variable là où la condition est attendue
- La variable ne sera donc accessible que dans les diverses alternatives du `if`
- Sans code supplémentaire, la condition implicte est que la variable est différente de 0.

```
1  // i n'existe pas encore ici
2  if( int i = roll(0,10); i > 5 )
3  {
4    std::cout << "Gagné ! " << i " > 5\n"; // i est accessible
5  }
6  else
7  {
8    std::cout << "Perdu :( " << i " ≤ 5\n"; // i est accessible
9  }
10
11 // i n'existe plus là
```

# Structure sélective - switch

**Objectif :**

- Modifie le flot de contrôle en sélectionnant un cas parmi N
- Les cas sont décrits comme une énumération de possibles
- Le type de choix doit être un entier ou une énumération

**Forme générale :**

```
switch( valeur )
{
  case cas1 : // code du cas 1
  break;

  case cas2 : // code du cas 2
  break;

  default : // code par défaut
}
```

# Structure sélective - switch

**Exemple :**

```
int jour;
std::cin >> jour;  // lit le n° du jour au clavier

switch (jour)
{
  case 1: std::cout << "lundi\n";     break;
  case 2: std::cout << "mardi\n";     break;
  case 3: std::cout << "mercredi\n";  break;
  case 4: std::cout << "jeudi\n";     break;
  case 5: std::cout << "vendredi\n";  break;
  case 6: std::cout << "samedi\n";    break;
  case 7: std::cout << "dimanche\n";    break;
  default: std::cout << "jour impossible\n";
}
```

# Structure sélective - switch

**Exemple - Sémantique de *fallthrough*:**

```
1  int jour;
2  std::cin >> jour;  // lit le n° du jour au clavier
3
4  // Le mardi : cravate rouge
5  // le vendredi : cravate grise
6  // le week-end : cravate bleue
7  // sinon : pas de cravate
8  switch (jour)
9  {
10   case 2: std::cout << "cravate rouge\n"; break;
11   case 5: std::cout << "cravate grise\n"; break;
12   case 6:
13
14   case 7: std::cout << "cravate bleue\n"; break;
15   default: std::cout << "pas de cravate\n";
16 }
```

# Structure sélective - switch

**Exemple - Sémantique de *fallthrough*:**

```
1  int jour;
2  std::cin >> jour;  // lit le n° du jour au clavier
3
4  switch (jour)
5  {
6    case 2: std::cout << "cravate rouge\n"; break;
7    case 5: std::cout << "cravate bleue\n"; break;
8    case 6:
9    [[fallthrough]];
10   case 7: std::cout << "cravate grise\n"; break;
11
12   default: std::cout << "pas de cravate\n";
13 }
```

# Structure sélective - switch

## Exemple - Interaction avec les enum:

```
1  enum class jour { lundi = 1, mardi, mercredi, jeudi, vendredi, samedi, dimanche};
2
3  jour j = current_day();
4
5  switch (jour)
6  {
7    case jour::mardi:    std::cout << "cravate rouge\n"; break;
8    case jour::vendredi: std::cout << "cravate bleue\n"; break;
9
10   case jour::samedi:
11   [[fallthrough]];
12   case jour::dimanche: std::cout << "cravate grise\n"; break;
13
14   default: std::cout << "pas de cravate\n";
15 }
```

# Structure sélective - switch

## Exemple - Initialisation interne:

```
1  enum class jour { lundi = 1, mardi, mercredi, jeudi, vendredi, samedi, dimanche};
2
3  switch (jour j = current_day())
4  {
5    case jour::mardi:    std::cout << "cravate rouge\n"; break;
6    case jour::vendredi: std::cout << "cravate bleue\n"; break;
7
8    case jour::samedi:
9    [[fallthrough]];
10   case jour::dimanche: std::cout << "cravate grise\n"; break;
11
12   default: std::cout << "pas de cravate\n";
13 }
14
15 // j n'est plus accessible
```

# Types Structurés

# Types Structurés - Énumérations

## Principes

- Une énumération abstrait un ensemble discret de valeurs numériques
- Ces valeurs partagent une même sémantique

## Cas d'usage

- Remplacer des valeurs “magiques”
- Création de types entiers à sémantique forte

## Forme générale

```
enum class identifiant { valeur1, …, valeurN };
```

## Remplacement de valeurs magiques

```
1  // Code classique à valeur magique
2  int main()
3  {
4    int jour = 4;
5
6    switch(jour)
7    {
8      case 1 : std::cout << "on est lundi !\n"; break;
9      case 2 : std::cout << "on est mardi !\n"; break;
10     case 3 : std::cout << "on est mercredi !\n"; break;
11     case 4 : std::cout << "on est jeudi !\n"; break;
12     case 4 : std::cout << "on est vendredi !\n"; break;
13     default: std::cout << "on est le week-end !\n"; break;
14   }
15 }
```

## Remplacement de valeurs magiques

```cpp
// Code à base d'énumérations
enum class jour { lundi = 1 , mardi, mercredi, jeudi, vendredi, samedi, dimanche};

int main()
{
  jour j = jour::jeudi;

  switch(j)
  {
    case jour::lundi    : std::cout << "on est lundi !\n"; break;
    case jour::mardi    : std::cout << "on est mardi !\n"; break;
    case jour::merdredi : std::cout << "on est mercredi !\n"; break;
    case jour::jeudi    : std::cout << "on est jeudi !\n"; break;
    case jour::vendredi : std::cout << "on est vendredi !\n"; break;
    default: std::cout << "on est le week-end !\n"; break;
  }
}
```

## Remplacement de valeurs magiques

```cpp
// Code à base d'énumérations - Aspect error-proof
enum class jour { lundi = 1 , mardi, mercredi, jeudi, vendredi, samedi, dimanche};

int main()
{
  jour j = 5;  // ne compile pas car jour n'est pas un int

  switch(j)
  {
    case jour::lundi    : std::cout << "on est lundi !\n"; break;
    case jour::mardi    : std::cout << "on est mardi !\n"; break;
    case jour::merdredi : std::cout << "on est mercredi !\n"; break;
    case jour::jeudi    : std::cout << "on est jeudi !\n"; break;
    case jour::vendredi : std::cout << "on est vendredi !\n"; break;
    default: std::cout << "on est le week-end !\n"; break;
  }
}
```

# Types Structurés - Énumérations

## Remplacement de valeurs magiques

- Il est possible de spécifier le type d'entier utilisé pour stocker la valeur des énumérés
- Le compilateur vérifie que les valeurs utilisés sont bien représentables dans le type utilisé

```
1  // Code à base d'enumérations
2  enum class jour : std::uint8_t
3  {
4    lundi = 1 , mardi, mercredi, jeudi, vendredi, samedi, dimanche
5  };
6
7  jour j = jour::jeudi;
```

# Types Structurés - Énumérations

## Remplacement de valeurs magiques

- Il est possible de spécifier le type d'entier utilisé pour stocker la valeur des énumérés
- Le compilateur vérifie que les valeurs utilisés sont bien représentables dans le type utilisé

```
// Code à base d'Énumérations
enum class jour : std::uint8_t
{
  lundi = 1 , mardi, mercredi, jeudi, vendredi, samedi
  // error: enumerator value '1500' is outside the range of underlying type 'unsigned char'
  , dimanche = 1500
};

jour j = jour::jeudi;
```

# Types Structurés - Énumérations

## Type entier à sémantique forte

- Il est possible de spécifier le type d'entier utilisé pour stocker la valeur des énumérés
- Il est possible de ne pas fournir d'énuméré
- Ces deux régles permettent de construire des **types entiers à sémantique forte**

```
1  // Code à base d'enumérations
2  enum class age : std::uint8_t {};
3
4  // La construction d'un âge se fait explicitement à partir d'un entier
5  age age_bebe{};   //  âge_bebe vaut 0
6  age age_ado{12};  //  âge_ado vaut 12
7
8  // Les ages et les entiers ne se melangent pas : Ces codes ne compilent pas :
9  age age_papy = 90;    // error: cannot convert 'int' to 'age' in initialization
10 int n = age_ado / 2;  // error: no match for 'operator/' (operand types are 'age' and 'int')
```

# Types Structurés - Aggrégats

**Principes**

- Définit une entité avec une sémantique précise
- Les éléments d'un aggrégat sont ses **données membres**
- Permet l'ajout de nouveaux types au langage

**Eléments primordiaux**

- Définir un aggrégat
- Connaître les comportements par défauts

# Types Structurés - Aggrégats

## Forme générale

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

  // etc ....

  type_memberN membre_N;
};
```

- Le nombre et le type des membres est arbitraire
- Le nom de l'aggrégat ou de ses membres est arbitraire
- Un aggrégat peut contenir un ou plusieurs aggrégats

# Types Structurés - Aggrégats

## Forme générale

```
1  // Aggregat simple
2  struct point3D
3  {
4    double x; // x est un membre de type double de point3D
5    double y;
6    double z;
7  };
8
9  // Aggregat composite
10 struct point3D_pondere
11 {
12   point3D point;  // Notez la réutilsiation de point3D
13   int     poids;  // poids est un membre supplémentaire
14 };
```

## 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 aggregats 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
```

# Aspects Impératifs

# Contenu

**Notion de fonctions**

- Gestion des paramètres
- Retour multiples
- Interaction avec les structures

**Polymorphisme fonctionnel**

- Surcharge de fonctions et d'opérateurs
- Fonctions génériques

# Notion de fonctions

# Notion de fonctions

## Principes de base

- Une fonction encapsule du savoir-faire
- C'est un groupe d'expressions réutilisables
- Une fonction consomme des **paramètres** pour produire une **valeur de retour**
- Utiliser une fonction passe par un **appel de fonction**

## Forme générale

```
type_de_retour nom_fonction( type_1 parametre_1, ..., type_N parametre_N )
{
  expression;
  // ...
  expression;

  return valeur_de_retour;
}
```

## Exemple

```cpp
#include <iostream>

// squared_difference consomme deux int et retourne un nouvel int
int squared_difference(int a, int b)
{
  auto diff = a - b;    // Expression
  return diff * diff;   // Retour de la fonction
}

int main()
{
  auto x = squared_difference(6, 10); // Appel à la fonction avec 6 et 10 en paramètre
  std::cout << x << "\n";             // x contient la valeur retournée par la fonction
}
```

```
16
```

## Exemple

```cpp
#include <iostream>

// void indique que la fonction ne renvoie rien
void tell_me(float x)
{
  std::cout << "Valeur de x = " << x << "\n";
  // pas de return
}

int main()
{
  auto u = 3.14159f;
  tell_me(u);
}
```

```
Valeur de x = 3.14159
```

### Fonction récursive

- Une fonction peut s'appeler elle-même
- Il est nécessaire de gérer manuellement son ou ses cas terminaux

### Exemple

```
1  int fibonacci(int n)
2  {
3    if(n<2) return 1; else return fibonacci(n-1) + fibonacci(n-2);
4  }
5
6  int main()
7  {
8    std::cout << fibonacci(10) << "\n";
9  }
```

```
89
```

# Passage des paramètres

## Passage par valeur

- La fonction reçoit une **copie** de ses paramètres
- Leurs modifications ne sont pas répercutées sur les paramètres eux-même

```
1  void f(double a)
2  {
3    a /= 2.5f;
4    std::cout << a << "\n";   //  Affiche 10.3
5  }
6
7  int main()
8  {
9    double x = 25.75;
10   std::cout << x << "\n";   //  Affiche 25.75
11   f(x);
12   std::cout << x << "\n";   //  Affiche 25.75
13 }
```

# Passage des paramètres

## Passage par référence

- La fonction reçoit une **référence** vers ses paramètres
- Leurs modifications sont répercutées sur les paramètres en eux-même

```
1  void f(double& a)
2  {
3    a /= 2.5f;
4    std::cout << a << "\n";   //  Affiche 10.3
5  }
6
7  int main()
8  {
9    double x = 25.75;
10   std::cout << x << "\n";   //  Affiche 25.75
11   f(x);
12   std::cout << x << "\n";   //  Affiche 10.3
13 }
```

# Passage des paramètres

## Passage par référence immuable

- La fonction reçoit une **référence immuable** vers ses paramètres
- Les paramètres ne peuvent pas être modifiés

```
1  void f(double const& a)
2  {
3    // a /= 2.5f;  ⟶ erreur de compilation
4    std::cout << a << "\n";   //  Affiche 25.75
5  }
6
7  int main()
8  {
9    double x = 25.75;
10   std::cout << x << "\n";   //  Affiche 25.75
11   f(x);
12   std::cout << x << "\n";   //  Affiche 25.75
13 }
```

# Passage des paramètres

## Cas particulier du paramètre pointeur

- Signale que le paramètre peut être omis
- Comportement similaire à une référence

```
1  void f(double* a)
2  {
3    if(a ≠ nullptr)
4      *a /= 2.5f;
5  }
6
7  int main()
8  {
9    double x = 25.75;
10   std::cout << x << "\n";   //  Affiche 25.75
11   f(&x);                    //  Passage de l'adresse
12   std::cout << x << "\n";   //  Affiche 10.3
13 }
```

# Paramètres par défaut

## Principe

- Il est possible de spécifier une valeur par défaut à un ou plusieurs paramètres
- Ces paramètres doivent être groupés en fin de liste des paramètres

```
1  #include <iostream>
2
3  void display(bool cursor, char c = '*', int n = 1)
4  {
5    if(cursor)
6      std::cout << "> ";
7
8    for (int i = 1; i ≤ n; ++i)
9      std::cout << c;
10
11   std::cout << "\n";
12 }
```

# Paramètres par défaut

## Principe

- Il est possible de spécifier une valeur par défaut à un ou plusieurs paramètres
- Ces paramètres doivent être groupés en fin de liste des paramètres

```
1  int main()
2  {
3    display(true,'+',3);
4    display(false,'$',5);
5    display(false,'!');
6    display(true);
7  }
```

```
>+++
$$$$$
!
> *
```

# Interaction avec les structures

## Passage d'un agrégat en paramètre de fonction

- Les agrégats se passent en paramètres de manière naturelle
- Cas particulier du cas paramètre pointeur

```
1  struct fraction
2  {
3    int num, denum;
4  };
5
6  float value(fraction a)
7  {
8    auto div = static_cast<float>(a.denum); // Conversion de a.denum en float
9    return a.num / div;
10 }
```

# Interaction avec les structures

## Passage d'un agrégat en paramètre de fonction

- Les agrégats se passent en paramètres de manière naturelle
- **Cas particulier du cas paramètre pointeur**

```
1  struct fraction
2  {
3    int num, denum;
4  };
5
6  float value_or(fraction* a, float v)
7  {
8    if( a != nullptr )                          // a est il disponible ?
9      return a->num / static_cast<float>(a->denum); // -> remplace .
10   else                                        // sinon renvoyons le 2e paramètre
11     return v;
12 }
```

# Interaction avec les structures

## Passage d'un agrégat en paramètre de fonction

- Les agrégats se passent en paramètres de manière naturelle
- **Cas particulier du cas paramètre pointeur**

```cpp
#include <iostream>

int main()
{
  fraction r{1,4};
  std::cout << value_or(&r, 0.f) << "\n";
  std::cout << value_or(nullptr, 13.37f) << "\n";
}
```

```
0.25
13.37
```

# Interaction avec les structures

## Renvoyer une structure depuis une fonction

- Explicite ou implicite
- Directe ou par morceaux

```
// Retour par construction implicite
fraction identity()
{
  return {0,1};
}

// Retour par construction explicite
fraction identity()
{
  return fraction{0,1};
}
```

# Interaction avec les structures

## Renvoyer une structure depuis une fonction

```
1  // Retour par construction directe implicite
2  fraction multiply(fraction a, fraction b)
3  {
4    return {a.num*b.num, a.denum*b.denum};
5  }
6
7  // Retour par construction par morceaux
8  fraction add(fraction a, fraction b)
9  {
10   fraction that;
11
12   that.num    = a.num*b.denum + a.denum*b.num;
13   that.denum  = a.denum * b.denum;
14
15   return that;
16 }
```

# Interaction avec les structures

## Notion de fonction membre

- Une fonction peut avoir une sémantique qui la rend indissociable du type qu'elle manipule
- Pour des raisons de clarté, le **code** de la fonction doit être associé au **code** de l'agrégat
- On définit alors une **fonction membre**

```
1  fraction x{178, 16};
2
3  x.simplify();   //  x est simplifié "en place"
```

## Problématiques

- Définition syntaxique d'une fonction membre
- Propriété des fonctions membres vis à vis de l'agrégat

# Interaction avec les structures

## Syntaxe des fonctions membres

```
1  struct fraction
2  {
3    int num, denum;
4
5    // Déclaration
6    void scale(int factor);
7  };
8
9  // Définition - Remarquer le préfixe fraction:: qui associe scale au type fraction
10 void fraction::scale(int factor)
11 {
12   // le num utilisé ici est celui de l'agrégat appelant cette fonction membre
13   num *= factor;
14 }
```

# Interaction avec les structures

## Syntaxe des fonctions membres

```
1  struct fraction
2  {
3    int num, denum;
4
5    // Définition "en-ligne"
6    // Si le code de la fonction est trivial, il est acceptable de définir
7    // la fonction directement au sein de l'agrégat
8    void scale(int factor)
9    {
10     num *= factor;
11   }
12 };
```

# Interaction avec les structures

## Syntaxe des fonctions membres

```
1  int main()
2  {
3    fraction x{17,23};
4
5    std::cout << x.num << " / " << x.denum << "\n";
6
7    x.scale(10);
8    std::cout << x.num << " / " << x.denum << "\n";
9  }
```

```
17 / 23
170 / 23
```

# Interaction avec les structures

## Notion de fonction immuable

- Comment respecter l'immutabilité d'une instance d’agrégat ?

```
1  struct fraction
2  {
3    int num, denum;
4
5    double value() { return num / static_cast<double>(denum); }
6  };
7
8  fraction x{4,7};
9  auto v = x.value(); //  OK
```

# Interaction avec les structures

## Notion de fonction immuable

- Comment respecter l'immutabilité d'une instance d’agrégat ?

```
1  struct fraction
2  {
3    int num, denum;
4
5    double value() { return num / static_cast<double>(denum); }
6  };
7
8  fraction const x{4,7};
9  // Erreur de compilation:
10 //  error: 'this' argument to member function 'value' has type 'const fraction'
11 //  but function is not marked const
12 //
13 auto v = x.value();
```

# Interaction avec les structures

## Notion de fonction immuable

- Comment respecter l'immutabilité d'une instance d'agrégat ?

```
1  struct fraction
2  {
3    int num, denum;
4
5    // value() ne modifie pas l'état de fraction
6    // Elle est donc compatible avec le caractère immuable d'une instance de fraction.
7    // Il est nécessaire d'indiquer cela via le mot-clé const
8    //                                              |
9    //              +-----------------------------------
10   //              v
11   double value() const { return num / static_cast<double>(denum); }
12 };
13
14 fraction const x{4,7};
15 auto v = x.value();     //  OK
```

# Retour multiple

## Problématique

- Certaines fonctions ont besoin de renvoyer plusieurs résultats
- Comment spécifier ce retour ?
- Comment récupérer ces valeurs ?

```
/* ??? */  quotient_reste( int a, int b )
{
  return /* ??? */;
}
```

## Solutions

- Renvoi d'un agrégat
- Utilisation de `std::tie`
- Décomposition structurelle

# Retour multiple

## Renvoi d'un agrégat

- Simple à utiliser
- Nécessite un type agrégat par fonction

```
1  struct sortie_qr
2  {
3    int quotient, reste;
4  };
5
6  sortie_qr quotient_reste( int a, int b )
7  {
8    return {a/b, a%b};
9  }
10
11 auto x = qotient_reste(17,4);
12 std::cout << x.quotient << " " << x.reste << "\n";
```

# Retour multiple

## Utilisation de `std::tie`

- Simple à utiliser
- Les sous-morceaux n'ont pas de sémantique claire

```
1  #include <tuple>
2
3  std::tuple<int,int> quotient_reste( int a, int b )
4  {
5    return {a/b, a%b};
6  }
7
8  int q,r;
9  std::tie(q,r) = qotient_reste(17,4);
10 std::cout << q << " " << r << "\n";
```

# Retour multiple

## Décomposition structurelle

- Syntaxe: `auto [nom1,nom2,...,nomN] = f();`
- `f` doit renvoyer une valeur similaire à un tuple ou à un agrégat
- Cette syntaxe **définit** des variables au lieu d'en réutiliser

```
1  #include <tuple>
2
3  std::tuple<int,int> quotient_reste( int a, int b )
4  {
5    return {a/b, a%b};
6  }
7
8  auto [q,r] = quotient_reste(17,4);
9  std::cout << q << " " << r << "\n";
```

# Polymorphisme fonctionnel

# Surcharge de fonction

**Motivations**

- Pourquoi des fonctions effectuant une opération similaire sur des types différents auraient un nom différent ?
- Simplifier l'usage du code en le rendant intuitif
- Premiers pas vers une notion de généricité

**Notion de polymorphisme *ad-hoc***

- Capacité d'associer un **nom de fonction** à différents comportements
- Mise en place d'un système de résolution basé sur les types
- Faire cohabiter dans un même programme des fonctions aux noms identiques

# Surcharge de fonction

## Principes

- Une fonction en C++ est identifiable de manière unique par sa **signature**
- Signature = Nom de la fonction + Liste des types de ses paramètres
- Plusieurs fonctions de même **nom** mais de signatures **différentes** peuvent donc coexister

> Le type de retour ne joue aucun rôle dans la signature d'une fonction

## Exemple

```
1  double f();               // signature: f + { }
2  double f(int a);          // signature: f + { int }
3  char   f(char x, char y); // signature: f + { char, char}
4
5  int f();                  // signature: f + { } -- collision avec double f()
```

# Surcharge de fonction

## Mise en place

```
1  std::string as_text(std::string const& s) { return "'" + s + "'";     }
2  std::string as_text(char  c)               { return std::string(1, c); }
3  std::string as_text(float f)
4  {
5    std::ostringstream o;
6    o << f << "f";
7    return o.str();
8  }
9
10 std::cout << as_text('Z')                 <<  "\n";
11 std::cout << as_text(3.5f)                <<  "\n";
12 std::cout << as_text("un test de texte")  <<  "\n";
```

```
Z
3.5f
'un test de texte'
```

# Surcharge d'opérateurs

**Motivation**

- Permettre de donner aux types définis par l'utilisateur une interface naturelle
- Rendre les types utilisateurs transparents au langage

**Principes**

- Un opérateur est une fonction avec un syntaxe infixe

```
a = b + c => operator=(a, operator+(b,c))
```

- Les opérateurs sont surchargeables comme des fonctions à partir du moment où au moins un paramètre est un type utilisateur

# Surcharge d'opérateurs - Opérateurs surchargeables

| Opérations | Opérateurs | Forme générale |
|---|---|---|
| Unaire logique | `!` | `bool operator!(T a)` |
| Unaire arithmétique | `-,~` | `R operator<@>(T a)` |
| Binaire arithmétique | `+,-,*,/,%` | `R operator<@>(T a, U b)` |
| Binaire bits à bits | `≪,≫,^,&,\|` | `R operator<@>(T a, U b)` |
| Comparaison | `=,≠,<,>,≤,≥` | `bool operator<@>(T a, U b)` |
| Logique | `&&,\|\|` | `bool operator<@>(T a, U b)` |

Le comportement d'un opérateur surchargé peut être quelconque.
Il est de bon ton de conserver une sémantique proche de l'original.

# Surcharge d'opérateur

## Exemple

```
struct fraction
{
  int num, denum;
};

fraction operator-(fraction const& arg)
{
  return { -arg.num, arg.denum };
}

fraction operator+(fraction const& lhs, fraction const& rhs)
{
  auto num    = lhs.num*rhs.denum + lhs.denum*rhs.num;
  auto denum  = lhs.denum*rhs.denum;

  return {num, denum};
}
```

# Surcharge d'opérateur

## Exemple

```
fraction operator+(fraction const& lhs, int rhs)
{
  auto num    = lhs.num + lhs.denum*rhs;
  auto denum  = lhs.denum;

  return {num, denum};
}

fraction operator+(int lhs, fraction const& rhs)
{
  auto num    = lhs*rhs.denum + lhs*rhs.num;
  auto denum  = rhs.denum;

  return {num, denum};
}
```

# Surcharge d'opérateur - Interaction avec les flux

## Exemple

```
#include <iostream>

std::ostream& operator<<(std::ostream& os, fraction const& f)
{
  os << f.num << "/" << f.denum;
  return os;
}

std::istream& operator>>(std::istream& is, fraction& f)
{
  is >> f.num  >> f.denum;
  return is;
}
```

# Fonctions génériques

## Motivation

- Large quantité de surcharge pour une fonction donnée
- Code extrémement 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 retracer *a psoteriori*.

# 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 dit des *templates* de fonctions

```
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 apramètriques
- L'appel de la fonction force le compilateur à déduire le type

effectif de ces paramètres en analysant le type des valeurs passer en paramètres. * Le code exact est regénéré 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 complexes à exprimer voire impossibles à déterminer par le développeur
- Le compilateur a toutes les informations nécessaire 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`

## Mode 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 etre calculer plus tard
// |
// v
  auto  addition(T1 a, T2 b)
        -> decltype(a+b)  // <-- decltype évalue le type de 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 etre calculer par le compilateur
// |
// v
  auto  addition(T1 a, T2 b)
  {
    // Le compilateur évalue le type de l'expression renvoyé par return
    return a + b;
  }
```

# La Bibliothèque Standard

# Algorithmes

## Objectifs

- Rendre la manipulation de séquence de données naturelle
- Fournir des algorithmes non triviaux
- Augmenter l'expressivité du code et sa maintenabilité
- Augmenter la cohésion du code

## Solutions

- Algorithmes standard
- Fonction anonymes

## Mise en situation - Code initiale

```
1  bool match_pattern(MemoryBuffer const& m)
2  {
3    return m.size() > 2 && m[0] == 'E' && m[1] == 'Z';
4  }
5
6  bool process_buffer(std::vector<MemoryBuffer> const& mems)
7  {
8    for(std::size_t i = 0; i< mems.size(); ++i)
9    {
10     if( match_pattern(mems[i]) )
11       return true;
12   }
13
14   return false;
15 }
```

## Mise en situation - Boucle d'intervalle

- Abstraction du processus d'itération
- Code généralisé pour tout conteneur

```
1  bool match_pattern(MemoryBuffer const& m)
2  {
3    return m.size() > 2 && m[0] == 'E' && m[1] == 'Z';
4  }
5
6  bool process_buffer(std::vector<MemoryBuffer> const& mems)
7  {
8    for(auto const& m : mems)
9    {
10     if( match_pattern(m) ) return true;
11   }
12
13   return false;
14 }
```

# Algorithmes comme abstraction de boucle

## Objectifs

- Fournir une implémentation de références des traitements classiques sur des séquences
- Abstraction complète du type de la séquence
- Maximiser la généricité du code manipulant une séquence

## Mise en pratique

- Notion d'**itérateur**
- Implémentation optimisée à la compilation
- Large gamme d'algorithmes

## Mise en situation - Itérateurs

```
1  bool match_pattern(MemoryBuffer const& m)
2  {
3    return m.size() > 2 && m[0] == 'E' && m[1] == 'Z';
4  }
5
6  bool process_buffer(std::vector<MemoryBuffer> const& mems)
7  {
8    std::vector<MemoryBuffer>::const_iterator b = mems.cbegin();
9    std::vector<MemoryBuffer>::const_iterator e = mems.cend();
10
11   while(b != e)
12   {
13     if( match_pattern(*b++) ) return true;
14   }
15
16   return false;
17 }
```

# Algorithmes standards

### Prédicats

- `all_of`, `any_of`, `none_of`
- `count`, `count_if`

### Modification

- `transform`, `for_each`
- `copy`, `replace`, `remove`
- `copy_if`, `replace_if`, `remove_if`

### Recherche

- `find`, `find_if`
- `search`, `search_if`
- `max_element`, `min_element`, `minmax_element`

# Algorithmes standards

### Calculs numériques

- `accumulate`, `reduce`, `transform_reduce`
- `inclusive_scan`, `exclusive_scan`
- `inner_product`

### Modification structurelle

- `reverse`, `rotate`, `shuffle`, `sample`
- `unique`, `fill`

### Tri

- `sort`, `stable_sort`, `partial_sort`, `nth_element`
- `partition`, `stable_partition`
- `is_sorted`, `is_partitioned`

## Mise en situation - Algorithme

```
1  bool match_pattern(MemoryBuffer const& m)
2  {
3    return m.size() > 2 && m[0] == 'E' && m[1] == 'Z';
4  }
5
6  bool process_buffer(std::vector<MemoryBuffer> const& mems)
7  {
8    auto it = std::find_if(mems.begin(), mems.end(), match_pattern);
9    return it != mems.end();
10 }
11
12 bool process_buffer2(std::vector<MemoryBuffer> const& mems)
13 {
14   return std::any_of(mems.begin(), mems.end(), match_pattern);
15 }
```

# Fonctions anonymes

**Limitations de l'interface des algorithmes**

- Passer une fonction en paramètre est limité
- Impossible d'utiliser une fonction surchargée
- Impossible d'utiliser une fonction générique

**Solutions**

- Objets fonctions
- **Fonctions anonymes**

# Fonctions anonymes - Syntaxe

## Définition d'une fonction anonyme

- Une fonction anonyme est l'équivalent fonction d'une variable temporaire
- Elle définit un fragment de code utilisable “sur place”

```
1    [ ](int a, float b, auto c)
2    {
3      auto x = (a+b)/2;
4      return x<c ? c : x;
5    }
```

# Fonctions anonymes - Syntaxe

## Définition d'une fonction anonyme

- Une fonction anonyme contient du code arbitraire comme une fonction normale
- Seule limitation, elle ne peut pas s'appeler récursivement

```
1    [ ](int a, float b, auto c)
2    {
3      // Corps de la fonction contenant un code arbitraire
4      auto x = (a+b)/2;
5      return x<c ? c : x;
6    }
```

# Fonctions anonymes - Syntaxe

## Définition d'une fonction anonyme - Capture

- La définition d'une fonction anonyme commence par son **environement de capture*
- Cet environement liste les variables utilisées à l'intérieur de la fonction mais définies à l'extérieur.
- Par défaut, cet environement est vide

```
1  //  --- Environement de capture
2  // |
3  // |
4  // v
5    [ ](int a, float b, auto c)
6    {
7      auto x = (a+b)/2;
8      return x<c ? c : x;
9    }
```

# Fonctions anonymes - Syntaxe

## Définition d'une fonction anonyme - Capture

- On liste dans l'environement les variables via leur nom
- On peut capturer une variable par référence en utilisant `&`
- `[=]` signifie que l'on capture tout par valeur
- `[&]` signifie que l'on capture tout par référence

```
1    int u = 9;
2    int k = 0;
3
4    [u,&k](int a, float b, auto c)
5    {
6      auto x = (a+b)/2;
7      k++;                // Le k extérieur est modifié
8
9      return x<c ? u : x; // Ce u est une copie du u extérieur
10   }
```

# Fonctions anonymes - Syntaxe

## Définition d'une fonction anonyme - Paramètres

- Une fonction anonyme peut avoir un nombre arbitraire de paramètres
- Ils définissent comment la fonction anonyme sera appelée

```
1  //     Liste de paramètres
2  //             |
3  //             v
4  //    <-------------------->
5    [ ](int a, float b, auto c)
6    {
7      auto x = (a+b)/2;
8      return x<c ? c : x;
9    }
```

# Fonctions anonymes - Syntaxe

## Définition d'une fonction anonyme - Paramètres

- Les paramètres peuvent avoir un type concret
- Les paramètres peuvent être générique. On utilise alors **auto**

```
1  //   Paramétres de type concrets
2  //             |
3  //      +------+
4  //      v      v
5    [ ](int a, float b, auto c)
6    {
7      auto x = (a+b)/2;
8      return x<c ? c : x;
9    }
```

# Fonctions anonymes - Syntaxe

## Définition d'une fonction anonyme

- Les paramètres peuvent avoir un type concret
- Les paramètres peuvent être générique. On utilise alors **auto**

```
1  //   Paramétre de type générique
2  //                      |
3  //                      |
4  //                      v
5    [ ](int a, float b, auto c)
6    {
7      auto x = (a+b)/2;
8      return x<c ? c : x;
9    }
```

# Fonctions anonymes - Utilisation

## Interaction avec les fonctions classiques

- Une fonction anonyme n'a pas de type humainement lisible
- Prendre une fonction anonyme en paramètre nécessite un paramètre template

```
1  template<typename Func>
2  void display_twice(Func f, int x)
3  {
4    std::cout << f(x) << "\n" << f(x+1) << "\n";
5  }
6
7  int main()
8  {
9    display_twice( [](auto i) { return 1.f/i; }, 4 );
10 }
```

```
0.25
0.2
```

# Fonctions anonymes - Utilisation

## Interaction avec les fonctions classiques

- Une fonction anonyme n'a pas de type humainement lisible
- On peut renvoyer une fonction anonyme depuis une fonction avec un type de retour automatique

```
1  template<typename Func> auto square(Func f)
2  {
3    return [f](auto x) { return f(f(x)); };
4  }
5
6  int main()
7  {
8    auto g = square([](auto x) { return x*x; });
9    std::cout << g(5) << "\n";
10 }
```

```
625
```

# Fonctions anonymes - Application

## Version finale

- Le code est plus localisé
- Le code ets plus optimisable

```
1  bool process_buffer2(std::vector<MemoryBuffer> const& mems)
2  {
3    return std::any_of( mems.begin(), mems.end()
4                      , [](MemoryBuffer const& m)
5                        {
6                          return m.size() > 2 && m[0] == 'E' && m[1] == 'Z';
7                        }
8                      );
9  }
```