

Développer en C++

Partie 1 – Eléments de base

Un peu d'histoire ...

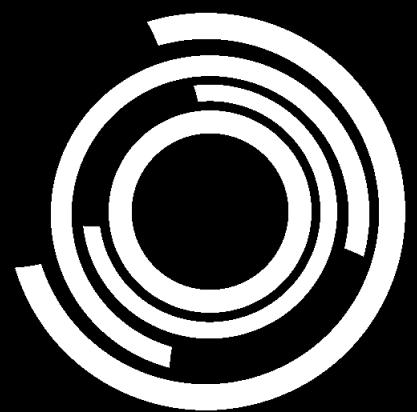
- 1978 Bjarne Stroustrup travaille sur Simula67 qui s'avère peu efficace
- 1980 Démarrage du projet *C with Classes*, du C orienté objet compilé via CFront.
- 1983 *C with Classes* devient C++
- 1985 Parution de *The C++ Programming Language*
- 1998 Première standardisation de C++ : ISO/IEC 14882 :1998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

Pourquoi C++14/17/... sont-ils importants ?

- *Make simple things simple* - Stroustrup 2014
- Les tâches simples doivent s'exprimer simplement
- Les tâches complexes ne doivent pas être trop complexes
- Rien ne doit être impossible
- Ne pas sacrifier les performances !

Objectifs

- Prise en main des éléments de bases de C++
 - C++ comme langage typé
 - Aspects impératifs : fonction & structures de données
 - Application immédiate
- Référentiel technique
 - C++ norme 2014
 - Adhérence stricte au standard



Programmeren en C++

Contenu

- Cycle de vie du logiciel
 - Code source
 - Compilation
 - Exécution
- Ce qu'il faudra retenir
 - Le programme C++ minimalisté
 - Le compilateur comme outil de tout les jours
 - **Survivre aux messages d'erreurs**

Cycle de vie du logiciel – Code source

- Pourquoi du « code source »
 - Fichier texte lisible par un humain
 - Exprime une série d'opération via un langage formalisé
 - Transformer en exécutable par un compilateur
- Notions de bases
 - Code minimal d'un code source C++
 - Eléments généraux du langage

Code source – Exemple minimaliste

```
int main(int argc, char** argv)
{
    return 0;
}
```

source.cpp

Code source – Exemple minimaliste

```
int main(int argc, char** argv)  
{  
    return 0;  
}
```

fonction principale

source.cpp

Code source – Exemple minimaliste

```
int main(int argc, char** argv)  
{  
    return 0;  
}
```

Corps de la fonction

source.cpp

Code source – Exemple minimaliste

```
int main(int argc, char** argv)  
{  
    return 0;  
}
```

Expressions
(*statements*)

source.cpp

Cycle de vie du logiciel – Compilation

- Principes généraux
 - Analyse et vérifie la correction du code source
 - Traduit le code en langage « CPU » (assembleur)
 - Agrège tous les éléments nécessaires à l'exécution
- Trois étapes de compilation
 - Pre-processing
 - Génération de code
 - Edition des liens

Cycle de vie du logiciel – Compilation

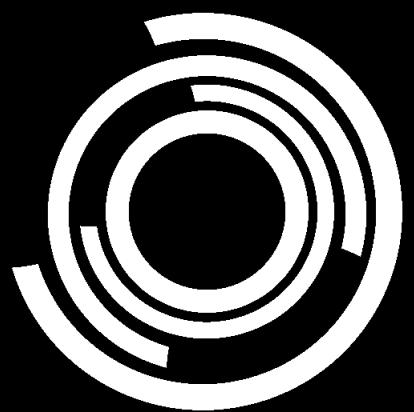
- Commande de compilation
 - `g++ <fichier source> -o <fichier executable> <option>`
 - Options: optimisation, version du standard
 - Possibilité de compiler plusieurs sources dans un exec.

- Exemple:

```
g++ source.cpp -o mon_prog -Wall -O3 -std=c++14
```

Cycle de vie du logiciel – Compilation

- Gestion des erreurs
 - Le code source est mal écrit
 - Le code source ne respecte pas les règles du langage
 - Le code « CPU » ne peut pas être généré
- Que faire en cas d'erreur de compilation ?
 - Le compilateur fournit une trace menant à l'erreur
 - Lire tranquillement le message et agir en conséquence
 - Bloqué ? Google, Stack Overflow



Types & Variables

Contenu

- Notion de type
 - Types de base du langage (natif et composite)
 - Qualificateurs
 - Inférence de type
- Ce qu'il faudra retenir
 - Quel type pour quelle tâche ?
 - Utilité des qualificateurs
 - Eléments de style

Types de base natifs

- Entiers
- Entiers de taille fixe
- Entiers à taille flexible
- Types à virgule flottantes
- Booléen

Types de base - Entiers

- Représentation d'une valeur entière signée
 - signed char, short, int
 - Supportent les opérations arithmétiques classiques
 - Taille relative : `sizeof(char) <= sizeof(short) <= sizeof(int)`
 - Bonne représentation pour les entiers relatifs (\mathbb{Z})
 - **On ne manipule pas les bits d'un entier signé**
- Représentation d'un agrégat de bits
 - unsigned char, unsigned short, unsigned int
 - Bien adaptés au manipulation de bits (masquage, décalage, etc...)
 - **On ne fait pas d'arithmétique sur un entier non-signé**

Types de base - Entiers

- Représentation d'une valeur entière signée

```
int          i = 4;  
short        u = 54;  
signed char c = 64;
```

```
i = 3 * i + 1; // i == 13  
u = -i;         // u == -13  
c = c + u;     // c == 51
```

Types de base - Entiers

- Représentation d'un agrégat de bit

```
unsigned char some = 0x3C;           // valeur hexadecimale 60
unsigned char other = 0b01010101; // valeur binaire 85

unsigned char r = some & other;    // r = 0b00010100 = 20
unsigned char s = some | other;    // s = 0b01111101 = 125

unsigned char d = r >> 2;          // d = 5
unsigned short l = (s << 8) | r; // l = 0b0111110100010100 = 32020
```

Types de base – Entiers à taille fixe

- Problématique
 - La taille des entiers natifs dépend du système
 - Comment s'assurer d'un nombre de bits minimal ?
 - Use case: sauvegarde sur disque, envoi sur le réseau ...
- Solution
 - C++ fourni un jeu de type de taille et de signe garanti
 - A utiliser pour s'assurer de propriété inter-système
 - **Ne doivent pas remplacer les types natifs pour l'arithmétique**

Types de base – Entiers à taille fixe

- Utilisation

- Disponible via #include <cstdint>
- std::{u}int{8,16,32,64}_t

```
#include <cstdint>

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

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

Types de base – Entiers à taille flexible

- Problématique
 - La valeur maximale de certaines grandeurs dépend du système
 - Ex : la taille d'un objet en mémoire, la distance entre deux objets, etc...
 - Comment s'assurer de stocker ces valeurs dans une variable de type adéquat ?
- Solution
 - C++ fourni un jeu de types de taille et de signe adaptatifs
 - **A utiliser de façon systématique**

Types de base – Entiers à taille flexible

- Utilisation:

```
#include <cstddef>

// std::size_t représente la taille d'un objet
long x;
std::size_t how_big = sizeof(x);

// std::ptrdiff_t la distance entre deux valeurs
// dans la mémoire
int value1, value2;
std::ptrdiff_t how_far = &value1 - &value2;
```

Types de base – Nombres à virgule flottante

- Problématique
 - Comment représenter un nombre réel ?
 - Précision ? Amplitude ?
 - Assurer un compromis entre précision et performance
- Solution
 - Utilisation du standard IEEE754
 - Représentation « à trou » de \mathbb{R}

Types de base – Nombres à virgule flottante

```
// Reel "simple précision" encodé sur 32 bits
float x = 0.2541368f;
```

```
// Reel "double précision" encodé sur 64 bits
double y = -3.14159265358979323846;
```

```
// Interaction avec les entiers
float bad_ratio = 3 / 2; // bad_ratio == 1.f
float good_ratio = 3.f / 2; // good_ratio == 1.5f
```

Types de base – Valeur booléenne

- Problématique
 - Représenter un réponse « oui/non »
 - Utiliser un type non-ambigüe
 - Interagir classiquement avec les autres expressions
- Solution
 - Introduction du type **bool**
 - **A utiliser de façon systématique**

Types de base – Valeur booléenne

```
// valeur booleene vrai & fausse  
bool v1 = true;  
bool v2 = false;
```

```
// expression booleene  
float x = 3.2f;  
bool k = x > 0.f;
```

```
// operations logiques  
bool r = (k && v2) || !v1;
```

Types de base composites

- Chaîne de caractères
 - `std::string`
- Tableau de taille fixe
 - `std::array<T, N>`
- Tableau de taille variable
 - `std::vector<T>`

Types de base composite – Chaîne de caractères

- `std::string`
- Représente une série de caractères (de type `char`)
- Accessible via `#include <string>`
- Support pour la copie, comparaison, concaténation, ...
- **A utiliser de façon systématique.**

Types de base composite – Chaîne de caractères

```
#include <string>

std::string empty; // chaîne vide
std::string basic = "une chaîne non-vide !";
std::string bar(15, '*'); // contient *****

// Accès à un caractère
char c = basic[5]; // c == 'h'
basic[0] = 'U';

// concaténation
std::string barbar = bar + bar;
```

Types de base composite – Chaîne de caractères

```
std::size_t s = basic.size();      // taille de la chaîne  
empty = basic;      // copie  
  
if (empty == basic)    // comparaison  
    empty.back() = '?'; // accès dernier élément  
  
// extraction de sous-chaines  
std::string sub = basic.substr(4, 5); // sub = "chaine"
```

Types de base composite – Tableau à taille fixe

- `std::array`
- Représente un tableau N valeurs
- Accessible via `#include <array>`
- Support pour la copie, comparaison, ...
- **A utiliser de façon systématique.**

Types de base composite – Tableau à taille fixe

```
#include <array>

// initialisation
std::array<double, 3> origin = { 0,0,0 };
std::array<double, 3> point1 = {1.5,-0.6,2.3 };

// manipulation
std::array<double, 3> point2 = point1;

for (int i = 0; i < point2.size(); ++i)
    point2[i] *= 10.;
```

Types de base composite – Tableau à taille variable

- `std::vector`
- Représente un tableau de taille arbitraire variable
- Accessible via `#include <vector>`
- Support pour la copie, comparaison, ...
- **A utiliser de façon systématique.**

Types de base composite – Tableau à taille variable

```
#include <vector>

std::vector<int> buffer(5); // tableaux de 5 int
std::vector<int> values = {1,2,3,4,5}; // tableaux de 5 int pré-rempli

buffer = values;      // copie
buffer.resize(10);   // agrandissement

// access aux éléments
for (int i = values.size(); i < buffer.size(); ++i)
    buffer[i] = i;

// Ajout en fin
buffer.push_back(99);
```

Qualificateur de types

- Objectifs
 - Renforcer la sémantique des types
 - Ajouter une propriété à un type
 - Clarifier la relation de la variable avec l'environnement
- Qualificateurs classiques
 - Immutabilité : $T \ const$
 - Références : $T\&$
 - Pointeurs : T^*

Qualificateur de types - Immutabilité

- Définition
 - Une variable **immutable** ne peut pas changer sa valeur
 - Mot-clé : **const**

```
int      x = 10; // variable
int const y = 25; // variable immutable

x = 9; // ok
x = y; // ok

y = 7; // error: assignment of read-only variable 'y'
```

Qualificateur de types - Référence

- Définition
 - Une référence est un *alias* d'une variable existante
 - Une référence est **obligatoirement** liée à une autre variable
 - Les accès à la référence affecte la variable référencée

```
std::string s = "Ex";
std::string& r1 = s;
std::string const& r2 = s;

r1 += "emple"; // modifie s
r2 += "!";    // error: cannot modify through reference to const
```

Qualificateur de types - pointeur

- Définition
 - Un pointeur fournit un accès indirect à une variable
 - Le **déréferencement** (*) du pointeur donne accès à la valeur de la variable pointée

```
std::string s = "Ex";
std::string* p1 = &s;
std::string const* p2 = &s;

*p1 += "emple"; // modifie s
*p2 += "!";    // error: cannot modify through pointer to const
```

Qualificateur de types - pointeur

- Autres propriétés
 - Un pointeur peut ne rien référencer: `nullptr`
 - Un pointeur peut changer sa « cible »
 - Un pointeur peut être immutable

```
std::string* p0 = nullptr;
p0 = p1;      // p0 "cible" la même variable que p1
p0 = nullptr; // p0 ne pointe plus rien

// QUIZZ : quel est le type de pxx ?
int const* const pxx;
```

Pointeur ou référence ?

- Utiliser une référence :
 - Pour obtenir un accès indirect à une variable
 - Cet accès nécessite d'être obligatoire et permanent
- Utiliser un pointeur :
 - Pour obtenir un accès indirect à une variable
 - Cet accès est optionnel (`nullptr`)
 - La variable accéder indirectement peu changer

Alias de types

- Problématiques :
 - Certains types sont très long
 - Certains types sont complexes à exprimer
 - Certains types pourraient avoir un nom plus clair
- Solution :
 - Notion de renommage (alias) de type
 - Mot-clé **using**

Alias de types

- Forme générale :

```
using nouveau_type = ancien_type;
```

```
#include <array>
```

```
using mass      = float;
using point3D = std::array<double, 3>;
```

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

Inférence de type automatique

- Déclarer une variable nécessite son type
- Dans le cas des initialisations, le type est déjà disponible
- Dans certains contextes, le type est complexe à exprimer

Inférence de type automatique

- Le mot-clé `auto`
 - `auto` remplace le type d'une variable lors d'une initialisation
 - Son type est déduit comme le type de la valeur de son initialiseur
 - `auto` est compatible avec des qualificateurs
 - Philosophie du ***Almost Always Auto***

```
float f = 3.f;  
  
std::size_t sz = sizeof(f);  
  
std::vector<std::uint8_t> bytes(sz);  
std::vector<std::uint8_t>::const_iterator b = bytes.begin();
```

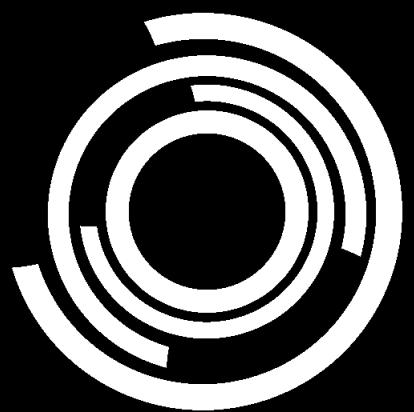


```
auto f = 3.f;  
  
auto sz = sizeof(f);  
  
std::vector<std::uint8_t> bytes(sz);  
auto b = bytes.begin();
```

Inférence de type automatique

- Le mot-clé `decltype`
 - Détermine le type d'une expression à la compilation
 - Contrairement à `auto`, il déduit le type exact
 - Cas particulier des références

```
float a, x = 0.7f;  
float& y = x;  
  
// u est de type float;  
decltype(3*cos(x)) u;  
  
// v est de type float&  
decltype(y) v = a;
```



Entrées/Sorties

Contenu

- Notion de flux
 - Types de flux
 - Manipulations de flux
 - Formatages/Structurations des entrées
- Ce qu'il faudra retenir
 - Quel flux pour quelle tâche
 - Savoir lire/écrire au travers d'un fichier simple

Flux console

- La famille cout/cin/cerr
 - Accessible via #include <iostream>
 - Fournit trois objets flux:
 - std::cin : entrée clavier
 - std::cout : sortie console
 - std::cerr : sortie console erreur

```
float x;
```

```
std::cout << "Donnez un réel positif: " << std::endl;
std::cin >> x;
std::cerr << "Erreur !\n";
```

Flux de chaîne

- std::stringstream
 - Accessible via #include <sstream>
 - Fournit deux objets flux:
 - std::ostringstream : insertion dans une chaîne
 - std::istringstream : extraction depuis une chaîne

```
std::ostringstream o;
o << "a= " << 17.3f << "\n";
std::cout << o.str();

int a, b, c;
std::istringstream i("9 8 75");
i >> a >> b >> c;
```

Flux de fichier

- std::fstream
 - Accessible via #include <fstream>
 - Fournit deux objets flux:
 - std::ofstream : fichier en écriture
 - std::ifstream : fichier en lecture
- Spécificités
 - L'accès se fait en mode texte ou binaire via 2 interfaces
 - Niveau d'abstraction très faible

A utiliser avec précaution

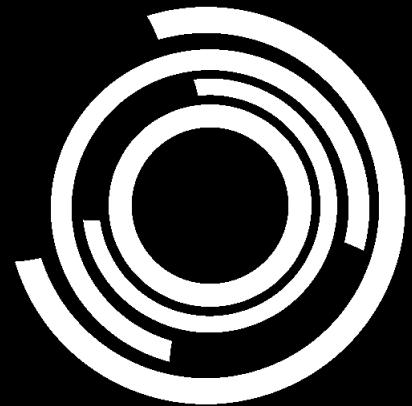
Flux de fichier

```
std::vector<double> data = { 1., 2., 3.5, -0.54 };

std::ofstream txtdata("data.txt");
std::ofstream bindata("data.bin", std::ostream::binary);

// Ecriture texte
txtdata << data[0] << " " << data[2] << "\n";

// Ecriture binaire
bindata.write( reinterpret_cast<const char*>(data.data())
               , data.size() * sizeof(double)
               );
```



Structures de contrôles

Contenu

- Structure de contrôle
 - Contrôle du flot du programme
 - Notion de bloc conditionnel
 - Notion de boucle déterministe & non-déterministes
- Ce qu'il faudra retenir
 - Savoir structurer son code
 - Savoir utiliser la bonne structure de contrôle

Structures de contrôles

- Structure conditionnelle
 - `if ... else ...`
- Structure de sélection
 - `switch`
- Structure répétitive déterministe
 - `for (...)`
- Structure répétitive non-déterministe
 - `while (...)`
 - `do { ... } while (...)`

Structures conditionnelles

- Objectif:
 - Permettre de choisir un fragment de code selon une condition arbitraire
 - « *si y a des œufs au supermarché, prends en une douzaine* »
- Forme générale:

```
if (condition)
{
    // code si condition == true
}
```

Structures conditionnelles

- Objectif:
 - Permettre de choisir un fragment de code selon une condition arbitraire
 - « *si y a des œufs au supermarché, prends en une douzaine , sinon prend du lait* »
- Forme générale:

```
if (condition)
{
    // code si condition == true
}
else
{
    // code si condition == false
}
```

Structures conditionnelles

- Forme en cascade:

```
if (condition1)
{
    // code si condition1 == true
}
else if (condition2)
{
    // code si condition2 == true
}
else
{
    // code si condition1 && condition2 == false
}
```

Structures conditionnelles

- Forme avec initialiseurs:
 - Possibilité d'initialiser une variable au sein du **if**
 - Cette variable n'est accessible que dans les branches conditionnelles
 - La condition implicite est: la variable crée a une valeur non-nulle

```
if (float* p = gimme_a_float() )  
{  
    *p = 4.f;  
}
```

Structures de sélection

- Objectif:
 - Modifie le flot du code en sélectionnant une branche parmi N
 - « mardi, prend la cravate rouge, mercredi la bleue, le reste de la semaine la grise »
- Forme générale:

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

    case cas2: // code du cas 2
        break;

    // ...

    default: // cas par défaut
}
```

Structures de sélection

- Mise en pratique :

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

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 << "samedi\n"; break;
    default: std::cout << "jour impossible\n";
}
```

Structures de sélection

- Mise en pratique : et les cravates alors ?

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

switch (jour)
{
    case 2: std::cout << "cravate rouge\n"; break;
    case 3: std::cout << "cravate bleue\n"; break;
    default: std::cout << "cravate grise\n";
}
```

Structures de sélection

- Et les cravates alors ? Sémantique de *fallthrough*

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

switch (jour)
{
    case 2: std::cout << "cravate rouge\n"; break;
    case 3: std::cout << "cravate bleue\n"; break;
    case 1:
    case 4:
    case 5:
    case 6:
    case 7: std::cout << "cravate grise\n"; break;
    default: std::cout << "cravate impossible\n";
}
```

Structures répétitives déterministes

- Objectif:
 - Répéter un bloc de code un nombre fois déterministe
 - Déterministe = nombre total connu à l'avance
 - « Tourne 7 fois ta langue dans ta bouche »
- Forme générale:

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

Structures répétitives déterministes

- Exemple : somme de 1 à N

```
int resultat = 0;  
int N = 10;  
  
for (int i = 1; i <= N; i++)  
{  
    resultat = resultat + i;  
}
```

Structures répétitives déterministes

- Cas particuliers : parcours d'un « équivalent tableau »

```
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;
}
```

Structures répétitives déterministes

- Cas particuliers : parcours d'un « équivalent tableau »

```
int resultat = 0;  
  
std::array<int, 10> vs = { 1,2,3,4,5,6,7,8,9,10 };  
  
for (auto const& v : vs)  
{  
    resultat = resultat + v;  
}
```

Structures répétitives déterministes

- Cas particuliers : parcours d'un « équivalent tableau »

```
int resultat = 0;
```

```
std::vector<int> vs = { 1,2,3,4,5,6,7,8,9,10 };
```

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

Structures répétitives non-déterministes

- Objectif:
 - Répéter un bloc de code de façon non-déterministe
 - Non-déterministe = l'arrêt dépend des données
 - « Va tout droit jusqu'à trouver le Burger King ! »
- Formes générales – 0 ou plusieurs itérations

```
while ( condition )
{
    // code à répéter
}
```

Structures répétitives non-déterministes

- Objectif:
 - Répéter un bloc de code de façon non-déterministe
 - Non-déterministe = l'arrêt dépend des données
 - « Mange ce bonbon, et les autres tant qu'il en reste »
- Formes générales – Au moins une itération

```
do
{
    // code à répéter
} while (condition);
```

Structures répétitives non-déterministes

- Exemple – Recherche de valeurs

```
// remplissage de x de manière aléatoire
std::vector<float> x = secret_filling(50);

int idx = 0;

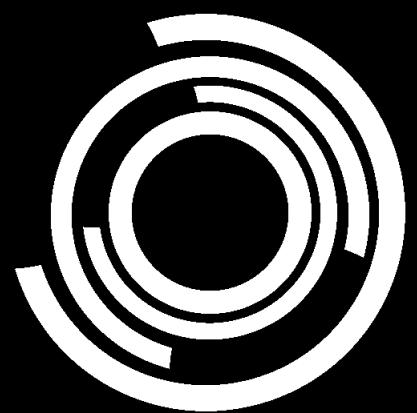
while (idx < x.size() && x[idx] != 1)
{
    idx++;
}

if ( idx < x.size() )
    std::cout << "valeur 1 trouvée !\n";
```

Structures répétitives non-déterministes

- Exemple – Somme de valeurs interactives

```
float n, sum = 0.f;  
  
do  
{  
    std::cin >> n;  
  
    sum += n;  
} while (n != 0.f);  
  
std::cout << sum << "\n";
```



Structures de données

Contenu

- Notion de structure
 - Types énumérés
 - Liste d'initialiseur, paire et tuple
 - Notion de structure définie par l'utilisateur
- Ce qu'il faudra retenir
 - Les différents niveaux de sémantiques mis en jeu
 - Quel structures de données (SDD) pour quelle tâche
 - Eléments de style

Types énumérés

- Principes
 - Abstraction d'un ensemble discret de valeurs numériques partageant un sens commun
 - Remplace avantageusement les valeurs « magiques »
 - Configurable en valeur et type sous-jacent
- Forme générale :

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

Types énumérés - Exemple

```
enum class jour { Lundi = 1, Mardi, Mercredi, Jeudi
                  , Vendredi, Samedi, Dimanche
                };

jour j = jour::Jeudi;
jour d = 4; // Erreur - valeur invalide

switch (j)
{
    case jour::Samedi   :
    case jour::Dimanche : std::cout << "WEEK END";
                          break;
    default             : std::cout << "Semaine :(";
                          break;
}
```

Types énumérés - Exemple

```
enum class jour : std::int8_t { Lundi = 5785, Mardi, Mercredi  
                                , Jeudi, Vendredi, Samedi, Dimanche  
}; // Erreur - valeur invalide  
  
jour j = jour::Jeudi;  
  
switch (j)  
{  
    case jour::Samedi :  
    case jour::Dimanche : std::cout << "WEEK END";  
                           break;  
    default            : std::cout << "Semaine :(";  
                           break;  
}
```

Listes d'initialiseurs

- Abstraction d'un ensemble de valeur du même type
- Utilisable comme « sac à valeur » sans sémantique
- Interface d'accès spécifique
- Utilisable dans des boucles **for**

Listes d'initialiseurs – Boucle for

```
std::uint8_t result = 0;

/*
    Mets les bits 0,1,2,4,5 et 7 de result à 1
    Notez que les valeurs prises par bits sont disjointes
*/
for (auto bits : { 1, 2, 4, 16, 32, 128 })
{
    result |= bits;
}
```

Paires et Tuples

- `std::pair`
 - Abstraction de deux valeurs de types quelconques
 - Ces valeurs ont une raison logique d'être couplées
 - Mais leur agrégat n'a pas de sémantique particulière
- `std::tuple`
 - Généralisation de la paire à $N \geq 1$ valeurs
 - Même sémantique que `std::pair`
 - Accès aux éléments généralisé

Paires et Tuples

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

int main()
{
    std::pair<int, float> p(13, .254f);
    auto s = std::make_pair('a', 3.5);

    std::cout << p.first << ", " << p.second << "\n";
    std::cout << std::get<0>(s) << ", " << std::get<1>(s) << "\n";
}
```

Paires et Tuples

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

int main()
{
    std::tuple<int, char> r = { 42, 'X' };
    auto t = std::make_tuple(1, 2., '3', 4.f, "5");

    std::cout << std::get<0>(t) << " " << std::get<1>(t) << " "
        << std::get<2>(t) << " " << std::get<3>(t) << "\n";

    auto rr = std::tuple_cat(r, r);

    int a, b;
    char c, d;

    std::tie(a, c, b, d) = rr;
}
```

Structure définie par l'utilisateur

- Notion de structure
 - Agrégat définissant une entité avec une sémantique claire
 - Les éléments d'une structure sont ses **données membres**
 - Permet l'ajout de nouveau type dans le langage
- Ce qui faudra retenir
 - Définir une structure
 - Comportements par défaut
 - Utiliser ses membres

Structure définie par l'utilisateur – Forme générale

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

    // etc .....

    type_memberN membre_N;
};

// Instanciation
nom_type variable;
```

Structure définie par l'utilisateur - Exemple

```
// point3D pondéré
struct weighted_point
{
    double x;
    double y;
    double z;

    int poids;
}

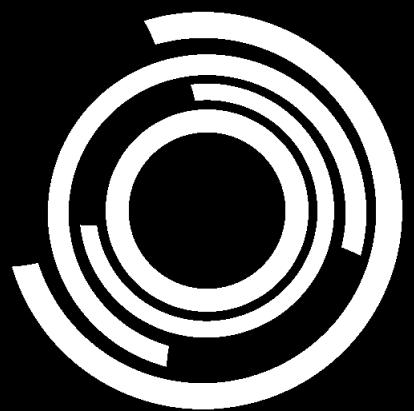
// Instanciation
weighted_point p;                                // sans initialisation
weighted_point q = {};;                          // initialization par zéro
weighted_point r = {1.5,0.5,-.333, 5}; // initialization explicite
```

Structure définie par l'utilisateur - Utilisation

```
weighted_point p;  
weighted_point q = {};  
weighted_point r = {1.5,0.5,-.333, 5};
```

```
// Affectation  
p = r;
```

```
// Access au membre individuel  
q.x = (p.z - r.x) / p.poids;
```



Aspects impératifs

Contenu

- Fonction
 - Eléments de factorisation du savoir-faire
 - Définition et déclaration de fonction
 - Surcharge de fonction
 - Fonction générique et inférence du type de retour
- Ce qu'il faudra retenir
 - Construire des fonctions pour factoriser le code
 - Concevoir une interface élégante via la surcharge

Fonctions

- Principes de bases
- Surcharge de fonctions
- Fonctions générique
- Type de Retour automatique

Fonctions – principes de base

- Principes de base
 - Groupe d'expressions encapsulé dans un format réutilisable
 - Accepte des **paramètres** et renvoi un **résultat**
- Définition d'une fonction :

```
type_retour nom_fonction(type1 param1, ..., typeN paramN)
{
    expressions
}
```

Fonctions – principes de base

- Exemple

```
int squared_difference(int a, int b)
{
    auto diff = a - b;
    return diff * diff;
}

// Appel
auto x = squared_difference(6, 10);
```

Fonctions – principes de base

- Exemple

```
void tell_me(float x)
{
    std::cout << "x is " << x << "\n";
}

// Appel
auto u = 1.2345f;
tell_me(u);
```

Fonctions – Passage des paramètres

- Passage par valeur

```
void f(float x);
```

- La valeur passée à la fonction est copié à l'intérieur

- Passage par référence vers une variable

```
void f(float& x);
```

- Permet de modifier le paramètre depuis la fonction

- Passage par référence vers une variable immutable

```
void f(float const& x);
```

- Permet d'éviter la copie d'objet de très grande taille

Fonctions – Passage des paramètres

```
void swap(int a, int b)
{
    auto t = a;
    a = b;
    b = t;
}
```

```
int x = 4, y = 90;
swap(x, y); // que valent x et y ???
```

Fonctions – Passage des paramètres

```
void swap(int& a, int& b)
{
    auto t = a;
    a = b;
    b = t;
}
```

```
int x = 4, y = 90;
swap(x, y); // que valent x et y ???
```

Fonctions variadiques via initializer_list

```
#include <initializer_list>

double sum(std::initializer_list<double> valeurs)
{
    double r = 0.;

    for (auto v : valeurs)
        r += v;

    return r;
}

// d = 49.54
auto d = sum({ 1., 3., 4.54, -9., 50. });
```

Fonctions – Interaction avec les Structures

```
// Retour par construction directe
weighted_point origin()
{
    return {};
}
```

```
// Retour par construction explicite
weighted_point origin()
{
    return weighted_point{};
}
```

Fonctions – Interaction avec les Structures

```
// Retour par construction directe + Passage par référence
weighted_point mirror(weighted_point const& p)
{
    return { -p.x, -p.y, -p.z, p.poids };
}
```

```
// Retour par construction explicite + Passage par référence
weighted_point mirror(weighted_point const& p)
{
    return weighted_point{ -p.x, -p.y, -p.z, p.poids };
}
```

Fonctions – Interaction avec les Structures

```
weighted_point shift_toward( weighted_point const& p
                            , float dx, float dy, float dz
                            )
{
    weighted_point that = p;

    that.x += dx;
    that.y += dy;
    that.z += dz;

    return that;
}
```

Fonctions – Interaction avec les Structures

```
// Passage par pointeur
void maybe_update(weighted_point* p, int factor)
{
    if (p != nullptr)
    {
        // la notation -> remplace . sur les
        // pointeurs de structures
        p->poids *= factor;
    }

}
```

Fonctions à retour multiple via Paires ou Tuples

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

std::pair<int, float> split(float x)
{
    auto ent = static_cast<int>(x);
    return { ent , x - ent };
}

int main()
{
    int i;
    float f;
    std::tie(i, f) = split(5.4321f);
    std::cout << i << ", " << f << "\n";
}
```

Fonctions – Paramètres par défaut

- Les paramètres d'une fonction peuvent avoir une valeur par défaut
- Uniquement en fin de liste des paramètres
- Permet « d'oublier » ces paramètre lors de l'appel

```
void display(char c = '*', int n = 1)
{
    for (int i = 1; i <= n; ++i)
        std::cout << c;
    std::cout << std::endl;
}
```

Fonctions – Paramètres par défaut

- Les paramètres d'une fonction peuvent avoir une valeur par défaut
- Uniquement en fin de liste des paramètres
- Permet « d'oublier » ces paramètre lors de l'appel

```
// Affiche:  *
display();
```

```
// Affiche:  #
display('#');
```

```
// Affiche: $$$$
display('$', 5);
```

Surcharge de Fonctions

- Une fonction = identifiant + liste de type de paramètres
- Des fonctions du même nom mais de paramétrages différents peuvent coexister dans un même programme
- On parle de **polymorphisme *ad hoc***
- Exemples
 - double f()
 - double f(int)
 - double f(int, char, char)
 - **int f()**



Surcharge de Fonctions

```
std::string as_text(std::string const& s) { return s; }
std::string as_text(char c) { return std::string(1, c); }
std::string as_text(float f)
{
    std::ostringstream o;
    o << f;
    return o.str();
}

auto s1 = as_text('Z');
auto s2 = as_text(3.141512f);
auto s3 = as_text("chaine chaine");
```

Surcharge d'opérateurs

- Une opérateur = fonction avec une syntaxe infixe

a = b + c \Rightarrow **operator=(a, operator+(b, c))**

- Les opérateurs sont **surchargables** comme des fonctions
- Ces surcharges doivent impliquer au moins un paramètre dont le type est une structure définie par l'utilisateur

Surcharge d'opérateurs – Listes des opérateurs surchargeables

Classe	Opérateurs	Forme générique
Opérateurs unaires logiques et arithmétiques	- ! ~	R operator<@>(T a)
Opérateurs binaires arithmétiques	+ - * / % << >> ^ &	R operator<@>(T a, U b)
Opérateurs de comparaison	== != < > <= >=	bool operator<@>(T a, U b)
Opérateurs logiques binaires	&&	R operator<@>(T a, U b) Attention au séquencement !

Attention : Si un opérateur surchargé peut avoir un comportement arbitraire, la bienséance nous conseil de conserver une sémantique proche de l'original

Surcharge d'opérateurs - Exemple

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

fraction operator-(fraction const& lhs)
{
    return { -lhs.num, lhs.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'opérateurs - 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 + rhs.num;
    auto denum = rhs.denum;

    return { num, denum };
}
```

Surcharge d'opérateurs – Interaction avec les flux

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

// Lecture depuis un flux
std::istream& operator>>(std::istream& is, fraction& f)
{
    is >> f.n >> f.d;
    return is;
}
```

Surcharge d'opérateurs

```
fraction r, a = { 1,3 }, b;
```

```
std::cout << "Donnez la fraction b:\n";  
std ::cin >> b;
```

```
r = (a + b) * 10;
```

```
std::cout << b << "\n";  
std::cout << r << "\n";
```

Fonctions génériques

- Motivation : Veut on vraiment écrire ce genre de code ?

```
double mini(double a, double b) { return a < b ? a : b; }
float  mini(float  a, float  b) { return a < b ? a : b; }
int    mini(int    a, int    b) { return a < b ? a : b; }
```

```
// etc ... etc ...
```

```
short  mini(short  a, short  b) { return a < b ? a : b; }
char   mini(char   a, char   b) { return a < b ? a : b; }
```

Fonctions génériques

- Motivation :
 - Et si on pouvait dire qu'une fonction fonctionne ***quelque soit le type de son/ses paramètres ?***
- Solution :
 - Notion de patron de fonction
 - Ecriture de code **générique**

Fonctions génériques

```
template<typename T>
T mini(T a, T b)
{
    return a < b ? a : b;
}
```

- Principe :
 - Remplacement des types par un ou plusieurs « paramètre de type »
 - L'appel de la fonction force le compilateur à déduire ces paramètres
 - Le code de la fonction est alors généré avec un vrai type

Fonctions génériques

```
template<typename T>
T mini(T a, T b)
{
    return a < b ? a : b;
}
```

- Paramètre **template** :
 - Introduit via la notation **template< >**
 - Contiennent au moins un paramètre de type
 - Chaque paramètre de type à un identifiant unique
 - Cet identifiant est introduit par le mot clé **typename**

Fonctions génériques

```
template<typename T>
T mini(T a, T b)
{
    return a < b ? a : b;
}
```

- Paramètre **template** :
- Chaque paramètre de type est utilisable normalement dans la définition de la fonction
- Les types non-utilisés devront être fournis à l'appel

Fonctions génériques

```
template<typename T>
T mini(T a, T b)
{
    return a < b ? a : b;
}
```

```
auto x = mini(4.f, 6.f);
auto y = mini(4, 6);
```

- Appel :
 - Se passe comme l'appel d'une fonction normale
 - Si une fonction générique n'est pas appelée, aucun code n'est généré
 - Le code généré est souvent « inliné »

Type de Retour automatique

- Principe :
 - Le type de retour de certaines fonctions génériques n'est pas trivialement exprimable car il dépend des types ou de relation entre les types
 - Notation pour indiquer que l'on délègue le calcul du type de retour au compilateur
- Mise en œuvre :
 - Mot clé **auto**
 - Mot clé **decltype**

Type de Retour automatique – Cas explicite

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



```
template< typename T1, typename T2 >
auto add(T1 const& a, T2 const& b) -> decltype(a+b)
{
    return a + b;
}
```

Type de Retour automatique – Cas implicite

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



```
template< typename T1, typename T2 >
auto add(T1 const& a, T2 const& b)
{
    return a + b;
}
```

Fonctions membres

- Principe :
 - Une fonction membre est une fonction définie à l'intérieur d'une structure
 - Elle a un accès direct à tout les membres de l'instance sur laquelle elle est appelée
 - Certain opérateur se surcharge comme une fonction membre

```
fraction r, x = { 178, 16 };

// Fonction membre arbitraire
x.simplify();

// Opérateur membre
r = x;
```

Fonctions membres – Mise en place

```
struct fraction
{
    int num, denum;

    // Déclaration
    void scale(int factor);
};

// Définition
void fraction::scale(int factor)
{
    num *= factor;
}
```

Cette notation `fraction::` indique que la fonction `scale` est un membre de `fraction`

Fonctions membres – Notation « inline »

```
struct fraction
{
    int num, denum;

    // Déclaration
    void scale(int factor)
    {
        num *= factor;
    }
};
```

Si le corps de la fonction membre est court, il peut suivre directement la déclaration à l'intérieur du corps de la structure.

A réservé aux fonctions triviales

Fonctions membres – Préservation de l’immutabilité

```
struct fraction
{
    int num, denum;

    fraction simplify() const
    {
        // gcd(x,y) calcule le PGCD de x et y
        // (elle sera disponible en C++17)
        auto d = gcd(num, denum);

        return fraction(num / d, denum / d);
    }
};
```

Si une fonction membre ne modifie pas l’état interne d’une instance, elle se doit de le signifier au compilateur via le mot clé **const**.

Lorsqu’une fonction non-const est appelé sur une instance immuable, le compilateur empêche la compilation

A faire de manière systématique

Fonctions membres – Constructeur

- Principe :
 - Un constructeur indique comment une structure peut s'initialiser
 - Constructeurs fournis automatiquement :
 - Constructeur par défaut
 - Constructeur de copie
 - Des constructeurs spécifiques supplémentaires sont implémentables

Fonctions membres – Constructeur

```
struct fraction
{
    // Constructeur par défaut
    fraction();

    // Constructeur de copie
    fraction(fraction const& src);

    // Constructeur custom #1
    fraction(int value) : num{ value }, denum{1}

    // Constructeur custom #2
    fraction(float value, int scale)
        : num{ value*scale }, denum{ scale }

    int num, denum;
};
```

Cette notation est une **liste d'initialisation**. Elle initialise les membres de la structure dans l'ordre avant d'exécuter le code du constructeur.

Fonctions membres – Constructeur

```
struct fraction
{
    // Constructeur par défaut
    fraction();

    // Constructeur de copie
    fraction(fraction const& src);

    // Constructeur custom #1
    fraction(int value) : num{ value }, denum{1}
    {}

    // Constructeur custom #2
    fraction(float value, int scale)
        : num{ value*scale }, denum{ scale }
    {}

    int num, denum;
};
```

// Appel du ctor par défaut
fraction f1a;
fraction f1b{};

// Appel du ctor de copie
fraction f2a(f1);
fraction f2b{f1};
fraction f2c = f1;

// Autres ctor
fraction f3(5);
fraction f4{0.25, 4};

Fonctions membres – Constructeur

```
struct fraction
{
    // Constructeur par défaut
    fraction();

    // Constructeur de copie
    fraction(fraction const& src);

    // Constructeur custom #1
    explicit fraction(int value)
        : num{ value }, denum{1}
    {}

    // Constructeur custom #2
    fraction(float value, int scale)
        : num{ value*scale }, denum{ scale }

    int num, denum;
};
```

explicit empêche l'appel automatique du constructeur.

A utiliser systématiquement sur les constructeurs à un seul paramètre.

Fonctions Membres- Listes des opérateurs surchargeables

Classe	Opérateurs	Forme générique
Opérateurs auto-référentiels	<code>+ = - = * = / = % = < < =</code> <code>> > = ^ = & = =</code>	<code>R operator<@>(U b)</code>
Opérateur de pré/post incrémentation et décrémentation	<code>++ --</code>	<code>T& operator++()</code> <code>T operator++(int)</code> <code>T& operator--()</code> <code>T operator--(int)</code>
Opérateur d'affectation	<code>=</code>	<code>T& operator=(T const& o)</code>
Opérateur d'indexation	<code>[]</code>	<code>R& operator[](U b)</code> <code>R& operator[](U b) const</code>
Opérateur appel de fonction	<code>()</code>	<code>R operator()(U... args)</code> <code>R operator()(U... args) const</code>

Fonctions membres – Opérateur auto-référentiel

```
struct fraction
{
    int num, denum;

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

        // Renvoie l'objet courant modifié
        return *this;
    }
};

fraction operator+(fraction const& lhs, fraction const& rhs)
{
    fraction that = lhs;
    return that += rhs;
}
```

Fonctions membres – Incrémentation

```
struct fraction
{
    int num, denum;

    fraction& operator++()
    {
        num += denum;

        // retour l'objet courant modifié
        return *this;
    }

    fraction operator++(int)
    {
        fraction tmp(*this); // copie
        operator++();        // incrémantation
        return tmp;          // retourne l'ancienne valeur
    }
};
```

```
fraction f1{4,5};

// Appel de op++()
auto f2 = ++f1;

// Autres de op++(int)
auto f3 = f1++;
```

Fonctions membres – Affectation

```
struct fraction
{
    int num, denum;

    // Habituellement inutile
    fraction& operator=(fraction const& other)
    {
        num    = other.num;
        denum = other.denum;

        // Renvoie l'objet courant modifié
        return *this;
    }
};
```

Fonctions membres – Affectation

```
struct fraction
{
    int num, denum;

    // Affectation depuis un autre type
    fraction& operator=(int value)
    {
        num    = value;
        denum = 1;

        // Renvoie l'objet courant modifié
        return *this;
    }
};
```

Fonctions membres – Indexation

```
struct triplet
{
    float& operator[](std::ptrdiff_t i)
    {
        return data[i];
    }

    float operator[](std::ptrdiff_t i) const
    {
        return data[i];
    }

    float data[3];
};

triplet a = { 0.1, 0.01, 0.001 };

a[0]    = 7.;
auto x = a[1];
```

Fonctions membres – Appel de fonction

```
struct scaler
{
    float operator()(float value) const
    {
        return value * scale;
    }

    float scale;
};

scaler s{ 10.f };

// u = 5.1f
auto u = s(0.51f);
```

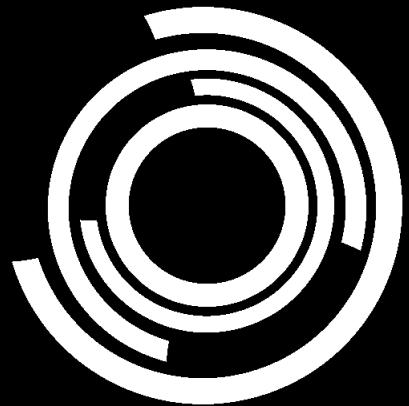
Fonctions membres – Appel de fonction

```
struct scaler
{
    template<typename T>
    T operator()(T value) const
    {
        return value * scale;
    }

    float scale;
};

scaler s{ 10.f };

// u = 30/2
auto u = s( fraction{ 3,2 } );
```



La Bibliothèque standard Aspect Algorithmiques

Abstraction de boucle

- Objectifs
 - Augmentez l'expressivité du code
 - S'assurer des performances/correction d'un algorithme
 - Augmentez la localité du code
- Éléments de réponses
 - Rappel - Boucle for automatique sur conteneur
 - Algorithmes standards

Rappel - Extension de la boucle for

```
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;  
}
```

Rappel - Extension de la boucle for

```
int resultat = 0;  
  
std::array<int, 10> vs = { 1,2,3,4,5,6,7,8,9,10 };  
  
for (auto const& v : vs)  
{  
    resultat = resultat + v;  
}
```

Rappel - Extension de la boucle for

```
int resultat = 0;  
  
std::vector<int> vs = { 1,2,3,4,5,6,7,8,9,10 };  
  
for (auto const& v : vs)  
{  
    resultat = resultat + v;  
}
```

Notions d'itérateurs

- Principes
 - Extension du pointeur
 - Contient un « curseur » vers un élément d'un conteneur
 - Tous les pointeurs sont des itérateurs
- Exemples
 - `.begin()` renvoie un itérateur vers le début d'un conteneur
 - `.end()` renvoie un itérateur vers l'élément après le dernier élément d'un conteneur

Abstraction de boucle : Un exemple in situ

```
bool match_pattern(MemoryBuffer const& m)
{
    return m.size() > 2 && m[0] == 'E' && m[1] == 'Z';
}

bool process_buffer(std::vector<MemoryBuffer> const& mems)
{
    std::vector<MemoryBuffer>::const_iterator b = mems.begin();
    std::vector<MemoryBuffer>::const_iterator e = mems.end();

    for(; b != e; b++)
    {
        if (match_pattern(*b)) return true;
    }

    return false;
}
```

Abstraction de boucle : Boucle d'interval

```
bool match_pattern(MemoryBuffer const& m)
{
    return m.size() > 2 && m[0] == 'E' && m[1] == 'Z';
}

bool process_buffer(std::vector<MemoryBuffer> const& mems)
{
    for(auto const& m : mems)
    {
        if (match_pattern(m)) return true;
    }

    return false;
}
```

Abstraction de boucle : Algorithmes Standards

- Objectifs
 - Fournir une implantation de référence des traitements classiques sur des conteneurs de données
 - S'abstraire du type effectif de conteneur
 - Donner du **vocabulaire** aux développeurs
- Mise en pratique
 - Repose sur la notion **d'Itérateur**
 - Implantation optimale déduite à la compilation
 - Large gamme d'algorithme et de variante

Les Algorithmes Standards

- `all_of`, `any_of`, `none_of`
- `for_each`
- `count`, `count_if`
- `mismatch`, `equal`
- `find`, `find_if`
- `find_end`, `find_first_of`
- `search`, `search_n`
- `nth_element`
- `max_element`, `min_element`
- `transform`
- `copy`, `copy_if`
- `remove`, `remove_if`
- `replace`, `replace_if`
- `reverse`, `rotate`, `shuffle`
- `sort`, `stable_sort`
- `fill`, `iota`
- `accumulate`
- `inner_product`

Les Algorithmes Standards

```
bool match_pattern(MemoryBuffer const& m)
{
    return m.size() > 2 && m[0] == 'E' && m[1] == 'Z';
}

bool process_buffer(std::vector<MemoryBuffer> const& mems)
{
    return std::find_if( mems.begin(), mems.end()
                        , match_pattern
                        ) != mems.end();
}
```

Les Fonctions Anonymes

- Objectifs
 - Augmenter la localité du code
 - Simplifier le design de fonction utilitaire
 - Notion de *closure* et de fonction d'ordre supérieur
- Principes
 - Définition en ligne de fonction
 - Bloc de code fonctionnel sans identité
 - Équivalent fonction des variables temporaires

Environnement

- Capture de l'environnement
 - [a] - capture a par copie
 - [&a] - capture a par référence
 - [=] - tout par copie
 - [&] - tout par référence

```
int x, n;  
  
auto f = [x](int a, int b)  
{  
    return a*x + b;  
};  
  
auto g = [&x]() { x++; };  
  
auto h = [&x, n]() { x *= n; };  
  
auto s = [&]()  
{  
    x = n;  
    n = 0;  
    return x;  
};
```

Paramètres

- C++11
 - Type(s) concret(s)
- C++14
 - Type(s) générique(s)
 - Liste de paramètres variadique

```
auto s11 = [](int a, int b)
{
    return a + b;
}
```

```
auto s14 = [](auto a, auto b)
{
    return a + b;
}
```

```
auto as_tuple = [](auto... args)
{
    return std::make_tuple(args...);
}
```

Type de Retour

- C++11
 - Automatique si la fonction n'est qu'un return
 - Autre cas, à spécifier via ->
- C++14
 - Déduction automatique

```
auto f11 = []() { return 4; };

auto g11 = [](int n) -> float
{
    float g = 1.f;
    for (int i = 1; i <= n; ++i) g *= i;

    return g;
};

auto g14 = [](int n)
{
    float g = 1.f;
    for (int i = 1; i <= n; ++i) g *= i;

    return g;
};
```

Application conjointe aux algorithmes

```
bool process_buffer(std::vector<MemoryBuffer> const& mems)
{
    return find_if( mems.begin(), mems.end()
                    , [](auto const& m)
                    {
                        return     m.size() > 2
                                && m[0] == 'E' && m[1] == 'Z';
                    }
                ) != mems.end();
}
```

Abstraction de boucle : Conclusion

- Impact
 - Les boucles for n'ont pas de sémantique
 - Les algorithmes standards enrichissent le vocabulaire du dev.
 - Customisable via les fonctions anonymes
- Préconisation
 - Préférez les algorithmes aux boucles nus
 - Augmentez la localité du code via les fonctions anonymes