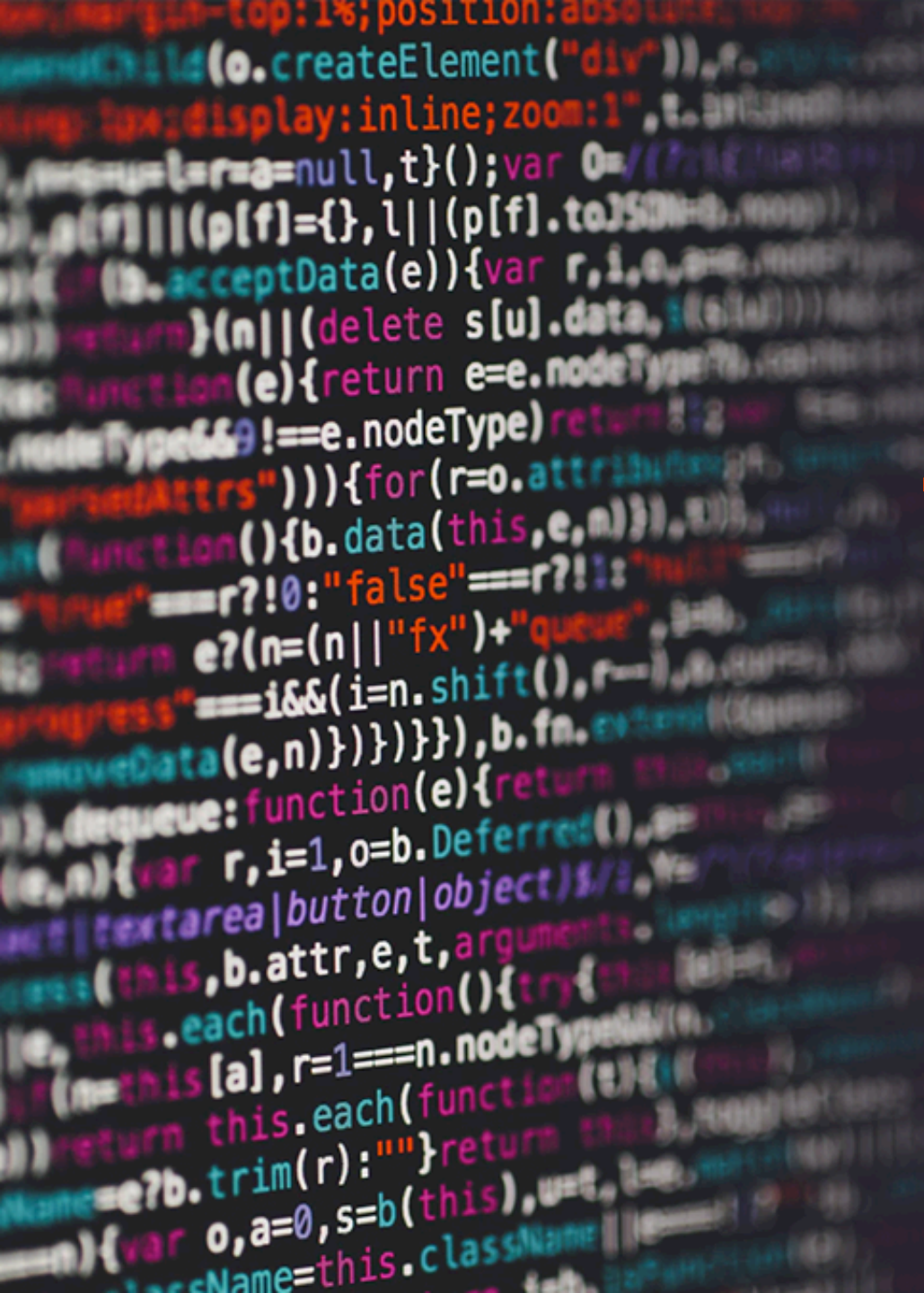

Introduction à la programmation fonctionnelle en C++

De la STL à Range-v3

LoOPS, septembre 2019
David Chamont, LAL

Une vision de la PF

- Se rapprocher de la logique mathématique :
  - une variable désigne toujours la même chose
  - avec les mêmes arguments, une fonction retourne toujours le même résultat ;
  - des méta-fonctions peuvent manipuler des fonctions ;
  - une algèbre des types permet de les recombiner.
- Compenser les inefficacités induites
  - évaluation paresseuse
  - structures de données immuables
  - dans les compilateurs C++ : l'optimisation de la valeur de retour (RVO), ...

Bénéfices de la PF

- En théorie
  - Prouvabilité
  - Code plus lisible
- En pratique :
  - réduire les changements d'état involontaires
    Michael Feathers : OO makes code understandable by encapsulating moving parts. FP makes code understandable by minimizing moving parts.
  - faciliter la parallélisation
    no shared states, no problem.

Un exemple

```
void count_lines_in_files( const vector<string> & files, vector<int> & nb_lines )  {
    vector<string>::iterator fileitr ;
    for ( fileitr = files.begin() ; fileitr != files.end() ; ++fileitr ) {
        int line_count = 0 ; char c = 0 ; ifstream in(*fileitr) ;
        while (in.get(c)) {
            if (c == '\n') { line_count++ ; }
        }
        nb_lines.push_back(line_count) ;
    }
}
```

En C++ ancien

- On peut depuis le début adopter un style de programmation fonctionnel
  - Propager *const* partout ou c'est possible.
  - Revenir à la pureté des fonctions centrales (une entrée, une sortie, faire la chasse aux variables globales).
  - Utiliser les algorithmes de la STL.

En C++ ancien

```
int count_lines( const string & filename ) {
    ifstream in(filename) ;
    return count( istreambuf_iterator<char>(in), istreambuf_iterator<char>(), '\n') ;
}

vector<int> count_lines_in_files( const vector<string> & files ) {
    vector<int> results(files.size()) ;
    transform(files.cbegin(), files.cend(), results.begin(), count_lines) ;
    return results ;
}
```

C++ moderne

- Un style fonctionnel plus lisible et plus performant
  - l'inférence de type avec *auto*, le *range-based for*, les alias de type paramétrés avec *using*, la définition de fonctions lambdas *[]*
  - les double références *&&*
- Algèbre de types
  - A*B => *std::tuple<A,B>*
  - A+B => *std::variant<A,B>*
  - *std::optional, std::future...*
- Variantes parallèles des algorithmes

C++ moderne

```
vector<int> count_lines_in_files( vector<string> const & files )  {
    vector<int> results(files.size()) ;
    transform(files.cbegin(), files.cend(), results.begin(),
        []( auto const & filename ){
            ifstream in(filename) ;
            using ifiterator = istreambuf_iterator<char> ;
            return count( ifiterator(in), ifiterator(), '\n') ;
    }) ;
    return results ;
}
```

C++ moderne

```
vector<int> count_lines_in_files( vector<string> const & files )  {
    vector<int> results(files.size()) ;
    std::transform( execution::par,
        files.cbegin(), files.cend(), results.begin(),
        []( auto const & filename ){
            ifstream in(filename) ;
            using ifiterator = istreambuf_iterator<char> ;
            return count( ifiterator(in), ifiterator(), '\n') ;
    }) ;
    return results ;
}
```

C++ futur

- Range-v3, bientôt intégré dans C++20
  - Un *range* se comporte comme une paire d'itérateurs de début et de fin.
  - Les algorithmes de Range-v3 recoivent des *ranges* en entrée, et retourne des *ranges* en sortie.
  - On peut chainer ces algorithmes avec l'opérateur |.
- Concepts, coroutines...

C++ futur

```
int count_lines( const string & filename )  {
    ifstream in(filename) ;
    using ifiterator = istreambuf_iterator<char> ;
    return count( ifiterator(in), ifiterator(), '\n') ;
}

vector<int> count_lines_in_files( const vector<string> & files )  {
    return files | transform(count_lines) ;
}
```

C++ futur

```
ifstream open_file( const string & filename ) {
    return ifstream(filename) ;
}

int count_lines( ifstream file ) {
    using ifiterator = istreambuf_iterator<char> ;
    return count( ifiterator(in), ifiterator(), '\n') ;
}

vector<int> count_lines_in_files( const vector<string> & files ) {
    return files | transform(open_file) | transform(count_lines) ;
}
```

Pour aller plus loin

- Echauffement :
  Programmation fonctionnelle en C++
  John Carmack & Arzar

- **En plat de résistance :**
  Functional Programming in C++
  Ivan Čukić

- Pour les nuits sans sommeil, avec des exemples en Haskell et en C++ :
  Category Theory for Programmers
  Bartosz Milewski