C++ fonctionnel

for_each avec une fonction

```
#include <iostream>
#include <algorithm>
#include <vector>

void print(int i) {
  std::cout << "(" << i << ") ";
}
int main() {
  std::vector<int> vect({1, 3, 4, 5});
  std::for_each(vect.begin(), vect.end(), print);
  std::cout << std::endl;
  return 0;
}
```

for_each avec lambda (moins de sucre)

```
std::for_each(std::begin(vect), std::end(vect),
              [](int i){
                  std::cout << "(" << i << ") ";
              });
```

Capture de variables par la lambda

```
std::function<void()> lambda; // définition
{ // Un contexte A
  int i = 42;
  double d = 3.14;
  lambda = [i, d] () { // contexte séparé de A
    int j = 7;
    cout << i << ' ' << d << ' ' << j << endl;
  };
}
lambda(); // Usage
```

Implémentation possible pour la lambda

```
class The_lambda {
  int i;
  double d;
public:
  The_lambda(int i_, double d_) : i(i_), d(d_) {}
  void operator()() const {
    int j = 7;
    cout << i << ' ' << d << ' ' << j << endl;
  }
};
```

lambda généralisée (C++14)

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

Un tableau de fonctions

```
map<const char, function<double(double, double)>>
    tab;
tab.insert(make_pair('+',
   [](double a, double b){ return a + b;} ));
tab.insert(make_pair('*',
   [](double a, double b){ return a * b;} ));

cout<< "3.5+4.5= " << tab['+'](3.5, 4.5) << endl;
cout<< "3.5*4.5= " << tab['*'](3.5, 4.5) << endl;
```

Premier exemple de currification

```
int add(int a, int b){ return a+b; }
auto addA= [](int b){ return add(42, b); };
function<int(int)> addB=std::bind(add, 42, _1);
```

Currification du générateur aléatoire

```
std::default_random_engine e;
std::uniform_int_distribution<> d(0, 10);

auto val1 = vector<int>{d(e), d(e), d(e)};

auto rnd = std::bind(d, e); // copie e dans rnd
auto val2 = vector<int>{rnd(), rnd(), rnd()};
```

std::optional (C++17)

```
std::optional<int> oEmpty;
std::optional<float> oFloat = std::nullopt;

std::optional<int> oInt(10);
std::optional oIntDeduced(10);

auto oComplex
  = make_optional<std::complex<double>>(3.0, 4.0);
-------------------------------------------------
std::optional<std::string> TryParse(Input input) {
    if (input.valid())
        return input.asString();

    return std::nullopt;
}
-------------------------------------------------
if (opt.has_value()) // ou bien: if (opt)
    std::cout << "valeur: " << opt.value() <<'\n';
-------------------------------------------------
try {
    std::cout << "valeur: " << opt.value() <<'\n';
} catch (const std::bad_optional_access& e) {
    std::cout << e.what() << "\n";
}
```

std::variant (C++17)

```
std::variant<int, float, std::string>
        intFloatString { "Hello" };
intFloatString = 10;

std::get<std::string>(intFloatString)
    += std::string(" World");
-------------------------------------------------
try {
    auto f = std::get<float>(intFloatString);
    std::cout << "float! " << f << "\n";
} catch (std::bad_variant_access&) { ... }
-------------------------------------------------
int *pval = std::get_if<int>(&intFloatString);
if (pval != nullptr)
  std::cout << "int! " << *pval << "\n";
-------------------------------------------------
std::visit(overload  {
 [](const int& i) { std::cout<< "int: " << i; },
 [](const float& f){ std::cout<< "float: " << f; },
 [](const std::string&s){std::cout<< "str: "<<s;}
}, intFloatString);
```

STL sans range

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

int main() {

  std::vector<int> numbers = {1, 2, 3, 4, 5, 6};
  std::vector<int> filtered;

  // Deux façons équivalentes pour filtrer
  std::for_each(numbers.begin(), numbers.end(),
                [&filtered] (int a) {
                  if (a%2==0)
                    filtered.push_back(a);
                });
  std::copy_if(numbers.begin(), numbers.end(),
               std::back_inserter(filtered),
               [] (int a) { return a%2 ==0; });

  // Et maintenant, l'affichage
  std::for_each(filtered.begin(), filtered.end(),
                [](int a) { std::cout << a<<", ";
                });
}
```

Les ranges C++20

```cpp
#include <vector>
//#include <ranges> // C++20 pas dans gcc/clang
#include <range/v3/all.hpp> // version utilisable
#include <iostream>

int main() {
  std::vector<int> numbers = {1, 2, 3, 4, 5, 6};

  auto results = numbers
    | ranges::views::filter([](int n){
        return n % 2 == 0; })
    | ranges::views::transform([](int n){
        return n * 2; });

  for (auto v: results) std::cout << v << " ";
}
```

Les ranges C++20

```cpp
#include <iostream>
#include <range/v3/all.hpp> // <ranges>

namespace views = ranges::views; // std::views

bool isPrime(int i) {
    for (int j=2; j*j <= i; ++j)
        if (i % j == 0) return false;
    return true;
}
int main() {
    auto odd = [](int i){ return i % 2 == 1; };

    for (int i: views::iota(0)
           | views::drop(1000000)
           | views::filter(odd)
           | views::filter(isPrime)
           | views::take(20)) {
        std::cout << i << " ";
    }
}
```

Les zip C++23

```cpp
#include <range/v3/all.hpp> // C++23 pas dans clang
int main() {
  std::vector<int>   u = { 1,  2,  3,  4,  5,  6};
  std::vector<float> d = {10, 20, 30, 40, 50, 60};

  auto results = ranges::views::zip(u,d)
   | ranges::views::filter([](auto pair){
       return pair.first % 2 == 0; })
   | ranges::views::transform([](auto pair){
       return pair.second * 2; });
  for (auto v: results) std::cout << v << " ";
```

Memoization générique en C++11

```cpp
template <typename ReturnType, typename ArgType>
std::function<ReturnType(ArgType)>
  memoize(std::function<ReturnType(ArgType)> fn){
  auto* memo = new std::map<ArgType, ReturnType>();
  return [fn, memo](ArgType key) -> ReturnType {
    if (memo->find(key) == memo->end())
      (*memo)[key] = fn(key);
    return (*memo)[key];
  };
}
int main() {
  std::function<unsigned long(unsigned)> fib =
    memoize<unsigned long, unsigned>(
      [&fib](unsigned n) {
        return (n<2) ? n : fib(n-2) + fib(n-1);
      }
   );
  std::cout << fib(60) << "\n";
```

C++ générique

Point avec des doubles ou des floats

```cpp
template <typename T>
class Point {
  T x_, y_;
 public:
  void move(T dx, T dy);
};
template <class T>
void Point::move(T dx, T dy) {
  x_ += dx;
  y_ += dy;
}
```

Type de retour postfixé

```cpp
template <typename T1, typename T2>
auto add(T1 t1, T2 t2) -> decltype(t1 + t2) {
   return t1 + t2;
} // /!\ add("one", 2) donne 'e' /!\
```

Contraintes de typage

```cpp
template <typename T1, typename T2>
auto addSafe(T1 t1, T2 t2) -> decltype(t1 + t2) {
   static_assert(std::is_arithmetic<T1>::value,
                 "Type T1 doit être un nombre");
   static_assert(std::is_arithmetic<T2>::value,
                 "Type T2 doit être un nombre");

   return t1 + t2;
}
```