

C++ fonctionnel

for_each avec une fonction

```
1 #include <iostream>
2 #include <algorithm>
3 #include <vector>
4
5 void print(int i) {
6     std::cout << "(" << i << ") ";
7 }
8
9 int main() {
10     std::vector<int> vect({1, 3, 4, 5});
11     std::for_each(vect.begin(), vect.end(), print);
12     std::cout << std::endl;
13     return 0;
14 }
```

for_each avec lambda (moins de sucre)

```
10 std::for_each(std::begin(vect), std::end(vect),
11             [](int i){
12                 std::cout << "(" << i << ") ";
13             });
```

Capture de variables par la lambda

```
1 std::function<void()> lambda; // définition
2 { // Un contexte A
3     int i = 42;
4     double d = 3.14;
5     lambda = [i, d] () { // contexte séparé de A
6         int j = 7;
7         cout << i << ' ' << d << ' ' << j << endl;
8     };
9 }
10 lambda(); // Usage
```

Implémentation possible pour la lambda

```
1 class The_lambda {
2     int i;
3     double d;
4 public:
5     The_lambda(int i_, double d_) : i(i_), d(d_) {}
6     void operator() const {
7         int j = 7;
8         cout << i << ' ' << d << ' ' << j << endl;
9     }
10 };
```

lambda généralisée (C++14)

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

Un tableau de fonctions

```
1 map<const char, function<double(double, double)>>
2     tab;
3 tab.insert(make_pair('+',
4             [](double a, double b){ return a + b; } ));
5 tab.insert(make_pair('*',
6             [](double a, double b){ return a * b; } ));
7
8 cout << "3.5+4.5= " << tab['+'](3.5, 4.5) << endl;
9 cout << "3.5*4.5= " << tab['*'](3.5, 4.5) << endl;
```

Premier exemple de currification

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

Curification du générateur aléatoire

```
1 std::default_random_engine e;
2 std::uniform_int_distribution<> d(0, 10);
3
4 auto val1 = vector<int>{d(e), d(e), d(e)};
5
6 auto rnd = std::bind(d, e); // copie e dans rnd
7 auto val2 = vector<int>{rnd(), rnd(), rnd()};
```

std::optional (C++17)

```
1 std::optional<int> oEmpty;
2 std::optional<float> oFloat = std::nullopt;
3
4 std::optional<int> oInt(10);
5 std::optional oIntDeduced(10);
6
7 auto oComplex
8     = make_optional<std::complex<double>>(3.0, 4.0);
9
10 -----
11 std::optional<std::string> TryParse(Input input) {
12     if (input.valid())
13         return input.asString();
14
15     return std::nullopt;
16 }
17
18 -----
19 if (opt.has_value()) // ou bien: if (opt)
20     std::cout << "valeur: " << opt.value() << '\n';
21
22 -----
23 try {
24     std::cout << "valeur: " << opt.value() << '\n';
25 } catch (const std::bad_optional_access& e) {
26     std::cout << e.what() << "\n";
27 }
```

std::variant (C++17)

```
1 std::variant<int, float, std::string>
2     intFloatString { "Hello" };
3 intFloatString = 10;
4
5 std::get<std::string>(intFloatString)
6     += std::string(" World");
7
8 try {
9     auto f = std::get<float>(intFloatString);
10    std::cout << "float! " << f << "\n";
11 } catch (std::bad_variant_access&) { ... }
12
13 -----
14 int *pval = std::get_if<int>(&intFloatString);
15 if (pval != nullptr)
16     std::cout << "int! " << *pval << "\n";
17
18 -----
19 std::visit(overload {
20     [](const int& i) { std::cout << "int: " << i; },
21     [](const float& f){ std::cout << "float: " << f; },
22     [](const std::string& s){ std::cout << "str: " << s; }
23 }, intFloatString);
```

STL sans range

```
1 #include <vector>
2 #include <algorithm>
3 #include <iostream>
4
5 int main() {
6
7     std::vector<int> numbers = {1, 2, 3, 4, 5, 6};
8     std::vector<int> filtered;
9
10    // Deux façons équivalentes pour filtrer
11    std::for_each(numbers.begin(), numbers.end(),
12                  [&filtered] (int a) {
13        if (a%2==0)
14            filtered.push_back(a);
15    });
16    std::copy_if(numbers.begin(), numbers.end(),
17                std::back_inserter(filtered),
18                [](int a) { return a%2 == 0; });
19
20    // Et maintenant, l'affichage
21    std::for_each(filtered.begin(), filtered.end(),
22                  [](int a) { std::cout << a<<" ";
23    });
24 }
```

Les ranges C++20

```
1 #include <vector>
2 // #include <ranges> // C++20 pas dans gcc/clang
3 #include <range/v3/all.hpp> // version utilisable
4 #include <iostream>
5
6 int main() {
7     std::vector<int> numbers = {1, 2, 3, 4, 5, 6};
8
9     auto results = numbers
10     | ranges::views::filter([](int n){
11         return n % 2 == 0; })
12     | ranges::views::transform([](int n){
13         return n * 2; });
14
15     for (auto v: results) std::cout << v << " ";
16 }
```

Les ranges C++20

```
1 #include <iostream>
2 #include <range/v3/all.hpp> // <ranges>
3
4 namespace views = ranges::views; // std::views
5
6 bool isPrime(int i) {
7     for (int j=2; j*j <= i; ++j)
8         if (i % j == 0) return false;
9     return true;
10 }
11 int main() {
12     auto odd = [](int i){ return i % 2 == 1; };
13
14     for (int i: views::iota(0)
15         | views::drop(1000000)
16         | views::filter(odd)
17         | views::filter(isPrime)
18         | views::take(20)) {
19         std::cout << i << " ";
20     }
21 }
```

Les zip C++23

```
1 #include <range/v3/all.hpp> // C++23 pas dans clang
2 int main() {
3
4     std::vector<int> u = { 1, 2, 3, 4, 5, 6};
5     std::vector<float> d = {10, 20, 30, 40, 50, 60};
6
7     auto results = ranges::views::zip(u,d)
8     | ranges::views::filter([](auto pair){
9         return pair.first % 2 == 0; })
10    | ranges::views::transform([](auto pair){
11        return pair.second * 2; });
12    for (auto v: results) std::cout << v << " ";
13 }
```

Memoization générique en C++11

```
1 template <typename ReturnT, typename ArgT>
2 std::function<ReturnT(ArgT)>
3 memoize(std::function<ReturnT(ArgT)>& fn){
4     auto* memo = new std::map<ArgT, ReturnT>();
5     return [fn, memo](ArgT key) -> ReturnT {
6         if (memo->find(key) == memo->end())
7             (*memo)[key] = fn(key);
8         return (*memo)[key];
9     };
10 }
11 int main() {
12     std::function<unsigned long(unsigned)> fib =
13     memoize<unsigned long, unsigned>(
14     [&fib](unsigned n) {
15         return (n<2) ? n : fib(n-2) + fib(n-1);
16     }
17 );
18     std::cout << fib(60) << "\n";
19 }
```

C++ générique

Point avec des doubles ou des floats

```
1 template <typename T>
2 class Point {
3     T x_, y_;
4 public:
5     void move(T dx, T dy);
6 };
7 template <class T>
8 void Point::move(T dx, T dy) {
9     x_ += dx;
10    y_ += dy;
11 }
```

Type de retour postfixé

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

Contraintes de typage

```
1 template <typename T1, typename T2>
2 auto addSafe(T1 t1, T2 t2) -> decltype(t1 + t2) {
3     static_assert(std::is_arithmetic<T1>::value,
4                   "Type T1 doit être un nombre");
5     static_assert(std::is_arithmetic<T2>::value,
6                   "Type T2 doit être un nombre");
7     return t1 + t2;
8 }
9 }
```