Writing efficient code

Sébastien Ponce
sebastien.ponce@cern.ch

CERN

Thematic CERN School of Computing 2025

Outline

1. Virtues of functional programming
   - Theory using haskell
2. Practical usage and efficiency
   - Functional programming in C++
   - Efficiency
3. It's all about helping the compilers
   - constness, exceptions
   - Avoiding useless copying
4. Do more at compile time
   - Templates
   - `constexpr` and `if constexpr`
   - Virtuality : a counter example
5. Conclusion

Virtues of functional programming

1 Virtues of functional programming
- Theory using haskell

2 Practical usage and efficiency

3 It's all about helping the compilers

4 Do more at compile time

5 Conclusion

Functional programming

Definition (Wikipedia) and rules

a programming paradigm where programs are constructed by applying and composing functions. Functions have

- no side effects
- no modification of the input
- new return values

Consequences

- no state, no globals, no members
- guaranteed thread-safety !

Usage

- dedicated languages : Haskell, Erlang, Lisp, ...
- modern C^{++}(lambdas, move, copy elision, ...)

Crash course in functional programing

haskell syntax already tells a lot - godbolt

```haskell
-- declaration
add :: Int -> Int -> Int
-- implementation
add x y = x + y
-- usage
five = add 3 2
add_42 = add 42
```

- everything is a function
- functions always take one single paremeter
- functions with more arguments are functions returning a function

Crash course in functional programing [2]

Functions are regular objects - godbolt

```
1  apply_operator :: (a -> a -> a) -> a -> a -> a
2  apply_operator op x y = op x y
3  inc = apply_operator add 1
4  double = apply_operator (\ x y -> x * y) 2
```

They can replace loops - godbolt

```
5  a = [1,2,3,4,5]
6  b = map double a
7  -- b is [2,4,6,8,10]
8  fact5 = fold (*) 1 a -- 120
```

Crash course in functional programing [3]

Actual looping is done via recursion - godbolt

```haskell
-- map definition
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs

-- fold definition
fold :: (a -> b -> b) -> b -> [a] -> b
fold f z []     = z
fold f z (x:xs) = f x (fold f z xs)
```

Crash course in functional programing [4]

Evaluation is lazy - godbolt

```
1  allEven = filter even [2..]
2  a = take 5 allEven
3  -- a is [2 4 6 8 10]
```

Crash course in functional programing [4]

Evaluation is lazy - godbolt

```
1  allEven = filter even [2..]
2  a = take 5 allEven
3  -- a is [2 4 6 8 10]
```

Prime computation - godbolt

```
4  filterPrime :: [Int] -> [Int]
5  filterPrime (p:xs) =
6    p : filterPrime (filter (\ x -> x `mod` p /= 0) xs)
7  allPrimes = filterPrime [2..]
8  a = take 10 allPrimes
9  -- a is [2 3 5 7 11 13 17 23 29 31]
```

What to conclude ?

- Nice, very elegant, but not so practical...

What to conclude ?

- Nice, very elegant, but not so practical...
- can we use that in python and C^{++}?
- and is all that really efficient ?

What to conclude ?

- Nice, very elegant, but not so practical...
- can we use that in python and C^{++}?
- and is all that really efficient ?

Short answers : Yes, and double Yes

Practical usage and efficiency

1 Virtues of functional programming

2 Practical usage and efficiency
- Functional programming in C++
- Efficiency

3 It's all about helping the compilers

4 Do more at compile time

5 Conclusion

Everything is a function, also in C++

Concept

- a class can implement `operator()`
- it's then a "functor" (no relation to functors in math)
- allows to use objects in place of functions
- with constructors and data members

First functor - godbolt

```
1  struct Adder {
2    int m_increment;
3    Adder(int increment) : m_increment(increment) {}
4    int operator()(int a) { return a + m_increment; }
5  };
6  Adder add_42{42};
7  int i = add_42(3); // 45
```

And we also have lambdas

Definition

a lambda is a function with no name

Syntax

```
[captures] (args) -> type { code; }
```

The type specification is optional

Usage example - godbolt

```
1  auto add = [](int a, int b) { return a+b; };
2  auto add_42 = [&add](int b) { return add(42, b); };
```

Lambdas are essentially syntactic sugar

Lambdas

```
1  [&sum, offset](int x) { sum += x + offset; }
```

Are just functors - cppinsight

```
1  struct MyFunc {
2    int& m_sum;
3    int m_offset;
4    MyFunc(int& s, int o) : m_sum(s), m_offset(o) {}
5    void operator(int x) { m_sum += x + m_offset; }
6  };
7  MyFunc(sum, offset)
```

By the way, as lambdas are functors, they can inherit from each other !
And this can be super useful.

Functions are regular objects

They can be passed around - godbolt

```
1  auto apply = [](auto f, auto a, auto b)
2    { return f(a, b); };
3  auto inc = [&](auto a) { return apply( add, 1, a ); };
4  auto doubleF = [&](auto a) {
5    return apply([](auto x, auto y) {return x*y;}, 2, a);
6  };
```

They can replace loops - godbolt

```
7  std::vector a{1,2,3,4,5};
8  auto fact =
9    std::ranges::fold_right(a, 1, std::multiplies());
10 std::ranges::transform
11   (begin(a), end(a), begin(a), doubleF);
12 -- a is [2,4,6,8,10]
```

Ranges

Reason of being

- provide easy manipulation of sets of data via views
- simplify the horrible iterator syntax

Syntax

Based on Unix like pipes, and used in range based loops

Example code - godbolt

```
1  std::vector<int> numbers{1, 2, 3, 4, 5, 6};
2  auto results =
3    numbers | filter([](int n){ return n % 2 == 0; })
4            | transform([](int n){ return n * 2; });
5  for (auto v: results) std::cout << v << " ";
```

Ranges are lazy evaluated

Example code - godbolt

```
1  using Gen = std::generator<unsigned int>;
2  Gen source() { for (unsigned x = 2; ; x++) co_yield x; }
3  Gen filter(Gen& g, int prime) {
4    for (unsigned x : g) { if ((x % prime) != 0) co_yield x; }
5  }
6  Gen prime(Gen& g) {
7    auto p = *g.begin(); co_yield p;
8    auto ng = filter(g, p);
9    for (auto n : prime(ng)) co_yield n;
10 }
11 auto numbers = source();
12 for (unsigned p : prime(numbers) | std::views::take(10)) {
13   std::cout << p << std::endl;
14 }
```

Practical usage and efficiency

1 Virtues of functional programming

2 Practical usage and efficiency
- Functional programming in C++
- Efficiency

3 It's all about helping the compilers

4 Do more at compile time

5 Conclusion

Is this efficient ?

Let's compile this code - godbolt

```
1  std::array<int, 6> numbers{1, 2, 3, 4, 5, 6};
2  auto results = numbers |
3    std::ranges::views::transform([](int n){ return n * 23; });
4  for (auto n : results) std::cout << n << " ";
```

Generated code with -O3

```
1  .L2:   imul    esi, DWORD PTR [rbx], 23
2         mov     edi, OFFSET FLAT:_ZSt4cout
3         add     rbx, 4
4         call    std::basic_ostream::operator<<(int)
5         cmp     rbp, rbx
6         jne     .L2
7  .LC0:  .long   1
8         .long   2
9         ...
```

Is this efficient ?

Let's compile this code - godbolt

```
1  std::array<int, 6> numbers{1, 2, 3, 4, 5, 6};
2  auto results = numbers |
3    std::ranges::views::transform([](int n){ return n * 23; });
4  for (auto n : results) std::cout << n << " ";
```

Generated code with -O3

~Optimized

```
1  .L2:   imul    esi, [rbx], 23
2         mov     edi, OFFSET FLAT:_ZSt4cout
3         add     rbx, 4
4         call    std::basic_ostream::operator<<(int)
5         cmp     rbp, rbx
6         jne     .L2
7  .LC0:  .long   1
8         .long   2
9         ...
```

Is that efficient ?

Another bit of C++code - godbolt

```
1  int foo() {
2    std::vector<int> a{1,2,3,4,5};
3    return std::ranges::fold_right(a, 1, std::multiplies());
4  }
5  std::cout << foo() << std::endl;
```

We get (gcc 14)

```
1  foo():
2    ...
3    call    operator new(unsigned long)
4    movdqa  xmm0, XMMWORD PTR .LC0[rip]
5    mov     esi, 20
6    mov     DWORD PTR [rax+16], 5
7    mov     rdi, rax
8    movups  XMMWORD PTR [rax], xmm0
9    call    operator delete(void*, unsigned long)
10   mov     eax, 120
```

Is that efficient ?

Another bit of C++code - godbolt

```
1  int foo() {
2    std::vector<int> a{1,2,3,4,5};
3    return std::ranges::fold_right(a, 1, std::multiplies());
4  }
5  std::cout << foo() << std:
```

Weird

We get (gcc 14)

```
1  foo():
2    ...
3    call    operator new(unsigned long)
4    movdqa  xmm0, XMMWORD PTR .LC0[rip]
5    mov     esi, 20
6    mov     DWORD PTR [rax+16], 5
7    mov     rdi, rax
8    movups  XMMWORD PTR [rax], xmm0
9    call    operator delete(void*, unsigned long)
10   mov     eax, 120
```

How do we explain that ?

Facts

- what could be precomputed was precomputed
- all functional code is gone
- but vector still created, filled and dropped
  - not even used !

Hypothesis

- creation/deletion of the vector had to be kept in case of side effects
- compiler did not know whether there would be some
- Bottom line :
  - not enough information given to the compiler
  - a.k.a. compiler not clever enough

Switching from vector to array

Another bit of C++code - godbolt

```
1  int foo() {
2    std::array<int, 5> a{1,2,3,4,5};
3    return std::ranges::fold_right(a, 1, std::multiplies());
4  }
5  foo();
```

We get (gcc 14)

```
1  foo():
2        mov     eax, 120
```

Switching from vector to array

Another bit of C++code - godbolt

```
1  int foo() {
2    std::array<int, 5> a{1,2,3,4,5};
3    return std::ranges::fold_left(a, 1, std::multiplies());
4  }
5  foo();
```

Optimized

We get (gcc 14)

```
1  foo():
2        mov     eax, 120
```

Using a more clever compiler

Another bit of C++code - godbolt

```
1  int foo() {
2    std::vector<int> a{1,2,3,4,5};
3    return std::ranges::fold_right(a, 1, std::multiplies());
4  }
5  foo();
```

We get (clang 18 / gcc 15)

```
1  foo():
2        mov     eax, 120
```

Using a more clever compiler

Another bit of C++code - godbolt

```
1  int foo() {
2    std::vector<int> a{1,2,3,4};
3    return std::ranges::fold_left(a, 1, std::multiplies());
4  }
5  foo();
```

Optimized

We get (clang 18 / gcc 15)

```
1  foo():
2        mov     eax, 120
```

It's all about helping the compilers

1 Virtues of functional programming

2 Practical usage and efficiency

3 It's all about helping the compilers
- constness, exceptions
- Avoiding useless copying

4 Do more at compile time

5 Conclusion

Constness

The const keyword

- indicates that the variable is constant
- this is all checked at compile time
- and used by the compiler for optimization

```
1  int const i = 6;
2  const int i = 6; // equivalent
3  // error: i is constant
4  i = 5;
5  auto const j = i; // works with auto
```

constness

Non const code - godbolt

```
1  void foo( int& k );
2  int bar() {
3    int k = 10;
4    foo(k);
5    return k*k+2*k+1;
6  }
```

We get

```
1  bar():
2    sub  rsp, 24
3    lea  rdi, [rsp+12]
4    mov  DW PTR [rsp+12], 10
5    call foo(int&)
6    mov  eax, DW PTR [rsp+12]
7    add  rsp, 24
8    mov  edx, eax
9    imul edx, eax
10   lea  eax, [rdx+1+rax*2]
```

constness

const aware code - godbolt

```
1  void foo( const int& k );
2  int bar() {
3    const int k = 10;
4    foo(k);
5    return k*k+2*k+1;
6  }
```

We get

```
1  bar():
2    sub  rsp, 24
3    lea  rdi, [rsp+12]
4    mov  DW PTR [rsp+12], 10
5    call foo(int const&)
6    mov  eax, 121
7    add  rsp, 24
```

Data replication can also help

data replication - godbolt

```
1  void foo( int k );
2  int bar() {
3    int k = 10;
4    foo(k);
5    return k*k+2*k+1;
6  }
```

We get

```
1  bar():
2    sub     rsp, 8
3    mov     edi, 10
4    call    foo(int)
5    mov     eax, 121
6    add     rsp, 8
```

C++ exception support

After a lot of thinking and experiencing, the conclusions of the community on exception handling are :

- Never write an exception specification
- Except possibly an empty one

C++ exception support

After a lot of thinking and experiencing, the conclusions of the community on exception handling are :

- Never write an exception specification
- Except possibly an empty one

one of the reasons : performance

- does not allow compiler optimizations
- on the contrary forces extra checks

C++ exception support

After a lot of thinking and experiencing, the conclusions of the community on exception handling are :

- Never write an exception specification
- Except possibly an empty one

one of the reasons : performance

- does not allow compiler optimizations
- on the contrary forces extra checks

Introducing noexcept

```
1  int f() noexcept;
```

- somehow equivalent to throw()
- meaning no exception can go out of the function
- but is checked at compile time
- thus allowing compiler optimizations

Impact on generated code - exceptions

C++ code - godbolt

```
1  struct MyExcept{};
2  int f(int a); // may throw
3
4  int foo() {
5    try {
6      int a = 23;
7      return f(a) + f(-a);
8    } catch (MyExcept& e) {
9      return 0;
10   }
11 }
```

Generated code

```
1   foo():
2     push    rbx
3     mov     edi, 23
4     call    f(int)
5     mov     edi, -23
6     mov     ebx, eax
7     call    f(int)
8     add     eax, ebx
9   .L1:
10    pop     rbx
11    ret
12    mov     rdi, rax
13    mov     rax, rdx
14    jmp     .L2
15    foo() [clone .cold]:
16  .L2:
17    sub     rax, 1
18    jne     .L8
19    call    __cxa_begin_catch
20    call    __cxa_end_catch
21    xor     eax, eax
22    jmp     .L1
23  .L8:
24    call    _Unwind_Resume
```

Impact on generated code - noexcept

C++ code - godbolt

```
1  struct MyExcept{};
2  int f(int a) noexcept;
3
4  int foo() {
5    try {
6      int a = 23;
7      return f(a) + f(-a);
8    } catch (MyExcept& e) {
9      return 0;
10   }
11 }
```

Generated code

```
1  foo():
2    push    rbx
3    mov     edi, 23
4    call    f(int)
5    mov     edi, -23
6    mov     ebx, eax
7    call    f(int)
8    add     eax, ebx
9    pop     rbx
```

It's all about helping the compilers

1 Virtues of functional programming

2 Practical usage and efficiency

3 It's all about helping the compilers
- constness, exceptions
- Avoiding useless copying

4 Do more at compile time

5 Conclusion

Useless copying: the problem

Inefficient code

```
1  void swap(std::vector<int> &a,
2            std::vector<int> &b) {
3    std::vector<int> c = a;
4    a = b;
5    b = c;
6  }
7  std::vector<int> v(10000), w(10000);
8  ...
9  swap(v, w);
```

Useless copying: the problem

Inefficient code

```
1  void swap(std::vector<int> &a,
2            std::vector<int> &b) {
3    std::vector<int> c = a;
4    a = b;
5    b = c;
6  }
7  std::vector<int> v(10000), w(10000);
8  ...
9  swap(v, w);
```

What happens during swap

- one allocation and one release for 10k `int`s
- a copy of 30k `int`s

Move semantics

The idea

- a new type of reference : rvalue references
  - used for "moving" objects
  - denoted by &&
- 2 new members in every class, with move semantic :

  a move constructor similar to copy constructor

  a move assignment operator similar to assignment operator (now called copy assignment operator)
  - used when original object can be reused

Move semantics

The idea

- a new type of reference : rvalue references
  - used for "moving" objects
  - denoted by &&
- 2 new members in every class, with move semantic :

  a move constructor similar to copy constructor

  a move assignment operator similar to assignment operator (now called copy assignment operator)
  - used when original object can be reused

Practically

```
1  T(const T&  other); // copy construction
2  T(      T&& other); // move construction
3  T& operator=(const T&  other); // copy assignment
4  T& operator=(      T&& other); // move assignment
```

Move semantics

A few important points concerning move semantic

- move assignment operator is allowed to destroy source
  - so do not reuse source afterward
- if not implemented, move falls back to copy version
- move is called by the compiler whenever possible
  - and can be forced via `std::move`

Move semantics

A few important points concerning move semantic

- move assignment operator is allowed to destroy source
  - so do not reuse source afterward
- if not implemented, move falls back to copy version
- move is called by the compiler whenever possible
  - and can be forced via `std::move`

Practically

```
1  void swap(T &a, T &b) {
2    T c = std::move(a);  // move construct
3    a = std::move(b);    // move assign
4    b = std::move(c);    // move assign
5  }
```

No allocation, no release, 3x3 pointers copied

Guaranteed copy elision

What is copy elision

```
1  struct Foo { ... };
2  Foo f() {
3    return Foo();
4  }
5  int main() {
6    // No copy here, compiler must elude the copy
7    Foo foo = f();
8  }
```

From C++17 on

The elision is guaranteed.

- superseeds move semantic in some cases
- so do not hesitate anymore to return plain objects in generators
  - and ban pointers for good

Do more at compile time

1 Virtues of functional programming

2 Practical usage and efficiency

3 It's all about helping the compilers

4 Do more at compile time
- Templates
- `constexpr` and `if constexpr`
- Virtuality : a counter example

5 Conclusion

Templates

Concept

- The C^{++}way to write reusable code
  - aka macros on steroids
- Applicable to functions and objects

```
1  template<typename T>
2  const T & max(const T &A, const T &B) {
3    return A > B ? A : B;
4  }
5
6  template<typename T>
7  struct Vector {
8    int m_len;
9    T* m_data;
10 };
```

Templates

Warning

These are really like macros

- they need to be defined before used
  - so all templated code has to be in headers
- they are compiled n times
- and thus each version is optimized individually !

```
1  template<typename T>
2  T func(T a) {
3    return a;
4  }
```

func(3)

```
1  int func(int a) {
2    return a;
3  }
```

func(5.2)

```
1  double func(double a) {
2    return a;
3  }
```

A realistic template usage

Generic printing - godbolt

```
1  template<typename T> struct Printer {
2    Printer(T const& obj) {
3      std::cout << "(" << typeid(T).name()
4                << ") " << obj << "\n";
5    }
6  };
7  Printer{5};                          // (i) 5
8  Printer("a text string");             // (A14_c) a text string
```

A realistic template usage

Generic printing - godbolt

```
1  template<typename T>
2  concept Container = requires( T v1 ) {
3    begin(v1);
4    end(v1);
5  };
6  template<Container T> struct Printer<T> {
7    Printer(T const& obj) {
8      std::cout << "(" << typeid(T).name() << ") ";
9      for( auto& item: obj) { std::cout << item << " "; }
10     std::cout << "\n";
11   }
12 };
13 Printer(std::vector<int>{1,2,3});
14 // (St6vectorIiSaIiEE) 1 2 3
15 Printer(std::array<double, 3>{2,3,4});
16 // (St5arrayIdLm3EE) 2 3 4
```

Pros and cons of templates

Gains

- more generic code, thus less code
- better compiler optimization
- allows some compile time processing
  - also known as template metaprograming

The drawbacks

- not so trivial syntax, especially when variadic
- heavy for the compiler if abused
- only usable for what you know at compile time

Do more at compile time

1 Virtues of functional programming

2 Practical usage and efficiency

3 It's all about helping the compilers

4 Do more at compile time
- Templates
- `constexpr` and `if constexpr`
- Virtuality : a counter example

5 Conclusion

Generalized Constant Expressions

Reason of being

- use functions to compute constant expressions at compile time

Generalized Constant Expressions

Reason of being

- use functions to compute constant expressions at compile time

Example

```
1  constexpr int f(int x) {
2    if (x > 1) return x * f(x - 1);
3    return 1;
4  }
5  constexpr int a = f(5); // computed at compile-time
6  int b = f(5); // maybe computed at compile-time
7  int n = ...;  // runtime value
8  int c = f(n); // computed at runtime
```

if constexpr

Compile time if statement

- takes a generalized constant expression
- decides which branch to keep at compile time
- drops the other branch

if constexpr

Compile time if statement

- takes a generalized constant expression
- decides which branch to keep at compile time
- drops the other branch

Generic printing - godbolt

```
1  template<typename T> struct Printer {
2    Print(T const& obj) {
3      std::cout << "(" << typeid(T).name() << ") ";
4      if constexpr( Container<T> ) {
5        for( auto& item: obj) { std::cout << item << " "; }
6      } else {
7        std::cout << obj;
8      }
9      std::cout << "\n";
10   }
11 };
```

Do more at compile time

1 Virtues of functional programming

2 Practical usage and efficiency

3 It's all about helping the compilers

4 Do more at compile time
- Templates
- `constexpr` and `if constexpr`
- Virtuality : a counter example

5 Conclusion

Virtuality in a nutshell

Principle

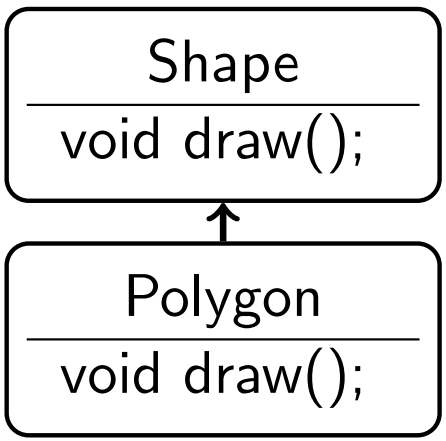
- a base class (aka interface) declares some method virtual
- children can overload these methods (as any other)
- for these method, late binding is applied
- that is most precise type is used

Virtuality in a nutshell

Principle

- a base class (aka interface) declares some method virtual
- children can overload these methods (as any other)
- for these method, late binding is applied
- that is most precise type is used

```
1  Polygon p;
2  p.draw(); // Polygon.draw
3
4  Shape & s = p;
5  s.draw(); // Shape.draw
```

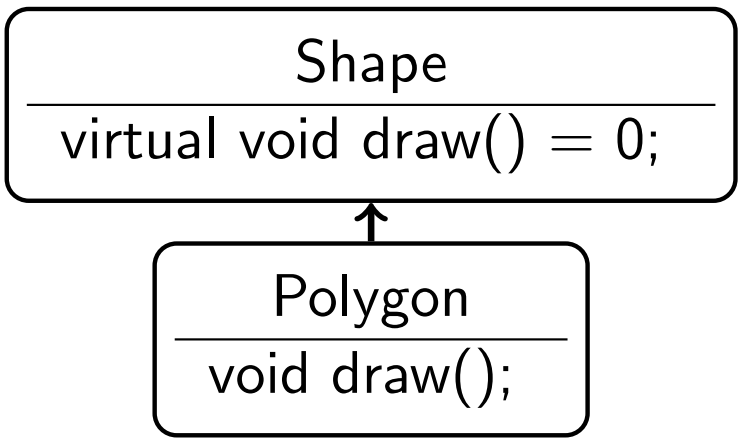


Virtuality in a nutshell

Principle

- a base class (aka interface) declares some method virtual
- children can overload these methods (as any other)
- for these method, late binding is applied
- that is most precise type is used

```
1  Polygon p;
2  p.draw(); // Polygon.draw
3
4  Shape & s = p;
5  s.draw(); // Polygon.draw
```



The price of virtuality

Actual implementation

- each object has an extra pointer
- to a "virtual table" object in memory
- where each virtual function points to the right overload

Cost

- extra virtual table in memory, per type
- each virtual call does
  - retrieve virtual table pointer
  - load virtual table into memory
  - lookup right call
  - effectively call
- and is thus much more costful than standard function call
- up to 20% difference in terms of nb of instructions

Actual price of virtuality

Comparison with templates - godbolt / godbolt

```
1   struct Interface {
2     virtual void tick(float n) = 0;
3   };
4   struct Counter : Interface {
5     float sum{0};
6     void tick(float v) override
7       { sum += v; }
8   };
9   void foo(Interface& c) {
10    for (int i = 0; i < 80000; ++i) {
11      for (int j = 0; j < i; ++j) {
12        c.tick(j);
13      }
14    }
15  }
16  int main() {
17    auto obj =
18      std::make_unique<Counter>();
19    foo(*obj);
20  }
```

```
21  struct Counter {
22    float sum{0};
23    void tick(float v) { sum += v; }
24  };
25  template<typename CounterType>
26  void foo(CounterType& c) {
27    for (int i = 0; i < 80000; ++i) {
28      for (int j = 0; j < i; ++j) {
29        c.tick(j);
30      }
31    }
32  }
```

Actual price of virtuality

Comparison with templates - godbolt / godbolt

```
1   struct Interface {
2     virtual void tick(float n) = 0;
3   };
4   struct Counter : Interface {
5     float sum{0};
6     void tick(float v) override
7       { sum += v; }
8   };
9   void foo(Interface& c) {
10    for (int i = 0; i < 80000; ++i) {
11      for (int j = 0; j < i; ++j) {
12        c.tick(j);
13      }
14    }
15  }
16  int main() {
17    auto obj =
18      std::make_unique<Counter>();
19    foo(*obj);
20  }
```

```
21  struct Counter {
22    float sum{0};
23    void tick(float v) { sum += v; }
24  };
25  template<typename CounterType>
26  void foo(CounterType& c) {
27    for (int i = 0; i < 80000; ++i) {
28      for (int j = 0; j < i; ++j) {
29        c.tick(j);
30      }
31    }
32  }
```

Timing	Time(s)	Nb instr(G)
virtual	10.8	35.2
templ	2.97	8.9

- measured on EPYC 7552, with gcc 9.1 and perf

A few explanations

Some consequences of virtuality

- more branching, killing the pipeline
  - here 6.4M vs 0.8M branches !
  - as virtual calls are branches
- lack of inlining possibilities
- lack of optimizations after inlining
  - e.g. auto vectorization

Note that the compiler is trying hard to help

- when it can, when it knows so give it all the knowledge !
- typical on my example

A few explanations

Some consequences of virtuality

- more branching, killing the pipeline
  - here 6.4M vs 0.8M branches !
  - as virtual calls are branches
- lack of inlining possibilities
- lack of optimizations after inlining
  - e.g. auto vectorization

Note that the compiler is trying hard to help

- when it can, when it knows so give it all the knowledge !
- typical on my example
  - declare `obj` on the stack and the compiler will "drop" virtuality
    - again : drop pointers !
  - gcc 15 does much better : 16G instructions and 3s

Should I use virtuality ?

Yes, when you cannot know anything at compile time

Typical cases

- you have no knowledge of the implementations of an interface
  - new ones may even be loaded dynamically via shared libraries
- you mix various implementations in a container
  - e.g. `std::vector<MyInterface>`
  - and there is no predefined set of implementations

Should I use virtuality ?

Yes, when you cannot know anything at compile time

Typical cases

- you have no knowledge of the implementations of an interface
  - new ones may even be loaded dynamically via shared libraries
- you mix various implementations in a container
  - e.g. `std::vector<MyInterface>`
  - and there is no predefined set of implementations

Typical alternatives

- templates when everything is compile time
  - allows full optimization of each case
  - and even static polymorphism through CRTP
    - Curiously recurring template pattern
- `std::variant`, `std::any` and visitor
  - when type definitions are known at compile type
  - but not necessary their usage

std::variant, std::any

Purpose

- type safe union and "void*"
- with visitor pattern

std::variant, std::any

Purpose

- type safe union and "void*"
- with visitor pattern

Example code - godbolt

```
1  using Message = std::variant<int, std::string>;
2  Message createMessage(bool error) {
3    if (error) return "Error"; else return 42;
4  }
5  int i = std::get<int>(createMessage(false));
6  struct Visitor {
7    void operator()(int n) const {
8      std::cout << "Int " << n << std::endl;
9    }
10   void operator()(const std::string &s) const {
11     std::cout << "String \"" << s << "\"" << std::endl;
12   }
13 };
14 std::visit(Visitor{}, createMessage(true));
```

std::variant, std::any

Or you use lambdas and their inheritance - godbolt

```
1  template <class ... P> struct Combine : P... {
2    using P::operator()...;
3  };
4  template <class ... F> Combine<F...> combine(F... fs) {
5    return { fs ... };
6  }
7  using Message = std::variant<int, std::string>;
8  Message createMessage(bool error) {
9    if (error) return "Error"; else return 42;
10 }
11 auto f = combine(
12     [](int n) { std::cout << "Int " << n << std::endl; },
13     [](string const &s) {
14       std::cout << "String \"" << s << "\"" << std::endl;
15     });
16 std::visit(f, createMessage(true));
```

A Visitor example

Virtual vs variant - godbolt

```
1  struct Point { virtual float getR() = 0; };
2  struct XYZPoint : Point {
3    float x, y, z;
4    float getR() override { return std::sqrt(x*x+y*y+z*z); }; };
5  struct RTPPoint : Point {
6    float r, theta, phi;
7    float getR() override { return r; } };
8  float sumR(std::vector<std::unique_ptr<Point>>& v) {
9    return std::accumulate(begin(v), end(v), 0.0f,
10   [&](float s, std::unique_ptr<Point>& p) { return s + p->getR();} );
11 }
12
13 struct XYZPoint { float x,y,z; };  struct RTPPoint { float r, theta, phi; };
14 using Point=std::variant<XYZPoint, RTPPoint>;
15 float sumR(std::vector<Point>& v) {
16   auto getR = combine(
17   [](XYZPoint& p) { return std::sqrt(p.x*p.x+p.y*p.y+p.z*p.z); },
18   [](RTPPoint& p) { return p.r; });
19   return std::accumulate(begin(v), end(v), 0.0f,
20   [&](float s, Point& p) { return s + std::visit(getR, p);} );
21 }
```

A Visitor example

Virtual vs variant - godbolt

```
1  struct Point { virtual float getR() = 0; };
2  struct XYZPoint : Point {
3    float x, y, z;
4    float getR() override { ret                *x+y*y+z*z); }; };
5  struct RTPPoint : Point {
6    float r, theta, phi;
7    float getR() overrid                     };
8  float sumR(std::vector                    _ptr<Point>>& v) {
9    return std::accumulate                  (v), end(v), 0.0f,
10   [&](float s, std::unique_ptr<Point>& p) { return s + p->getR();} );
11 }
12
13 struct XYZPoint { float x,y,z; };  struct RTPPoint { float r, theta, phi; };
14 using Point=std::variant<XYZPoint, RTPPoi
15 float sumR(std::vector<Point>& v) {
16   auto getR = combine(
17   [](XYZPoint& p) { return std                *p.y+p.z*p.z); },
18   [](RTPPoint& p) { return r
19   return std::accumulate(be                   , 0.0f,
20   [&](float s, Point& p) { i                  std::visit(getR, p);} );
21 }
```

took 3500μs

took 2050μs

Conclusion

Key messages of the day

- functional programming is useful
  - can simplify the code
  - ensures thread safety
  - helps the compiler to optimize better
- the key to performance is the compiler
  - help it by providing information
  - constness, noexcept, fixed lengths, ...
- a lot can be done at compile time
  - find out what you know at compile time
  - make best use of it (templates, variants, ...)

Index of Godbolt code examples

1. func prog basics - godbolt 5
2. func prog apply - godbolt 6
3. func prog no loops - godbolt 6
4. func prog map/fold - godbolt 7
5. func prog lazy - godbolt 8
6. func prog primes - godbolt 8
7. C++functor - godbolt 11
8. C++lambdas - godbolt 12
9. C++ func - godbolt 14
10. C++ no loop - godbolt 14
11. C++ ranges - godbolt 15
12. C++ primes - godbolt 16
13. transform efficiency - godbolt 18
14. fold efficiency - godbolt 19
15. array vs vector - godbolt 21
16. better compiler - godbolt 22
17. missing const - godbolt 25
18. const usage - godbolt 26
19. data replication - godbolt 27
20. exception - godbolt 29
21. noexcept - godbolt 30
22. Generic printing - godbolt 39
23. if constexpr - godbolt 44
24. Counter virtual - godbolt 49
25. Counter templates - godbolt 49
26. Visitor - godbolt 52
27. Lambda inheritance - godbolt 53
28. Virtual vs variant - godbolt 54