Writing efficient code

Sébastien Ponce
sebastien.ponce@cern.ch

CERN

Thematic CERN School of Computing 2024

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

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, ...)

thematic CERN School of Computing

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

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

They can replace loops - godbolt

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

Crash course in functional programing [3]

Actual looping is done via recursion - godbolt

```
-- 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

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

Crash course in functional programing [4]

Evaluation is lazy - godbolt

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

Prime computation - godbolt

```haskell
filterPrime :: [Int] -> [Int]
filterPrime (p:xs) =
  p : filterPrime (filter (\ x -> x `mod` p /= 0) xs)
allPrimes = filterPrime [2..]
a = take 10 allPrimes
-- 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

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

```cpp
struct Adder {
  int m_increment;
  Adder(int increment) : m_increment(increment) {}
  int operator()(int a) { return a + m_increment; }
};
Adder add_42{42};
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

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

Lambdas are essentially syntactic sugar

Lambdas

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

Are just functors - cppinsight

```
struct MyFunc {
  int& m_sum;
  int m_offset;
  MyFunc(int& s, int o) : m_sum(s), m_offset(o) {}
  void operator(int x) { m_sum += x + m_offset; }
};
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

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

They can replace loops - godbolt

```
std::vector a{1,2,3,4,5};
auto fact =
  std::ranges::fold_right(a, 1, std::multiplies());
std::ranges::transform
  (begin(a), end(a), begin(a), doubleF);
-- 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

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

Ranges are lazy evaluated

Example code - godbolt

```
using Gen = std::generator<unsigned int>;
Gen source() { for (unsigned x = 2; ; x++) co_yield x; }
Gen filter(Gen& g, int prime) {
  for (unsigned x : g) { if ((x % prime) != 0) co_yield x; }
}
Gen prime(Gen& g) {
  auto p = *g.begin(); co_yield p;
  auto ng = filter(g, p);
  for (auto n : prime(ng)) co_yield n;
}
auto numbers = source();
for (unsigned p : prime(numbers) | std::views::take(10)) {
  std::cout << p << std::endl;
}
```

Practical usage and efficiency

Is this efficient ?

Let's compile this code - godbolt

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

Generated code with -O3

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

Is this efficient ?

Let's compile this code - godbolt

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

Generated code with -O3

```asm
.L2:   imul    esi, [illegible]bx], 23
       mov     edi, [illegible]FLAT:_ZSt4cout
       add     rbx, 4
       call    std::basic_ostream::operator<<(int)
       cmp     rbp, rbx
       jne     .L2
.LC0:  .long   1
       .long   2
       ...
```

~Optimized

Is that efficient ?

Another bit of C^{++}code - godbolt

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

We get (gcc trunk

```
foo():
  ...
  call     operator new(unsigned long)
  movdqa   xmm0, XMMWORD PTR .LC0[rip]
  mov      esi, 20
  mov      DWORD PTR [rax+16], 5
  mov      rdi, rax
  movups   XMMWORD PTR [rax], xmm0
  call     operator delete(void*, unsigned long)
  mov      eax, 120
```

Is that efficient ?

Another bit of C++ code - godbolt

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

We get (gcc trunk

```
foo():
  ...
  call     operator new(unsigned long)
  movdqa   xmm0, XMMWORD PTR .LC0[rip]
  mov      esi, 20
  mov      DWORD PTR [rax+16], 5
  mov      rdi, rax
  movups   XMMWORD PTR [rax], xmm0
  call     operator delete(void*, unsigned long)
  mov      eax, 120
```

Weird

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 :
  - no enough information given to the compiler
  - a.k.a. compiler not clever enough

Switching from vector to array

Another bit of C++ code - godbolt

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

We get (gcc trunk)

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

Switching from vector to array

Another bit of C++ code - godbolt

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

Optimized

We get (gcc trunk)

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

Using a more clever compiler

Another bit of C++ code - godbolt

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

We get (clang 18)

```asm
foo():
        mov     eax, 120
```

Using a more clever compiler

Another bit of C++ code - godbolt

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

Optimized

We get (clang 18)

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

It's all about helping the compilers

Constness

The const keyword

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

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

constness

Non const code - godbolt

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

We get

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

constness

const aware code - godbolt

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

We get

```asm
bar():
   sub  rsp, 24
   lea  rdi, [rsp+12]
   mov  DW PTR [rsp+12], 10
   call foo(int const&)
   mov  eax, 121
   add  rsp, 24
```

CERN School of Computing

Data replication can also help

data replication - godbolt

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

We get

```
bar():
  sub     rsp, 8
  mov     edi, 10
  call    foo(int)
  mov     eax, 121
  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

```
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

```cpp
struct MyExcept{};
int f(int a); // may throw

int foo() {
  try {
    int a = 23;
    return f(a) + f(-a);
  } catch (MyExcept& e) {
    return 0;
  }
}
```

Generated code

```asm
foo():
  push    rbx
  mov     edi, 23
  call    f(int)
  mov     edi, -23
  mov     ebx, eax
  call    f(int)
  add     eax, ebx
.L1:
  pop     rbx
  ret
  mov     rdi, rax
  mov     rax, rdx
  jmp     .L2
  foo() [clone .cold]:
.L2:
  sub     rax, 1
  jne     .L8
  call    __cxa_begin_catch
  call    __cxa_end_catch
  xor     eax, eax
  jmp     .L1
.L8:
  call    _Unwind_Resume
```

Impact on generated code - noexcept

C++code - godbolt

```cpp
struct MyExcept{};
int f(int a) noexcept;

int foo() {
  try {
    int a = 23;
    return f(a) + f(-a);
  } catch (MyExcept& e) {
    return 0;
  }
}
```

Generated code

```asm
foo():
    push    rbx
    mov     edi, 23
    call    f(int)
    mov     edi, -23
    mov     ebx, eax
    call    f(int)
    add     eax, ebx
    pop     rbx
```

It's all about helping the compilers

Useless copying: the problem

Inefficient code

```
void swap(std::vector<int> &a,
          std::vector<int> &b) {
  std::vector<int> c = a;
  a = b;
  b = c;
}
std::vector<int> v(10000), w(10000);
...
swap(v, w);
```

Useless copying: the problem

Inefficient code

```
void swap(std::vector<int> &a,
          std::vector<int> &b) {
  std::vector<int> c = a;
  a = b;
  b = c;
}
std::vector<int> v(10000), w(10000);
...
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

```
T(const T&  other); // copy construction
T(       T&& other); // move construction
T& operator=(const T&  other); // copy assignment
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

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

No allocation, no release, 3x3 pointers copied

Guaranteed copy elision

What is copy elision

```
struct Foo { ... };
Foo f() {
  return Foo();
}
int main() {
  // No copy here, compiler must elude the copy
  Foo foo = f();
}
```

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

Templates

Concept

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

```cpp
template<typename T>
const T & max(const T &A, const T &B) {
  return A > B ? A : B;
}

template<typename T>
struct Vector {
  int m_len;
  T* m_data;
};
```

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 !

```cpp
template<typename T>
T func(T a) {
  return a;
}
```

func(3)

```cpp
int func(int a) {
  return a;
}
```

func(5.2)

```cpp
double func(double a) {
  return a;
}
```

A realistic template usage

Generic printing - godbolt

```
template<typename T> struct Print {
  Print(T const& obj) {
    std::cout << "(" << typeid(T).name()
              << ") " << obj << "\n";
  }
};
Print{5};                       // (i) 5
Print("a text string");         // (A14_c) a text string
```

A realistic template usage

Generic printing - godbolt

```
template<typename T>
concept Container = requires( T v1 ) {
  begin(v1);
  end(v1);
};
template<Container T> struct Printer<T> {
  Printer(T const& obj) {
    std::cout << "(" << typeid(T).name() << ") ";
    for( auto& item: obj) { std::cout << item << " "; }
    std::cout << "\n";
  }
};
Print(std::vector<int>{1,2,3});
// (St6vectorIiSaIiEE) 1 2 3
Printer(std::array<double, 3>{2,3,4});
// (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

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

```
template<typename T> struct Print {
  Print(T const& obj) {
    std::cout << "(" << typeid(T).name() << ") ";
    if constexpr( Container<T> ) {
      for( auto& item: obj) { std::cout << item << " "; }
    } else {
      std::cout << obj;
    }
    std::cout << "\n";
  }
};
```

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

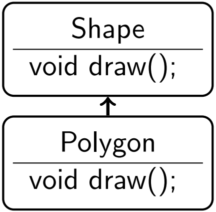
Virtuality in a nutshell

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

```
Polygon p;
p.draw(); // Polygon.draw

Shape & s = p;
s.draw(); // Shape.draw
```



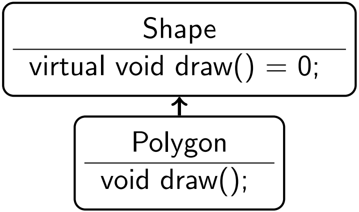
Virtuality in a nutshell

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

```
Polygon p;
p.draw(); // Polygon.draw

Shape & s = p;
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

CERN School of Computing

Actual price of virtuality

Comparison with templates - godbolt / godbolt

```
struct Interface {
  virtual void tick(float n) = 0;
};
struct Counter : Interface {
  float sum{0};
  void tick(float v) override
    { sum += v; }
};
void foo(Interface& c) {
  for (int i = 0; i < 80000; ++i) {
    for (int j = 0; j < i; ++j) {
      c.tick(j);
    }
  }
}
int main() {
  auto obj =
    std::make_unique<Counter>();
  foo(*obj);
}
```

```
struct Counter {
  float sum{0};
  void tick(float v) { sum += v; }
};
template<typename CounterType>
void foo(CounterType& c) {
  for (int i = 0; i < 80000; ++i) {
    for (int j = 0; j < i; ++j) {
      c.tick(j);
    }
  }
}
```

Actual price of virtuality

Comparison with templates - godbolt / godbolt

```
struct Interface {
  virtual void tick(float n) = 0;
};
struct Counter : Interface {
  float sum{0};
  void tick(float v) override
    { sum += v; }
};
void foo(Interface& c) {
  for (int i = 0; i < 80000; ++i) {
    for (int j = 0; j < i; ++j) {
      c.tick(j);
    }
  }
}
int main() {
  auto obj =
    std::make_unique<Counter>();
  foo(*obj);
}
```

```
struct Counter {
  float sum{0};
  void tick(float v) { sum += v; }
};
template<typename CounterType>
void foo(CounterType& c) {
  for (int i = 0; i < 80000; ++i) {
    for (int j = 0; j < i; ++j) {
      c.tick(j);
    }
  }
}
```

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 10/12 does much better : 22/16G instructions and 3s

CERN School of Computing

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

```
using Message = std::variant<int, std::string>;
Message createMessage(bool error) {
  if (error) return "Error"; else return 42;
}
int i = std::get<int>(createMessage(false));
struct Visitor {
  void operator()(int n) const {
    std::cout << "Int " << n << std::endl;
  }
  void operator()(const std::string &s) const {
    std::cout << "String \"" << s << "\"" << std::endl;
  }
};
std::visit(Visitor{}, createMessage(true));
```

std::variant, std::any

Or you use lambdas and their inheritance - godbolt

```
template <class ... P> struct Combine : P... {
  using P::operator()...;
};
template <class ... F> Combine<F...> combine(F... fs) {
  return { fs ... };
}
using Message = std::variant<int, std::string>;
Message createMessage(bool error) {
  if (error) return "Error"; else return 42;
}
auto f = combine(
    [](int n) { std::cout << "Int " << n << std::endl; },
    [](string const &s) {
      std::cout << "String \"" << s << "\"" << std::endl;
    });
std::visit(f, createMessage(true));
```

thematic CSC
CERN School of Computing

A Visitor example

Virtual vs variant - godbolt

```
struct Point { virtual float getR() = 0; };
struct XYZPoint : Point {
  float x, y, z;
  float getR() override { return std::sqrt(x*x+y*y+z*z); }; };
struct RTPPoint : Point {
  float r, theta, phi;
  float getR() override { return r; } };
float sumR(std::vector<std::unique_ptr<Point>>& v) {
  return std::accumulate(begin(v), end(v), 0.0f,
  [&](float s, std::unique_ptr<Point>& p) { return s + p->getR();} );
}

struct XYZPoint { float x,y,z; };  struct RTPPoint { float r, theta, phi; };
using Point=std::variant<XYZPoint, RTPPoint>;
float sumR(std::vector<Point>& v) {
  auto getR = combine(
  [](XYZPoint& p) { return std::sqrt(p.x*p.x+p.y*p.y+p.z*p.z); },
  [](RTPPoint& p) { return p.r; });
  return std::accumulate(begin(v), end(v), 0.0f,
  [&](float s, Point& p) { return s + std::visit(getR, p);} );
}
```

thematic CERN School of Computing

A Visitor example

Virtual vs variant - godbolt

```
struct Point { virtual float getR() = 0; };
struct XYZPoint : Point {
  float x, y, z;
  float getR() override { ret … *x+y*y+z*z); }; };
struct RTPPoint : Point {
  float r, theta, phi;
  float getR() overrid … };
float sumR(std::vector … e_ptr<Point>>& v) {
  return std::accumulate( … n(v), end(v), 0.0f,
  [&](float s, std::unique_ptr<Point>& p) { return s + p->getR();} );
}

struct XYZPoint { float x,y,z; };  struct RTPPoint { float r, theta, phi; };
using Point=std::variant<XYZPoint, RTPP …
float sumR(std::vector<Point>& v) {
  auto getR = combine(
  [](XYZPoint& p) { return s … y*p.y+p.z*p.z); },
  [](RTPPoint& p) { return …
  return std::accumulate(be … ), 0.0f,
  [&](float s, Point& p) { r … + std::visit(getR, p);} );
}
```

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