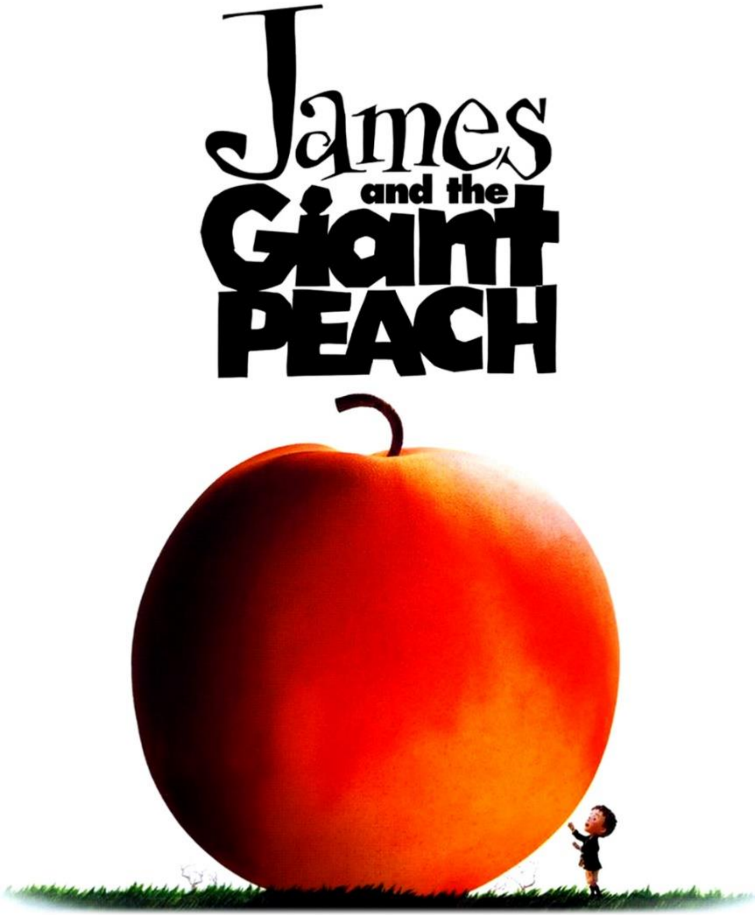
Meta: Toward generative C++

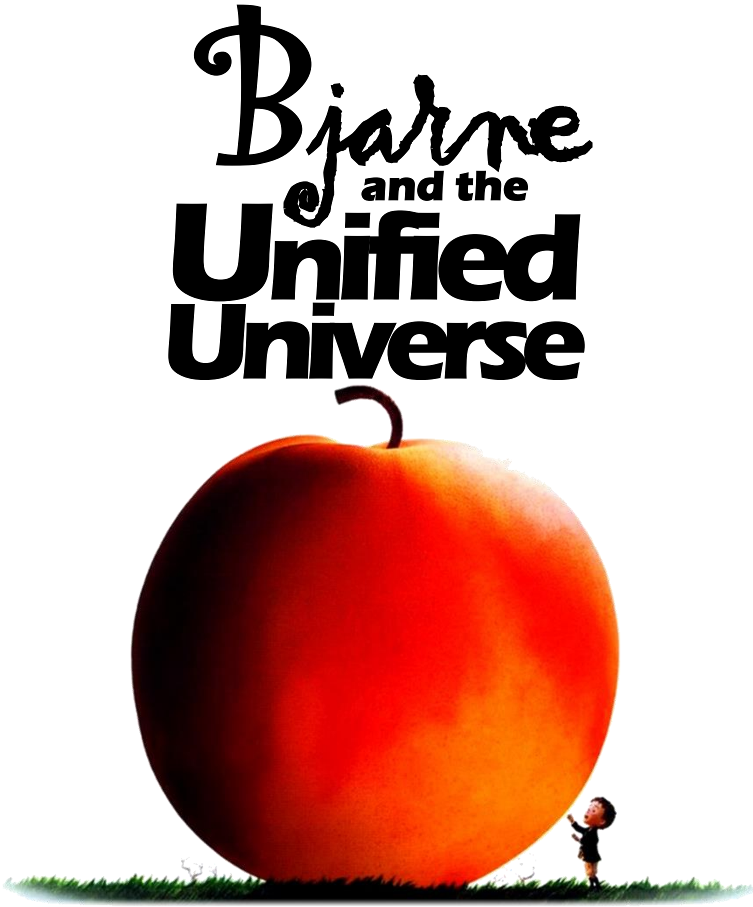
Goal: Making C++ more powerful, and simpler

Herb Sutter

Now Playing



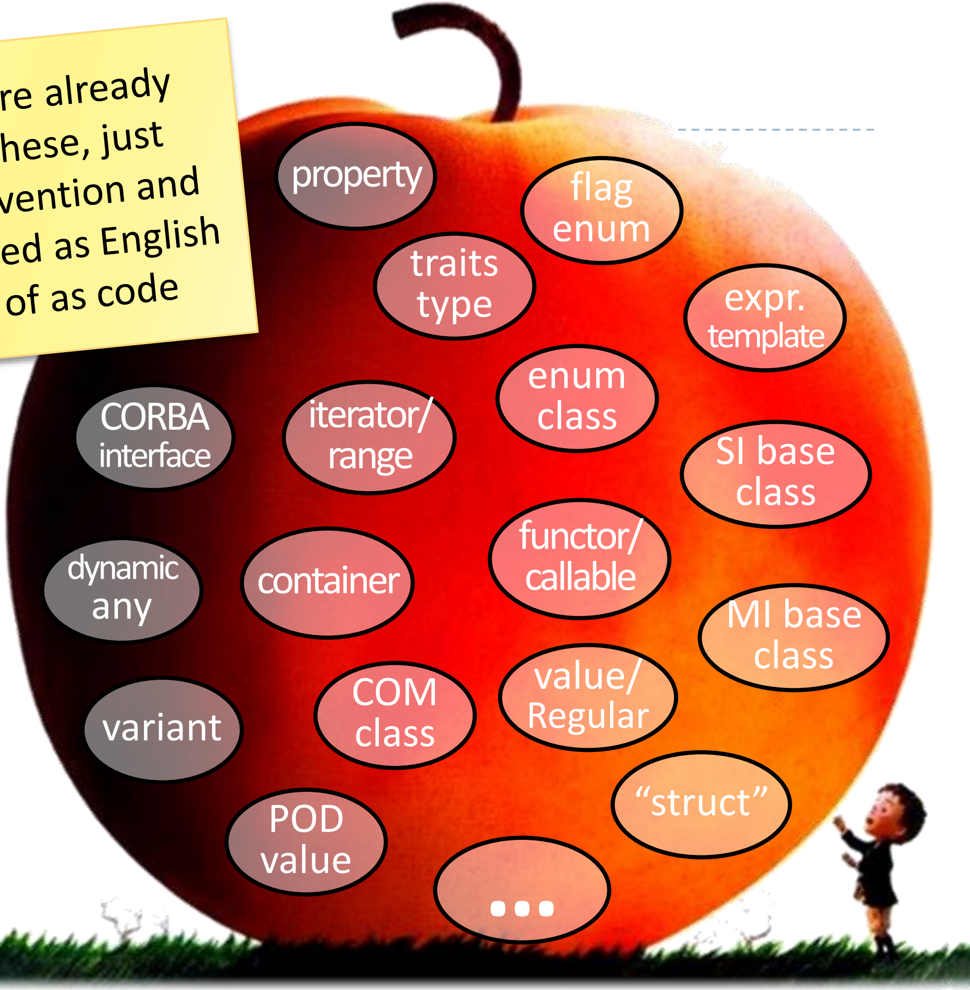
Now Playing



Now Playing

- The C++ type system is unified!

note: we're already writing these, just (a) by convention and (b) described as English instead of as code



Now Playing

- The C++ type system is unified!

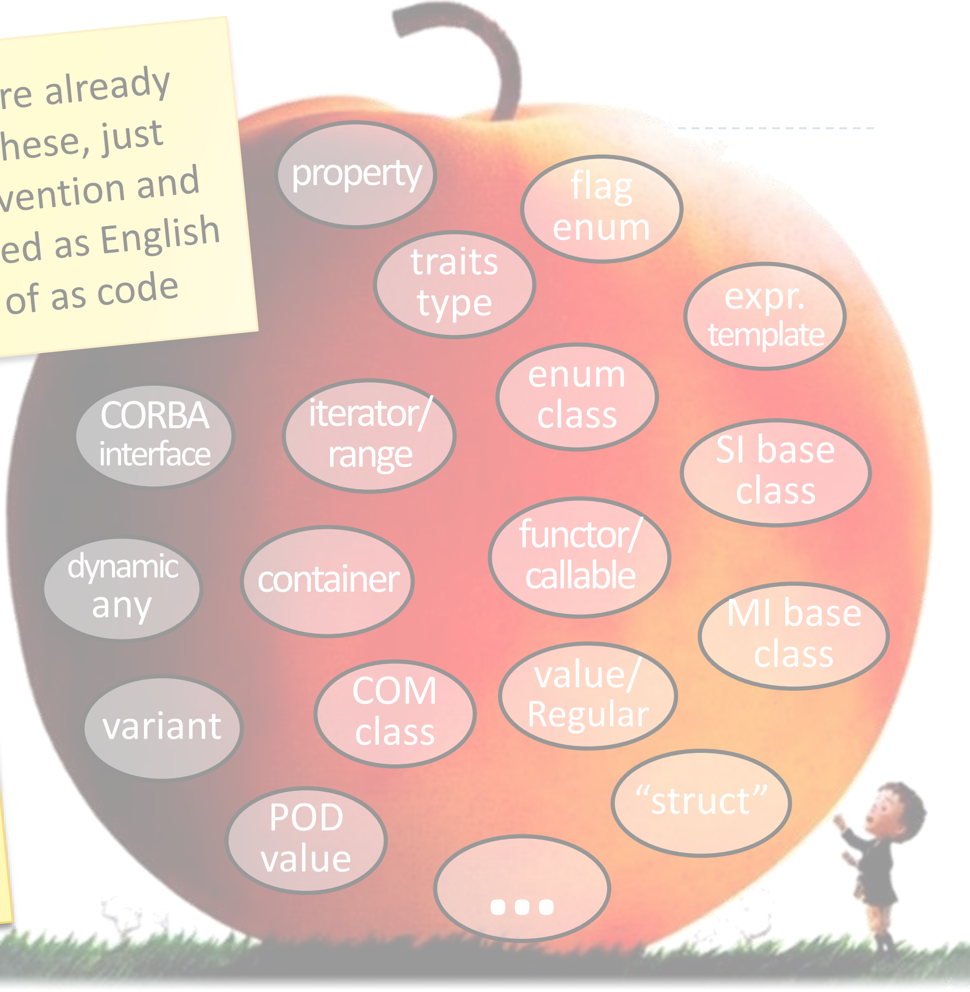
note: we're already writing these, just
(a) by convention and
(b) described as English instead of as code

Metaclasses goal in a nutshell:

to **name a subset** of the universe of classes having **common characteristics**,

express that subset using **compile-time code**, and

make **classes easier to write** by letting class authors **use the name as a generalized opt-in** to get those characteristics.



property

flag enum

traits type

expr. template

CORBA interface

iterator/ range

enum class

SI base class

dynamic any

container

functor/ callable

MI base class

variant

COM class

value/ Regular

"struct"

POD value

…

The language at work

Source code

```
class Point {
  int x, y;
};
```

```
struct MyClass : Base {
  void f() { /*...*/ }
  // ...
};
```

Compiler

```
for (m : members)
  if (!v.has_access())
    if(is_class())
      v.make_private();
    else // is_struct()
      v.make_public();

for (f : functions) {

  if (f.is_virtual_in_base_class()
      && !f.is_virtual())
    f.make_virtual();

  if (!f.is_virtual_in_base_class()
      && f.specified_override())
    ERROR("does not override");

  if (f.is_destructor())
    if (members_dtors_noexcept())
      f.make_noexcept();

}
```

Definition

```
class Point {
private:
  int x, y;
public:
  Point() =default;
  ~Point() noexcept =default;
  Point(const Point&) =default;
  Point& operator=(const Point&) =default;
  Point(Point&&) =default;
  Point& operator=(const Point&&) =default;
};
```

```
class MyClass : public Base {
public:
  virtual void f() { /*...*/ }
  // ...
};
```

The language at work

Source code

```
class Point {
  int x, y;
};
```

```
struct MyClass : Base {
  void f() { /*...*/ }
  // ...
};
```

Compiler

Q: What if you could write your own code here, and give a name to a group of defaults & behaviors?

(treat it as ordinary code, share it as a library, etc.)

Definition

```
class Point {
private:
  int x, y;
public:
  Point() =default;
  ~Point() noexcept =default;
  Point(const Point&) =default;
  Point& operator=(const Point&) =default;
  Point(Point&&) =default;
  Point& operator=(const Point&&) =default;
};
```

```
class MyClass : public Base {
public:
  virtual void f() { /*...*/ }
  // ...
};
```

The languag

Source code

```
class Point {
  int x, y;
};
```

```
struct MyClass : Base {
  void f() { /*...*/ }
  // ...
};
```

C

could write your own code here, and give a name to a group of defaults & behaviors?

(treat it as ordinary code, share it as a library, etc.)

Definition

```
class Point {
private:
  int x, y;
public:
  Point() =default;
  ~Point() noexcept =default;
  Point(const Point&) =default;
  Point& operator=(const Point&) =default;
  Point(Point&&) =default;
  Point& operator=(const Point&&) =default;
};
```

```
cl
pu

};
```

nothing too crazy!

just participating in interpreting the meaning of definitions

not making the language grammar mutable

no grammar difference except allowing a metaclass name instead of general “class”

not making definitions mutable after the fact

no difference at all in classes, no bifurcation of the type system

Metaclasses

- **$class** denotes a metaclass.

```
namespace std::experimental {              // for illustration
  $class interface { /*...public pure virtual fns only + by default...*/ };
}
```

more specific than “class”

```
interface Shape { /*... public virtual enforced + default ...*/ };
```

- Typical uses:
  - Enforce rules (e.g., “all functions must be public and virtual”)
  - Provide defaults (e.g., “functions are public and virtual by default”)
  - Provide implicitly generated functions (e.g., “has virtual destructor by default,” “has full comparison operators and default memberwise implementations”)

interface (user code)

C++17

```
class Shape {
public:
    virtual int area() const =0;
    virtual void scale_by(double factor) =0;
    virtual ~Shape() noexcept { };

    // careful not to write a nonpublic or
    // nonvirtual function, or a copy/move
    // operation, or a data member; no
    // enforcement under maintenance
};
```

Proposed

```
interface Shape {
    int area() const;
    void scale_by(double factor);
};
```

default + enforce: all public pure virtual functions
enforce: no data members, no copy/move

interface (implementation)

```
$class interface {
   ~interface() noexcept { }
   constexpr {
      compiler.require($interface.variables().empty(),
         "interfaces may not contain data members");
      for (auto f : $interface.functions()) {
         compiler.require(!f.is_copy() && !f.is_move(),
             "interfaces may not copy or move; consider a virtual clone()");
         if (!f.has_access()) f.make_public();
         compiler.require(f.is_public(), "interface functions must be public");
         f.make_pure_virtual();
      }
   }
};
```

interface (implementation)

$class ⇒ metaclass

```
$class interface {
    ~interface() noexcept { }
    constexpr {
        compiler.require($interface.variables().empty(),
            "interfaces may
        for (auto f : $interface.functio
            compiler.require(!f.is_copy() && !f.is_move(),
                "interfaces may not copy
            if (!f.has_access()) f.make_public();
            compiler.require(f.is_public(), "interf
            f.make_pure_virtual();
        }
    }
};
```

for each function in the instantiating class

enforce constraints, integrated with compiler messages

apply defaults where not specified by the user

define a type ⇒ metaprogram runs here

```
interface Shape {
    int area() const;
    void scale_by(double factor);
    pair<int,int> get_extents() const;
};
```

interface (implementation)

```
$class interface {
   ~interface() noexcept { }
   constexpr {
      compiler.require($interface.variables().empty(),
         "interfaces may not contain data members");
                           ctions()) {
                           py() && !f.is_move(),
                           opy or move; consider a
                           ke_public();
                           lic(), "interface funct                    ;
```

Look ma, no standardese!

Define language-like features using the language itself – can read the source code to "language features" like we can read the source code to STL and other libs

Bonus: Does my spec have a bug? Unit-test and debug it as usual… it's just code

We do not have unit testing and debugging for "standardese"

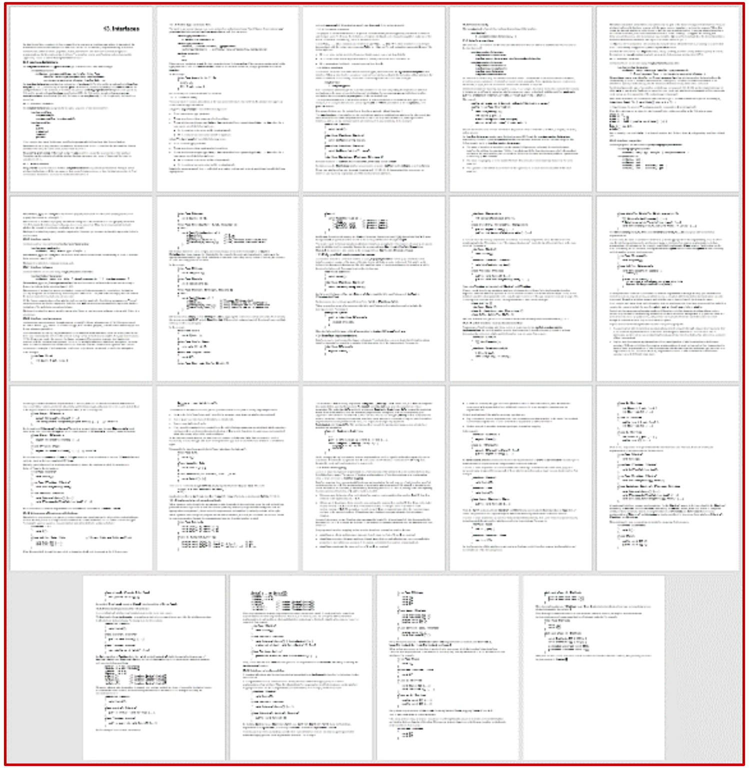
+ no loss in usability, expressiveness, diagnostics, performance, …

even compared to other languages that added this as a built-in language feature

```
interface Shape {
    int area() const;
    void scale_by(double factor);
    pair<int,int> get_extents() const;
};
```

interface (implementation)

C# language: ~18pg, English



Proposed C++: ~10 lines, testable code

```
$class interface {
   ~interface() noexcept { }
   constexpr {
      compiler.require($interface.variables().empty(),
         "interfaces may not contain data members");
      for (auto f : $interface.functions()) {
         compiler.require(!f.is_copy() && !f.is_move(),
            "interfaces may not copy or move; "
            "consider a virtual clone()");
         if (!f.has_access()) f.make_public();
         compiler.require(f.is_public(),
            "interface functions must be public");
         f.make_pure_virtual();
      }
   }
};
```

interface (user code)

C#, Java

```
interface Shape {
    int area();
    void scale_by(double factor);
    // ...
}
```

Proposed C++

```
interface Shape {
    int area() const;
    void scale_by(double factor);
    // ...
};
```

value (user code)

C++17

```
class Point {
    int x = 0, y = 0;
public:
    Point(int, int);
    // ... behavior functions ...
    Point() = default;
    friend bool operator==(const Point& a, const Point& b)
        { return a.x == b.x && a.y == b.y; }
    friend bool operator!=(const Point& a, const Point& b)
        { return !(a == b); }
    friend bool operator< (const Point& a, const Point& b)
        { return a.x < b.x || (a.x == b.x && a.y < b.y); }
    friend bool operator> (const Point& a, const Point& b)
        { return b < a; }
    friend bool operator>=(const Point& a, const Point& b)
        { return !(a < b); }
    friend bool operator<=(const Point& a, const Point& b)
        { return !(b < a); }
};
```

Proposed

```
value Point {
    int x = 0, y = 0;
    Point(int, int);
    // ... behavior functions ...
};
```

default + enforce: copy/move, comparisons, default ctor

default (opt): private data, public functions

enforce: no virtual functions

value (implementation)

```
$class basic_value {
   basic_value()                                   = default;
   basic_value(const basic_value& that)            = default;
   basic_value(basic_value&& that)                 = default;
   basic_value& operator=(const basic_value& that) = default;
   basic_value& operator=(basic_value&& that)      = default;

   constexpr {
      for (auto f : $basic_value.variables())
         if (!f.has_access()) f.make_private();

      for (auto f : $basic_value.functions()) {
         if (!f.has_access()) f.make_public();
         compiler.require(!f.is_protected(), "a value type may not have a protected function");
         compiler.require(!f.is_virtual(),   "a value type may not have a virtual function");
         compiler.require(!f.is_destructor() || f.is_public(), "a value destructor must be public");
      }
   }
};

$class value : basic_value, ordered { };
```

value (imple

```
value Point {
    int x = 0, y = 0;
    Point(int, int);
};

Point p(50, 100), p2;   // ok, default constructible
p2 = get_some_point();  // ok, copyable
if (p == p2) { /*…*/ }  // ok, == available
set<Point> s;           // ok, < available
```

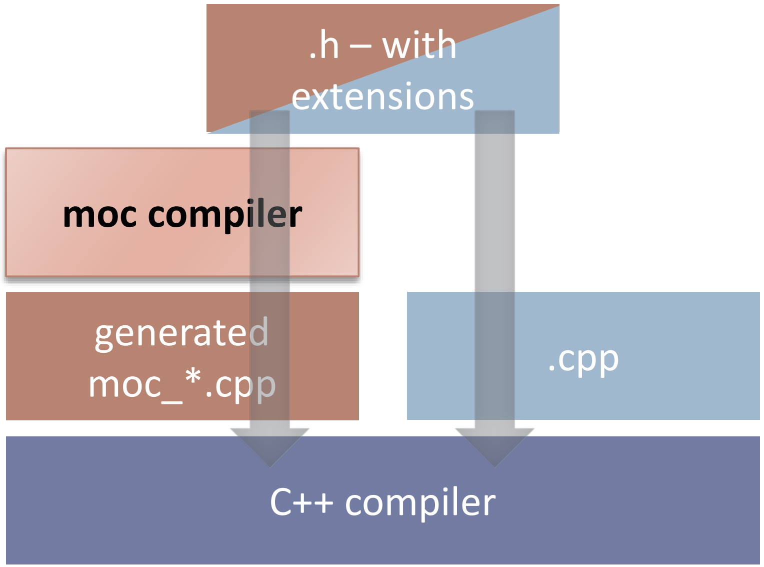
```
$class basic_value {
   basic_value()
   basic_value(const basic_v
   basic_value(basic_value&&
   basic_value& operator=(co
   basic_value& operator=(ba
   constexpr {
      for (auto f : $basic_v
         if (!f.has_access()
      for (auto f : $basic_value.functions()) {
         if (!f.has_access()) f.make_public();
         compiler.require(!f.is_protected(), "a value type may not have a protected function");
         compiler.require(!f.is_virtual(),   "a value type may not have a virtual function");
         compiler.require(!f.is_destructor() || f.is_public(), "a value destructor must be public");
      }
   }
};
```

ordered provides <, >, <=, >=, ==, !=

```
$class value : basic_value, ordered { };
```

When you can’t express it all in C++ code

Qt moc



.h – with extensions

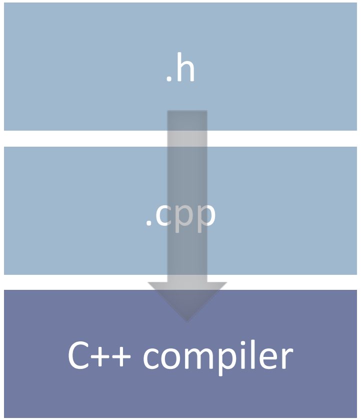
moc compiler

generated moc_*.cpp

.cpp

C++ compiler

Proposed



.h

.cpp

C++ compiler

podio (particle physics data models, Benedikt Hegner)

Today (separate YAML script)

```
ExampleHit :
  Description : "Example Hit"
  Author : "B. Hegner"
  Members:
   - double x      // x-coordinate
   - double y      // y-coordinate
   - double z      // z-coordinate
   - double energy // measured
```

generate: 5 interrelated classes…
X, XCollection, XConst, XData, XObj
how: separate code generator

podio (particle physics data models, Benedikt Hegner)

Today (separate YAML script)

```
ExampleHit :
  Description : "Example Hit"
  Author : "B. Hegner"
  Members:
   - double x      // x-coordinate
   - double y      // y-coordinate
   - double z      // z-coordinate
   - double energy // measured
```

generate: 5 interrelated classes…
X, XCollection, XConst, XData, XObj
how: separate code generator

Proposed C++ (strawman)

```
podio::datatype ExampleHit {
  string Description = "Example Hit";
  string Author = "B. Hegner";

  double x;       // x-coordinate
  double y;       // y-coordinate
  double z;       // z-coordinate
  double energy; // measured
};
```

default + enforce: constexpr static strings
generate: same 5 classes
how: during normal C++ compilation

Goals

- Expand C++'s abstraction vocabulary beyond class/struct/union/enum
- Enable writing compiler-enforced coding standards, hardware interface patterns, etc.
- Enable writing "language extensions" as library code, with equal usability & efficiency
  - Incl. valuable extensions we'd never standardize in the language because they're too narrow (e.g., interface)
- Eliminate the need for side languages & compilers (e.g., Qt moc, COM IDL/MIDL, C++/CX)

Benefits for users

Don't have to wait for a new compiler

Can share "new language features" as libraries

Can even add productivity features themselves

Benefits for standardization

More features as libraries ⇒ easier evolution

Testable code ⇒ higher-quality proposals

Benefits for C++ implementations

< new language features ⇒ < compiler work

Can deprecate and remove classes of extensions

Meta: Toward generative C++

Goal: Making C++ more powerful, <u>and</u> simpler

Questions?