accu 2013

C++ now 2013

Dynamic C++

POCO
C++ PORTABLE COMPONENTS

alex@pocoproject.org

Dynamic C++ ? What ???

- this is NOT about C++ as a dynamic language
- this IS about *dynamic-language-like* C++ solutions for interfacing
  - *diverse data sources*
  - *dynamic language environments*

Presentation Content

- The Problem
- The Solutions
  - boost::any
  - boost::variant
  - boost::type_erasure
  - folly::dynamic
  - Poco::Dynamic::Var
- Comparisons / evaluations
- Conclusion

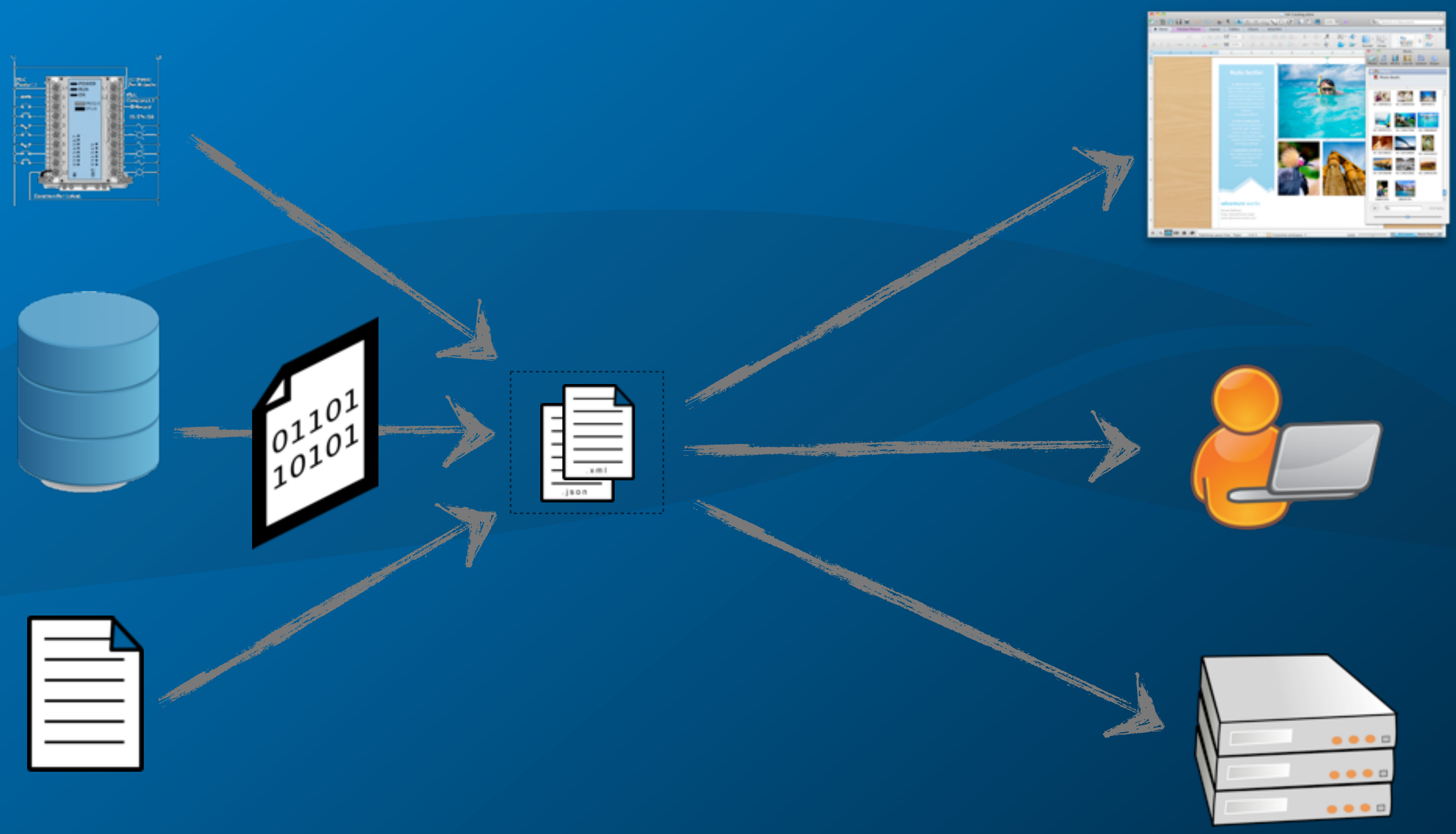
"Without a good library, most interesting tasks are hard to do in C++; but given a good library, almost any task can be made easy."

-- Bjarne Stroustrup

"As to which is more important, Dynamic or static, both are absolutely essential, even when they are in conflict."

-- Robert Pirsig, *Metaphysics of Quality*

The Problem



Dynamic C++ Concerns

- storing value
- performing operations (mostly conversions)
- retrieving value
- runtime performance
  - speed
  - memory usage
- code size
- ease of use

Dynamic Data Storage Strategies

- heap[1] (void* + new)
  - allocation overhead
  - memory cleanup
- stack[1] (union[2] + placement new)
  - size
  - alignment
  - ~destruction
- hybrid (a.k.a. *small object optimization*)
  - runtime detection performance penalty

---

[1] Commonly used nomenclature - internal/external would be more appropriate
[2] Usually raw storage and alignment (plus pointer for SOO)

Dynamic Data Operations Support

- type conversions
  - static[1]
  - dynamic
- standard language operations (+, -, ==, ...)
- custom operations

[1] Delegated to compiler, with runtime narrowing check

Dynamic Recipe Ingredients

- (placement) new
- void*
- union
- virtual functions
- templates

boost::any

```
any a;
any b(42);
any c("42"); // no support for arrays
any c(std::string("42")); // ok, object
any d;
d = c;
std::cout << any_cast<std::string>(d);
```

boost::any

- a container for values of any type
- does not attempt conversion between types
- accommodates any type in a single container
- generic solution for the first half of the problem
- "syntactic sugar" - template without template syntax

```
class any
{
 public:
  template<typename T>
  any(const T& value):content(new holder<T>(value)) { }
};
```

boost::any - under the hood

```
class any {
  template<typename T>
  any(const T& value):content(new holder<T>(value)){ }
  // ...
  struct placeholder
  {
    virtual ~placeholder(){ }
    // ...
    virtual const std::type_info & type() const = 0;
    // ...
  };

  template<typename T>
  struct holder : public placeholder
  {
    holder(const T& value):held(value){ }
    // ...
    T held;
  };
};
```

boost::any - use case

```
using boost::any;
using boost::any_cast;
using std::string;

std::list<any> values;

short ival = 42;
string sval = "fourty two";

values.push_back(ival);
values.push_back(sval);

string strval = values[0]; // oops!, compile error
strval = any_cast<string>(values[0]); // still oops!, throw

strval = any_cast<string>(values[1]); // OK
short itval = any_cast<short>(values[0]); // OK
int itval = any_cast<int>(values[0]); // throw
```

dynamic on receiving but *static* on the giving end

boost::any - summary

- generic container for values of different types
- simple to understand and use
- useful when *sender* and *receiver* (or *caller* and *callee*) know *exactly* what to expect but the "middle man" does not
- dynamic receiving
- static giving (stricter than language)
- heap alloc overhead
- virtual overhead

boost::variant

- *"safe, generic, stack-based discriminated union container, offering a simple solution for manipulating an object from a heterogeneous set of types in a uniform manner"*
- can hold any type specified at compile time
- default construction (first type must be default-constructible)

```
boost::variant<int, std::string> v;

v = "hello";

std::cout << v << std::endl; // "hello"

std::string& str = boost::get<std::string>(v);

str += " world! "; // "hello world!"
```

boost::variant - conversion

- on the extraction side, very similar (static) to any
- programmer must keep in mind:
  - what types are held ?
  - what is the type currently held ?

```
boost::variant<short, std::string> v(123);
std::string s = boost::get<std::string>(v); // throws
int i = v; // compile error
int i = boost::get<int>(v); // throws!
int i = boost::get<short>(v); // OK
```

boost::variant visitors welcome

> supports compile-time checked visitation

```
// conversion via visitation
struct str_int_converter : public static_visitor<int>
{
    int operator()(int i) const
    {
        return i;
    }

    int operator()(const std::string & str) const
    {
        return NumberParser::parse(str);
    }
};

variant<int, std::string> u("123");
int i = apply_visitor(str_int_converter(), u); // i == 123
```

boost::variant visitors welcome

- modify value generically

```
// modification via visitation
struct doubling_converter : public static_visitor<int>
{

  template <typename T>
  void operator()( T & operand ) const
  {
    operand += operand;
  }

};


variant<int, double, std::string> u("123");
apply_visitor(doubling_converter(), u); // u == "123123"
```

boost::variant - default construction

```
variant()
{
  new( storage_.address() ) internal_T0();
  indicate_which(0); // index of the first bounded type
}
```

boost::variant - internals (construction)

```
template <typename T>
variant(const T& operand)
{
   convert_construct(operand, 1L);
}
```

```
template <typename T> void convert_construct(
          T& operand
        , int
        , mpl::false_ = mpl::false_() //true for another variant
        )
{
  indicate_which(initializer::initialize(
    storage_.address(), operand));
}
```

boost::variant - internals (storage)

```
// active type indicator
which_t which_; // int or smaller, stack or heap

// storage itself:
typedef typename detail::variant::make_storage<
          internal_types, never_uses_backup_flag
        >::type storage_t;

storage_t storage_;
```

boost::variant - internals (storage)

```
template <std::size_t size_, std::size_t alignment_>
struct aligned_storage_imp
{
  union data_t
  {
    char buf[size_];

    typename mpl::eval_if_c<alignment_ == std::size_t(-1)
      , mpl::identity<detail::max_align>
      , type_with_alignment<alignment_>
    >::type align_;
  } data_;

  void* address() const
  {
     return const_cast<aligned_storage_imp*>(this);
  }
};
```

boost::variant - summary

- strongly typed
- more complex than any
- default storage on stack (can be configured to allocate heap)
- no built-in data conversion
- implementation complexity due to *never-empty* guarantee
- visitors perform reasonably fast (no virtual calls)
- implementation employs *"all known means to minimize the size of variant objects; often unsuccessful due to alignment issues, and potentially harmful to runtime speed, so not enabled by default"*

variant or any ?

```
// if you find yourself doing this:

if (a.type() == typeid(int))
  // ...
else if (a.type() == typeid(string))

// then use boost::variant with visitor
```

boost::type_erasure

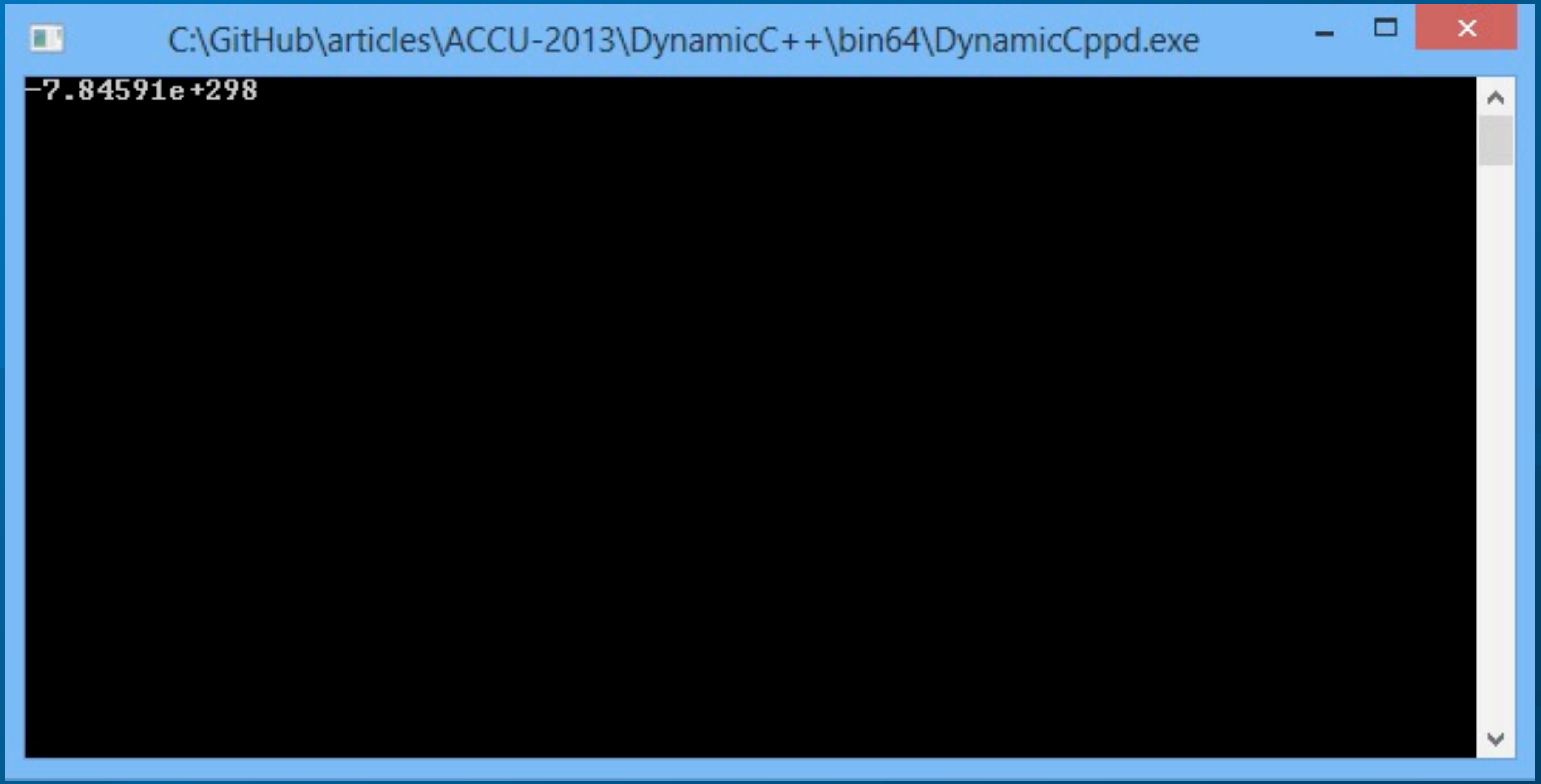
- a generalization of boost::any and boost::function
- name confusing (proposed alternatives: *Staged Typing*, *Deferred Typing*, *Type Interface*, *Interface*, *Static Interface*, *Value Interface*, *Structural Typing*, *Duck Typing*, and *Run-Time Concepts*)
- addresses the limitations of virtual functions:
  - intrusiveness
  - dynamic memory management
  - limited ability to apply multiple independent concepts to a single object

```
any<mpl::vector<copy_constructible<>, typeid_<> >> x(10);
int i = any_cast<int>(x); // i == 10
```

boost::type_erasure - adding ops to any

```
any<
    mpl::vector<copy_constructible<>,
        typeid_<>,
        incrementable<>,
        ostreamable<>
    >
> x(10);
++x;
std::cout << x << std::endl; // prints 11

// ...

typedef any<boost::mpl::vector<copy_constructible<>,
        typeid_<>, addable<>, ostreamable<>>
> any_type;
any_type x(1.1);
any_type y(1);
any_type z(x + y);
std::cout << z << std::endl; // prints ???
```

boost::type_erasure - adding oops! to any



- the underlying types of the arguments of + must match *exactly* or the behavior is *undefined*
- it is possible to get an exception by adding `relaxed_match` concept

boost::type_erasure - placeholders

- let’s try to do that again ...
- capture relationships among different types through *placeholders*

```
double d = 1.1;
int i = 1;

typedef boost::mpl::vector<
    copy_constructible<_a>,
    copy_constructible<_b>,
    typeid_<_a>,
    addable<_a, _b, _a>
> requirements;

tuple<requirements, _a, _b> t(d, i);
any<requirements, _a> x(get<0>(t) + get<1>(t));
std::cout << any_cast<double>(x) << std::endl; // bingo! 2.1
std::cout << any_cast<int>(x) << std::endl; // throws!
```

boost::type_erasure - valued conversion

```
int i = 123;

any<...> s(i);

std::string str = s.to_string();
```

boost::type_erasure - conversion scaffolding

```
template<class F, class T>
struct to_string
{
    // conversion function
    static T apply(const F& from, T& to)
    {
        return to = NumberFormatter::format(from);
    }
};

namespace boost {
namespace type_erasure {
    template<class F, class T, class Base> // binding
    struct concept_interface<::to_string<F, T>, Base, F> : Base
    {
        typedef typename rebind_any<Base, T>::type IntType;
        T to_string(IntType arg = IntType())
        {
            return call(::to_string<C, T>(), *this, arg);
        }
    };
}
```

boost::type_erasure - valued conversion

```
typedef any<to_string<_self, std::string>, _self&> stringable;

int i = 123;

stringable s(i);

std::string str = s.to_string();
```

boost::type_erasure - internals (storage)

```
// storage
struct storage
{
    storage() {}
    template<class T>
    storage(const T& arg) : data(new T(arg)) {}
    void* data;
};

// binding of concept to actual type
typedef ::boost::type_erasure::binding<Concept> table_type;

// actual storage
::boost::type_erasure::detail::storage data;
                              table_type table;
```

boost::type_erasure - summary

- extends any/variant by introducing a mechanism for “attaching” operations to types at compile time
- addresses the limitations of virtual functions
- uses heap allocation to store values
- complex framework
- slippery scenarios can easily take a naïve user into UB-land
- perhaps another layer of abstraction would help?

Facebook folly::dynamic

- *"runtime dynamically typed value for C++, similar to the way languages with runtime type systems work"*
- holds types from predefined set of types
- supports "objects" and "arrays" (JSON was the motivation)
- union-based, implementation reminiscent of boost::variant
- runtime operation checking
- linux-only (Ubuntu/Fedora, and even there build is not easy!)

```
dynamic twelve = 12; // dynamic holding an int
dynamic str = "string"; // FBString
dynamic nul = nullptr;
dynamic boolean = false;
```

folly::dynamic - usage

- clean, intuitive and reasonably predictable interface

```
std::string str("123");
dynamic var(str);
int i = var.asInt(); // 123
double d = var.asDouble(); // 123.0
var = i; // 123
std::string s = var.asString().c_str(); // FBString!
var = d; // 123
s = var.asString().c_str(); // "123.0"
```

folly::dynamic - storage

- somewhat similar to boost::variant

```
union Data
{
  explicit Data() : nul(nullptr) {}

  void* nul;
  Array array;
  bool boolean;
  double doubl;
  int64_t integer;
  fbstring string;

  typename std::aligned_storage<
    sizeof(std::unordered_map<int,int>),
    alignof(std::unordered_map<int,int>)
  >::type objectBuffer;
} u_;
```

folly::dynamic - get stored type

```
template<class T>
T dynamic::asImpl() const
{
  switch (type())
  {
  case INT64:    return to<T>(*get_nothrow<int64_t>());
  case DOUBLE:   return to<T>(*get_nothrow<double>());
  case BOOL:     return to<T>(*get_nothrow<bool>());
  case STRING:   return to<T>(*get_nothrow<fbstring>());
  default:
    throw TypeError("int/double/bool/string", type());
  }
}

inline fbstring dynamic::asString() const
{
  return asImpl<fbstring>();
}
```

folly::dynamic - get stored type

```
template<> struct dynamic::GetAddrImpl<bool> {
  static bool* get(Data& d) { return &d.boolean; }
};

template<> struct dynamic::GetAddrImpl<int64_t> {
  static int64_t* get(Data& d) { return &d.integer; }
};

template<class T>
T* dynamic::getAddress() {
  return GetAddrImpl<T>::get(u_);
}

template<class T>
T* dynamic::get_nothrow() {
  if (type_ != TypeInfo<T>::type) {
    return nullptr;
  }
  return getAddress<T>();
}
```

folly::dynamic - conversion

- to<>() functions in “folly/Conv.h”
- written by Andrei Alexandrescu
- using V8 double-conversion

folly::dynamic - arrays and objects

- iteration

```
dynamic array = {2, 3, "foo"};

for (auto& val : array)
{
  doSomethingWith(val);
}
```

- objects

```
dynamic obj = dynamic::object(2, 3)("hello", "world")("x", 4);

for (auto& pair : obj.items())
{
  processKey(pair.first);
  processValue(pair.second);
}
```

folly::dynamic summary

- built around Facebook's JSON needs
- uses C++11 and boost extensively
- performs very well (excellent design/performance balance)
- good example of user-friendly interface
- not portable (optimized for Linux/g++)
- holding only predefined types
- hard to build (build system rudimentary, many small dependencies and outdated documentation)

Poco::Dynamic::Var (ex DynamicAny)

- boost::any + value conversion
- aims to be a general-purpose dynamic typing facility
- balance between perfomance and ease of use
- transparent conversion (except *to* std::string)
- conversion is checked (e.g. no signedness loss or narrowing conversion loss of precision)
- any type, extensible for UDT through VarHolder specialization
- optional small object optimization (W.I.P.)

Poco::Dynamic::Var - under the hood

```
namespace Poco {
namespace Dynamic {

class Var
{
public:
    // ...
    template <typename T>
    Var(const T& val):
        _pHolder(new VarHolderImpl<T>(val))
    {
    }

    // ...
private:
    VarHolder* _pHolder;
};

} }
```

* Design based on boost::any

Poco::Dynamic::Var - under the hood

```
namespace Poco {
namespace Dynamic {

class VarHolder
{
public:
    virtual ~VarHolder();
    virtual void convert(int& val) const;
    // ...
protected:
    VarHolder();
    // ...
};

template <typename T> // for end-user extensions
class VarHolderImpl: public VarHolder
{
    //...
};

template <> // native and frequent types specializations courtesy of POCO
class VarHolderImpl<int>: public VarHolder
{
    //...
};

//...
}}
```

Poco::Dynamic::Var - checked narrowing

```
template <typename F, typename T>
void convertToSmallerUnsigned(const F& from, T& to) const
    /// This function is meant for converting unsigned integral data types,
    /// from larger to smaller type. Since lower limit is always 0 for unsigned
    /// types, only the upper limit is checked, thus saving some cycles
    /// compared to the signed version of the function. If the value to be
    /// converted does not exceed the maximum value for the target type,
    /// the conversion is performed.
{
    poco_static_assert (std::numeric_limits<F>::is_specialized);
    poco_static_assert (std::numeric_limits<T>::is_specialized);
    poco_static_assert (!std::numeric_limits<F>::is_signed);
    poco_static_assert (!std::numeric_limits<T>::is_signed);

    checkUpperLimit<F,T>(from);
    to = static_cast<T>(from);
}
```

Poco::Dynamic::Var - to/from string

```
template <>
class VarHolderImpl<std::string>: public VarHolder
{
public:
// ...
    void convert(Int16& val) const
    {
        int v = NumberParser::parse(_val); // uses V8 double-conversion
        convertToSmaller(v, val);
    }
};

template <>
class VarHolderImpl<double>: public VarHolder
{
public:
//...
    void convert(std::string& val) const
    {
        val = NumberFormatter::format(_val);
    }
};
```

Poco::Dynamic::Var - to/from number

```
template <>
class VarHolderImpl<Int16>: public VarHolder
{
public:
// ...
    void convert(Int32& val) const
    {
        convertToSmaller(_val, val);
    }
};

template <>
class VarHolderImpl<UInt32>: public VarHolder
{
public:
//...
    void convert(Int32& val) const
    {
        convertSignedToUnsigned(_val, val);
    }
};
```

Poco::Dynamic::Var - emergency EXIT

```
class Var
{
// ...
    template <typename T>
    const T& extract() const
        /// Returns a const reference to the actual value.
        /// Must be instantiated with the exact type of
        /// the stored value, otherwise a BadCastException is thrown.
        /// Throws InvalidAccessException if Var is empty.
    {
        VarHolder* pHolder = content();

        if (pHolder && pHolder->type() == typeid(T))
        {
            VarHolderImpl<T>* pHolderImpl = static_cast<VarHolderImpl<T>*>(pHolder);
            return pHolderImpl->value();
        }
        else if (!pHolder)
            throw InvalidAccessException("Can not extract empty value.");
        else
            throw BadCastException(format("Can not convert %s to %s.",
                pHolder->type().name(),
                typeid(T).name()));
    }
// ...
};
```

Var in Practical Use

```
std::string str("42");
Var v1 = str; // "42"
int i = v1 // 42
v1 = i; // 42
++v1; // 43
double d = v1; // 43.0
Var v2 = d + 1.0; // 44.0
float f = v2 + 1; // 45.0
```

```
DynamicStruct aStruct;
aStruct["First Name"] = "Bart";
aStruct["Last Name"] = "Simpson";
aStruct["Age"] = 10;
Var a1(aStruct);
std::string res = a1.toString(); // no implicit conversion :-(
// { "Age": 10, "First Name": "Bart", "Last Name" : "Simpson" }
```

```
Dynamic::Struct<int> aStruct;
aStruct[0] = "First";
aStruct[1] = "Second";
aStruct[2] = 3;
std::string res = aStruct.toString();//{ "0" : "First", "1" : "Second", "2" : 3 }
```

```
std::string s1("string");
Poco::Int8 s2(23);
std::vector<Var> s3;
s3.push_back(s1);
s3.push_back(s2);
Var a1(s3);
std::string res = a1.toString(); // ["string", 23]
```

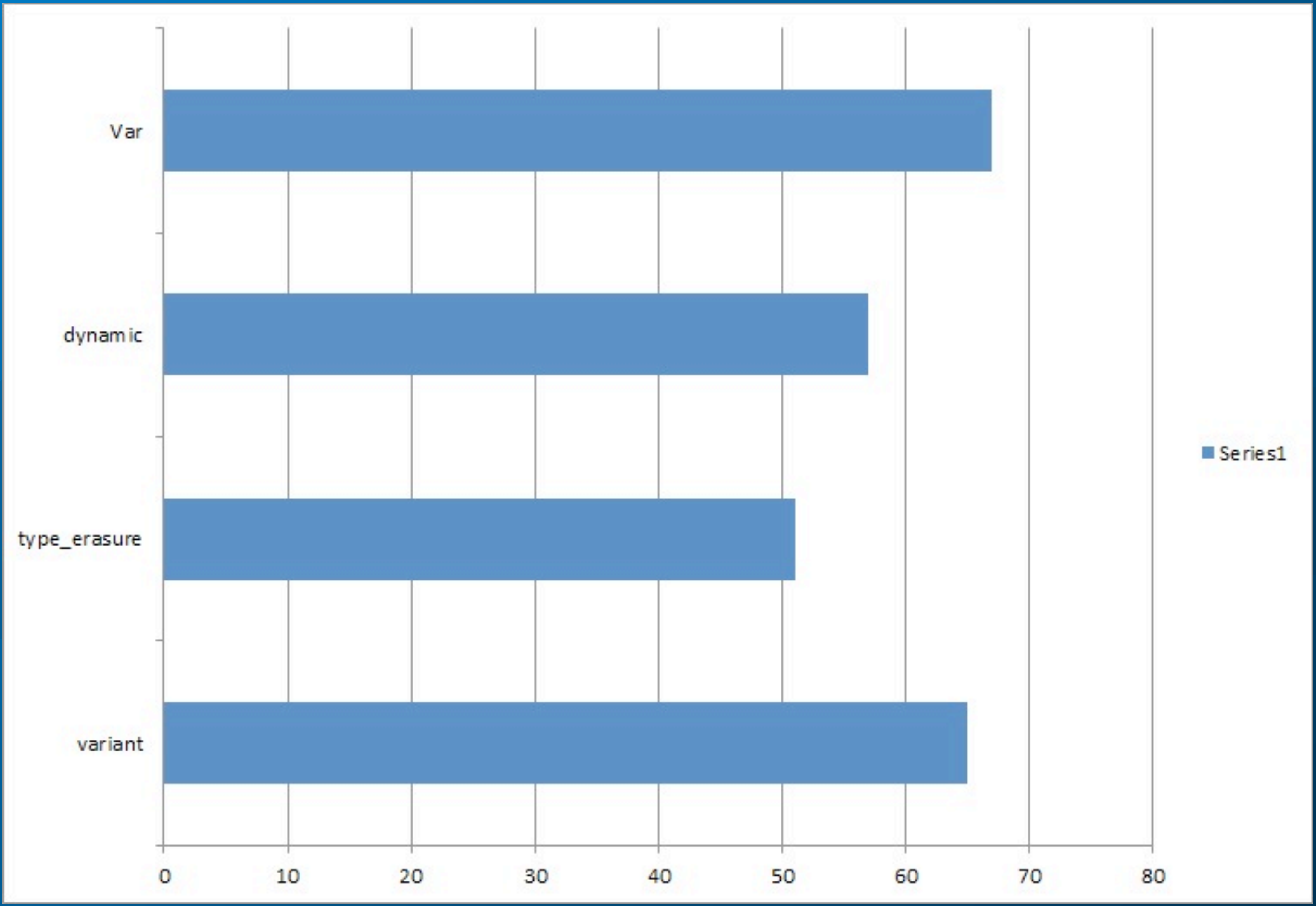
Tere's no such thing as free lunch ...

```
Binary sizes:
============
 Linux
 -----
  5160 AnySize.o
 23668 DynamicAnySizeExtract.o

 25152 DynamicAnySizeConvert.o
  9600 lexical_cast_size.o

 Windows
 -------
  26,924 AnySize.obj
  96,028 DynamicAnySizeExtract.obj

 103,943 DynamicAnySizeConvert.obj
  84,217 lexical_cast_size.obj



Lines of code:
=============
 Any               145
 DynamicAny*     3,588
 lexical_cast      971
```

Benchmarks

```
std::string s

Var var(s);

int i = var;

double d = var;

var = i;

s = var;

var = d;

s = var.toString();
```

Benchmark results

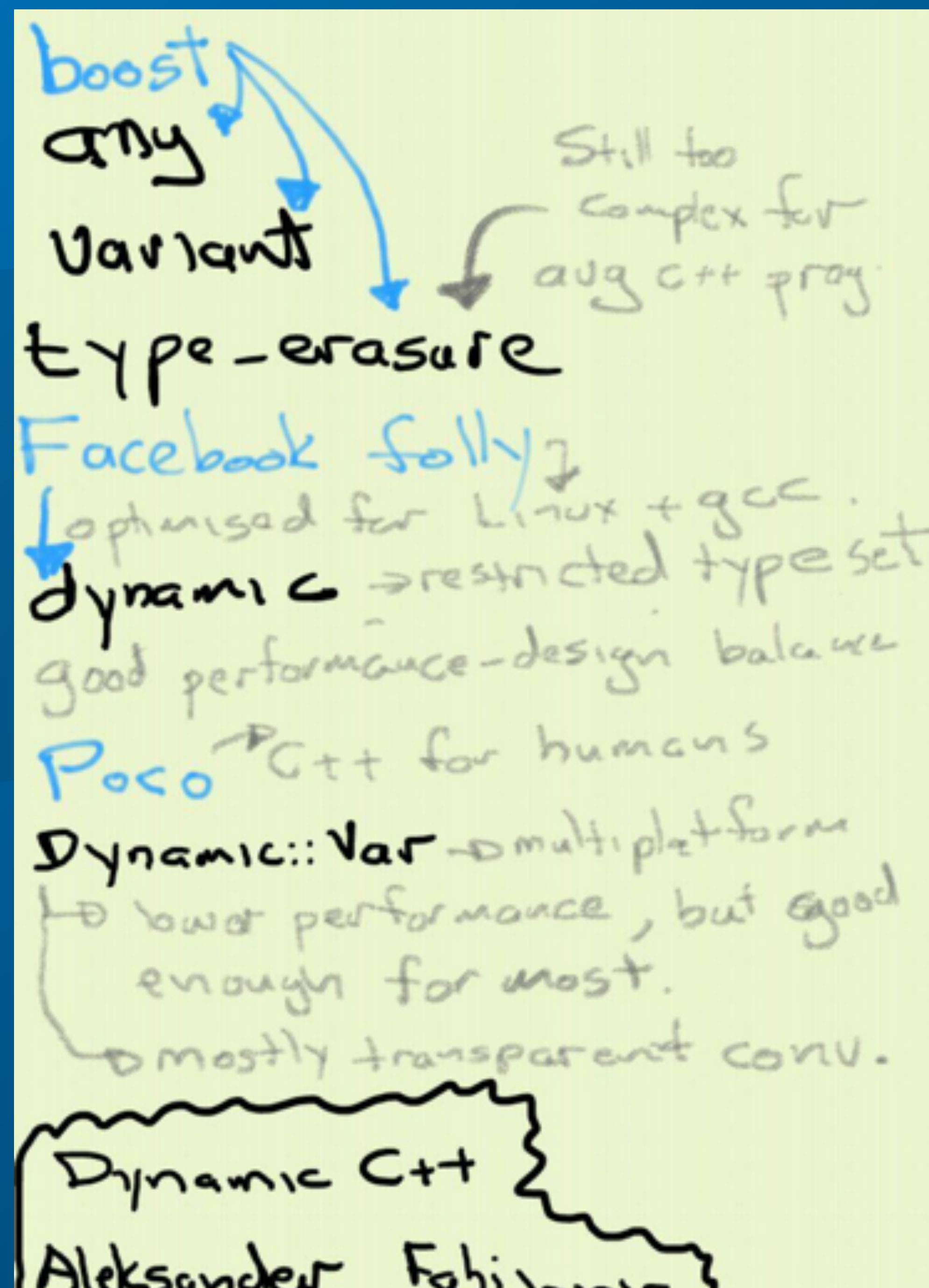


Features Comparison

class / feature	assignment	operations	conversion	retrieval
any	all	none	none	external
variant	predefined	external	none	external
type_erasure	all	“internal”	none	external
dynamic	predefined	internal	predefined	interface
Var	all	internal	specialized	automatic

Dynamic C++ in a nutshell

By Schalk Cronjé



C++ Reflection Time

"the biggest problem facing C++ today is the lack of a large set of de jure and de facto standard libraries"

--Herb Sutter

Dynamic C++ Reflection Time

?