Variations on Variant

CoreCPP Meetup
April 2020

roi.barkan@gmail.com

Agenda

- Introduction
  - What is std::variant, what is it good for
  - Typical uses for Variant
- Variants vs. Unions
  - Examples
  - Existing Approaches
  - Variants with Commonality
- Intrusive Variants
- Streams of Variants

Introduction

What is a variant

- Cppreference.com:
  - The class template std::variant represents a type-safe **union**.
- Boost.org:
  - The variant class template is a safe, generic, stack-based *discriminated* **union** container.
- Plain English - a **union** that knows (holds) its type.

```
union MyUnion {
    int integer;
    double real;
};
//...
void foo(const MyUnion& uni) {
    cout << uni.integer << endl;
}
```

```
using MyVariant = variant<int,double>;
//....
void bar(const MyVariant& var) {
    cout << get<int>(var);
}
```

Memory Layout

```
struct {int i; char c; double d;};
```


```
tuple<int,char,double>
```


```
union {int i; char c; double d;};
```


```
variant<int,char,double>
```

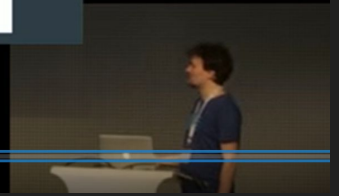
What is it Good for ?

- State Machines
- Value-Semantics for Dynamic Types
  - Commands
- Success/fail
  - expected<T>
- Exists/void
  - optional<T>
- Pattern-Matching

State Machines

```
struct Circling {
  double mTimeSinceLastShot = ENEMY_SHOT_DELAY;
  int mNextCirclePosIndex = 0;
};

struct FlyToCenter { };

struct ShootingFromCenter {
  double mTimeSinceLastShot = ENEMY_SHOT_DELAY;
  double mTimeSpentInCenter = 0;
};

struct FlyOut {
  int mTargetCornerIndex;
};

using State = std::variant<
  Circling,
  FlyToCenter,
  ShootingFromCenter,
  FlyOut>;
```

Meeting C++ 2018
Nikolai Wuttke
std::variant and the power of pattern matching



More State Machines

cppcon | 2018
THE C++ CONFERENCE • BELLEVUE, WASHINGTON

EVENTS AND STATES

EVENTS

```
struct event_connect { std::string_view address; };
struct event_connected {};
struct event_disconnect {};
struct event_timeout {};
using event = std::variant<event_connect, event_connected, event_disconnect, event_timeout>;
```

STATES

```
struct state_idle {};
struct state_connecting {
  static constexpr int n_max = 3;
  int n = 0;
  std::string address;
};
struct state_connected {};
using state = std::variant<state_idle, state_connecting, state_connected>;
```

<epam> CppCon 2018 | Effective replacement of dynamic polymorphism with std::variant 43

Effective replacement of dynamic polymorphism with std::variant

Commands

cppcon | 2016

THE C++ CONFERENCE • BELLEVUE, WASHINGTON

Variants
Past, Present,
and Future

Command

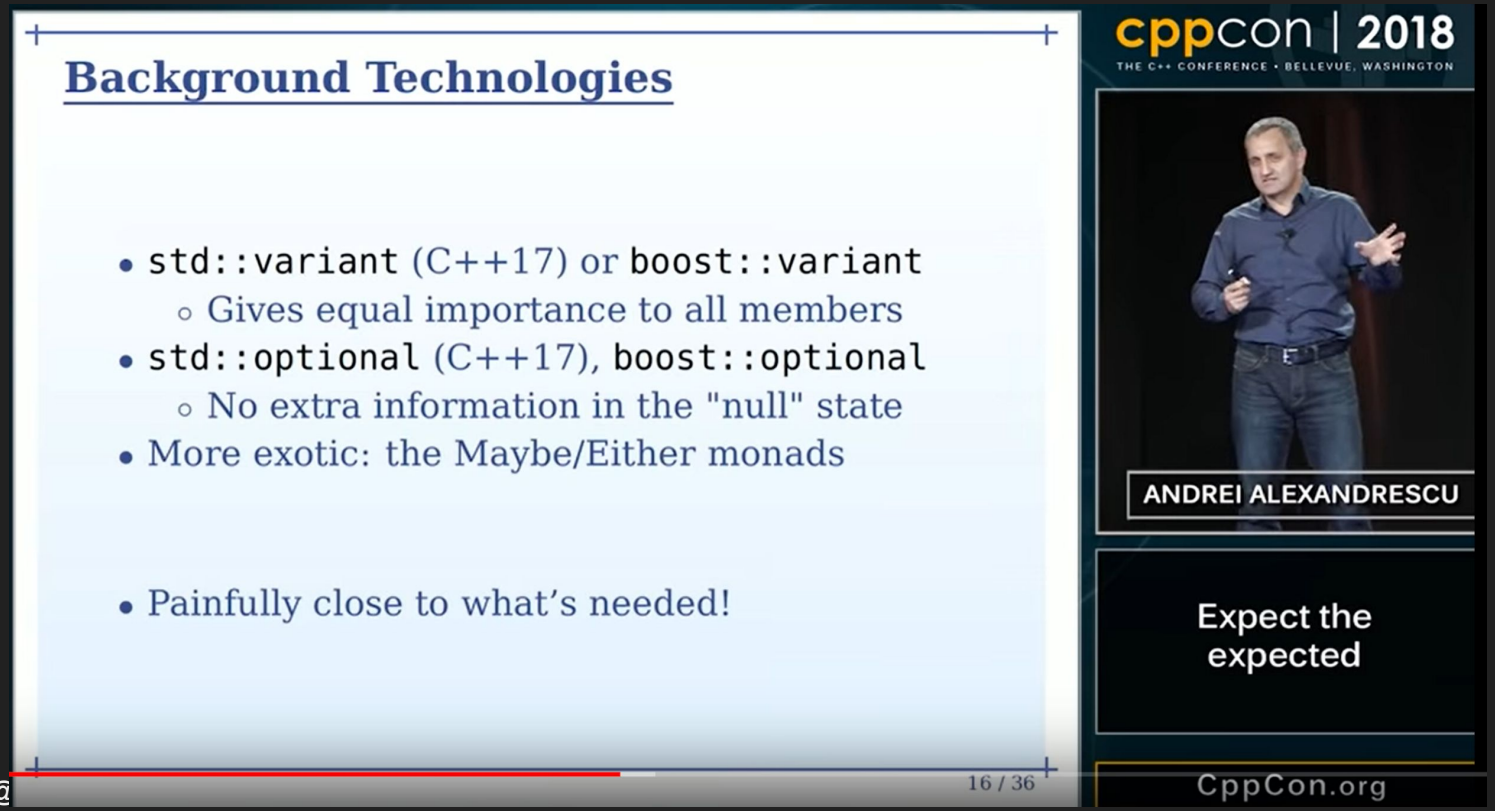
```
struct command {
  enum type { SET_SCORE, FIRE_MISSILE, FIRE_LASER, ROTATE };
  virtual type getType();
};

struct set_score : commmand {
  type getType() { return SET_SCORE; } override final;
  double value;
};
```

Now

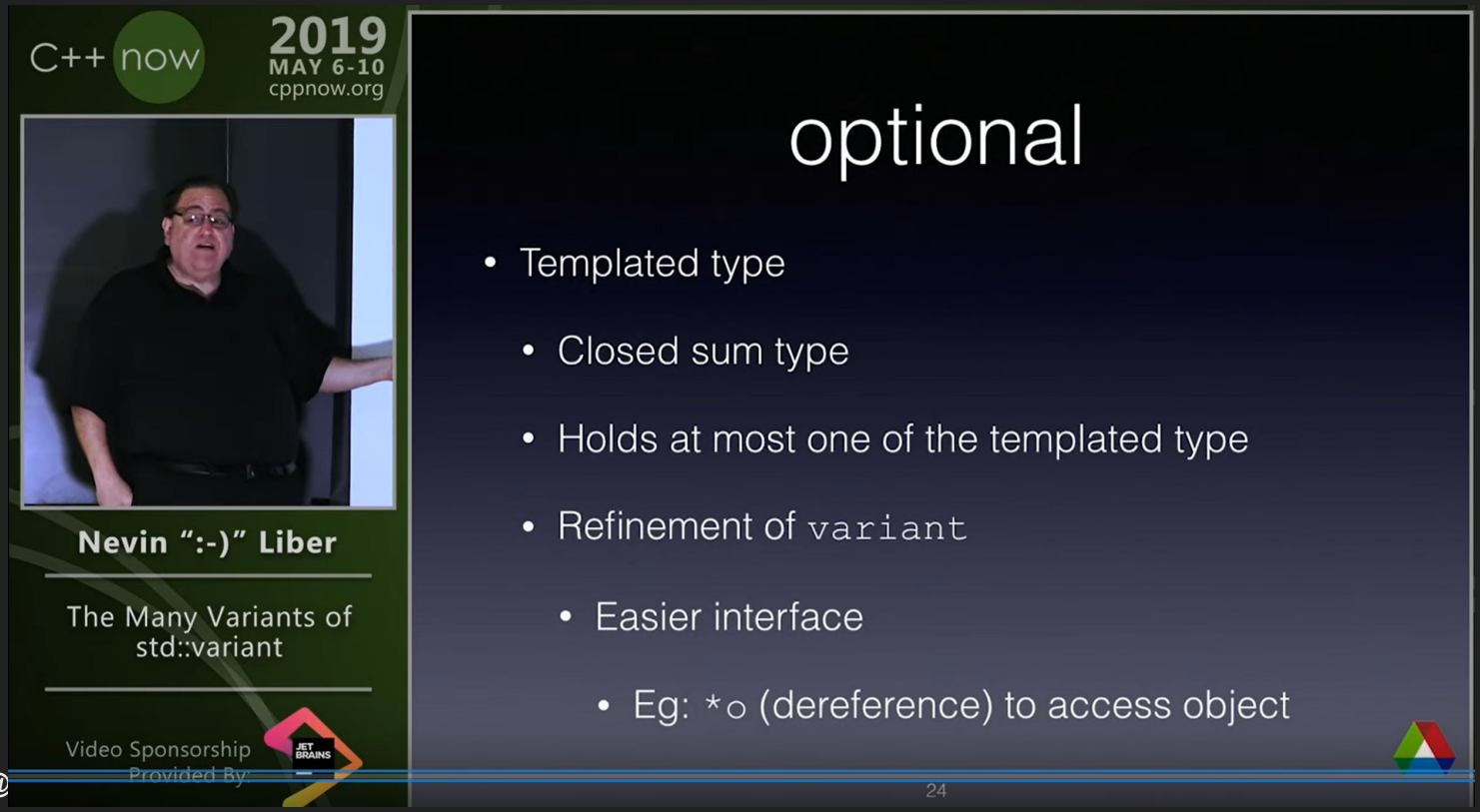
```
struct set_score {
  double value;
};
using command = variant<set_score, fire_missile, fire_laser, rotate>;
```

12 / 65

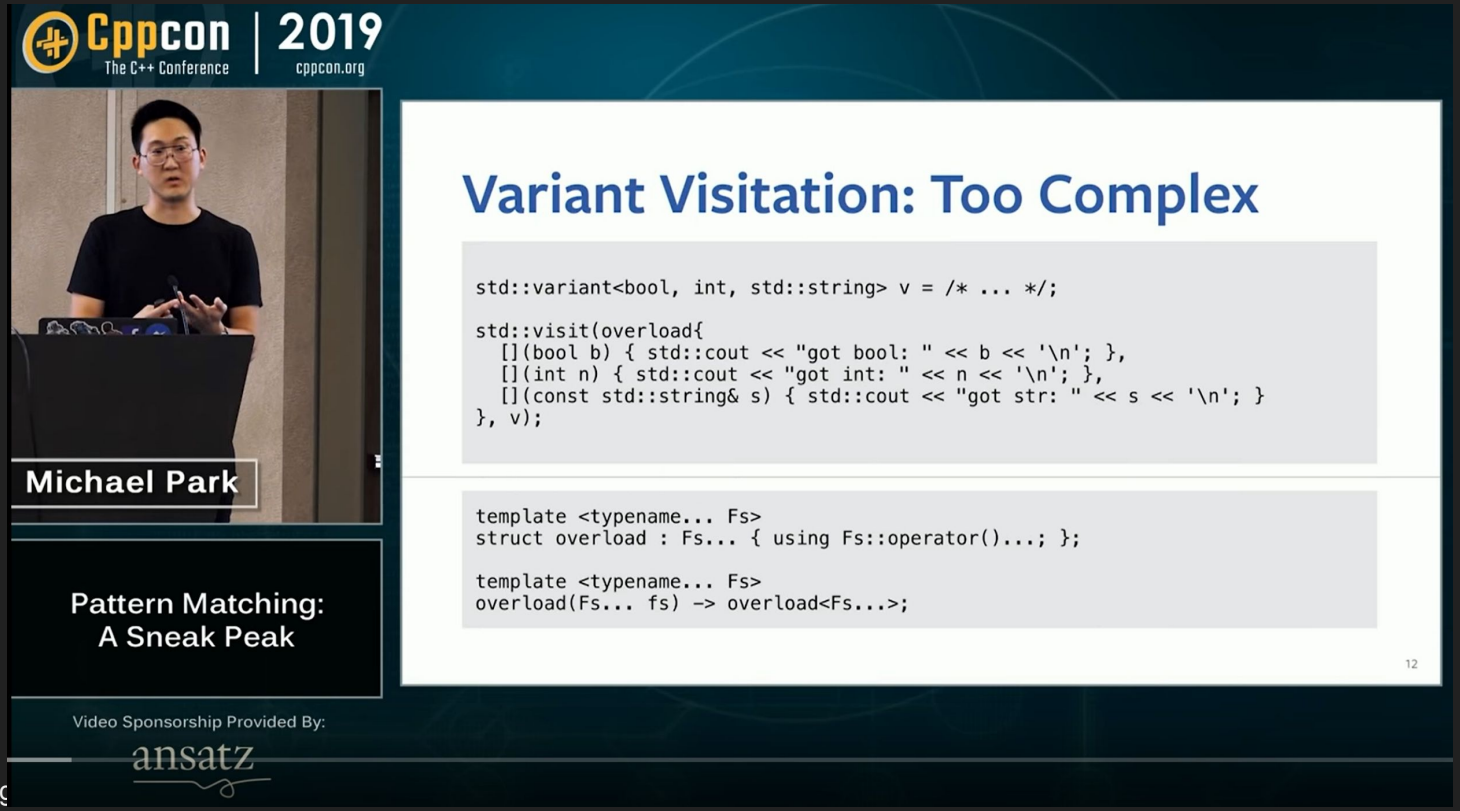
expected


Background Technologies
std::variant (C++17) or boost::variant
Gives equal importance to all members
std::optional (C++17), boost::optional
No extra information in the "null" state
More exotic: the Maybe/Either monads
Painfully close to what's needed!
16 / 36
cppcon | 2018
THE C++ CONFERENCE • BELLEVUE, WASHINGTON
ANDREI ALEXANDRESCU
Expect the expected
CppCon.org

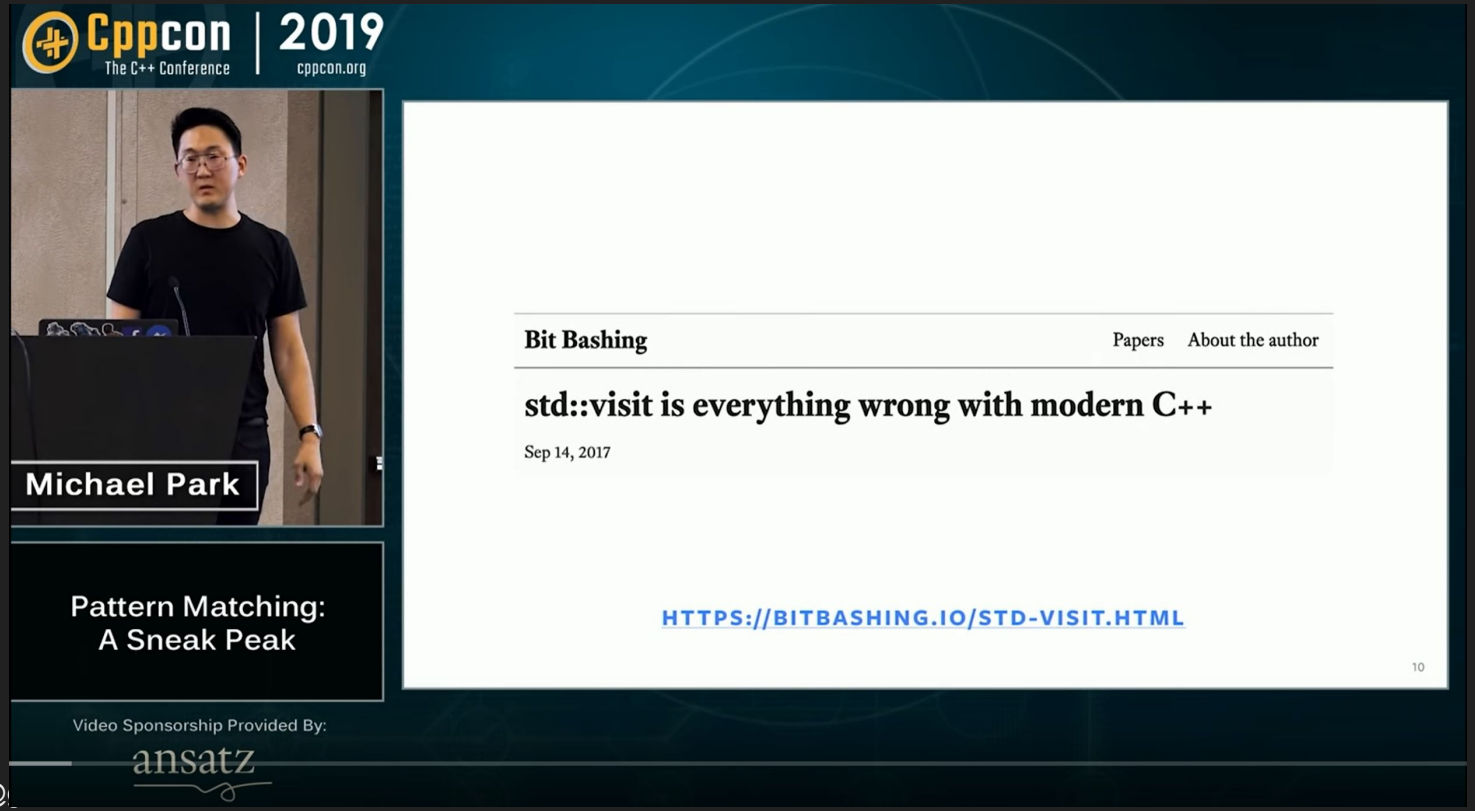
Optional


C++ now
2019
MAY 6-10
cppnow.org
Nevin ":-)" Liber
The Many Variants of
std::variant
Video Sponsorship
Provided By:
JET BRAINS
optional
• Templated type
• Closed sum type
• Holds at most one of the templated type
• Refinement of variant
• Easier interface
• Eg: *o (dereference) to access object
24

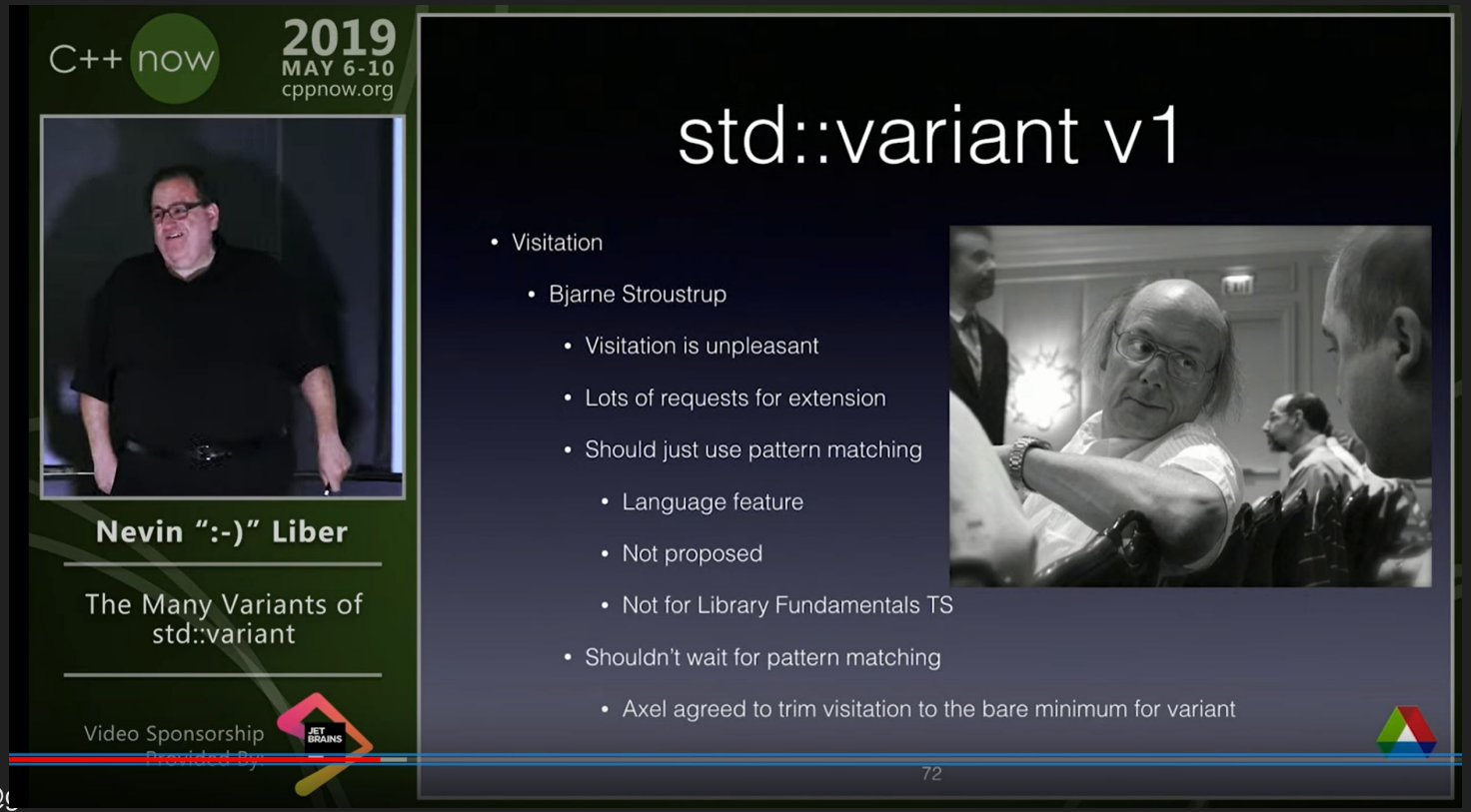
Pattern Matching



Pattern Matching?



Pattern Matching ??



Variants vs. Unions

Tag is private → variant is safe

- Only constructor and assignment-operator change the tag.
- No chance of user error
- Compiler knows all the alternatives

```
union IdentityCard {
  IDNumber id;
  PassportNumber passport;
  UUID factoryCertificate;
};
enum IDType { CITIZEN, TOURIST, ROBOT };
void checkID(IdentityCard id, IDType type) {
  switch (type) {
      case TOURIST: id.passport.check();
      case CITIZEN: id.passport.check();
  }
}
```

Bugs in this C Code:

- 😈 Access the incorrect member
- 😈 Forget missing types
- 😈 Forget to break

```
using IdentityCard = variant<IDNumber,
                             PassportNumber, UUID>;
void checkID(IdentityCard id) {
    visit([](auto& x) {x.check();},id);
}
```

How do they Actually do it in C

tag	int/char/double

- Explicit Header:

```
struct IdentityCard {
    IDType type;
    union /*value*/ {
        IDNumber id;
        PassportNumber passport;
    };
};
```

- Implicit Header:

```
union IdentityCard {
    struct Header { IDType type; };
    struct Citizen {IDType type;
        IDNumber id;
    };
    struct Tourist {IDType type;
        PassportNumber passport;
    };
};
```

- Over time header gets more data
  - Expiration date, Photo, …
- Type specific functions need access to header.
- Opportunity to use C++:

```
struct  IDHeader {
    IDType type; /* ... */
};
struct Citizen : public IDHeader {
    IDNumber id;
};
struct Tourist : public IDHeader {
    PassportNumber id;
};
union IdentityCard {
    IDHeader header;
    Citizen citizen;
    Visitor visitor;
};
```

Keeping the C Layout

- Header with type is common
  - Network Protocols - TCP/IP, Finance
  - File Formats - ELF
  - Serialization - Cap’n Proto, Apache Avro
- C layout is important
  - Compatibility with existing code
- Goal - Be safer than C, keep the layout
  - Sacrifice some safety

intrusive_variant

tag	int/char/double

```
using ID_intrusive = intrusive_variant<
    IDType, offsetof(IDHeader, type),
    std::integral_constant<IDType, IDType::CITIZEN>, Citizen,
    std::integral_constant<IDType, IDType::TOURIST>, Tourist>;
// ... Alternatively ...
using ID_intrusive = intrusive_variant<
    IDType, offsetof(IDHeader, type),
    Citizen::header_value_t, Citizen,
    Tourist::header_value_t, Tourist>;
```

```
union IdentityCard {
    struct Header { IDType type;
    };
    struct Citizen {IDType type;
        IDNumber id;
    };
    struct Tourist {IDType type;
        PassportNumber passport;
    };
};
```

- User dictates the type and location of the tag
- visit() is still O(1)
  - Potentially larger lookup table

Adding C++ Safety

- Intrusive_variant has safe visit() and links the Type with the Tag
- Still - it might mess up the offset, and allow mixing unrelated types
- Class Hierarchies can do better:
  - Base class is essentially a header.
  - Const correctness can ensure derived classes don’t change tag.
  - Use a (constexpr) lambda to get the tag of each type
- Utilities:
  - `is_base_of<>` - to make sure all types have the right base
  - decltype() to get to type and static members

variant_of_base

```
struct IDHeader {
  IDHeader(IDType type) : m_type(type) {}
  const IDType m_type;
};
struct Citizen : public IDHeader {
  Citizen() : IDHeader(c_type) {}
  static constexpr IDType c_type = CITIZEN;
  IDNumber id;
};
struct Tourist : public IDHeader {
  Tourist() : IDHeader(c_type) {}
  static constexpr IDType c_type = TOURIST;
  PassportNumber passport;
};
```

```
using ID_with_base =
    variant_of_base<IDHeader,
                    [](const IDHeader &hdr) {
                      return hdr.getType(); },
                    [](auto *x) {
                      return
      remove_reference_t<decltype(*x)>::c_type;
                    },
                    Citizen, Tourist>;
```

Arrays of Variants

- vector<variant<IDNumber, Passport, UUID>>

- vector<intrusive_variant<Citizen, Tourist, Robot>>

- C-Style (real world) - use only the RAM we need

condensed_variant utilities

- condensed_variant_iterator: const forward iterator over variants
- condensed_variant_queue: emplace_back/pop_front container
- Root of the logic is knowing how much to jump:

```
const Base *next =
    visit([](const auto *p) { return static_cast<const Base *>(p + 1); },
          myVariant);
```

  - Key point: `(p + 1)` knows the correct type.

Summary so far

- variants are different than unions
- real-world unions already have tags (and header)
- Intrusive_variant - C++ safety with high C compatibility
- Variant_of_base - add C++ to your code
- condensed_variant - real world streams of binary data

Q & A

roi.barkan@gmail.com

Bonus - Devirtualization

Blast from the Past

Virtual dispatch analysis

+ Promotes flexibility & decoupling
+ Best for large, unbounded hierarchies
+ Automatic 'load balancing' of icache

– Wastes space in the hot zone
– Relatively costly
– Performs poorly on small/closed hierarchies
– Can't change object type in-situ
– Pay for *potential*, not *realized* flexibility

Devirtualization, take 2

```
class Base {
  struct VTable {
    int (*get)(const Base&);
    int (*set)(Base&, int);
  };
  static VTable vtbl[totalClasses];
  uint8_t tag;
public:
  int get() const {
    return (vtbl[tag].get)(*this);
  }
  int set(int x) {
    (vtbl[tag].set)(*this, x);
  }
};
```


The Cost of Virtual Functions

- Everyone is afraid of Branch Mispredictions
- However - Processors have relatively good predictors.
  - Processors learn your program and the data as it runs..
- Compilers typically only see the program (or part of it)
  - Virtues of PGO, Virtues of LTO
  - C++20: `[[likely]]`
- Devirtualization lets compilers break through virtual calls: inline, inspect, rearrange code.

std::visit for devirtualization

- Base class with some Implementation

```
struct Base {
  virtual int foo(){};
};
struct D1 final : public Base {
  virtual int foo() override { return 1; };
};
struct D2 final : public Base {
  virtual int foo() override { return 2; };
};
```

- Variant of pointers to the same hierarchy.

```
variant<D1*,Base*> myVariant;
visit([](auto* p) {p->foo();}, myVariant);
```

Compiler Explorer

COMPILER EXPLORER | Add... | More | Share | Other

C++ source #1

Save/Load | Add new... | Vim | CppInsights | Quick-bench | C++

```cpp
#include <variant>

using std::variant;

struct Base {
    virtual ~Base() = default;
    virtual int foo() = 0;
};

struct Common final : public Base {
    virtual int foo() {
        return 750;
    }
};

struct Rare final : public Base {
    virtual int foo() {return 322;}
};

int test(std::variant<Common*,Base*> pBase) {
    return visit([](auto*p) { return p->foo();}, pBase);
}
```

x86-64 clang (trunk) (Editor #1, Compiler #1) C++

x86-64 clang (trunk) | -O3 -std=c++17

Output... | Filter... | Libraries | Add new... | Add tool...

```asm
test(std::variant<Common*, Base*>):    # @test(std::variant<Common*, Base*>)
        sub     rsp, 24
        mov     qword ptr [rsp + 8], rdi
        mov     byte ptr [rsp + 16], sil
        cmp     sil, -1
        movzx   eax, sil
        mov     rcx, -1
        cmovne  rcx, rax
        mov     rdi, rsp
        lea     rsi, [rsp + 8]
        call    qword ptr [8*rcx + std::__detail::__variant::__gen_vtable<true, i
        add     rsp, 24
        ret
std::__detail::__variant::__gen_vtable_impl<true, std::__detail::__variant::_Mult
        mov     eax, 750
        ret
std::__detail::__variant::__gen_vtable_impl<true, std::__detail::__variant::_Mult
        mov     rdi, qword ptr [rsi]
        mov     rax, qword ptr [rdi]
        jmp     qword ptr [rax + 16]    # TAILCALL
std::__detail::__variant::__gen_vtable<true, int, test(std::variant<Common*, Base
        .quad   std::__detail::__variant::__gen_vtable_impl<true, std::__detail::
        .quad   std::__detail::__variant::__gen_vtable_impl<true, std::__detail::
```

“Slower” visit

COMPILER EXPLORER

Add... More Share

C++ source #1

A Save/Load + Add new... Vim CppInsights Quick-bench C++

```cpp
#include <utility>
#include <variant>
using namespace std;

struct Base {
  virtual ~Base() = default;
  virtual int foo() = 0;
};

struct Common final : public Base {
  virtual inline int foo() __attribute__((always_inline)) { return 750; }
};

struct Rare final : public Base {
  virtual int foo() { return 322; }
};

template <typename Visitor, typename Variant, size_t Index = 0>
decltype(auto) visit_no_table(Visitor &&visitor, Variant &&variant) {
  if (Index == variant.index()) /*[[likely]]*/ {
    return visitor(*get_if<Index>(&forward<Variant>(variant)));
  }
  if constexpr (Index + 1 < variant_size_v<remove_reference_t<Variant>>) {
    return visit_no_table<Visitor, Variant, Index + 1>(
        forward<Visitor>(visitor), std::forward<Variant>(variant));
  }
  return visitor(*get_if<Index>(&forward<Variant>(variant)));
}

int test(std::variant<Common *, Base *> pBase) {
  return visit_no_table([](auto *p) { return p->foo(); }, pBase);
}
```

x86-64 clang (trunk) (Editor #1, Compiler #1) C++

x86-64 clang (trunk) -O3 -std=c++17

A Output... Filter... Libraries + Add new... Add tool...

```asm
test(std::variant<Common*, Base*>): # @test(std::variant<Common*, Base*>)
  test sil, sil
  je .LBB0_1
  mov rax, qword ptr [rdi]
  jmp qword ptr [rax + 16] # TAILCALL
.LBB0_1:
  mov eax, 750
  ret
```