Variations on variants

Roi Barkan

Slides online here

C++ now
2021
MAY 2-7
Aspen, Colorado, USA

Hi, I'm Roi

- Roi Barkan - רועי ברקן
- I live in Tel Aviv, Israel
- C++ developer since 2000
- VP Technologies @ Istra Research
  - Finance, Low Latency, in Israel
  - careers@istraresearch.com
- First time to CPPNow
  - Try to be active in chat, Q&A
  - Please - ask questions, make comments

Agenda

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

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 << std::get<int>(var);
 //or
 std::visit(
   [](auto &item) {cout << item;}, 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>
```

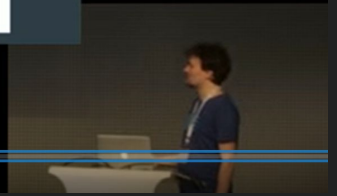
What is it Good for ?

- State Machines
- Value-Semantics for Dynamic Types
  - Commands
- Success/fail
  - expected<T>
- Exists/void
  - optional<T>
- Run-Time Dispatch (Polymorphism)
- 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

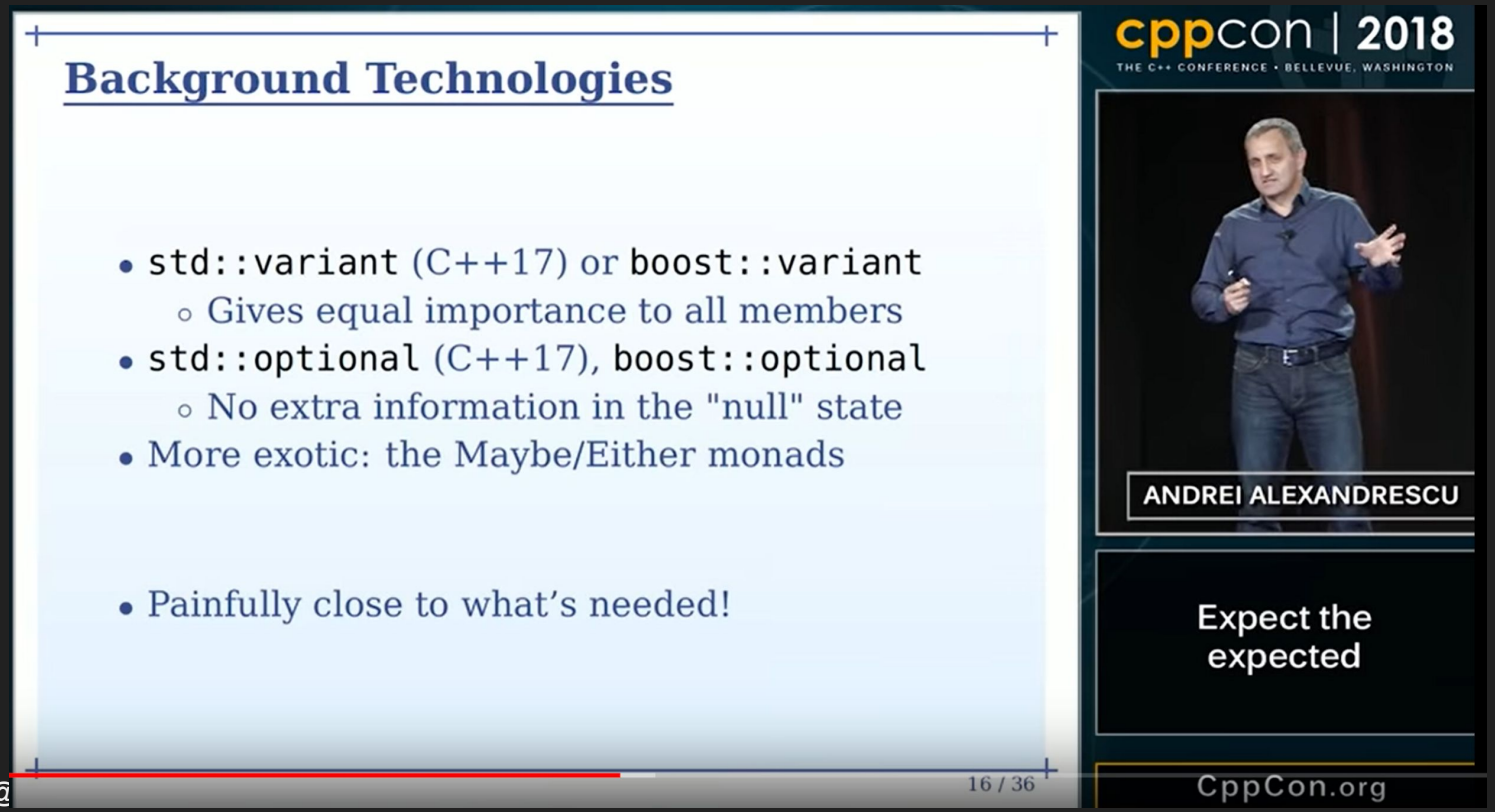
```
struct command {
  enum type { SET_SCORE, FIRE_MISSILE, FIRE_LASER, ROTATE };
  virtual type getType();
};

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

Now

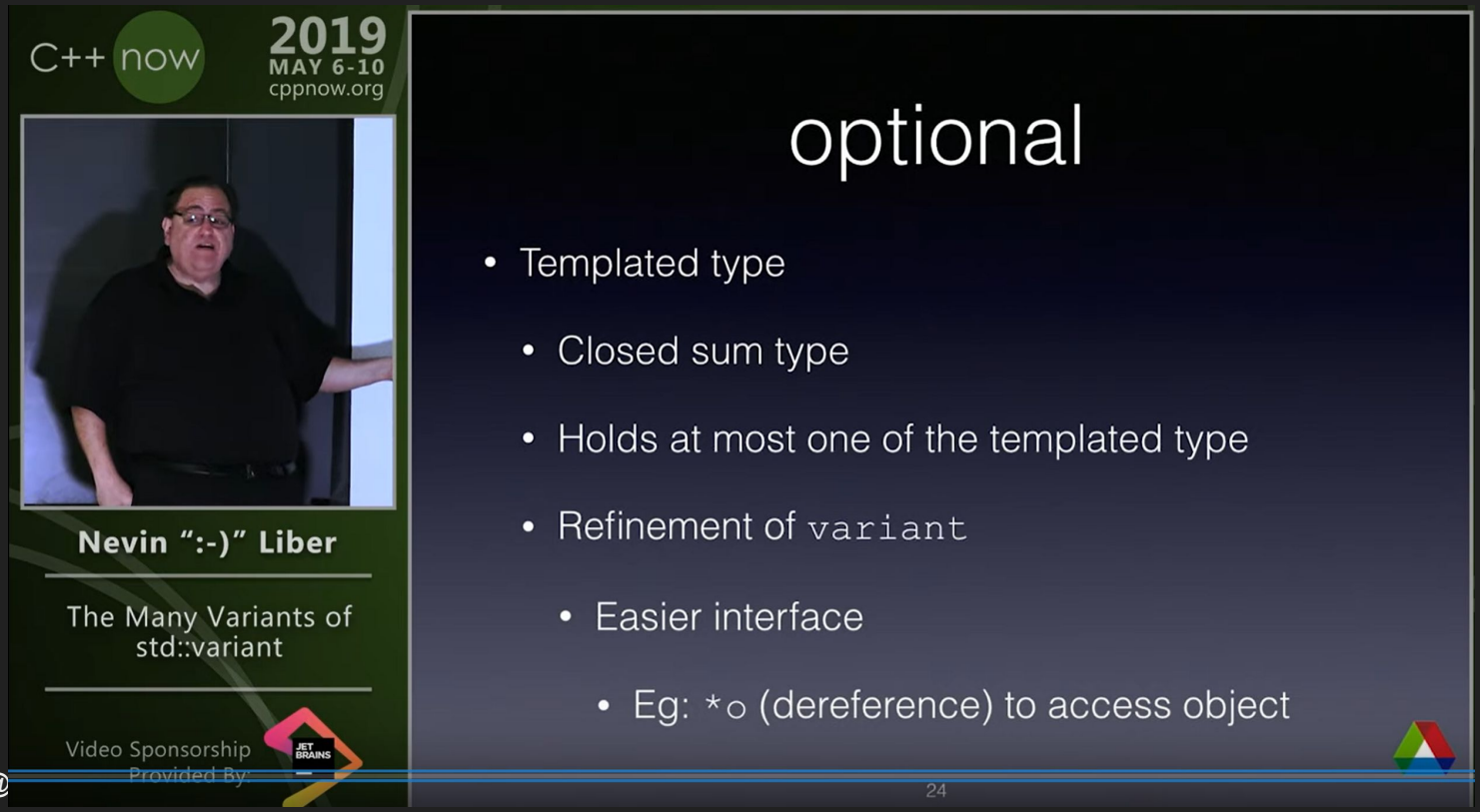
```
struct set_score {
  double value;
};
using command = variant<set_score, fire_missile, fire_laser, rotate>;
```

12 / 65

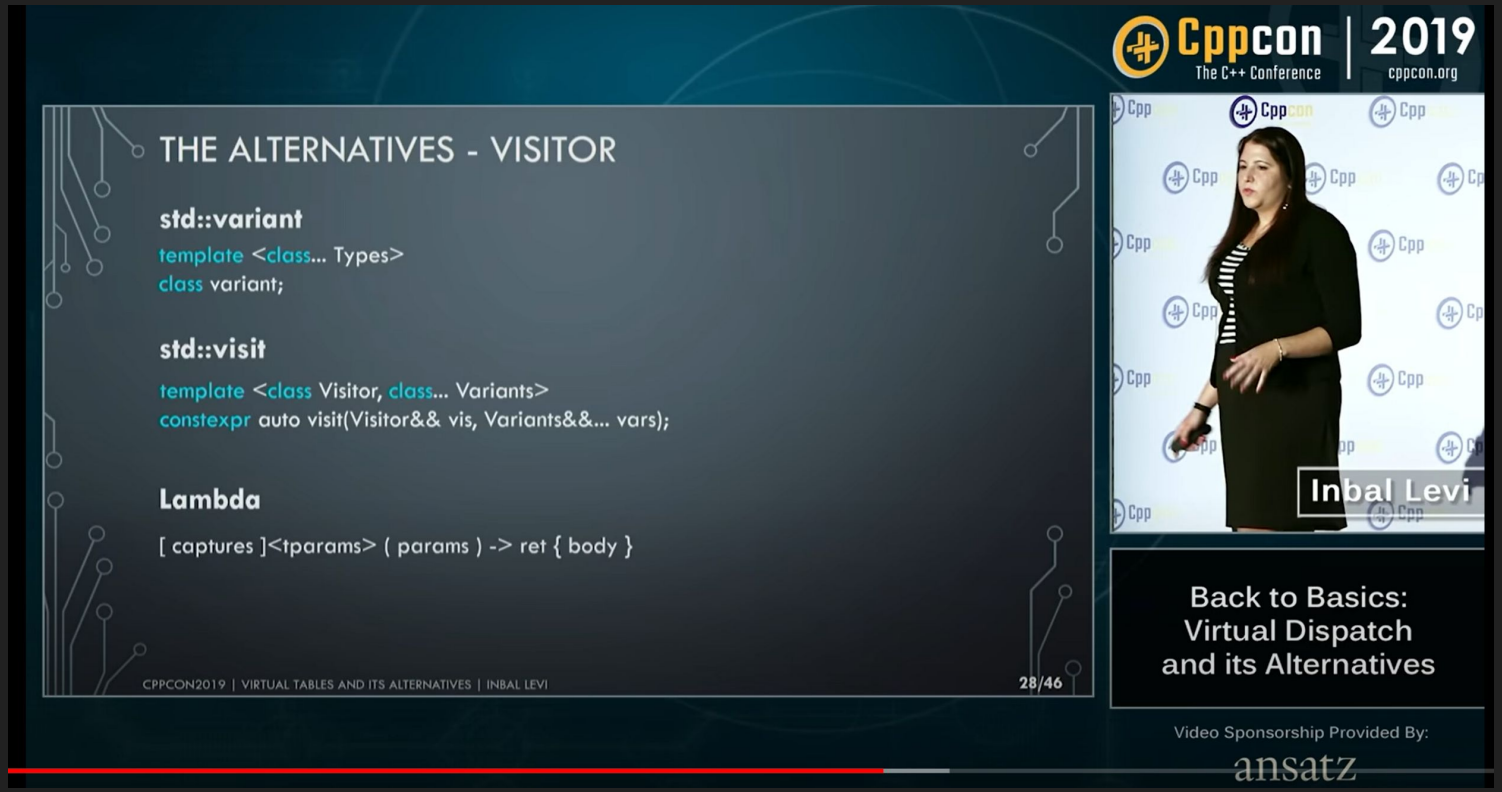
expected


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

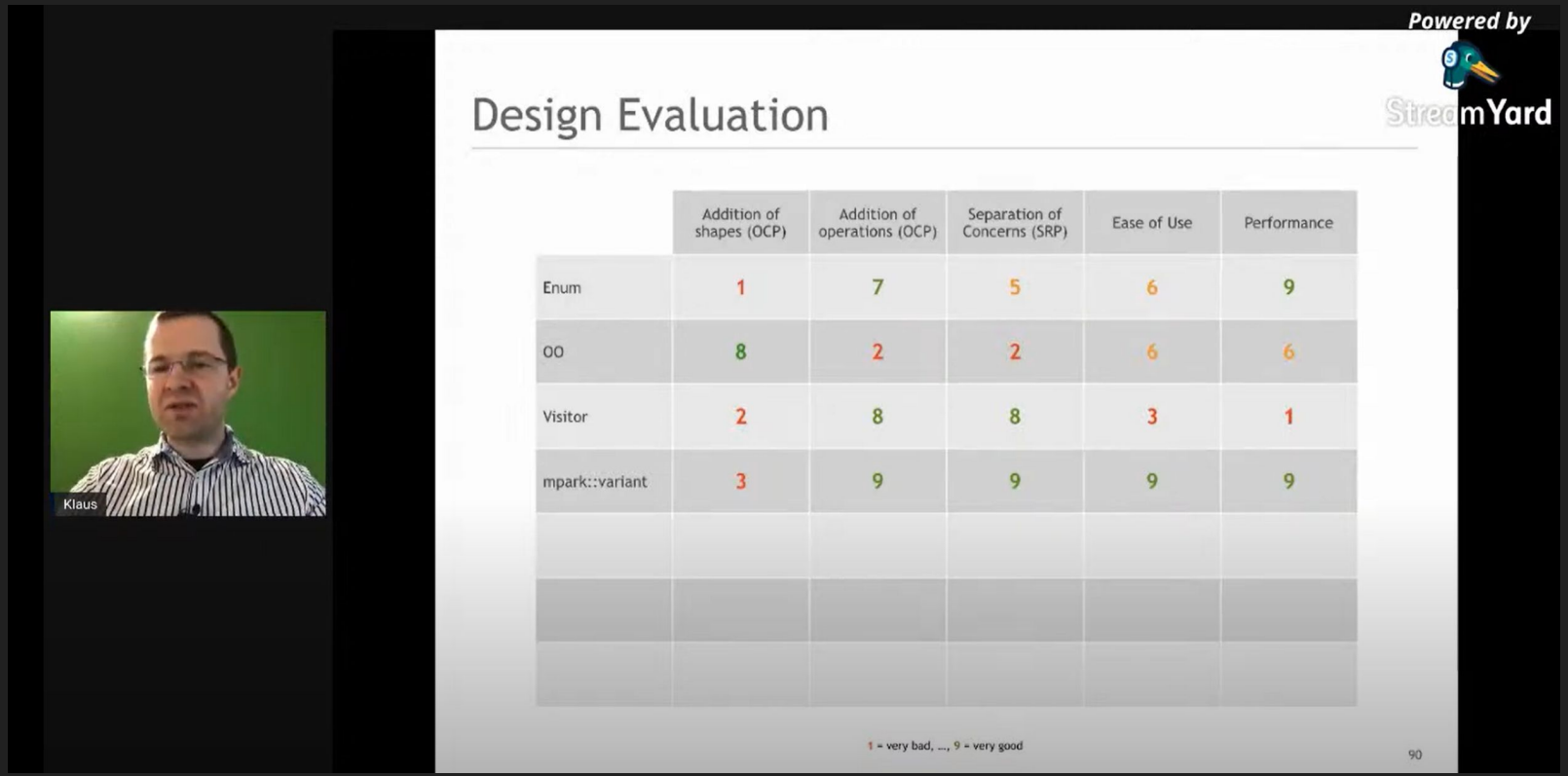
Optional


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

Run-Time Dispatch

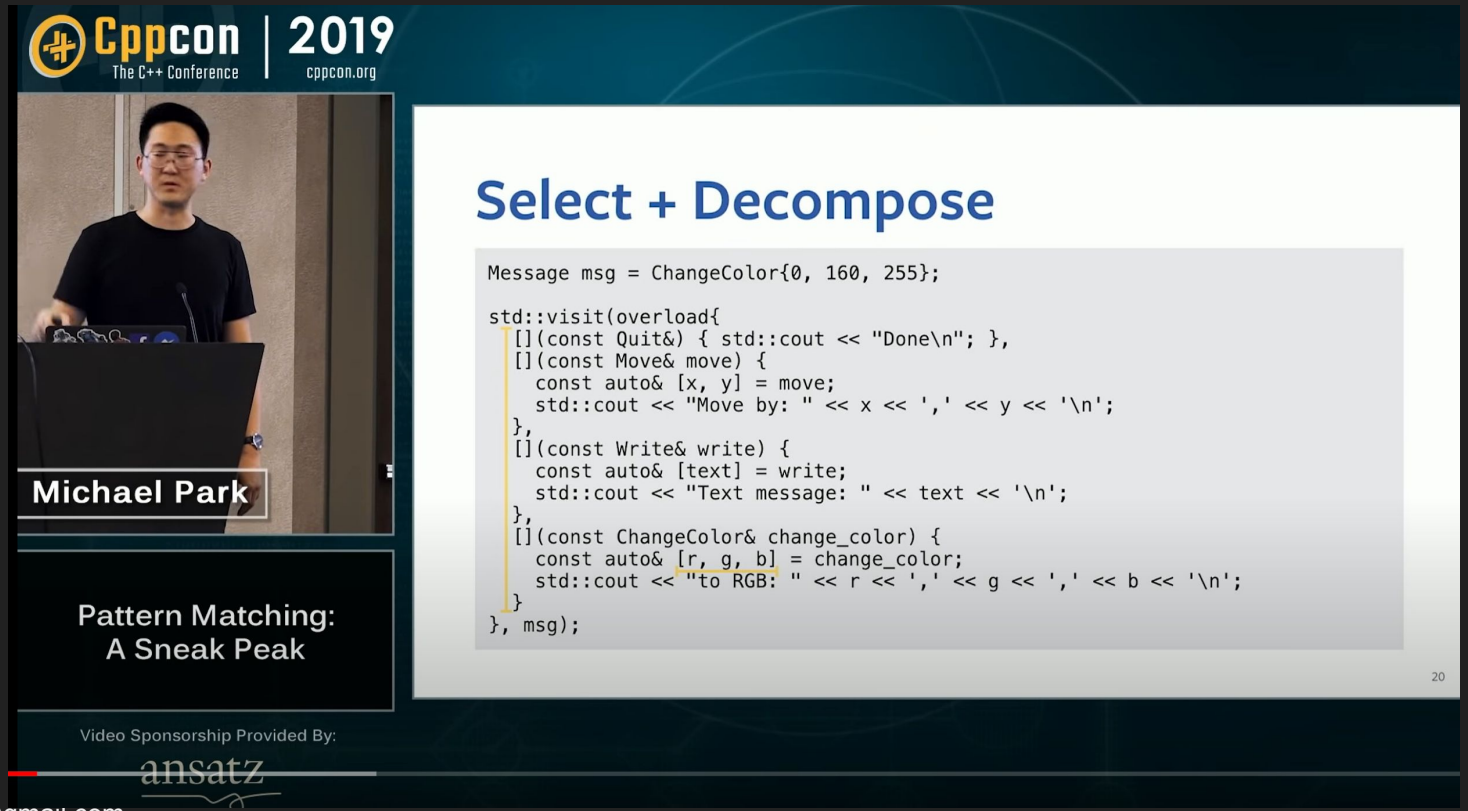


More Run-Time Dispatch

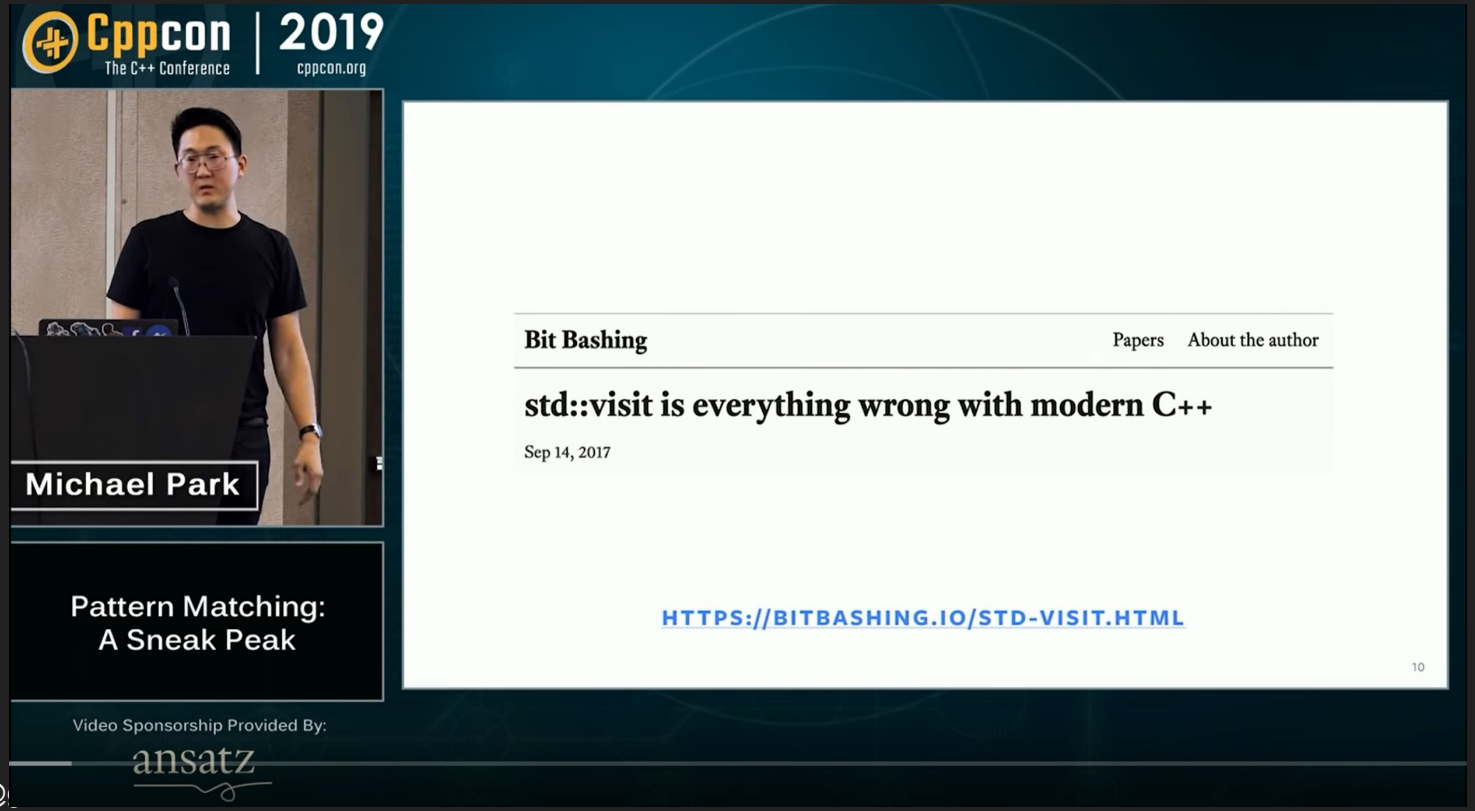


	Addition of shapes (OCP)	Addition of operations (OCP)	Separation of Concerns (SRP)	Ease of Use	Performance
Enum	1	7	5	6	9
OO	8	2	2	6	6
Visitor	2	8	8	3	1
mpark::variant	3	9	9	9	9

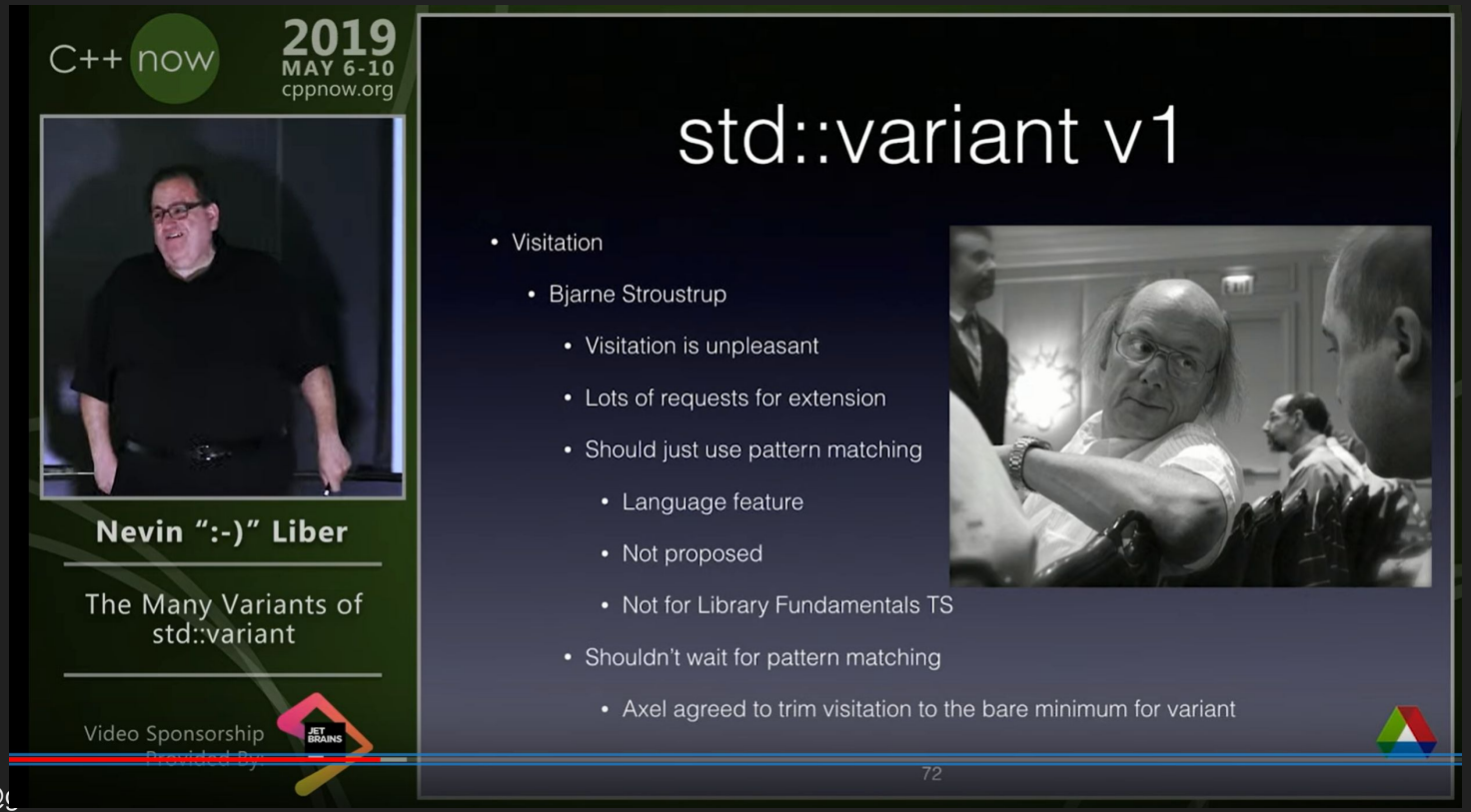
Pattern Matching



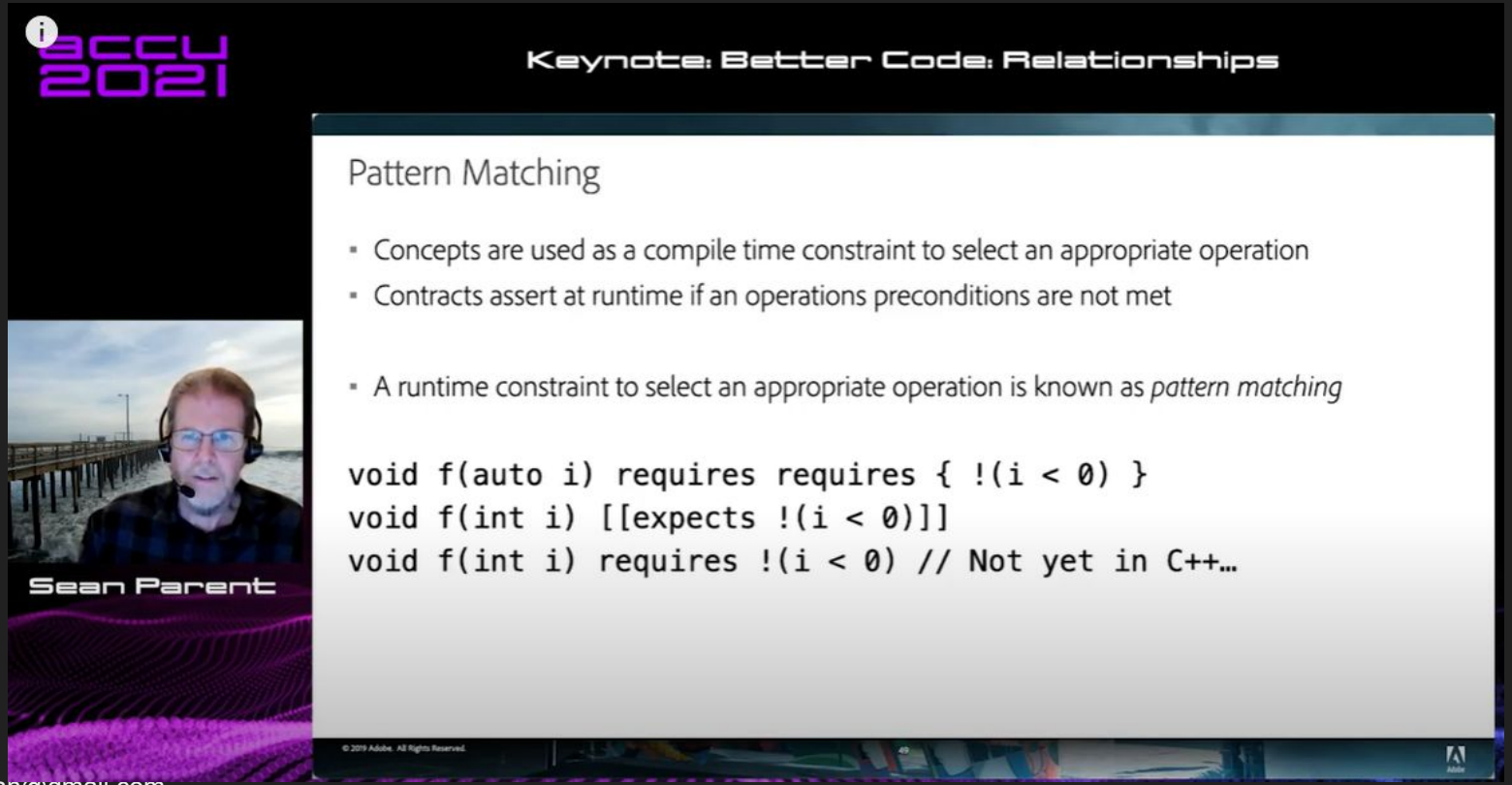
Pattern Matching - improved visit()



Pattern Matching - Anticipated



Pattern Matching vs. Concepts/Contracts



Pattern Matching - inspect()

Before	After

```
switch (x) {
  case 0: std::cout << "got zero"; break;
  case 1: std::cout << "got one"; break;
  default: std::cout << "don't care";
}
```

```
inspect (x) {
  0 => { std::cout << "got zero"; }
  1 => { std::cout << "got one"; }
  __ => { std::cout << "don't care"; }
};
```

```
auto&& [x, y] = p;
if (x == 0 && y == 0) {
  std::cout << "on origin";
} else if (x == 0) {
  std::cout << "on y-axis";
} else if (y == 0) {
  std::cout << "on x-axis";
} else {
  std::cout << x << ',' << y;
}
```

```
inspect (p) {
  [0, 0] => { std::cout << "on origin"; }
  [0, y] => { std::cout << "on y-axis"; }
  [x, 0] => { std::cout << "on x-axis"; }
  [x, y] => { std::cout << x << ',' << y; }
};
```

inspect() variants

Before

```
struct visitor {
  void operator()(int i) const {
    os << "got int: " << i;
  }
  void operator()(float f) const {
    os << "got float: " << f;
  }
  std::ostream& os;
};
std::visit(visitor{strm}, v);
```

After

```
inspect (v) {
  <int> i => {
    strm << "got int: " << i;
  }
  <float> f => {
    strm << "got float: " << f;
  }
};
```

Inspect() in a nutshell

- Runtime matching types and values
  - `switch() + std::visit() + [structured, bindings]`
- tuple, variant, any, polymorphic objects, extractors
- First-match, recursive inspection, no fallthrough, [[strict]]
- Second tier priority for C++23 (should make progress)

Pattern Matching FTW

```
template <typename T>
void Node<T>::balance() {
  *this = inspect (*this) {
    // left-left case
    //
    //        (Black) z              (Red) y
    //         /     \               /     \
    //     (Red) y    d        (Black) x  (Black) z
    //      /     \       ->     /   \      /   \
    //  (Red) x    c            a     b    c     d
    //   /    \
    //  a      b
    [case Black, (*?) [case Red, (*?) [case Red, a, x, b], y, c], z, d]
      => Node{Red, std::make_shared<Node>(Black, a, x, b),
                   y,
                   std::make_shared<Node>(Black, c, z, d)};
    [case Black, (*?) [case Red, a, x, (*?) [case Red, b, y, c]], z, d]  // left-right case
      => Node{Red, std::make_shared<Node>(Black, a, x, b),
                   y,
                   std::make_shared<Node>(Black, c, z, d)};
    [case Black, a, x, (*?) [case Red, (*?) [case Red, b, y, c], z, d]]  // right-left case
      => Node{Red, std::make_shared<Node>(Black, a, x, b),
                   y,
                   std::make_shared<Node>(Black, c, z, d)};
    [case Black, a, x, (*?) [case Red, b, y, (*?) [case Red, c, z, d]]]  // right-right case
      => Node{Red, std::make_shared<Node>(Black, a, x, b),
                   y,
                   std::make_shared<Node>(Black, c, z, d)};
    self => self;  // do nothing
  };
}
```

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

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 code need access to header.
- Preprocessor (??):

```
#define HEADER_FIELDS          \
  IDType type;                 \
  Date expiration;

struct Header {  HEADER_FIELDS };
struct Citizen {  HEADER_FIELDS
  IDNumber id;
};
struct Tourist { HEADER_FIELDS
  PassportNumber id;
};
union IdentityCard {
  Header header;
  Citizen citizen;
  Tourist tourist;
};
```

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

Unions in the C++ Standard

tag	int/char/double

- Accessing header fields is properly defined in many cases.
- Standard-layout class is part of the C++ standard
  - "Standard-layout classes are useful for communicating with code written in other programming languages."
  - Various constraints (no virtuals, single access control, all non-statics in same class,...)
  - C++11: `static_assert(std::is_standard_layout<Citizen>::value, "not standard");`
  - C++17: `static_assert(std::is_standard_layout_v<Citizen>);`
- Layout-compatibility allows accessing members without knowing the type
  - "In a standard-layout union with an active member of struct type T1, it is permitted to read a non-static data member m of another union member of struct type T2 provided m is part of the common initial sequence of T1 and T2"
  - C++20: `static_assert(std::is_corresponding_member(&Header::type, &Citizen::type));`

intrusive_variant

tag	int/char/double

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

```
union IdentityCard {
  struct IDHeader {
    const IDType type;
  } header;
  struct Citizen {
    const IDType type;
    static constexpr IDType TAG =
                IDType::CITIZEN;
    IDNumber id;
  } citizen;
  struct Tourist {
    const IDType type;
    static constexpr IDType TAG =
                IDType::TOURIST;
    PassportNumber passport;
  } tourist;
};
```

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

Different Approaches to get the Tag

- Offset of the field in the object

```
getTag<IDType>(hdr, offsetof(Hdr, tag));
getTag<IDType>(hdr, std::integral_constant<size_t, offsetof(Hdr, tag)>());
```

- Pointer to the field

```
getTag<IDType>(hdr, &Hdr::tag);
```

- Call a member function

```
getTag<IDType>(hdr, &Hdr::getTag);
```

  - Useful when the tag is private
  - Useful when the tag is calculated

- Call a free-function / lambda

```
getTag<IDType>(hdr, [](const Hdr& h) { return h.tag; });
```

  - Less intrusive

Implementation of Tag Extraction

```
template <typename R, typename T, typename Offset>
R getTag(const T& x,
         Offset offset) requires std::convertible_to<Offset, size_t> {
  return *reinterpret_cast<const R*>(reinterpret_cast<const char*>(&x) +
                                     offset);
}
template <typename R, typename T, typename M>
R getTag(const T& x, M method) requires std::invocable<M, T> {
  return std::invoke(method, x);
}
```

- std::invoke() is flexible - functions, lambdas, pointer-to-member

Add C++20 Safety

tag	int/char/double

```
using ID_intrusive = decltype(decl_safe_intrusive_variant(
    &Citizen::type, IDType::CITIZEN,
    &Tourist::type, IDType::TOURIST));
```

- Helper function to ease type deduction.
- Validate corresponding members.
  - No need to deal with offsets, etc.
- constexpr generic lambdas can be used for extra beauty

Questions, Comments...

roi.barkan@gmail.com

Istra Research

More C++ Safety

- Intrusive_variant has safe visit() and links the Type with the Tag
- Still - we need to add boilerplate and can still have bugs.
- Class Hierarchies can do better:
  - Base class is essentially a header.
  - 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
- The problem - base class with data members isn't *standard-layout*
  - But it generally works. Perhaps we can update the standard

Sean Parent on variant member similarity

variant_of_base

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

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

Arrays of Variants

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

- 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 headers)
- Intrusive_variant - C++ safety with high C compatibility
- Variant_of_base - add classes to your code
  - Not standard-layout, undefined behavior
  - Perhaps we should widen the rules, add [[standard_layout]] ?
- condensed_variant - real world streams of binary data

Questions, Comments...

roi.barkan@gmail.com

Istra Research

Assisting 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 {
  int foo() override { return 1; };
};
struct D2 final : public Base {
  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 | Sponsors PC-lint | 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 {
    int foo() override {
        return 750;
    }
};

struct Rare final : public Base {
    int foo() override {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++20

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<tru
        add     rsp, 24
        ret
std::__detail::__variant::__gen_vtable_impl<true, std::__detail::__variant::_
        mov     eax, 750
        ret
std::__detail::__variant::__gen_vtable_impl<true, std::__detail::__variant::_
        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*,
        .quad   std::__detail::__variant::__gen_vtable_impl<true, std::__deta
        .quad   std::__detail::__variant::__gen_vtable_impl<true, std::__deta
```

“Slower” visit (https://godbolt.org/z/eTcEa5oqE)

COMPILER EXPLORER

C++ source #1

```cpp
#include <utility>
#include <variant>

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

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

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

template <typename Visitor, typename Variant, std::size_t Index = 0>
decltype(auto) visit_no_table(Visitor&& visitor, Variant&& variant) {
  if (Index == variant.index()) {
    return visitor(*get_if<Index>(&std::forward<Variant>(variant)));
  }
  if constexpr (Index + 1 <
                std::variant_size_v<std::remove_reference_t<Variant>>) {
    return visit_no_table<Visitor, Variant, Index + 1>(
        std::forward<Visitor>(visitor), std::forward<Variant>(variant));
  }
    return visitor(*get_if<Index>(&std::forward<Variant>(variant)));
}
int test(std::variant<Common*, Base*> myVariant) {
  return visit_no_table([](auto* p) { return p->foo(); }, myVariant);
}
```

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

x86-64 clang (trunk) -std=c++20 -O3

```asm
test(std::variant<Common*, Base*>):    # @test(std::variant<Common*, Base*>)
        test    sil, sil
        je      .LBB0_3
        cmp     sil, 1
        jne     .LBB0_4
        mov     rax, qword ptr [rdi]
        jmp     qword ptr [rax]                 # TAILCALL
.LBB0_3:
        mov     eax, 750
        ret
.LBB0_4:
        ud2
```

Output (0/0) x86-64 clang (trunk) - cached (126846B) ~2178 lines filtered

#1 with x86-64 clang (trunk)

Thank You

roi.barkan@gmail.com