Variants

Past, Present, and Future

Bloomberg

David Sankel - 9/19/2016 - CppCon 2016

Timeline

- *2004*. Boost.Variant released in Boost 1.31.0.
- *2016*. `std::variant` voted into C++17.
- *2020*. Language-based variants in C++20?

Motivating Examples

A `PersonId` has a name or a "John Doe" index, but not both.

```
class PersonId {
public:
  std::string getName();
  int getJohnDoeId();

  // etc.
private:
  bool m_hasName;
  std::string m_name;
  int m_johnDoeId;
};
```

Document Undefined Behavior

```
class PersonId {
public:
  // Behavior is undefined unless 'hasName() == true'.
  std::string getName()

  // Behavior is undefined unless 'hasJohnDoeId() == true'.
  int getJohnDoeId();

  bool hasName() { return m_hasName; }
  bool hasJohnDoeId() { return !m_hasName; }

  // etc.
};
```

Problems with this approach

- No compiler help for checking preconditions
- Wasted space
- Error-prone Boilerplate

Another example

```
struct command {
  virtual std::ostream& put(std::ostream&)=0;
  virtual ~command();
};

struct set_score : commmand {
  std::ostream& put(std::ostream&) override final;
  double value;
};
```

- Allocation and memory management required.
- Virtual function required for every operation over commands. Functions are "split".
- Lots of boilerplate.
- Open instead of closed.

```
struct command {
  enum type { SET_SCORE, FIRE_MISSILE, FIRE_LASER, ROTATE };
  virtual type getType()=0;
  virtual ~command();
};

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

- Allocation and memory management still required.
- Functions are no longer "split".
- Even more boilerplate, this time error prone.
- Still open instead of closed.

```
template<typename LeafData>
struct BinaryTreeNode {
  // etc.
};
template<typename LeafData>
struct BinaryTreeBranch : BinaryTreeNode<LeafData> {
  std::unique_ptr<BinaryTreeNode<LeafData>> left;
  std::unique_ptr<BinaryTreeNode<LeafData>> right;
  // etc.
};
template<typename LeafData>
struct BinaryTreeLeaf : BinaryTreeNode<LeafData> {
  LeafData data;
  // etc.
};
```

And vs. or for combining types

`struct` has one of `X` *and* one of `Y`

```
struct S {
  X x;
  Y y;
};
```

`variant` has one of `X` *or* one of `Y`

```
variant<X,Y>
```

PersonId

```
class PersonId {
public:
  std::string getName();
  int getJohnDoeId();

  // etc.
private:
  bool m_hasName;
  std::string m_name;
  int m_johnDoeId;
};
```

Now

```
using PersonId = variant<std::string, int>;
```

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, rot
```

The Many Names of Variant

- *or* type
- Sum type (⨁)
- Discriminated union (⊎)
- Algebraic Data Type (ADT) "|" operator
- "One of" type

Variants vs. Inheritance

Inheritance	Variant
open to new alternatives	closed to new alternatives
closed to new operations	open to new operations
multi-level	single level
OO	functional
complex	simple

Review Boost.Variant

Creation and Assignment

```
// Default constructs to first alternative.
boost::variant<std::string, int> v;

// Assignment to alternative types.
v = 3;
v = "Hello World";
```

Extract a Value with get

```
void output(const boost::variant<std::string, int>& v)
{
  if(std::string const * const s = boost::get<std::string>(&v))
    std::cout << "I got a string: " << *s << std:endl;
  else
    std::cout << "I got an int: " << boost::get<int>(v) << std:e
}
```

Extract a Value with a visitor

```
struct OutputVisitor
{
  using result_type = void;
  void operator()(const std::string& s) const {
    std::cout << "I got a string: " << s << std:endl;
  }
  void operator()(const int i) const {
    std::cout << "I got an int: " << i << std:endl;
  }
};

void output(const boost::variant<std::string, int>& v)
{
  boost::apply_visitor(OutputVisitor(), v);
}
```

Use get or visitor?

Visitor benefits:

- Compile-time guarantee that all cases are handled.

Get benefits:

- Code next to usage.
- Succinct syntax.

12 Years of Experience

Newtypes

```
using command = boost::variant<set_score,
                               fire_missile,
                               fire_laser,
                               rotate>;
```

Typedefs are problematic:

- Lack of forward declaration hurts compilation times.
- Error messages unreadable.
- Unique type better matches intended semantics.

Use inheritance

```
struct command : boost::variant<set_score,
                                fire_missile,
                                fire_laser,
                                rotate>
{
  template <typename T,
            typename X = disable_if_same_or_derived<command, T>>
  command(T&& t);

  command(const command& original);

  command(command&& original);
};
```

Works, but...

- Lots of boilerplate.
- Especially when working with older compilers.

Wrap it.

```
struct command
{
  using Value = boost::variant<set_score,
                               fire_missile,
                               fire_laser,
                               rotate>;

  Value value;
};
```

Simpler, but...

- Users need to "unwrap" by accessing the `value` field.

Recursion

```
template<typename LeafData>
struct BinaryTreeNode {
  // etc.
};
template<typename LeafData>
struct BinaryTreeBranch : BinaryTreeNode<LeafData> {
  std::unique_ptr<BinaryTreeNode<LeafData>> left;
  std::unique_ptr<BinaryTreeNode<LeafData>> right;
  // etc.
};
template<typename LeafData>
struct BinaryTreeLeaf : BinaryTreeNode<LeafData> {
  LeafData data;
  // etc.
};
```

Recursion

```
template<typename Tree>
struct BinaryTreeBranch{
  Tree left;
  Tree right;
};

template<typename LeafData>
using BinaryTree = typename boost::make_recursive_variant<
    LeafData,
    BinaryTreeBranch<boost::recursive_variant_>
  >::type;
```

- Tight syntax.
- Hidden allocation.

Direct Recursion

```
template <typename LeafData>
struct BinaryTree;

template <typename LeafData>
struct BinaryTreeBranch {
  std::shared_ptr<BinaryTree<LeafData>> left;
  std::shared_ptr<BinaryTree<LeafData>> right;
};

template <typename LeafData>
struct BinaryTree {
  using Value = boost::variant<LeafData, BinaryTreeBranch<LeafDa
  Value value;
};
```

- Control over allocation.
- Forward declarations.
- More straightforward.
- Not much longer.

The assignment problem

```
boost::variant<A, B> v = A( /*...*/ );
v = B( /*...*/ );
```

What happens on the second line?

- v's `A` is destructed.
- v's index is set to `B`.
- v's `B` is initialized to the right-hand side value.

Boost.Variant move constructor exception solution

- Copy-construct the content of the left-hand side to the heap; call the pointer to this data backup.
- Destroy the content of the left-hand side.
- Copy-construct the content of the right-hand side in the (now-empty) storage of the left-hand side.
- In the event of failure, copy backup to the left-hand side storage.
- In the event of success, deallocate the data pointed to by backup.

Workarounds

- Ensure all types are no-throw copy constructable.
- Ensure one type is no-throw default constructable.
- Always use `boost::blank` as the first alternative for variant types.

std::variant

- Axel Naumann (axel@cern.ch)
- P0088R3
- Implementations
  - Anthony Williams: https://bitbucket.org/anthonyw/variant
  - Eric Fiselier: https://github.com/efcs/libcxx/tree/variant

Change 1: apply_visitor renamed

```
struct OutputVisitor
{
  void operator()(const std::string& s) const {
    std::cout << "I got a string: " << s << std:endl;
  }
  void operator()(const int i) const {
    std::cout << "I got an int: " << i << std:endl;
  }
};

void output(const std::variant<std::string, int>& v)
{
  std::visit(OutputVisitor(), v);
}
```

Change 2: get reworked

```
void output(const std::variant<std::string, int>& v)
{
  if(std::string const * const s = std::get_if<std::string>(&v))
    std::cout << "I got a string: " << *s << std:endl;
  else
    std::cout << "I got an int: " << std::get<int>(v) << std:enc
}
```

Change 3: index-based access.

```
void output(const std::variant<std::string, int>& v)
{
  if(std::string const * const s = std::get_if<0>(&v))
    std::cout << "I got a string: " << *s << std:endl;
  else
    std::cout << "I got an int: " << std::get<1>(v) << std:endl;
}
```

Change 4: duplicated entries

```
std::variant<std::string, std::string> v1;
std::variant<std::size_t, unsigned> v2;
```

Change 5: No special recursion support

```
template <typename LeafData>
struct BinaryTree;

template <typename LeafData>
struct BinaryTreeBranch {
  std::shared_ptr<BinaryTree<LeafData>> left;
  std::shared_ptr<BinaryTree<LeafData>> right;
};

template <typename LeafData>
struct BinaryTree {
  using Value = std::variant<LeafData, BinaryTreeBranch<LeafDat
  Value value;
};
```

Change 6: `boost::blank` renamed to `std::monostate`

Change 7: allocation at assignment removed

Alternative 1: Explicit empty

- If an exception is thrown on assignment, put variant into the empty state.
- A default constructed variant is in the empty state.

Pros:

- Predictable space usage
- "Teachable"

Cons:

- Error-prone
- Complex semantics

Alternative 2: Double Buffer

- Never-empty guarentee.
- Use double buffering to recover in case of throw on assignment.
- Use single buffering for "friendly" types.

Pros:

- Simple semantics.
- Never-empty is what most situations need.

Cons:

- Difficult to predict and control space requirements.
- High price for exceedingly rare situation.

What we got: Rarely Empty

- Empty state called `valueless_by_exception`.
- If an exception is thrown on assignment, put variant into the `valueless_by_exception` state.
- "friendly" types cannot get into the `valueless_by_exception` state.

Pros:

- Predictable space usage
- "Teachable"
- Simple semantics in normal usage
- Consensus

Cons:

- Complexity in exceptional case.

When do we deal with `valueless_by_exception` variants?

std::variant

- Mostly incremental improvements over Boost.Variant.
- Handling of exceptions on assignment is the big change.
- In C++17 working paper.

Language support for variant.

What it looks like

```
// This lvariant implements a value representing the various
// commands available in a hypothetical shooter game.
lvariant command {
  std::size_t    set_score;    // Set the score to value.
  std::monostate fire_missile; // Fire a missile.
  unsigned       fire_laser;   // Fire a laser with the specifie
                               // intensity.
  double         rotate;       // Rotate the ship by the specifi
                               // degrees.
};
```

Creation of Alternatives

```
// Create a new command 'cmd' that sets the score to '10'.
command cmd = command::set_score( 10 );
```

Basic Pattern Matching

```
// Output a human readable string corresponding to the specifiec
// command to the specified 'stream'.
std::ostream& operator<<( std::ostream& stream, const command cm
  inspect( cmd ) {
    set_score value =>
      stream << "Set the score to " << value << ".\n";
    fire_missile m =>
      stream << "Fire a missile.\n";
    fire_laser intensity =>
      stream << "Fire a laser with " << intensity << " intensity
    rotate degrees =>:
      stream << "Rotate by " << degrees << " degrees.\n";
  }
}
```

Is a library solution sufficient?

```
using command = std::variant<
  std::size_t,    // Set the score to the specified value.
  std::monostate, // Fire a missile.
  unsigned,       // Fire a laser with the specified intensity.
  double          // Rotate the ship by the specified degrees.
  >;
```

- Two visit options:
  - type based (`std::get<T>(some_command)` where `T` is a type and `apply_visitor` style functions)
  - index based (`std::get<n>(some_command)` where `n` is a number)

Index based visit is nasty

```
// huh?
std::get<3>( some_command );
```

So maybe...

```
const std::size_t set_score = 0;
const std::size_t fire_missle = 1;
const std::size_t fire_laser = 2;
const std::size_t rotate = 3;
```

- More boilerplate.
- Error prone.

Type based visit is nasty

```
using command = std::variant<
  std::size_t,    // Set the score to the specified value.
  std::monostate, // Fire a missile.
  unsigned,       // Fire a laser with the specified intensity.
  double          // Rotate the ship by the specified degrees.
  >;
```

What is `std::get<unsigned>`?

- Does it compile?
- Does it give me the `set_score` alternative?
- Is it platform dependant?

All answers are bad.

Type based visit is nasty

So use tags as types...

```
struct set_score { std::size_t value; };
struct fire_missile {};
struct fire_laser { unsigned intensity; };
struct rotate { double degrees; };
using command = std::variant<
  set_score,
  fire_missle,
  fire_laser,
  rotate,
  >;
```

- More boilerplate.
- Introduction of types that *may not make sense in isolation*.

struct/tuple connection

We could use tuples instead of structs everywhere.

```
// point type as a struct
struct point {
  double x;
  double y;
  double z;
};

// point type as a tuple
using point = std::tuple< double, double, double >;
```

The `std::tuple` version has all the same problems as the indexed variant though.

So use type-based dispatch then...

```
struct x { double value; };
struct y { double value; };
struct z { double value; };

using point = std::tuple< x, y, z >;
```

- What is an x?
- Would anyone recommend doing this instead of using a `struct`?
- Tuples have their use, but so do `structs`.

lvariants relate very closely to structs.

```
// copoint type as lvariant
lvariant copoint {
  double x;
  double y;
  double z;
};

// copoint type as a variant
using copoint = std::variant< double, double, double >;
```

Same problems...

Same reasoning...

```
struct x { double value; };
struct y { double value; };
struct z { double value; };

using copoint = std::variant< x, y, z >;
```

- What is an x?
- Would anyone recommend doing this instead of using an `lvariant`?
- `std::variants` have their use, but so do `lvariants`.

A sampling of std::variant problems:

- Unhelpful error messages.
- Ugly visitor code.
- Portability issues.

```
// Is this code future proof? Not likely.
using database_handle = std::variant<ORACLE_HANDLE, BERKELEY_HAN
```

Variants are a simple and common need, but an library-only solution is too complex.

Our proposal...a basic language-based variant.

```
lvariant json_value {
  std::map<std::string, json_value> object;
  std::vector<json_value> array;
  std::string string;
  double number;
  bool boolean;
  std::monostate null_;
};
```

Pattern matching is closely tied to Ivariants

```
inspect( cmd ) {
  set_rotation r => // ...
  set_position {0.0, 0.0} =>
    // handle the origin case specially...
  set_position {x, y} =>
    // handle other cases ...
}
```

Pattern match integrals and enums

```
inspect(i) {
  0 =>
    std::cout << "Can't say I'm positive or negative on this syr
              << std::endl;
  6 =>
    std::cout << "Perfect!" << std::endl;
  _ =>
    std::cout << "I don't know what to do with this." << std::er
}
```

Pattern match structs

```
struct player {
  std::string name;
  int hitpoints;
  int coins;
};

void log_player( const player & p ) {
  inspect(p) {
    {n,h,c}
      => std::cout << n << " has " << h << " hitpoints and "
                   << c << " coins.";
  }
}
```

```
void get_hint( const player & p ) {
  inspect(p) {

    {hitpoints:1}
      => std::cout << "You're almost dead. Quit!" << std::endl;

    {hitpoints:10, coins:10}
      => std::cout << "I need the hints from you!" << std::endl;

    {coins:10}
      => std::cout << "Get more hitpoints!" << std::endl;

    {hitpoints:10}
      => std::cout << "Get more ammo!" << std::endl;
```

How can we enable types to opt-in?

```
template <class T1, class T2>
class pair {
  T1 m_first;
  T2 m_second;

public:
  // etc.

  operator extract(std::tuple_piece<T1> x, std::tuple_piece<T2>
    x.set( &this->first );
    y.set( &this->second );
  }
};
```

Variants: Past, Present, and Future

Related posts at davidsankel.com.

Questions?

Bloomberg

David Sankel - 9/19/2016 - CppCon 2016

Extra Slides

Overload

- Vicente Escriba
- P0051R1

```
void output(const std::variant<std::string, int>& v)
{
  return std::visit(
    std::overload(
      [](const std::string & s) {
        std::cout << "I got a string: " << s << std:endl;
      },
      [](const int i) {
        std::cout << "I got an int: " << i << std:endl;
      } ),
    v );
}
```

Visitors with extra arguments.

```
std::ostream &put(std::ostream &, const BinaryTree<int> &tree);

struct Put {
  using result_type = std::ostream &;

  std::ostream &operator()(std::ostream &out, int i) const {
    return out << i;
  }

  std::ostream &operator()(
      std::ostream &out,
      const BinaryTreeBranch<int> &branch) const {
    out << "B(";
    put(out, *branch.left) << ' ';
    put(out, *branch.right) << ')';
```