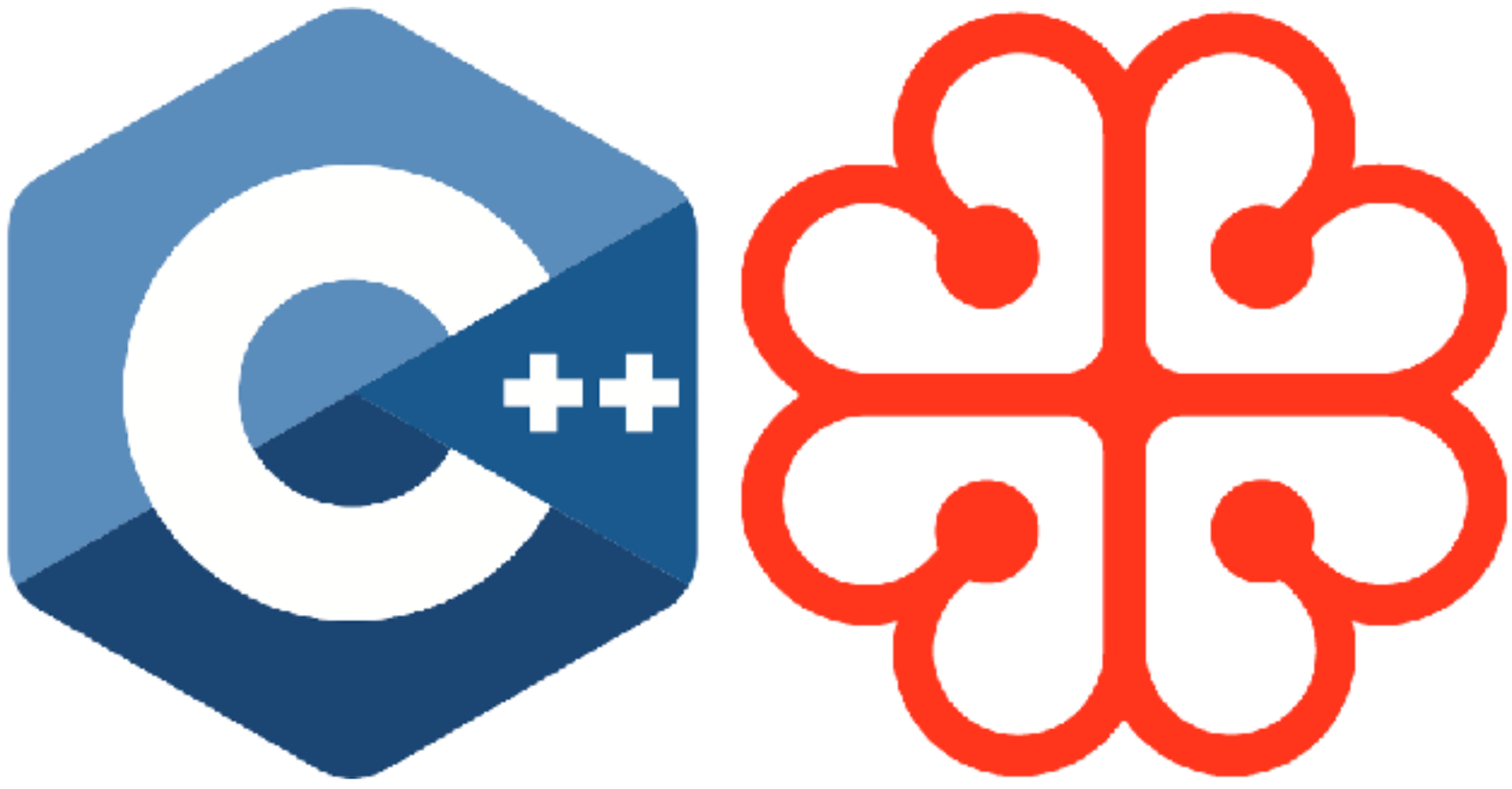
C++ Montréal



Gabriel Aubut-Lussier

Druide

Algebraic data types

Algebraic data types

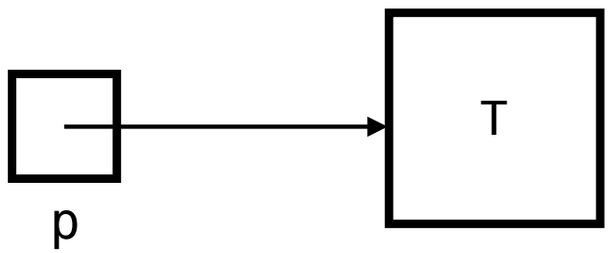
- This talk was put together in a single week
- It borrows a lot from various sources, there shall be references
- Borrowed slides have black border
- Source is in the bottom right corner

std::optional<T>

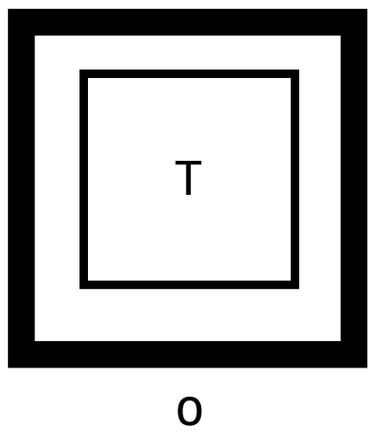
Conceptual Model

- Represents the notion of an optional object
- Models a discriminated union of `T` and `nullopt_t`
- `T *` wrapped up in a value type

```
T *p = nullptr;
p = new T(/* ... */);
```




```
optional<T> o;
o = T(/* ... */);
```



Quick Overview

```
optional<string> x = "hello";
assert(x);                 // `explicit operator bool`
assert(*x == "hello");  // `operator*` (unchecked access)

optional<string> y;
assert(!y.has_value());                    // `has_value`
assert(y.value_or("world") == "world");  // `value_or`

try {
  auto s = y.value();  // `value` (checked access)
} catch (const bad_optional_access&) {}

y = x;  // assignment
assert(y != nullopt);
assert(y == x);

// `optional` invokes `string::~string` correctly.
```

Use Cases

- Optional Return Value
- Optional Function Parameter
- Optional Data Member

Delta from Boost.Optional

	C++17	Boost 1.65.1
Empty Tag	`nullopt`	`none`
In-Place Constructor Tag	`in_place`	`in_place_init`
Forwarding Constructor	**Yes**	**No**
Conditional Explicit	Yes	No
Reference Type Support	No	Yes
has_value();	Yes	No
operator<<	No	Yes
`T* get_ptr();`	No	Yes

std::optional<T>

Some gotchas

Optional Function Parameter

Before	After
`void f(Light);`	`void f(optional<Light>);`
`void g(const Heavy &) {}`	`void g(const optional<Heavy> &);`
	This can be a copy!

Relational Operators

- `nullopt_t` compares less than any `T`
- All of the operators compare the engaged-ness of `optional`, then defer to the corresponding operator of `T`.
- Mixed comparisons are allowed.
  - `optional<T> == optional<U>`
  - `optional<T> == U`

Optionalizing

```
class Car {
  public:
  constexpr int MAX_SPEED = 300;  // in km/h

  // Returns the current speed in km/h.
  // Returns nullopt if the speedometer is non-functional.
  optional<int> get_speed() const;

  bool can_accelerate() const {
    return get_speed() < MAX_SPEED;
  }
};
```

Not a compile-time error!

`bool operator<(const optional<T> &, const U &);` is used, and `nullopt` is considered less than any `T`!

std::optional<T>

overhead

Sizeof optional

```
static_assert(sizeof(xmm_registers) == 256);
```

Sizeof optional

```
static_assert(sizeof(xmm_registers) == 256);
using xmm_registers_opt = std::optional<xmm_registers>;
static_assert(sizeof(xmm_registers_opt) == 264);
```

Compact optional

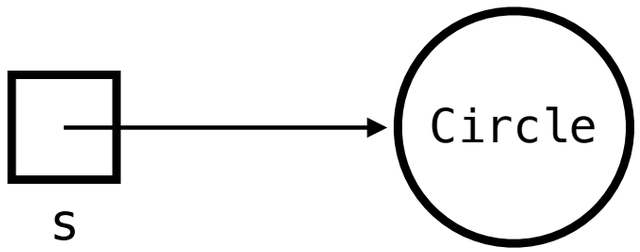
- No space overhead
- Sacrifice a single value
- Customize the sacrifice using a policy
- https://github.com/akrzemi1/markable

std::variant<T…>

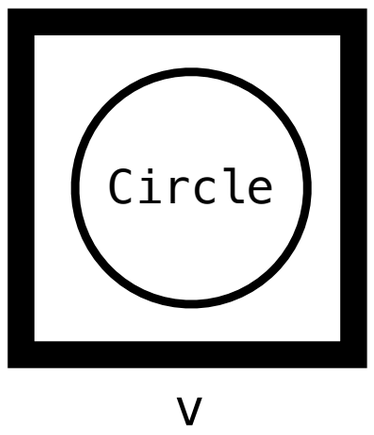
Conceptual Model

- A type-safe `union`
- Models a discriminated union of `Ts...`
- `AbstractBase *` wrapped up in a value type

```
Shape *s =
  new Circle(/* ... */);
```




```
variant<Circle, Square> v =
  Circle(/* ... */);
```



Quick Overview

```
variant<int, string> x = "hello";
assert(holds_alternative<string>(x));  // `holds_alternative`
assert(get<string>(x) == "hello");     // `get` (checked)

variant<int, string> y;  // default-constructs to `int`
assert(y.index() == 0);  // `index`
assert(*get_if<int>(&y) == 0);  // `get_if` (checked)

try {
  auto s = get<string>(y);  // `get` (checked)
} catch (const bad_variant_access &) {}

y = x;  // assignment
assert(holds_alternative<string>(y));
assert(y == x);

// `variant` invokes `string::~string` correctly.
```

Variant Visitation

```
struct Cat   { /* ... */ };
struct Dog   { /* ... */ };
struct Horse { /* ... */ };

using Animal = variant<Cat, Dog, Horse>;

string get_sound(const Animal &animal) {
  struct GetSound {
    string operator()(const Cat   &) const { return "meow";  }
    string operator()(const Dog   &) const { return "woof";  }
    string operator()(const Horse &) const { return "neigh"; }
  };
  return visit(GetSound{}, animal);
}
```

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>
```

Delta from Boost.Variant

	C++17	Boost 1.65.1
Empty Type	`monostate`	`blank`
Visitation	`visit`	`apply_visitor`
Non-throwing `get`	`get_if(&v)`	`get(&v)`
Dynamic-allocation during type-changing operation	**No**	**Yes**
valueless_by_exception	**Yes**	**No**
Reference Type Support	No	Yes
Index-based access	Yes	No
Special recursion support	No	Yes
In-place Constructors / `emplace`	Yes	No

std::variant<T…>

Some gotchas

Forwarding Constructor

Which alternative is constructed here?

```
variant<string, bool> v("abc");

template <typename T>
struct id { using type = T; };

struct FUN {
  id<string> operator()(string) const;
  id<bool>   operator()(bool)   const;

};

typename invoke_result_t<FUN, decltype("abc")>::type
```

EricWF
@Eric01

Following

variant<string, bool> v = "abc" initializes the second alternative. That's boolshit.

12:23 AM - 8 Feb 2017

28 Retweets **66** Likes

2 28 66

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.

Louis Dionne Programming and Categories, Oh My!

A mathematical intuition for empty variants and tuples

14 Jul 2015

$$\prod_{x \in \emptyset} x = 1$$

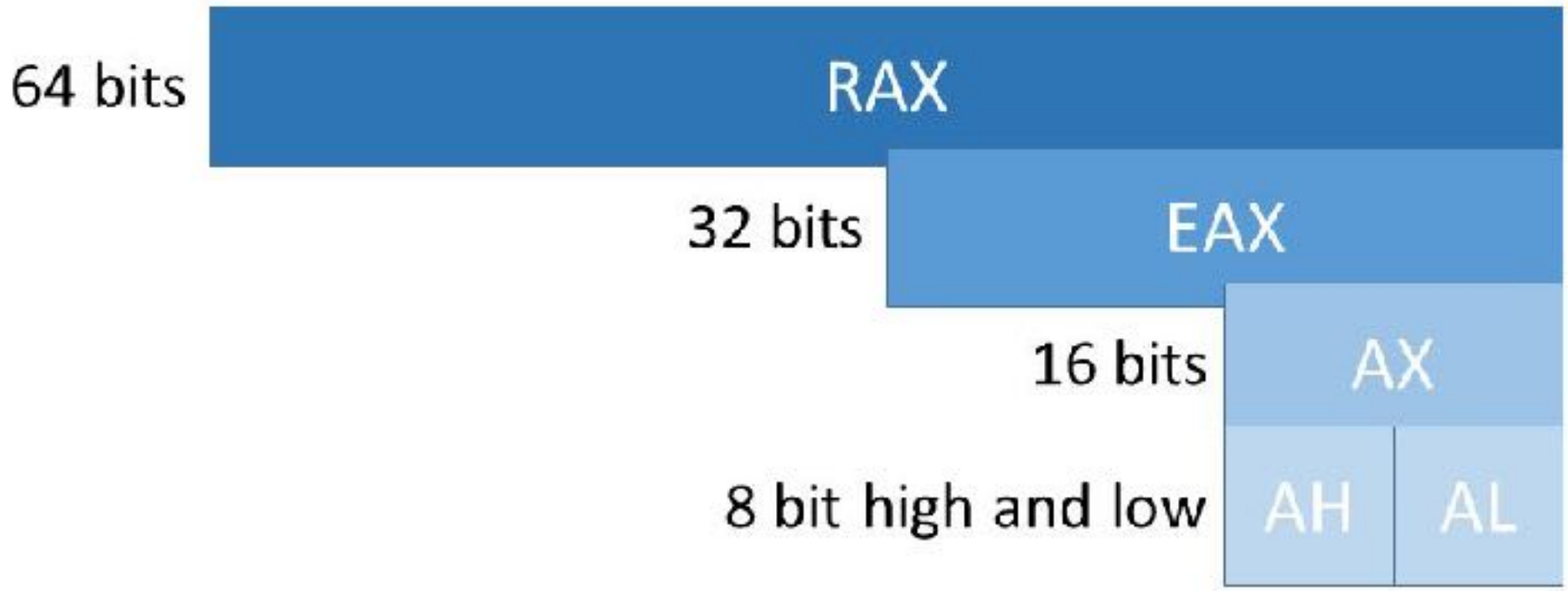
$$\sum_{x \in \emptyset} x = 0$$

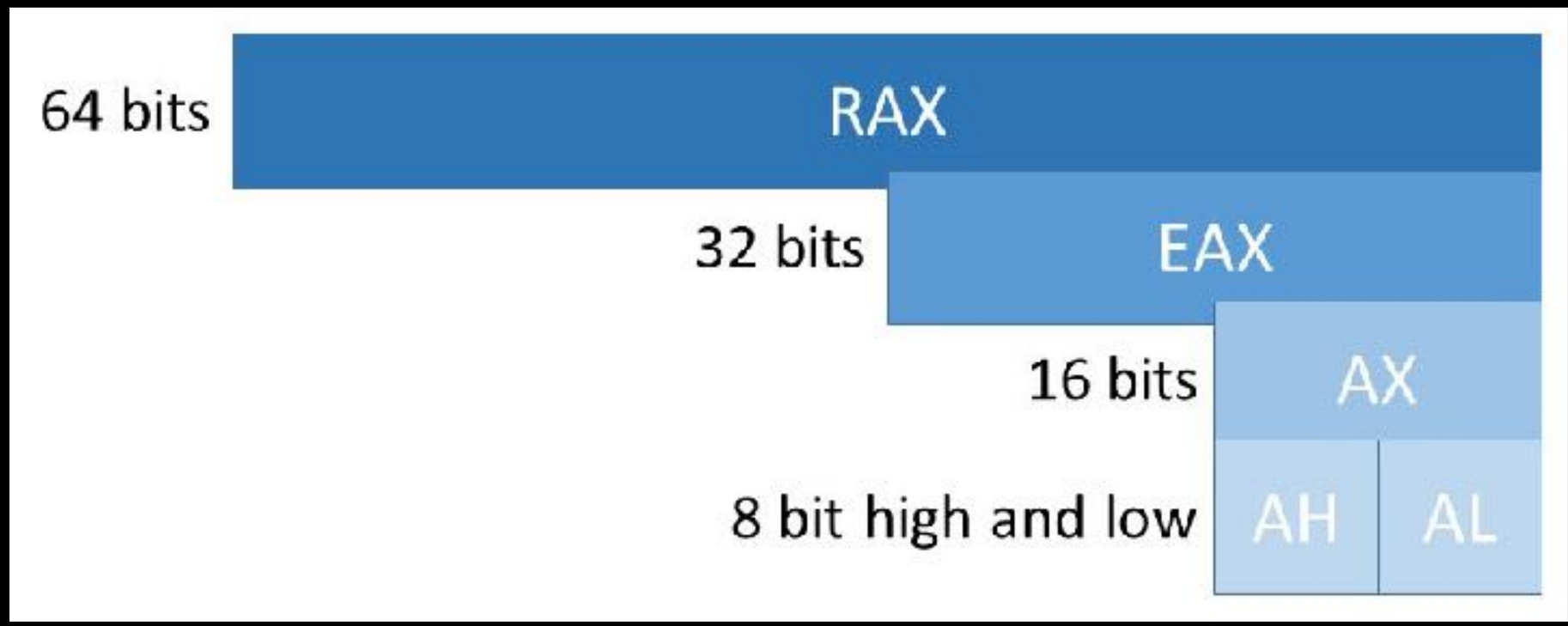
This is convenient in mathematics, and I also find it quite intuitive. Naively, I think the design of `variant<>` and `tuple<>` should follow this.

std::variant<T…>
overhead

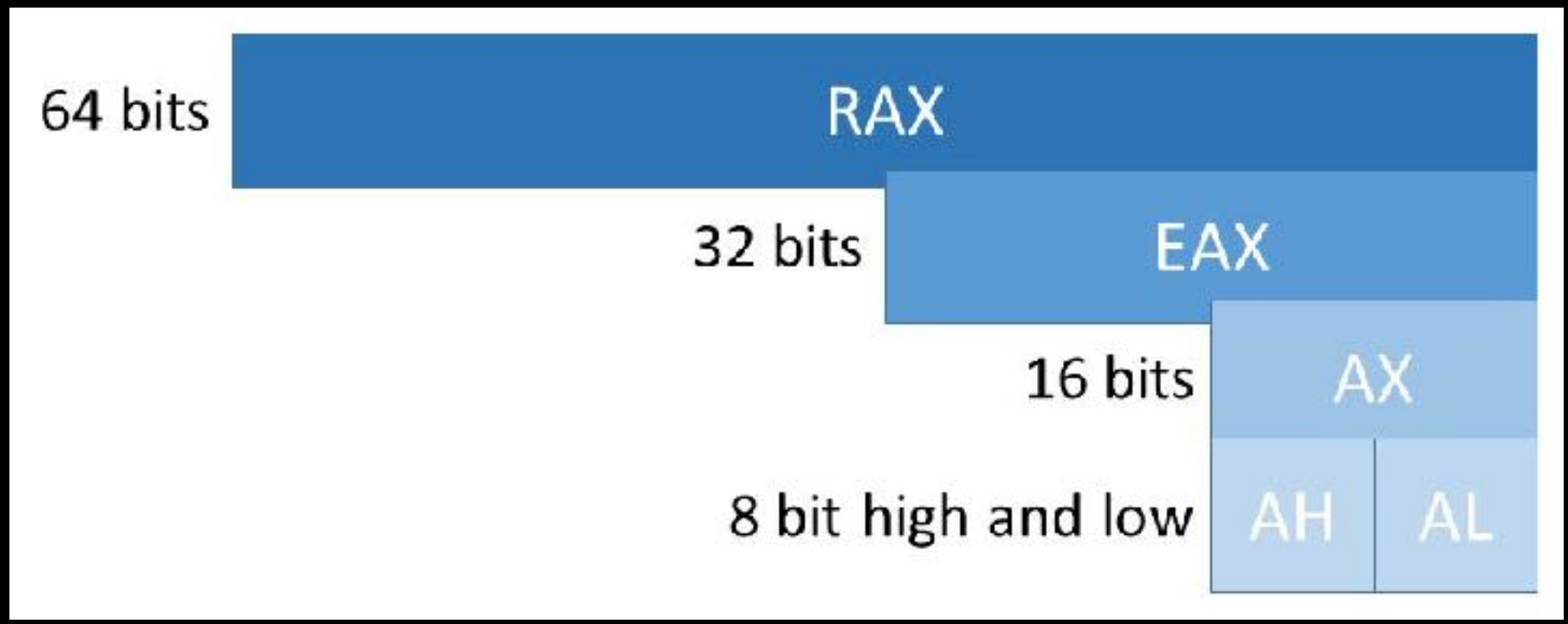
Register Layout

- x86 has gone through 16, 32, and 64 bit versions
- Registers can be addressed in whole or in part

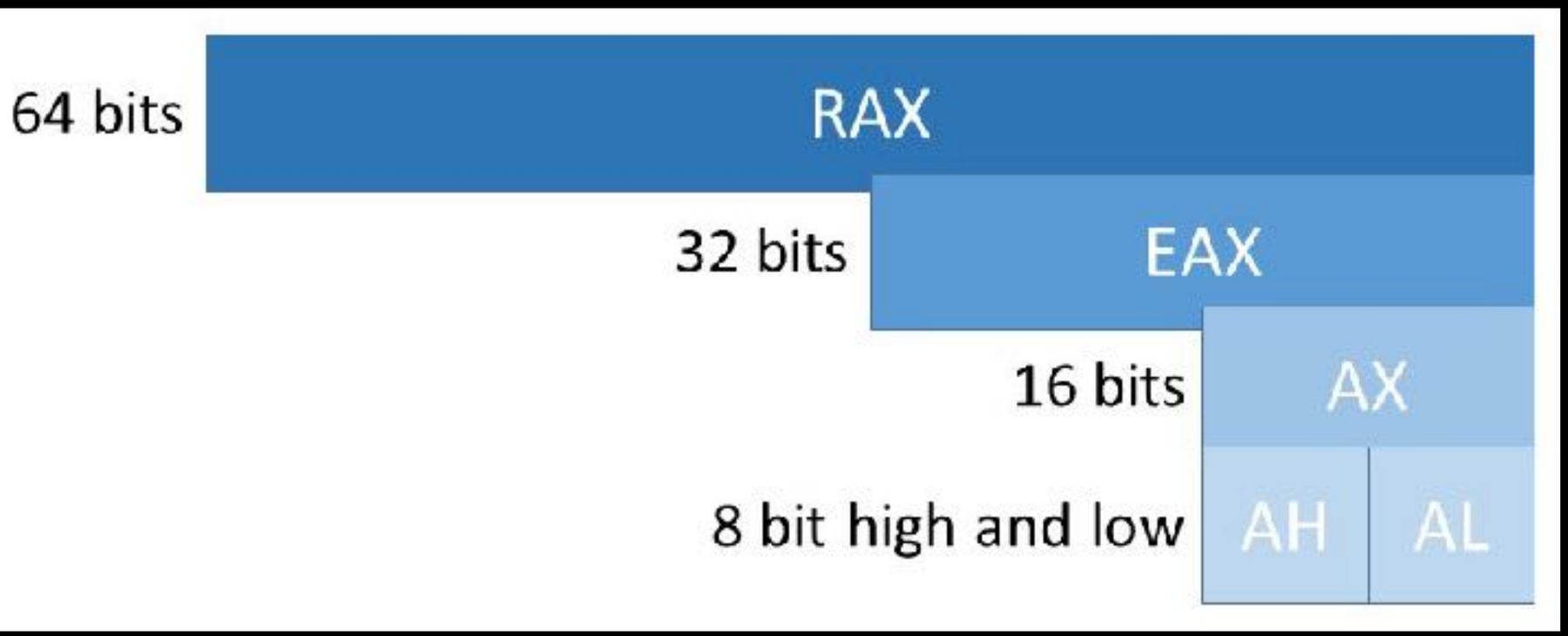



64 bits
RAX
32 bits
EAX
16 bits
AX
8 bit high and low
AH
AL

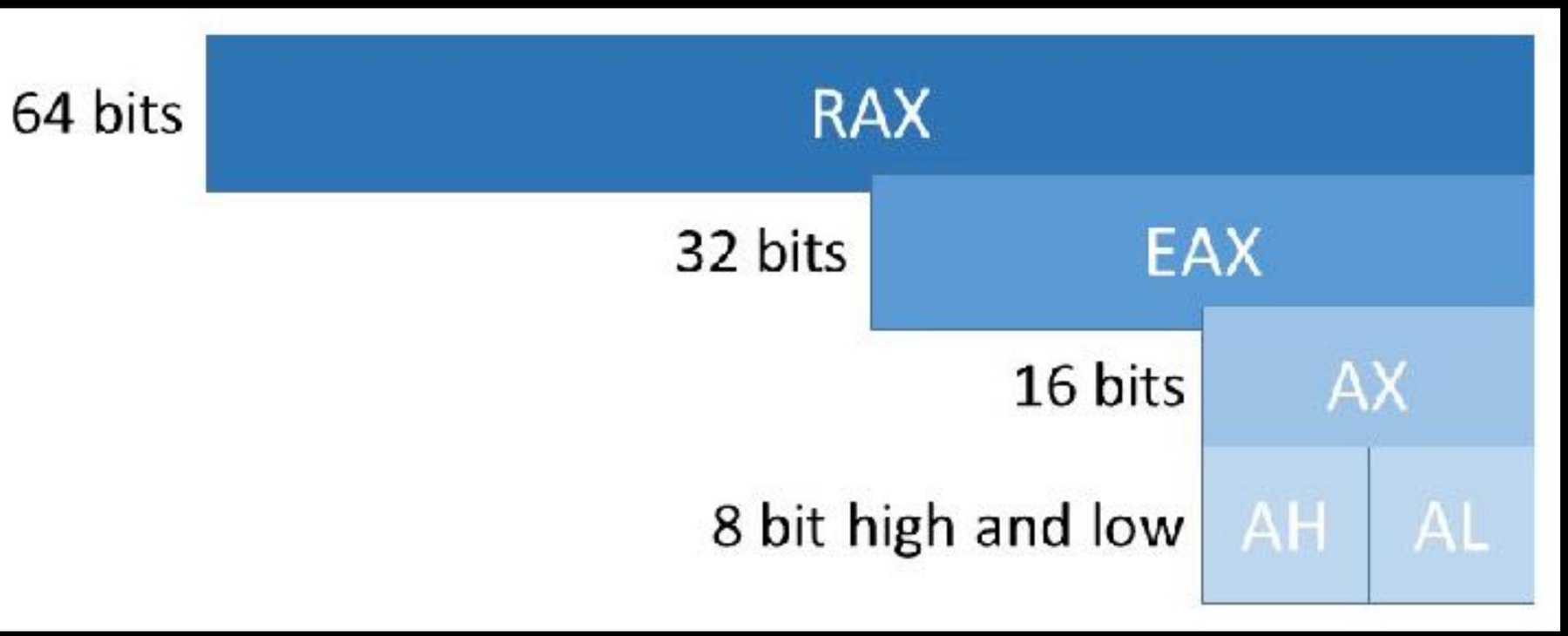
Hardware union!



Hardware union!



Hardware union!




```
union registers
{
  uint64_t rax;
  uint32_t eax;
  uint16_t ax;
  struct {
    uint8_t al;
    uint8_t ah;
  };
};
```

Sizeof union

```
union registers
{
   uint64_t rax;
   uint32_t eax;
   uint16_t ax;
   struct {
      uint8_t al;
      uint8_t ah;
   };
};
static_assert(sizeof(registers) == 8);
```

Sizeof variant

```
union registers
{
   uint64_t rax;
   uint32_t eax;
   uint16_t ax;
   struct {
      uint8_t al;
      uint8_t ah;
   };
};
static_assert(sizeof(registers) == 8);

using vregisters = std::variant<uint8_t, uint16_t,
     uint32_t, uint64_t>;
static_assert(sizeof(vregisters) == 16);
```

LEVEL 1

Types as sets of values

How many values?

```
bool;
```

How many values?

```
bool;
```

2 (`true` and `false`)

How many values?

```
char;
```

How many values?

```
char;
```

256

How many values?

```
void;
```

How many values?

```
void;
```

0

How many values?

```
struct Foo {};
```

How many values?

```
struct Foo {};
```

1

How many values?

```
enum CommonCharacterEncodings {
    UTF-8,
    ASCII,
    ISO-8859-1
};
```

How many values?

```
enum CommonCharacterEncodings {
    UTF-8,
    ASCII,
    ISO-8859-1
};
```

3

How many values?

```
template <typename T>
struct Foo {
    T t;
};
```

How many values?

```
template <typename T>
struct Foo {
    T t;
};
```

|T|

LEVEL 2

Aggregating Types

How many values?

```
std::pair<char, bool>;
```

How many values?

```
std::pair<char, bool>;
```

256 * 2 = 512

How many values?

```
struct Foo {
    char a;
    bool b;
};
```

How many values?

```
struct Foo {
    char a;
    bool b;
};
```

256 * 2 = 512

How many values?

```
std::tuple<bool, bool, bool>;
```

How many values?

```
std::tuple<bool, bool, bool>;
```

2 * 2 * 2 = 8

How many values?

```
template <typename T, typename U>
struct Foo {
    T t;
    U u;
};
```

How many values?

```
template <typename T, typename U>
struct Foo {
    T t;
    U u;
};
```

|T| * |U|

LEVEL 3

Alternating Types

How many values?

```
std::optional<char>;
```

How many values?

```
std::optional<char>;
```

256 + 1 = 257

How many values?

```
std::variant<char, bool>;
```

How many values?

```
std::variant<char, bool>;
```

256 + 2 = 258

How many values?

```
template <typename T, typename U>
struct Foo {
    std::variant<T, U> v;
};
```

How many values?

```
template <typename T, typename U>
struct Foo {
    std::variant<T, U> v;
};
```

|T| + |U|

Why care about this?

- Expect requirements to change over time
- Expect different developers to contribute code over time
- Expect complexity to keep growing

How is this helping?

- Eliminate errors before the program can even be executed
- Reduce the amount of runtime checks required
- Reduce the amount of tests required
- Reduce occasions for mistakes when evolving the code
- Reduce the context size surrounding a piece of code

Make illegal states unrepresentable

Yaron Minsky

```
enum class CitizenStatus {
  Minor,
  Adult,
  Senior,
  Deceased
}

class Citizen
{
  SocialSecurityNumber ssNumber;
  CitizenStatus status;
  Date birth;
  Date death;

  void checkInvariant()
  {
    bool hasDeathCertificate = death.isValid();
    bool isDeceased = status == CitizenStatus::Deceased;
    assert(hasDeathCertificate == isDeceased);
  }
};
```

```
enum class AgeClass {
  Minor,
  Adult,
  Senior
}

class Citizen
{
  class Deceased {
    Date death;
  };
  class Alive {
    AgeClass age;
  };
  using CitizenStatus = std::variant<Alive, Deceased>;
  SocialSecurityNumber ssNumber;
  Date birth;
  CitizenStatus status;
};
```

Correct by construction

If it doesn’t work, it doesn’t matter how fast it doesn’t work.

Mich Ravera

“There is no necessity…” to use sub-routines

…

“However it is usually advantageous”

“The reason for this will be discussed below”

THE USE OF SUB-ROUTINES IN PROGRAMMES

D. J. Wheeler

Cambridge & Illinois Universities

A sub-routine may perhaps best be described as a self-contained part of a programme, which is capable of being used in different programmes. It is an entity of its own within a programme. There is no necessity to compose a programme of a set of distinct sub-routines; for the programme can be written as a complete unit, with no divisions into smaller parts. However it is usually advantageous to arrange that a programme is comprised of a set of sub-routines, some of which have been made specially for the particular programme while others are available from a 'library' of standard sub-routines. The reasons for this will be discussed below.

When a programme has been made from a set of sub-routines the breakdown of the code is more complete than it would otherwise be. This allows the coder to concentrate on one section of a programme at a time without the overall detailed programme continually intruding. Thus the sub-routines can be more easily coded and the tested in isolation from the rest of the programme. When the entire programme has to be

h the foreknowledge that the

takes in the sub-routines is

t one order of magnitude below

sted portions of the programme:)

ub-routines exist for the major

hen the task of constructing the

easier to use a sub-routine which will meet the specifications with a small amount of manipulation than to make one specially for the purpose.

It should be pointed out that the preparation of a library sub-routine requires a considerable amount of work. This is much greater than the effort merely required to code the sub-routine in its simplest possible form. It will usually be necessary to code it in the library standard form and this may detract from its efficiency in time and space. It may be desirable to code it in such a manner that the operation is generalized to some extent. However, even after it has been coded and tested there still remains the considerable task of writing a description so that people not acquainted with the interior coding can nevertheless use it easily. This last task may be the most difficult.

Besides the organization of the individual sub-routines there remains the method of the general organization of the library. How are the sub-routines going to be stored? Are they going to be stored on punched paper tape or are they going to be available in the auxiliary store of the machine? Usually it will be found that it is not possible to write the sub-routines such that they may be put into arbitrary positions in the store-although in certain machines this is now possible. Usually some translation process will have to be

Here he says something courageous: although you don’t *need* subroutines, they are “usually advantageous”.

“allows the coder to concentrate on one section of a programme at a time without the overall detailed programme continually intruding”

THE USE OF SUB-ROUTINES IN PROGRAMMES

D. J. Wheeler

Cambridge & Illinois Universities

A sub-routine may perhaps best be described as a self-contained part of a programme, which is capable of being used in different programmes. It is an entity of its own within a programme. There is no necessity to compose a programme of a set of distinct sub-routines; for the programme can be written as a complete unit, with no divisions into smaller parts. However it is usually advantageous to arrange that a programme is comprised of a set of sub-routines, some of which have been made specially for the particular programme while others are available from a 'library' of standard sub-routines. The reasons for this will be discussed below.

When a programme has been made from a set of sub-routines the breakdown of the code is more complete than it would otherwise be. This allows the coder to concentrate on one section of a programme at a time without the overall detailed programme continually intruding. Thus the sub-

easier to use a sub-routine which will meet the specifications with a small amount of manipulation than to make one specially for the purpose.

It should be pointed out that the preparation of a library sub-routine requires a considerable amount of work. This is much greater than the effort merely required to code the sub-routine in its simplest possible form. It will usually be necessary to code it in the library standard form and this may detract from its efficiency in time and space. It may be desirable to code it in such a manner that the operation is generalized to some extent. However, even after it has been coded and tested there still remains the considerable task of writing a description so that people not acquainted with the interior coding can nevertheless use it easily. This last task may be the most difficult.

Besides the organization of the individual sub-routines there remains the method of the general organization of the library. How are the sub-routines going to be stored? Are they going to be stored on punched paper tape or are they going to be available in the auxiliary store of the machine? Usually it will be found that it is not possible to write the sub-routines such that they may be put into arbitrary positions in the store-although in certain machines this is now possible. Usually some translation process will have to be

“allows the coder to concentrate on one section at a time”
The rest of the programme doesn’t intrude.
You can concentrate on each part in solitude.
It’s like you would code, could code in a box.
This kind of stuff will blow off your socks!

"Thus the subroutines can be more **easily coded and tested in isolation** from the rest of the programme."

THE USE OF SUB-ROUTINES IN PROGRAMMES

D. J. Wheeler

Cambridge & Illinois Universities

A sub-routine may perhaps best be described as a self-contained part of a programme, which is capable of being used in different programmes. It is an entity of its own within a programme. There is no necessity to compose a programme of a set of distinct sub-routines; for the programme can be written as a complete unit, with no divisions into smaller parts. However it is usually advantageous to arrange that a programme is comprised of a set of sub-routines, some of which have been made specially for the particular programme while others are available from a 'library' of standard sub-routines. The reasons for this will be discussed below.

When a programme has been made from a set of sub-routines the breakdown of the code is more complete than it would otherwise be. This allows the coder to concentrate on one section of a pro- at a time without the overall detailed me continually intruding. Thus the sub- s can be more easily coded and the in isolation from the rest of the pro- . When the entire programme has to be it is with the foreknowledge that the nce of mistakes in the sub-routines is or at least one order of magnitude below f the untested portions of the programme!) library sub-routines exist for the major f a code then the task of constructing the easier to use a sub-routine which will meet the specifications with a small amount of manipulation than to make one specially for the purpose.

It should be pointed out that the preparation of a library sub-routine requires a considerable amount of work. This is much greater than the effort merely required to code the sub-routine in its simplest possible form. It will usually be necessary to code it in the library standard form and this may detract from its efficiency in time and space. It may be desirable to code it in such a manner that the operation is generalized to some extent. However, even after it has been coded and tested there still remains the considerable task of writing a description so that people not acquainted with the interior coding can nevertheless use it easily. This last task may be the most difficult.

Besides the organization of the individual sub-routines there remains the method of the general organization of the library. How are the sub-routines going to be stored? Are they going to be stored on punched paper tape or are they going to be available in the auxiliary store of the machine? Usually it will be found that it is not possible to write the sub-routines such that they may be put into arbitrary positions in the store- although in certain machines this is now possible. Usually some translation process will have to be

And "tested in isolation"!
This is a revelation!
Such a fascination.
It calls for a celebration.
This technique deserves adulation
... and replication.

- SEPARATION OF CONCERNS
- Isolation
- Focus without program intrusion
- (Re-use)

So enough of the sarcasm.
But there is this big chasm
between the promises of functions,
and how they are used by us bumpkins.
(Note that re-use is possibly the least important.
That at least is my argument.)

We all know that generally it is not a good idea to use global variables. This is basically the extreme of exposing side-effects (the global scope). Many of the programmers who don’t use global variables don’t realize that the same principles apply to fields, properties, parameters, and variables on a more limited scale: don’t mutate them unless you have a good reason.(…)

Wes Dyer

References

- Michael Park — Enhanced support for value semantics in C++ 17
https://www.youtube.com/watch?v=LmiDF2YheAM

- Ben Deane — Using Types Effectively
https://www.youtube.com/watch?v=ojZbFIQSdl8

- David Sankel — Variants - Past, present and future
https://www.youtube.com/watch?v=k3O4EKX4z1c

- Louis Dionne — A mathematical intuition for empty variants and tuples
http://ldionne.com/2015/07/14/empty-variants-and-tuples/

- Andrzej Krzemieński — Efficient optional values
https://github.com/akrzemi1/markable

References

- Tony Van Eerd — Postmodern C++ https://www.youtube.com/watch?v=QTLn3goa3A8
- Herb Sutter — Writing good C++… by default https://www.youtube.com/watch?v=hEx5DNLWGgA
- Herb Sutter — Leak-freedom in C++… by default https://www.youtube.com/watch?v=JfmTagWcqoE
- Sean Parent — Better code, data structures https://www.youtube.com/watch?v=sWgDk-o-6ZE