Types and data in C++17

FORTRAN

number of data	0	number of types ==> 1	N	*
0				
0/1				
1		T		
N		T[N]		
*				

Let's C ...

number of data	number of types ==> 0	1	N	*
0	`void`			
0/1				
1		`T`		
N		`T[N]`	`struct S{T1 t1;…TN tn;}`	
*				

C disaster area

number of data	number of types ==> 0	1	N	*
0	void			
0/1		T*	union U{T1 t1;…TN tn;}	void*
1		T		
N		T[N]	struct S{T1 t1;…TN tn;}	void*
*		dynamic T*	dynamic U*	dyn. void*

C++98

number of data		number of types ==>		
	0	1	N	*
0	void			
0/1		T*	union U{T1 t1;…TN tn;}	void*
1		T		
N		T[N] bitset<N>	struct S{T1 t1;…TN tn;} pair<T1,T2>	void*
*		dynamic T* vector<T>	dynamic U*	dyn. void*

C++11

number of data	number of types ==> 0	1	N	*
0	void			
0/1		T*	union U{T1 t1;…TN tn;}	void*
		unique_ptr<T> shared_ptr<T>		
1		T		
N		T[N] array<T,N> bitset<N>	struct S{T1 t1;…TN tn;} tuple<T1,…TN> pair<T1,T2>	void*
*		dynamic T*	dynamic U*	dyn. void*
		vector<T>		

C++17

number of data		number of types ==>		
	0	1	N	*
0	void			
0/1		T*	union U{T1 t1;…TN tn;}	void*
		unique_ptr<T> shared_ptr<T> optional<T>	variant<monostate,T1,…,TN>	any
1		T	variant<T1,…,TN>	
N		T[N] array<T,N> bitset<N>	struct S{T1 t1;…TN tn;} tuple<T1,…TN> pair<T1,T2>	void* array<any,N>
*		dynamic T*	dynamic U*	dyn. void*
		vector<T>	vector<variant<T1,…TN>>	vector<any>

How did we get here?

If we can supply a feature as a library, we should do so […] However, a library design should not be an excuse for inelegant interfaces, for irregular interfaces, for stylistic differences from built-in language features, or overelaboration. It is always easy to add another function to a class, so library components have a tendency to bloat. Note […] the dramatic differences in the interfaces to std::any, std::optional, and std::variant.

[…] std::variant, std::optional, and std::any have a long history as independent proposals. That wouldn’t be too bad if the reason was that significant improvements were added during the process.

B.Dawes/H.Hinnant/B.Stroustrup/D.Vandevoorde/M.Wong: P0939R0: Direction for ISO C++ (2018-02-10).

any optional variant of interface

```
any a
make_any<T>(args)

a = 42
a.emplace<T>(args)


a.has_value()
any_cast<T>(a)
any_cast<T*>(&a)


a.type()


a.reset()

// may throw
bad_any_cast
: bad_cast
```

```
optional<T> o
make_optional<T>(args)

o = 42
o.emplace(args)


if (o) | o.hasvalue()
   *o  | o.value()
       | o.valueOr(42)


o = null_opt
o.reset()

// may throw
bad_optional_access
: exception
```

```
variant<Ts...> v
variant<monostate,...>

v = 42
v.emplace<T>(args)
v.emplace<Index>(args)


get<T>(v)
get<Index>(v)
visit(Visitor, v)

v.index()
holds_alternative<T>(v)


// may throw
bad_variant_access
: exception
```

Ability to Learn & Teach?

- + --

visit a variant: have a visitor

```
struct MyVisitor
{
  void operator()(char c)        { std::cout << c; }
  void operator()(int i)         { std::cout << i; }
  void operator()(std::string s) { std::cout << s; }

  void operator()(auto x)        { std::cout << x; }
};

void demo1()
{
  using myvariant = std::variant<char, int, double, std::string>;
  auto v = std::vector<myvariant>{ 'C', "++", 17, "->", 20.01 };

  for (auto e : v)
    std::visit(MyVisitor{}, e);
}
```

context aware visitor

```
// http://en.cppreference.com/w/cpp/utility/variant/visit

template<class... Ts>
struct overloaded : Ts... { using Ts::operator()...; };
template<class... Ts> overloaded(Ts...) -> overloaded<Ts...>;

void demo2()
{
  using myvariant = std::variant<char, int, double, std::string>;
  auto v = std::vector<myvariant>{ 'C', "++", 17, "->", 20.01 };

  for (auto e: v)
  {
    std::visit(overloaded
    { [&](char c)        { std::cout << c; },
      [&](int i)         { std::cout << i; },
      [&](std::string s) { std::cout << s; },
      [&](auto x)        { std::cout << x; }
    }, e);
  }
}
```