

Optional types

Victor Eijkhout, Susan Lindsey

Fall 2023

last formatted: October 28, 2023

Optional

1. Result or error

Dealing with computations that can fail:

```
bool MaybeSqrt( float &x ) {  
    if ( x>=0 ) {  
        x = std::sqrt(x); return true;  
    } else return false;  
}
```

Inelegant. Better solution:

```
optional<float> MaybeSqrt( float x ) { /* .... */ }
```

'result or no-such-thing

2. Optional results

The most elegant solution to 'a number or an error' is to have a single quantity that you can query whether it's valid.

```
#include <optional>
using std::optional;
```

```
1  // union/optroot.cpp
2  optional<float> MaybeRoot(float x) {
3      if (x<0)
4          return {};
5      else
6          return std::sqrt(x);
7  };
8      /* ... */
9      for ( auto x : {2.f,-2.f} )
10         if ( auto root = MaybeRoot(x) ; root.has_value() )
11             cout << "Root is " << root.value() << '\n';
12         else
13             cout << "could not take root of " << x << '\n';
```

Exercise 1

Write a function *first_factor* that optionally returns the smallest factor of a given input.

```
// primes/optfactor.cpp
auto factor = first_factor(number);
if (factor.has_value())
    cout << "Found factor: " << factor.value() << '\n';
else
    cout << "Prime number\n";
```

3. Mistake

Trying to take the value for something that doesn't have one leads to a `bad_optional_access` exception:

Code:

```
1 // union/optional.cpp
2 optional<float> maybe_number = {};
3 try {
4     cout << maybe_number.value() <<
        '\n';
5 } catch
    (std::bad_optional_access) {
6     cout << "failed to get value\n";
7 }
```

Output:

failed to get value

4. Expected

Expect double, return info string if not:

```
std::expected<double,string>
    square_root( double x ) {
    auto result = sqrt(x);
    if (x<0)
    return
        std::unexpected("negative");
    else if
        (x<limits<double>::min())
    return
        std::unexpected("underflow");
    else return result;
}
```

```
auto root = square_root(x);
if (x)
    cout << "Root=" <<
        root.value() << '\n';
else if (root.error()==/* et
        cetera */ )
    /* handle the problem */
```

Variant

5. Square root with variant

```
// union/optroot.cpp
#include <variant>
using std::variant,
    std::get_if;
/* ... */
variant<bool,float>
    RootVariant(float x) {
    if (x<0)
        return false;
    else
        return std::sqrt(x);
};
```

```
// union/optroot.cpp
for ( auto x : {2.f,-2.f} ) {
    auto okroot =
        RootVariant(x);
    auto root =
        get_if<float>(&okroot);
    if ( root )
        cout << "Root is " <<
            *root << '\n';
    auto nope =
        get_if<bool>(&okroot);
    if (nope)
        cout << "could not take
            root of " << x << '\n';
}
```

6. More variant examples

Illustrating the usage:

```
1 // union/intdoublestring.cpp
2  variant<int,double,string> union_ids;
```

We can use the `index` function to see what variant is used (0,1,2 in this case) and get the value accordingly:

```
1 // union/intdoublestring.cpp
2  union_ids = 3.5;
3  switch ( union_ids.index() ) {
4  case 1 :
5      cout << "Double case: " << std::get<double>(union_ids) << '\n';
6  }
```

7. Variant methods

```
1 // union/intdoublestring.cpp
2  variant<int,double,string> union_ids;
```

Get the index of what the variant contains:

```
1 // union/intdoublestring.cpp
2  union_ids = 3.5;
3  switch ( union_ids.index() ) {
4  case 1 :
5      cout << "Double case: " << std::get<double>(union_ids) << '\n';
6  }
```

```
1 // union/intdoublestring.cpp
2  union_ids = "Hello world";
3  if ( auto union_int = get_if<int>(&union_ids) ; union_int )
4      cout << "Int: " << *union_int << '\n';
5  else if ( auto union_string = get_if<string>(&union_ids) ;
6            union_string )
7      cout << "String: " << *union_string << '\n';
```

(Takes pointer to variant, returns pointer to value)

Exercise 2

Write a routine that computes the roots of the quadratic equation

$$ax^2 + bx + c = 0.$$

The routine should return two roots, or one root, or an indication that the equation has no solutions.

Code:

```
1 // union/quadratic.cpp
2 for ( auto coefficients :
3     { quadratic{.a=2.0,
4         .b=1.5, .c=2.5},
5       quadratic{.a=1.0,
6         .b=4.0, .c=4.0},
7       quadratic{.a=2.2,
8         .b=5.1, .c=2.5}
9     } ) {
10     auto result =
11         compute_roots(coefficients);
```

Output:

```
With a=2 b=1.5 c=2.5
No root
With a=2.2 b=5.1 c=2.5
Root1: -0.703978 root2:
        -1.6142
With a=1 b=4 c=4
Single root: -2
```

8. Implementation of quadratic polynomial

We represent the polynomial

$$ax^2 + bx + c$$

as

```
// union/quadlib.hpp  
struct quadratic {  
    double a,b,c;  
};
```

Exercise 3

Write a function

```
double discriminant( quadratic coefficients );
```

that computes $b^2 - 4ac$, and test:

```
1 // union/quadtest.cpp
2 TEST_CASE( "discriminant" ) {
3     quadratic one{0., 2.5, 0.};
4     REQUIRE( discriminant( one ) == Catch::Approx(6.25) );
5     quadratic two{1., 0., 1.5};
6     REQUIRE( discriminant( two ) == Catch::Approx(-6.) );
7     quadratic three{.1, .1, .1*.5};
8     REQUIRE( discriminant( three ) == Catch::Approx(-.01) );
9 }
```

Exercise 4

Write a function

```
bool discriminant_zero( quadratic coefficients );
```

that passes the test

```
1 // union/quadtest.cpp
2 quadratic coefficients{a,b,c};
3 d = discriminant( coefficients );
4 z = discriminant_zero( coefficients );
5 INFO( a << ", " << b << ", " << c << " d=" << d );
6 REQUIRE( z );
```

Using for instance the values:

```
a = 2; b = 4; c = 2;
a = 2; b = sqrt(40); c = 5; // !!!
a = 3; b = 0; c = 0.;
```

Exercise 5

Write the function *simple_root* that returns the single root. For confirmation, test

```
1 // union/quadtest.cpp
2 auto r = simple_root(coefficients);
3 REQUIRE( evaluate(coefficients,r)==Catch::Approx(0.).margin(1.e-14) );
```


Exercise 6

Write a function that returns the two roots as a `indexcstdpair`:

```
pair<double,double> double_root( quadratic coefficients );
```

Test:

```
1 // union/quadtest.cpp
2 quadratic coefficients{a,b,c};
3 auto [r1,r2] = double_root(coefficients);
4 auto
5     e1 = evaluate(coefficients,r1),
6     e2 = evaluate(coefficients,r2);
7 REQUIRE( evaluate(coefficients,r1)==Catch::Approx(0.).margin(1.e-14)
8         );
9 REQUIRE( evaluate(coefficients,r2)==Catch::Approx(0.).margin(1.e-14)
10         );
```

Exercise 7

Write a function

```
variant< bool,double, pair<double,double> >  
    compute_roots( quadratic coefficients);
```

Test:

```
1 // union/quadtest.cpp  
2 TEST_CASE( "full test" ) {  
3     double a,b,c; int index;  
4     SECTION( "no root" ) {  
5         a=2.0; b=1.5; c=2.5;  
6         index = 0;  
7     }  
8     SECTION( "single root" ) {  
9         a=1.0; b=4.0; c=4.0;  
10        index = 1;  
11    }  
12    SECTION( "double root" ) {  
13        a=2.2; b=5.1; c=2.5;  
14        index = 2;  
15    }  
16    quadratic  
17        coefficients{.a=a,.b=b,.c=c};  
18    auto result =  
19        compute_roots(coefficients);  
20    REQUIRE( result.index()==index );  
21 }
```

Exercise 8

Instead of a `bool`, return a *monostate*.