Optional types

Victor Eijkhout, Susan Lindsey

Fall 2023

last formatted: October 17, 2023

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 ) { /* .... */ }
```

2. Create optional

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

optional<float> f {
  if (something)
  // result if success
  return 3.14;
  else
  // indicate failure
  return {};
}
```

3. 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>
```

```
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';
```

4. Optional value

```
auto maybe_x = f();
if (f.has_value())
// do something with f.value();
```

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
```

5. 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 */
```