If it ain't one type it's another

Victor Eijkhout, Susan Lindsey

Fall 2022

last formatted: March 27, 2023

Tuples

1. Example for this lecture

Example: compute square root, or report that the input is negative

2. Returning two things

Simple solution:

```
bool RootOrError(float &x) {
  if (x<0)
    return false;
  else
    x = sqrt(x);
  return true;
};
  /* ... */
  for ( auto x : {2.f,-2.f} )
    if (RootOrError(x))
      cout << "Root is " << x << '\n';
    else
      cout << "could not take root of " << x << '\n';
```

Other solution: tuples

3. Function returning tuple

How do you return two things of different types?

```
#include <tuple>
using std::make_tuple, std::tuple;

tuple<bool,float> maybe_root1(float x) {
  if (x<0)
    return make_tuple<bool,float>(false,-1);
  else
    return make_tuple<bool,float>(true,sqrt(x));
};

```

(not the best solution for the ‘root’ code)

4. Returning tuple with type deduction

Return type deduction:

```
auto maybe_root1(float x) {
  if (x<0)
    return make_tuple
      <bool,float>(false,-1);
  else
    return make_tuple
      <bool,float>
        (true,sqrt(x));
};
```

Alternative:

```
tuple<bool,float>
    maybe_root2(float x) {
  if (x<0)
    return {false,-1};
  else
    return {true,sqrt(x)};
};
```

Note: use *pair* for *tuple* of two.

5. Catching a returned tuple

The calling code is particularly elegant:

Code:

```
auto [succeed,y] = maybe_root2(x);
if (succeed)
  cout << "Root of " << x
       << " is " << y << '\n';
else
  cout << "Sorry, " << x
       << " is negative" << '\n';
```

Output:

```
Root of 2 is 1.41421
Sorry, -2 is negative
```

This is known as structured binding.

6. C++11 style tuples

```
#include <tuple>

std::tuple<int,double,char> id = \
    std::make_tuple<int,double,char>( 3, 5.12, 'f' );
    // or:
    std::make_tuple( 3, 5.12, 'f' );
double result = std::get<1>(id);
std::get<0>(id) += 1;

// also:
std::pair<int,char> ic = make_pair( 24, 'd' );
```

Annoyance: all that ‘get’ting.

Optional

7. 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>
```

```
optional<float> MaybeRoot(float x) {
  if (x<0)
    return {};
  else
    return sqrt(x);
};
  /* ... */
  for ( auto x : {2.f,-2.f} )
    if ( auto root = MaybeRoot(x) ; root.has_value() )
      cout << "Root is " << root.value() << '\n';
    else
      cout << "could not take root of " << x << '\n';
```

8. Create optional

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

optional<float> f {
  if (something) return 3.14;
  else return {};
}
```

Expected (C++23)

9. 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 */
```

Variants

10. Variant

- Tuple of value and bool: we really need only one
- variant: it *is* one or the other
- You can set it to either, test which one it is.

11. Variant methods

```
variant<int,double,string> union_ids;
```

Get the index of what the variant contains:

```
union_ids = 3.5;
switch ( union_ids.index() ) {
case 1 :
  cout << "Double case: " << std::get<double>(union_ids) << '\n';
}
```

```
union_ids = "Hello world";
if ( auto union_int = get_if<int>(&union_ids) ; union_int )
  cout << "Int: " << *union_int << '\n';
else if ( auto union_string = get_if<string>(&union_ids) ; union_string )
  cout << "String: " << *union_string << '\n';
```

(Takes pointer to variant, returns pointer to value)

Exercise 1

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

```
auto factor = first_factor(number);
if (factor.has_value())
  cout << "Found factor: " << factor.value() << '\n';
```

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:

```
for ( auto coefficients :
      { make_tuple(2.0, 1.5, 2.5),
        make_tuple(1.0, 4.0, 4.0),
        make_tuple(2.2, 5.1, 2.5)
       } ) {
  auto result =
     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
```