If it ain't one type it's another

Victor Eijkhout, Susan Lindsey

Fall 2022

last formatted: October 13, 2022

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. 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.

4. Function returning tuple

Return type deduction:

```
#include <tuple>
using std::make_tuple,
    std::tuple;
  /* ... */
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)};
};
```

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
[stl] tuple:

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

This is known as structured binding.

6. Tuple solution

```
#include <tuple>
using std::tuple, std::pair;
  /* ... */
pair<bool,float> RootAndValid(float x) {
  if (x<0)
    return {false,x};
  else
    return {true,sqrt(x)};
};
  /* ... */
  for ( auto x : {2.f,-2.f} )
    if ( auto [ok,root] = RootAndValid(x) ; ok )
      cout << "Root is " << root << '\n';
    else
      cout << "could not take root of " << x << '\n';
```

Variants

7. 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.

8. Square root with variant

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

```
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';
}
```

9. More variant methods

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

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';
```

10. 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';
```

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

```
[union] quadratic:

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
```