If it ain't one type it's another

Victor Eijkhout, Susan Lindsey

Fall 2024

last formatted: November 7, 2024

Tuples

1. Examples for this lecture

Example: two roots of a quadratic polynomial

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

Example: roots of a quadratic polynomial: zero or one or two.

2. Returning two things

Simple solution:

```
1 // union/optroot.cpp
2 bool RootOrError(float &x) {
3   if (x<0)
4     return false;
5   else
6     x = std::sqrt(x);
7   return true;
8 };
9     /* ... */
10  for ( auto x : {2.f,-2.f} )
11    if (RootOrError(x))
12      cout << "Root is "
13           << x << '\n';
14    else
15      cout
16        << "could not take root of "
17        << x << '\n';
```

Other solution: tuples

3. Tuple (de)construction

Construct:

```
tuple<int,char,double> icd;
auto icd = make_tuple(5,'d',3.14);
```

Deconstruct ('structured binding'):

```
auto [i,c,d] = icd;
// or by reference:
auto& [i,c,d] = icd;
```

4. Pairs

```
pair<char,float> cf = make_pair('a',1.1f);

auto [c,f] = cf;
// or
auto c=cf.first; auto f=cf.second;
```

5. Function returning tuple

How do you return two things of different types?

```
1  #include <tuple>
2  using std::make_tuple, std::tuple;
3
4  tuple<bool,float> maybe_root1(float x) {
5    if (x<0)
6      return make_tuple<bool,float>(false,-1);
7    else
8      return make_tuple<bool,float>(true,sqrt(x));
9  };
10
```

(not the best solution for the 'root' code)

6. Returning tuple with type deduction

Return type deduction:

```
1 // stl/tuple.cpp
2 auto maybe_root1(float x) {
3   if (x<0)
4     return make_tuple
5       <bool,float>(false,-1);
6   else
7     return make_tuple
8       <bool,float>
9         (true,sqrt(x));
10 };
```

Alternative:

```
1 // stl/tuple.cpp
2 tuple<bool,float>
3     maybe_root2(float x) {
4   if (x<0)
5     return {false,-1};
6   else
7     return {true,sqrt(x)};
8 };
```

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

7. Catching a returned tuple

The calling code is particularly elegant:

Code:

```
1 // stl/tuple.cpp
2 auto [succeed,y] = maybe_root2(x);
3 if (succeed)
4   cout << "Root of " << x
5        << " is " << y << '\n';
6 else
7   cout << "Sorry, " << x
8        << " is negative" << '\n';
```

Output:

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

This is known as structured binding.

8. Discriminant

Discriminant of the quadratic polynomial

Definition:

```
1 // union/abctuple.cpp
2 double discriminant
3     ( quadratic q ) {
4   auto [a,b,c] = q;
5   return b*b-4*a*c;
6 }
```

Use:

```
1 // union/abctuple.cpp
2 auto d = discriminant( sunk );
3 cout << "discriminant: "
4      << d << '\n';
```

Exercise 1

Write the function *discriminant* that makes this code work:

```
auto roots = abc_roots( sunk );
auto [xplus,xminus] = roots;
cout << xplus << "," << xminus << '\n';
```

9. 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

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>
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 };
```

11. Create optional

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

12. Testing and getting value

Two ways:

```
1 // union/optroot.cpp
2 for ( auto x : {2.f,-2.f} )
3   if ( auto root = MaybeRoot(x)
       ; root.has_value() )
4     cout << "Root is "
5          << root.value() <<
       '\n';
6   else
7     cout
8       << "could not take root
       of "
9       << x << '\n';
```

```
1 // union/optroot.cpp
2 for ( auto x : {2.f,-2.f} )
3   if ( auto root = MaybeRoot(x) ;
       root )
4     cout << "Root is "
5          << *root << '\n';
6   else
7     cout
8       << "could not take root of "
9       << x << '\n';
```

Expected (C++23)

13. Expected

Expect double, return info string if not:

Function returning expected:

```
1 // union/expected.cpp
2 std::expected<double,string>
3 square_root( double x ) {
4   auto result = sqrt(x);
5   if (x<0)
6     return
       std::unexpected("negative");
7   else if
       (x<limits<double>::min())
8     return
       std::unexpected("underflow");
9   else return result;
10 }
```

In Use:

```
1 // union/expected.cpp
2 auto root = square_root(x);
3 if (x)
4   cout << "Root=" << root.value()
       << '\n';
5 else if (root.error()==/* et
       cetera */ )
6   /* handle the problem */
```

Variants

14. **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.

15. 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) ;
        union_string )
6   cout << "String: " << *union_string << '\n';
```

(Takes pointer to variant, returns pointer to value)

Exercise 2

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

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

Exercise 3

Continue the 'abc' exercise above and write a function that returns optionally a string, saying 'one' or 'two' for the number of roots:

```
1 // union/abctuple.cpp
2 auto num_solutions = how_many_roots(sunk);
3 if ( not num_solutions.has_value() )
4   cout << "none\n";
5 else
6   cout << num_solutions.value() << '\n';
```

Exercise 4

```
1 // union/abctuple.cpp
2 auto root_cases = abc_cases( sunk );
3 switch (root_cases.index()) {
4 case 0 : cout << "No roots\n"; break;
5 case 1 : cout << "Single root: " << get<1>(root_cases); break;
6 case 2 : {
7   auto xs = get<2>(root_cases);
8   auto [xp,xm] = xs;
9   cout << "Roots: " << xp << "," << xm << '\n';
10 } ; break;
11 }
```