

If it ain't one type it's another

Victor Eijkhout, Susan Lindsey

Fall 2024

last formatted: November 11, 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

Headers

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

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

A `pair` is the same as a `tuple` of two elements:

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

but there are two extra data members *first* *second*

```
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?

```
#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)

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. Exercises for the 'abc' formula

Introduce synonym:

```
1 // union/abctuple.cpp
2 using quadratic = tuple<double,double,double>;
3     /* ... */
4 // polynomial:  $x^2 - 2$ 
5 auto sunk = quadratic(1.,0.,-2);
```

You can base this off the file `abctuple.cpp` in the repository

9. 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 `abc_roots` that makes this code work:

```
1 // union/abctuple.cpp
2 auto roots = abc_roots( sunk );
3 auto [xplus,xminus] = roots;
4 cout << xplus << "," << xminus << '\n';
```

10. 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

11. 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 };
```

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);
4         root.has_value() )
5         cout << "Root is "
6             << root.value()
7             << '\n';
8 else
9     cout
10    << "could not take root
11    of "
12    << x << '\n';
```

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


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,

```
1 // union/abctuple.cpp
2 optional<string> how_many_roots( quadratic q );
```

saying 'one' or 'two' if there are real 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';
```

Expected (C++23)

13. Expected

Expect double, return info string if not:

Function returning expected: In Use:

```
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 }
```

```
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. Square root with variant

Return square root or false result:

```
1 // union/optroot.cpp
2 #include <variant>
3 using std::variant, std::get_if;
4     /* ... */
5 variant<bool,float>
6     RootVariant(float x) {
7     if (x<0)
8         return false;
9     else
10        return std::sqrt(x);
11 };
```

```
1 // union/optroot.cpp
2 for ( auto x : {2.f,-2.f} ) {
3     auto okroot = RootVariant(x);
4     auto root =
5         get_if<float>(&okroot);
6     if ( root )
7         cout << "Root is "
8             << *root << '\n';
9     auto nope =
10        get_if<bool>(&okroot);
11    if (nope)
12        cout
13            << "could not take root of "
14            << x << '\n';
15 }
```

16. More variant examples

Illustrating the usage:

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

The index return 0,1,2, and we 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 }
```


17. 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 4

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.

```
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 }
```

18. Monostate

How did you handle the case of no roots?

Possibility:

```
return std::monostate{};
```

'boolean with only one value'