An overview of C++17's structured bindings

This talk will cover a new feature introduced in C++17 to declare and assign multiple variables from a tuple or struct. This new feature has surprising depth which we’ll explore throughout the talk, including discussions on API design, error handling, and performance.

C++17

Herb Sutter :

> C++17 will pervasively change the way we write C++ code, just as C++11 did. As these and other new features become available, we're going to see new code using structured bindings, if/switch scope variables, string_view, optional, any, variant, Parallel STL, and more all over the place.
>
> Here's my personal litmus test for whether we've changed the way people program: Just as you can take a look at a screenful of code and tell that it's C++11 (not C++98), if you can look at a screenful of code and tell that it's C++17, then we've changed the way we program C++. I think C++17 will meet that bar.

(https://herbsutter.com/2016/06/30/trip-report-summer-iso-c-standards-meeting-oulu/)

Goals

Help you understand how this can transform your code

Make you comfortable using and recommending structured bindings in your, and your organization's code

My job was easy, P0144R2 is a well written proposal, easy to read and understand

Agenda

Intro

- Syntactic sugar
- Quick examples

Implementation

- What’s the compiler up to?

Use cases

- Pairs well with selection statements with initializers
- No more out variables

Performance

- Expert or novice feature?

Advanced uses

- Enabling structured bindings for your types

Syntactic sugar

```
#include <tuple>
#include <stdio.h>

std::tuple<int, double> square(int num) {
    return std::make_tuple(num, num * num);
}

void example(int i)
{
    auto [input, output] = square(i);
    printf("%d*%d = %f\n", input, input, output);
}
```

asm

Tuple and tie

std::tuple

Heterogeneous container

std::tie

Creates a tuple of lvalue references

Pair	Tuple
`std::pair<int, double> p; printf("%d %fn", p.first, p.second);`	`std::tuple<int, double> t; printf("%d %fn", std::get<0>(t), std::get<1>(t));`

The old way

```cpp
#include <tuple>
#include <stdio.h>

std::tuple<int, double> square(int num) {
    return std::make_tuple(num, num * num);
}

void example(int i)
{
    int input;
    double output;
    std::tie(input, output) = square(i);
    printf("%d*%d = %f\n", input, input, output);
}
```

asm

What's wrong with the old way?

- Separately declared variables, maybe uninitialized
- Possibly wasted work doing default initialization
- Default initialization sometimes a problem

Example 1

```
void foo() {
    float points[3] = {1, 2, 3};
    auto [x, y, z] = points;
    printf("x %f y %f z %f\n", x, y, z);
}
```

Example 2

```cpp
struct S {
    float x, y, z;
};

void foo() {
    S s{1.0f, 2.0f, 3.0f};
    auto [x, y, z] = s;
    printf("x %f y %f z %f\n", x, y, z);
}
```

Example 3

```
std::unordered_map<int, int> map;

void foo() {
    for (auto [key, value] : map)
        printf("%d:%d\n", key, value);
}
```

What's the compiler up to? part 1

```
float points[3] = {1, 2, 3};
auto [x, y, z] = points;
↓
auto __a = expression;
auto& x = __a[0];
auto& y = __a[1];
auto& z = __a[2];
---------------------------------
S s{1.0f, 2.0f, 3.0f};
auto [x, y, z] = s;
↓
auto __a = expression;
auto& x = __a.x;
auto& y = __a.y;
auto& z = __a.z;
```

What's the compiler up to? part 2

```
auto & [x, y, z] = expression;

↓

auto & __a = expression;
auto& x = __a.mem1;
auto& y = __a.mem2;
auto& z = __a.mem3;
```

What's the compiler up to? part 3

```
int (&foo())[3]
{
    static int a[3] = {1, 2, 3};
    return a;
}

int main()
{
    auto &[x, y, z] = foo();
    x = 10;
    printf("%d\n", foo()[0]);
}

↓

prints 10
```

asm

Syntax

At first glance, a little unusual

But, has commonality with other syntax for introducing names into a scope (lambda captures)

Many other variations considered, all more worse (more worse is a technical term)

https://www.youtube.com/watch?v=430o2HMODj4&feature=youtu.be&t=15m50s

Uses

Predicated return value (value must be able to exist on error, otherwise see std::optional)

```
if (auto [val, success] = atoi(str); success) {
    printf("val %d\n", val);
} else {
    printf("couldn't convert str\n");
}
```

Uses

Iterator ranges

```
if (auto [begin, end] = str.find("needle"); begin != end) {
    std::transform(begin, end, begin, ::toupper);
}
```

Uses

Structure decomposition

```
struct BigType
{
    int common1, common2;
    ... // lots of other members
    float common3;
};

void foo(const BigType &bt)
{
    const auto &[common1, common2, common3] = bt;
    // use common parts
}
```

APIs part 1

```
output name(inputs)
```

...sometime later...

```
output name(inputs, output, output, output)
```

Breaks referential locality!
Breaks idempotence!
Muddies what is input and what is output
Harder to compose (must know about all necessary intermediate storage)
Harder to reason about ownership, lifetimes, state (we should always strive for value semantics, regular interfaces)
Typically dealt with by using funny names (outParams)

APIs part 2

The Hard To Misuse Positive Score List

10. It's impossible to get wrong.
9. The compiler/linker won't let you get it wrong.
8. The compiler will warn if you get it wrong.
7. The obvious use is (probably) the correct one.
6. The name tells you how to use it.
5. Do it right or it will always break at runtime.
4. Follow common convention and you'll get it right.
3. Read the documentation and you'll get it right.
2. Read the implementation and you'll get it right.
1. Read the correct mailing list thread and you'll get it right.

https://ozlabs.org/~rusty/index.cgi/tech/2008-03-30.html

Positives

- Consistency in functions

```
outputs name(inputs)
```

- Handles more use cases than std::tie, and sometimes more efficient

```
struct S { int i; unique_ptr<widget> w; };
S f() { return {0, make_unique<widget>()}; }
auto [ my_i, my_w ] = f();
```

- Easier on the eyes (improves signal to noise ratio)

```
if (auto [iter, inserted] = map.emplace(...); inserted) {
    ...
}
```

Negatives

- Like auto function parameters, can possibly lead to subtle type bugs (e.g. swapping of return value names)

worth it

Performance

```cpp
// http://en.cppreference.com/w/cpp/language/copy_elision
#include <iostream>
#include <vector>

struct Noisy
{
    Noisy() { std::cout << "constructed\n"; }
    Noisy(const Noisy&) { std::cout << "copy-constructed\n"; }
    Noisy(Noisy&&) { std::cout << "move-constructed\n"; }
    ~Noisy() { std::cout << "destructed\n"; }
};

std::vector<Noisy> f()
{
    std::vector<Noisy> v = std::vector<Noisy>(3); // copy elision when initializing
                                                  // v from a temporary
                                                  // (guaranteed in C++17)
    return v; // NRVO from v to the returned nameless temporary (not guaranteed in C++17)
}             // or the move constructor is called if optimizations are disabled

void g(std::vector<Noisy> arg)
{
    std::cout << "arg.size() = " << arg.size() << '\n';
}

int main()
{
    std::vector<Noisy> v = f(); // copy elision in initialization of v
                                // from the result of f() (guaranteed in C++17)
    g(f());                     // copy elision in initialization of the
                                // parameter of g() from the result of f()
                                // (guaranteed in C++17)
}
```

Noisy output

```
constructed
constructed
constructed
constructed
constructed
constructed
arg.size() = 3
destructed
destructed
destructed
destructed
destructed
destructed
```

Supporting user types

```cpp
struct S {int i; char c[27]; double d;};
S f();

// add tuple_size and tuple_element support
namespace std {
    template<> struct tuple_element<0,S> { using type = int; };
    template<> struct tuple_element<1,S> { using type = string; };
    template<> struct tuple_element<2,S> { using type = double; };
    template<> struct tuple_size<S>: public integral_constant<size_t,3> {};
}
template<int> void get(const S&) = delete;
template<> auto get<0>(const S& x) { return x.i; }
template<> auto get<1>(const S& x) { return string{c,i}; }
template<> auto get<2>(const S& x) { return x.d; }
auto [ n, s, val ] = f();
```

Summary

Improves the syntax of many cumbersome, but common idioms

Reduces incidental types

Makes interfaces more regular without a performance penalty

Improves signal to noise ratio