Structured bindings uncovered

Dawid Zalewski

github.com/zaldawid
zaldawid@gmail.com
saxion.edu

Who is he?

- ~25 year on & off playing with computers
- + some microfluidics, thermodynamics, real-time, cryo-cooling, embedded, Bayesian methods, …
- C, C#, Python, C++, Java, …
- teaching (mostly) programming @ SAXION UNIVERSITY OF APPLIED SCIENCES

Outline

- Structured bindings 101
- Tuple-like objects
- How it binds (*aka ‘we need to go deeper’*)
- Let’s tie it up!

Structured bindings in the wild

```
std::map<std::string, int> counts;

auto result = counts.emplace("word", 42);


if ( result.second ){
  // inserted a new element
}
else{
  // "word" already existed
}
```

> The type of result is:
> std::pair<iterator, bool>

Structured bindings in the wild

```
std::map<std::string, int> counts;

auto [iter, inserted] = counts.emplace("word", 42);


if ( inserted ){
  // inserted a new element
}
else{
  // "word" already existed
}
```

The type of inserted is:
bool

Structured bindings in the wild

```
std::map<std::string, int> counts;

if ( auto [iter, inserted] = counts.emplace("word", 42); inserted ) {
  // inserted a new element
}
else{
  // "word" already existed
}
```

Structured bindings anatomy

type specifier — *identifier list*

```
auto [x, y, z] = f();
```

optional: & or && — *expression*

*Only placeholder **auto** is allowed*

Structured bindings 101: arrays

```
int nums[] = {1, 2, 3};
auto [x, y, z] = nums;
```

```
x == 1
y == 2
z == 3
```

```
auto [x, ...]  = nums;
auto [_, _, z] = nums;
```

No way to do it ☹️

```
std::array nums = {1, 2, 3};
auto [x, y, z] = nums;
```

Not really an array decomposition. 🤔

Structured bindings 101: public members

```
struct to_bind_t {
  std::string word;
  int count;
};

void func() {
  to_bind_t to_bind{"alice", 42};
  auto [w, c] = to_bind;
}
```

w is a binding to the word data member
c is a binding to the count data member

In C++17 only objects with all public data members can be decomposed

Structured bindings 101: accessible members

```
class to_bind_t {
  std::string word;
  int count;
  to_bind_t(std::string word, int count);
  friend void func();
};

void func() {
  to_bind_t to_bind{"alice", 42};
  auto [w, c] = to_bind;
}
```

c++20 only

In C++20 objects with accessible data members can be decomoposed

Take-aways (so far)

1. Structured bindings can be used to decompose raw arrays and objects with accessible data members.

Structuted bindings 101: tuples

```
std::tuple to_bind{std::string("alice"), 42};

auto [w, c] = to_bind;
```

w is a binding to the first tuple element
c is a binding to the second tuple element

This works because the following are valid for type: `T = decltype(to_bind):`

- `std::tuple_size<T>::value` `// number of T's elements`
- `std::tuple_element<I, T>::type` `// type of the I-th element`
- `return_type std::get<I>(T)` `// retrieves I-th element`

Enabling tuple-like access

Structured bindings for any object can be enabled by providing tuple-like access:

```
template <class T>
struct tuple_size;

template <size_t I, class T>
struct tuple_element;

template <std::size_t I>
return_type get(T t);
```

Primary templates in ::std

Not so special function template

Provided in `::std` for: **`std::tuple`**, **`std::pair`** and ~~**`std::array`**~~

Enabling tuple-like access

Structured bindings for any object can be enabled by providing tuple-like access:

```
class My {
  int n_;
public:
  My(int n): n_{n} {}
  int number() const {
    return n_;
  }
};

auto my = My(42);
auto [number] = my;
```

Intended use

Template specializations

```
template<>
struct std::tuple_size<My>{
  static constexpr int value = 1;
};

template<>
struct std::tuple_element<0, My>{
  using type = int;
};

template <std::size_t I>
auto get(const My& m) {
  return m.number();
}
```

Enabling tuple-like access

```
auto my = My(42);

~~~~~~~~

auto [number] = my;

number = 24;

~~~~~~~~

auto& [number] = my;

number = 24;
```

The value of **number** is **24**

The value of **my.n_** is still **42**

Enabling tuple-like access

```
class My {
public:
  int number() const {
    return n_;
  }
}

template <std::size_t I>
auto get(const My& m) {
  return m.number();
}
```

Enabling tuple-like **write** access

```
class My {
  int n_;
public:
  My(int n): n_{n}{}

  int& number() { return n_; }

  const int& number() const { return n_; }

};
```

Enabling tuple-like **write** access

```
template <std::size_t I>
decltype(auto) get(const My& m) {
  return m.number();
}

template <std::size_t I>
decltype(auto) get(My& m) {
  return m.number();
}

template <std::size_t I>
decltype(auto) get(My&& m) {
  return m.number();
}
```

All three overloads are needed to support structured bindings

decltype(auto) deduces the type from the return statement

Enabling tuple-like **write** access

```
auto my = My(42);
```

~~~~~~~~

```
auto [number] = my;

number = 24;
```

The value of **number** is **24**
The value of **my.n_** is still **42**

~~~~~~~~

```
auto& [number] = my;

number = 24;
```

The value of **number** is **24**
The value of **my.n_** is also **24**

Take-aways (so far)

2. Structured bindings work with objects* that expose tuple-like API :
   - std::tuple_size<T>
   - std::tuple_element<I, T>
   - get<I>(T)

* In standard library: std::array, std::pair, std::tuple

A bit of Python

```
def splitter(str_: AnyStr) -> Tuple[chr, AnyStr]:
    return (str_[0], str_[1:])

str = "alice"

while str:
    front, str = splitter(str)
    print(front)
    # or do something useful
```

bash

```
> python splitter.py
a
l
i
c
e
```

A bit of C++

Return by reference

```
struct splitter {

  std::string str_;

  char& front() { return str_.front(); }
  const char& front() const { return str_.front(); }

  splitter rest() const { return {str_.substr(1)}; }

  operator bool() const { return !str_.empty(); }

};
```

Return by value

Tuple-like access for splitter

```
template<>
struct std::tuple_size<splitter>:
    public std::integral_constant<std::size_t, 2> {};

template<>
struct std::tuple_element<0, splitter> {
    using type = char;
};

template<>
struct std::tuple_element<1, splitter> {
    using type = splitter;
};
```

Tuple-like access for splitter

```
template<std::size_t I> decltype(auto) get(const splitter& s){
  if constexpr (I == 0){ return s.front(); }
  else { return s.rest(); }
}

template<std::size_t I> decltype(auto) get(splitter& s){
  if constexpr (I == 0){ return s.front(); }
  else { return s.rest(); }
}

template<std::size_t I> decltype(auto) get(splitter&& s){
  if constexpr (I == 0){ return s.front(); }
  else { return s.rest(); }
}
```

`splitter` in action

First try

```
auto str = splitter{"alice"};

while (str){
  auto [front, str] = str;
  std::cout << front << "\n";
};
```

😞

The initializer cannot refer to one of the identifiers

Second try

```
auto str = splitter{"alice"};

while (str){
  auto [front, rest] = str;
  str = std::move(rest);
  std::cout << front << "\n";
};
```

😕

`splitter` in action

Second try

```
auto str = splitter{"alice"};

while (str){
  auto [front, rest] = str;
  str = std::move(rest);
  std::cout << front << "\n";
};
```

```
splitter's move assignment calls: 5
splitters's copy ctor calls: 5
```

```
struct splitter {
  splitter rest() const {
    return {str_.substr(1)};
  }
};
splitter get<0>(splitter& s){
  return s.rest();
}
```

Copy elision?

```
auto [front, rest] = str;
```

splitter in action

Second try

```
auto str = splitter{"alice"};

while (str){
  auto [front, rest] = str;
  str = std::move(rest);
  std::cout << front << "\n";
};

splitter’s move assignment calls: 5
splitters’s copy ctor calls: 5
```

Third try

```
auto str = splitter{"alice"};

while (str){
  auto& [front, rest] = str;
  str = std::move(rest);
  std::cout << front << "\n";
};

splitter’s move assignment calls: 5
splitter’s copy ctor calls: 0
```

🤯

How it doesn't bind

```
auto [front, rest] = str;
```

```
auto front = get<0>(str);
auto rest  = get<1>(str);
```

That's totally not what happens 🤔

How it binds

```
auto [front, rest] = str;

auto __e = str;
```

unnamed entity

Here be the copy ctor call!

```
aliastype front = get<0>(__e);
aliastype rest  = get<1>(__e);
```

front and **rest** are names, not references:

```
std::is_reference_v<decltype(rest)>          → false
std::is_same_v<decltype(rest), splitter>     → true
```

Structured bindings: placeholder type

The type specifier refers to this anonymous entity!

```
auto [front, rest] = str;

auto __e = str;
```

Lvalue

```
auto& [front, rest] = str;

auto& __e = str;
```

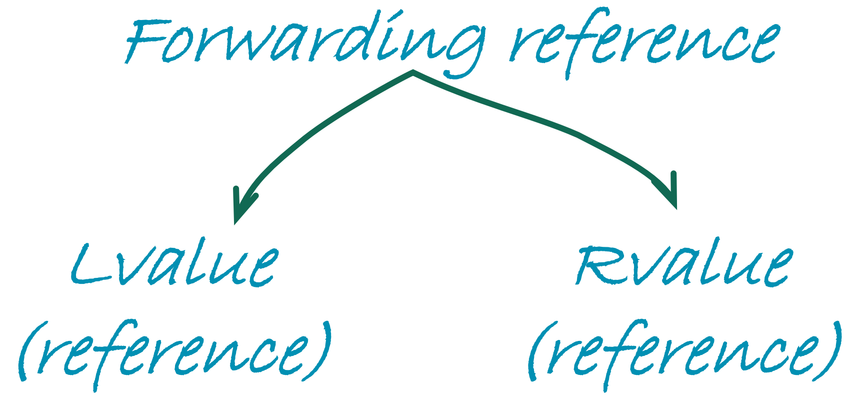
Lvalue (reference)

```
auto&& [front, rest] = str;

auto&& __e = str;
```

Forwarding reference

Structured bindings: placeholder type

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

auto&& __e = expression;
```



Type of **__e** is deduced as if:

```
template <typename E>
void function(E&& __e);

function(expression);
```

How it binds: tuple-like objects

All qualifiers apply to the unnamed entity:

__e is move-constrcuted from obj

```
Type obj;

auto [x, y] = obj;                       ⇨   Type __e = obj;

auto [x, y] = std::move(obj);            ⇨   Type __e = std::move(obj);

auto& [x, y] = obj;                      ⇨   Type& __e = obj;

const auto& [x, y] = obj;                ⇨   const Type& __e = obj;

auto&& [x, y] = obj;                     ⇨   Type& __e = obj;

auto&& [x, y] = std::move(obj);          ⇨   Type&& __e = std::move(obj);
```

How it binds: tuple-like objects

All qualifiers apply to the unnamed entity:

Better not!

```
Type fun() { return Type(); }

auto [x, y] = fun();                     →   Type __e = fun();

auto [x, y] = std::move(fun());          →   Type __e = std::move(fun());

auto& [x, y] = fun();                    →   Type& __e = fun();   (struck through)

const auto& [x, y] = fun();              →   const Type& __e = fun();

auto&& [x, y] = fun();                   →   Type&& __e = fun();

auto&& [x, y] = std::move(fun());        →   Type&& __e = std::move(fun());
```

Take-aways (so far)

3. Structured bindings work with an anonymous object under the hood – the type specifier refers to this object.

When copying is painful

```
class My {
  int n_;
public:
  My(int n): n_{n}{}
  My(const My&) = delete;
  int number() const { return n_; }
};
```

Call to a deleted function

```
My my(42);
auto [number] = my;
```

```
class My {
  int n_;
  std::array<long, 1048576> big_;
public:
  My(int n): n_{n}, big{} {}
  int number() const { return n_; }
};
```

8 MB of data copied

```
My my(42);
auto [number] = my;
```

When copying is painful: use a proxy

```
class My {
  int n_;
  std::array<long, 1048576> big_;
public:
  My(int n): n_{n}{}
  My(const My&) = delete;
  int number() const { return n_; }
  std::tuple<int> proxy() { return {n_}; }
};

My my(42);
auto [number] = my.proxy();
```

Or:

```
std::tuple<int&> proxy();
std::tuple<const int&> proxy();
```

Take-aways (so far)

4. Be careful with custom objects and structured bindings by-copy.

How it binds: tuple-like objects

```
auto [front, rest] = str;

auto __e = str;

aliastype front = get<0>(__e);
aliastype rest = get<1>(__e);
```

An object with tuple-like access API

That's (roughly) only a half-truth 😳

front and **rest** are not references:

```
std::is_reference_v<decltype(rest)>          ⇒ false
std::is_same_v<decltype(rest), splitter>     ⇒ true
```

Back to the splitter

```
struct splitter {

  std::string str_;

  char& front() { return str_.front(); }
  const char& front() const { return str_.front(); }

  splitter rest() const { return {str_.substr(1)}; }

  operator bool() const { return !str_.empty(); }

};
```

Lvalues

Rvalue

How it binds: tuple-like objects

```
auto [front, rest] = str;

auto __e = str;

aliastype front = r0;
aliastype  rest = r1;
```

variables introduced behind the scenes

```
char&        r0 = get<0>(__e);
splitter&& r1 = get<1>(__e);
```

front and **rest** are names that refer to **r0** and **r1**:

Variable **ri** is:

		initializer	
`std::tuple_element_t<i>&`	`if`	`get<i>(__e)`	is an lvalue
`std::tuple_element_t<i>&&`	`if`	`get<i>(__e)`	is an rvalue

How it binds: the strange and the weird

```
struct splitter {
  std::string str_;
  splitter rest() const { return {str_.substr(1)}; }
};

// +tuple-like access to splitter

auto str = splitter{"alice"};

auto& rest = get<1>(str);
```

← *Totally won't compile**

*with an error like: `cannot bind non-const lvalue reference to a temporary`

How it binds: the strange and the weird

```
struct splitter {
  std::string str_;
  splitter rest() const { return {str_.substr(1)}; }
};

// +tuple-like access to splitter

auto str = splitter{"alice"};

auto& [front, rest] = str;

rest = splitter("bob");
```

No problem whatsoever!

Even this is possible!

How it binds: the strange and the weird

```
struct splitter {
  std::string str_;
  splitter rest() const { return {str_.substr(1)}; }
};

auto str = splitter{"alice"};

auto& __e = str;

front = aliasfor(r0);
rest  = aliasfor(r1);

rest = splitter("bob");
```

```
char& r0 = get<0>(__e);
splitter&& r1 = get<1>(__e);
```

Take-aways (so far)

5. Structured bindings' identifiers for tuple-like objects refer to (hidden) variables whose types are inferred from the initializers (get<I>(T))

Let’s tie it up!

type specifier — *identifier list*

```
auto [x, y, z] = f();
```

optional: & or && — *expression*

- Structured bindings introduce names (identifiers).
- Those names alias elements of the object denoted by expression.
- The type specifier refers to an anonymous object, not to the identifiers.
- Things get even more complicated for objects tuple-like access.

From Python to C++ (now for real)

```python
def splitter(str_: AnyStr) -> Tuple[chr, AnyStr]:
    return (str_[0], str_[1:])

str = "alice"

while str:
    front, str = splitter(str)
    print(front)
```

From Python to C++ (now for real)

```
std::tuple<char, std::string> splitter(std::string str){
  return {str.front(), str.substr(1)};
}

str = "alice"


while str:
    front, str = splitter(str)
    print(front)
```

From Python to C++ (now for real)

```
std::tuple<char, std::string> splitter(std::string str){
  return {str.front(), str.substr(1)};
}

auto str = std::string("alice");
char front;

while str:
    front, str = splitter(str)
    print(front)
```

From Python to C++ (now for real)

```
std::tuple<char, std::string> splitter(std::string str){
  return {str.front(), str.substr(1)};
}

auto str = std::string("alice");
char front;

while(!str.empty()){
  std::tie(front, str) = splitter(std::move(str));
  std::cout << front << "\n";
}
```

Back to struct splitter

```
struct splitter {...};

// +tuple-like access to splitter

auto str = splitter("alice");
char front;

while(str){
  std::tie(front, str) = str;
  std::cout << front << "\n";
}
```

Won't work, because std::tie is a hack 😒

Why std::tie?

```
std::tie(front, str) = splitter(std::move(str));
```

The function splitter

```
template <typename... Ts>
tuple<Ts&...> tie(Ts&... t) noexcept {
    return tuple<Ts&...>(t...);
}
```

```
std::tuple<char&, splitter&>(front, str) = splitter(std::move(str));
```

We just need
an assignment operator

```
std::tuple<char, std::string>
```

Enabling tie for any class (with tuple-like access)

```
struct splitter {

  std::string str_;

  operator tuple<char, splitter>() const {
    return {str_.front(), str_.substr(1)};
  }

};

std::tie(front, str) = str;
```

Enabling tie for any class (with tuple-like access)

```
struct splitter {

  std::string str_;

  operator tuple<char, splitter>() const {
    return {str_.front(), str_.substr(1)};
  }

};

std::tie(front, str) = static_cast<tuple<char, splitter>>(str);
```

Again a proxy

Enabling tie for any class

```
struct splitter {...};

// +tuple-like access to splitter

auto str = splitter("alice");
char front;

while(str){
  any_tie(front, str) = str;
  std::cout << front << "\n";
}
```

any_tie: initialization

```
template<typename...Ts>
struct any_tie {
  std::tuple<Ts&...> tpl_;

  any_tie(Ts& ...ts) noexcept : tpl_{ts...} {}

};

auto str = splitter("alice");
char front;

any_tie(front, str)   ⇒   any_tie<char, splitter>

                           tpl_ = std::tuple<char&, splitter&>(front, str)
```

any_tie: operator=

```
template<typename...Ts>
struct any_tie {
  …
  template <typename TL>
  any_tie& operator=(TL&& tl) {









    return *this;
  }
};
```

any_tie: operator=

```
template<typename...Ts>
struct any_tie {

  template <typename TL>
  any_tie& operator=(TL&& tl) {

    const auto size = std::tuple_size_v<std::remove_cvref_t<TL>>;
    for (auto i; i < size; ++i){
      std::get<i>(tpl_) = get<i>(tl);
    }

    return *this;
  }
};
```

Removing const& C++20 way!

Won't work – i is not usable in constant expression

any_tie: operator=

```
template<typename...Ts>
struct any_tie {

  template <typename TL>
  any_tie& operator=(TL&& tl) {

    const auto size = std::tuple_size_v<std::remove_cvref_t<TL>>;

    assign(std::forward<TL>(tl), std::make_index_sequence<size>());

    return *this;

  }
};
```

Compile-time sequence:

`std::index_sequence<0, 1, ..., (size - 1)>`

any_tie: operator=

```
template<typename...Ts>
struct any_tie {
  template <typename TL>
  any_tie& operator=(TL&& tl) {...}

  template<typename TL, std::size_t...Idx>
  void assign(TL&& tl, std::index_sequence<Idx...>) {

    tpl_ = std::forward_as_tuple(get<Idx>(std::forward<TL>(tl))...);

  }
};
```

Enabling tie for any class: any_tie

```
template<typename...Ts>
struct any_tie {
  std::tuple<Ts&...> tpl_;
  any_tie(Ts& ...ts) noexcept : tpl_{ts...} {}
  template <typename TL>
  any_tie& operator=(TL&& tl) {
    const auto size = std::tuple_size_v<std::remove_cvref_t<TL>>;
    assign(std::forward<TL>(tl), std::make_index_sequence<size>());
    return *this;
  }
  template<typename TL, std::size_t...Idx>
  void assign(TL&& tl, std::index_sequence<Idx...>) {
    tpl_ = std::forward_as_tuple(get<Idx>(std::forward<TL>(tl))...);
  }
};
```

Enabling tie for any class

```
struct splitter {...};

// +tuple-like access to splitter

auto str = splitter("alice");
char front;

while(str){
  any_tie(front, str) = str;
  std::cout << front << "\n";
}
```

Take-aways (so far)

6. Structured bindings exist for simple tasks.
   There are better tools for complex scenarios.

Structured Bindings Uncovered

TIME FOR ANSWERS

Dawid Zalewski
github.com/zaldawid
zaldawid@gmail.com
saxion.edu