`{"initializer", "lists", "unleashed"}`

Daniel Frey, 2022-07-13

Today's Menu

- List-Initialization
- Fun Facts!
- Effects of List-Initialization
- `std::initializer_list`
- Nesting and Recursion
- I Like to Move It, Move It!

List-Initialization

List-Initialization

- https://en.cppreference.com/w/cpp/language/list_initialization
- Introduced in C++11
- Initialises an object from a *braced-init-list* (C++ grammar rule)
- `{ arg1, arg2, … }` is a *braced-init-list*
- Two basic syntax forms:
    - Direct-List-Initialization
    - Copy-List-Initialization

Direct-List-Initialization

```
T object { arg1, arg2, … };
T { arg1, arg2, … }
new T { arg1, arg2, … }
Class { T member { arg1, arg2, … }; };
Class::Class() : member{ arg1, arg2, … } { …
```

Both explicit and non-explicit constructors are considered.

Copy-List-Initialization

```
T object = { arg1, arg2, … };
function( { arg1, arg2, … } )
return { arg1, arg2, … };
object[ { arg1, arg2, … } ]
object = { arg1, arg2, … }
U( { arg1, arg2, … } )
Class { T member = { arg1, arg2, … }; };
```

Both explicit and non-explicit constructors are considered,
but only non-explicit constructors may be called.

Fun Facts!

Not an Expression!?

- A *braced-init-list* is not an expression and therefore has no type, e.g.
  - `decltype({1,2})` is ill-formed
- Having no type implies that template type deduction cannot deduce a type that matches a *braced-init-list*,
  - so given the declaration `template<typename T> void f(T);` the expression `f({1,2})` is ill-formed
- Special rules for type deduction using `auto`, as well as special rules for overload resolution apply, see https://cppreference.com

Sequenced Before

- Every initializer clause is “sequenced before” any initializer clause that follows it in the *braced-init-list*

Narrowing Conversions

List-Initialization limits the allowed implicit conversions by prohibiting the following:

- conversion from floating-point type to integer type
- conversion from `long double` to `double` or `float` and conversion from `double` to `float`, except where the source is a constant expression and overflow does not occur
- conversion from an integer type to a floating-point type, except where the source is a constant expression whose value can be stored exactly in the target type

Narrowing Conversions
(continued)

List-Initialization limits the allowed implicit conversions by prohibiting the following:

- conversion from integer or unscoped enumeration type to integer type that cannot represent all values of the original, except where source is a constant expression whose value can be stored exactly in the target type

- conversion from pointer type or pointer-to-member type to `bool`

Effects of List-Initialization

Effects of List-Initialization

The effects of list-initialization of an object of type T are:

- If T is an aggregate type and the *braced-init-list* has a single element of the same or derived type, the object is initialised from that element
- Otherwise, if T is a character array and the *braced-init-list* has a single element that is an appropriately-typed string literal, the array is initialised from that string as usual
- Otherwise, if T is an aggregate type, aggregate initialisation is performed
- Otherwise, if the *braced-init-list* is empty and T is a class type with a default constructor, value-initialisation is performed

Effects of List-Initialization
(continued)

- Otherwise, if T is a specialisation of `std::initializer_list`, the T object is direct-initialized or copy-initialized, depending on context, from the *braced-init-list*
- Otherwise, the constructors of T are considered in two phases:
  - All constructors that take a `std::initializer_list` as the only argument, or as the first argument and all other arguments have default values, are examined, and matched by overload resolution against the first argument
  - If the previous stage does not produce a match, all constructors of T participate in overload resolution against the set of arguments that consist of the elements of the *braced-init-list*, with the restriction that only non-narrowing conversions are allowed

Effects of List-Initialization
(continued)

- Otherwise, well… you get the idea. This is C++, we don't do easy ;)
- See https://en.cppreference.com/w/cpp/language/list_initialization
- Takeaway: Take arguments of type `std::initializer_list` if you want to handle *braced-init-list* yourself

```
std::initializer_list
```

std::initializer_list

- https://en.cppreference.com/w/cpp/utility/initializer_list
- `template<typename T> class std::initializer_list;`
- An object of `std::initializer_list<T>` is a lightweight proxy object that provides access to an array of objects of type `const T`
- The underlying array is a temporary array
- The underlying array may be allocated in read-only memory
- Copying a `std::initializer_list` does not copy the underlying objects

std::initializer_list

A `std::initializer_list` object is automatically constructed when:

- a *braced-init-list* is used to list-initialise an object, where the corresponding constructor accepts an `std::initializer_list` parameter
- a *braced-init-list* is used as the right operand of assignment or as a function call argument, and the corresponding assignment operator/function accepts an `std::initializer_list` parameter
- a *braced-init-list* is bound to `auto`, including in a range-based for loop:
  `for( int x : {1,2,3} ) { … }`

Nesting and Recursion

Nesting

```
std::vector< std::vector< std::string > > v = {
  { "first", "vector" }, // two elements
  { "second", std::string(), "vector"s, {} }, // four
  {}, // zero
  { { "blablabla", 3 }, "wait, what?" } // two
};
```

JSON basics

As seen in your favourite JSON library:

```
jvalue v1 = true;
jvalue v2 = 42;
jvalue v3 = "string";
```

JSON basics

```
struct jvalue
{
  using variant_t = std::variant<bool,int,std::string,…>;
  variant_t value;

  jvalue( const jvalue& ) = default;
  jvalue( jvalue&& ) = default;

  template<typename T> jvalue( T&& v )
    : value(std::forward<T>(v))
  {}
};
```

JSON objects

```
jvalue v = {
  { "first", "key/value pair" },
  { "second", 42 },
  { "third", true }
};
```

Recursion

```
jvalue v = {
  { "first", "key/value pair" },
  { "second", 42 },
  { "third", true },
  { "fourth", // nested jvalue
              {
                { "nested1", "foo" },
                { "nested2", false },
                { "nested3", 1701 }
              }
  }
};
```

Recursion

```
struct jvalue
{
  using jobject_t = std::map<std::string, jvalue>;
  using jarray_t = std::vector<jvalue>;

  using variant_t =
    std::variant<bool,int,std::string,…,jarray_t,jobject_t>;

  jvalue( std::initializer_list<jobject_t::value_type> il )
    : value(il)
  {}
};
```

Recursion

```
jvalue jarray( std::initializer_list<jvalue> il )
{
  return il;
}
```

```
jvalue v = {
  { "first", "key/value pair" },
  { "second", jarray( {42, false, "Hello"} ) },
  { "third", true }
};
```

I Like to Move It, Move It!

Move It!

```
struct jmember
{
  mutable std::string key;
  mutable jvalue value;
};
```

Move It!

```
struct jvalue
{
  template<typename T> jvalue( T&& v );

  jvalue( std::initializer_list<jmember>&& il ); // move it!
  jvalue( const std::initializer_list<jmember>& il ); // copy

  jvalue( std::initializer_list<jmember>& il )
    : jvalue( const_cast<…>(il) )
  {}
};
```

Move It!

```
jvalue::jvalue( const std::initializer_list<jmember>& il )
{
  jobject_t obj;
  for(const auto& [k,v] : il) {
    obj.emplace(k,v);
  }
  value = std::move(obj);
}
```

Move It!

```
jvalue::jvalue( std::initializer_list<jmember>&& il )
{
  jobject_t obj;
  for(const auto& [k,v] : il) {
    obj.emplace( std::move(k), std::move(v) ); // move it!
  }
  value = std::move(obj);
}
```

Thank you!

Questions?