Modern C++

Florian “Florob” Zeitz

2016-08-25

Initialization: Old and dated


```
1  class fnord {};
2
3  std::vector<float> v;
4
5  std::string r(fnord());
6
7  std::string h("Foo");
8
9  std::map<int, float> m;
10 m[3] = 3.14f;
11 m[42] = 23.f;
12
13 std::string x = "Hello!";
```

```
15 int ys[] = {1, 2, 3, 4, 5};
16
17 // error: non-aggregate type
18 // 'std::vector<int>' cannot
19 // be initialized with an
20 // initializer list
21 std::vector<int> zs = {1, 2};
22
```

Initialization: Old and dated


```
1  class fnord {};
2
3  std::vector<float> v;
4
5  std::string r(fnord());
6
7  std::string h("Foo");
8
9  std::map<int, float> m;
10 m[3] = 3.14f;
11 m[42] = 23.f;
12
13 std::string x = "Hello!";
```

```
15 int ys[] = {1, 2, 3, 4, 5};
16
17 // error: non-aggregate type
18 // 'std::vector<int>' cannot
19 // be initialized with an
20 // initializer list
21 std::vector<int> zs = {1, 2};
22
```

- there is a deliberate bug here, see it?

Initialization: New and hip


```
1  class fnord {};
2
3  std::vector<float> v;
4
5  // error: no matching constructor
6  // for initialization of 'std::string'
7  std::string r{fnord()};
8
9  std::string h{"Foo"};
10
11 std::map<int, float> m = { {3, 3.14f}, {42, 23.f} };
12
13 int ys[] = {1, 2, 3, 4, 5};
14
15 std::vector<int> zs = {1, 2};
```

Uniform Initialization

- Many ways to initialize objects
- Some of them confusing (C++'s "most vexing parse")
- Some of them inconsistent (e. g. C-style arrays vs `std::vector`)
- C++11 allows using `{}` for all initializations
- forbids implicit narrowing `int(3.4f)` vs. `int{3.4f}`

std::initializer_list<T>

- proxy to an array of `T`
- iterable
- created automatically from a *braced-init-list*
- constructors taking a `std::initializer_list<T>` are preferred when performing list initialization

When not to use this

- few exceptions where this can't be used
- `std::vector{5, 3}`
  - `std::vector::vector(size_type count, T const &value)`
  - `std::vector::vector(std::initializer_list<T> init)`
- when narrowing is desired: `int(val)`

Delegating Constructors


```
1  struct Vehicle {
2          int speed;
3          int tires;
4          int gas = 20;
5
6          Vehicle(int speed, int tires)
7              : speed{speed}, tires{tires} {}
8          Vehicle(int speed)
9              : Vehicle{speed, 4} {}
10 };
```

auto

- C++ supports type deduction
- the `auto` keyword has been repurposed for this
- Pop Quiz: What was the old purpose
- no complex type inference (e. g. no Hindley-Milner)

auto in variable declarations


```
1  auto x = 3l;
2
3  auto const &y = long{x};
4
5  auto a = new std::array<float, 23>;
6
7  auto const v = std::vector<int> {1, 2, 3};
8  auto it = begin(v);
```

- type is determined based on the initializer
- template argument deduction is used to find the type
- modifiers can be used to guide type deduction
- when used to declare multiple variables, types must match

auto in return type position


```
1  auto randflip() -> int {
2          return 1; // Chosen by fair coin flip
3  }
4
5  template <typename T>
6  auto get_last(T v) {
7          return *--end(v);
8  }
```

- with arrow: trailing return type syntax, no type deduction
- without arrow: type is deduced from the operand of the return statement (C++14)

Almost Always Auto (AAA) Style

- using `auto` gives you:
  correctness, performance, maintainability, and robustness
- less importantly it is convenient
- forces you to initialize variables
- unnecessary type conversions are avoided
- variable types *track* changes made elsewhere
- where necessary committing to a type is still possible
- promotes writing against interfaces, not implementation

AAA Style Example


```
1  for (auto it = begin(v), e = end(v); it != e; ++it) {}
2
3  // Determine type from literal
4  auto integer = 42;
5  auto floating = 23.f;
6  auto precise = 23.;
7  auto rope = "Hello"s; // std::string since C++14
8
9  // Commit to a type
10 auto f = int{v.front()}; // can't narrow
11 auto w = std::vector<int> {v};
```

Range-based for loops


```
1  auto v = std::map<int, std::string> { {5, "Five"},
2                                        {3, "Three"} };
3
4  for (auto it = begin(v), e = end(v); it != e; ++it) {
5          std::cout << it->first << ' ' << it->second << '\n';
6  }
7
8  for (auto i : v) {
9          std::cout << i.first << ' ' << i.second << '\n';
10 }
```

- more readable for loop syntax
- available for arrays, and where `begin()` and `end()` are defined
- desugars to a regular for loop

nullptr

- `NULL` and `0` are integer constants
- passing `NULL` as a templated argument will resolve to **`int`**
- `std::nullptr_t` is a new non-pointer type with implicit conversions to the null pointer value of any type
- **`nullptr`** is *the* new null pointer constant
- overloading for **`nullptr`** is possible

```
1 void func(double *x) {
2         std::cout << "pointer to double\n";
3 }
4
5 void func(std::nullptr_t nptr) {
6         std::cout << "nullptr\n";
7 }
```

lvalues and rvalues

- lvalues:
  `var, *var, vec[0]`
- rvalues:
  `23, foo + bar, fun()`
- lvalues are things that can be to the left of an `operator=`
- lvalues are things that can have their address taken
- lvalues are (mostly) temporaries

References


```
1  int x = 42;
2
3  // ref binds only to lvalues
4  int &r = x;
5
6  // const ref binds to lvalues or rvalues
7  int const &cr1 = x;
8  int const &cr2 = 23;
9
10 // rvalue ref binds only to rvalues
11 int &&rr = 7;
```

Move semantics: Motivation


```
1 template <typename T>
2 void swap(T &x, T &y) {
3         T temp = x;
4         x = y;
5         y = temp;
6 }
```

- say `T` is `std::vector`
- copy of `x` in `temp`
- copy of `y` in `x` (old `x` destructed)
- copy of `temp` in `y` (old `y` destructed)
- `temp` gets destructed

$\Rightarrow$ ideally we would reuse heap allocations

Move semantics: move constructor

- a move constructor is defined as `Type(Type &&other)`
- creates a new object, reusing parts of `other`
- moved object should remain in an "empty" but *valid* state
- e. g. reuse the heap allocation of a `std::vector`

Move semantics: move assignment

- a move assignment operator is defined as `Type& operator=(Type &&other)`
- similar to move constructor, but assignment

Move semantics: Rule of Five

- Rule of Three says:
  "If you overload any of destructor, copy constructor, or assignment operator, overload all of them"
- in C++11 move constructor and move assignment have to be added to the list

$\Rightarrow$ *Rule of Five*

Move semantics: Copy and Swap


```
1  class DoubleVect {
2          std::vector<double> v;
3  public:
4          DoubleVect() = default;
5          DoubleVect(DoubleVect const &other) = default;
6          DoubleVect(DoubleVect &&other) = default;
7          DoubleVect& operator=(DoubleVect other) {
8                  v.swap(other.v);
9                  return *this;
10         }
11 };
```

- instead of defining two assignment operators utilise copy and move constructor
- more correctly “construct and swap”

Move semantics: std::move()


```
1 auto v = std::vector<int> {1, 2, 3};
2 auto v2(std::move(v));
```

- move constructors are only called on rvalues
- sometimes it is desirable to move from lvalues
- `std::move()` converts to an rvalue reference

Move semantics: other uses


```
1 auto s = std::string {"OHAI!"};
2 auto vs = std::vector<std::string>{};
3 vs.push_back(std::move(s));
```

- overload other functions that would have to copy
- e. g. `vector::push_back()`

Perfect Forwarding


```
1 template <typename T, typename... Args>
2 std::vector<T> elem_vector(Args&&... args) {
3         return std::vector<T> {
4                 T(std::forward<Args>(args)...)
5         };
6 }
```

- "forward" arguments to another function *exactly* as given
- used by e. g. `vector::emplace_back()`

Smart Pointer

- pointer that provide something beyond regular pointers
- usually resource management
- C++11 adds to new ones to the STL: `unique_ptr` and `shared_ptr`

unique_ptr


```
1 auto di = std::unique_ptr<int>{ new int{15} }; // C++11
2 auto ui = std::make_unique<int>(15); // C++14
3 auto v = std::vector<std::unique_ptr<int>>{};
4 v.push_back(std::move(ui));
```

- replaces `auto_ptr`
- *owns* the allocation
- frees memory on destruction
- not copyable, only moves
- new preferred way to create heap allocations
- use `make_unique<T>()` to avoid exception safety problems

shared_ptr


```
1 auto si = std::make_shared<int>(15);
2 auto si2 = si;
3 *si2 += 5;
4 std::cout << *si << ' ' << *si2 << '\n'; // 20 20
```

- pointer type providing *shared ownership*
- frees memory on destruction of the last pointer

⇒ reference counted

- copy bumps reference count
- `make_shared<T>()` uses a single allocation for metadata and data
  - fewer allocator calls
  - better spacial locality

Guidance

- when possible use `unique_ptr<T>`
  - simplest semantics
  - more efficient then `shared_ptr<T>`
  - more flexible: can be moved to other smart or raw pointer
- when shared ownership is required use `shared_ptr<T>`
- don't *overuse* smart pointers
  - required parameters should still be passed as `T&`
  - optional parameters should still be passed as `T*`
  - consumers don't have to care about stack vs. heap allocation
  - in particular `void func(std::unique_ptr<T> &up)` is an *anti-pattern*
  - use `std::unique_ptr<T>` parameters for sinks

Lambdas


```
1  auto v = std::vector<int> {3, 5, 2, 1, 4};
2  std::sort(begin(v), end(v), [](int a, int b) -> bool {
3      return a >= b;
4  });
5  auto x = 4;
6  v.erase(std::remove_if(begin(v), end(v), [x](int n) {
7          return n > x;
8  }), end(v));
9  std::for_each(begin(v), end(v), [](auto x) { // C++14
10     std::cout << x << '\n';
11 });
```

- syntax for creating unnamed function objects
- can capture variables (closure)

Capture specification


```
1 auto all_byval = [=]() { /* ... */};
2 auto all_byref = [&]() { /* ... */};
3 auto mixed     = [x, &v]() { /* ... */};
4 auto byval_ovr = [=, &v]() { /* ... */};
```

- the way variables are captured can be specified
- default is specified by `=` (by value) or `&` (by reference)
- specific variables can be captured differently

What we didn't catch

- improvements for template meta programing (e. g. `enable_if`)
- user-defined literals
- **`using`** as replacement of/extension to **`typedef`**
- **`constexpr`**
- attributes: `[[deprecated]]`

Some links

- Herb Sutter's "Elements of Modern C++ Style"
- C++ Core Guidelines

Thank you for your attention. Any questions?

https://babelmonkeys.de/~florob/talks/OC-2016-08-25-modern-cpp.pdf