Modern(ish) C++

Catch-up Session

Michal Sudwoj

13.07.2022

Contents

- `auto`
- Type Constraints (Concepts)
- Lambda Expressions
- Structured Bindings
- Range-Based `for`
- Demo / Hands-on: `std::vector`

`auto`: Type Deduction (C++11)[1]

Variable definition: deduce type from definition

```
auto i =  1 ; // `i` is `int`
auto d =  1.; // `d` is `double`
auto c = '1'; // `c` is `char`

auto & j = i; // `j` is `int &`
auto & k = j; // `k` is `int &`
```

[1] https://en.cppreference.com/w/cpp/language/auto
[2] https://cppinsights.io/s/6a805cbf

`auto`: Type Deduction (C++11)

Cave: Expression Templates (eg. `Eigen`[1], ...)

```
double a;
Eigen::Vector x, y;

auto z = a * x + y; // `z` is not `Eigen::Vector`,
 ↪  but something else ...
```

[1] https://eigen.tuxfamily.org/dox/TopicPitfalls.html#title3

[2] https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Res-auto

[3] https://herbsutter.com/2013/08/12/gotw-94-solution-aaa-style-almost-always-auto/

`auto`: Abbreviated Function Template (C++20)[1]

```cpp
bool is_positive(auto x) { return x > 0; }
// is equivalent to
template<class T>
bool is_positive(T x) { return x > 0; }

auto add(auto x, auto y) { return x + y; }
// is equivalent to
template<class T, class U>
auto add(T x, U y) { return x + y; }
```

[1] https://en.cppreference.com/w/cpp/language/function_template#Abbreviated_function_template

auto: Type Constraints (C++20)

```cpp
#include <concepts>

// OK
std::integral auto i = 1;

// error: deduced type 'double' does not satisfy
 ↪  'integral'
std::integral auto d = 1.;

// OK
std::integral auto c = '1';
```

Lambda Expressions (C++11)[1]

Create an anonymous function, which can capture its environment (**a closure**[2])

```
std::vector<int> xs = {1, /* ..., */ 100};
std::for_each(xs.cbegin(), xs.cend(),
  [](int x) { // this is a lambda
    bool div3 = (x % 3 == 0);
    bool div5 = (x % 5 == 0);
    if (div3 && div5) { std::cout << " fizz buzz,"; }
    else if (div3) { std::cout << " fizz,"; }
    else if (div5) { std::cout << " buzz,"; }
    else { std::cout << ' ' << x << ','; }
  }
);
```

[1] https://en.cppreference.com/w/cpp/language/lambda
[2] https://en.wikipedia.org/wiki/Closure_(computer_programming)
[3] https://cppinsights.io/s/49e154eb

Lambda Expressions (C++11)

```
auto add = [](int x, int y) { return x + y; };

// is __similar__ to

auto add(int x, int y) { return x + y; };

// but is **actually equivalent** to

class Adder { // some unique compiler-generated name
  public:
    auto operator()(int x, int y) { return x + y; }
};
auto add = Adder{};
```

[1] https://cppinsights.io/s/b8a37260

Lambda Expressions (C++11): Captures[1]

```cpp
int a = 1;
int b = 2;
int c = 3;
// don't capture anything
auto add = [](int x, int y) { return x + y; };
// capture `a` **by value**
auto add_a = [a](int x) { return x + a; };
// capture `a` **by reference**
auto inc_a = [&a]() { ++a; };
// capture what is needed **by value**
auto add_ab = [=](int x) { return x + a * b; };
// capture what is needed **by reference**
auto inc_ab = [&]() { ++a; ++b; };
```

[1] https://en.cppreference.com/w/cpp/language/lambda#Lambda_capture

[2] https://cppinsights.io/s/f493611f

Lambda Expressions (C++11): Captures[1]

We can also create new bindings:

```
// equivalent to `add_a`
auto add_c = [c = a](int x) { return x + c; };
// equivalent to `inc_a`
auto inc_c = [&c = a]() { ++c; };
```

[1] https://en.cppreference.com/w/cpp/language/lambda#Lambda_capture

[2] https://cppinsights.io/s/f493611f

Lambda Expressions (C++11): Captures

```
// given
int i = 1; char c = '1'; double d = 1.0;
// then
auto f = [&, d = d + 1](auto x) { ++i; return x + d;
 ↪  };
// is equivalent to
class SomeUniqueCompilerGeneratedName {
  private:
    int & i; double d;
  public:
    SomeUniqueCompilerGeneratedName(int & i, double
 ↪  d) : i{i}, d{d + 1} {}
    auto operator()(double x) { ++i; return x + d; }
};
SomeUniqueCompilerGeneratedName f(i, d);
```

[1] https://cppinsights.io/s/e2f81bf1

Lambda Expressions (C++11)

1. since the compiler generates a unique name for each lambda's class, the use of `auto` is required
   - the class name is generated by the compiler → not known to the programmer

     ```
     SomeUniqueCompilerGeneratedName f = [](){};
     ```
2. each lambda is a unique object
   - different class
   - different instance

[1] https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#SS-lambdas

Structured Bindings (C++17)[1]: Before C++11

```
std::vector<std::tuple<double, double>> points = { /*
 ↪  ... */ };


for (int i = 0; i < points.size(); ++i) {
  // ugh ...
  double x = std::get<0>(points[i]);
  double y = std::get<1>(points[i]);
  // do something ...
}
```

[1] https://en.cppreference.com/w/cpp/language/structured_binding
[2] https://cppinsights.io/s/fc6308a5

Structured Bindings (C++17): Since C++11

```
std::vector<std::tuple<double, double>> points = { /*
 ↪  ... */ };


for (int i = 0; i < points.size(); ++i) {
  // better, but still ugh ...
  double x, y;
  std::tie(x, y) = points[i];
  // do something ...
}
```

[1] https://cppinsights.io/s/9adaa2e8

Structured Bindings (C++17): Since C++17

```
std::vector<std::tuple<double, double>> points = { /*
 ↪  ... */ };


for (int i = 0; i < points.size(); ++i) {
  // nice!
  auto [x, y] = points[i];
  // do something ...
}
```

[1] https://cppinsights.io/s/229ac736

Structured Bindings (C++17)

Also works for arrays, structs, …

```
double p[2] = {1.0, 2.0};
auto [px, py] = p;

struct Vector3 {
  double x;
  double y;
  double z;
};

Vector3 v{3.0, 4.0, 5.0};
auto [vx, vy, vz] = v;
```

[1] https://cppinsights.io/s/9ffa5aa0

Range-based `for` (C++11)[1]

```
for (auto & point : points) {
  std::cout << point << '\n';
}
// is desugared (approximately) to
{ // new scope to not leak variables
  auto range = points; // handle side-effects!
  auto begin = range.begin();
  auto end   = range.end();
  for (; begin != end; ++begin) {
    auto & point = *begin;
    std::cout << point << '\n';
  }
}
```

[1] https://en.cppreference.com/w/cpp/language/range-for
[2] https://cppinsights.io/s/a4a0464e

Conclusion

```cpp
struct Point { double x, y; };
std::vector<Point> points = { /* ... */ };
Point origin = { 0.0, 0.0 };
auto dist = [origin](double x, double y) {
  auto dx = origin.x - x;
  auto dy = origin.y - y;
  return std::sqrt(dx * dx + dy * dy);
};
std::sort(points.begin(), points.end(),
  [&dist](auto a, auto b) {
    return dist(a.x, a.y) < dist(b.x, b.z);
  }
);
for (auto & [x, y] : points) {
  std::cout << dist(x, y) << ' ' << x << ' ' << y << '\n';
}
```

[1] https://cppinsights.io/s/f5e9202b

Questions?