⚡ C++23 deducing this

Sarthak Sehgal

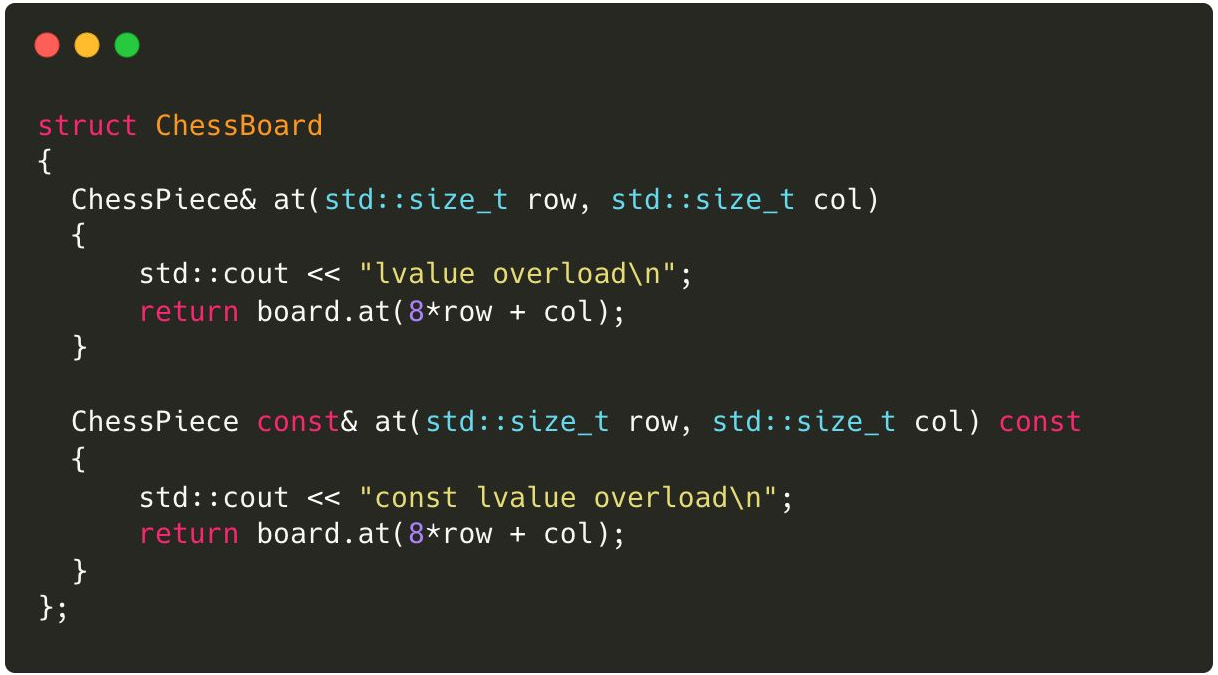
Quoting Tech Lead @ Maven Securities

Maven

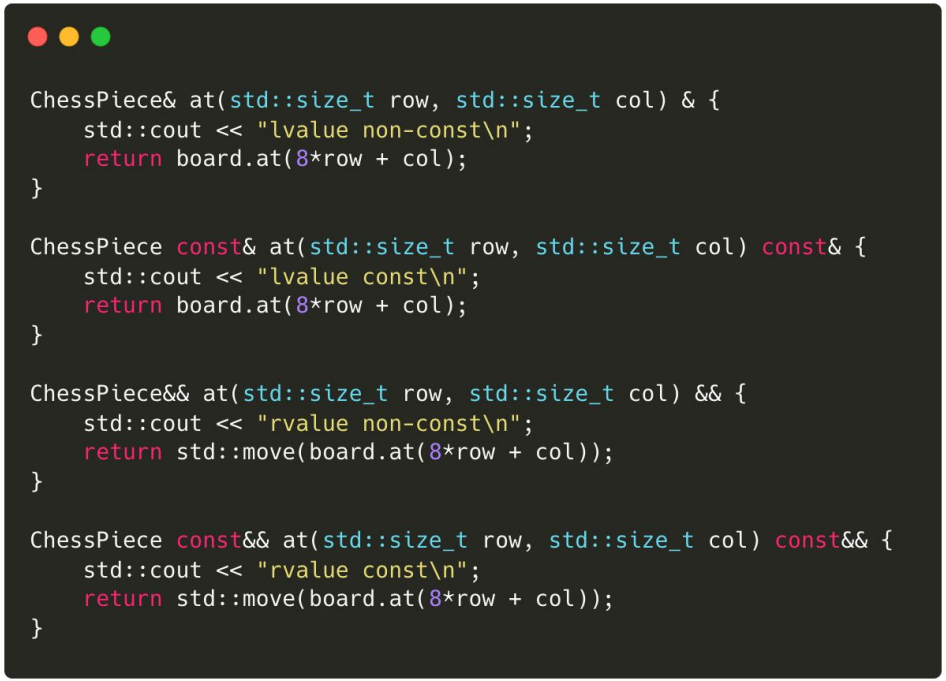
Chicago | London | Amsterdam

CppCon 2025

```
struct ChessBoard
{
  ChessPiece& at(std::size_t row, std::size_t col)
  {
      std::cout << "lvalue overload\n";
      return board.at(8*row + col);
  }
};
```



```
struct ChessBoard
{
  ChessPiece& at(std::size_t row, std::size_t col)
  {
      std::cout << "lvalue overload\n";
      return board.at(8*row + col);
  }

  ChessPiece const& at(std::size_t row, std::size_t col) const
  {
      std::cout << "const lvalue overload\n";
      return board.at(8*row + col);
  }
};
```

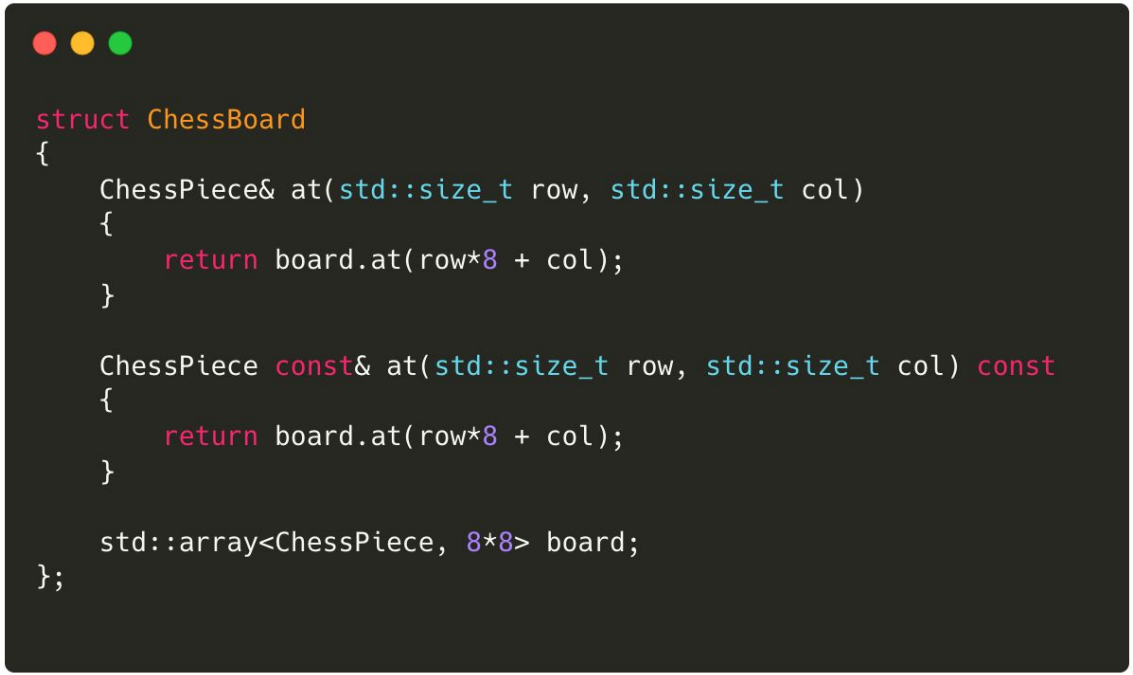


```
ChessPiece& at(std::size_t row, std::size_t col) & {
    std::cout << "lvalue non-const\n";
    return board.at(8*row + col);
}

ChessPiece const& at(std::size_t row, std::size_t col) const& {
    std::cout << "lvalue const\n";
    return board.at(8*row + col);
}

ChessPiece&& at(std::size_t row, std::size_t col) && {
    std::cout << "rvalue non-const\n";
    return std::move(board.at(8*row + col));
}

ChessPiece const&& at(std::size_t row, std::size_t col) const&& {
    std::cout << "rvalue const\n";
    return std::move(board.at(8*row + col));
}
```

```
ChessPiece& at(std::size_t row, std::size_t col) & {
    std::cout << "lvalue non-const\n";
    return board.a
}

ChessPiece const&
    std::cout << "
    return board.a
}

ChessPiece&& at(st
    std::cout << "
    return std::mo
}

ChessPiece const&&
    std::cout << "rvalue const\n";
    return std::move(board.at(8*row + col));
}
```


```
auto&& at(this auto&& self, std::size_t row, std::size_t col) {
    return std::forward<decltype(self)>(self).board.at(8*row + col);
}
```

* = if you have C++23

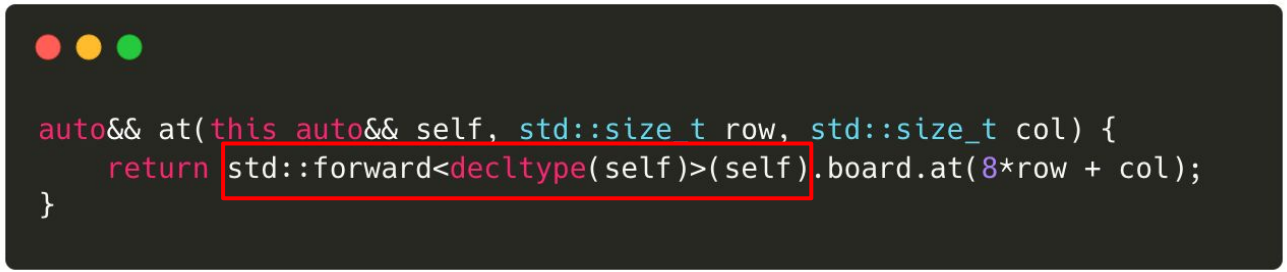


```
struct ChessBoard
{
    ChessPiece& at(std::size_t row, std::size_t col)
    {
        return board.at(row*8 + col);
    }

    ChessPiece const& at(std::size_t row, std::size_t col) const
    {
        return board.at(row*8 + col);
    }

    std::array<ChessPiece, 8*8> board;
};
```

```cpp
struct ChessBoard
{
private:
    std::array<ChessPiece, 8*8> board;

    static auto& at(auto& self, std::size_t row, std::size_t col)
    {
        return self.board.at(8*row + col);
    }

public:
    ChessPiece& at(std::size_t row, std::size_t col)
    {
        return at(*this, row, col);
    }

    ChessPiece const& at(std::size_t row, std::size_t col) const
    {
        return at(*this, row, col);
    }
};
```

DRY - Don't Repeat Yourself

```
auto&& at(this auto&& self, std::size_t row, std::size_t col) {
    return std::forward<decltype(self)>(self).board.at(8*row + col);
}
```


auto&& at(this auto&& self, std::size_t row, std::size_t col) {
return std::forward<decltype(self)>(self).board.at(8*row + col);
}

```cpp
class ChessPiece
{
public:
  auto&& at(this auto&& self, std::size_t row, std::size_t col) {
      return std::forward<decltype(self)>(self).board.at(8*row + col);
  }
};

...

ChessBoard cb;
ChessPiece myPiece = std::move(cb).at(0,0);
```

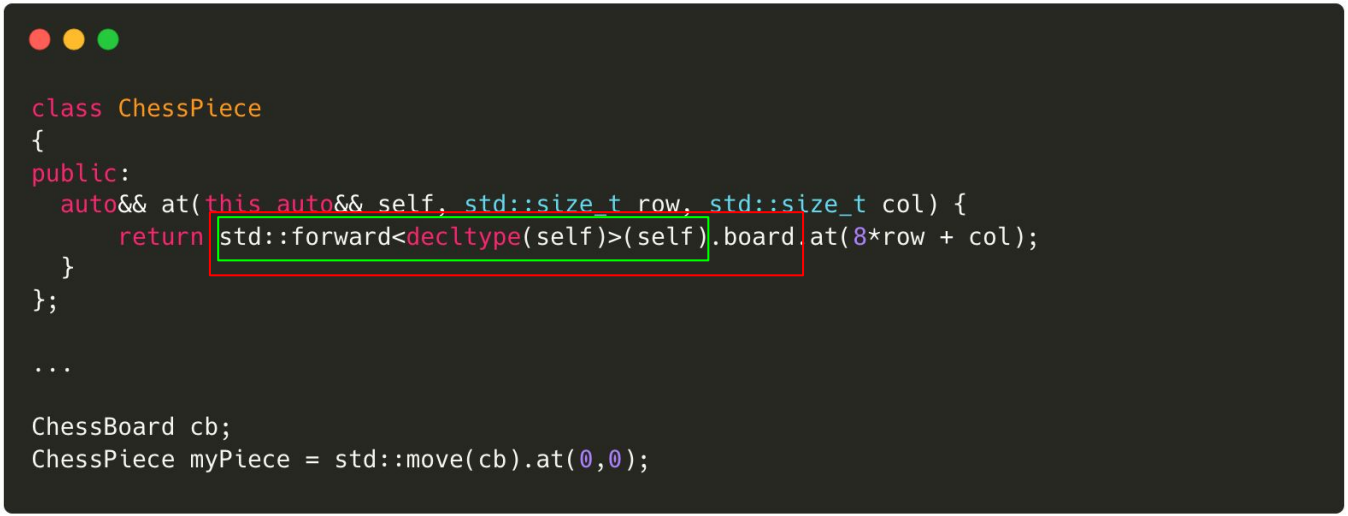
rvalue non-const

ChessPiece move or copy?

```cpp
class ChessPiece
{
public:
  auto&& at(this auto&& self, std::size_t row, std::size_t col) {
      return std::forward<decltype(self)>(self).board.at(8*row + col);
  }
};

...

ChessBoard cb;
ChessPiece myPiece = std::move(cb).at(0,0);
```

rvalue non-const

ChessPiece move or **copy**?



```
class ChessPiece
{
public:
  auto&& at(this auto&& self, std::size_t row, std::size_t col) {
      return std::forward<decltype(self)>(self).board.at(8*row + col);
  }
};

...

ChessBoard cb;
ChessPiece myPiece = std::move(cb).at(0,0);
```

"Copy Constructed"

```cpp
class ChessPiece
{
public:
  auto&& at(this auto&& self, std::size_t row, std::size_t col) {
      // return std::forward<decltype(self)>(self).board.at(8*row + col);
      return std::forward_like<decltype(self)>(self.board.at(8*row + col));
  }
};

...

ChessBoard cb;
ChessPiece myPiece = std::move(cb).at(0,0);
```

"Move Constructed"

```cpp
template< class T, class U >
constexpr auto&& forward_like( U&& x ) noexcept;
```

Returns a reference to x which has similar properties to T&&

```cpp
struct IntegerPointer
{
    IntegerPointer(int i)
    : ptr{new int(i)}
    {}

    auto&& get(this auto&& self)
    {
        return *std::forward<decltype(self)>(self).ptr;
    }

    ~IntegerPointer() { delete ptr; }

    int* ptr = nullptr;
};
```

```cpp
IntegerPointer const ip(1);
auto&& i = ip.get();
```

int& instead of const int&

```
1 auto&& get(this auto&& self)
2 {
3     return std::forward_like<decltype(self)>(*self.ptr);
4 }
```

```
template< class T, class U >
constexpr auto&& forward_like( U&& x ) noexcept;
```

Returns a reference to x which has similar properties to T&&

Fin

Maven

mavensecurities.com/jobs