
Optiver

Optiver

Compilers Coursework Into

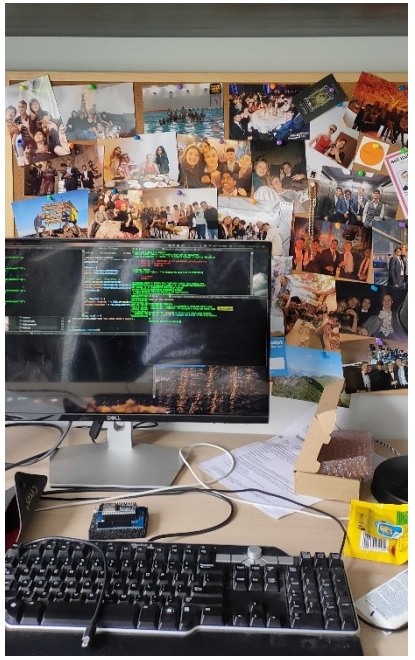
How to use a compiler to write your compiler

Simon Staal – 2026-02-12

Introduction

About me

- MEng Electronic & Information Engineering
- Graduated 2023 valedictorian
- Compilers TA for 5 years
- 7+ years of programming C++
- 2.5 years at Optiver, Amsterdam office



Agenda

1. OOP
2. Leveraging Type Information
3. RAII
4. Optiver

Optiver

What is OOP?

OOP in C++

virtual

```
2   class Shape
3   {
4   public:
5       virtual ~Shape() = default;
6       virtual double GetArea() const;
7   };
```

OOP in C++

virtual

```
 9  class Square : public Shape
10  {
11  public:
12      Square(double sideLength)
13          : mSideLength(sideLength)
14      {}
15
16      double GetArea() const
17      {
18          return mSideLength * mSideLength;
19      }
20  private:
21      double mSideLength;
22  };
```

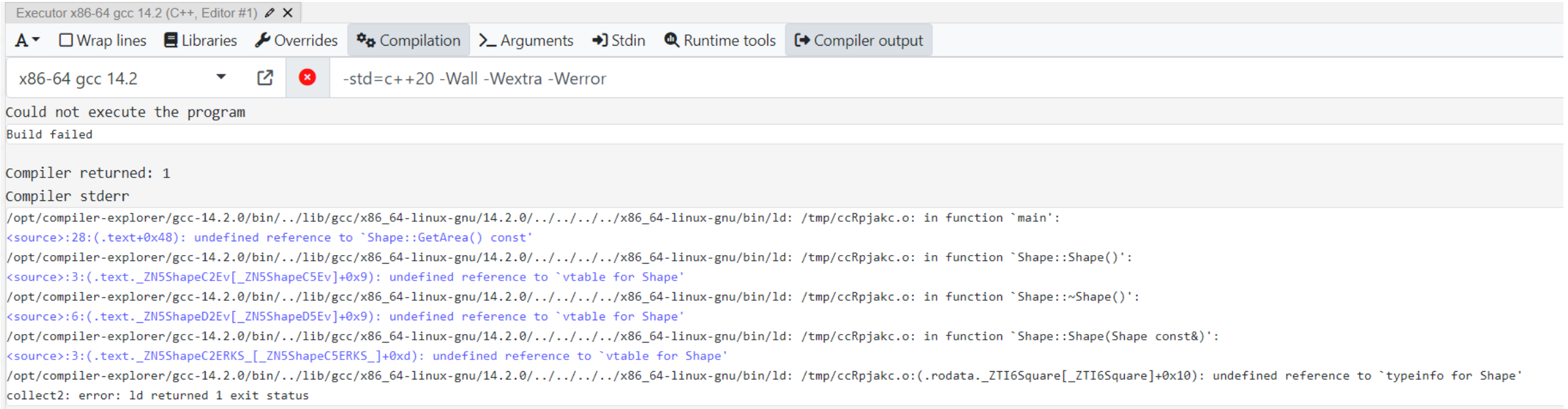
OOP in C++

What will the output be?

```
25  int main()
26  {
27      Shape s = Square{1.0};
28      std::cout << s.GetArea() << std::endl;
29  }
```

OOP in C++

What will the output be?



OOP in C++

What is going on?

```
2   class Shape
3   {
4   public:
5       virtual ~Shape() = default;
6       virtual double GetArea() const;
7   };

25  int main()
26  {
27      Shape s = Square{1.0};
28      std::cout << s.GetArea() << std::endl;
29  }
```

Never defined

"slicing"

OOP in C++

How to fix: Pure virtual

```
3  class Shape
4  {
5  public:
6      virtual ~Shape() = default;
7      virtual double GetArea() const = 0;
8  };
```

```
<source>: In function 'int main()':
<source>:27:31: error: cannot allocate an object of abstract type 'Shape'
   27 |     const Shape s = Square{1.0};
      |                               ^
<source>:3:7: note:   because the following virtual functions are pure within 'Shape':
    3 | class Shape
      |       ^~~~~
<source>:7:20: note:     'virtual double Shape::GetArea() const'
    7 |     virtual double GetArea() const = 0;
      |                    ^~~~~~~
<source>:27:17: error: cannot declare variable 's' to be of abstract type 'Shape'
   27 |     const Shape s = Square{1.0};
      |                 ^
```

Compiler Flags

- Compiler errors are (mostly) good ☺
- Transforms runtime error you might miss into obvious issues
- Google / ChatGPT / Claude how to fix them

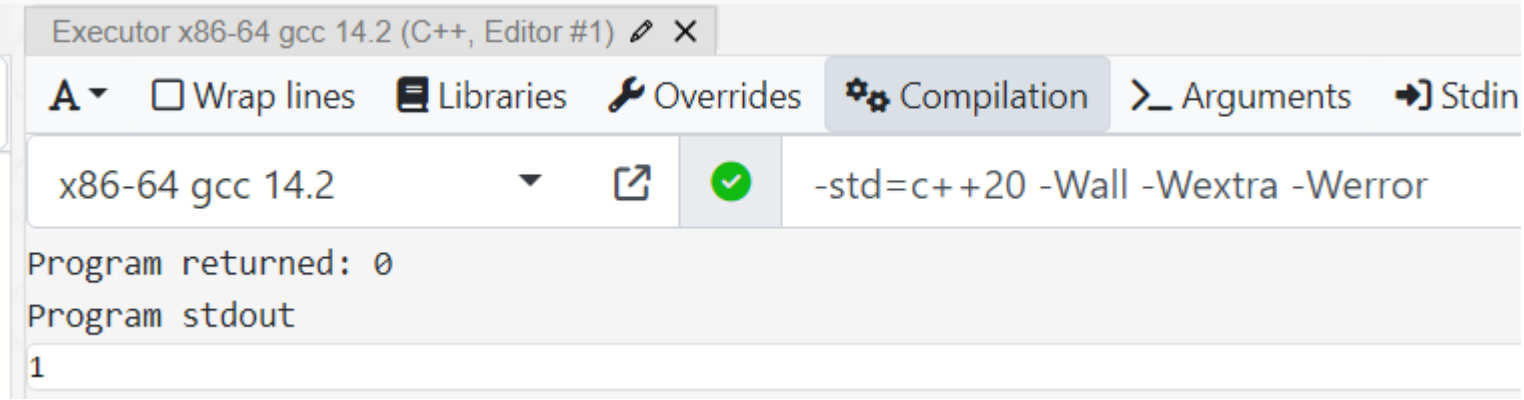
```
-std=c++23 -Wall -Wextra -Wsuggest-override -Werror
```

OOP in C++

How to fix: using pointers

Why does this help?

```
25  int main()
26  {
27      const Shape& s = Square{1.0};
28      std::cout << s.GetArea() << std::endl;
29  }
```



OOP in C?

virtual under the hood: function pointers

```
2   class Shape
3   {
4   public:
5       virtual ~Shape() = default;
6       virtual double GetArea() const;
7   };
```

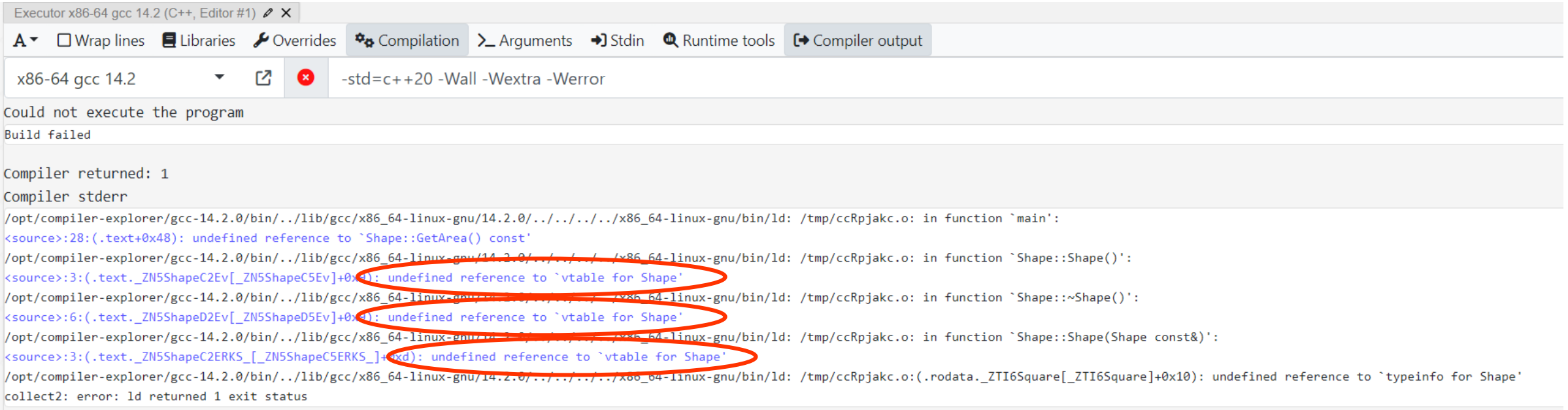
In C++ we call this a “vtable”

```
4   struct Shape
5   {
6       double (*GetArea)(const void* self);
7   };
```

How does calling this work?

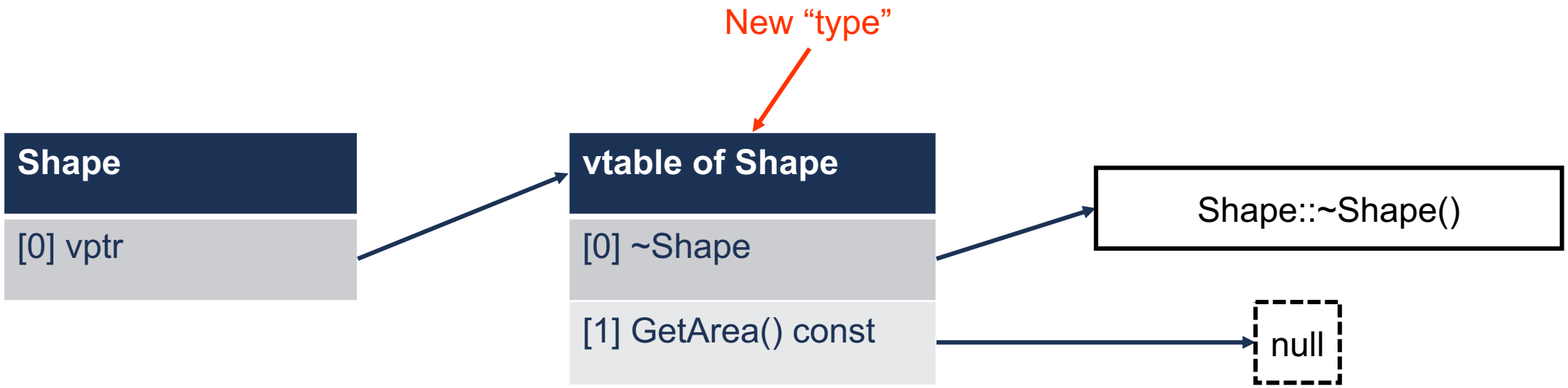
OOP in C++

Linker errors



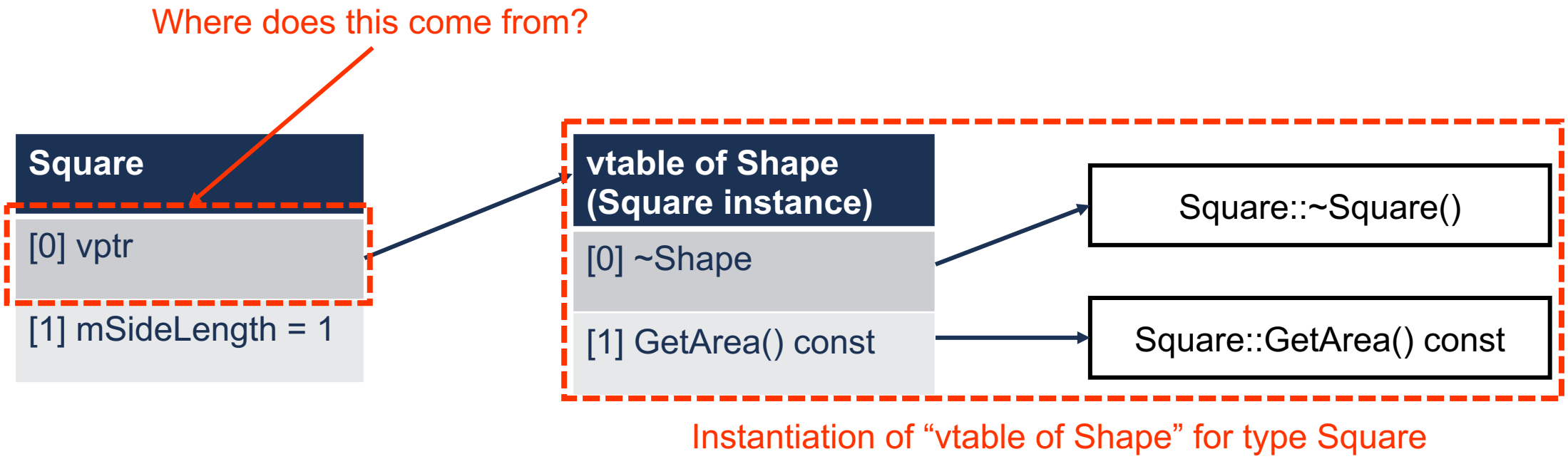
OOP in C++

virtual under the hood: base class



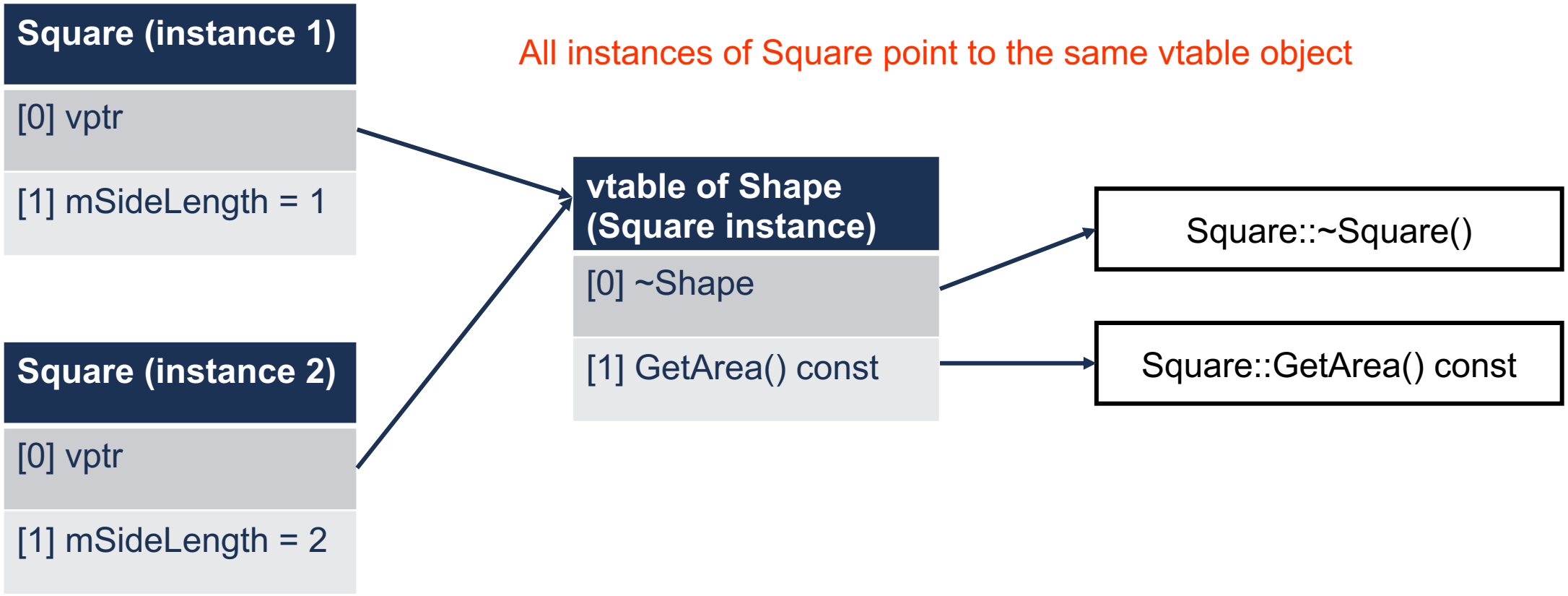
OOP in C++

virtual under the hood: derived class



OOP in C++

virtual under the hood: multiple instances



OOP in C?

Implementing inheritance

```
4  struct Shape
5  {
6      double (*GetArea)(const void* self);
7  };
```

OOP in C?

Implementing inheritance

```
 9  struct Square
10  {
11      struct Shape mBase;
12      double mSideLength;
13  };
14
15  double Square_GetArea(const void* self)
16  {
17      const struct Square* this = self;
18      return this->mSideLength * this->mSideLength;
19  }
20
21  // "Constructor", e.g. Square() in C++
22  struct Square* Square_Init(struct Square* this, double sideLength)
23  {
24      this->mBase.GetArea = &Square_GetArea; // In C++ the compiler will do this for you
25      this->mSideLength = sideLength;
26      return this;
27  }
```

Equivalent to struct Square：public Shape

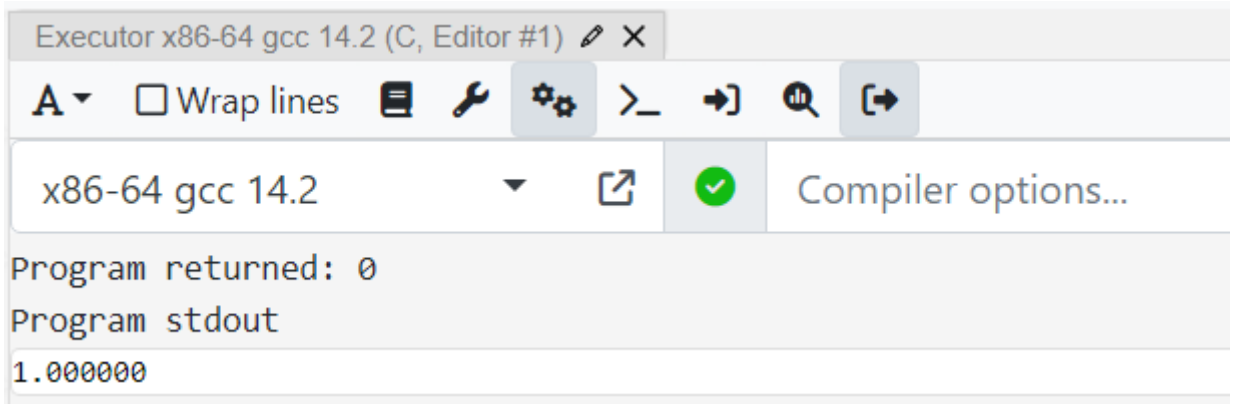
Equivalent to double GetArea() const override

OOP in C?

Inheritance in action

```
29  int main()
30  {
31      const struct Shape* shape = &Square_Init(malloc(sizeof(struct Square)), 1.0)->mBase;
32      printf("%f\n", shape->GetArea(shape));
33  }
```

Equivalent of new Square(1)



OOP in C?

What happens if we forget to init?

```
29  int main()
30  {
31      const stru                                  struct Square));
32      printf("%f
33  }
```

Why does slicing happen?

Compiler vs programmer managed memory

- Shape is using compiler managed stack memory
- Compiler will prevent you from making illegal memory access

```
15  double Square_GetArea(const void* self)
16  {
17      const struct Square* this = self;
18      return this->length * this->length;
19  }
```

- Shape& / Shape* is just an address, compiler (at call site) doesn't own the memory

OOP in C++

override and final specifiers

```
10  class Square final : public Shape
11  {
12  public:
13      Square(double sideLength)
14          : mSideLength(sideLength)
15      {}
16
17      double GetArea() const override
18      {
19          return mSideLength * mSideLength;
20      }
21  private:
22      double mSideLength;
23  };
```

Compiler API Design

AST Node

```
4  class Node
5  {
6      virtual ~Node() = default;
7      virtual void EmitRISC(std::ostream&, Context&) const = 0;
8      virtual void Print(std::ostream&) const = 0;
9  };
```

Compiler API Design

Implementing ReturnStatement

```
14  class ReturnStatement final : public Node
15  {
16  public:
17      ReturnStatement(NodePtr expression) : mExpression{expression} {}
18
19      void EmitRISC(std::ostream&, Context&) const override;
20      void Print(std::ostream&) const override;
21
22  private:
23      NodePtr mExpression;
24  };
```

Compiler API Design

Implementing ReturnStatement

```
31  void ReturnStatement::EmitRISC(std::ostream& out, Context& context) const
32  {
33      // Set destination register to return register
34      context.SetDestinationRegister("a0");
35
36      // Compile contained expression (into return register)
37      if (mExpression)
38          mExpression->EmitRISC(out, context);
39
40      // Exit function
41      out << "j " << context.GetFunctionEndLabel() << std::endl;
42  }
```

Compiler API Design

Is everything a Node?

- No! Only things that produce assembly are
- It’s okay to have plain old data types
- It’s okay to have more specialized interfaces
- Don’t try and shoehorn everything to be a Node

Type Safety

Plain old data: enum class

- Type-safe enum
- Perfect for representing a finite set of labelled values

```
4  enum class TypeSpecifier
5  {
6      INT,
7      // FLOAT,
8      // ...
9  };
```

Type Safety

Supporting new types

```
44  void ReturnStatement::EmitRISC(std::ostream& out, Context& context) const
45  {
46      // Set destination register to return register
47      switch(context.GetFunctionReturnType())
48      {
49          case TypeSpecifier::INT:
50              context.SetDestinationRegister("a0");
51              break;
52      }
```

```
<source>: In member function 'virtual void ReturnStatement::EmitRISC(std::ostream&, Context&) const':
<source>:47:11: error: enumeration value 'FLOAT' not handled in switch [-Werror=switch]
   47 |     switch(context.GetFunctionReturnType())
      |            ^
cc1plus: all warnings being treated as errors
```

Type Safety

Supporting new types

```
40  void ReturnStatement::EmitRISC(std::ostream& out, Context& context) const
41  {
42      // Set destination register to return register
43      switch(context.GetFunctionReturnType())
44      {
45          case TypeSpecifier::INT:
46              context.SetDestinationRegister("a0");
47              break;
48          case TypeSpecifier::FLOAT:
49              context.SetDestinationRegister("fa0");
50              break;
51      }
```

Type Safety

Alternative: No type safety

- Consider using a `std::string` instead

```
40  void ReturnStatement::EmitRISC(std::ostream& out, Context& context) const
41  {
42      // Set destination register to return register
43      if (context.GetFunctionReturnType() == "INT")
44      {
45          context.SetDestinationRegister("a0");
46      }
47      // Compiler cannot help you when you want to add FLOAT support
```

- Will just generate incorrect assembly
- Relies on sufficient test case coverage to catch
- You will loose marks on silly mistakes like this

Compiler API Evolution

BinaryAdd: Sample implementation

- Sometimes you will need to expose more information. Consider:

```
34  class BinaryAdd final : public Node
35  {
36  public:
37      void EmitRISC(std::ostream&, Context&) const override;
38      void Print(std::ostream&) const override;
39
40  private:
41      NodePtr mLeft;
42      NodePtr mRight;
43  };
```

Compiler API Evolution

BinaryAdd: Sample implementation

```
50  void BinaryAdd::EmitRISC(std::ostream &out, Context &context) const
51  {
52      // Acquire a register for the left operand
53      auto leftReg = context.AcquireRegister();
54      mLeft->EmitRISC(out, context);
55
56      // Acquire a register for the right operand
57      auto rightReg = context.AcquireRegister();
58      mRight->EmitRISC(out, context);
59
60      // Perform the add into the current destination register
61      out << "add " << context.GetDestinationRegister() << ", "
62          << leftReg << ", " << rightReg << "\n";
63
64      // Release temporary registers
65      context.ReleaseRegister(leftReg);
66      context.ReleaseRegister(rightReg);
67  }
```

Compiler API Evolution

BinaryAdd: How to deal with pointer arithmetic?

- We need to be able to check if one of the operands is a pointer to implement pointer arithmetic
- Do NOT add this to `Node`
  - Does it make sense for a `FunctionDefinition` to implement this? Or a `ForLoop`?
- Update types in parser file and the compiler will tell you what needs to be changed
  - Avoid `dynamic_cast` (or C-style casting) as much as possible

```
31  class Expression : public Node
32  {
33  public:
34      virtual bool IsPointerType() const = 0;
35  };
```

Compiler API Evolution

BinaryAdd: Update API

```
39  class BinaryAdd final : public Expression
40  {
41  public:
42      void EmitRISC(std::ostream&, Context&) const override;
43      void Print(std::ostream&) const override;
44      bool IsPointerType() const override;
45
46  private:
47      ExpressionPtr mLeft;
48      ExpressionPtr mRight;
49  };
```

Compiler API Evolution

BinaryAdd: Update implementation

```
54  void BinaryAdd::EmitRISC(std::ostream &out, Context &context) const
55  {
56      // Acquire registers and compile operands
57      ...
58
59      // Scale the non-pointer operand by 4 (32-bit architecture)
60      if (mLeft->IsPointerType() && !mRight->IsPointerType()) {
61          out << "  slli " << rightReg << ", " << rightReg << ", 2\n";
62      } else if (mRight->IsPointerType() && !mLeft->IsPointerType()) {
63          out << "  slli " << leftReg << ", " << leftReg << ", 2\n";
64      }
65
66      // Perform the add into the current destination register
67      out << "add " << context.GetDestinationRegister() << ", "
68          << leftReg << ", " << rightReg << "\n";
69
70      // Release temporary registers
71      context.ReleaseRegister(leftReg);
72      context.ReleaseRegister(rightReg);
73  }
```

Optiver

What is RAII?

RAII

Introduction

- RAII: Resource Allocation Is Initialisation
- Acquires necessary resources on construction
- Relinquishes necessary resources on destruction
- Moves responsibility of resource management from programmer to compiler (for stack variables)
- Originated from C++11
- E.g. `std::vector` de-allocates the heap memory it owns when it's destructed

RAII

std::unique_ptr

- Memory is an example of a “resource”
- We want to avoid “leaking” memory
- Compiler can do the work for us!
- *Ideally*: never use new or delete again*

```
108  void example()
109  {
110      int* raw = new int(1);
111      std::unique_ptr<int> smart = std::make_unique<int>(3); // allocates own memory
112      std::unique_ptr<int> alsoSmart{ new int(2) }; // takes ownership of "existing" memory
113  } // `delete` called on smart's underlying pointer by smart's destructor, raw gets leaked
```

RAII

std::unique_ptr: How it works

- What happens if we made a copy of smart?

```
108  void example()
109  {
<source>: In function 'void example()':
<source>:112:38: error: use of deleted function 'std::unique_ptr<_Tp, _Dp>::unique_ptr(const std::unique_ptr<_Tp, _Dp>&)
  112 |     std::unique_ptr<int> smartCopy = smart;
      |                                      ^~~~~
113  }  // What w      ere?
```

- If we can't copy std::unique_ptr how can we pass them around?

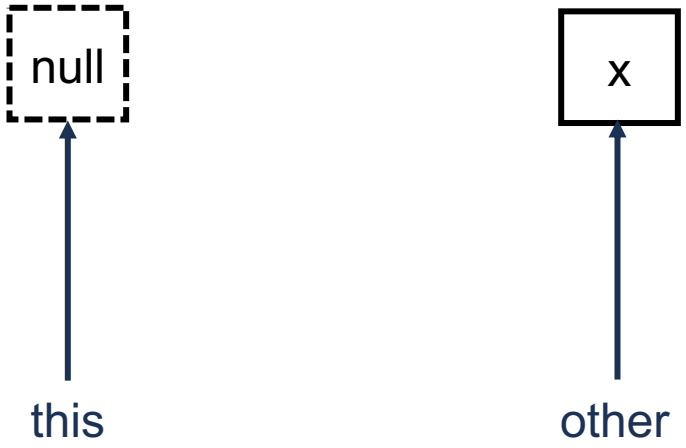
RAII

Move semantics

```
108  class MyUniquePtr {
109  public:
110      MyUniquePtr(int* raw) : mOwned(raw) {}
111      ~MyUniquePtr() { delete mOwned; }
112      MyUniquePtr(const MyUniquePtr& other) = delete;
113      MyUniquePtr(MyUniquePtr&& other) {
114          std::swap(mOwned, other.mOwned); // "steal" other's resource
115      }
116      MyUniquePtr& operator=(const MyUniquePtr& other) = delete;
117      MyUniquePtr& operator=(MyUniquePtr&& other) {
118          std::swap(mOwned, other.mOwned);
119          return *this;
120      }
121  private:
122      int* mOwned = nullptr;
123  };
```

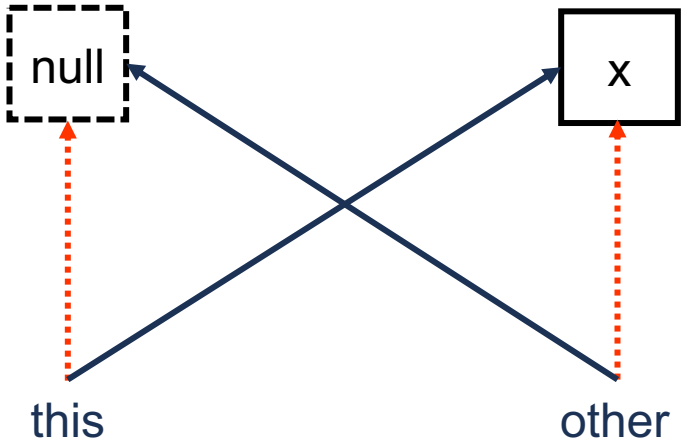
RAII

Move semantics



RAII

Move semantics



RAII

Move semantics

```
108  class MyUniquePtr {
109  public:
110      MyUniquePtr(int* raw) : mOwned(raw) {}
111      ~MyUniquePtr() { delete mOwned; }
112      MyUniquePtr(const MyUniquePtr& other) = delete;
113      MyUniquePtr(MyUniquePtr&& other) {
114          std::swap(mOwned, other.mOwned); // "steal" other's resource
115      }
116      MyUniquePtr& operator=(const MyUniquePtr& other) = delete;
117      MyUniquePtr& operator=(MyUniquePtr&& other) {
118          std::swap(mOwned, other.mOwned);
119          return *this;
120      }
121  private:
122      int* mOwned = nullptr;
123  };
```

Move Semantics

l-values and r-values

- In a nutshell:
- T& is an l-value: represents a variable
- T&& is an r-value: represents a temporary
- std::move(x) casts its argument into an r-value
  - Does not “move” anything – just a cast
- How could we convert an r-value into an l-value?
- const T& accepts both types of values. Why can it do this?

Move Semantics

l-values and r-values: example

```
108  void test(int& x) { std::cout << "l-value\n"; }
109  void test(int&& x) { std::cout << "r-value\n"; }
110
111  int main()
112  {
113      test(1);
114      int x = 2;
115      test(x);
116      test(std::move(x));
117      std::cout << x << '\n'; // What is the value of x here?
118  }
```

Move Semantics

Passing around std::unique_ptr

```
111  void example(std::unique_ptr<int>&& x)
112  {
113      // Note: x is an l-value of type "r-value reference to std::unique_ptr"
114      std::unique_ptr<int> mine = std::move(x); // another std::move is required here
115  }
116
117  int main()
118  {
119      example(std::unique_ptr<int>{}); // invoke directly with temporary
120      auto x = std::make_unique<int>(1);
121      example(x); // compile-error: x is an l-value
122      example(std::move(x)); // casts x to r-value
123  }
```

RAII

For your compiler

- Try your best not to leak memory (good practice)
  - Unfortunately, you need to use raw pointers in bison parser
  - Don’t use std::shared_ptr
  - using NodePtr = std::unique_ptr<const Node>
- Can you think of any other applications?

```
54  void BinaryAdd::EmitRISC(std::ostream &out, Context &context) const
55  {
56      // Acquire a register for the left operand
57      auto leftReg = context.AcquireRegister();
58      mLeft->EmitRISC(out, context);
59
60      // Acquire a register for the right operand
61      auto rightReg = context.AcquireRegister();
62      mRight->EmitRISC(out, context);
63
64      ...
65
66      // Release temporary registers
67      context.ReleaseRegister(leftReg);
68      context.ReleaseRegister(rightReg);
69  }
```

Wrap-up

Summary

- Keep your code as simple as possible (avoid branches)
- Reuse your code as much as possible (no copy-pasting)
- Let the compiler help catch your mistakes (make your intent explicit)
- Don’t be afraid to make simplifying assumptions
  - Try to ensure these are very “visible” so you can potentially fix them later
- Good opportunity to explore other std library classes
- Learn :)

Optiver

Questions?

Simon Staal
Software Engineer

simonstaal@optiver.com

What we do

Three tracks, choose yours

Trading

Analyse live markets, weigh risk and reward, and adapt strategies in real time.

Technology

Build and optimise the systems that power our trading – from low-latency pricing engines to advanced FPGA hardware.

Research

Translate complex data into models and strategies using maths, statistics and machine learning.

At Optiver, ideas move fast between these teams
– from hypothesis to production to live trading – often within hours.

Finding your path to build impact

Programmes

Short, focused experiences that give you a look into trading, tech and research.

Internship

Hands-on experience working on real projects — real systems, real strategies, real impact.

Graduate

move you straight into full-time ownership, combining global training in Amsterdam with immediate responsibility in your team.

Each path helps you turn your strengths into meaningful impact – no finance background required.

Programmes: Discover your fit

Applications Open

- 5-day immersive experience in trading, technology or research
- Work with our teams on real-world challenges
- Learn how your skills translate into market impact
- Meet our teams and explore different career paths

Opportunities in 2026

Future Focus (Quants) - closed

Career Kickstarter (Trading)

Career Kickstarter (Tech)

Career Kickstarter (Women in Tech) - closed

Internships: Real projects, real impact

Applications Open

- Paid, full-time summer experience in Amsterdam
- Structured training and close collaboration with experienced teams
- Ownership of projects with measurable business impact
- Pathway to a full-time offer
- **Applications closing in a few weeks**

Opportunities in 2026

Quantitative Research Internship - closed

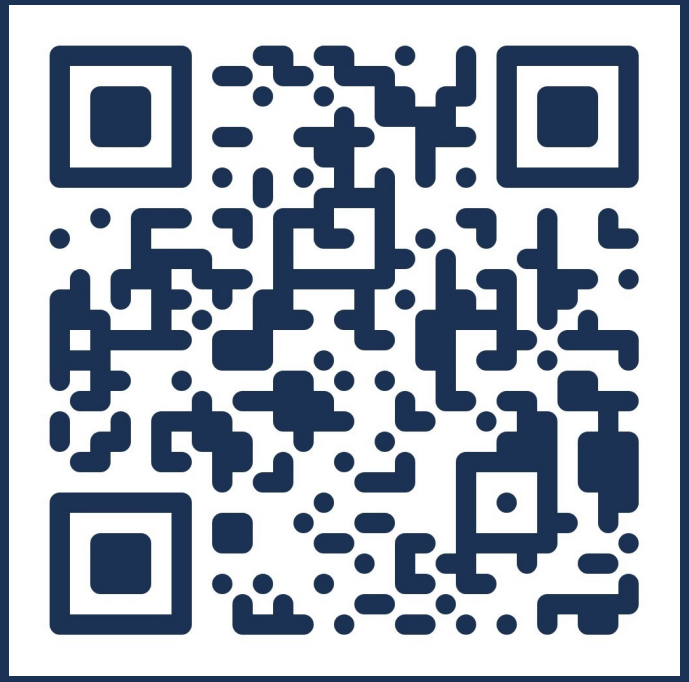
Software Engineering Internship

FPGA Engineering Internship - closed

Quantitative Trading Internship - closed

Links

Compilers coursework intro



Applications

Lecture feedback