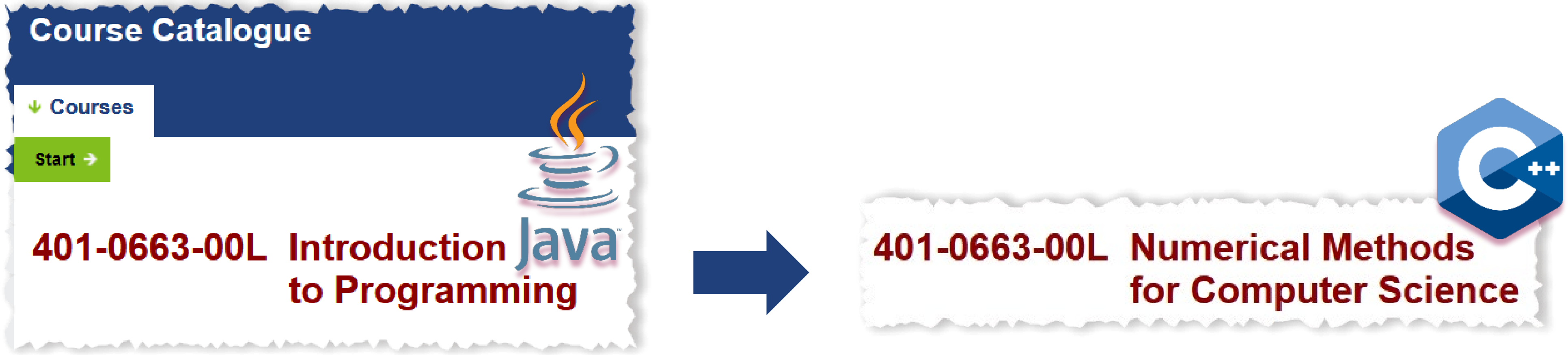
Java to C++ (for NumCS)
1. Core Language

Felix Friedrich

Malte Schwerhoff

D-INFK 2023

This Course



- Goal: ease transition from Java to C++
- Expects initial knowledge in Java
- Tailored to NumCS – not a generic C++ course
- Not self-contained: the gist rather than the details

- Second iteration: **Please interact, ask**, and **give feedback**

Who Is Involved?

Dr. Felix Friedrich

Dr. Malte Schwerhoff

- D-INFK service lecturers
- Experienced C++ lecturers
- Lecturers first two weeks

picture maybe somewhat dated

Dr. Vasile Gradinaru

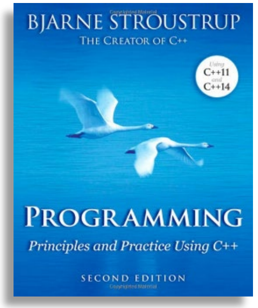
- D-MATH (CSE) lecturer
- Experienced numerical methods lecturer
- Main lecturer (after 2nd week)

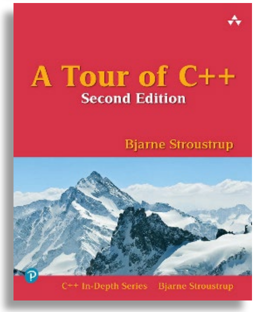
Good Sources of Information (on C++ and Eigen)

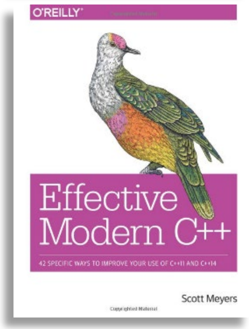
- https://cppreference.com

- Programming: Principles and Practice Using C++
  (Stroustrup; https://u.ethz.ch/2MgzB+)
- A Tour of C++
  (Stroustrup; https://u.ethz.ch/aeH7d+)
- Effective Modern C++
  (Scott Meyers; https://u.ethz.ch/m1DvT+)

- Eigen (linear algebra library): https://eigen.tuxfamily.org/dox/







A Bit of Background on C++

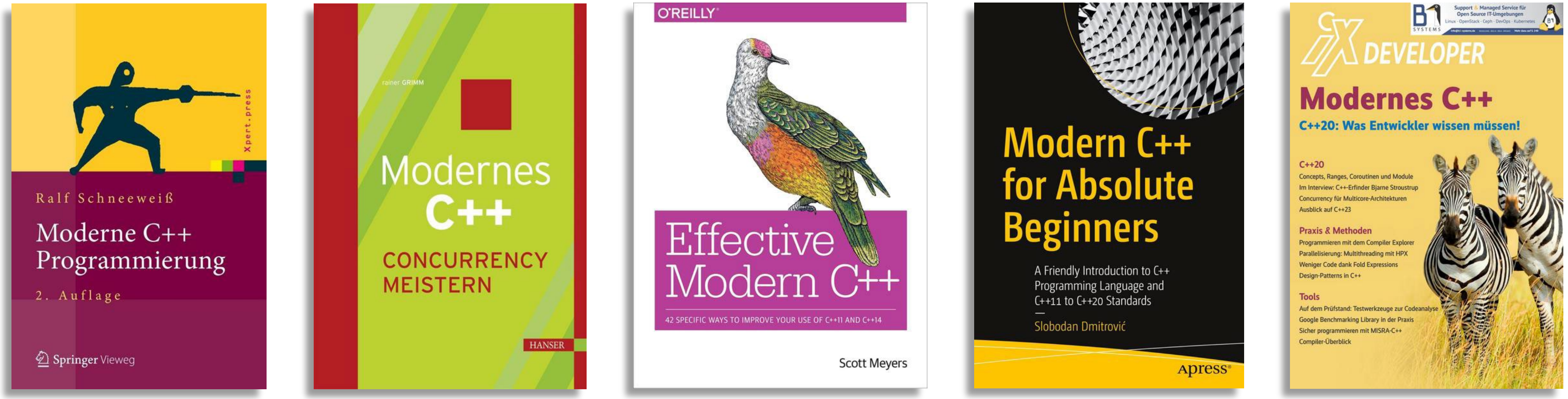
Why C++

- Important language in computational science, engineering, high-performance computing
- NumCS uses C++ to discuss implementations of numerical methods
- Often combined with Python (not this course): user-friendly Python interface to a high-performance C++ implementation

401-0663-00L Numerical Methods for Computer Science

What C++ Is Like

- C++ has a certain reputation … partly deserved
- But modern C++ (≥ C++11, latest standard is C++20) has come a long way



What C++ Is Like

- C++ is not exactly beginner-friendly
  - Hardware and OS “shine through” — great for experts, hard for beginners
  - Many choices — great for experts, hard for beginners
  - Explicit choices — explicate decisions, at cost of syntactical convenience

- But C++ is also not opinionated. Two examples:
  - Java-style OOP with runtime polymorphism;
    but also compile-time polymorphism and static meta-programming
  - Value semantics (think Java primitive types);
    but also reference semantics (think Java objects)

C++ vs. Java – Big-Picture Differences

Compiled to machine code

- Binaries not portable (OS, hardware)
- Static code optimisations

Compiled to intermediate code

- Portable, but requires intermediate layer (JVM)
- Runtime code optimisations

Emphasises efficiency (runtime, memory)

Emphasises safety

Extended C

- Inherited efficiency emphasis … and some design problems
- Largely interoperable with C

Designed from scratch, with C/C++ lessons-learned in mind

What Does C++ Code Look Like?

```cpp
#include <iostream>
#include <limits>

double average(std::istream&);

int main() {
  std::cout << "Enter values to average: ";
  double avg = average(std::cin);
  std::cout << "Average = " << avg << "\n";
}

// Returns the average of numbers read from the given input stream
double average(std::istream& in) {
  double value, sum = 0; // Attention: value is not initialised
  unsigned count = 0;

  while (in >> value) {
    ++count;
    sum += value;
  }

  return count != 0 ? sum / count : std::numeric_limits<double>::signaling_NaN();
}
```

Java vs. C++:
Some Similarities and Differences

Some Similarities between C++ and Java

- Core syntax
  - Case sensitivity, curly braces, parentheses, semicolon, comments
- Operators
  - `+`, `*`, `=`, `+=`, `==`, `++`, `<`, `&&`, ...
  - Same syntax, precedence, short-circuiting, mixed types conversion
- Fundamental types: `int`, `float`, `double`, `char`, `bool`; also `void`
- Control structures: `return`, `if-else`, `switch`, `for`, `while`, ...
- Code modularity
  - Functions, classes
  - Namespaces (and *modules* with C++20)

Differences: Fundamental Types in C++

- Signed and unsigned integral types
  - E.g. `int` vs. `unsigned int` (or just `unsigned`)
- Width of `bool`, `char`, `int`, `long` is implementation-defined, with lower bounds
  - E.g. an `int` is at least 16 bits wide, a `long` 32 bits
  - Fixed-width types introduced with C++11, e.g. `int32_t`
- https://en.cppreference.com/w/cpp/language/types
- https://en.cppreference.com/w/cpp/types/integer

OS	Architecture	Size of "long" type
Windows	IA-32	4 bytes
	Intel® 64	4 bytes
Linux	IA-32	4 bytes
	Intel® 64	8 bytes
mac OS	Intel® 64	8 bytes

Size of 'long integer' data type (C++) on various architectures and OS (intel.com)

The Price of Efficiency and Generality

- Background
  1. C++ aims to enable efficient programs on any hardware & OS
  2. C++ is old (limited language design experience, fewer hardware & OS standards, significantly fewer computational resources)

- Intuitive trade-off: rigidly defined language semantics increase safety, but complicate efficient hardware & OS support

- C++ language standard grants freedom to language implementers
  1. Implementation-defined behaviour: behaviour may vary, must be documented
  2. Unspecified behaviour: implementation-defined, no need to document
  3. Undefined behaviour: if present, program is not required to do anything meaningful

Common Sources of Undefined Behaviour

- *Signed* integer over-/underflow
  - Java: well-defined
- Out-of-bounds array access, e.g. `data[data.size()]`
  - Java: runtime check, exception
  - C++: `data.at(idx)` performs runtime check
- Null-pointer dereferencing
  - Java: runtime check, exception

- Read & write *same memory location* in single expression, e.g. `cout << (x + x++)`
  - Java: well-defined
- Reading uninitialized variables
  - Java: compiler check for local variables, default initialisation for member variables
- Accessing destructed objects/deallocated memory
  - Java: cannot happen, since garbage collector manages memory

Differences: Initialisation

- C++ default-initialises variables iff the type declares a default constructor
  - Applies to local variables, and member variables (object fields)
- No default constructors for fundamental types (int, double, ...)

C++
```cpp
int count; // Not initialised
cout << count; // Undefined behaviour
```

Java
```java
int count; // Not initialised
println(count); // Compiler error
```

- C++: reading uninitialised variables is undefined behaviour
- Java:
  - Guaranteed compiler error for local variables
  - Default initialisation for member variables

Undefined Behaviour

- Compile- or runtime checks might be performed, but no guarantees
- Keep existence in mind
  - To avoid common problems (previous slides)
  - During debugging (undefined behaviour often results in nondeterministic behaviour)

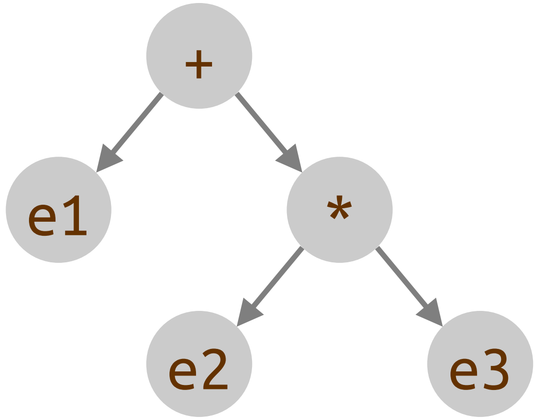
- Otherwise not central for this course
  - We will not go into further details
  - Newer C++ versions sometimes replace *undefined* by *unspecified behaviour*

- If interested: https://en.cppreference.com/w/cpp/language/ub
- Demo of what *may* happen: https://gcc.godbolt.org/z/MsjPnWs5Y

Differences: Expression Evaluation

HANDOUT SLIDE

- Evaluation order of subexpressions
  - *Unspecified behaviour* in C++
  - Specified as left to right in Java
- Compound expression example: `e1 + e2 * e3`
  - Java: `e1` evaluated first, then `e2`, finally `e3`
  - C++: Any order allowed



- Likewise for function call arguments, e.g. `fun(e1, e2, e3)`
- Relevant (only) when expressions have side-effects

Differences: Automatic Numerical Conversions

HANDOUT SLIDE

- Automatic conversion to `bool`
  - In C++, variables of fundamental type can be used as `bool` without explicit casting
- Automatic narrowing between numeric types
  - In C++, e.g. assigning a `double` to an `int` does not require an explicit cast
- Automatic numerical promotions in mixed numerical expressions
  - E.g. `some_short * some_double + some_int` → all values promoted to doubles
  - Rules analogous to Java's: https://en.cppreference.com/w/cpp/language/operator_arithmetic#Conversions
  - C++-specific: signed promoted to unsigned → `-1 < 1u` is false (https://www.modernescpp.com/index.php/safe-comparisons-of-integrals-with-c-20)

Differences: Type Inference

- C++ is statically typed, just like Java
- Limited support for type inference includes
  - Type parameter interference → templates, 2nd week
  - Keyword `auto`

```
std::vector<int> data = ...;
std::vector<int>::iterator head = data.begin();
```

vs.

```
std::vector<int> data = ...;
auto head = data.begin();
```

- Types serve as code documentation → don't overuse `auto`

```
auto model = extract_model(...); // What's model? Which functions does it provide?
```

Values vs. References

Value Types / Pass by Value

- Consider the example on the right
- Can you explain the observed behaviour?

```
void swap(int x, int y) {
  int tmp = x;
  x = y;
  y = tmp;
}

int main() {
  int x = 1;
  int y = 2;
  swap(x, y);
  std::cout << x << ", " << y; // 1, 2
}
```

just as in Java

Value Types / Pass by Value

- Consider the extended example. Can you explain the observed behaviour?

```
struct Coordinate { // Java class
  int x;
  int y;

  Coordinate(int a, int b) {
    x = a;
    y = b;
  }
};
```

```
void swap(Coordinate c) {
  int tmp = c.x;
  c.x = c.y;
  c.y = tmp;
}

int main() {
  Coordinate c(1, 2); // Obj. construction
  swap(c);
  std::cout << c.x << ", " << c.y; // 1, 2
}
```

not as in Java

Value Types / Pass by Value

- By default, types in C++ are value types
  - Fundamental types (`int`, etc.): *same* as in Java
  - Instances of classes: *different* from Java

- Value types (of statically-known size) can be allocated on the stack
  - Typically faster than heap accesses

- Pass by value: values are copied upon function calls
- Return by value: analogous

Outlook: Values vs. Pointers

- Java's *references* are (a simplified version of) *pointers* in C++

```
Coordinate* location = new Coordinate(1, 2);
```

  - The pointer value (memory address) is still passed by value (i.e. copied), as in Java

- C++ has no garbage collector → manual memory management
  - Every `new` needs a `delete`
  - Danger of memory leaks, accessing deleted objects, freeing memory twice
  - *Smart pointers* simplify memory management (while preserving efficiency)

- Pointers are not relevant for NumCS → not discussed further

Pass by Reference / Reference Types

- Pass by value not always desirable
  1. Effects: modifying a shared object (e.g. a shared counter in a concurrent system)
  2. Efficiency: avoiding copying large data (e.g. a huge matrix)

- Values can be shared via C++ references
  - Still no heap involved (efficiency)

Pass by Reference / Reference Types

- Pass-by-reference semantics with C++ references

```
struct Coordinate {
  ... // same as before
};
```

```
void swap(Coordinate& c) {
  int tmp = c.x;
  c.x = c.y;
  c.y = tmp;
}

int main() {
  Coordinate c(1, 2);
  swap(c);
  std::cout << c.x << ", " << c.y; // 2, 1
}
```

just as in Java

Pass by Reference / Reference Types

- References *can* (nearly) be used everywhere, e.g.

```
int anakin_skywalker = 7; // Happy birthday!
int& darth_vader = anakin_skywalker;
darth_vader = 40; // Oh oh, Anakin rapidly aged ...
```

- Reference types are *typically* used
  - As function parameters
  - In lambda expressions (2nd week)
  - As member variables (classes, this Thursday)

Pass by Reference / Reference Types

- References must be initialised

```
int& darth_vader; // Compiler error
```

- References cannot be re-aliased:

```
int& darth_vader = anakin_skywalker;
darth_vader = kermit_the_frog;
  // vader (and thus anakin) gets kermit's value,
  // but vader does not become an alias of kermit
```

Pass by Reference / Reference Types – With Fundamental Types

```
void swap(int& x, int& y) {
  int tmp = x;
  x = y;
  y = tmp;
}

int main() {
  int x = 1;
  int y = 2;
  swap(x, y);
  std::cout << x << ", " << y; // 2, 1
}
```

- Can you do this in Java?

Constants

- Java's `final` is `const` in C++

```
const unsigned speed_of_light = 299792458;
...
speed_of_light *= 2; // Compiler error
```

- A program is const-correct if all variables not intended to be mutated are typed as `const`

- Concept lifted to *constant expressions* (= evaluable at runtime) to support compile-time meta-programming
  - No further details here
  - If interested: https://en.cppreference.com/w/cpp/language/constant_expression

Const-References

- Consider the code below. Can it be improved?

```
bool is_sorted(std::vector<int> v) {
  for (unsigned i = 1; i < v.size(); ++i) {
    if (v[i] < v[i-1])
      return false;
  }

  return true;
}


std::vector<int> v(10'000'000); // Size 10e7
...
if (is_sorted(v)) ...
```

Const-References

- Yes: const-reference for efficiency (no copy) and safety (no mutation)

```
bool is_sorted(const std::vector<int>& v) {
  // ... as before ...
}


std::vector<int> v(10'000'000);
...
if (is_sorted(v)) ...
```

Functions & Operators

Functions

- Core syntax as in Java: R `name(`$T1\ p1$`, …, `$Tn\ pn$`) {` *body* `}`

- Function overloading

```
void process(int data) { ... }
void process(double data) { ... }

int x = ...;

process(x); // calls 1st function
process(3.13); // calls 2nd function
```

- Default parameters

```
// Returns true if data contains elem
// at least rep times
bool occurs(
    int elem,
    std::vector<int> data,
    unsigned rep = 1) {

  ...
}

std::cout << occurs(0, my_data);
std::cout << occurs(77, my_data, 2);
```

Functions

HANDOUT SLIDE

- Variadic functions

```
double average(double values...) {
  // compute average of given values
}

double avg1 = average(2.5, 1.7);
double avg2 = average(avg1, 13, 7.4, 28.9);
```

- Declaration very similar to Java – looks nice …
- … but usage inside the function does not, thus omitted
- https://en.cppreference.com/w/cpp/utility/variadic
- Alternative: variadic *templates* (2nd week)

Operator Overloading

- In contrast to Java, C++ supports overloading of operators

```
struct Rational {
  int n; int d;

  Rational(int _n, int _d) { n = _n; d = _d; }
};
```

```
Rational r1(1, 3);
Rational r2(5, 7);

std::cout << (r1 + r2);
```

```
Rational operator+(const Rational& l, const Rational& r) {
  return Rational(l.n * r.d + l.d * r.n,  l.d * r.d);
}
```

```
std::ostream& operator<<(std::ostream out&, const Rational& r) {
  out << r.n << ", " << r.d;
  return out; // enables chaining of <<
}
```

- https://en.cppreference.com/w/cpp/language/operators

Separate Compilation

Semantics of #include

main.cpp

```cpp
#include <rational.cpp>

int main() {
  Rational r1 = ...;
  ...
}
```

rational.cpp

```cpp
struct Rational {
  int n;
  int d;

  Rational(int a, int b) {
    n = a;
    d = b;
  }
};

...
```

compile main.cpp

main.cpp (as compiled)

```cpp
struct Rational {
  int n;
  int d;

  Rational(int a, int b) {
    n = a;
    d = b;
  }
};

...

int main() {
  Rational r1 = ...;
  ...
}
```

- `#include <file.cpp>` is replaced with the content of `file.cpp`
- Possible disadvantages?

Includes and Separate Compilation

- Fact: `#include <lib.cpp>` is replaced with the contents of file `lib.cpp`
- New goal: Enable separate compilation of units of code (e.g. classes, libraries); resulting machine code then linked afterwards
  - Improves performance (shared library only compiled once)
  - Allows binary distributions (e.g. closed-source libraries)
- Question: Given how `#include` works, how to enable separate compilation?
- C++ solution: Separation of declarations from definitions in separate files
  - Header files `.h` contain declarations, implementations reside in `.cpp` files

Header vs. Implementation

main.cpp

```
#include <rat.h>

int main() {
  Rat r1 = ...;
  ...
}
```

rat.h

```
struct Rat {
  int n;
  int d;

  Rat(int a, int b);
};

Rat operator+(
  const Rat& l,
  const Rat& r);

// more declarations ...
```

rat.cpp

```
Rat::Rat(int a, int b) {
    n = a;
    d = b;
  }
};

Rat operator+(const Rat& l, const Rat& r) {
  return Rat(l.n * r.d + l.d * r.n, l.d * r.d);
}

// more implementations ...
```

- Declarations suffice for compilation

- `rat.cpp` and `main.cpp` can be compiled separately

  `$ g++ -c main.cpp` (produces `main.o`)

  `$ g++ -c rat.cpp` (produces `rat.o`)

- … and linked together later

  `$ g++ rat.o main.o -o main` (produces executable `main`)

Preventing Re-inclusion

- If `rat.h` is (transitively) included multiple times, symbols are redeclared
- Avoid by using *pre-processor macros* (not discussed further) to establish include guards:

rat.h
```
#ifndef RAT_H
#define RAT_H

struct Rat {
  int n;
  int d;

  Rat(int a, int b);
};

// more declarations ...

#endif // RAT.H
```

- Alternative using *preprocessor directive* (not discussed further)
  - Certain advantages over include guards
  - Not supported by all compilers

rat.h
```
#pragma once

struct Rat {
  int n;
  int d;

  Rat(int a, int b);
};

// more declarations ...
```

Separate Compilation on Code Expert

- Code Expert environments typically don’t give you access to the compiler command-line
  - Advantage: simpler to use, less potential for mistakes
  - Disadvantage: separate compilation might not be possible

- Probably only relevant if you create your own files
  (in contrast to completing given skeleton files)