Java to C++ (for NumCS)

2. Classes, Standard Library, Iterators and Containers, Eigen

Felix Friedrich

Malte Schwerhoff

D-INFK 2023

Classes

C++ Class Example

- Classes in C++ resemble, but are not identical to, Java classes

```
class rational {                                              rational.h
  int n;
  int d; // INV d != 0

public:
  rational(int nom, int den); // Most general constructor
  rational(int n); // Conversion from int
  rational(); // Default constructor

  // Operators
  rational& operator+=(const rational& r);
  bool operator==(const rational&r) const;

private:
  void print(std::ostream& out) const;
  void parse(std::istream& in);
};
```

Analogous to Java

- Default visibility is `private`
- Member variables (e.g. `int n`)
- Member functions (e.g. `print()`)
- Member function overloading
  (in this case, constructor overloading)

Different from Java

- Member operators
  - Merely glorified member functions
  - Recall last lecture on operator overloading
- Const member functions

Separation of Declaration and Definition

- For completeness: include guards (recall last lecture)

rational.h

```
#ifndef RATIONAL_H
#define RATIONAL_H

class rational {
  int n;
  int d;
  ...
};

#endif // RATIONAL_H
```

Separation of Declaration and Definition

rational.h

```
class rational {
  int n;
  int d; // INV d != 0

public:
  rational(int nom, int den);
  rational(int n);
  rational();

  rational& operator+=(const rational& r);
  bool operator==(const rational&r) const;

private:
  void print(std::ostream& out) const;
  void parse(std::istream& in);
};
```

rational.cpp

```
rational& rational::operator+=(const rational& r) {
  n = n * r.d + d * r.n;
  d *= r.d;
  return *this;
}

// Parses rational from string "<int>:<int>"
void rational::parse(std::istream& in) {
  char ignore;
  in >> n >> ignore >> d;
  assert(ignore == ':');
  assert(d != 0);
}

...
```

access the (namespace of) class rational

return receiver object (for operator chaining)

For space reasons, we won’t always separate declarations from definitions on slides

Interlude: Classes vs. Structs

- The *only* technical difference between `struct` and `class` is default visibility
  - Class members are private by default, struct members public

```
class rational {
  int n;
  int d;

public:
  rational(int nom, int den);

  ...
};
```

==

```
struct rational {
private:
  int n;
  int d;

public:
  rational(int nom, int den);

  ...
};
```

- Common pattern (not a technical decision)
  - Structs hold data, but provide little functionality (e.g. nodes of a linked list)
  - Classes for functionality (e.g. a `List` container, with `append(elem)` etc.)

Accessing Members

- Unqualified access
  - E.g. here: just `n`
  - Accesses *this-receiver's* member variables
  - Ambiguous if shadowed, e.g. by local variable

- Qualified access to *other* receiver
  - E.g. here: `r.n`
  - Accesses member variable `n` of object `r`
    (if r is value- or reference-typed)

```
class rational {
  int n;
  int d; // INV d != 0

public:
  rational& operator+=(const rational& r) {
    n = n * r.d + d * r.n;
    d *= r.d;
    return *this;
  }

  ...
};
```

- Qualified access to *this* receiver
  - E.g. `this->n` instead of just `n`; equivalent to `(*this).n`
  - `this` is a pointer (Java-style reference), not a C++ reference
  - No discussion of pointers, since irrelevant for NumCS

- Member functions
  - Analogous
  - E.g. just `print(…)`,
    or `this->print(…)`

Constructors: Member Variable Initialisation

- Looks different than in Java

```
rational::rational(int nom, int den)
  : n(nom), d(den) // initialisation
{} // empty body
```

- Behaves analogously

```
rational r(2, 3);
  // Now r.n == 2, r.d == 3
```

```
class rational {
  int n;
  int d;

public:
  rational(int nom, int den);
  rational(int nom);
  rational();

  ...
};
```

- Use initialiser lists (left), not assignments (reason: performance)

```
rational::rational(int nom, int den)
  : n(nom), d(den)
{}
```

```
rational::rational(int nom, int den) {
  n = nom; d = den;
}
```

Constructors: Calling Other Constructors

```
rational::rational(int nom, int den)
  : n(nom), d(den)
{}
```

- Constructor delegation

```
rational::rational(int nom): rational(nom, 1) {}

rational::rational(): rational(0) {}
```

```
class rational {
  int n;
  int d;

public:
  rational(int nom, int den);
  rational(int nom);
  rational();

  ...
};
```

Constructors: Call-Sites

```
class rational {
public:
  rational(int nom, int den); // c1
  rational(int nom); // c2
  rational(); // c3
  rational(const rational& r); // c4

  ...
};
```

- Calling constructor c1 (calls behaviourally equivalent since C++17)

```
rational r1a(2, 3);
rational r1b = rational(2, 3);
auto r1c = rational(2, 3);
```

- Calling converting constructor c2

```
rational r2a = 5;
// or explicit forms, as for c1
```

- Calling default constructor c3

```
rational r3a;
// or explicit forms, as for c1
// except for rational r3a()
```

- Calling copy constructor c4

```
rational r4a = r1a;
// or explicit forms, as for c1
```

Constructors: Call-Sites

- The fun doesn’t stop there (but we do)

```
std::vector<int> data = {1, 2, 3}; // list-initialization, useful for containers
int i(1.2); // ok (implicit narrowing allowed)
int j{1.2}; // error (explicit cast needed)
```

- Our recommendation:
  - Follow course material, and code style used there
  - Use syntax intuitive for/familiar to you (if not instructed otherwise)

Constructors: Call-Sites

C++: Rules for Different Ways of Initialization

		always has defined value	narrowing is error	works for `initializer_list<>`	`explicit` conversion supported	works for aggregates	works for `auto`	works for members
	Type `i;`	no	-	no	-	✓ (no init)	no	✓
	Type `i{};`	✓	-	✓	-	✓	no	✓
	Type `i();`	function declaration						
direct initialization	*Type* `i{x};`	✓	✓[1]	✓	✓	✓	✓[2]	✓
	Type `i(x);`	✓	no	no	✓	since C++20, not nested	✓	no
	Type `i(x, y);`	✓ (2 args)	no	no	✓	since C++20, not nested	✓	no
copy initialization	*Type* `i = x;`	✓	no	no	no	no	✓	✓
	Type `i = {x};`	✓	✓[1]	✓	no	✓	✓ init-list	✓
	Type `i = (x);`	✓ (1 arg)	no	no	no	since C++20, not nested	✓ (1 arg)	✓ (1 arg)
	Type `i = (x, y);`	✓ (last arg)	no	no	no	since C++20, not nested	✓ (last arg)	✓ (last arg)

[1]: **g++** needs **`-pedantic-errors`** or **`-Werror=narrowing`** to detect narrowing errors

[2]: **`std::initializer_list<>`** before g++ 5, clang 3.8, and Visual Studio 2015

josuttis.com/cpp/c++initialization.pdf

@NicoJosuttis

Initialisation vs. Assignments

- Compare the two C++ snippets s1, s2 below
  - What is the operational behaviour?
  - Are they equivalent to each other?

s1

```
rational r =
  rational(2, 3);
```

s2

```
rational r;
r = rational(2, 3);
```

```
class rational {
  int n;
  int d;

public:
  rational(int nom, int den);
  rational();
  ...
};
```

Initialisation vs. Assignments

- Compare the two C++ snippets s1, s2 below
  - What is the operational behaviour?
  - Are they equivalent to each other?

s1
```
rational r =
  rational(2, 3);
```

s2
```
rational r;
r = rational(2, 3);
```

```
class rational {
  int n;
  int d;

public:
  rational(int nom, int den);
  rational();
  ...
};
```

- Different syntactical forms, same behaviours

s1
```
rational r(2, 3);
```

s2
```
rational r/*()*/;
rational tmp(2, 3);
r = tmp;
```

s2
```
rational r/*()*/;
rational tmp(2, 3);
r.operator=(tmp);
```

Initialisation vs. Assignments

```
rational r =
  rational(2, 3);
```

→

```
rational r(2, 3);
```

1. New object `r` initialised via `c1`

```
rational r;
r = rational(2, 3);
```

→

```
rational r/*()*/;
rational tmp(2, 3);
r.operator=(tmp);
```

1. New object `r` initialised via `c2`
2. New object `tmp` initialised via `c1`
3. `r` gets `tmp`’s content

```
class rational {
  int n;
  int d;

public:
  rational(int nom, int den); // c1
  rational(); // c2

  // Assignment operator (simplified)
  void operator=(const rational& r) {
    n = r.n;
    d = r.d;
  }
  ...
};
```

Initialisation vs. Assignments

Take-away message: adjust your Java mindset

- Java-style references (i.e. C++ pointers) are an *indirection*
  - Variables hold memory addresses, i.e. indirectly point to objects
  - Assignments swing the pointer, but do not affect the pointed-to object
- No indirection with C++ value types
  - Variables directly hold objects
  - Assignments must thus directly affect objects

Compiler-Generated Member Functions

```
class rational {
public:
  int n;
  int d;
};
```

Compiler generates various default implementations, e.g.

- Default constructor `rational r;`
  (`n`, `d` remain uninitialised, since no def. constr. for `int`)
- Copy constructor `auto r2 = rational(r1);`
- Member-wise constructor `rational r = {1, 2};`
  (but only if all member variables public)
- Assignment operator `r1 = r2;`

Keyword `delete` can be used to prevent compiler-generation
(https://en.cppreference.com/w/cpp/language/function#Deleted_functions)

Compiler-Generated Member Functions

- Convenient, if mostly boilerplate code
- But several conditions apply, e.g.
  - Def. constr. only generated if all member variables’ types are default-constructable
  - Member-wise constructor not generated if (at least) one member variable is private
  - If default constructor is user-provided, member-wise constructor is not generated
- Behaviour can cause confusion – keep in mind while debugging
  - Code works – because of compiler-generated functions
  - Seemingly unrelated change breaks it – since used functions no longer generated

Const Member Functions

```
class rational {
  int n;
  int d;

public:
  bool operator==(const rational& r) const {
    return n == r.n && d == r.d;
  };

  rational& operator=(const rational& r) {
    n = r.n;
    ...
  }

  ...
};
```

- Equality operator is `const`
  - Cannot modify receiver's member variables (`this->n/d`)
  - Stronger guarantee to callers
- Assignment operator cannot be `const`
  - `const` argument still reasonable

Const Member Functions

- Only const member functions can be invoked on a const receiver

```
const rational r1(1, 2);
rational r2 = r1;
assert(r1 == r2);
r1 = r2;
```

- Per line above
  - What happens?
  - Problem or OK?

```
class rational {
  int n;
  int d;

public:
  rational(int nom, int den);
  rational(const rational& r);
  ...
  bool operator==(const rational& r) const;
  rational& operator=(const rational& r);
  ...
};
```

Const Member Functions

- Only const member functions can be invoked on a const receiver

```
const rational r1(1, 2);
rational r2 = r1;

// r1.operator==(r2) - ok
assert(r1 == r2);

// r1.operator=(r2) - compiler error
r1 = r2;
```

```
class rational {
  int n;
  int d;

public:
  rational(int nom, int den);
  rational(const rational& r);
  ...
  bool operator==(const rational& r) const;
  rational& operator=(const rational& r);
  ...
};
```

- Observations
  - Passing `const r1` to copy-constructor allowed, since it takes a `const`
  - Calling equality operator on r1 allowed, since operator is `const`
  - Calling assignment operator on r1 forbidden, since not `const`

Classes: Conclusion

- Many aspects similar to Java, e.g. constructors, members, visibility
- But value types necessitate different behaviour, and technical machinery
  - Don’t debug with Java-style references in your head
- More differences introduced by combining value types with, e.g.
  - Pointers (Java-style references/dynamically allocated memory)
  - Subtyping and inheritance
- All interesting stuff … but what you saw today should suffice for NumCS

Object Lifetimes and References

Static Memory: Scope-Based Lifetimes

- Statically allocated objects have scope-based lifetimes (in C++)
  - Values (what we have seen so far) are always statically allocated

- Compiler can insert deallocation code to automatically destroy object at scope end
  - Sounds great: efficient (no garbage collector) and convenient

```
void print(rational rat1) {
  ...
}

int main() {
  ...
  rational rat2(1, 2);

  if (...) {
    rational rat3(1, 2);
    print(rat3);
    ...
  }
  ...
}
```

Dynamic Memory: Custom Lifetimes

- Scope-based lifetime not always suitable
  - E.g. new node should not be destroyed when call to `add()` terminates
- Dynamically allocated memory
  - `Node* n = new node(elem)` to create
  - `delete n` to destroy
  - Manual memory management is very error-prone
    → garbage collectors (Java), ownership types (Rust), smart pointers (C++)
- Irrelevant for NumCS, no further discussion

```
class linked_list {
  ...
  add(int elem) {
    node n = node(elem);
    // ... append n as new
    // last node ...
  }
  ...
};
```

Scope-Based Lifetimes and References

- Objects are destroyed at scope end → problem with references?

```
void print(const rational& r1) {
  std::cout << r1.n << ...;
}

int main() {
  rational r2(1, 2);
  print(r2);
  ...
}
```

- How about the code to the right?
- Not a problem: the lifetime of an object from a caller's scope (`r2`) always exceeds the duration of the call

Scope-Based Lifetimes and References

- Objects are destroyed at scope end → problem with references?

```
rational& get_next() {
  rational r1(...);
  return r1;
}

int main() {
  auto r2 = get_next();
  std::cout << r2.n << ...;
}
```

- How about the code to the right?
- **Problem:**
  - `r1` is deallocated at end of `get_next()`
  - returned reference is thus invalid
    → undefined behaviour

- Similar danger with reference-typed member variables

Scope-Based Lifetimes and References

- Objects are destroyed at scope end → risk of "zombie" references
  - Using such a reference is undefined behaviour
  - Compiler might warn you (but probably only in obvious cases)
- It is your responsibility to ensure that objects remain alive (in scope) longer than any reference to them
- In particular, never *return* a reference to a local object
  - Might be the only danger in NumCS

```
rational& get_next() {
  rational r1(...);
  return r1;
}
```

Why Return by Reference?

- Give callers access to values inside other values
  - Canonical example: containers
  - In general: object graphs

vector full of *values*

```
std::vector<rational> data =
    {rational(1, 2), rational(3, 4)};
```

```
data[0] = rational(5, 6);
```

reference to value *inside* container

```
rational& e0 = data[0];
rational tmp = rational(5, 6);
e0.operator=(tmp);
```

mutates that value

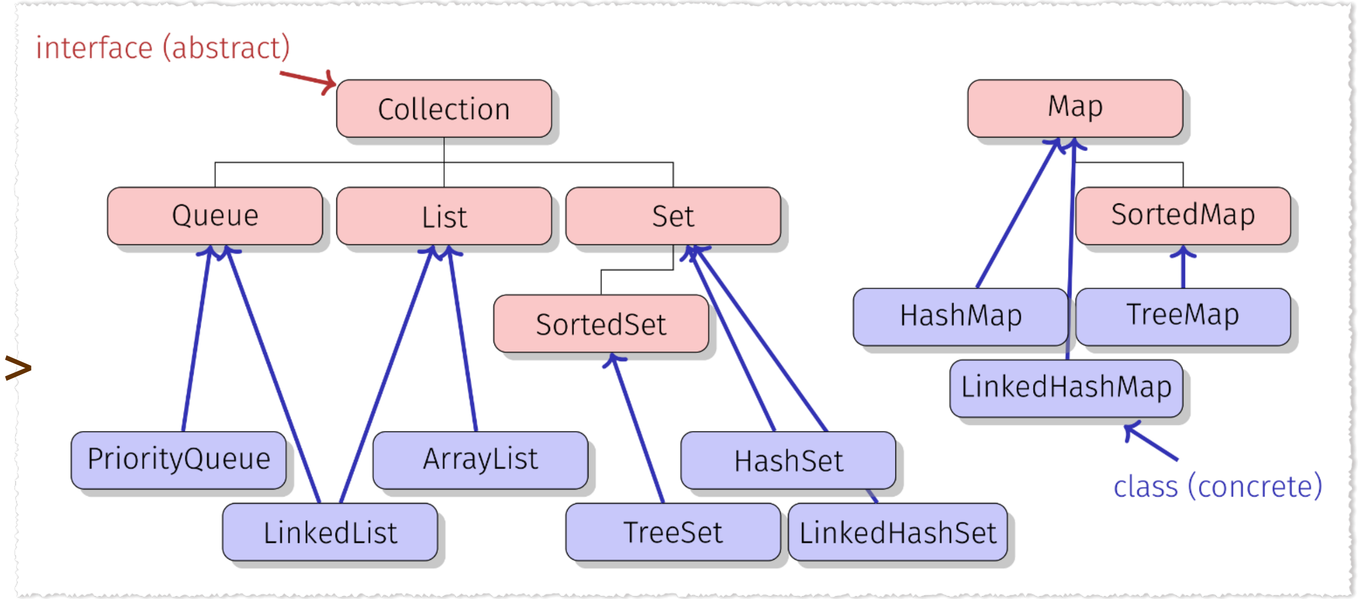
```
T& std::vector<T>::operator[](unsigned idx);
```

- Without return by reference
  - a copy of the value would be returned
  - mutations would not affect the (values in the) vector

Containers

Java Collections

- Working horse of every-day programming
- Different classes have different properties, e.g.
  - Collection of values vs. map from keys to values
  - Ordered vs. unordered
  - Duplicates (dis)allowed
  - Efficient insert vs. find
- Parametric in their elements
  - `List<Rational>` vs. `List<String>`



C++ Containers

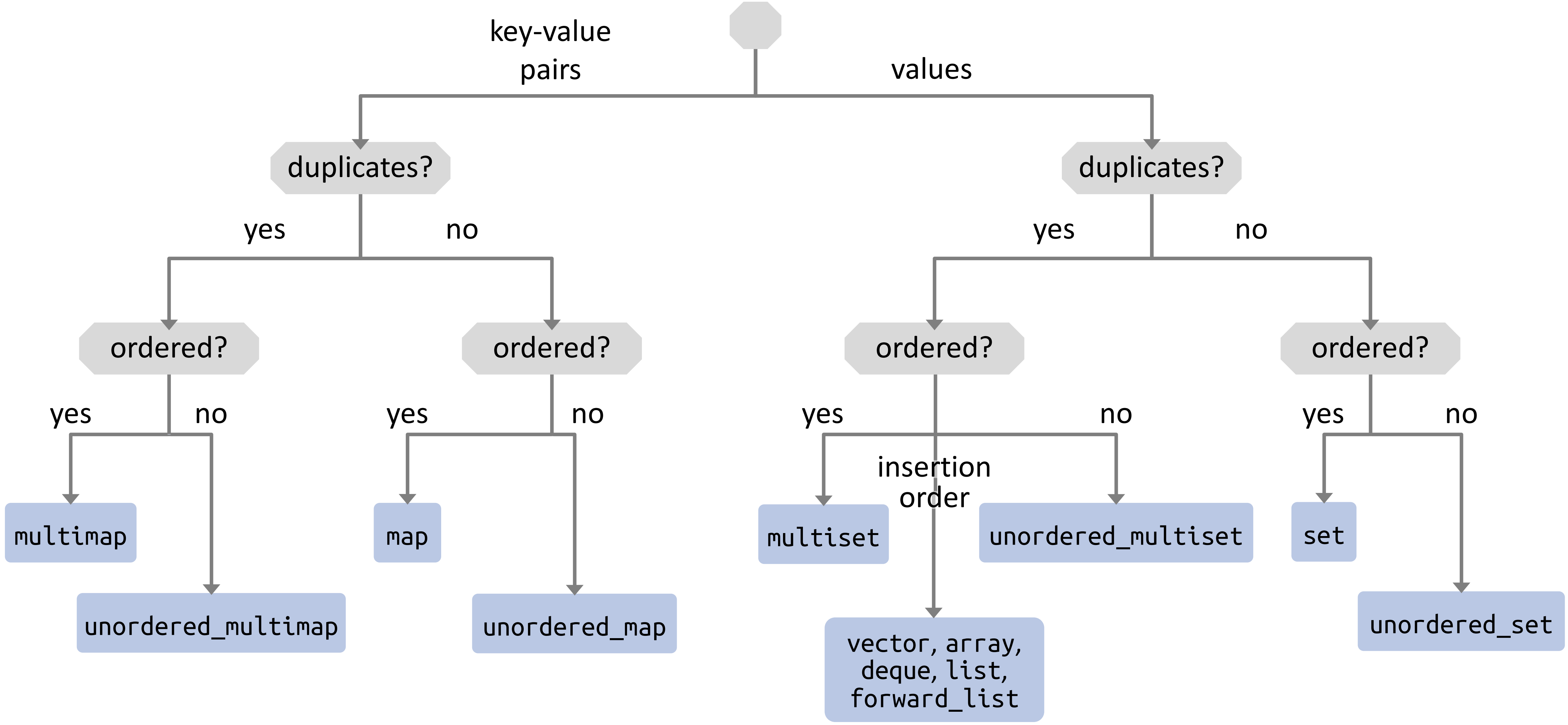
- Part of C++ standard library
  - https://en.cppreference.com/w/cpp/container

- Analogous to Java collections
  - Parametric in their elements
  - Different containers have different properties

- Standard library includes generic algorithms on the containers
  - E.g. `find()`, `fill()`, `partition()`, `sort()`
  - https://en.cppreference.com/w/cpp/algorithm

- Important related concept: iterators (similar idea as in Java)

C++ Containers: Element-related properties



C++ Containers: Functionality and Efficiency

- Offered functionality varies between containers.
  E.g. index-based access `data[i]`:
  - `std::vector<T>`: yes
  - `std::set<T>`: no

- Efficiency (time and space) varies between containers.
  E.g. searching an element (in a container of size n):
  - `std::vector<T>`: $O(n)$
  - `std::unordered_set<T>`: $O(1)$
  - `std::set<T>`: $O(\log n)$

- All similar to Java, no surprise here

Iterators

```
std::vector<int> data = {3, 1, -14, 1, 5, 9};
std::vector<int>::iterator first = data.begin();
auto past_the_end = data.end();
```

type name

first element

after last element

- https://en.cppreference.com/w/cpp/iterator
- Same general idea as in Java
- Each container provides a *container-specific* iterator, and member functions `begin()` and `end()`

Iterators

- Iterators support *at least* forwards container traversal

C++

```cpp
std::vector<int> data =
    {3, 1, -14, 1, 5, 9};

for (auto it = data.begin();
      it != data.end();
      ++it) {

  std::cout << *it << ' ';
}
```

start at first element

stop if past end

advance by one

access underlying element

Java

```java
ArrayList<Integer> data =
    ...;

for (Iterator<Integer> iter = list.iterator();
      iter.hasNext();
      ) {

  Integer elem = iter.next();
  println(elem);
}
```

Interlude: Range-Based for-Loop

- Compiler desugares range-based for-loop into iterator-based loop

```
for (auto elem : data) {
  std::cout << elem << ' ';
}
```

➡

```
for (auto it = data.begin();
     it != data.end();
     ++it) {

  std::cout << *it << ' ';
}
```

- Keep value semantics in mind:
  avoid copying elements by taking references – and play it const-safe

```
for (auto& elem : data) {
  std::cout << elem << ' ';
}
```

```
for (const auto& elem : data) {
  std::cout << elem << ' ';
}
```

Iterators

- Similar to Java, but in wider use, and *potentially* more powerful

```
std::vector<int> data = {3, 1, -14, 1, 5, 9};

auto max_it = std::max_element(data.begin(), data.end());

std::cout << "max_element " << *max_it
          << " at " << std::distance(data.begin(), max_it);

auto shorter_end = data.end() - 2;
auto found_it = std::find(data.begin(), shorter_end, 9);

if (found_it == shorter_end) {
  std::cout << "element not found";
} else {
  std::cout << "found element " << *found_it;
  data.erase(found_it);
}
```

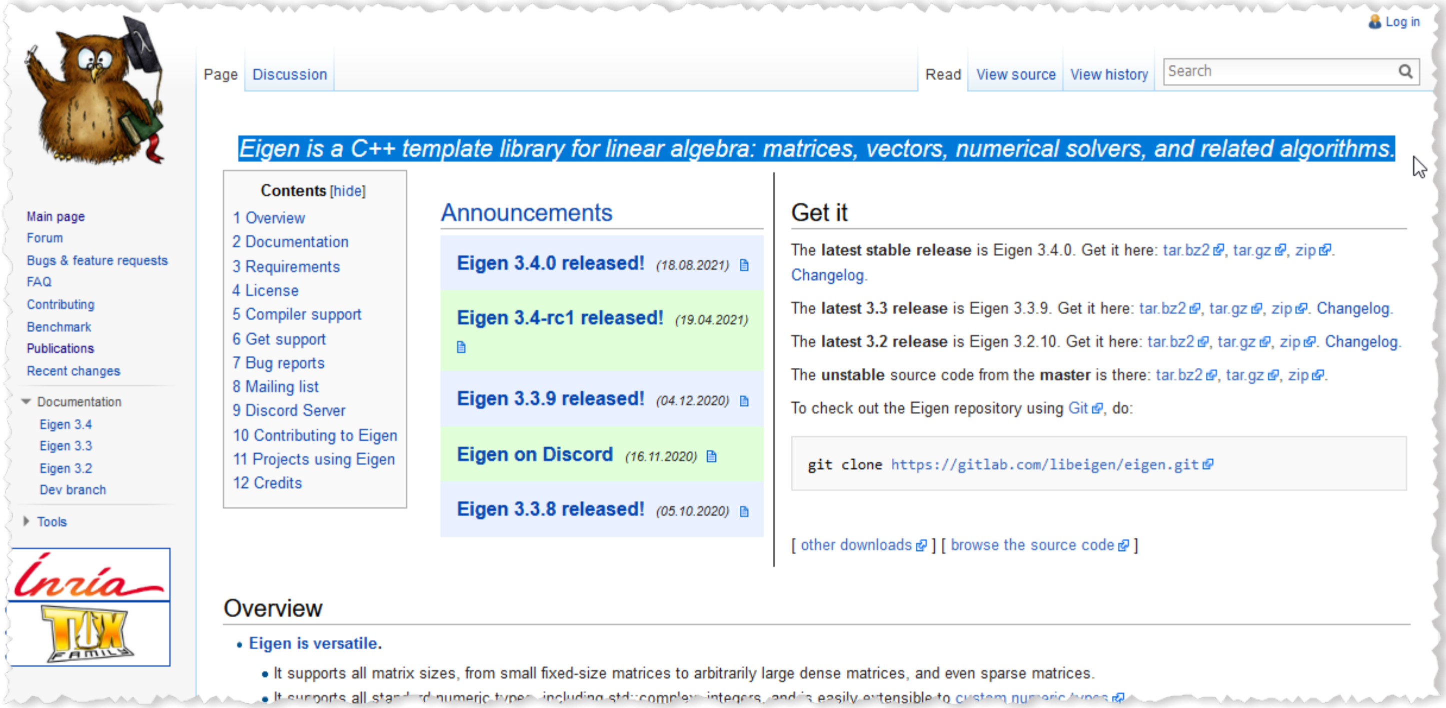
- Many functions (e.g. `std::distance`, `std::find`, `std::vector<T>::erase`) work with iterators
- Functionality of iterators varies between containers
- E.g. `it - n` or `it1 < it2` not always supported

Eigen

(only a mini-teaser)

What is Eigen?

- https://eigen.tuxfamily.org/
- C++ template library for linear algebra:
  - matrices, vectors
  - numerical solvers
  - related algorithms



- Eigen is
  - versatile (e.g. matrix shapes; dense, sparse; reals or complex)
  - fast (e.g. vectorisation)
  - reliable, and more

Eigen Examples

Example:

```
#include <iostream>
#include <Eigen/Dense>

int main() {
  Eigen::Matrix2d a;
  a << 1, 2,
       3, 4;
  Eigen::Vector3d v(1,2,3);
  std::cout << "a * 2.5 =\n" << a * 2.5 << '\n';
  std::cout << "0.1 * v =\n" << 0.1 * v << '\n';
  std::cout << "Doing v *= 2;" << '\n';
  v *= 2;
  std::cout << "Now v =\n" << v << '\n';
}
```

Output:

```
a * 2.5 =
2.5   5
7.5  10
0.1 * v =
0.1
0.2
0.3
Doing v *= 2;
Now v =
2
4
6
```

- Value types
- Lots of operator overloading

Eigen Examples

Example:

```
ArrayXXi A = ArrayXXi::Random(4,4).abs();
cout << "Here is the initial matrix A:\n" << A << "\n";
for(auto row : A.rowwise())
  std::sort(row.begin(), row.end());
cout << "Here is the sorted matrix A:\n" << A << "\n";
```

Output:

```
Here is the initial matrix A:
7 9 5 3
2 6 1 0
6 3 0 9
6 6 3 9
Here is the sorted matrix A:
3 5 7 9
0 1 2 6
0 3 6 9
3 6 6 9
```

- Iterators to integrate with, e.g. range-based `for`-loop, `std::sort`