HPI Hasso Plattner Institut
IT Systems Engineering | Universität Potsdam

WOCHE 2

DYOD

AGENDA

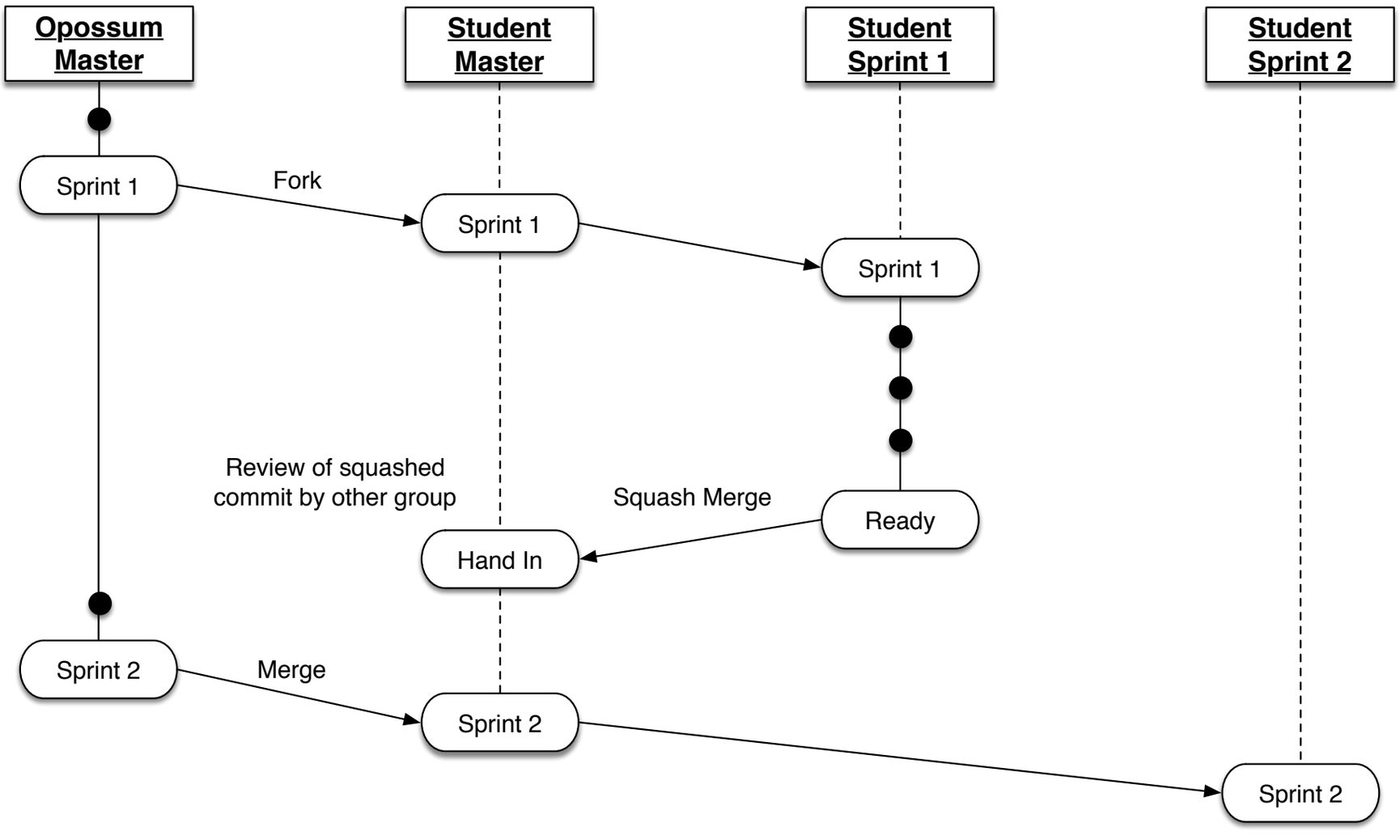
- Organization
- Templates
- RAII
- Smart Pointers
- Dictionary Encoding

ORGANIZATION

- Next week: Reformationstag -> No class
- Did you organize in groups yet?
- If you have not joined us at Piazza
  - piazza.com/hpi.uni-potsdam.de/fall2018/dyod
- Any problems during setup?

ORGANIZATION

- Sprint review organization



AGENDA

- Organization
- **Templates**
- RAII
- Smart Pointers
- Dictionary Encoding

TEMPLATES – FUNCTIONS

```
1 template <typename T> T multiply(T x, T y) {
2   return x * y;
3 }
4
5 double a = 4.0, b = 5.0;
6 multiply<double>(a, b);
7
8 int c = 7, d = 8;
9 multiply<int>(c, d);
```

What would need to change to allow multiplication of Ints and Doubles?

TEMPLATES – FUNCTIONS

```
1 template <typename T> T multiply(T x, T y) {
2   return x * y;
3 }
4
5 double a = 4.0, b = 5.0;
6 multiply<double>(a, b);
7
8 int c = 7, d = 8;
9 multiply<int>(c, d);
10
11 multiply(c, d);
```

TEMPLATES – CLASSES

```
1 template <typename T> class Calc {
2   public:
3     T multiply(T x, T y);
4     T add(T x, T y);
5 };
6
7 template <typename T> T Calc<T>::multiply(T x, T y) {
8   return x * y;
9 }
10
11 template <typename T> T Calc<T>::add(T x, T y) {
12   return x + y;
13 }
14
15 int main() {
16   double a = 4.0, b = 5.0;
17   Calc<double> c;
18   c.multiply(a, b);
19 }
```

Templates need to be defined in the same compilation unit

TEMPLATES IN OPOSSUM

- Data types
  - Segments
  - Operators
  - Statistics
- Encodings

```
1 chunk.add_segment(std::make_shared<ValueSegment<int>>());
2 chunk.add_segment(std::make_shared<ValueSegment<float>>());
3
4 std::vector<std::shared_ptr<ValueSegment>> _columns;
5
6 std::vector<std::shared_ptr<ValueSegment<int>>> _columns;
7
8 std::vector<std::shared_ptr<BaseSegment>> _columns;
```

TEMPLATES – SPECIALIZATION

```
1 template <>
2 class vector<bool> {
3   // Bitmap;
4 };
```

```
1 template <int rows, int columns>
2 class Matrix {
3   // Normal matrix implementation
4 };
5
6 template <int rows>
7 class Matrix<rows, 1> {
8   // Special matrix implementation
9 };
```

AGENDA

- Organization
- Templates
- **RAII**
- Smart Pointers
- Dictionary Encoding

RAII – RESOURCE ACQUISITION IS INITIALIZATION

RAII is a programming technique that binds the life cycle of a resource that must be acquired before use to the lifetime of an object.

[…] It also guarantees that all resources are released when the lifetime of their controlling object ends, in reverse order of acquisition.

The reference

RAII OR SBRM – MOTIVATION

```
1 void foo() {
2   ClassA* ca = new ClassA;
3
4   ca->someOperation();
5   ca->someOperationB();
6   ca->someOperationC();
7
8   delete ca;
9 }
```

```
1 void foo() {
2   ClassA ca;
3
4   ca.someOperation();
5   ca.someOperationB();
6   ca.someOperationC();
7 }
```

RAII OR SBRM – MOTIVATION

```
1  void write_to_file (const std::string & message) {
2      static std::mutex mutex;
3
4      mutex.lock()
5
6      std::ofstream file("opossum.txt");
7      if (!file.is_open())
8          throw std::runtime_error("unable to open the
9                                    opossum");
10
11     file << message << std::endl;
12
13     mutex.unlock()
14 }
```


https://en.wikipedia.org/wiki/Resource_acquisition_is_initialization

RAII OR SBRM – MOTIVATION

```
1  void write_to_file (const std::string & message) {
2      static std::mutex mutex;
3
4      std::lock_guard<std::mutex> lock(mutex);
5
6      std::ofstream file("opossum.txt");
7      if (!file.is_open())
8          throw std::runtime_error("unable to open the
9                                    opossum");
10
11     file << message << std::endl;
12 }
```


https://en.wikipedia.org/wiki/Resource_acquisition_is_initialization

RAII OR SBRM – BENEFITS

- Encapsulation
  - Resource management is centralized in class definition
- Safety
  - You cannot forget to delete / free a resource
  - Destructors are called during exception handling
- Locality
  - Constructor and destructor side by side

AGENDA

- Organization
- Templates
- RAII
- **Smart Pointers**
- Dictionary Encoding

RAW POINTERS – HAVE FUN KEEPING TRACK

```
1 SomeClass* scp = new SomeClass;
2
3 OtherClass* ocp = new OtherClass(scp);
4 WeirdClass* wcp = new WeirdClass(scp);
5
6 scp = new SomeOtherClass;
7
8 delete scp;
```

SMART POINTERS – MOTIVATION

- Motivation: Lifetime management of objects
  - *new (malloc)* also includes declaration of ownership
  - Possibility to lose objects → Resource leaks
  - Copying of p → Observation of ownership necessary

```
1 SomeClass* scp = new SomeClass;
2
3 OtherClass* ocp = new OtherClass(scp);
4 WeirdClass* wcp = new WeirdClass(scp);
5
6 scp = new SomeOtherClass;
7
8 delete scp;
```

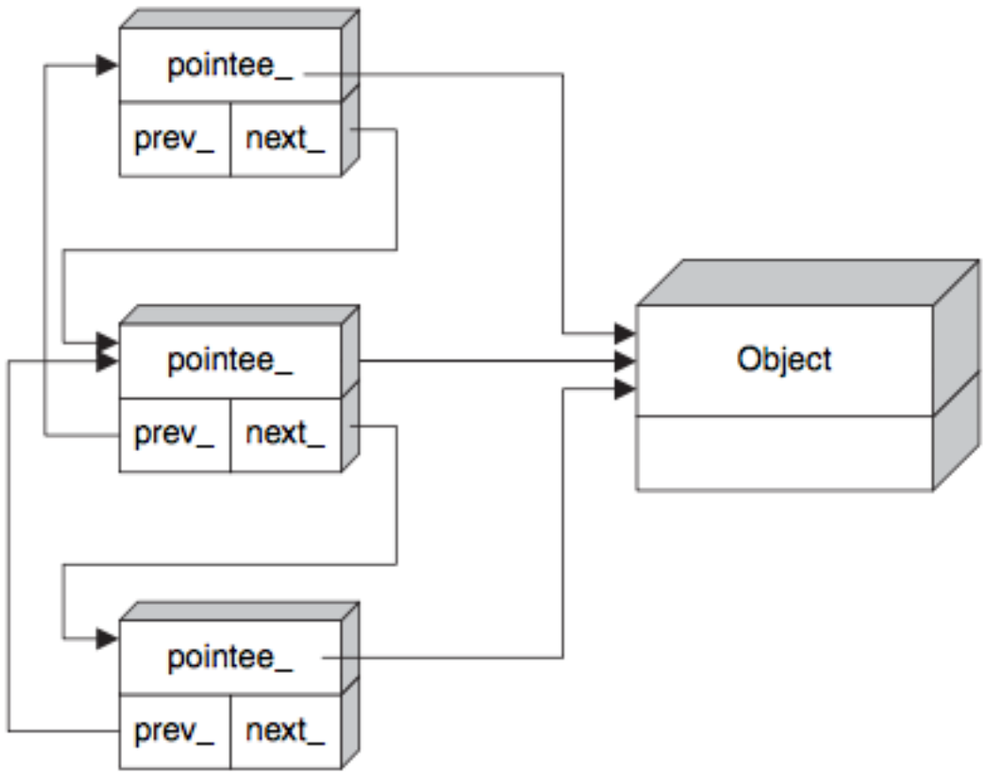
SMART POINTERS – WHAT IS A SMART POINTER?

- Exactly mimics *regular* pointers' syntax and some semantics
  - Pointer-like behavior (proxy)
  - Ownership management
    - Transfer of ownership
    - Releasing objects
- Transparent for the developer

SMART POINTERS – SHARED OWNERSHIP HANDLING

- Ideas? - Standard does not specify an implementation
  - Reference Linking

SMART POINTERS – REFERENCE LINKING



Modern C++ Design, Andrei Alexandrescu

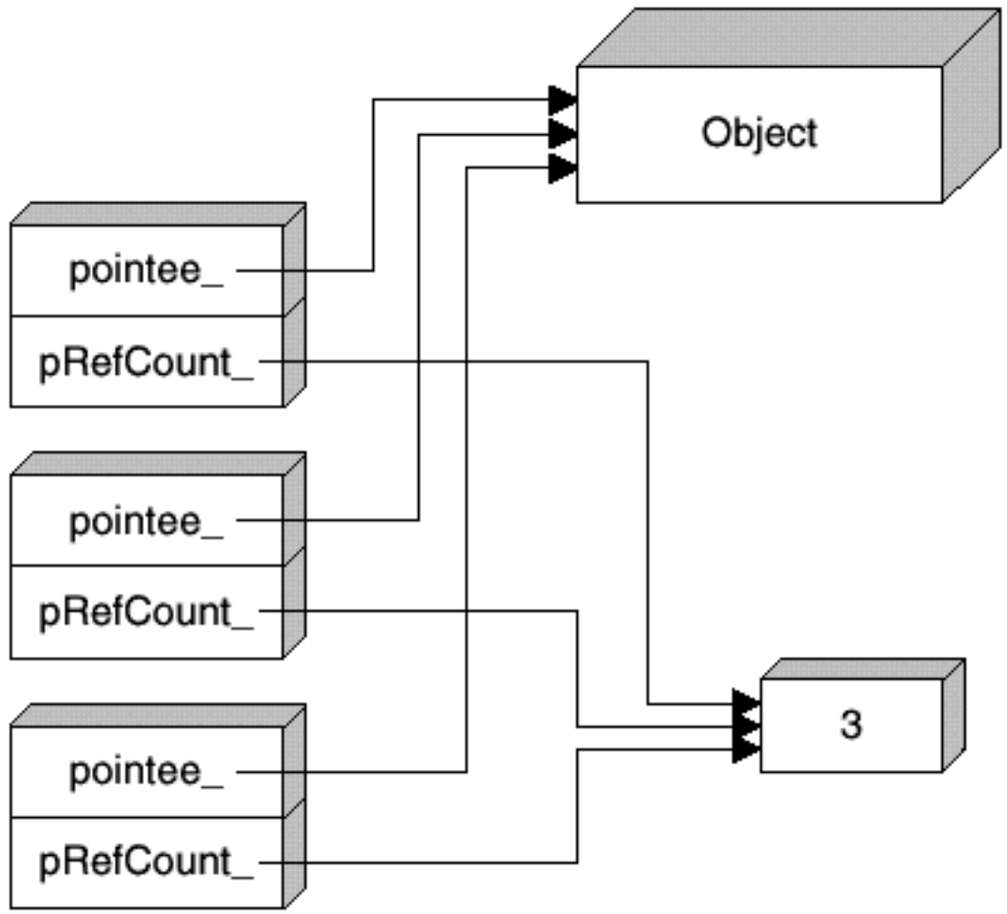
SMART POINTERS – OWNERSHIP HANDLING

- Ideas? - Standard does not specify an implementation
  - Reference Linking
  - **Reference Counting**

SMART POINTERS – REFERENCE COUNTING

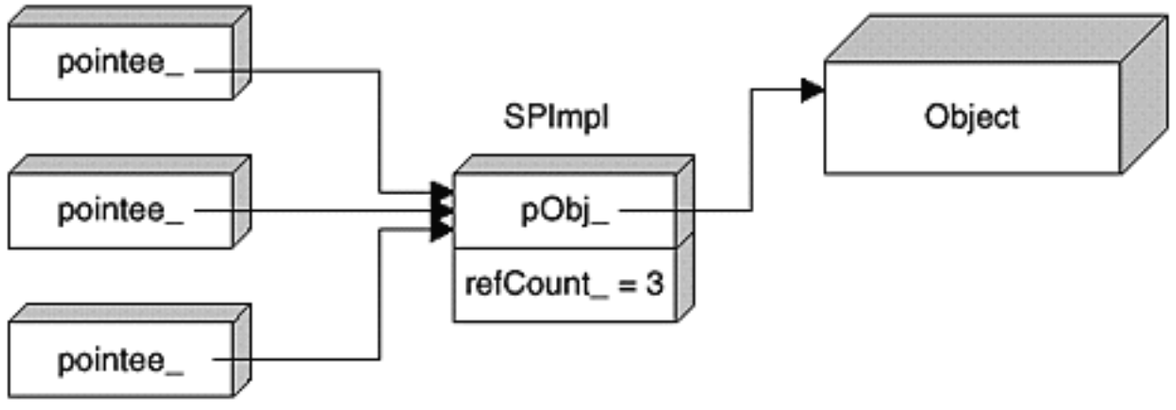
- Issue with reference counting?
  - Overhead
  - Synchronization issues
- How to implement reference counting?

SMART POINTERS – REFERENCE COUNTING

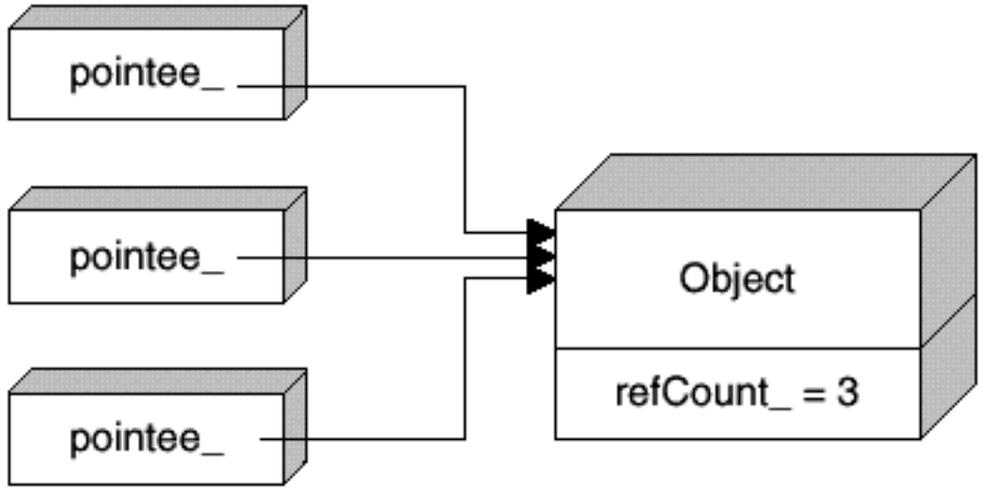


Modern C++ Design, Andrei Alexandrescu

SMART POINTERS – REFERENCE COUNTING



SMART POINTERS – REFERENCE COUNTING



Modern C++ Design, Andrei Alexandrescu

SMART POINTERS - C++

- Defined in <memory>
- std::unique_ptr<T>
  - Implicitly deleted copy constructor & copy assignment
- std::shared_ptr<T>
  - Reference counting
  - Thread safety?
- std::weak_ptr<T>
  - Does not affect ownership

SMART POINTERS – STD HELPERS

- std::make_shared - why?
  - Single memory allocation
    - std::shared_ptr<T>(new T(args...))
  - ~~Exception safety:~~
    - ~~f(std::shared_ptr<int>(new int(42)), g())~~
- std::make_unique
  - Exception safety, convenience and consistency

SMART POINTERS – CONSTNESS

```
1         auto p1 = std::make_shared<const SomeClass>();
2 const   auto p2 = std::make_shared<       SomeClass>();
3 const   auto p3 = std::make_shared<const SomeClass>();
4
5 p1->ConstMemberFunction();
6 p1->NonConstMemberFunction();
7
8 p2 = std::make_shared<SomeClass>();
9 p2->NonConstMemberFunction();
10
11 p3->NonConstMemberFunction();
12 p3->ConstMemberFunction();
13 p3 = std::make_shared<const SomeClass>();
```

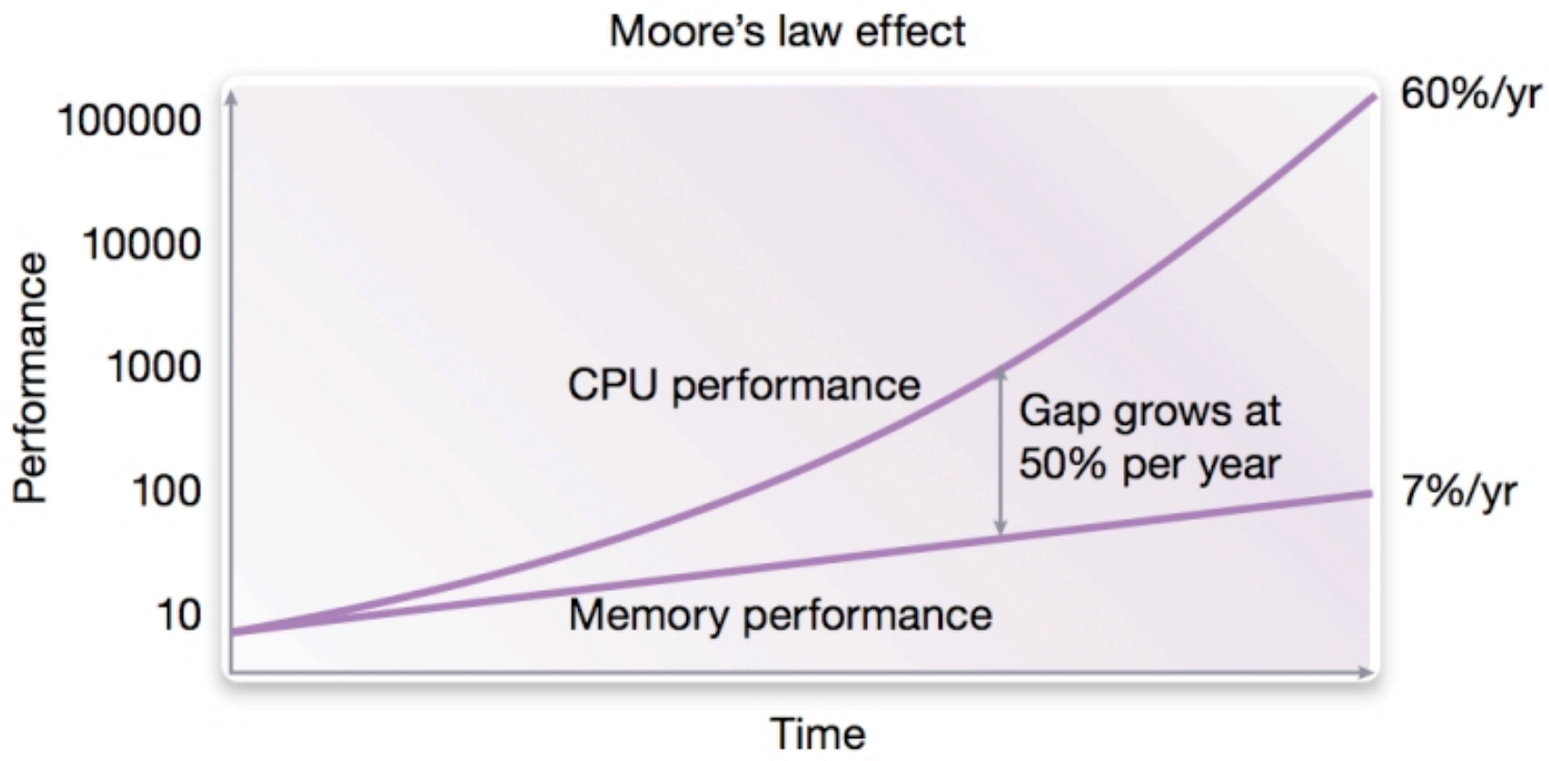
SMART POINTERS – CONSTNESS

```
1        auto p1 = std::make_shared<const SomeClass>();
2  const auto p2 = std::make_shared<      SomeClass>();
3  const auto p3 = std::make_shared<const SomeClass>();
4
5  p1->ConstMemberFunction();
6  p1->NonConstMemberFunction();
7
8  p2 = std::make_shared<SomeClass>();
9  p2->NonConstMemberFunction();
10
11 p3->NonConstMemberFunction();
12 p3->ConstMemberFunction();
13 p3 = std::make_shared<const SomeClass>();
```

AGENDA

- Organization
- Templates
- RAII
- Smart Pointers
- **Dictionary Encoding**

DICTIONARY ENCODING – MOTIVATION



- Memory access is the new bottleneck
- Decrease number of bits used for data representation

DICTIONARY ENCODING – MOTIVATION

- Dictionary encoding is an “easy-to-implement” fixed-width compression and basis for several other compression techniques
- Idea: encode every distinct value of a vector (large) with a distinct fixed-length *integer* value (small)

DICTIONARY ENCODING – EXAMPLE: SAMPLE DATA

- World population: 8 billion records

recID	fname	lname	gender	city	country	birthday
…	…	…	…	…	…	…
39	John	Smith	m	Chicago	USA	12.03.1964
40	Mary	Brown	f	London	UK	12.05.1964
41	Jane	Doe	f	Palo Alto	USA	23.04.1976
42	John	Doe	m	Palo Alto	USA	17.06.1952
43	Peter	Schmidt	m	Potsdam	GER	11.11.1975
…	…	…	…		…	…

DICTIONARY ENCODING – EXAMPLE: ENCODE A COLUMN

recID	fname
…	…
39	John
40	Mary
41	Jane
42	John
43	Peter
…	…

Dictionary for "fname"

valueID	Value
…	…
23	John
24	Mary
25	Jane
26	Peter
…	…

Attribute Vector for "fname"

position	valueID
…	…
39	23
40	24
41	25
42	23
43	26
…	…

- Dictionary stores all distinct values with an implicit valueID
- Attribute vector stores valueIDs for all entries in the column (positions are stored implicitly)

DICTIONARY ENCODING – EXAMPLE: COMPRESSION RATE

- 5 million distinct values, all have a size of 50 B
  - Bits required per valueID: ceil($\log_2(5{,}000{,}000)$) b = 23
  - Dictionary size: 5 * 10^6 * 50 B = 250 * 10^6 B = 0.250 GB
  - Attribute vector size: 8 * 10^9 * 23b = 23 * 10^9 B = 23 GB
  - Uncompressed: 8 * 10^9 * 50 B = 400 * 10^9 B = 400 GB
  - compression rate = uncompressed size / compressed size
    = 400GB / (23 GB + 0.250 GB) ≈ 17

DICTIONARY ENCODING – QUERY DATA

recID	fname
…	…
39	John
40	Mary
41	Jane
42	John
43	Peter
…	…

Dictionary for “fname”

valueID	Value
…	…
23	John
24	Mary
25	Jane
26	Peter
…	…

Attribute Vector for “fname”

position	valueID
…	…
39	23
40	24
41	25
42	23
43	26
…	…

- Retrieve all persons (recIDs) with name “Mary”
  - 1. Search valueID for “Mary” (requested value)
  - 2. Scan Attribute vector for “24” (found valueID)

DICTIONARY ENCODING – SORTED DICTIONARY: ADVANTAGES

- Dictionary entries are sorted by their value
  - Dictionary search complexity: O(log(n)) instead O(n)
  - Speed up range queries
  - Dictionary entries can be further compressed

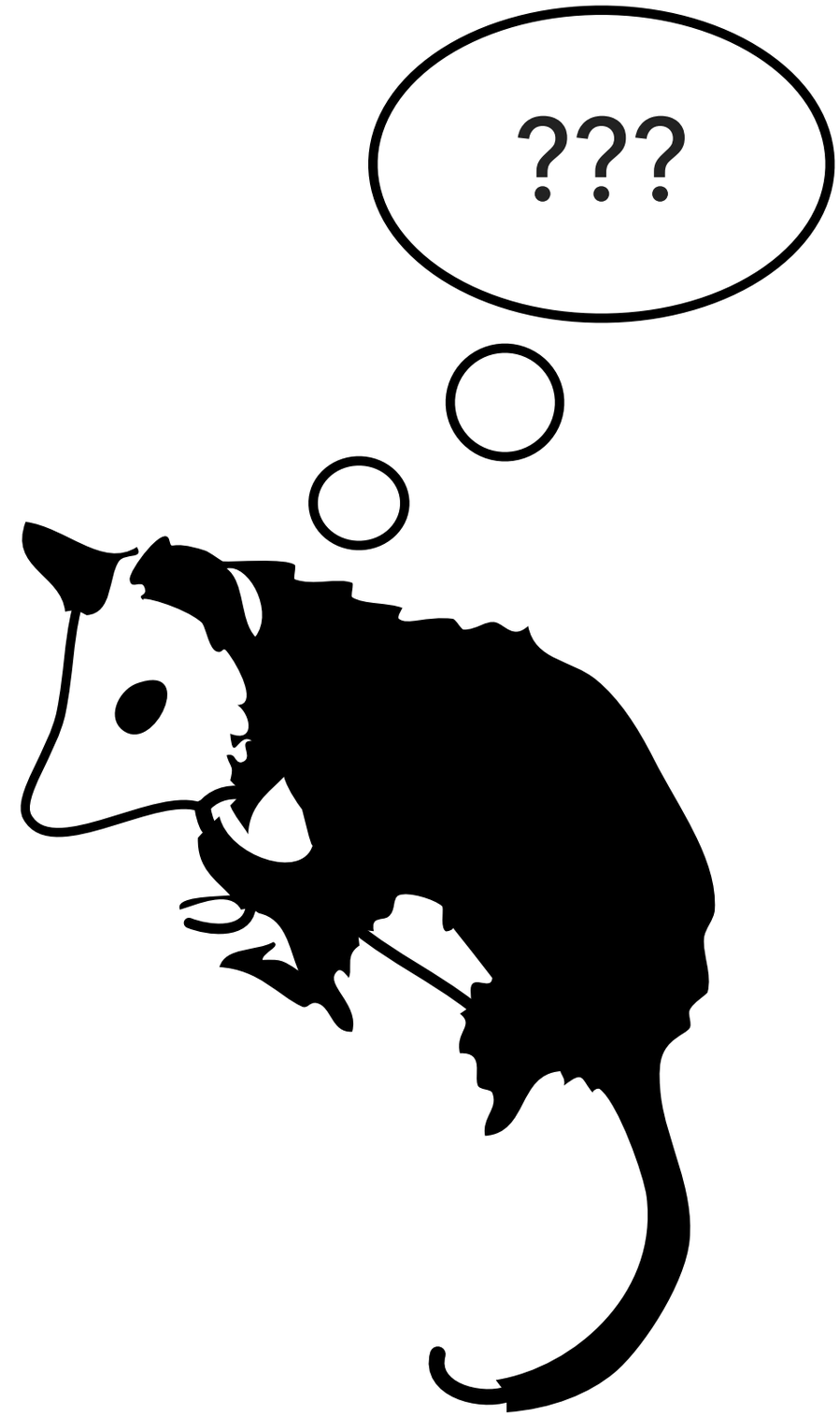
DICTIONARY ENCODING – DISADVANTAGES

- Dictionary entries are sorted by their value
  - Resorting for every new value that does not belong to the end of the sorted sequence (relatively cheap)
    - Updating the attribute vector (costly)
- Dictionary adds additional indirection for materialization
- Overhead for large number of data modifying operations

DICTIONARY ENCODING – IN OPOSSUM

- Dictionary encoding is applied to immutable chunks
- Sorted dictionaries are used
- valueIDs are of type uint8_t, uint16_t, uint32_t

QUESTIONS



Hasso Plattner Institut

IT Systems Engineering | Universität Potsdam

Build your own Database

Week 2 - Follow-Up

Reminder

```
class SomeClass {
  public:
    SomeClass() { std::cout << "SomeClass Constructor" << std::endl; }
    ~SomeClass() { std::cout << "SomeClass Destructor" << std::endl; }
};

void wait_5(const std::shared_ptr<SomeClass>& s_ptr) {
    std::this_thread::sleep_for(std::chrono::seconds(5));
}

int main() {
    auto p = std::make_shared<SomeClass>();

    std::thread t(wait_5, p);
    p.reset();
    t.join();

    std::cout << "The End!" << std::endl;
}
```

Expectation

```
class SomeClass {
  public:
    SomeClass() { std::cout << "SomeClass Constructor" << std::endl; }
    ~SomeClass() { std::cout << "SomeClass Destructor" << std::endl; }
};

void wait_5(const std::shared_ptr<SomeClass>& s_ptr) {
    std::this_thread::sleep_for(std::chrono::seconds(5));
}

int main() {
    auto p = std::make_shared<SomeClass>();

    std::thread t(wait_5, p);
    p.reset();
    t.join();

    std::cout << "The End!" << std::endl;
}
```

We reset the pointer here, so we would expect the destructor to be called right away, not after 5 seconds

Expectation

```
class SomeClass {
  public:
    SomeClass() { std::cout << "SomeClass Constructor" << std::endl; }
    ~SomeClass() { std::cout << "SomeClass Destructor" << std::endl; }
};

void wait_5(const std::shared_ptr<SomeClass>& s_ptr) {
    std::this_thread::sleep_for(std::chrono::seconds(5));
}

int main() {
    auto p = std::make_shared<SomeClass>();

    std::thread t(wait_5, p);
    p.reset();
    t.join();

    std::cout << "The End!" << std::endl;
}
```

We reset the pointer here, so we would expect the destructor to be called right away, not after 5 seconds

Understanding what happens

```
int main() {
    auto p = std::make_shared<SomeClass>();
    std::cout << "(1) " << p.use_count() << std::endl;
    std::thread t(wait_5, p);
    std::cout << "(2) " << p.use_count() << std::endl;
    p.reset();
    std::cout << "(3) " << p.use_count() << std::endl;
    t.join();
    std::cout << "(4) " << p.use_count() << std::endl;


    std::cout << "The End!" << 
}
```

```
[:~/tmp/sourceCode] $ ./Shared
SomeClass Constructor
(1) 1
(2) 2
(3) 0
SomeClass Destructor
(4) 0
The End!
```

Understanding what happens

```
int main() {
    auto p = std::make_shared<SomeClass>();
    std::cout << "(1) " << p.use_count() << std::endl;
    // std::thread t(wait_5, p);
    wait_5(p);
    std::cout << "(2) " << p.use_count() << std::endl;
    p.reset();
    std::cout << "(3) " << p.use_count() << std::endl;
    // t.join();
    // std::cout << "(4) " << p.use_count() << std::endl;


    std::cout << "The End!" << 
}
```

```
SomeClass Constructor
(1) 1
(2) 1
SomeClass Destructor
(3) 0
The End!
```

Why does using std::thread result in a copy?

https://en.cppreference.com/w/cpp/thread/thread/

(We like cppreference.com more than cplusplus.com)

Why does using std::thread result in a copy?

std::thread

Defined in header <thread>

```
class thread;        (since C++11)
```

The class thread represents a single thread of execution. Threads allow multiple functions to execute concurrently.

Threads begin execution immediately upon construction of the associated thread object (pending any OS scheduling delays), starting at the top-level function provided as a constructor argument. The return value of the top-level function is ignored and if it terminates by throwing an exception, std::terminate is called. The top-level function may communicate its return value or an exception to the caller via std::promise or by modifying shared variables (which may require synchronization, see std::mutex and std::atomic)

std::thread objects may also be in the state that does not represent any thread (after default construction, move from, detach, or join), and a thread of execution may be not associated with any thread objects (after detach).

No two std::thread objects may represent the same thread of execution; std::thread is not *CopyConstructible* or *CopyAssignable*, although it is *MoveConstructible* and *MoveAssignable*.

Member types

Member type	Definition
native_handle_type	*implementation-defined*

Member classes

id	represents the *id* of a thread (public member class)

Member functions

(constructor)	constructs new thread object (public member function)
(destructor)	destructs the thread object, underlying thread must be joined or detached (public member function)
operator=	moves the thread object (public member function)
Observers	
joinable	checks whether the thread is joinable, i.e. potentially running in parallel context

Why does using std::thread result in a copy?

std::thread::thread

```
thread() noexcept;                                              (1)  (since C++11)
thread( thread&& other ) noexcept;                              (2)  (since C++11)
template< class Function, class... Args >
explicit thread( Function&& f, Args&&... args );                (3)  (since C++11)
thread(const thread&) = delete;                                 (4)  (since C++11)
```

Why does using std::thread result in a copy?

3) Creates new `std::thread` object and associates it with a thread of execution. The new thread of execution starts executing

```
std::invoke(decay_copy(std::forward<Function>(f)), decay_copy(std::forward<Args>(args))...)
```

where decay_copy is defined as

```
template <class T>
std::decay_t<T> decay_copy(T&& v) { return std::forward<T>(v); }
```

Except that the calls to decay_copy are evaluated in the context of the caller, so that any exceptions thrown during evaluation and copying/moving of the arguments are thrown in the current thread, without starting the new thread.

The completion of the invocation of the constructor *synchronizes-with* (as defined in `std::memory_order`) the beginning of the invocation of the copy of *f* on the new thread of execution.

This constructor does not participate in overload resolution if `std::decay_t<Function>` is the same type as `std::thread`. (since C++14)

Why does using std::thread result in a copy?

Notes

The arguments to the thread function are moved or copied by value. If a reference argument needs to be passed to the thread function, it has to be wrapped (e.g. with `std::ref` or `std::cref`).

Any return value from the function is ignored. If the function throws an exception, `std::terminate` is called. In order to pass return values or exceptions back to the calling thread, `std::promise` or `std::async` may be used.

highlight

- `highlight` can be used to format code
- `pbpaste | cat | /usr/local/bin/highlight -O rtf --font Consolas -W -J 70 -j 3 --src-lang c++ --style peaksea | pbcopy`