Hasso Plattner Institut
IT Systems Engineering | Universität Potsdam

WOCHE 2

DYOD

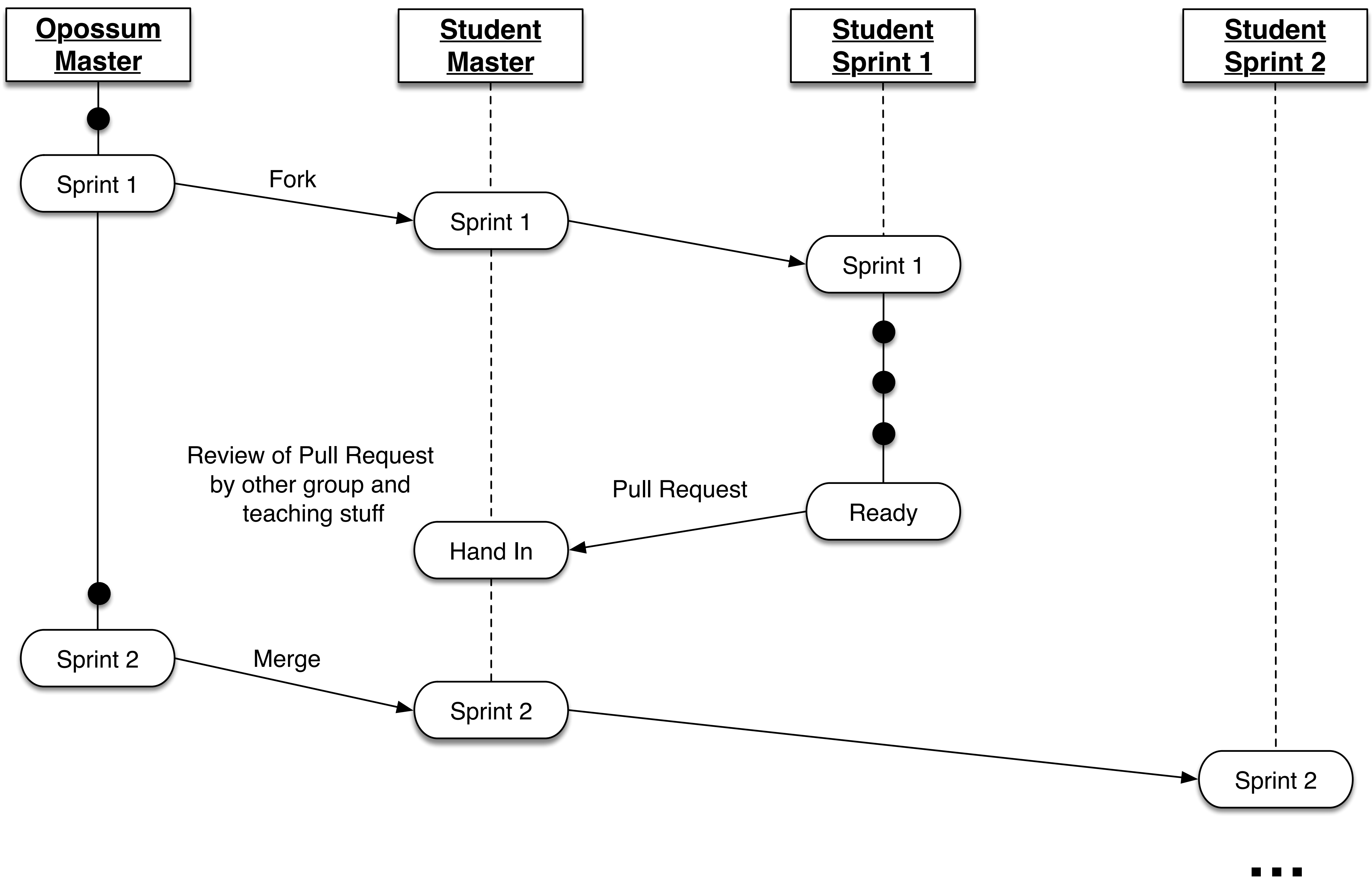
AGENDA

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

ORGANIZATION

- Next week: Dies 20 Jahre HPI → No class
- Did you find a group yet?
- If you have not joined us at Piazza:
  - piazza.com/hpi.uni-potsdam.de/fall2019/dyod
- Any problems during setup/coding?

ORGANIZATION



C++ CORE GUIDELINES - MAILINGLIST

http://tiny.cc/cppCoreGuidelines

Markus Dreseler
[cppcoreguidelines] ES.20: Always initialize an object
To: cppcoreguidelines@lists.myhpi.de

Inbox - Exchange 18. October 2019 at [illegible]
MD

ES.20: Always initialize an object

Reason

Avoid used-before-set errors and their associated undefined behavior. Avoid problems with comprehension of complex initialization. Simplify refactoring.

Example

```
void use(int arg)
{
    int i;   // bad: uninitialized variable
    // ...
    i = 7;   // initialize i
}
```

No, `i = 7` does not initialize `i`; it assigns to it. Also, `i` can be read in the `...` part. Better:

```
void use(int arg)   // OK
{
    int i = 7;   // OK: initialized
    string s;    // OK: default initialized
    // ...
}
```

Note

The *always initialize* rule is deliberately stronger than the *an object must be set before used* language rule. The latter, more relaxed rule, catches the technical bugs, but:

- It leads to less readable code
- It encourages people to declare names in greater than necessary scopes
- It leads to harder to read code
- It leads to logic bugs by encouraging complex code
- It hampers refactoring

The *always initialize* rule is a style rule aimed to improve maintainability as well as a rule protecting against used-before-set errors.

Example

Here is an example that is often considered to demonstrate the need for a more relaxed rule for initialization

```
widget i;    // "widget" a type that's expensive to initialize, possibly a large POD
widget j;
```

AGENDA

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

TEMPLATES - FUNCTIONS

```
template <typename T> T multiply(T x, T y) {
  return x * y;
}

double a = 4.0, b = 5.0;
multiply<double>(a, b);

int c = 7, d = 8;
multiply<int>(c, d);
```

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

TEMPLATES - FUNCTIONS

```
template <typename T> T multiply(T x, T y) {
  return x * y;
}

double a = 4.0, b = 5.0;
multiply<double>(a, b);

int c = 7, d = 8;
multiply<int>(c, d);

multiply(c, d);
```

TEMPLATES - CLASSES

```cpp
template <typename T> class Calc {
  public:
    T multiply(T x, T y);
    T add(T x, T y);
};

template <typename T> T Calc<T>::multiply(T x, T y) {
  return x * y;
}

template <typename T> T Calc<T>::add(T x, T y) {
  return x + y;
}

int main() {
  double a = 4.0, b = 5.0;
  Calc<double> c;
  c.multiply(a, b);
}
```

Usually, templates need to be defined in the same compilation unit

TEMPLATES IN OPOSSUM

- Example from sprint 1

```
chunk.add_segment(std::make_shared<ValueSegment<int>>());
chunk.add_segment(std::make_shared<ValueSegment<float>>());

std::vector<std::shared_ptr<ValueSegment>> _columns;

std::vector<std::shared_ptr<ValueSegment<int>>> _columns;

std::vector<std::shared_ptr<BaseSegment>> _columns;
```

- We also use templates to make operators, statistics independent of the data and encoding type

TEMPLATES – SPECIALIZATION

```
template <>
class vector<bool> {
  // Bitmap;
};
```

```
template <int rows, int columns>
class Matrix {
  // Normal matrix implementation
};

template <int rows>
class Matrix<rows, 1> {
  // Special matrix implementation
};
```

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

```cpp
void foo() {
  ClassA* ca = new ClassA;

  ca->someOperation();
  ca->someOperationB();
  ca->someOperationC();

  delete ca;
}
```

```cpp
void foo() {
  ClassA ca;

  ca.someOperation();
  ca.someOperationB();
  ca.someOperationC();
}
```

RAII OR SBRM – MOTIVATION

```
void write_to_file (const std::string& message) {
    static std::mutex mutex;

    mutex.lock();

    std::ofstream file("opossum.txt");
    if (!file.is_open())
        throw std::runtime_error("unable to open the
                                  opossum");

    file << message << std::endl;

    mutex.unlock();
}
```


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

RAII OR SBRM – MOTIVATION

```
void write_to_file (const std::string & message) {
    static std::mutex mutex;

    std::lock_guard<std::mutex> lock(mutex);

    std::ofstream file("opossum.txt");
    if (!file.is_open())
        throw std::runtime_error("unable to open the
                                  opossum");

    file << message << std::endl;
}
```


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

```
SomeClass* scp = new SomeClass;

OtherClass* ocp = new OtherClass(scp);
WeirdClass* wcp = new WeirdClass(scp);

scp = new SomeOtherClass;

delete scp;
```

Hasso Plattner Institut
IT Systems Engineering | Universität Potsdam

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

```
SomeClass* scp = new SomeClass;

OtherClass* ocp = new OtherClass(scp);
WeirdClass* wcp = new WeirdClass(scp);

scp = new SomeOtherClass;

delete scp;
```

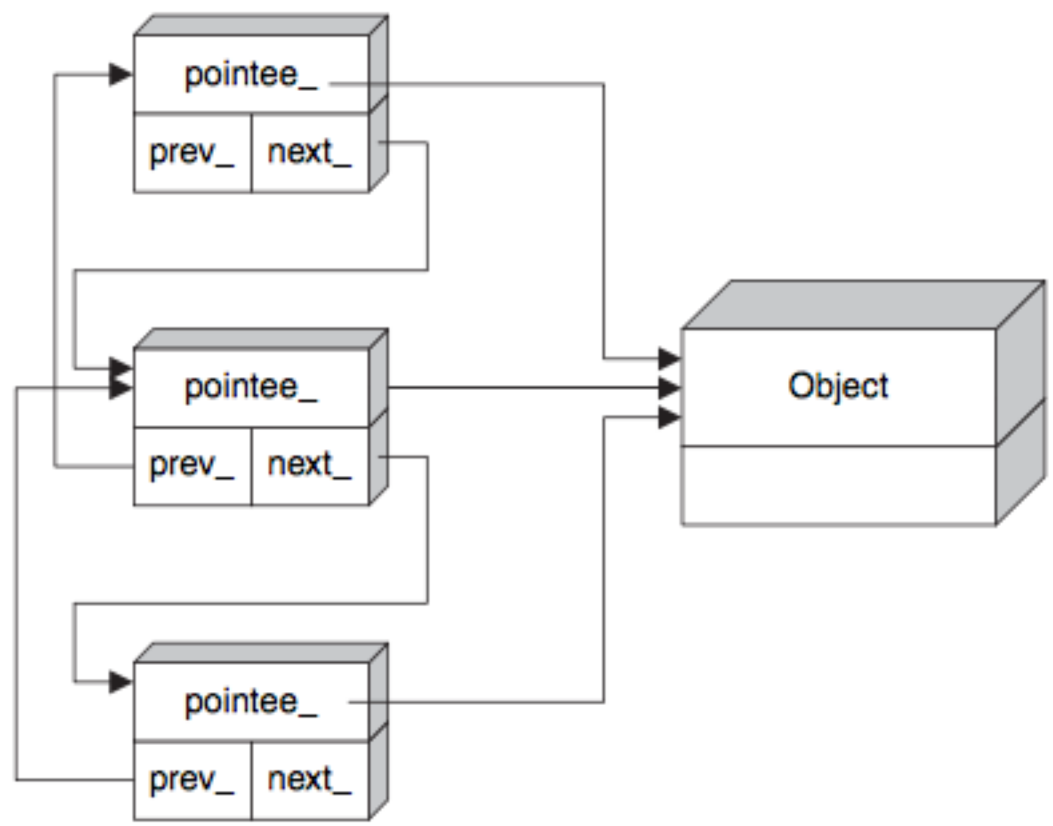
SMART POINTERS – WHAT IS A SMART POINTER?

- Exactly mimics *regular* pointers' syntax and some semantics
  - Pointer-like behavior (proxy)
  - Transparent for the developer
  - Ownership management
    - Ownership type: shared or unique
    - Releasing objects

SMART POINTERS - SHARED OWNERSHIP HANDLING

- Standard does not specify an implementation
  - Reference Linking

SMART POINTERS – REFERENCE LINKING



Modern C++ Design, Andrei Alexandrescu

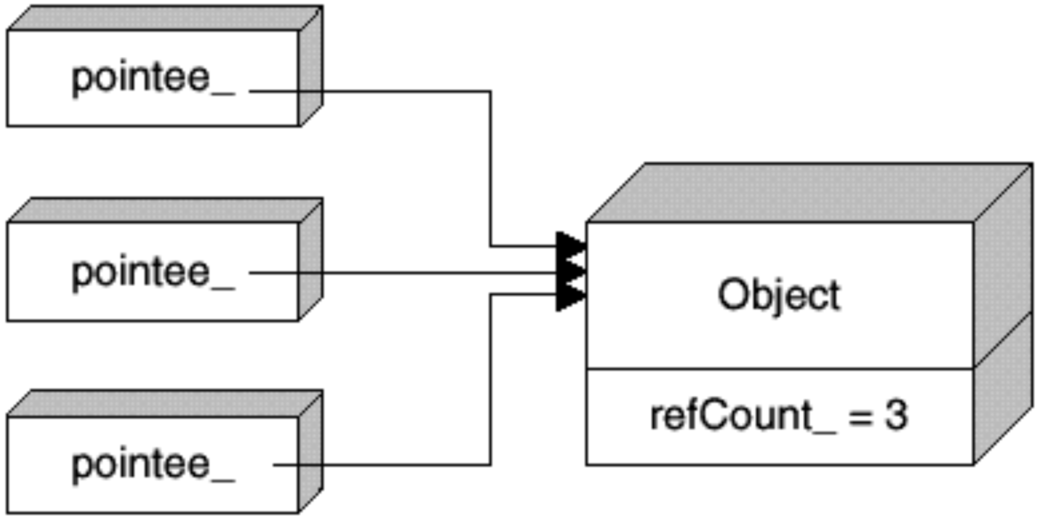
SMART POINTERS – OWNERSHIP HANDLING

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

SMART POINTERS - REFERENCE COUNTING

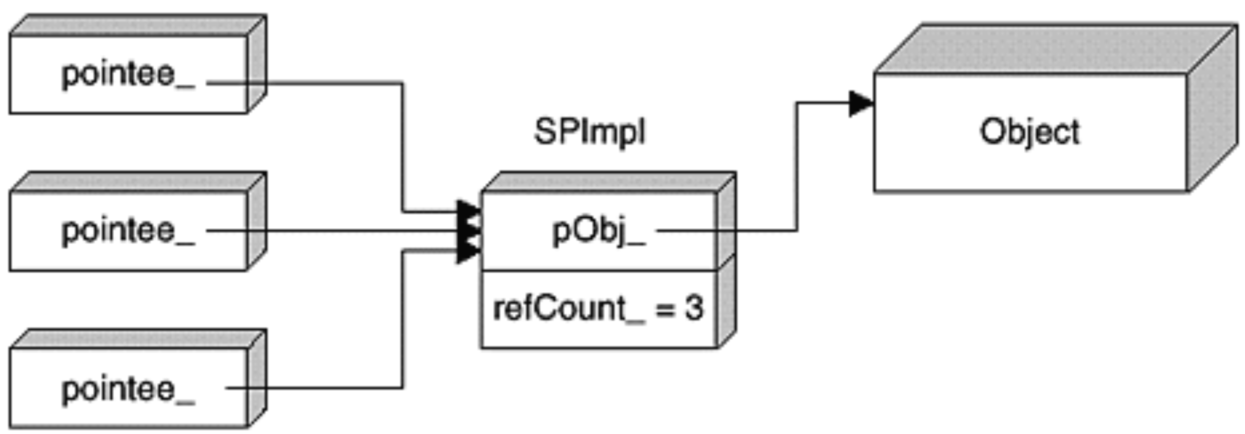
- Issue with reference counting?
  - Overhead
  - Synchronization issues
- How to implement reference counting?

SMART POINTERS – REFERENCE COUNTING – OPTION A



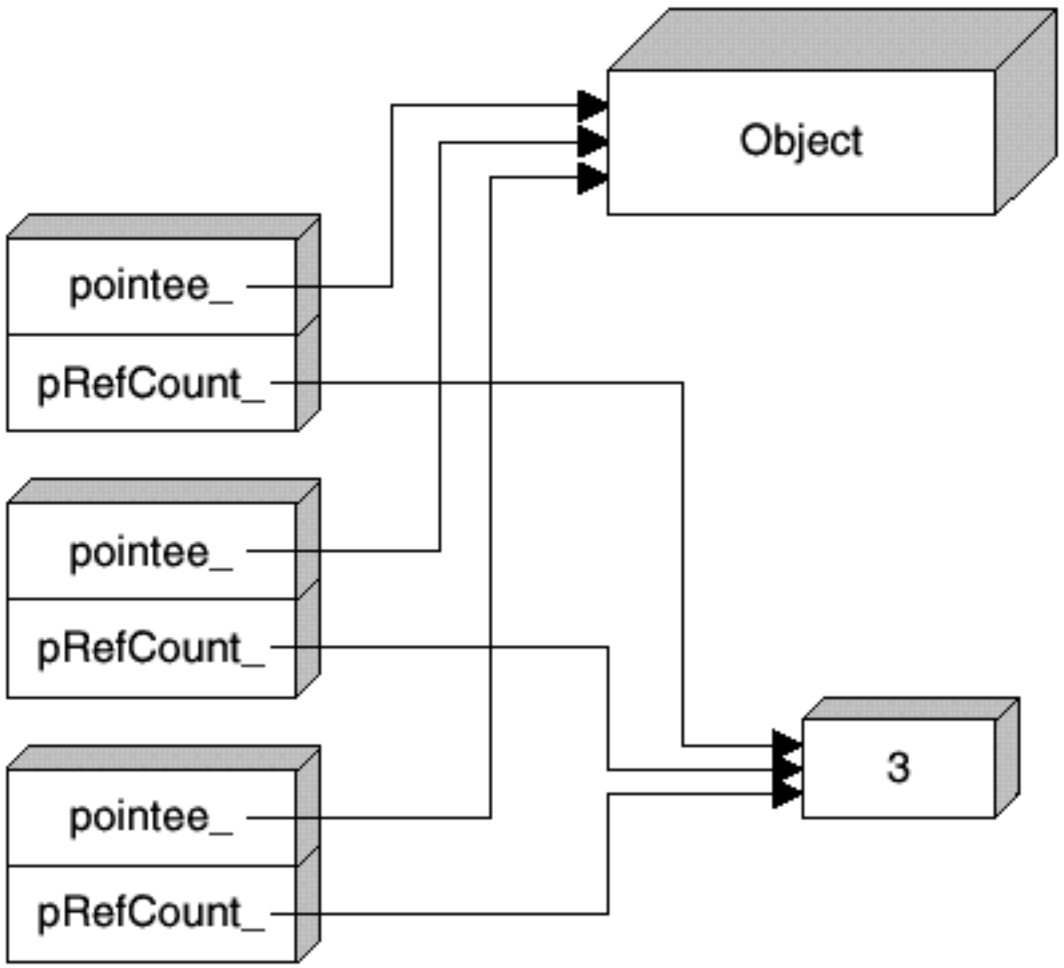
Modern C++ Design, Andrei Alexandrescu

SMART POINTERS – REFERENCE COUNTING – OPTION B



Modern C++ Design, Andrei Alexandrescu

SMART POINTERS - REFERENCE COUNTING - OPTION C



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?
  - Overhead
- 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...))
- std::make_unique
  - Convenience and consistency

SMART POINTERS – CONSTNESS

```
      auto p1 = std::make_shared<const SomeClass>();
const auto p2 = std::make_shared<      SomeClass>();
const auto p3 = std::make_shared<const SomeClass>();

p1->ConstMemberFunction();
p1->NonConstMemberFunction();

p2 = std::make_shared<SomeClass>();
p2->NonConstMemberFunction();

p3->NonConstMemberFunction();
p3->ConstMemberFunction();
p3 = std::make_shared<const SomeClass>();
```

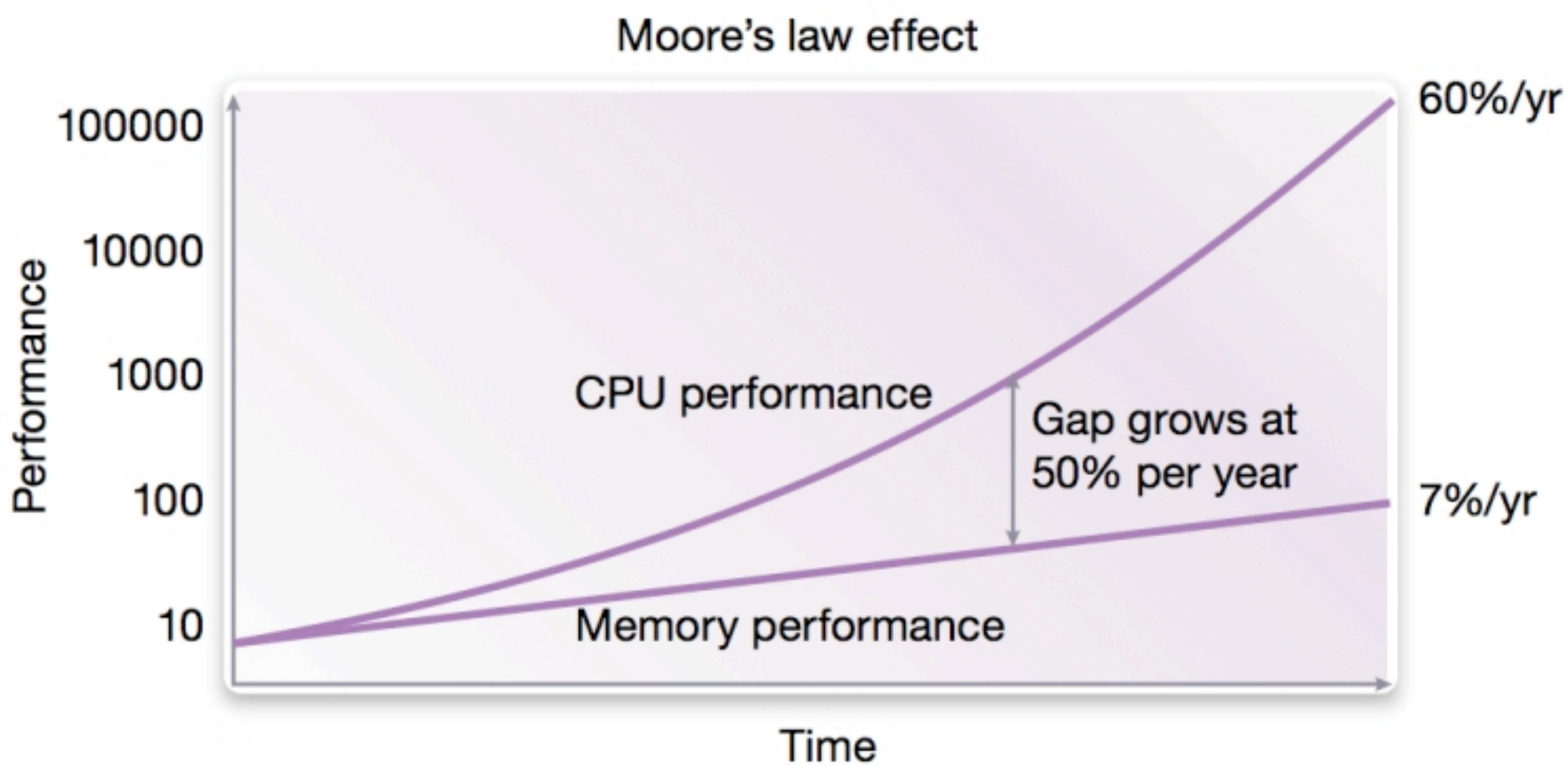
SMART POINTERS – CONSTNESS

```
      auto p1 = std::make_shared<const SomeClass>();
const auto p2 = std::make_shared<      SomeClass>();
const auto p3 = std::make_shared<const SomeClass>();

p1->ConstMemberFunction();
p1->NonConstMemberFunction();

p2 = std::make_shared<SomeClass>();
p2->NonConstMemberFunction();

p3->NonConstMemberFunction();
p3->ConstMemberFunction();
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

Hasso Plattner Institut
IT Systems Engineering | Universität Potsdam

DICTIONARY ENCODING – DISADVANTAGES

- Dictionary entries are sorted by their value
  - Resorting for every new value that cannot be appended 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

Hasso Plattner Institut
IT Systems Engineering | Universität Potsdam

QUESTIONS