Hasso Plattner Institut
IT Systems Engineering | Universität Potsdam

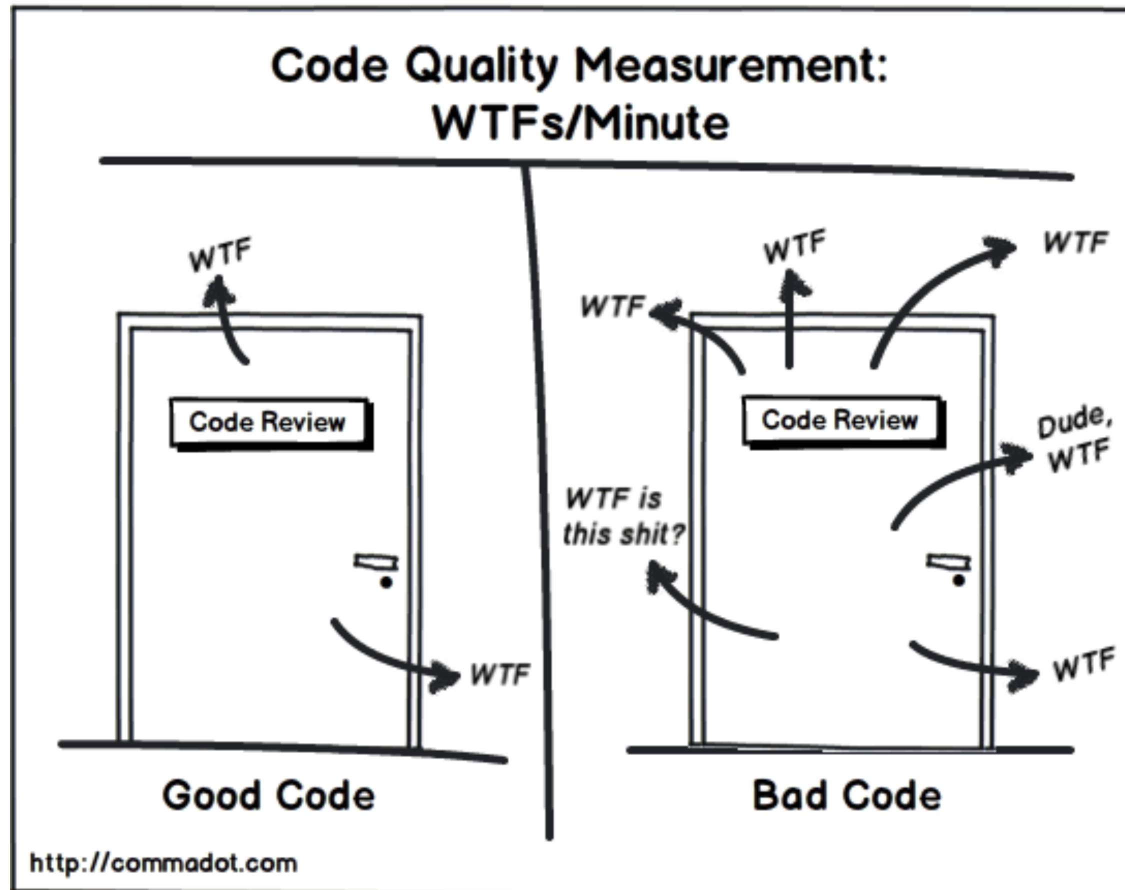
Build your own Database

Week 4

Outlook

1. Review Sprint 1
2. Move Constructors / std::move
3. How to keep track / clang-tidy
4. Questions for Sprint 2

On Code Reviews



On Code Reviews

You are not your code. When someone gives you respectful feedback on it, detachment is the appropriate state of mind. Embrace "Request Changes" and use it to tighten your feedback loop and increase your velocity.

But: Not every review comment is necessarily correct.

https://medium.com/@schrockn/on-code-reviews-b1c7c94d868c

Honour thy Hyrise Style Guide

```
this->segments.push_back(segment);

this->column_count()
...
```

```
for (int i = 0; i < this->column_count(); i++) {
```

```
for (int index = 0; index < this->column_count(); index++) {
```

```
protected:
  std::vector<std::shared_ptr<BaseSegment>> segments;
```

Honour thy Hyrise Style Guide

Many rules in the style guide are debatable. There is nothing inherently wrong with prefixing all class accesses with this->.

What is important is the consistency across the code base.

HPI Hasso Plattner Institut

Error Handling

- Some groups check for most edge cases, others do not
- We have no standard rules for error handling so far
- Fail loud and early
- Always do checks when they are (almost) free, especially when they are in the control path (not the per-row data path)
- Use `DebugAssert` for expensive checks (some groups already did)
- User-facing assertions should not be DebugAsserts

Error Handling

```
std::vector<Chunk> _chunks;
Chunk &Table::get_chunk(ChunkID chunk_id) {
```

A
```
return _chunks.at(chunk_id);
```

B
```
return _chunks[chunk_id];
```

C
```
if (chunk_id >= _chunks.size())
  throw std::runtime_error(...)
return _chunks.at(chunk_id);
```

D
```
DebugAssert(chunk_id < _chunks.size(), "...");
return _chunks[chunk_id];
```

```
}
```

HPI Hasso Plattner Institut

Error Handling

- Most STL-Containers can help us a lot at almost zero cost

```
std::map<std::string, Table> _tables;
```

A

```
_tables[name] = table;
```

B

```
Assert(_tables.find(name) == _tables.end(), name +
  " already exists");
_table[name] = table;
```

C

```
_tables.insert({name, table});
```

D

```
auto r = _tables.insert({name, table});
if(!r.second) throw std::runtime_error("...");
```

E

```
auto r = _tables.insert({name, table});
Assert(r.second, "...");
```

Error Handling

- What can we improve about this code?

```
std::map<std::string, Table> _tables;

if (_tables.find(name) != _tables.end()) {
  _tables.erase(name);
}
```

```
size_type erase( const key_type& key );                                  (3)
```

3) Removes the element (if one exists) with the key equivalent to key.

```
std::map<std::string, Table> _tables;

_tables.erase(name);
```

HPI Hasso Plattner Institut

Error Handling

- How can we further improve this?

```
std::map<std::string, Table> _tables;

_tables.erase(name);
```

Return value

3) Number of elements removed.

```
std::map<std::string, Table> _tables;

const auto num_deleted = _tables.erase(name);
Assert(num_deleted == 1, "Error deleting table...");
```

Error Handling

- Careful – what‛s the problem with this?

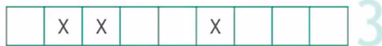
```
std::map<std::string, Table> _tables;

DebugAssert(_tables.erase(name), "...");
```

Know the STL!

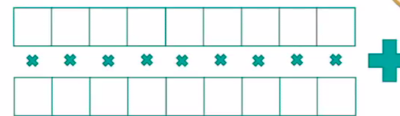
conference.accu.org

NUMERIC ALGORITHMS

count

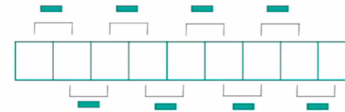


inner_product



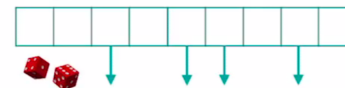
accumulate/(transform_)reduce

adjacent_difference



partial_sum
(transform_)inclusive_scan
(transform_)exclusive_scan

sample



@JoBoccara

76

accu 2018
April 11 - 14

@ACCUconf

25:42 / 53:30

105 STL Algorithms in Less Than an Hour - Jonathan Boccara [ACCU 2018]

2.734 Aufrufe

118 2 TEILEN

https://www.youtube.com/watch?v=bXkWuUe9V2I

Talking about data structures…

```
std::vector<std::string> StorageManager::table_names() const {
  std::vector<std::string> names(_tables.size());
  for (auto& map_entry : _tables) {
    names.push_back(map_entry.first);
  }
  return names;
}
```

```
std::vector<std::string> StorageManager::table_names() const {
  std::vector<std::string> names{};
  names.reserve(_tables.size());
  for (auto& map_entry : _tables) {
    names.push_back(map_entry.first);
  }
  return names;
}
```

Talking about data structures…

```
std::unordered_map<size_t, std::shared_ptr<Table>> tables;
tables.emplace(std::hash{}(table_name), table);
```

The hash is NOT the key of a hash map.
It is used to identify POTENTIAL matches – the hash map still has to check these for equality

Also, when testing something that uses a hash map, try a degenerate hash function.

Initializer Lists

- What is the problem with this code?

```
class Table {
  Table(const size_t chunk_size) {
    _chunk_size = chunk_size;
    _chunks.push_back(std::make_shared<Chunk>());
  }

protected:
  size_t _chunk_size;
  [...]
};
```

Initializer Lists

```
class Table {
  Table(const size_t chunk_size) {
    _chunk_size = chunk_size;
    _chunks.push_back(std::make_shared<Chunk>());
  }

protected:
  const size_t _chunk_size;
  [...]
};
```

```
[:~/Desktop/tmp] 3s $ g++-6 test.cpp -std=c++17
test.cpp: In constructor 'Table::Table(size_t)':
test.cpp:5:3: error: uninitialized const member in 'const size_t {aka const long unsigned int}' [-fpermissive]
   Table(const size_t chunk_size) {
   ^~~~~
test.cpp:10:16: note: 'const size_t Table::_chunk_size' should be initialized
   const size_t _chunk_size;
                ^~~~~~~~~~~
test.cpp:6:19: error: assignment of read-only member 'Table::_chunk_size'
     _chunk_size = chunk_size;
                   ^~~~~~~~~~
```

Initializer Lists

- It is better to initialize members in the constructor‘s initialization list

```
class Table {
  Table(const size_t chunk_size) : _chunk_size(chunk_size) {
    _chunks.push_back(std::make_shared<Chunk>());
  }

protected:
  const size_t _chunk_size;
  [...]
};
```

Miscellaneous

```
uint64_t Table::row_count() const {
  return static_cast<uint64_t>(
    (_chunks.size() - 1) * _chunk_size + _current_chunk->size());
}
```

With the information you had so far, this was a valid approach.
However, tables may have chunks of different sizes, e.g., when they are the temporary result of an operator.

Miscellaneous

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

// …
_columns[current_index].get()->append(values[current_index]);
```

Treat shared_ptrs just like a regular pointer

Miscellaneous

- Not a problem now, but might become in the future…

```
void Table::append(std::initializer_list<AllTypeVariant> values) {
  if (_chunk_size != 0 && _chunks.back().size() >= _chunk_size) {
    _chunks.push_back(Chunk());
    for (auto type_it = _column_types.begin(); type_it !=
         _column_types.end(); type_it++) {
      auto segment = make_shared_by_data_type<BaseSegment,
                     ValueSegment>(*type_it);
      _chunks.back().add_segment(segment);
    }
  }
  _chunks.back().append(values);
}
```

Hasso Plattner Institut

Miscellaneous

- Not a problem now, but might become in the future…

```
void Table::append(std::initializer_list<AllTypeVariant> values) {
  if (_chunk_size != 0 && _chunks.back().size() >= _chunk_size) {
    Chunk new_chunk;
    for (auto type_it = _column_types.begin(); type_it !=
          column_types.end(); type_it++) {
      auto segment = make_shared_by_data_type<BaseSegment,
                     ValueSegment>(*type_it);
      new_chunk.add_segment(segment);
    }
    _chunks.emplace_back(std::move(new_chunk));
  }
  _chunks.back().append(values);
}
```

HPI Hasso Plattner Institut

Miscellaneous

- Make the code shorter

```
void Table::append(const std::vector<AllTypeVariant>& values) {
  if (_chunk_size != 0 && _chunks.back().size() >= _chunk_size) {
    Chunk new_chunk;
    for (const auto& type : _column_types) {
      auto segment = make_shared_by_data_type<BaseSegment,
        ValueSegment>(type);
      new_chunk.add_segment(segment);
    }
    _chunks.emplace_back(std::move(new_chunk));
  }
  _chunks.back().append(values);
}
```

Hasso Plattner Institut

Miscellaneous

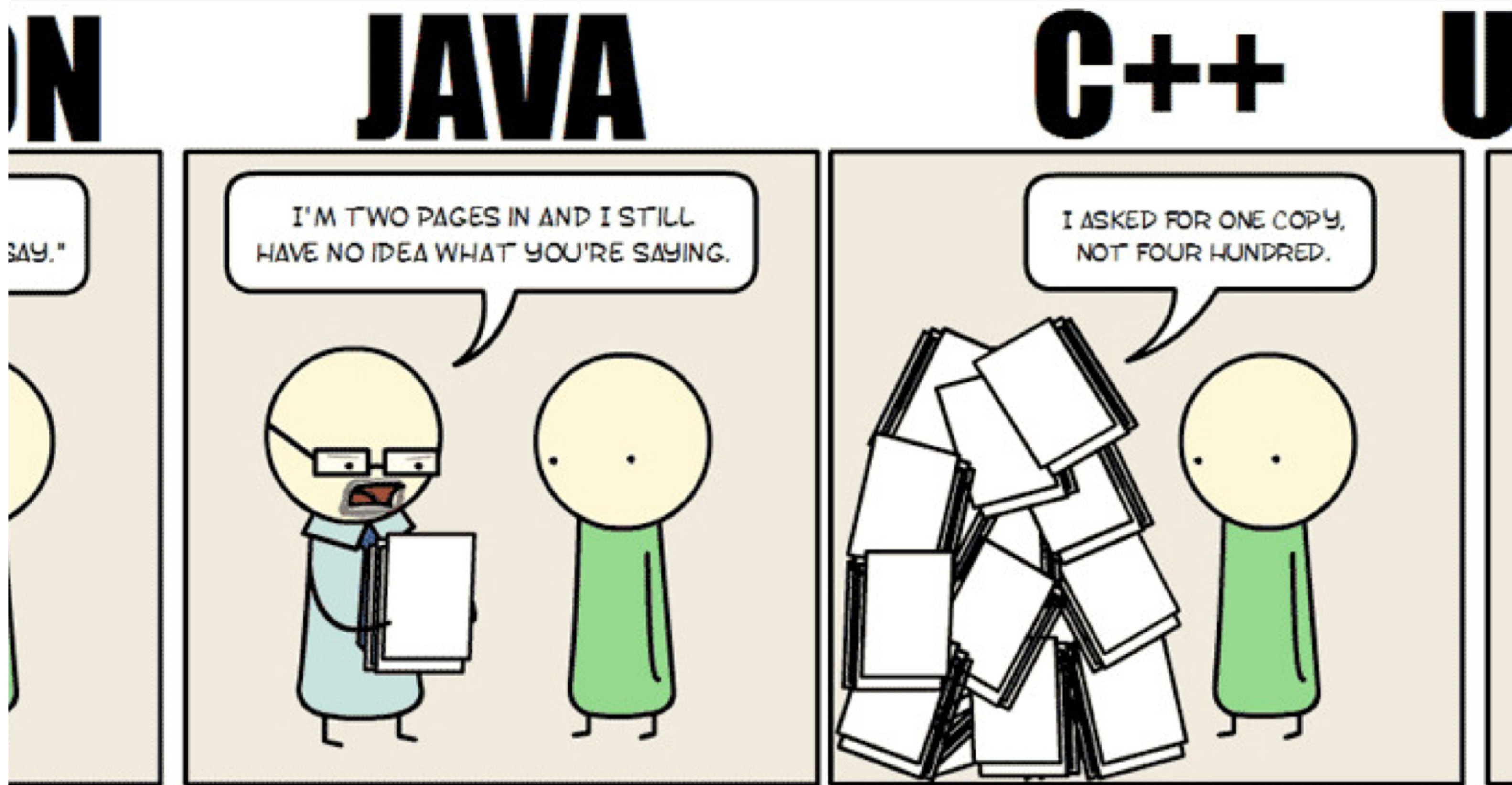
```
for (int i = 0; i < this->column_count(); i++) {
  auto value = values[i];
  this->segments[i]->append(value);
}
```

```
for (int i = 0; i < column_count(); i++) {
  auto value = values[i];
  _segments[i]->append(value);
}
```

```
for (int i = 0; i < column_count(); i++) {
  auto& value = values[i];
  _segments[i]->append(value);
}
```

```
for (int i = 0; i < column_count(); i++) {
  const auto& value = values[i];
  _segments[i]->append(value);
}
```

Deleting Copy Constructors



Deleting Copy Constructors

- Big classes in our database should not be copyable
- Deleted copy constructors should be default
- In this sprint: `BaseSegment`

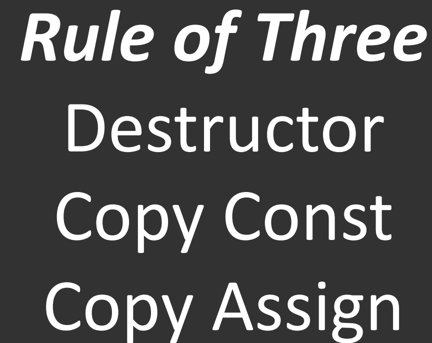
HPI Hasso Plattner Institut

Avoiding Copies

- We want to avoid unnecessary copies as much as possible
- (Some copies make sense – most times, there is no point passing an integer by reference)
- How does the compiler know when to avoid copies and how can we help?

Avoiding Copies for a String

```
class string {
    char *buf;

public:
    string(const char *str) {
        size_t size = strlen(str) + 1;
        buf = (char*)malloc(size);
        memcpy(buf, str, size);
    }
    void print() { std::cout <<
};
```



Avoiding Copies for a String

```
class string {
    char *buf;
public:
    string(const char *str) {
        size_t size = strlen(str) + 1;
        buf = (char*)malloc(size);
        memcpy(buf, str, size);
    }
    ~string() { free(buf); }
    string(const string& that) {
        size_t size = strlen(that.buf) + 1;
        buf = (char*)malloc(size);
        memcpy(buf, that.buf, size);
    }
     string& operator=(const string & that) {[...]}
    void print() { std::cout << buf << std::endl; }
};
```

Avoiding Copies for a String

```
class string {
    char *buf;
public:
    string(const char *str) {
        size_t size = strlen(str) + 1;
        buf = (char*)malloc(size);
        memcpy(buf, str, size);
        std::cout << "allocated " << size << " bytes" << std::endl;
    }
    ~string() { free(buf); }
    string(const string& that) {
        size_t size = strlen(that.buf) + 1;
        buf = (char*)malloc(size);
        memcpy(buf, that.buf, size);
        std::cout << "allocated " << size << " bytes" << std::endl;
    }
    string& operator=(const string & that) {[...]}
    void print() { std::cout << buf << std::endl; }
};
```

Avoiding Copies for a String

```
int main() {
    string a("test");
    a.print();

    string b(a);
    b.print();

    string c = a;
    c.print();
}
```

```
[:~/Desktop/tmp] $ g++-6 test.cpp -std=c++03 -Wall -Wextra && ./a.out
allocated 5 bytes
test
allocated 5 bytes
test
allocated 5 bytes
test
```

Avoiding Copies for a String

```
// I also modifi[...]tements to print the string

int main() {
    std::vector<stri[...]g> v;
    v.push_back("test");
}
```

implicit constructor

```
⌂36% [:~/Desktop/tmp] $ g++-6 test.cpp -std=c++1z -Wall -Wextra && ./a.out
allocated 5 bytes for "test"
allocated 5 bytes for "test"
```

HPI Hasso Plattner Institut

Avoiding Copies for a String

```
// I also modified the print statements to print the string

int main() {
    std::vector<string> v;
    v.push_back("foo");
    v.push_back("bar");
}
```

```
[:~/Desktop/tmp] $ g++-6 test.cpp -std=c++1z -Wall -Wextra && ./a.out
allocated 4 bytes for "foo" (constructor)
allocated 4 bytes for "foo" (copy constructor)
allocated 4 bytes for "bar" (constructor)
allocated 4 bytes for "bar" (copy constructor)
allocated 4 bytes for "foo" (copy constructor)
```

Avoiding Copies for a String

„The purpose of a move constructor is to steal as many resources as it can from the original object, as fast as possible, because the original does not need to have a meaningful value any more, because it is going to be destroyed (or sometimes assigned to) in a moment anyway."

https://akrzemi1.wordpress.com/2011/08/11/move-constructor/

Avoiding Copies for a String

```
class string {
    [...]
    string(string&& that) : buf(that.buf) {
        that.buf = nullptr;
        std::cout << "moved " << buf << std::endl;
    }
};
```

Avoiding Copies for a String

```
class string {
    [...]
    string(string&& that) : buf(that.buf) {
        that.buf = nullptr;
        std::cout << "moved " << buf << std::endl;
    }
    string& operator=(string&& that) {
        free(buf);
        buf = that.buf;
        that.buf = nullptr;
        std::cout << "moved " << buf << std::endl;
        return *this;
    }
};
```

Avoiding Copies for a String

```
string get_test() {
    return string("test");
}

int main() {
    std::vector<string> v;
    v.push_back("foo");
    v.push_back(get_test());
}
```

```
[:~/Desktop/tmp] $ g++-6 test.cpp -std=c++1z -Wall -Wextra && ./a.out
allocated 4 bytes for "foo" (constructor)
moved foo
allocated 5 bytes for "test" (constructor)
moved test
allocated 4 bytes for "foo" (copy constructor)
```

Avoiding Copies for a String

```
class string {
    [...]
    string(string&& that) noexcept : buf(that.buf) {
        that.buf = nullptr;
        std::cout << "moved " << buf << std::endl;
    }
    string& operator=(string&& that) noexcept {

    }
};
```

If a search for a matching exception handler leaves a function marked `noexcept` or `noexcept(true)`, `std::terminate` is called immediately.

```
[:~/Desktop/tmp] $ g++-6 test.cpp -std=c++1z -Wall -Wextra && ./a.out
allocated 4 bytes for "foo" (constructor)
moved foo
allocated 5 bytes for "test" (constructor)
moved test
moved foo
```

Avoiding Copies for a String

```
int main() {
    string a("baz");
    std::vector<string> v;

    v.push_back(a);

    // we'll never use a again...
}
```

```
[:~/Desktop/tmp] $ g++-6 test.cpp -std=c++1z -Wall -Wextra && ./a.out
allocated 4 bytes for "baz" (constructor)
allocated 4 bytes for "baz" (copy constructor)
```

Avoiding Copies for a String

```
int main() {
    string a("baz");
    std::vector<string> v;

    v.push_back(std::move(a));

    // we'll never use a again...
}
```

```
[:~/Desktop/tmp] $ g++-6 test.cpp -std=c++1z -Wall -Wextra && ./a.out
allocated 4 bytes for "baz" (constructor)
moved baz
```

Hasso Plattner Institut

Avoiding Copies for a String

```
int main() {
    string a("baz");
    std::vector<string> v;

    v.push_back(std::move(a));

    // we'll never use a again...

    string b(a);
    // but you promised :(
}
```

```
[:~/Desktop/tmp] $ g++-6 test.cpp -std=c++1z -Wall -Wextra -O3 && ./a.out
allocated 4 bytes for "baz" (constructor)
moved baz
Segmentation fault: 11
```

```
string(string&& that) : buf(that.buf) {
    that.buf = nullptr;
}
```

HPI Hasso Plattner Institut

Use after move

- From a submission:

```
TEST_F(StorageTableTest, MoveConstructor)
{
   Table t2 = std::move(t);
   EXPECT_EQ(t.chunk_count(), 0u);
   EXPECT_EQ(t2.chunk_count(), 1u);
 }
```

What is std::move?

- What does std::move do?
- From an instruction POV: Nothing
- „std::move is used to *indicate* that an object t may be "moved from", i.e. allowing the efficient transfer of resources from t to another object.
- „In particular, std::move produces an xvalue expression that identifies its argument t. It is exactly equivalent to a static_cast to an rvalue reference type."

```
template <typename T>
typename remove_reference<T>::type&& move(T&& arg) {
  return static_cast<typename remove_reference<T>::type&&>(arg);
}
```

Ensure Moves

```
~~string(const string& that) {~~
~~    size_t size = strlen(that.buf) + 1;~~
~~    buf = (char*)malloc(size);~~
~~    memcpy(buf, that.buf, size);~~
~~}~~
string(const string& that) = delete;
```

Hasso Plattner Institut

What does this mean for Opossum?

- You hopefully now have a better idea why we delete the copy constructors and how moves work

```
void Table::append(std::initializer_list<AllTypeVariant> values) {
  if (_chunk_size > 0 && _chunks.back().size() == _chunk_size) {
    Chunk new_chunk;
    for (auto&& type : _column_types) {
      new_chunk.add_segment(make_shared_by_data_type<BaseSegment,
                                                    ValueSegment>(type));
    }
    _chunks.push_back(std::move(new_chunk));
  }

  _chunks.back().append(values);
}
```

Temporaries are automatically xvalues and should not be moved

Named Return Value Optimization

```
string get_foo() {
    return string("foo");
}

string get_baz() {
    return std::move(string("baz"));
}

int main() {
    get_foo();
    get_baz();
}
```

```
[:~/Desktop/tmp] $ g++-6 test.cpp -std=c++1z -Wall -Wextra -O3 && ./a.out
allocated 4 bytes for "foo" (constructor)
allocated 4 bytes for "baz" (constructor)
moved baz
```

How to keep track of that?

- All those things that you should look out for might be hard to track, especially in the beginning
- Luckily, there are tools that can help with that
- In the Hyrise CI, we automatically run clang-tidy

clang-tidy

```cpp
#include <iostream>
#include <string>


void print(std::string string) {
  std::cout << string << std::endl;
}


int main() {
  auto foo = std::string{"foo"};
  print(foo);
}
```

```
/Users/markus/tmp/tidy.cpp:4:24: error: the parameter 'string' is copied for each invocation but only used as a
 const reference; consider making it a const reference [performance-unnecessary-value-param,-warnings-as-errors
]
void print(std::string string) {
                       ^
           const      &
```

clang-tidy

```cpp
#include <iostream>
#include <string>

void print(std::string string) {
  std::cout << string << std::endl;
}

int main() {
  auto foo = std::string{"foo"};
  print(std::move(foo));
  std::cout << foo << std::endl;
}
```

```
/Users/markus/tmp/tidy.cpp:11:16: error: 'foo' used after it was moved [bugprone-use-after-move,-warnings-as-errors]
  std::cout << foo << std::endl;
               ^
/Users/markus/tmp/tidy.cpp:10:9: note: move occurred here
  print(std::move(foo));
        ^
```

clang-tidy

```
class MyClass {
  int i;
}


int main() {
}
```

```
/Users/markus/tmp/tidy.cpp:2:7: error: invalid case style for private member 'i' [readability-identifier-naming
,-warnings-as-errors]
  int i;
      ^
      _i
```

clang-tidy

```cpp
#include <iostream>


int main(int argc, char** argv) {
  auto has_argument = argc > 1;
  if (has_argument == false) {
    std::cout << "Please pass argument." << std::endl;
  }
}
```

```
/Users/markus/tmp/tidy.cpp:5:23: error: redundant boolean literal supplied to boolean operator [readability-simplify-boolean-expr,-warnings-as-errors]
  if (has_argument == false) {
      ~~~~~~~~~~~~~~~~~~^~~~~~
      !has_argument
```

clang-tidy

```cpp
#include <iostream>


int main(int argc, char** argv) {
  auto has_argument = argc > 1;
  if (has_argument) {
    std::cout << "Received an argument." << std::endl;
  } else {
    std::cout << "Received an argument." << std::endl;
  }
}
```

```
/Users/markus/tmp/tidy.cpp:5:3: error: if with identical then and else branches [bugprone-branch-clone,-warning
s-as-errors]
  if (has_argument) {
  ^
/Users/markus/tmp/tidy.cpp:7:5: note: else branch starts here
  } else {
    ^
```

clang-tidy

Unit tests, clang-tidy, and other CI steps are a safety net – do not rely on them!

I like, I wish

Next Steps

- Any Questions about Sprint 2?
- Hand-in: 12. 11. 2019, 11:59 pm