WOCHEN 2
BYOD
AGENDA

- Organization
- Templates
- RAII
- Smart Pointers
If you have not joined us at Piazza:

- piazza.com/hpi.uni-potsdam.de/fall2017/byod

Any problems during setup?
What would need to change to allow multiplication of Ints and Doubles?
TEMPLATES - FUNCTIONS

```cpp
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 }
1 chunk.add_column(std::make_shared<ValueColumn<int>>());
2 chunk.add_column(std::make_shared<ValueColumn<float>>());
3
4 std::vector<std::shared_ptr<ValueColumn>> _columns;
5
6 std::vector<std::shared_ptr<ValueColumn<int>>> _columns;
7
8 std::vector<std::shared_ptr<BaseColumn>> _columns;
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
};
RAII - MOTIVATION

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

```cpp
1 int main() {
2    ClassA ca;
3
4    ca.someOperation();
5    ca.someOperationB();
6    ca.someOperationC();
7 }
```
RAII - MOTIVATION

```cpp
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 - MOTIVATION

```cpp
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 - 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
POINTERS – HAVE FUN KEEPING TRACK

```cpp
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
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
  - Deep Copy (Copy on Write)
  - Reference Linking
SMART POINTERS – REFERENCE LINKING
SMART POINTERS – OWNERSHIP HANDLING

- Ideas? - Standard does not specify an implementation
  - Deep Copy (Copy on Write)
  - Reference Linking
  - Reference Counting
  - ...

18
SMART POINTERS – REFERENCE COUNTING

- Issue with reference counting?
  - Overhead
  - Synchronization issues
- How to implement reference counting?
SMART POINTERS – REFERENCE COUNTING
SMART POINTERS – REFERENCE COUNTING
SMART POINTERS – REFERENCE COUNTING
SMART POINTERS – REFERENCE COUNTING

A `shared_ptr<T>` contains a pointer of type `T*` and an object of type `__shared_count`. The `shared_count` contains a pointer of type `_Sp_counted_base*` which points to the object that maintains the reference-counts and destroys the managed resource.

gcc libstdc++ memory manual
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
There is some variation to this rule. We will get into this later.
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

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->NonConstMemberFunction();
p2 = std::make_shared<SomeClass>();
p3->ConstMemberFunction();
p3->NonConstMemberFunction();
p3 = std::make_shared<const SomeClass>();
SMART POINTERS – CONSTNESS

```cpp
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->NonConstMemberFunction();
9  p2 = std::make_shared<SomeClass>();
10
11 p3->ConstMemberFunction();
12 p3->NonConstMemberFunction();
13 p3 = std::make_shared<const SomeClass>();
```
» Code review information
» Compression and encoding
» Presentation of sprint 2