Object Relational Mapping

In C++, With ODB

Zoltán Borók-Nagy (boroknagyz@gmail.com)

Agenda

- Motivation
- Popular ORM patterns for database handling
  - Active Record
  - Data Access Object (DAO)
- ODB, an advanced ORM tool for C++
- Hot it works?
  - POD types
  - Object Relationships
  - Inheritance
- Querying the database

Motivation

- OO languages give us classes and structs
- Objects only exists while the program runs
- In some cases we need persistence
- We want to store relationships and write queries

- ORM helps you to save objects and their relations in a database

Motivation

```
String sql = "SELECT ... FROM persons WHERE id = 10"
DbCommand cmd = new DbCommand(connection, sql);
Result res = cmd.Execute();
String name = res[0]["FIRST_NAME"];

/* ======================================== */

Person p = database.load<Person>(10);
string name = p.getFirstName();
```

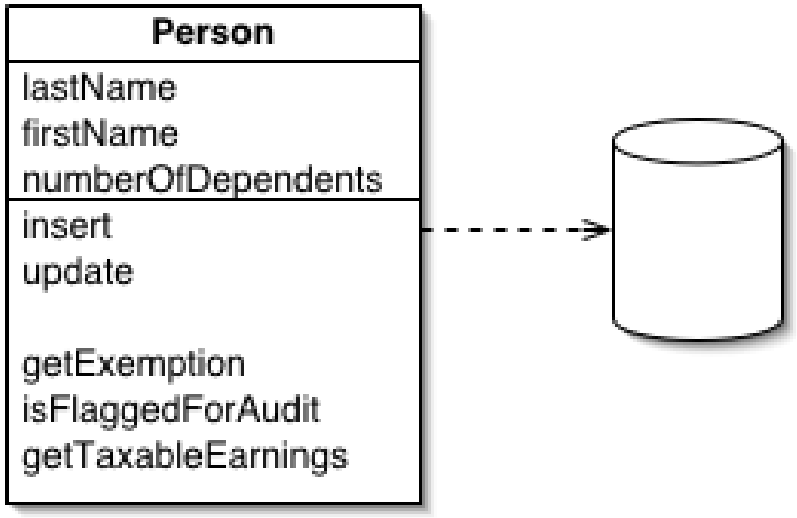
Active Record

„An object that wraps a row in a database table or view, encapsulates the database access, and adds domain logic on that data.”

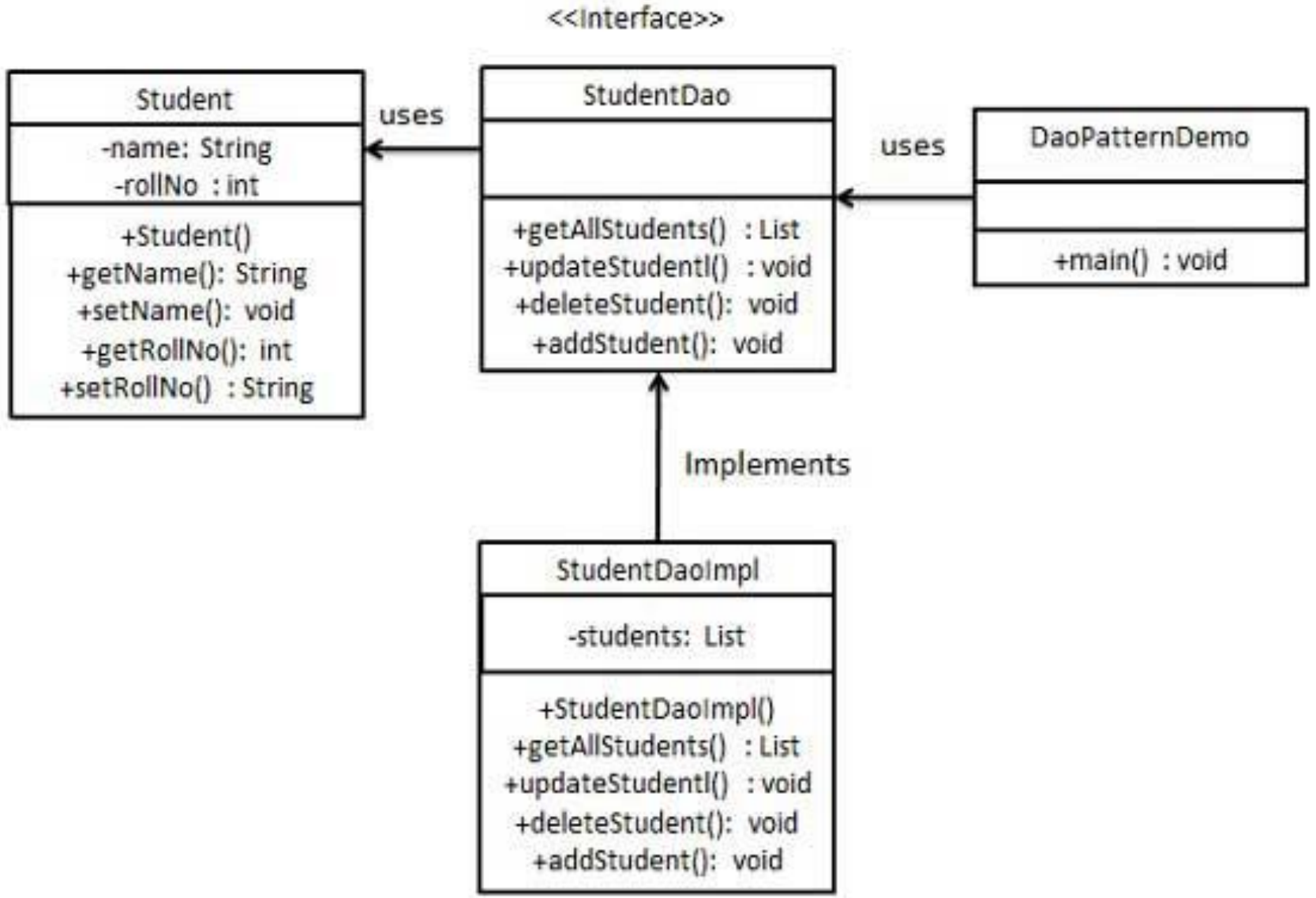
- class structure match to the record structure
- business logic
- construction methods
- update methods
- static finder methods

Active Record

```
Person p = new Person();

p.setFirstName("Carl");
p.setLastName("Sagan");

p.insert();

Person p2 = Person::find(42);
p2.setBirthDate("03/11/1952");
p2.update();
```



Data Access Object (DAO)



Data Access Object (DAO)

```
StudentDao studentDao = new StudentDaoImpl();
//print all students
for (Student student : studentDao.getAllStudents()) {
   System.out.println("Student: [RollNo : " + student.getRollNo() + ", Name : " + student.getName() + " ]");
}
//update student
Student student =studentDao.getAllStudents().get(0);
student.setName("Michael");
studentDao.updateStudent(student);

//get the student
studentDao.getStudent(0);
System.out.println("Student: [RollNo : " + student.getRollNo() + ", Name : " + student.getName() + " ]");
```

ORM tools

- Generate mapping automatically
- Controlled by annotations, pragmas, attributes
- Can handle complex class hierarchies, relationships
- Much less boilerplate code

- Free and commercial frameworks

Mapping a simple class/struct

```
#pragma db object
struct Person
{
  string email;
  string name;
  unsigned short age;
};
```

Person		
email	name	age
nc@gmail.com	Nicole Kidman	46

```
Person p{“nc@gmail.com”,
         “Nicole Kidman”,
         46};
```

```
CREATE TABLE Person(
  email VARCHAR (255) NOT NULL PRIMARY KEY,
  name VARCHAR (255),
  age    INTEGER);

INSERT INTO Person VALUES (‘nc@gmail.com’, ‘Nicole Kidman’, 46);
```

Unidirectional Relationships

To-One relation

```
#pragma db object
class employer
{
  ...

  #pragma db id
  std::string name;
};
```

```
#pragma db object
class employee
{
  ...
  #pragma db id
  unsigned long id;
  std::string name;

  #pragma db not_null
  shared_ptr<employer> employer;
};
```

To-One relation

```
CREATE TABLE employer (
  name VARCHAR (255) NOT NULL PRIMARY KEY);

CREATE TABLE employee (
  id BIGINT UNSIGNED NOT NULL PRIMARY KEY,
  name VARCHAR (255),
  employer VARCHAR (255) NOT NULL REFERENCES employer (name));
```

employer
name
BigBoss
OtherBoss

employee		
id	name	employer
1	John	BigBoss
2	Jane	OtherBoss

To-Many relation

```
#pragma db object
class project
{
  ...

  #pragma db id
  std::string name;
};
```

```
#pragma db object
class employee
{
  ...

  #pragma db id
  unsigned long id;

  #pragma db value_not_null unordered
  std::vector<shared_ptr<project> > projects;
};
```

To-Many relation

```
CREATE TABLE project (
  name VARCHAR (255) NOT NULL PRIMARY KEY);

CREATE TABLE employee (
  id BIGINT UNSIGNED NOT NULL PRIMARY KEY);

CREATE TABLE employee_projects (
  object_id BIGINT UNSIGNED NOT NULL,
  value VARCHAR (255) NOT NULL REFERENCES project (name));
```

employee
id
1
2

employee_project	
object_id	value
1	CodeCompass
1	CodeChecker
2	CodeCompass

project
name
CodeCompass
CodeChecker

Bidirectional Relationships

One-To-One

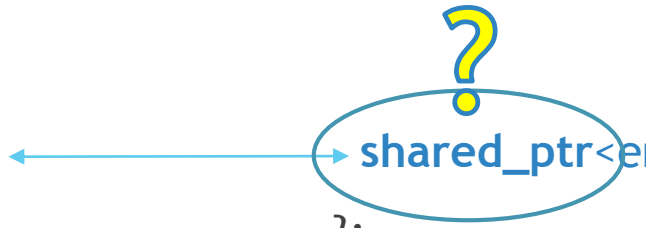
```
#pragma db object
class employee
{
  ...

  #pragma db id
  string name;

  #pragma db not_null
  shared_ptr<position> position;
};
```

```
#pragma db object
class position
{
  ...

  #pragma db id
  unsigned long id;


  shared_ptr<employee> employee;
};
```

One-To-One

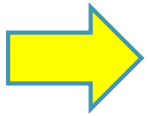
```
#pragma db object
class employee
{
  ...

  #pragma db id
  string name;

  #pragma db not_null
  shared_ptr<position> position;
};
```

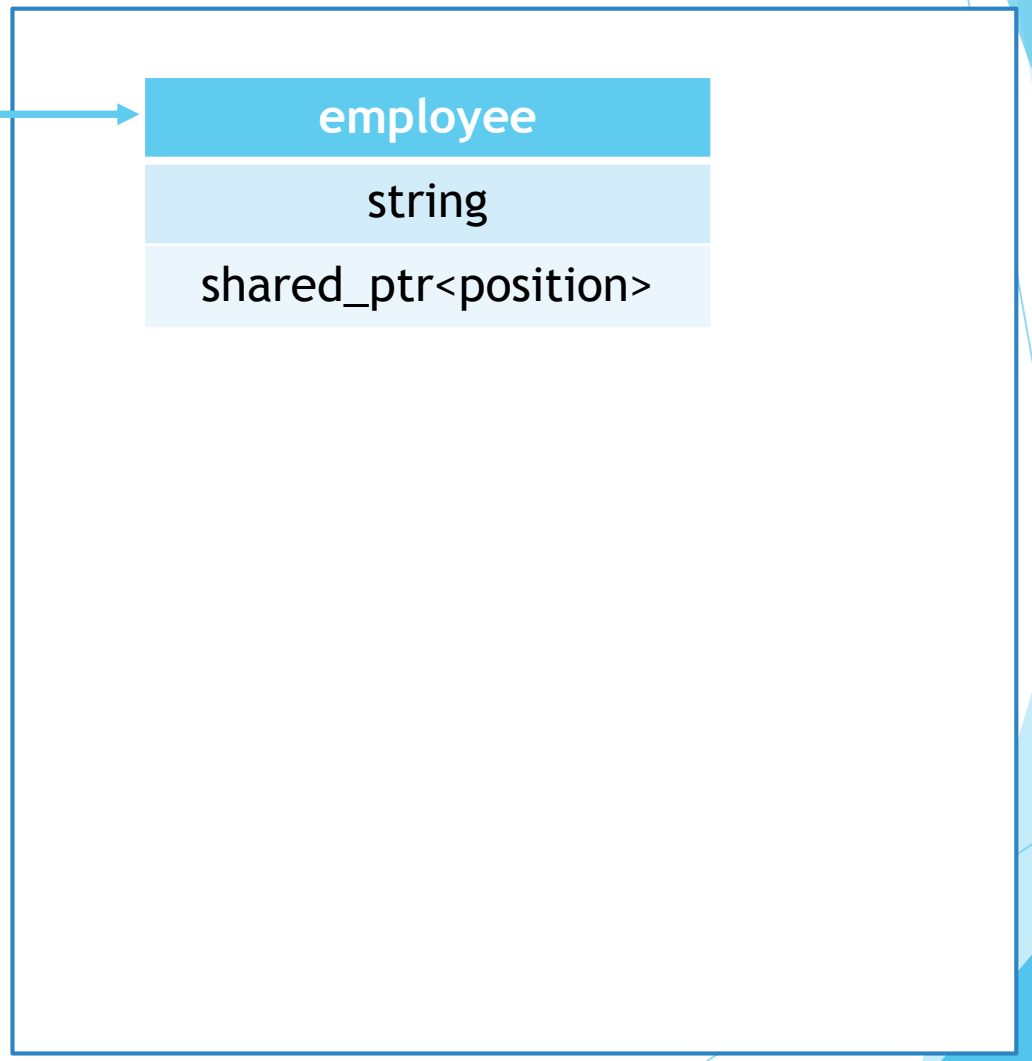
```
#pragma db object
class position
{
  ...

  #pragma db id
  unsigned long id;

  shared_ptr<employee> employee;
};
```



Intermezzo: strong vs weak references

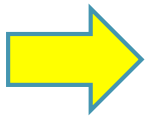
```
void f()
{
  auto se =
    make_shared<employee>();
…



}
```

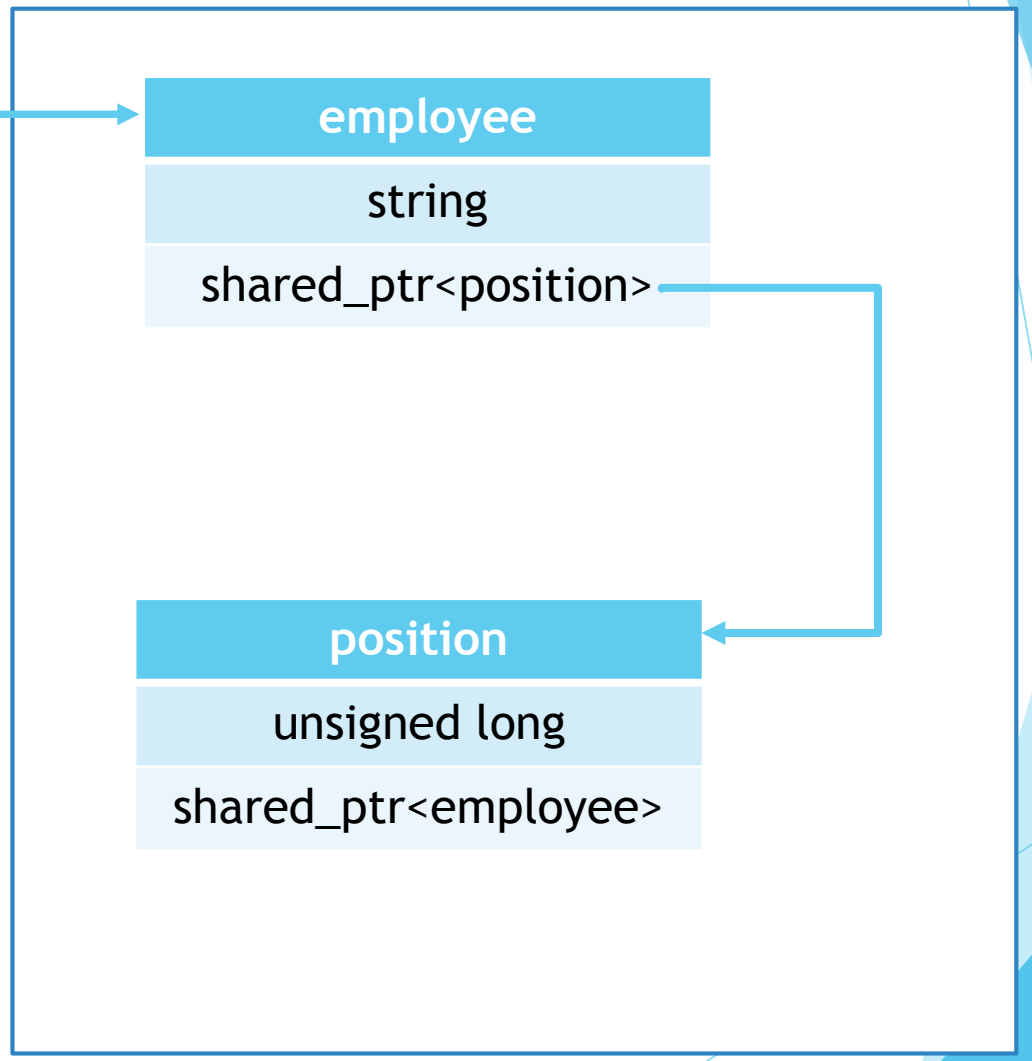




Intermezzo: strong vs weak references

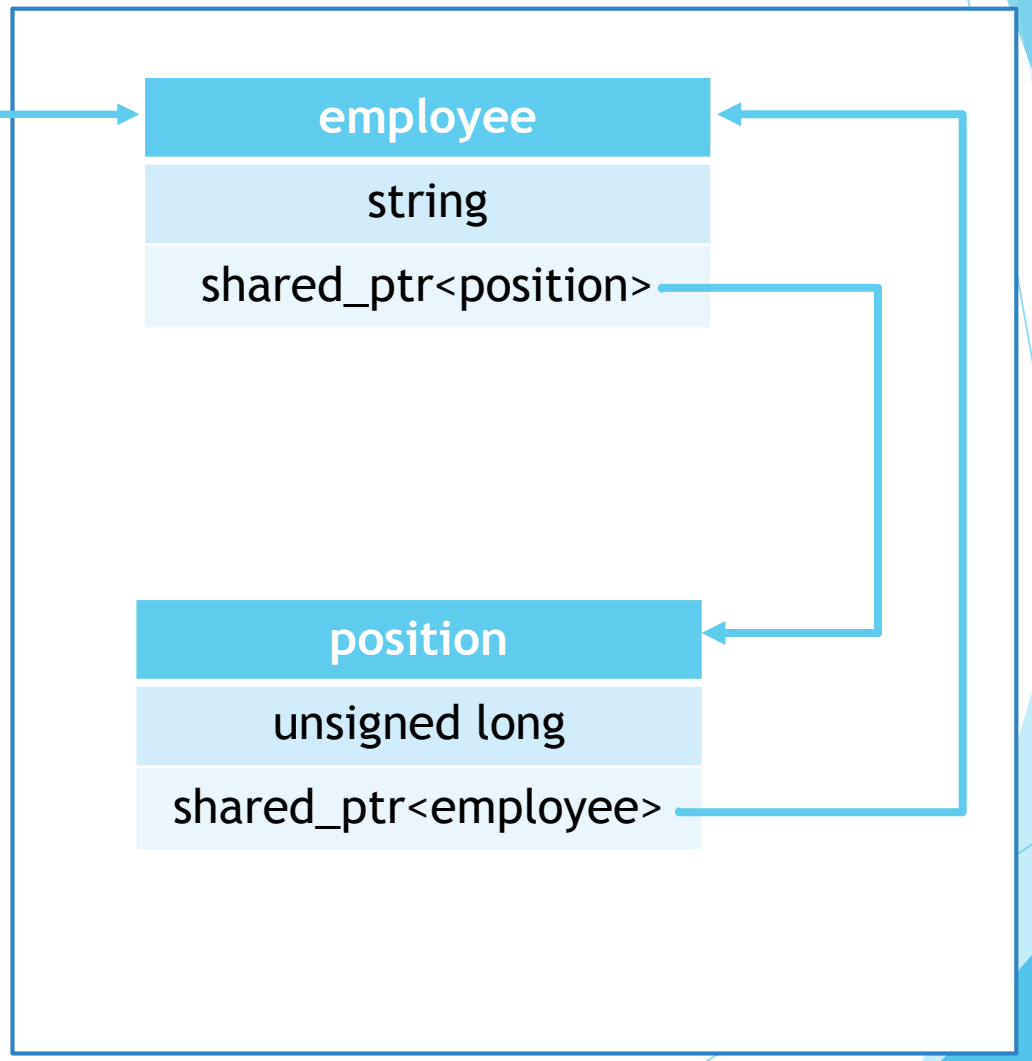
```
void f()
{
  auto se =
    make_shared<employee>();
  se->position =
    make_shared<position>();


}
```





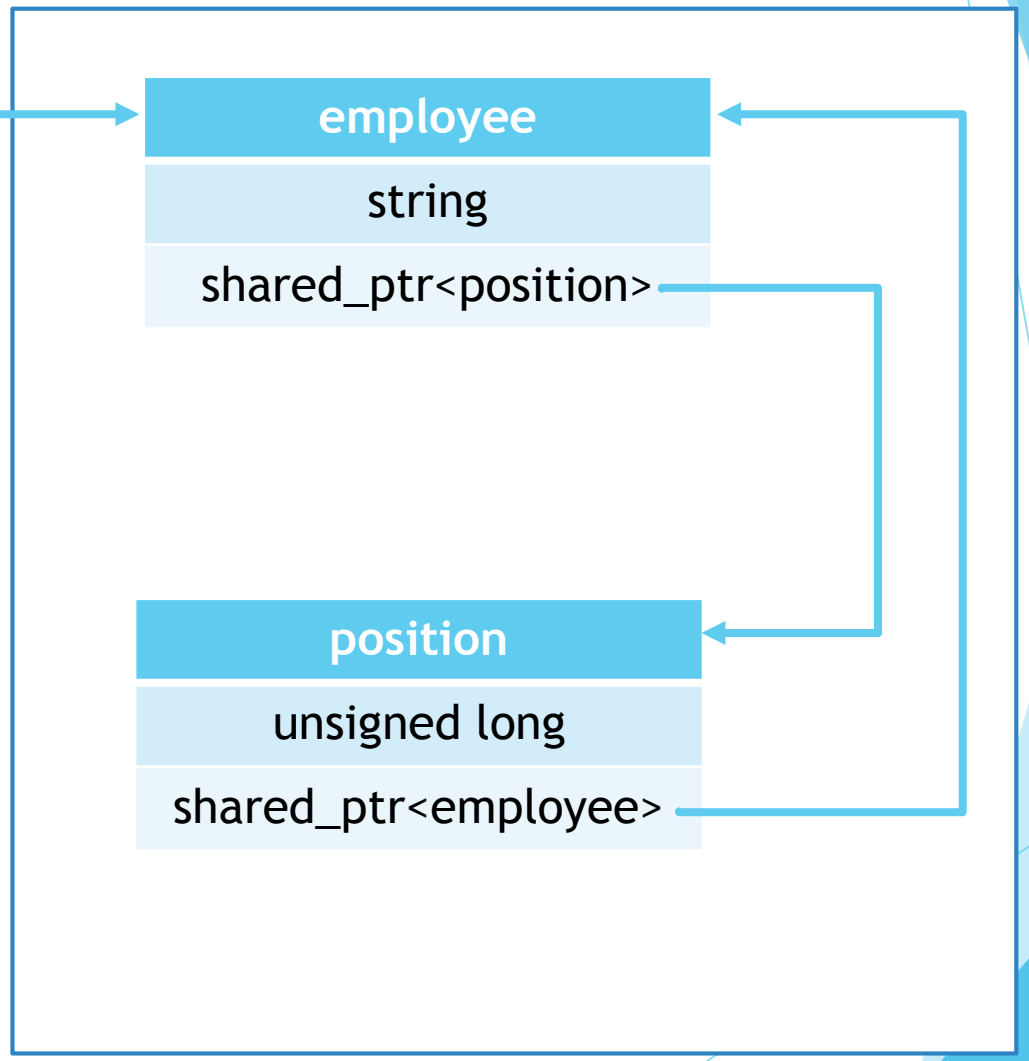
Intermezzo: strong vs weak references

```
void f()
{
  auto se =
    make_shared<employee>();
  se->position =
    make_shared<position>();
  se->position->employee = se;

}
```



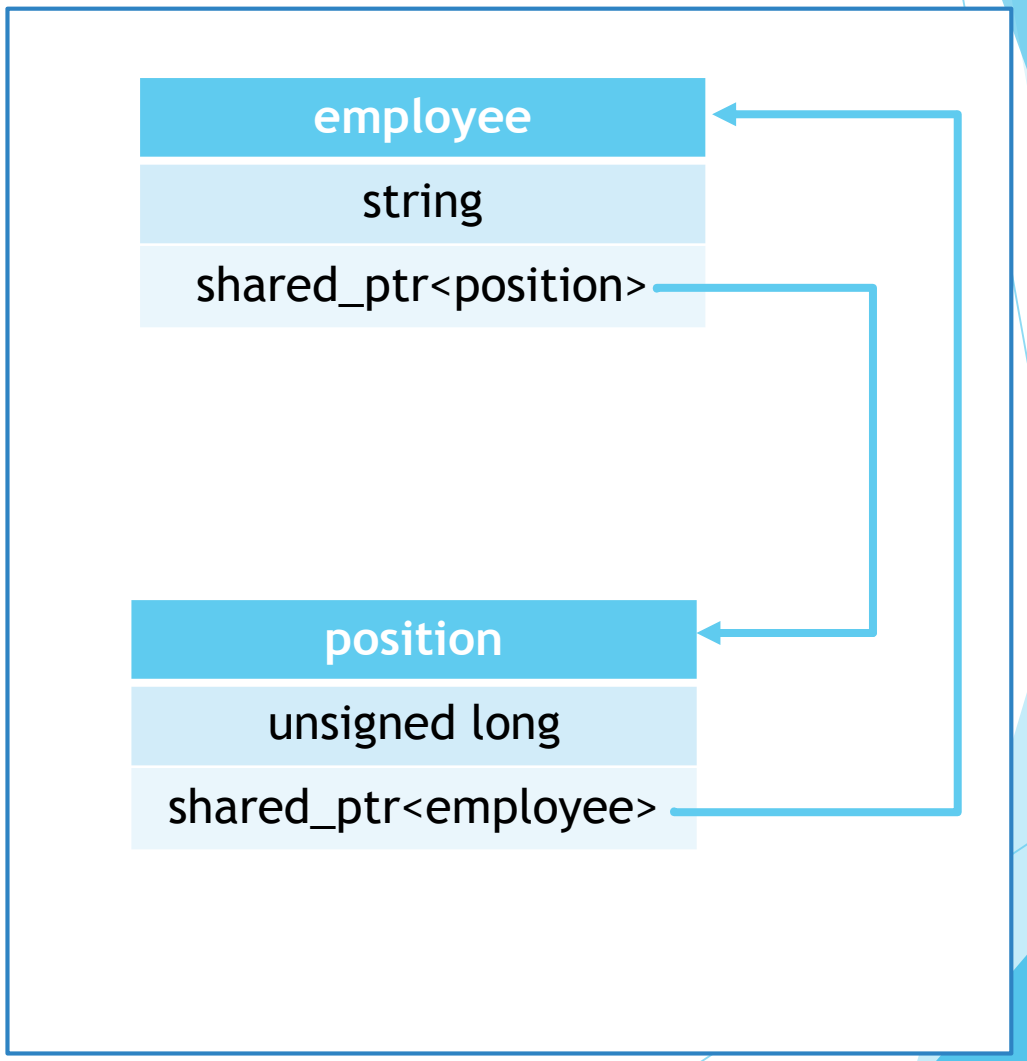
Intermezzo: strong vs weak references

```
void f()
{
  auto se =
    make_shared<employee>();
…

// se.~shared_ptr();
}
```



Intermezzo: strong vs weak references

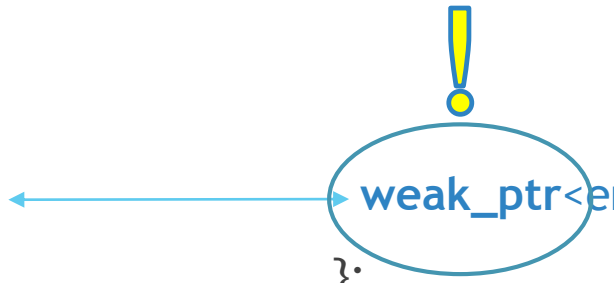
```
void f()
{
  auto se =
    make_shared<employee>();
…

// se.~shared_ptr();
}
```



One-To-One (with weak_ptr)

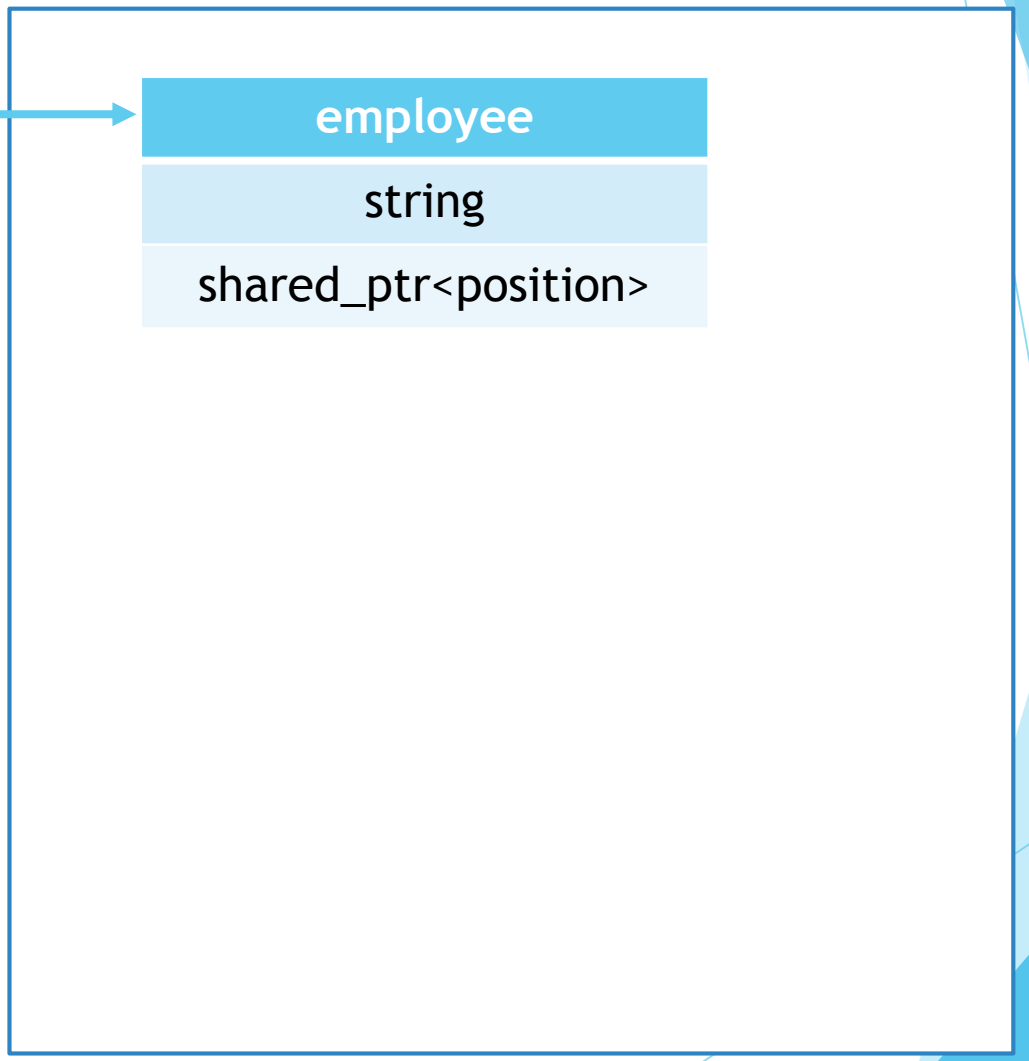
```
#pragma db object
class employee
{
  ...

  #pragma db id
  string name;

  #pragma db not_null
  shared_ptr<position> position;
};
```

```
#pragma db object
class position
{
  ...

  #pragma db id
  unsigned long id;

  weak_ptr<employee> employee;
};
```



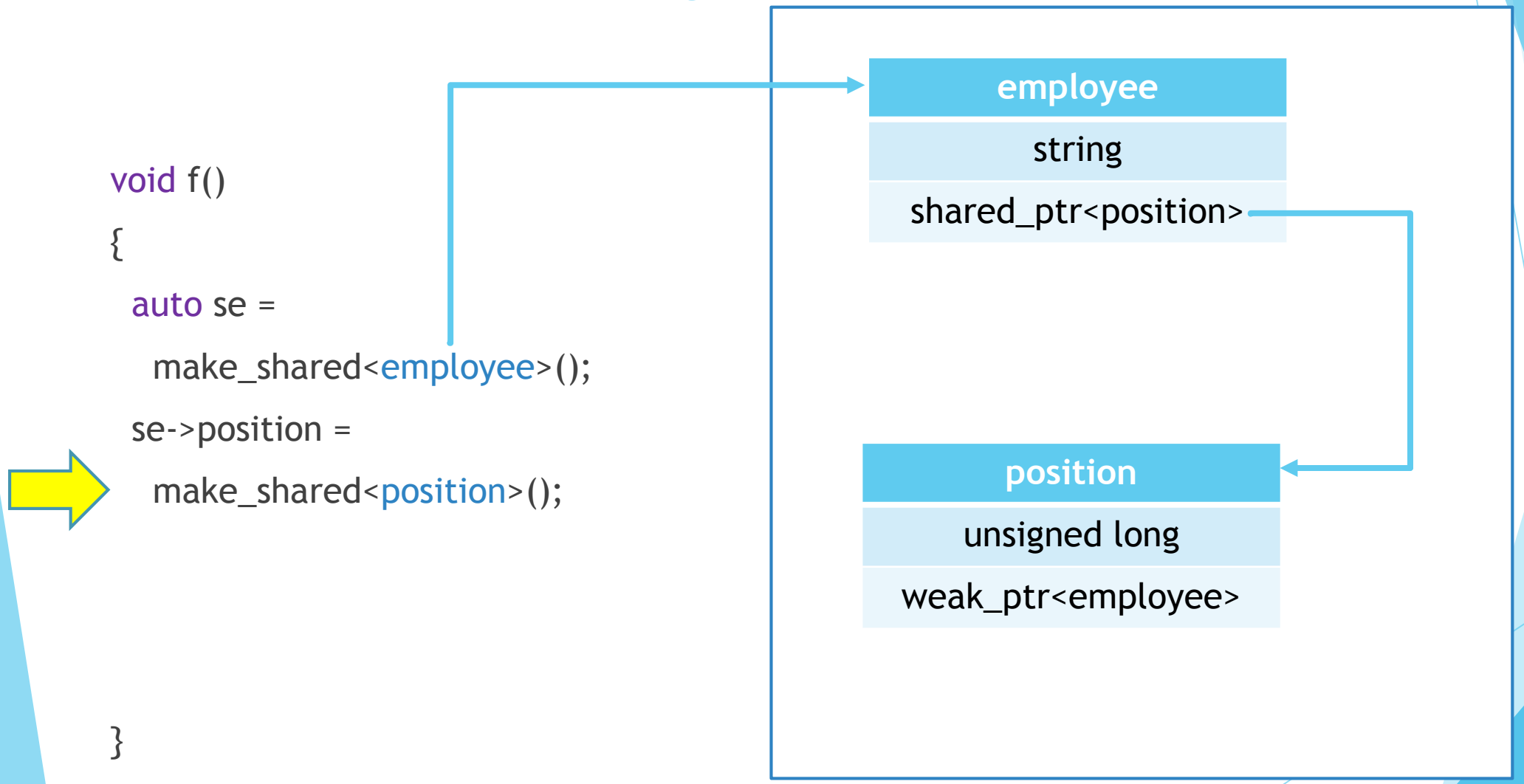
Intermezzo: strong vs weak references

```
void f()
{
  auto se =
    make_shared<employee>();
…



}
```



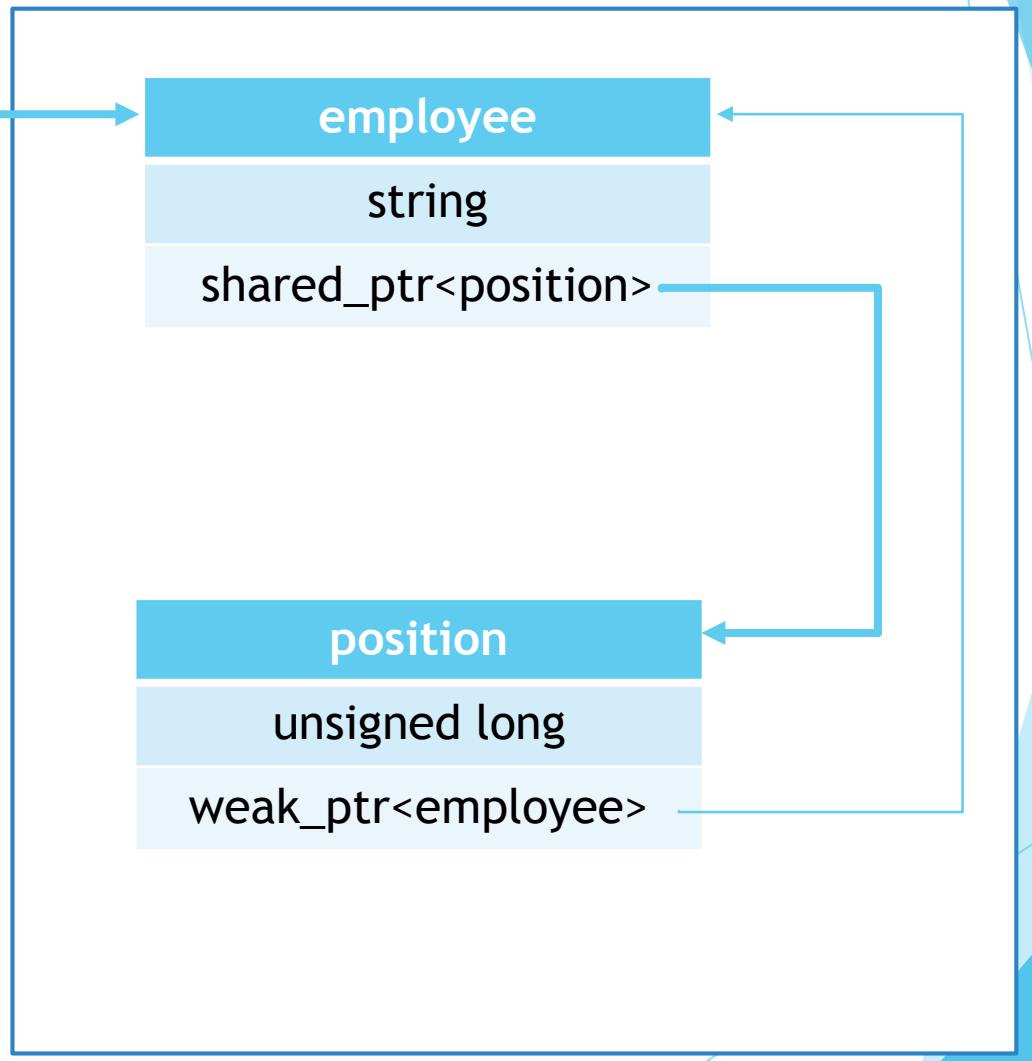
Intermezzo: strong vs weak references

```
void f()
{
  auto se =
    make_shared<employee>();
  se->position =
    make_shared<position>();



}
```



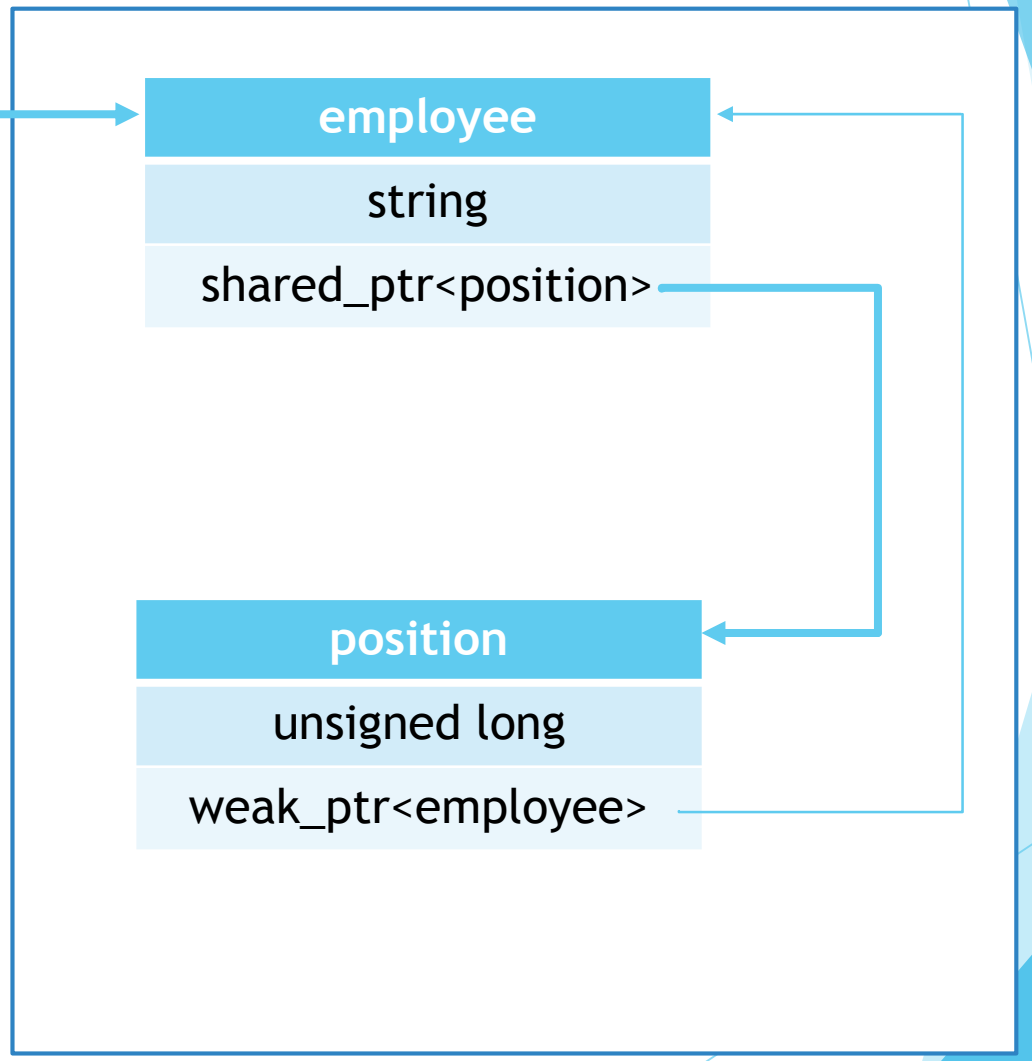
Intermezzo: strong vs weak references

```
void f()
{
  auto se =
    make_shared<employee>();
  se->position =
    make_shared<position>();
  se->position->employee = se;

}
```



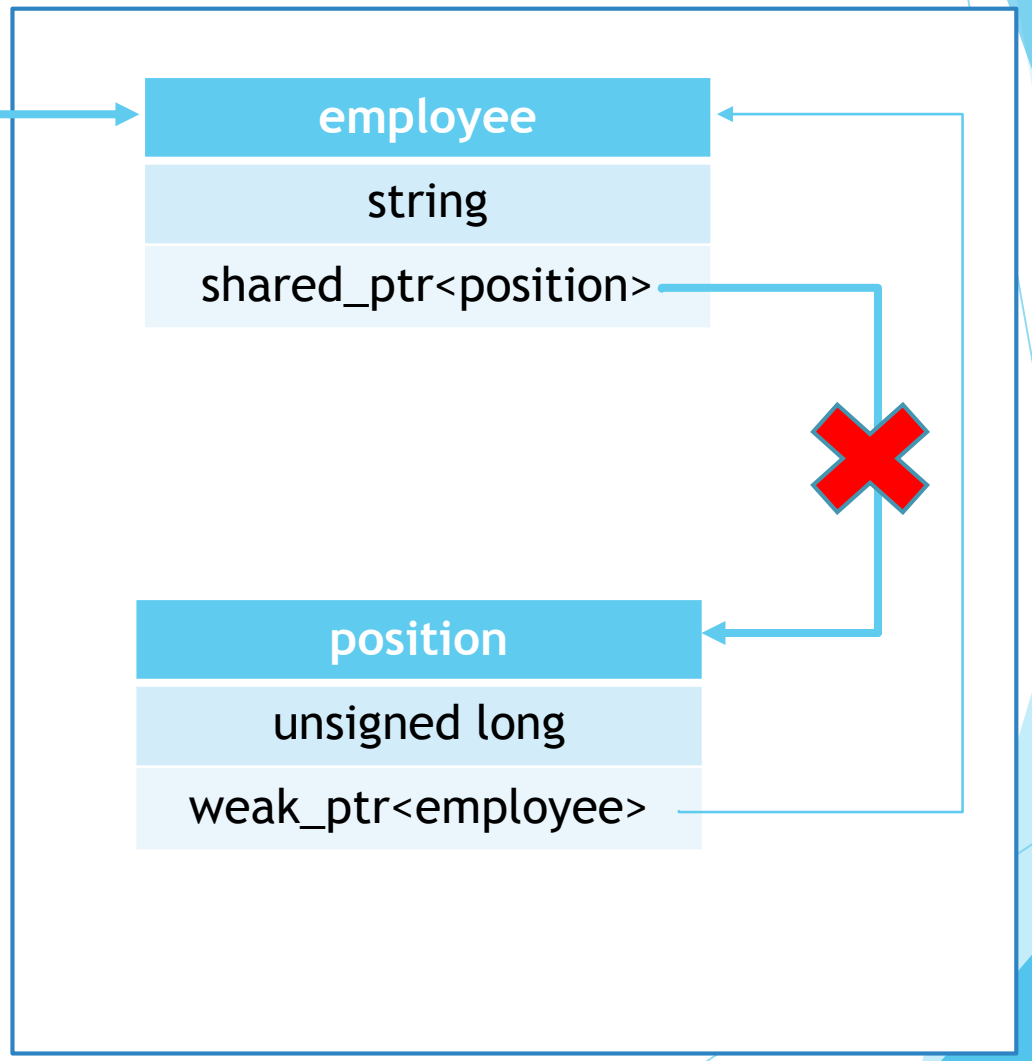
Intermezzo: strong vs weak references

```
void f()
{
  auto se =
    make_shared<employee>();
…

// se.~shared_ptr();
}
```



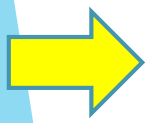
Intermezzo: strong vs weak references

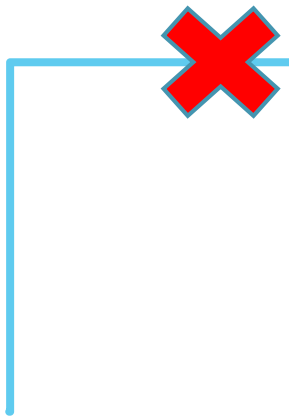
```
void f()
{
  auto se =
    make_shared<employee>();
…

// se.~shared_ptr();
}
```

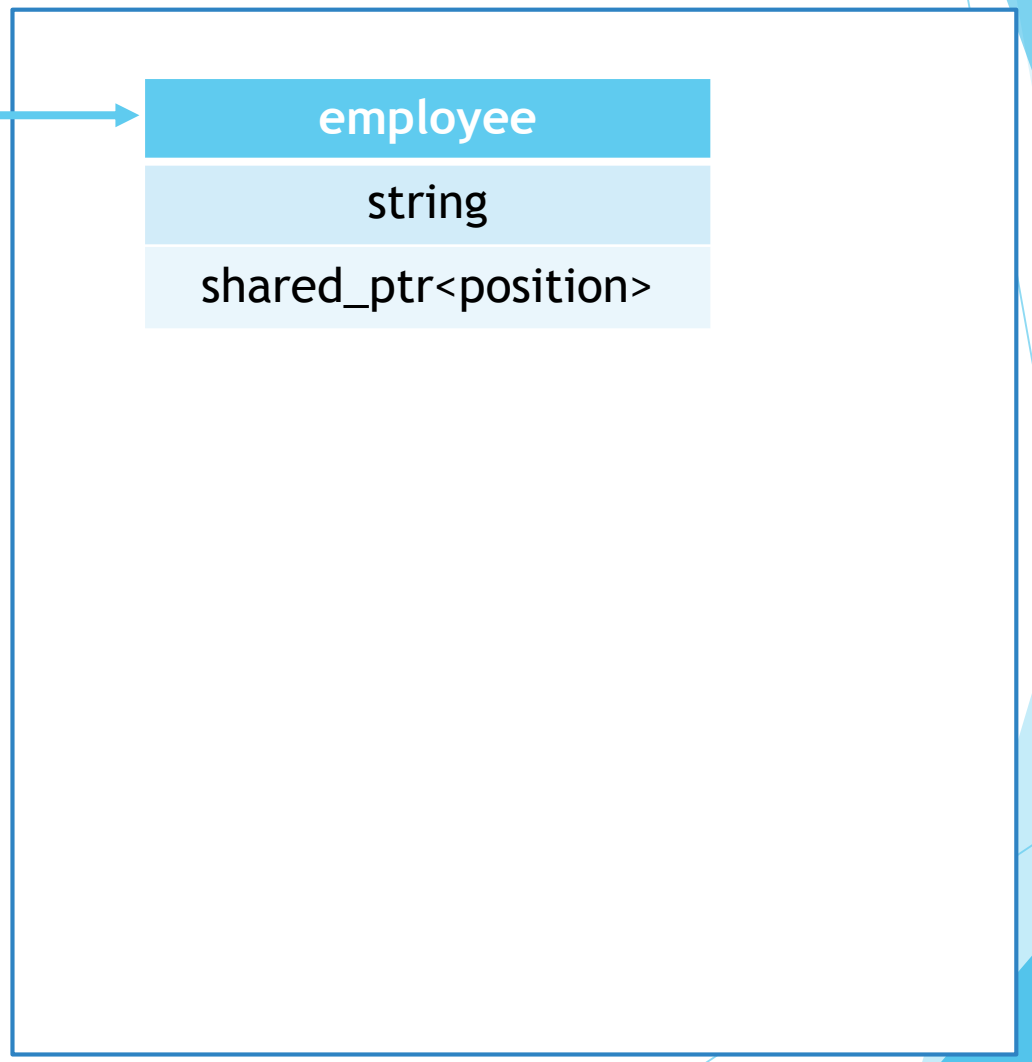


Intermezzo: strong vs weak references

```
void f()
{
  auto se =
    make_shared<employee>();
…

// se.~shared_ptr();
}
```

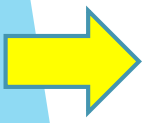


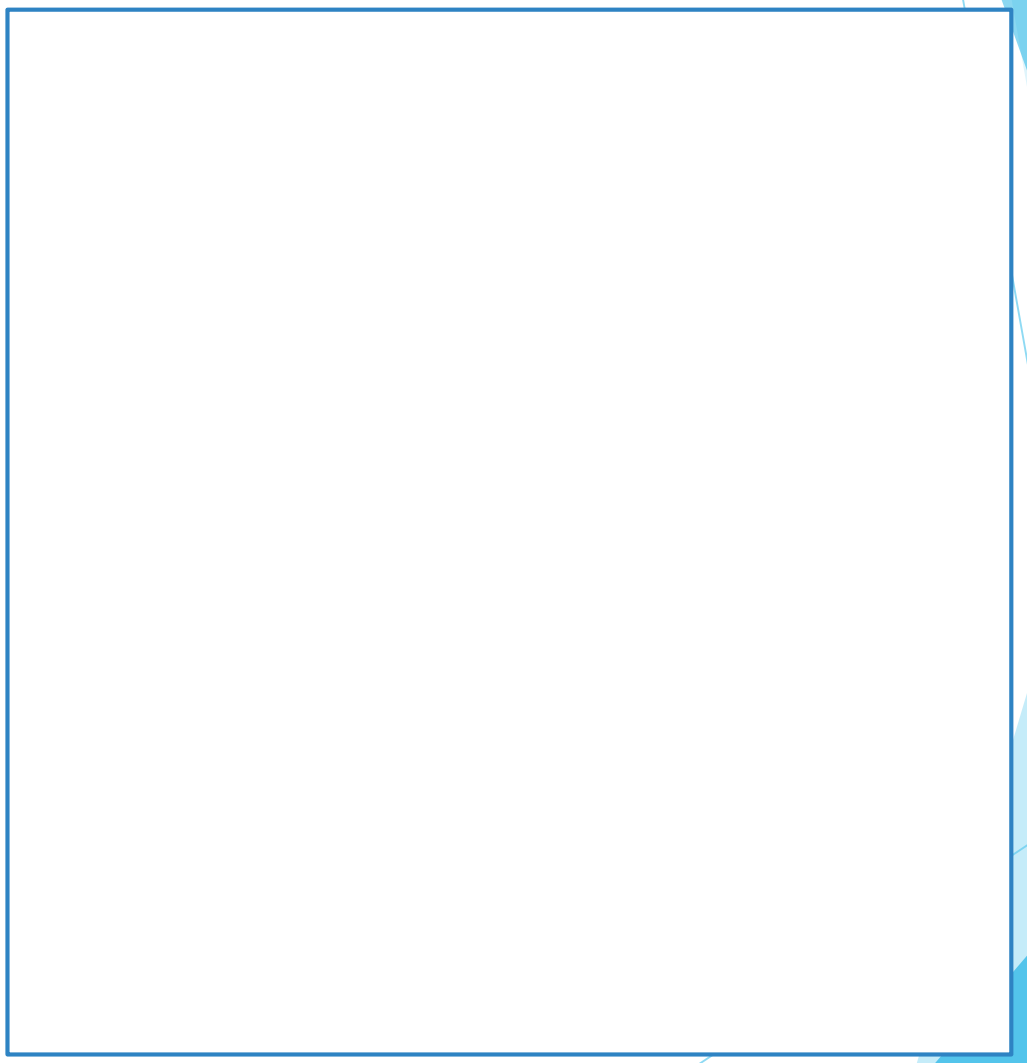




Intermezzo: strong vs weak references

```
void f()
{
  auto se =
    make_shared<employee>();
…

// se.~shared_ptr();
}
```





One-To-One

```
#pragma db object
class employee
{
  ...

  #pragma db id
  string name;

  #pragma db not_null
  shared_ptr<position> position;
};
```



```
#pragma db object
class position
{
  ...

  #pragma db id
  unsigned long id;


  weak_ptr<employee> employee;
};
```

One-To-One

```
CREATE TABLE position (
  id BIGINT UNSIGNED NOT NULL PRIMARY KEY,
  employee BIGINT UNSIGNED REFERENCES employee (id));

CREATE TABLE employee (
  name VARCHAR (255) NOT NULL PRIMARY KEY,
  position BIGINT UNSIGNED NOT NULL REFERENCES position (id));
```

employee	
name	position
John	1
Jane	2

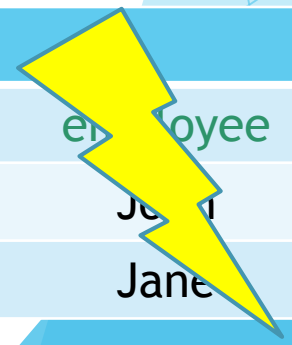
position	
id	employee
1	John
2	Jane

One-To-One

```
CREATE TABLE position (
  id BIGINT UNSIGNED NOT NULL PRIMARY KEY,
  employee BIGINT UNSIGNED REFERENCES employee (id));

CREATE TABLE employee (
  name VARCHAR (255) NOT NULL PRIMARY KEY,
  position BIGINT UNSIGNED NOT NULL REFERENCES position (id));
```

employee	
name	position
John	1
Jane	2

position	
id	employee
1	John
2	Jane



One-To-One (with inverse)

```
#pragma db object
class employee
{
  ...

  #pragma db id
  string name;

  #pragma db not_null
  shared_ptr<position> position;
};
```

```
#pragma db object
class position
{
  ...

  #pragma db id
  unsigned long id;
  #pragma db inverse(position)
  weak_ptr<employee> employee;
};
```

One-To-One (with inverse)

```
#pragma db object
class employee
{
  ...

  #pragma db id
  string name;

  #pragma db not_null
  shared_ptr<position> position;
};
```

```
#pragma db object
class position
{
  ...

  #pragma db id
  unsigned long id;
  #pragma db inverse(position)
  weak_ptr<employee> employee;
};
```

One-To-One (with inverse)

```
CREATE TABLE position (
  id BIGINT UNSIGNED NOT NULL PRIMARY KEY);

CREATE TABLE employee (
  name VARCHAR (255) NOT NULL PRIMARY KEY,
  position BIGINT UNSIGNED NOT NULL REFERENCES position (id));
```

employee	
name	position
John	1
Jane	2

position
id
1
2

One-To-Many

```
#pragma db object
class employee
{
  ...

  #pragma db id
  unsigned long id;

  #pragma db not_null
  shared_ptr<employer> employer;
};
```



```
#pragma db object
class employer
{
  ...

  #pragma db id
  std::string name;

  #pragma db value_not_null inverse(employer)
  std::vector<weak_ptr<employee> > employees;
};
```

One-To-Many

```
CREATE TABLE employer (
  name VARCHAR (255) NOT NULL PRIMARY KEY);

CREATE TABLE employee (
  id BIGINT UNSIGNED NOT NULL PRIMARY KEY,
  employer VARCHAR (255) NOT NULL REFERENCES employer (name));
```

employer
name
Michael
Andrew

employee	
id	employer
0	Michael
1	Andrew
2	Michael
3	Michael

One-To-Many (alternative inverse)

```
#pragma db object
class employee
{
  ...

  #pragma db id
  unsigned long id;

  #pragma db not_null inverse(employees)
  shared_ptr<employer> employer;
};
```



```
#pragma db object
class employer
{
  ...

  #pragma db id
  std::string name;

  #pragma db value_not_null
  std::vector<weak_ptr<employee> > employees;
};
```

One-To-Many (alternative inverse)

```
CREATE TABLE employer (
  name VARCHAR (255) NOT NULL PRIMARY KEY);

CREATE TABLE employer_employees (
  object_id VARCHAR (255) NOT NULL,
  value BIGINT UNSIGNED NOT NULL REFERENCES employee (id));

CREATE TABLE employee (
  id BIGINT UNSIGNED NOT NULL PRIMARY KEY);
```

employer
name
Michael
Andrew

employer_employees	
object_id	value
Michael	0
Michael	2
Michael	3
Andrew	1

employee
id
0
1
2
3

Many-To-Many

```
#pragma db object
class employee
{
  ...

  #pragma db id
  unsigned long id;

  #pragma db value_not_null unordered
  std::vector<shared_ptr<project> > projects;
};
```

```
#pragma db object
class project
{
  ...

  #pragma db id
  std::string name;

  #pragma db value_not_null inverse(projects)
  std::vector<weak_ptr<employee> > employees;
};
```

Many-To-Many

```
CREATE TABLE project (
  name VARCHAR (255) NOT NULL PRIMARY KEY);

CREATE TABLE employee (
  id BIGINT UNSIGNED NOT NULL PRIMARY KEY);

CREATE TABLE employee_projects (
  object_id BIGINT UNSIGNED NOT NULL,
  value VARCHAR (255) NOT NULL REFERENCES project (name));
```

project
name
CodeCompass

employee
id
0

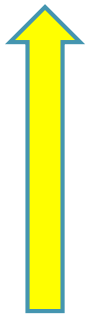
employee_projects	
object_id	value
0	CodeCompass

Be lazy

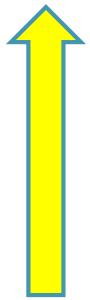
```
unsigned long id = ...
string name;

session s;
transaction t (db.begin ());

shared_ptr<employee> e(db.load<employee>(id));
name = e->employer->name;

t.commit ();
```

Be lazy

```
#pragma db object
class employer
{
  ...

  #pragma db value_not_null inverse(employer)
  std::vector<lazy_weak_ptr<employee> > employees;
};
```



```
#pragma db object
class employee
{
  ...

  #pragma db not_null
  lazy_shared_ptr<employer> employer;
};
```



Be lazy

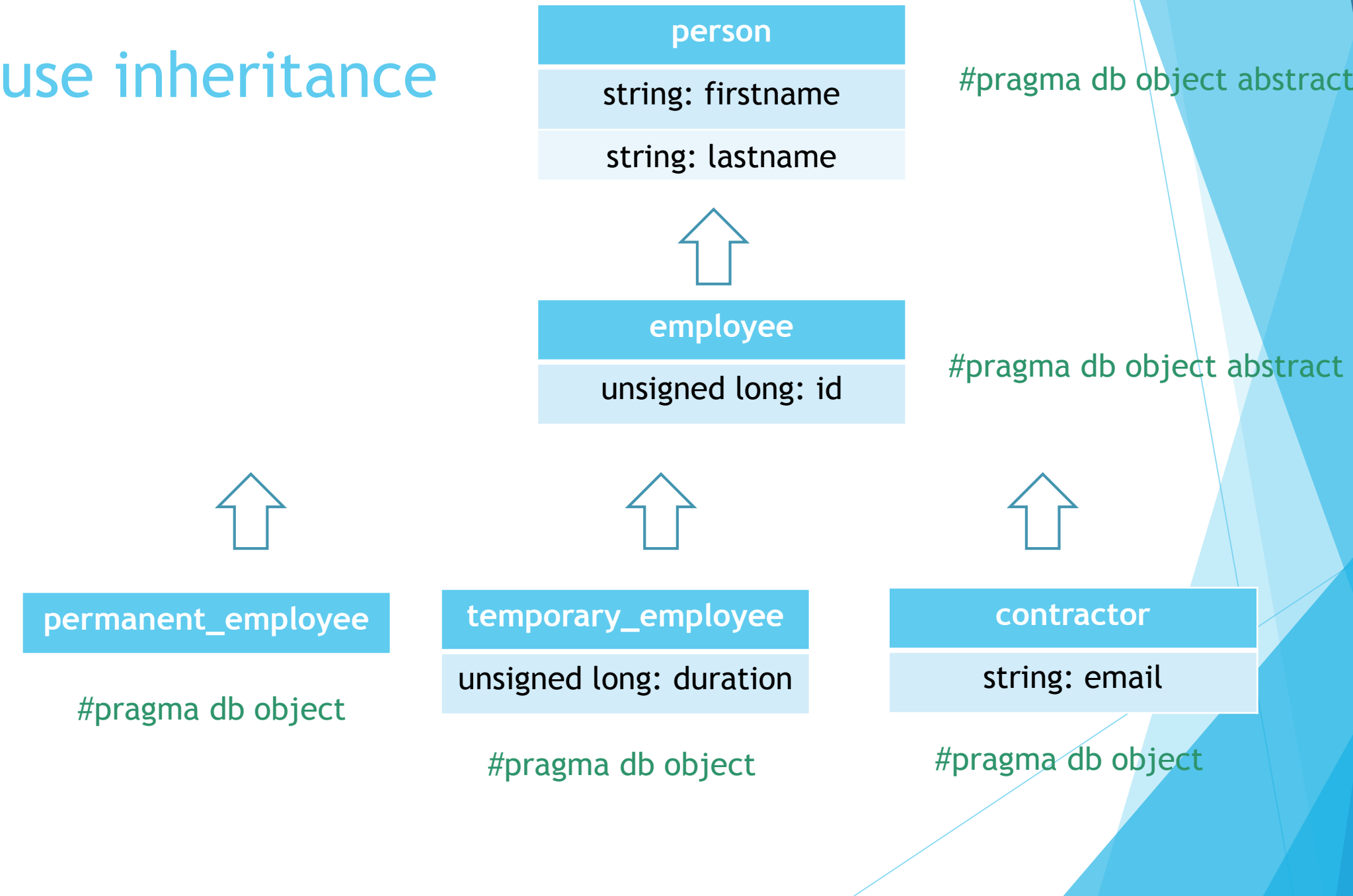
```
unsigned long id = ...
string name;

session s;
transaction t (db.begin ());

shared_ptr<employee> e(db.load<employee>(id));
e->employer.load();
name = e->employer->name;

t.commit ();
```

Sections!!

Inheritance

reuse inheritance and polymorphism

Reuse inheritance



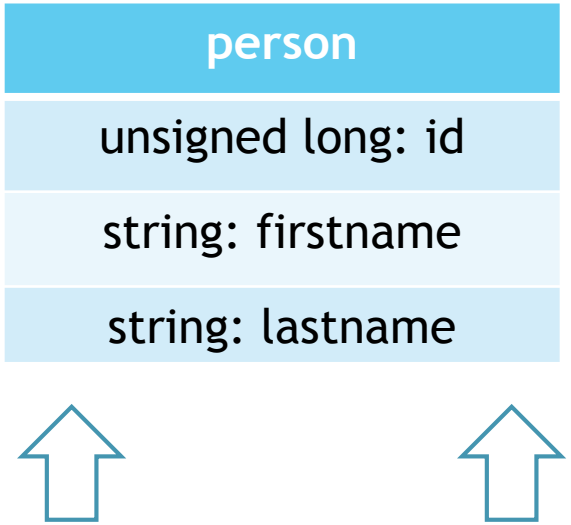
Reuse inheritance

```
CREATE TABLE permanent_employee (
  first TEXT NOT NULL,
  last TEXT NOT NULL,
  id BIGINT UNSIGNED NOT NULL PRIMARY KEY AUTO_INCREMENT);

CREATE TABLE temporary_employee (
  first TEXT NOT NULL,
  last TEXT NOT NULL,
  id BIGINT UNSIGNED NOT NULL PRIMARY KEY AUTO_INCREMENT,
  duration BIGINT UNSIGNED NOT NULL);

CREATE TABLE contractor (
  first TEXT NOT NULL,
  last TEXT NOT NULL,
  email VARCHAR (255) NOT NULL PRIMARY KEY);
```

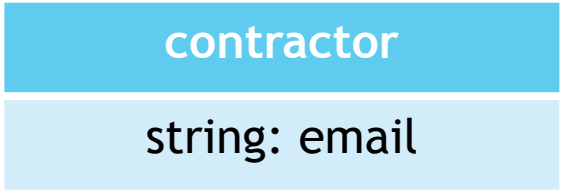
Polymorphic inheritance

- There are three approaches:
  - table-per-hierarchy
  - table-per-difference (only it is supported by ODB)
  - table-per-class

Polymorphic inheritance



#pragma db object polymorphic



#pragma db object

#pragma db object

Polymorphic inheritance (table-per-diff)

```
CREATE TABLE person (
  id BIGINT UNSIGNED NOT NULL PRIMARY KEY
AUTO_INCREMENT,
  typeid VARCHAR(255) NOT NULL,
  first TEXT NOT NULL,
  last TEXT NOT NULL);

CREATE TABLE employee (
  id BIGINT UNSIGNED NOT NULL PRIMARY KEY,
  temporary TINYINT(1) NOT NULL,
  CONSTRAINT employee_id_fk
    FOREIGN KEY (id)
    REFERENCES person (id)
    ON DELETE CASCADE);
```

```
CREATE TABLE contractor (
  id BIGINT UNSIGNED NOT NULL PRIMARY KEY,
  email TEXT NOT NULL,
  CONSTRAINT contractor_id_fk
    FOREIGN KEY (id)
    REFERENCES person (id)
    ON DELETE CASCADE);
```

Querying the Database

ODB Query language

```
typedef odb::query<person> query;

query q1 (query::first == "John" || query::age == 31);

query q2 (query::first.in ("John", "Jack", "Jane"));
query q2b (query::first.in_range (names.begin (), names.end ()));

query q3 (query::name.like ("J%"));

query q4 ((query::first == "John" || query::first == "Jane") && query::age < 31);

query q4 (query::employer->name == "Complex Systems Inc”)
```

Executing a Query

```
typedef odb::query<person> query;

// auto result = db.query<person> (q1);
auto result = db.query<person> (query::first == "John" || query::age == 31);

for (auto& p: result)
{
    cout << p.first << " " << p.last << ": " << p.age << endl;
}
```

Views

```
#pragma db view object(employee)
struct employee_name
{
  std::string first;
  std::string last;
};

#pragma db view object(employee)
struct employee_count
{
  #pragma db column("count(" + employee::id_ + ")")
  std::size_t count;
};
```

Views

```
#pragma db view object(employee) object(employer) \
  query ((?) + "GROUP BY" + employer::name_)
struct employer_age
{
  #pragma db column(employer::name_)
  std::string employer_name;
  #pragma db column("min(" + employee::age_ + ")")
  unsigned short min_age;
  #pragma db column("max(" + employee::age_ + ")")
  unsigned short max_age;
};
```

Execute queries on views

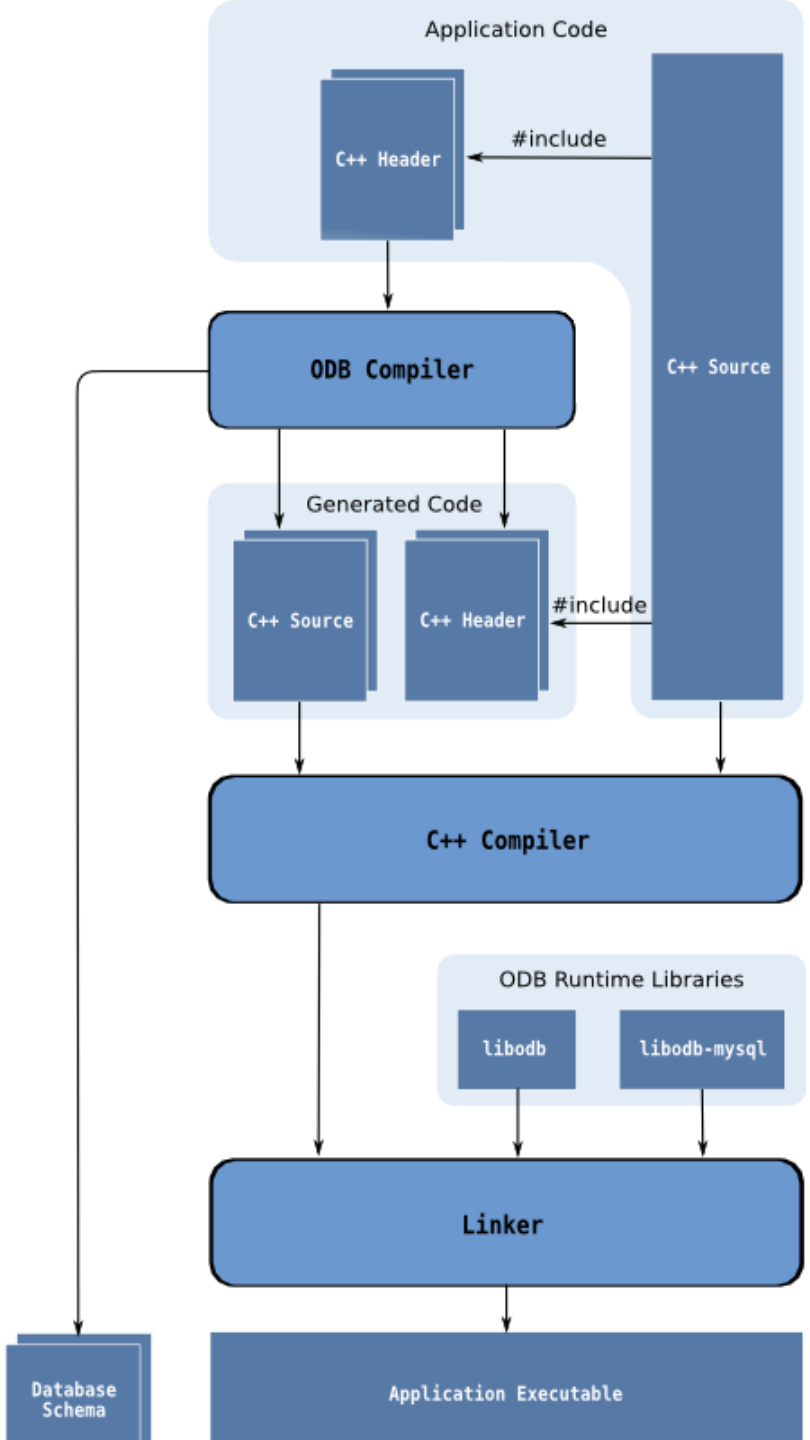
```
transaction t (db->begin ());

cout << "Employees under 31" << endl;
for (auto& ename: db->query<employee_name> (query::age < 31))
{
  cout << " " << ename.first << " " << ename.last << endl;
}

t.commit();
```

Database indices

```
#pragma db object
class object
{
  // An index that includes all of the p1's nested members.
  #pragma db index
  point p1;

  point p2;

  #pragma db index
  string name;
  // An index that includes only p2.x and p2.y.
  #pragma db index("p2_xy_i") members(p2.x, p2.y)
};
```

```
#pragma db value
struct point
{
  int x;
  int y;
  int z;
};
```

ODB overview



Experiences in CodeCompass

- All persistent classes are defined in CodeCompass/trunk/model
- Be lazy
- Use relations judiciously
- Use containers judiciously

sections?

More informations

- http://www.codesynthesis.com/products/odb/doc/manual.xhtml
- http://www.codesynthesis.com/products/odb/examples.xhtml

Thank you