Object-Oriented Databases
Commercial OODBMS: ObjectStore

- ObjectStore PSE Pro for C++
- Virtual Memory Architecture
- Managing Persistent Object Data

Progress ObjectStore

- Both Java and C++ environments supported
- ObjectStore Personal Storage Edition (PSE) Pro
  - lightweight object database
  - large, single-user databases
  - small memory footprint (~500kB)
  - multithreaded
  - embedded systems, mobile computing and desktop applications
- ObjectStore Enterprise
  - high-performance, distributed, multi-user database
  - distributed, persistent, transactional object caching
  - clustering, online backup, replication, high availability
- Migration of applications to from PSE to Enterprise is easy

ObjectStore Architecture

- Virtual memory mapping architecture extends operating system virtual memory architecture to provide persistence
  - logical versus physical address
  - physical memory and secondary storage
  - page faulting
  - address translation

  assumed knowledge from operating system courses
- Characteristics of the ObjectStore architecture
  - virtual
  - shared
  - distributed
  - heterogeneous
  - persistent
  - transactional

Virtual Memory Mapping Architecture

- Logical versus physical address
  - data is uniquely referenced within the database using a 4-part key

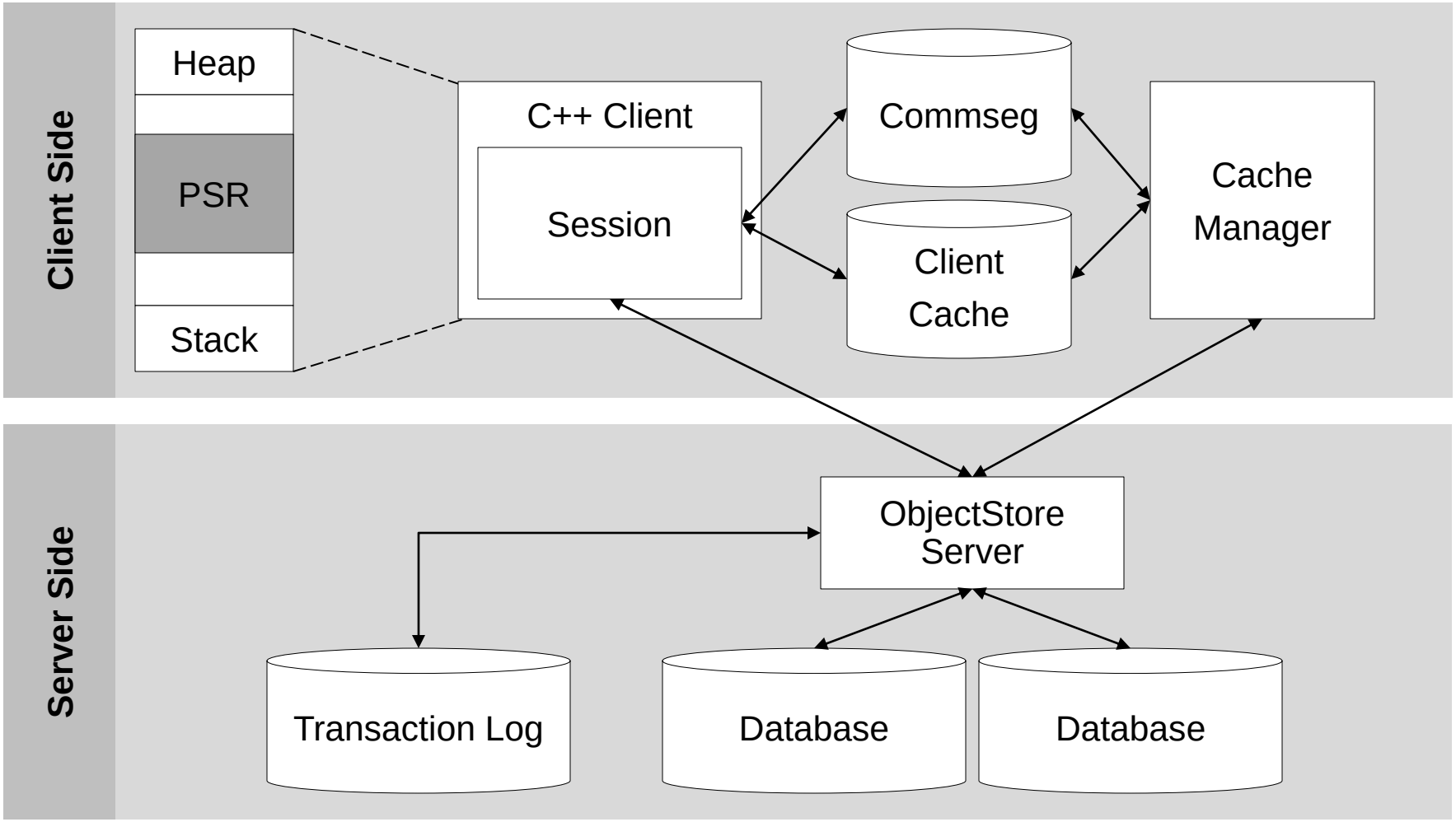
database	segment	cluster	offset in cluster

  - yields a theoretical address space of $0...2^{128}$
  - data is mapped from this 128 bit range into a reserved area of the database client application's virtual memory (<< 2^{128} address space)
  - reserved area is called *persistent storage region* (PSR)
- Physical memory and secondary storage
  - all data accessed by client application must reside in PSR
  - cache serves as secondary storage for operating system (instead of swap file) for persistent data mapped to logical address space
  - cache holds recently accessed data even across transactions

Virtual Memory Mapping Architecture

- Page faulting
  - ObjectStore maps data into application when a fault interrupt occurs
  - data is paged into memory from cache if not in PSR or fetched from the server if not in cache
  - demand paging is primary means by which data gets from database into cache and then into application
- Address translation
  - address translation is done when data is fetched into cache
  - retranslation can occur when PSR gets nearly full
  - updated pages are translated back to logical addressing schema before being written back to database
  - trade-off: ability to use direct software pointers yields performance and modelling advantages, but translating pointers and pre-reserving address space has scalability implications

Architecture Overview



Server Side Components

- Server
  - serves out pages and enforces ACID semantics using “page permits”
  - co-operates with other servers in two-phase commits
  - automatic recovery mechanism when restarted
- Database
  - managed by one server (but server can manage multiple databases)
  - binary files storing pages of memory containing C++ objects
  - normally deployed in the file system on server-local discs
- Transaction Log
  - each server owns transaction log to which updated pages are written
  - pages only propagated to the database when transaction commits
  - used for automatic recovery, faster commits and MVCC mechanism

Client-Side Components

- Client
  - C++ program linked with the ObjectStore libraries
  - interacts with the database and manages objects
  - pages automatically fetched from database as needed and cached
- Cache
  - one cache memory mapped file per client process
  - has a fixed size that cannot change once the client has started
  - all pages fetched from the database by this client are held in cache
  - pages can be retained in the cache between transactions
- Commseg
  - one commseg memory mapped file per client process
  - contains meta-information about every page in the cache
  - stores *permit* and a *lock* for every page in the cache
  - permits can be retained between transactions

Client-Side Components

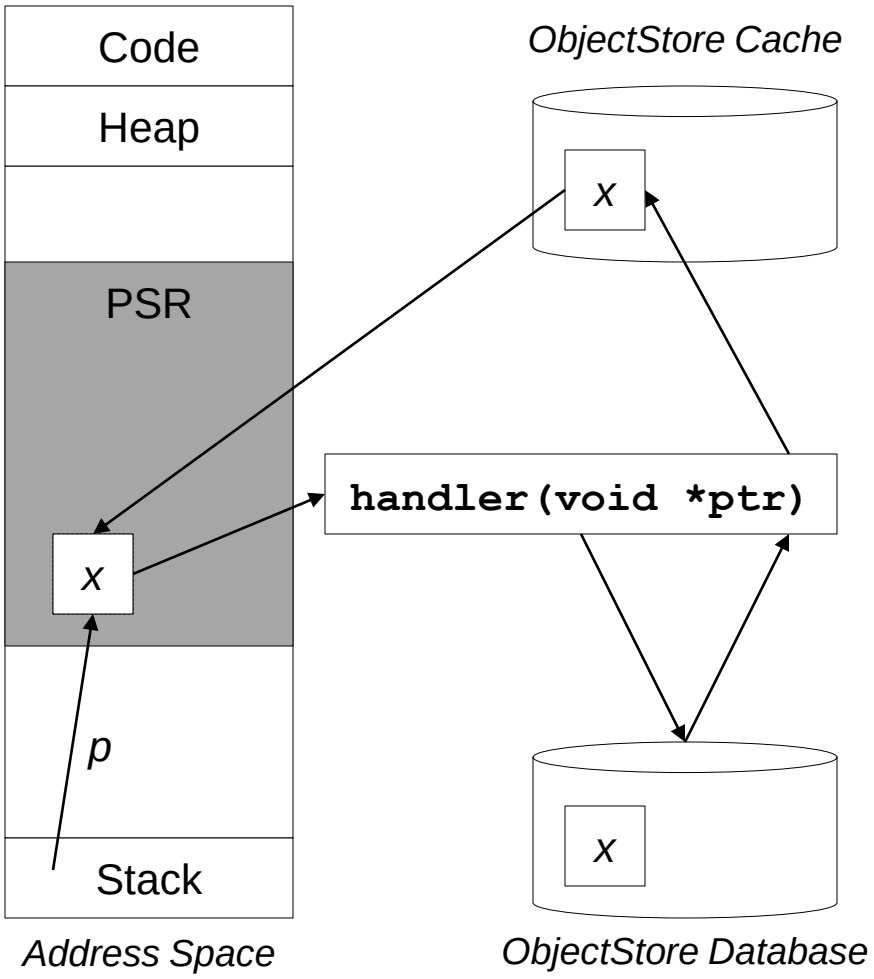
- Cache Manager
  - one cache manager process per client machine is shared by all clients on that machine
  - handles permit revokes
  - reads/writes to cache and commseg files
  - not directly involved in page fetch in any way
- Persistent Storage Region
  - is a reserved area of the virtual address space of the C++ program
  - address of persistent objects used by client mapped into PSR
  - value of pointers to persistent objects will be in the range of the PSR
  - at the end of every transaction the PSR is cleared and can be reused for the next transaction

Fetching and Mapping Pages

- Client automatically fetches and maps pages
  - pages are fetched “lazily” as needed
  - pages are held in the client cache
- Pointer swizzling used to translate logical addresses on fetched page into physical addresses within PSR
  - C++ pointers to already fetched objects
  - C++ pointers to ranges pre-reserved for yet-to-be fetched objects
- Server permits and client locks acquired automatically to ensure transaction consistency
- Existing page swapped out if not enough room in cache to hold new page
  - updated pages are sent to the server
  - read-only pages are dropped from cache as copy exists in database

Fetching and Mapping Pages

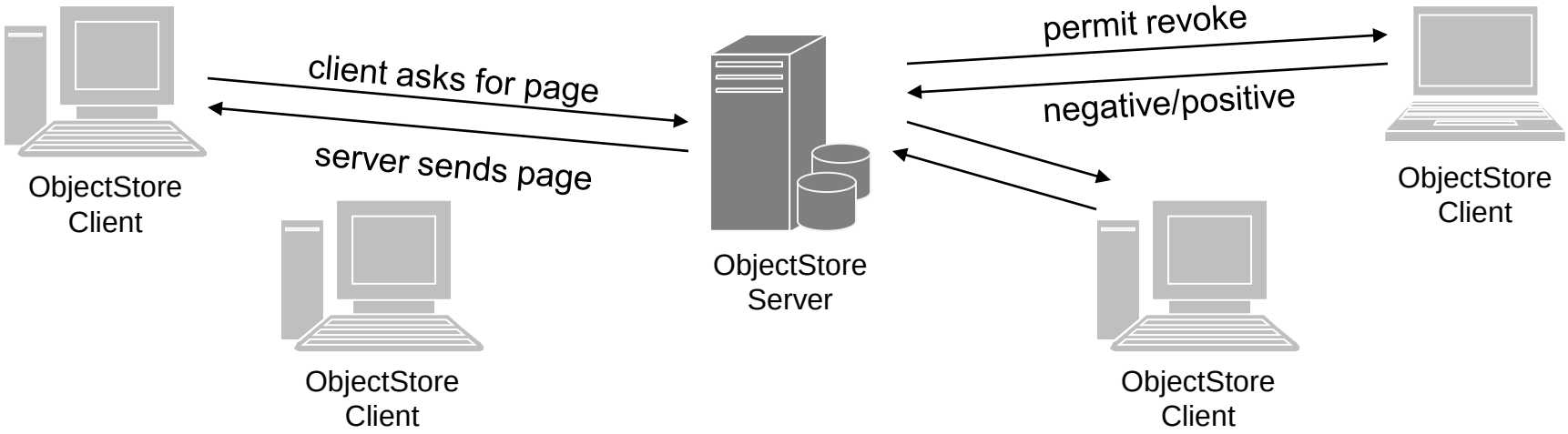
- ObjectStore installs SIGSEGV (segment violation) handler
- Program obtains pointer *p* to object on page *x*
- Dereferencing *p* causes the SIGSEGV handler to be called
- Virtual mapping table is consulted and page fetched from server and stored in the cache
- Page *x* is mapped to the address space and execution continues



Cache-Forward Architecture

- Key to ability of ObjectStore to provide high performance
  - data is cached across transaction boundaries
  - number of times locks must be acquired is reduced
  - cached data is kept in a globally consistent state
- ObjectStore maintains two types of locks on pages
  - **transaction locks** represent the state of a page during transaction
  - **ownership permits** represent the state of a page in the cache
- Permits are tracked by server and locks are taken by client
  - server serves permits on pages that are sent to the client
  - a client can then lock pages according to the given permit

Shared Virtual Memory



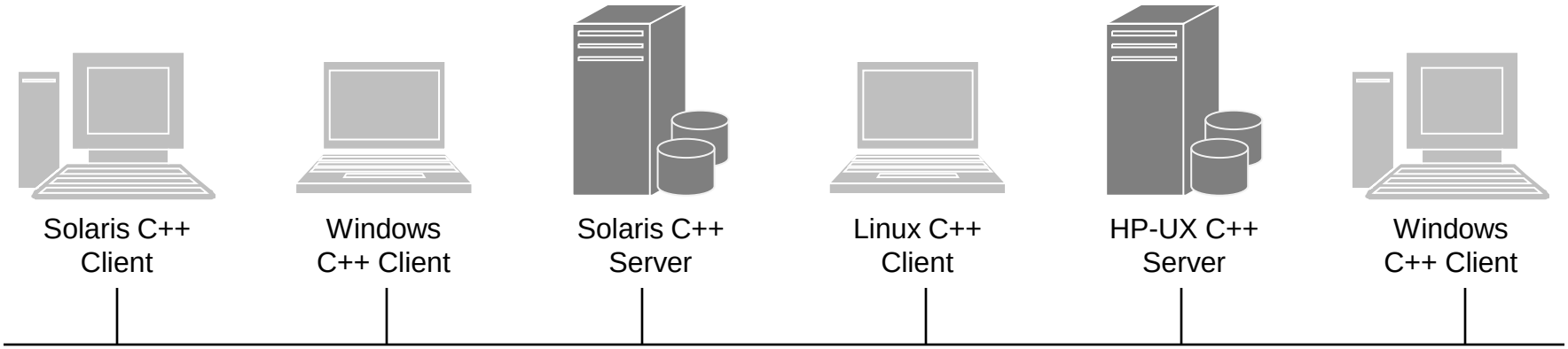
- ObjectStore uses a lazy call-back mechanism for permits
- Server maintains a table of permits assignments to clients
- When a client requests a page from the server
  - server checks for other clients with permit for page and permit types
  - server issues call-back if one or more clients have conflicting permits

Page Permits and Locks

- Read permit
  - client can lock page for reading without consulting the server
  - many clients can hold a read permit for a page simultaneously
- Write permits
  - client can lock page for reading or writing without asking the server
  - only one client can hold a write permit for a page at any given time
- Cache manager inspects permit and lock status for call-back
  - ✓ POSITIVE
  - ✘ NEGATIVE (but permit is flagged to be revoked at transaction end)

Permit	Lock	Response
read	read	✘ *server only calls back permit if other client needs to write*
read	no lock	✓
write	read	✓ *permit for page downgraded to read*
write	write	✘
write	no lock	✓

Distribution and Heterogeneity



- Clients can access objects in different remote databases in the same transaction
- Clients and servers can run on different platforms
  - physical object layout transformed automatically by client runtime when page mapped into cache
  - database records which platform wrote to each page last

Persistence

- ObjectStore uses persistence by instantiation in C++
- Overloaded persistent **`new`** operator takes three arguments
  - allocation of the new object
  - type spec of the new object
  - optionally, how many objects are to be allocated
- Several options for object allocation
  - transiently on the heap
  - database
  - segment
  - cluster
  - next to another object

Note: *given the virtual memory architecture it is helpful to co-locate objects which are used together to achieve high performance designs and implementations*

- Persistence is orthogonal to the type of an object and one codebase can be used for transient and persistent objects

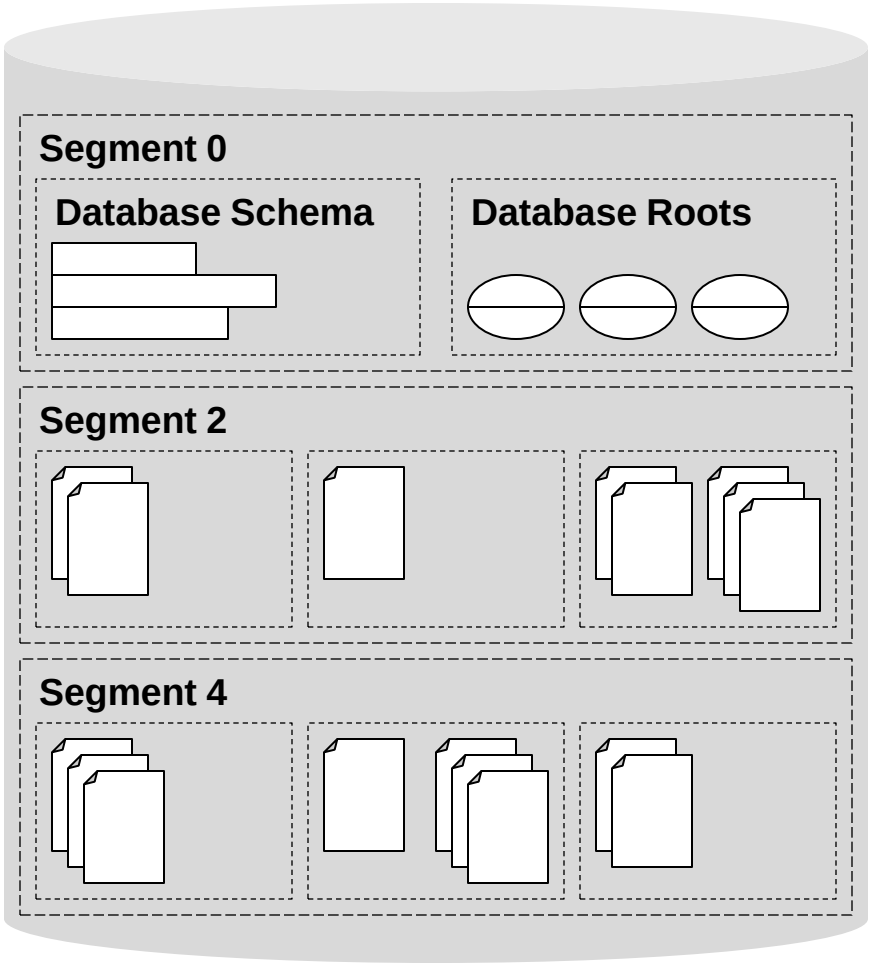
Transactions

- Support for basic ACID properties of transactional systems
- Atomicity
  - after commit it is guaranteed that data was written and is recoverable
  - after abort all changes are undone
- Consistency
  - it is impossible to apply or lose updates while data is being written
- Isolation
  - serialisability (CPSR) is guaranteed by two-phase locking (2PL)
  - Multi-View Concurrency Control (MVCC) provides serialisability for read-only transactions using snapshots instead of locks
- Durability
  - changes are written to the transaction log first
  - background process propagates changes to the database

Transaction Types

- Read or Write
  - Read transaction throws an exception if a page write lock is requested
- Local or Global
  - Local only allows the initiating thread to execute
  - Global allows all threads in a session to share the transaction
- Lexical or Dynamic
  - Lexical transactions automatically retry on deadlock
  - Lexical must start and end in same code block
  - Lexical transactions are always thread-local
  - Dynamic transactions are the lower level **`os_transaction`** class
  - Dynamic transactions are better suited to multi-threaded applications

Database Layout



- Memory **pages** held in hierarchy of **clusters** within **segments**
- Segments
  - define logical partitioning of objects
  - Segment 0: schema segment contains database schema and database roots
  - Segment 2: default segment
  - Segment 4: first user-created segment
  - maximally 2^{32} segments per database
- Clusters
  - group closely related objects
  - each segment has a default cluster 0, other clusters created by user
  - maximally 2^{31} clusters per segment

Developing Applications

- Programmer uses the ObjectStore libraries
  - **`objectstore`** ObjectStore runtime
  - **`os_database`** database management functionality
  - **`os_transaction`** transaction handles and functionality
  - **`os_typespec`** functionality to determine type specification
  - **`os_database_root`** creation, retrieval and removal of roots
  - **`os_segment`** segment access and management
  - **`os_cluster`** cluster access and management
- Development process
  - writing of persistent classes, schema file and application logic
  - compilation of schema file with **`pssg`** compiler
  - compilation of classes with C++ compiler
  - linking of object code

Managing Databases

```
#include <os_pse/ostore.hh>

int main(int argc, char **argv, char **envp)
{
   objectstore::initalize();
   os_database *db = os_database::create("publications.db", 0664, 1);

   ...
   db->save();
   db->close();
   db->destroy();
   objectstore::shutdown();
}
```

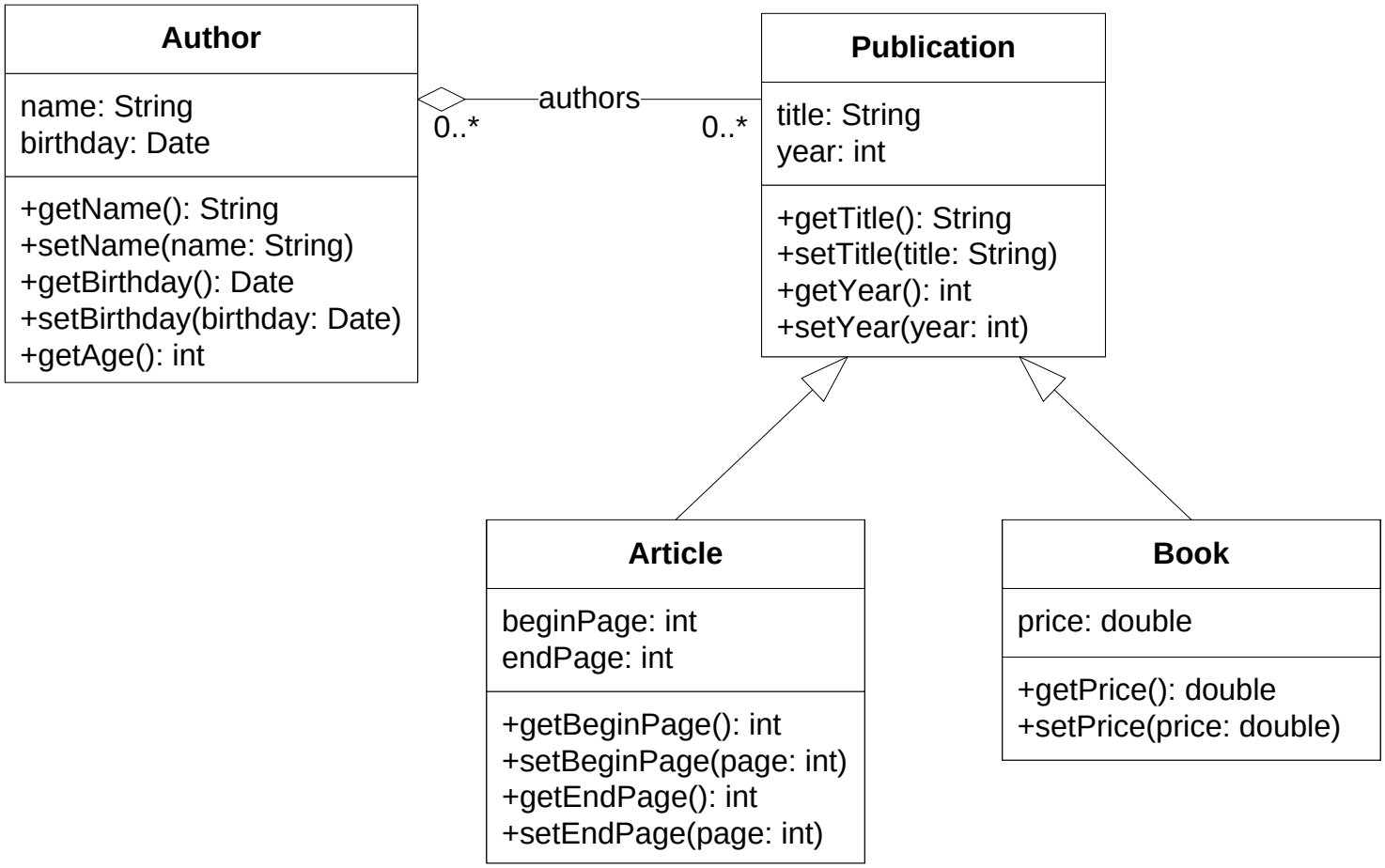
- Database management is provided by **`os_database`**
  - **`create()`** creates a new database
  - **`open()`** opens an existing database
  - **`save()`** saves the database and makes changes permanent
  - **`close()`** closes an open database, but does not save state
  - **`destroy()`** deletes a database

Transactions

```
objectstore::initialize();
os_transaction::initialize();
os_database *db = os_database::open("publications.db", 0, 1);

OS_BEGIN_TXN(txn0, 0, os_transaction::update)
{

   ...

   os_transaction *txn = os_transaction::get_current();
   txn->abort();
}
OS_END_TXN(txn0)
```

- Transaction functionality is provided by **`os_transaction`**
  - all interactions with the database must be in a transaction
  - transactions can be nested arbitrarily
- Ways of defining and working with transactions
  - directly using class **`os_transaction`** (dynamic)
  - using macros provided with the ObjectStore libraries (lexical)

Running Example



Class Author

```
class Author
{
private:
   const char *_name;
   time_t _birthdate;
public:
   // Constructor and destructor
   Author(const char *name);
   ~Author();
   // Getters and setters
   const char* getName() const;
   void setName(const char *name);
   const struct tm* getBirthdate() const;
   void setBirthdate(int day,
                     int month, int year);
   // Derived methods
   int getAge() const;
};
```

```
#include "Author.h"

Author::Author(const char *name)
{
   this->setName(name);
   _birthdate = 0;
}

Author::~Author(void)
{
   if (_name) {
      delete [] _name;
      _name = 0;
   }
}

const char* Author::getName()
{
   return _name;
}

...
```

Creating Persistent Objects

- Objects in the database are created with the overloaded persistent **`new`** operator
  - creating a single persistent object

```
os_database *db = os_database::open("publications.db", 0, 1);
Author *scheel = new(db, os_ts<Author>::get()) Author("Matthias Geel");
db->close();
```

  - creating a persistent array of objects

```
void Author::setName(const char* name)
{
   delete [] _name;
   _name = 0;
   if (name) {
      int length = static_cast<int>(strlen(name)) + 1;
      _name = new(os_cluster::of(this),          // allocate in the same cluster
                  os_typespec::get_char(),       // get char type spec
                  length) char[length];          // create an array of size length
     strcpy_s(_name, length, name);
     _name[length] = 0;
   }
}
```

Updating and Deleting Persistent Objects

```
os_database *db = os_database::open("publications.db", 0, 1);

Author *moira = new(db, os_ts<Author>::get()) Author("Moira Norrie");
moira->setName("Moira C. Norrie");

db->save();      // page with updated version of object is sent to server

delete moira;
moira = 0;

db->save();      // page without the object is sent to server
```

- Changes to persistent objects are propagated to database automatically when pages are sent back to server
  - client application updates memory-mapped version of persistent objects using standard C++
  - persistent objects are deleted by deleting the memory-mapped version of object using standard C++
- Fully transparent to the application developer

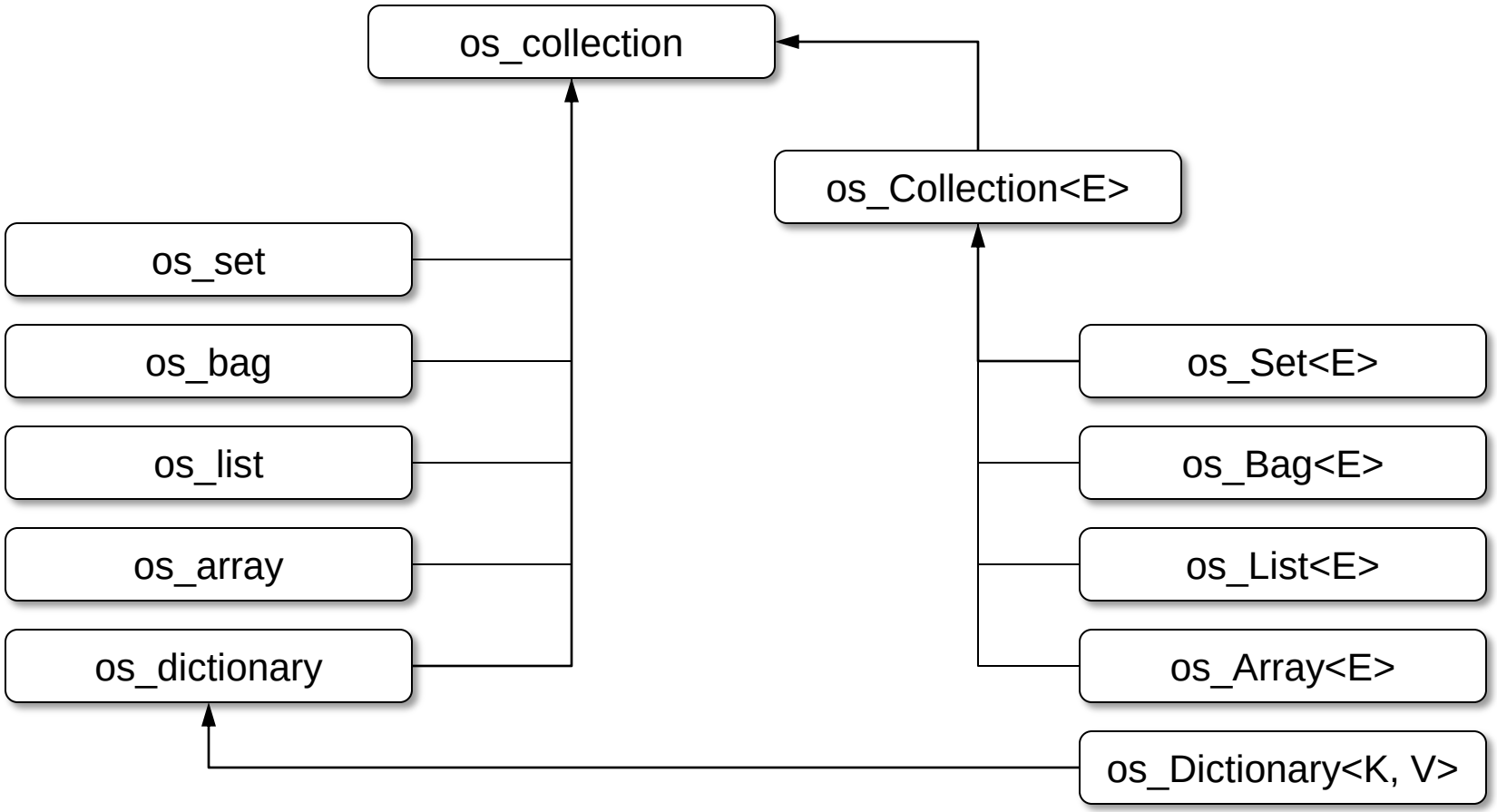
Collections and Relationships

```
// Forward declaration
class Publication;

class Author
{
private:
   ...
   friend class Publication;
   os_Set<Publication*>* _authors;
public:
   ...
   void addPublication(Publication *p);
   void removePublication(Publication *p);
};
```

```
// Forward declaration
class Author;

class Publication
{
private:
   ...
   friend class Author;
   os_List<Author*>* _authoredBy;
public:
   ...
   void addAuthor(Author *a);
   void removeAuthor(Author *a);
};
```

- Relationships between classes modelled as collections
- ObjectStore collection facility
  - a library of non-templated and templated collection types
  - traversal, manipulation, and retrieval functionality
  - represented by class **`os_collection`**

Collection Hierarchy



Collections Example

- Creating a collection

```
Author::Author(const char *name)
{
   ...
   _authors = new(os_cluster::of(this),
                  os_Set<Publication*>::get_os_typespec()) os_Set<Publication*>();
}
```

- Accessing and manipulating a collection

```
void Author::addPublication(const Publication *p)
{
   _authors->insert((Publication*) p);
   os_List<Author*> *authoredBy = p->_authoredBy;
   authoredBy->insert(this);
}
```

- Deleting a collection

```
Author::~Author(void)
{
   ...
   delete _authors;
   _authors = 0;
}
```

Cursors over Collections

```
const os_Set<Publication*>* Author::getPublications() const
{
   os_Set<Publication*> *result = new(
         os_database::get_transient_database(),
         os_Set<Publication*>::get_os_typespec()) os_Set<Publication*>();
   os_Cursor<Publication*> c(*_authors);
   for (Publication *publication = c.first(); c.more(); publication = c.next()) {
      result->insert(publication);
   }
   return result;
}
```

- Cursors are used to navigate and manipulate collections
  - represented by class **`os_Cursor`**
  - **`first()`** positions the cursor at the first element
  - **`next()`** moves the cursor to the next element
  - **`more()`** returns true if the cursor points to an element
- Cursor can be reused by rebinding it to another collection

Queries over Collections

```
// Find all authors younger than 30 with more than 10 publications

char* query = "this->getAge() < 30 && this->getPublicationCount() > 10";
os_Set<Author*> &result = _authors->query("*Author", query, _db);

os_Cursor<Author*> c(result);
for (Author *author = c.first(); c.more(); author = c.next()) {
   cout << author->getName() << endl;
}
```

- Queries are evaluated over collections by specifying the element type, query string and schema database
  - query string indicates the selection criterion, either specified in C++ or as a pattern matching expression
  - support for function calls in query strings restricted to basic types
  - support for nested queries

Database Roots

```
Author* Database::retrieveAuthor(const char *name)
{
   os_Dictionary<char*, Author*> *authors = 0;
   os_database_root *root = _db->find_root("authors");
   if (!root) {
      root = _db->create_root("authors");
      authors = new(_db, os_Dictionary<char*, Author*>::get_os_typespec())
                                                    os_Dictionary<char*, Author*>();
      root->set_value(authors);
   }
   authors = (os_Dictionary<char*, Author*>*) root->get_value();
   return authors->pick((char*) name);
}
```

- Database roots are persistent objects which have been labelled with a well-known name
- Represented by class **`os_database_root`**
  - root name held as a **`char*`**
  - pointer to the object of interest held as a **`void*`**

Literature

- ObjectStore
  - http://www.progress.com/objectstore/

Next Week
Commercial OODBMS: Objectivity/DB

- Objectivity/DB for .NET
- Logical Storage Model: Federated Databases
- Language Integrated Queries (LINQ)