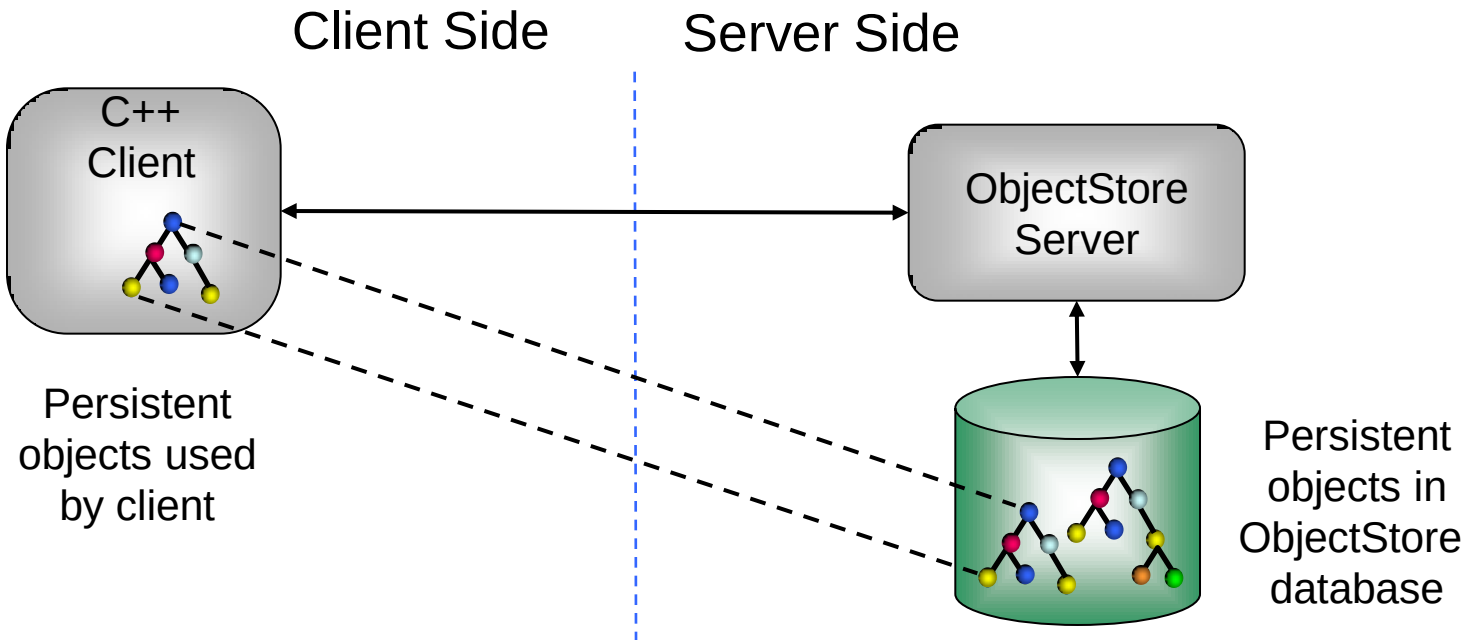
Introduction to ObjectStore

Adrian Marriott
Independent Consultant

Agenda

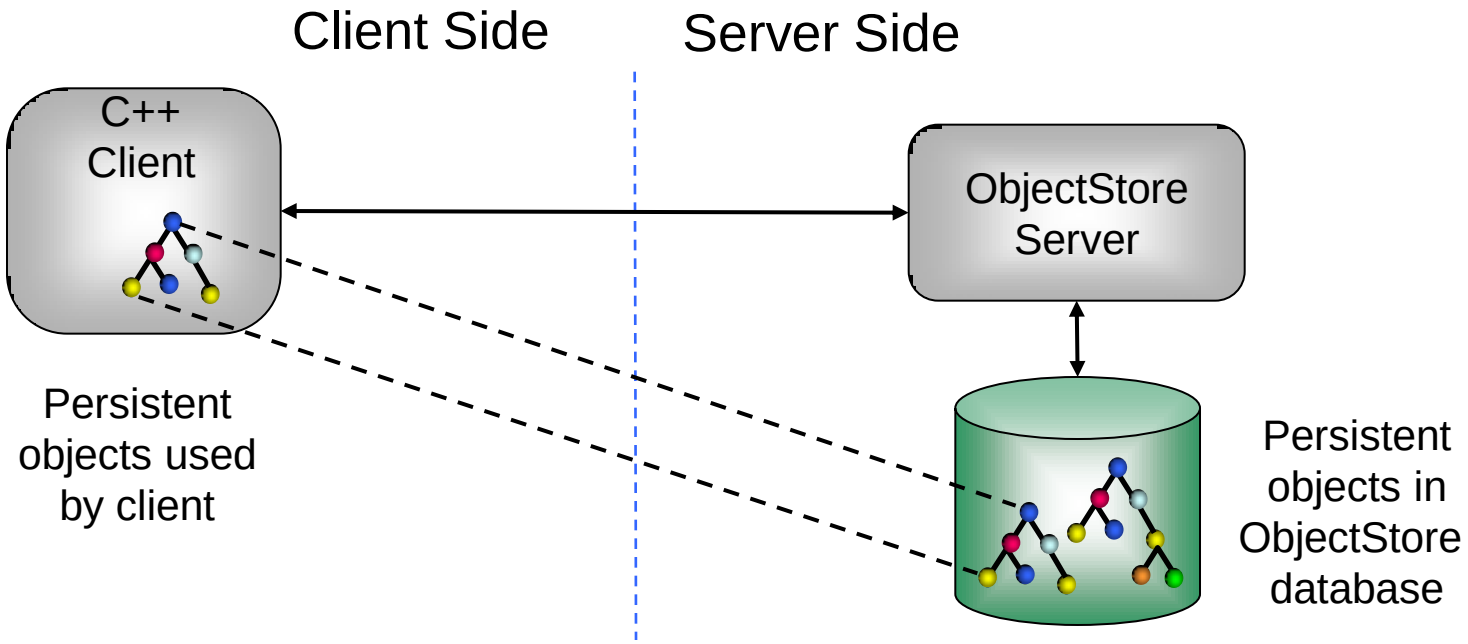
- What is ObjectStore?
- ObjectStore Components
- Fetching and Mapping a Page
- ObjectStore Programming
- Opening and Closing a database
- Using Transactions
- Creating and Deleting Persistent Objects
- Finding Initial Persistent Objects
- Segments and Clusters
- Writing and Using a Persistent Class
- Conclusion

What is ObjectStore?



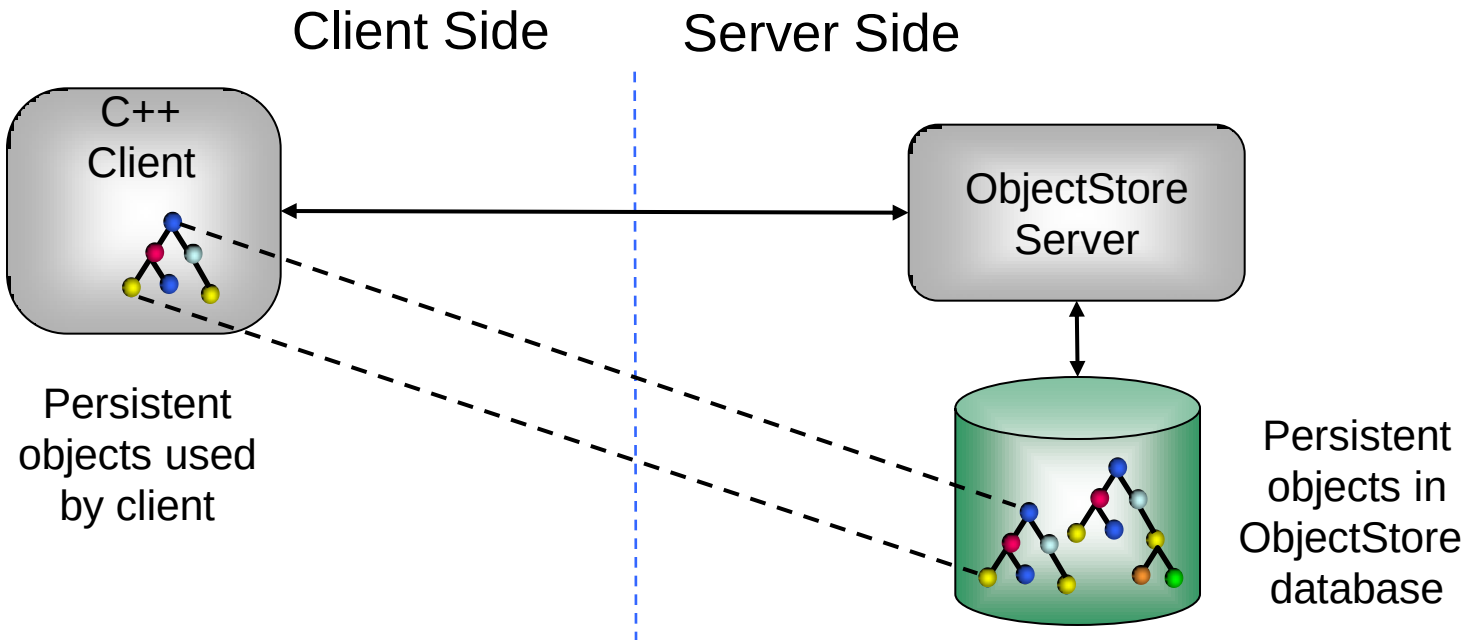
- ObjectStore is a ‘pure’ object database
- Stores C++ objects in the same format as they are used in memory
- Accessed and updated using the same C++ syntax as heap-allocated objects

What is ObjectStore?



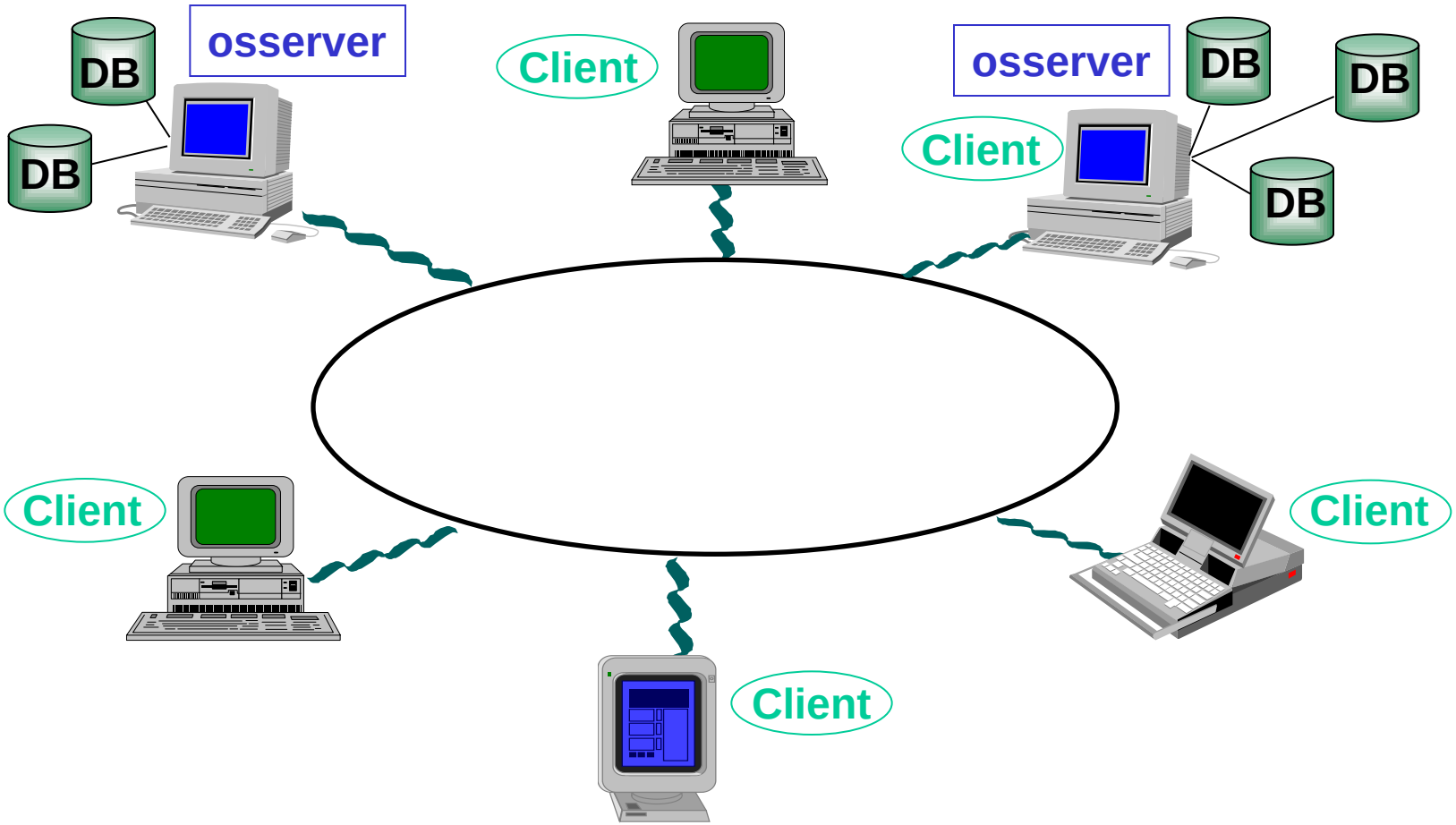
- Database contains memory pages
- Client starts a transaction and uses persistent objects exactly ‘as if’ they were heap allocated
- Pages fetched automatically on an ‘as needed’ basis as the client navigates persistent pointers

What is ObjectStore?

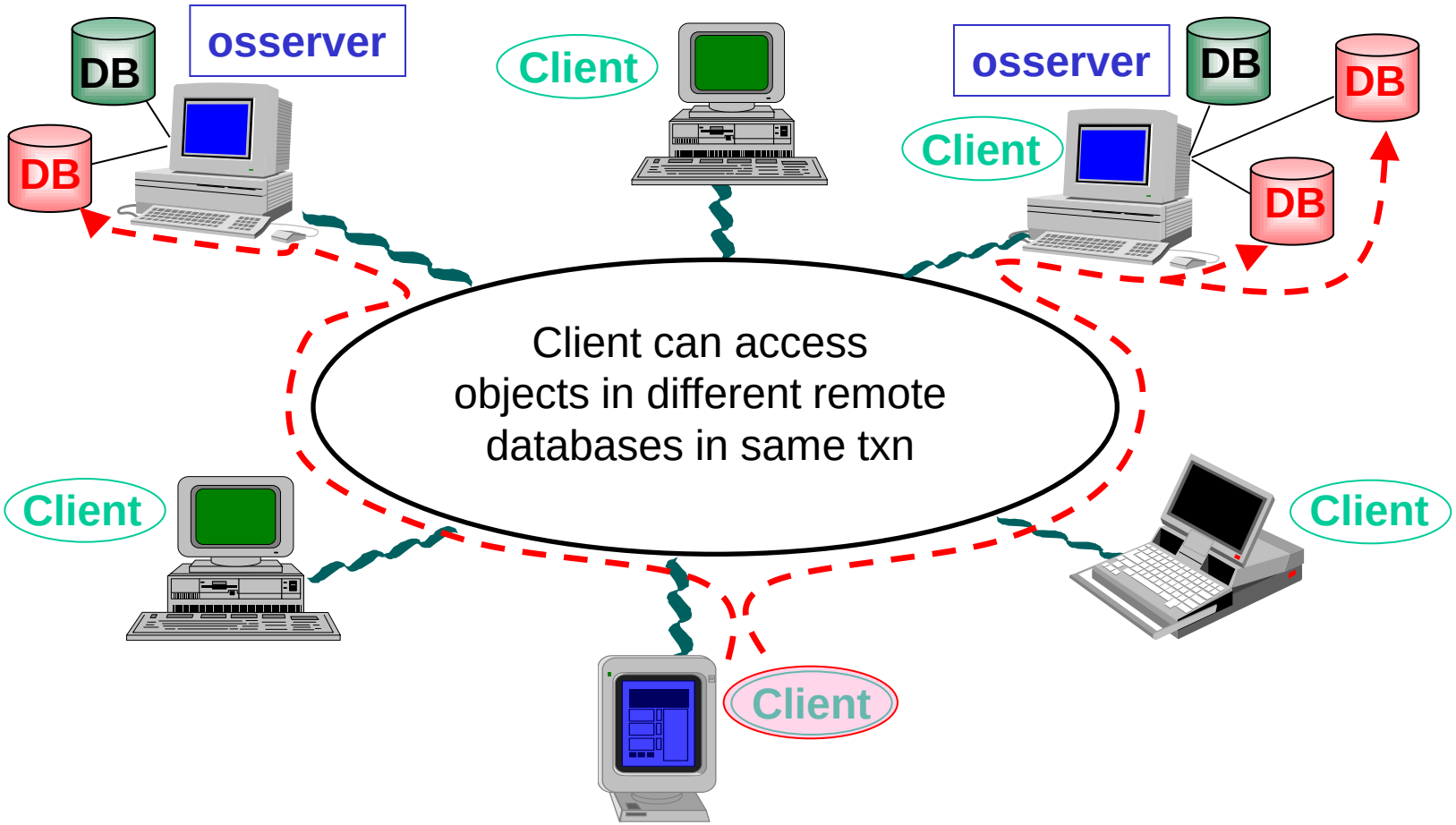


- Server coordinates sharing of pages between multiple clients
- Page read/write permits and locks are managed automatically to ensure transaction consistency

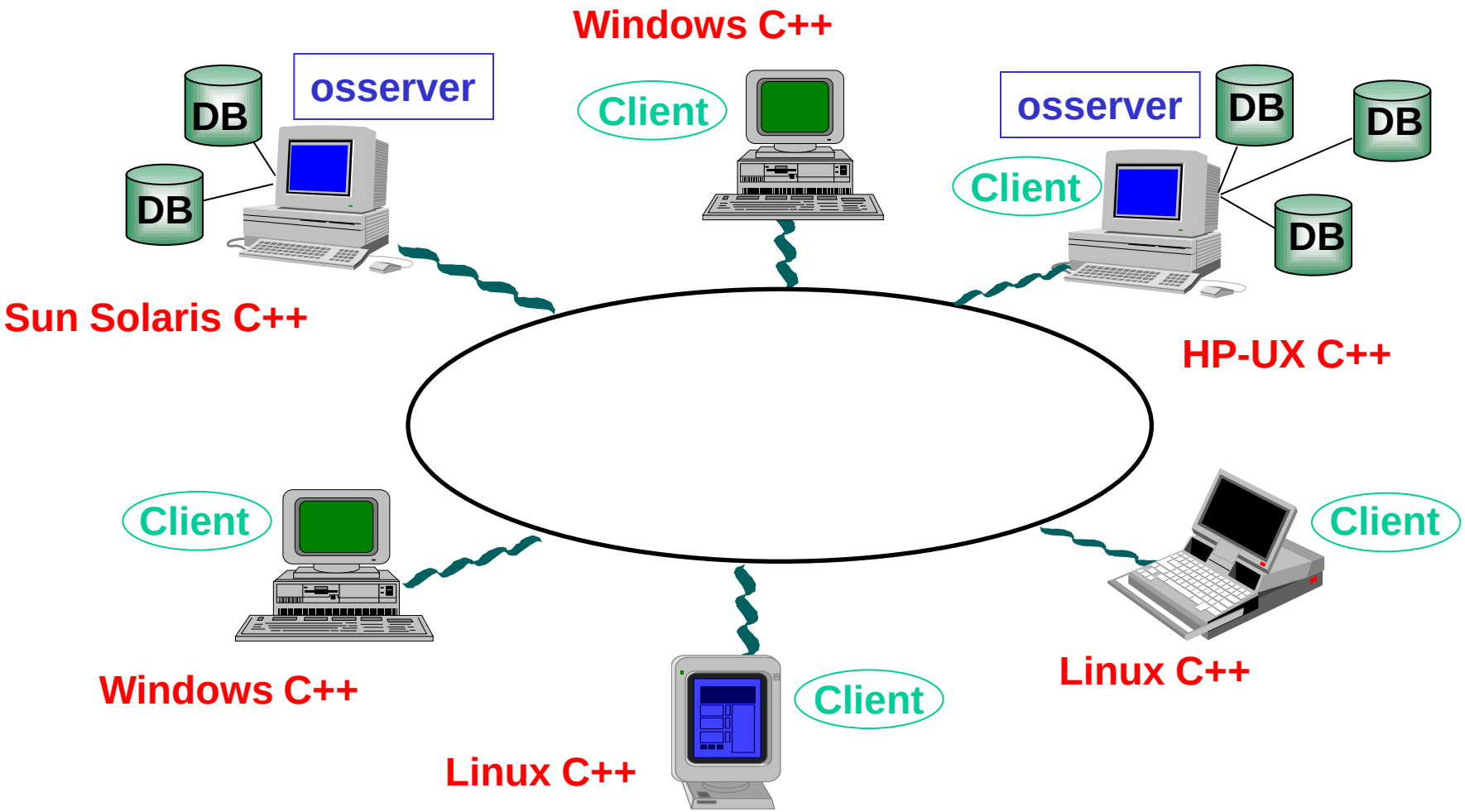
ObjectStore is Distributed



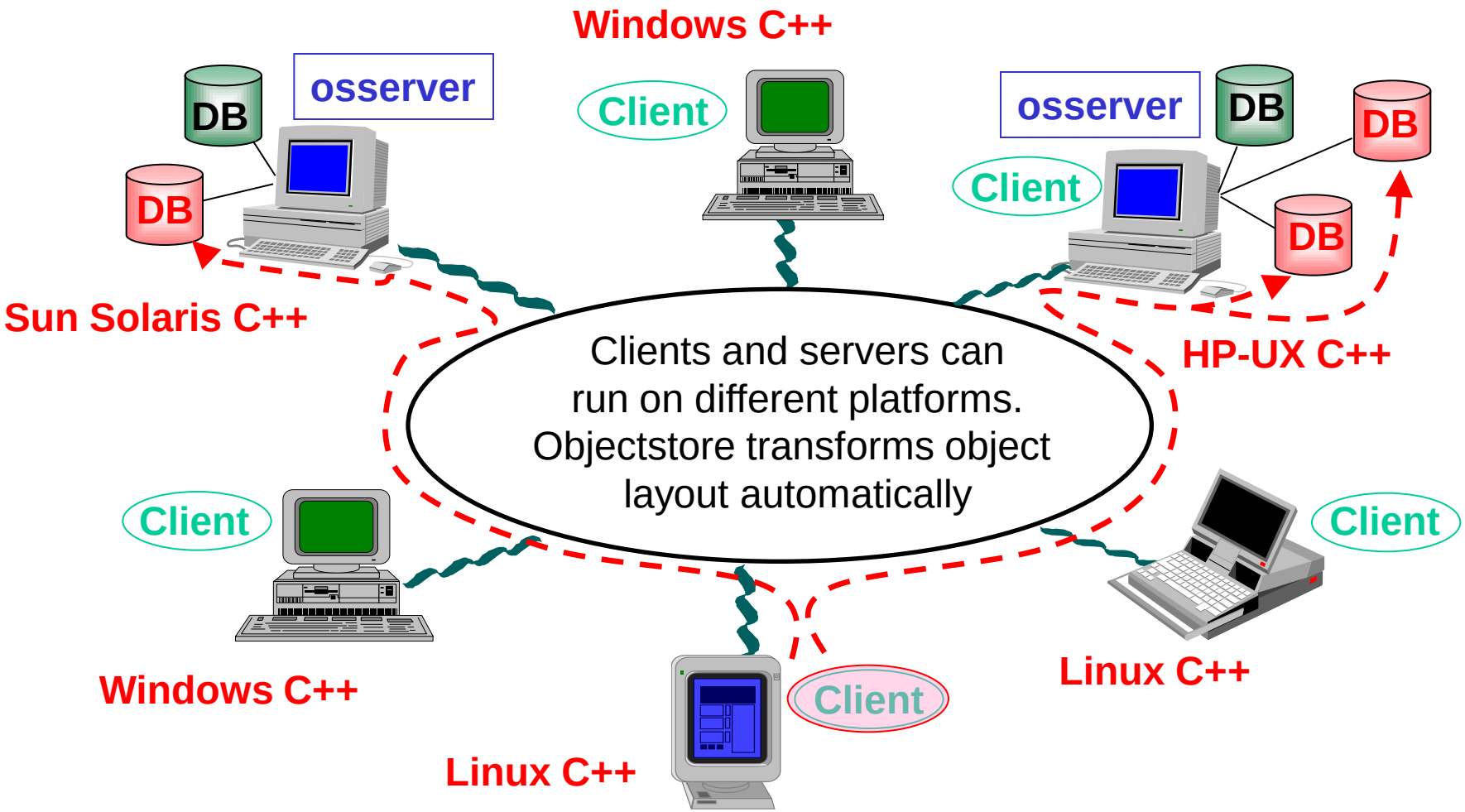
ObjectStore is Distributed



ObjectStore is Heterogeneous



ObjectStore is Heterogeneous



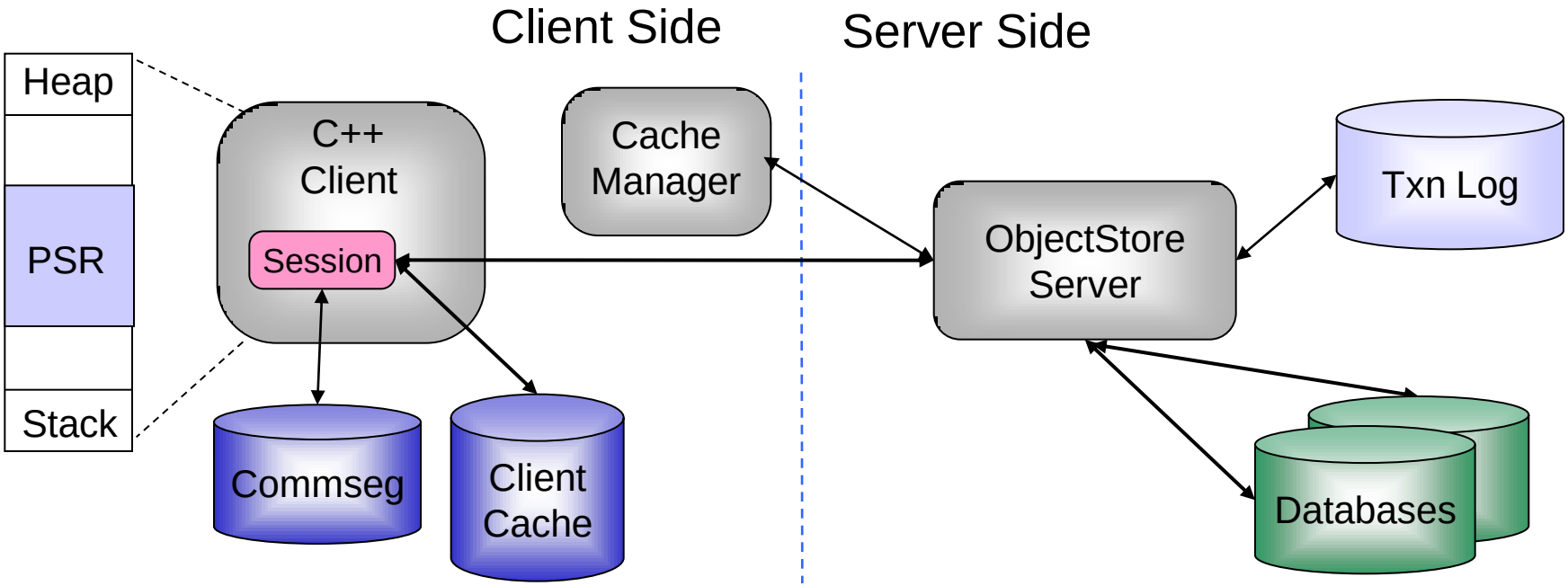
Agenda

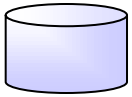
- What is ObjectStore?
- ObjectStore Components
- Fetching and Mapping a Page
- ObjectStore Programming
- Opening and Closing a database
- Using Transactions
- Creating and Deleting Persistent Objects
- Finding Initial Persistent Objects
- Segments and Clusters
- Writing and Using a Persistent Class
- Conclusion

ObjectStore Components

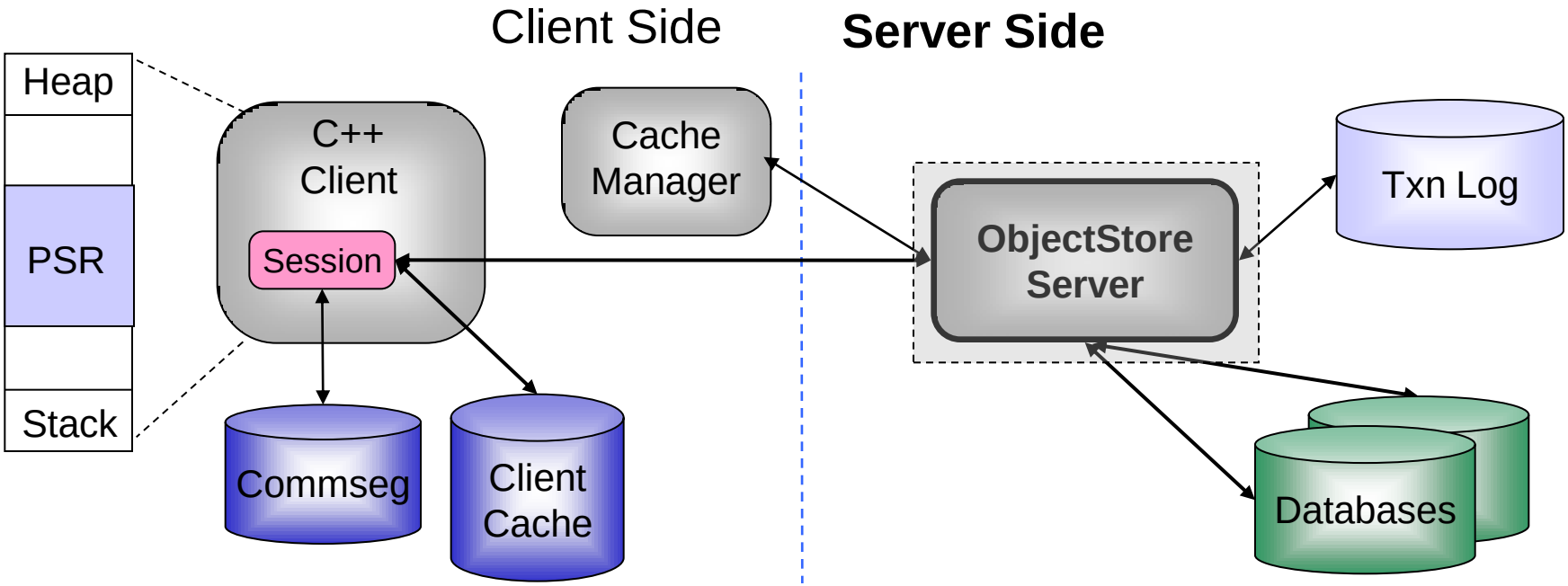
- Server process
- Databases
- Transaction Log file
- Client C++ program
- Cache
- Commseg
- Cache Manager process
- Persistent Storage region (PSR)

ObjectStore Components



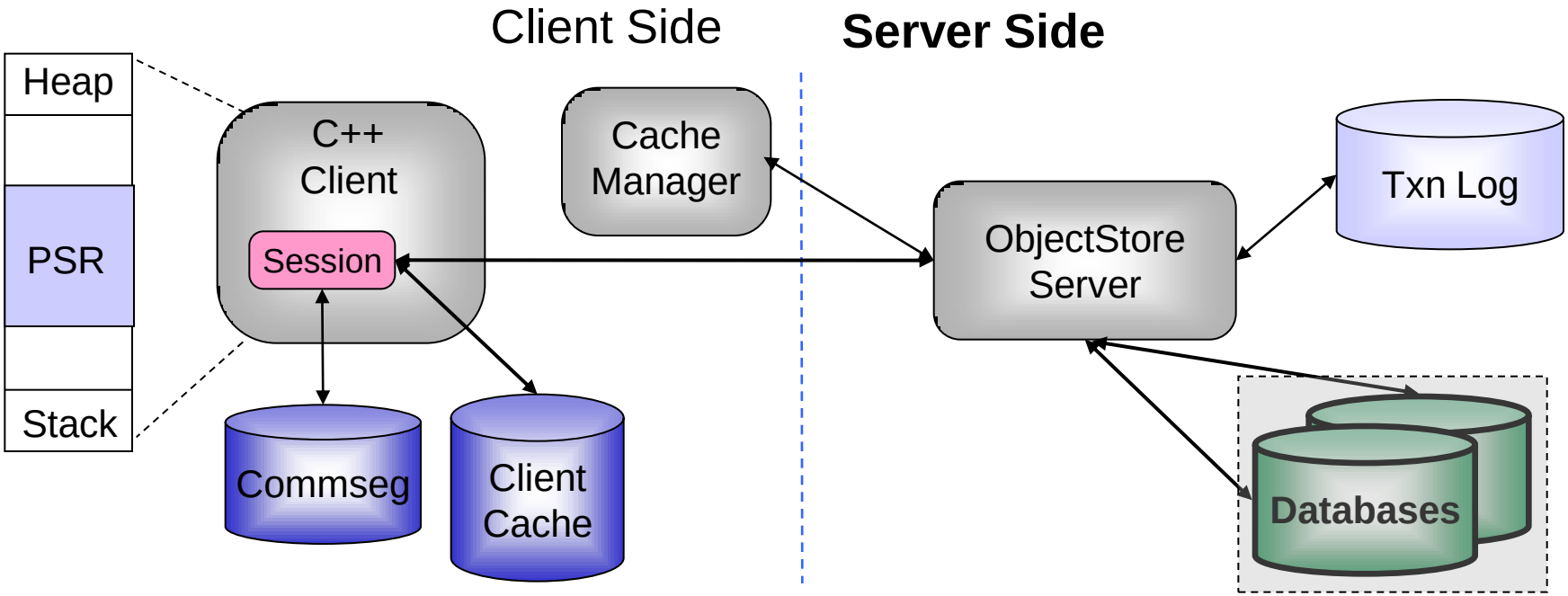
- We will examine each component in turn…
  - Processes shown as  icons
  - Files shown as  icons

ObjectStore Server



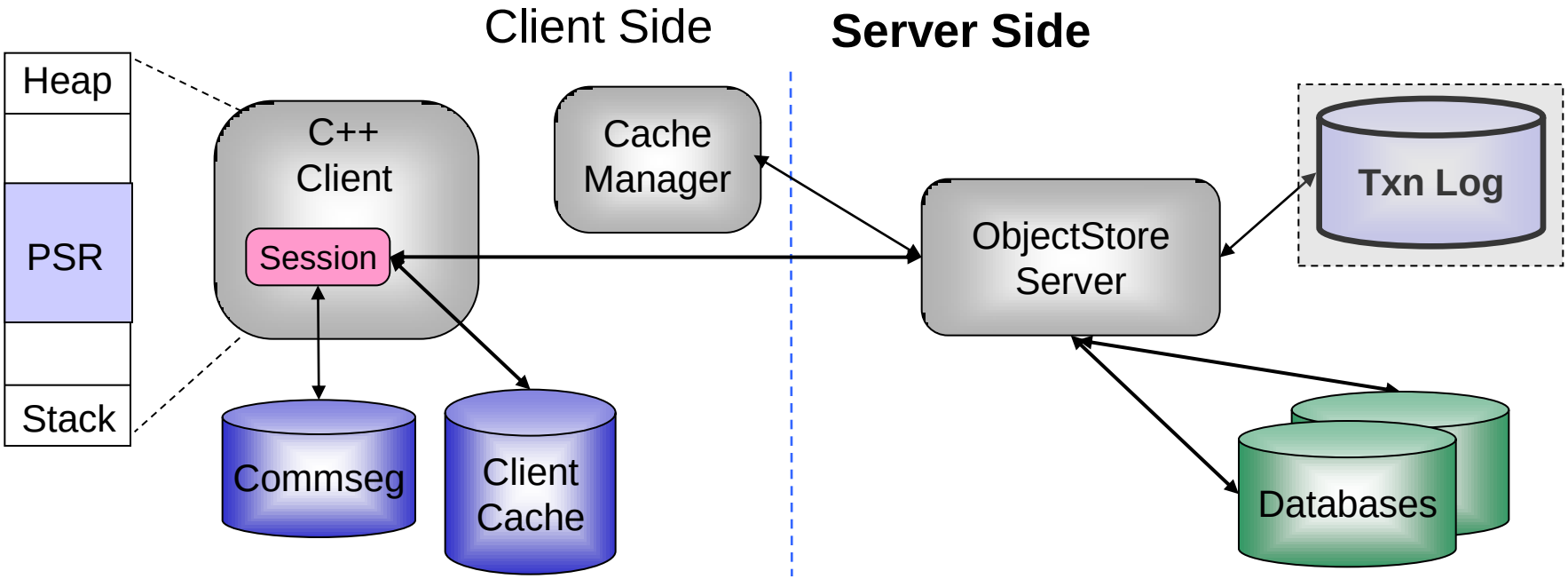
- Program is called `osserver.exe`
- Mediates all access to the databases it controls
- Serves out pages to clients
- Enforces transaction semantics by tracking '*page permits*'
- Co-operates with other servers in two-phase commits
- Automatic recovery mechanism when restarted
- Normally one `osserver` process per machine

ObjectStore Databases



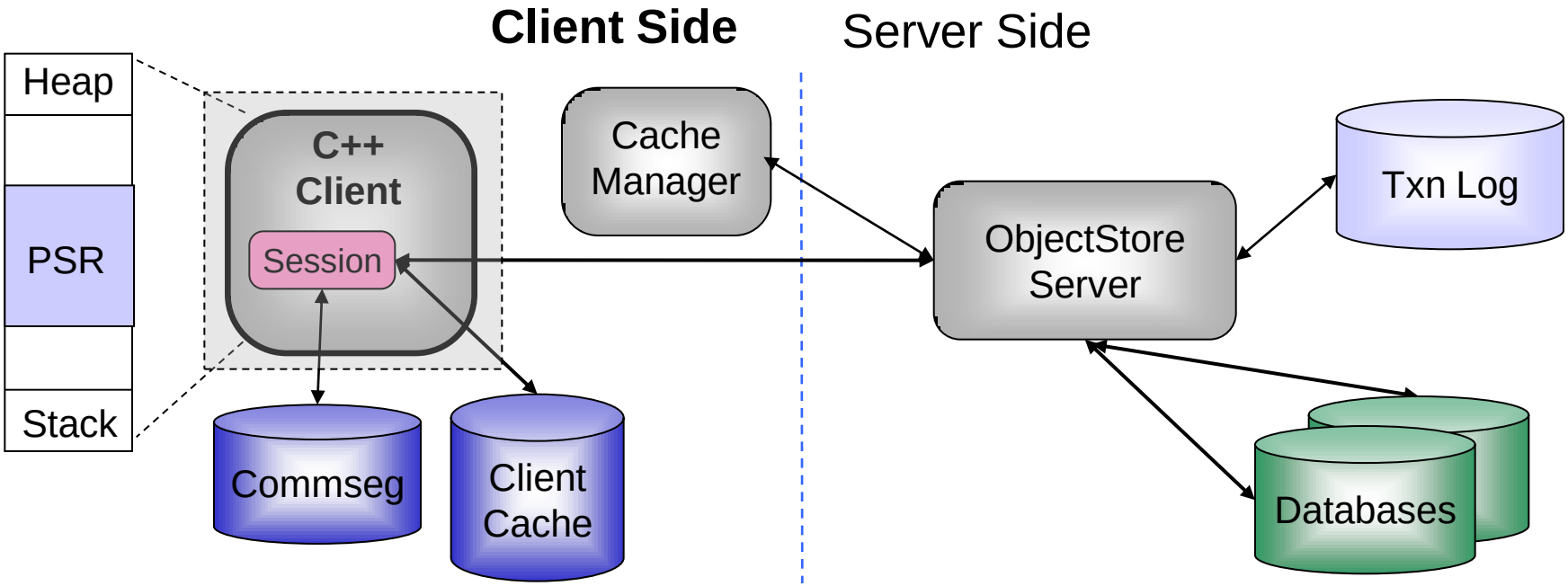
- Each database is managed by a single `osserver`
- ObjectStore databases are binary files held in the file system
- Each `osserver` can manage multiple databases
- Databases contain pages of memory containing C++ objects
- *Pages* are held in *Clusters*, and clusters are held in *Segments*
- Databases are normally deployed on server-local discs

ObjectStore Transaction Log



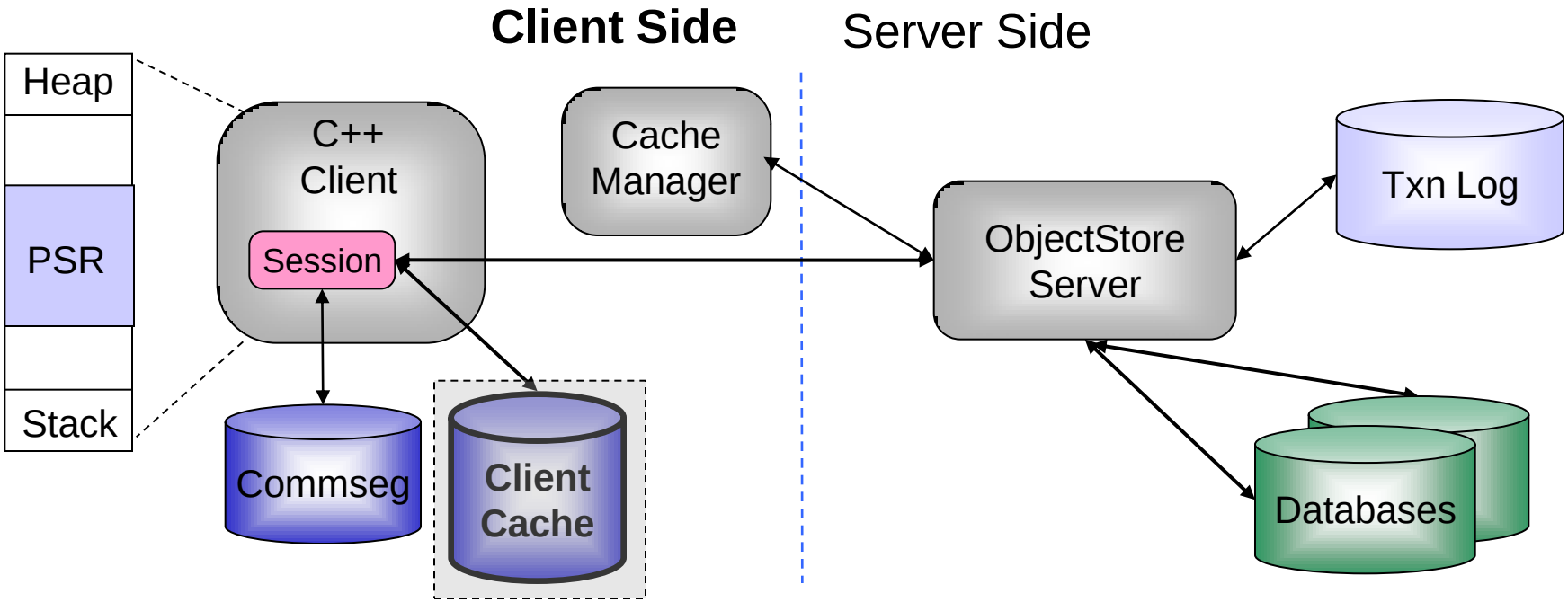
- Each `osserver` owns a binary file called the transaction log
- Updated pages are written to the transaction log
- Pages only ‘propagated’ to the database when txn commits
- Txn Log used for automatic recovery
- Txn Log allows faster commits;
- Txn Log used to implement *MVCC* mechanism

ObjectStore C++ Client



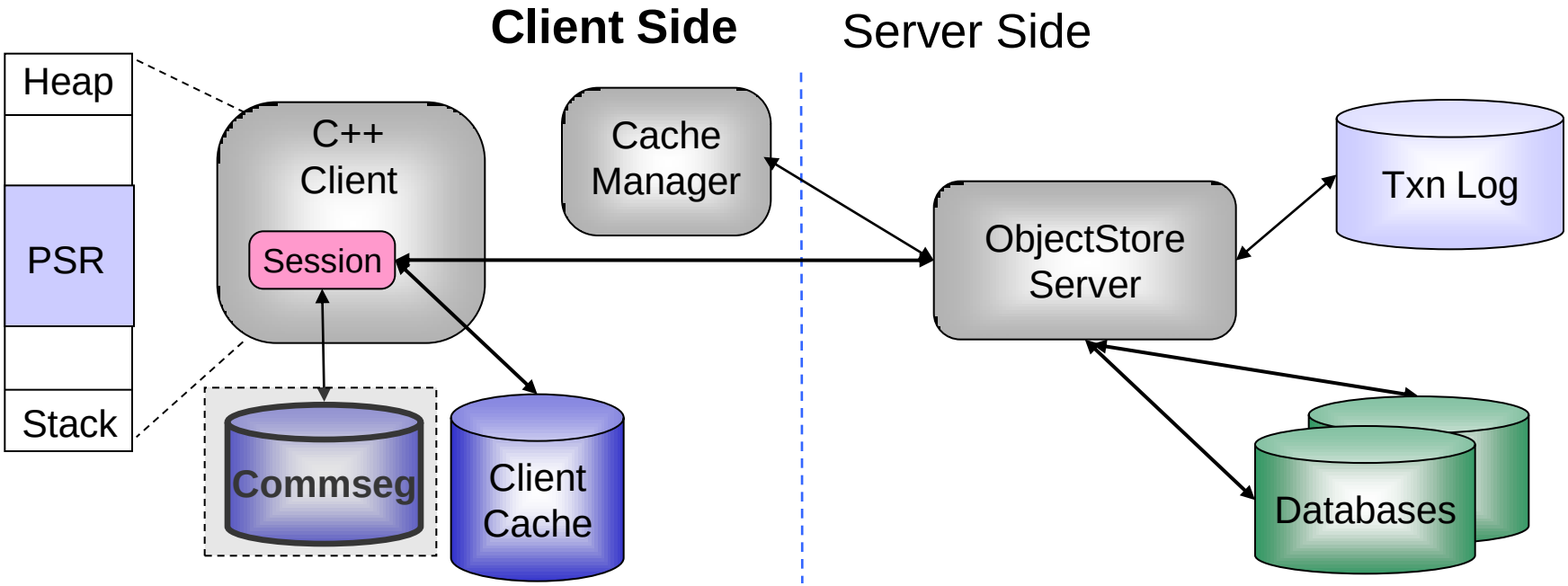
- The C++ client is the program you write
- It is linked with the ObjectStore libraries
- It opens databases and runs transactions against them
- It allocates objects, calls methods on them, and deletes them
- Pages are fetched automatically from the DB as needed
- ObjectStore automatically maintains a cache of recently accessed pages

ObjectStore Cache



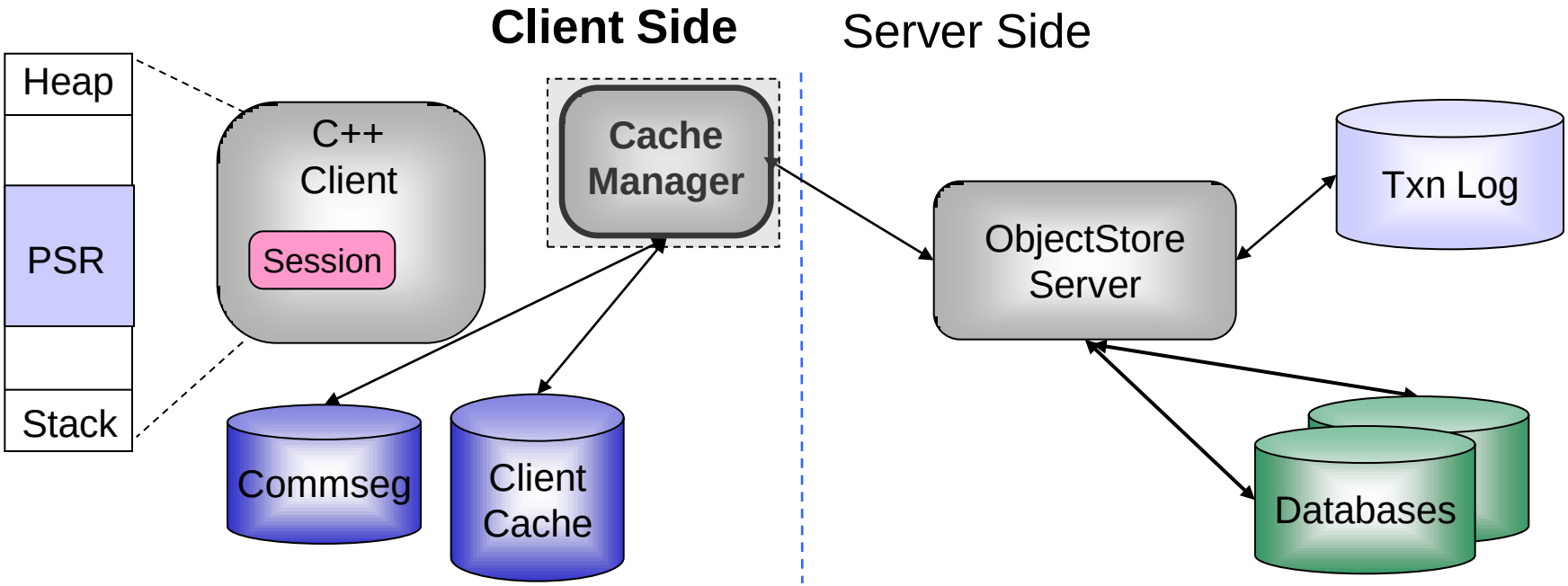
- There is one cache file per client process
- The cache is a memory mapped file
- It has a fixed size; it cannot change once the client has started
- Every page fetched from the database by this client is held here
- Pages can be retained in the cache between transactions

ObjectStore Commseg



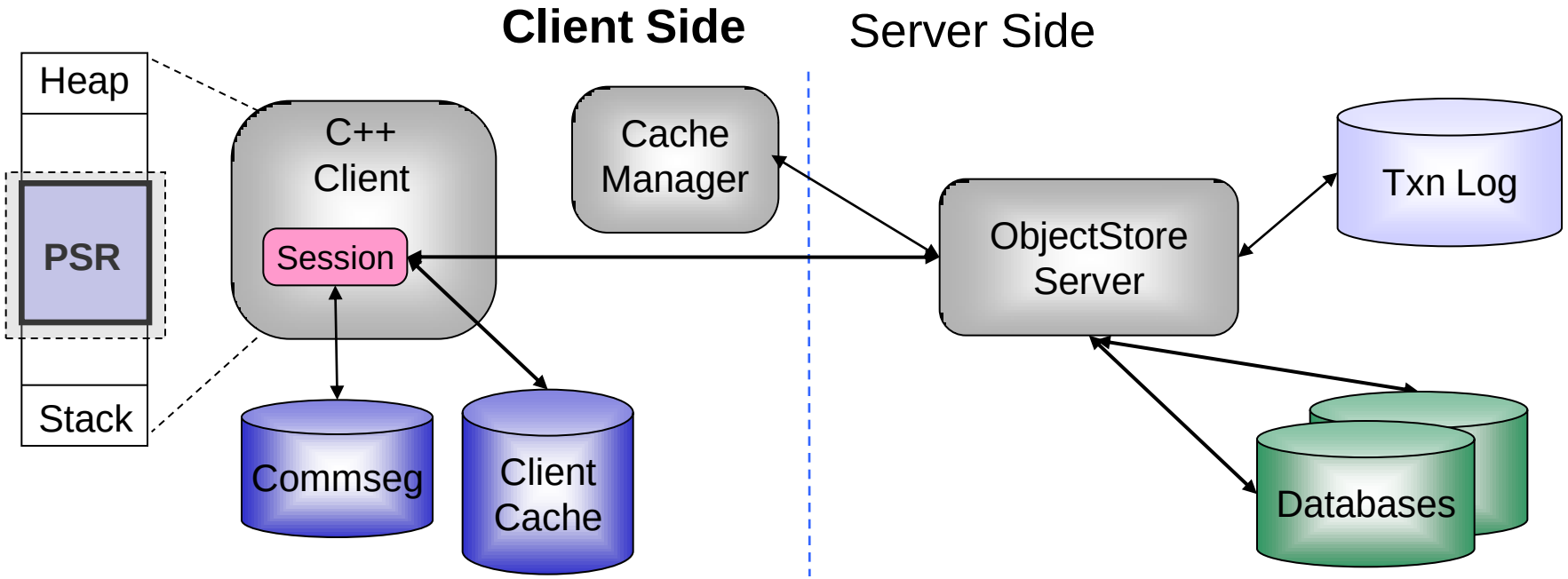
- There is one commseg file per client process
- The commseg is a memory mapped file
- It contains meta-information about every page in the cache
- There is a '*permit*' and a '*lock*' for every page in the cache
- Permits can be retained between transactions

ObjectStore Cache Manager



- There is one cache manager process per client machine
- All clients on that machine share a single cache manager
- Its main job is to handle *permit revokes*
- It reads and writes to the cache and commseg files
- It is NOT involved in page fetch directly in any way

ObjectStore Persistent Storage Region



- Each C++ program has a virtual address space (32 or 64-bit)
- The PSR is an area of this address space reserved by ObjectStore
- Address of every persistent object the client uses is mapped into the PSR
- The value of pointers to persistent objects will be in the range of the PSR
- At the end of every transaction the PSR is cleared…
- …so it can be reused for the next transaction

Agenda

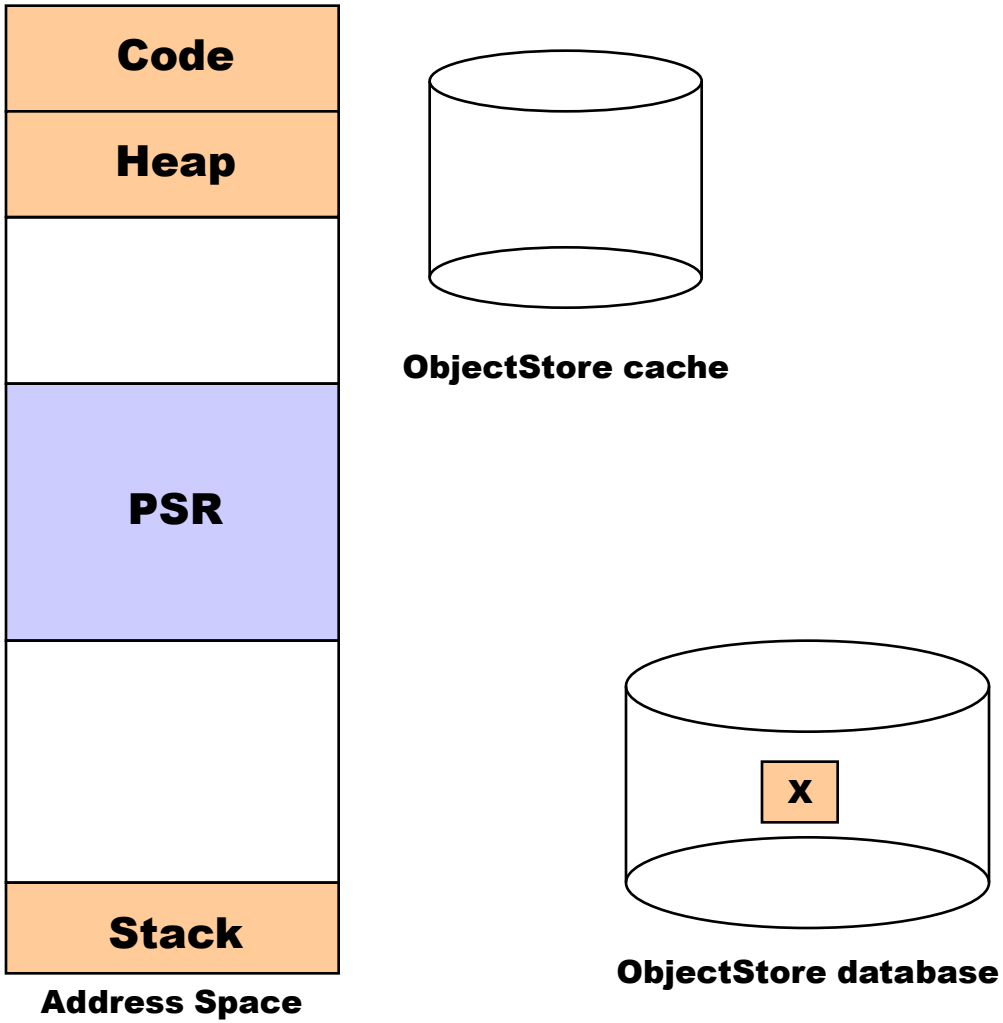
- What is ObjectStore?
- ObjectStore Components
- Fetching and Mapping a Page
- ObjectStore Programming
- Opening and Closing a database
- Using Transactions
- Creating and Deleting Persistent Objects
- Finding Initial Persistent Objects
- Segments and Clusters
- Writing and Using a Persistent Class
- Conclusion

Fetching and Mapping a Page

- Pages are automatically fetched and mapped by the client ‘lazily’ as needed
- Pages are held in the client cache
- Pointers on the fetched page are ‘swizzled’ so they point into the PSR…
  - …either at objects already fetched
  - …or at ranges reserved for objects yet-to-be fetched
- Server permits and client locks are acquired automatically to ensure transaction consistency

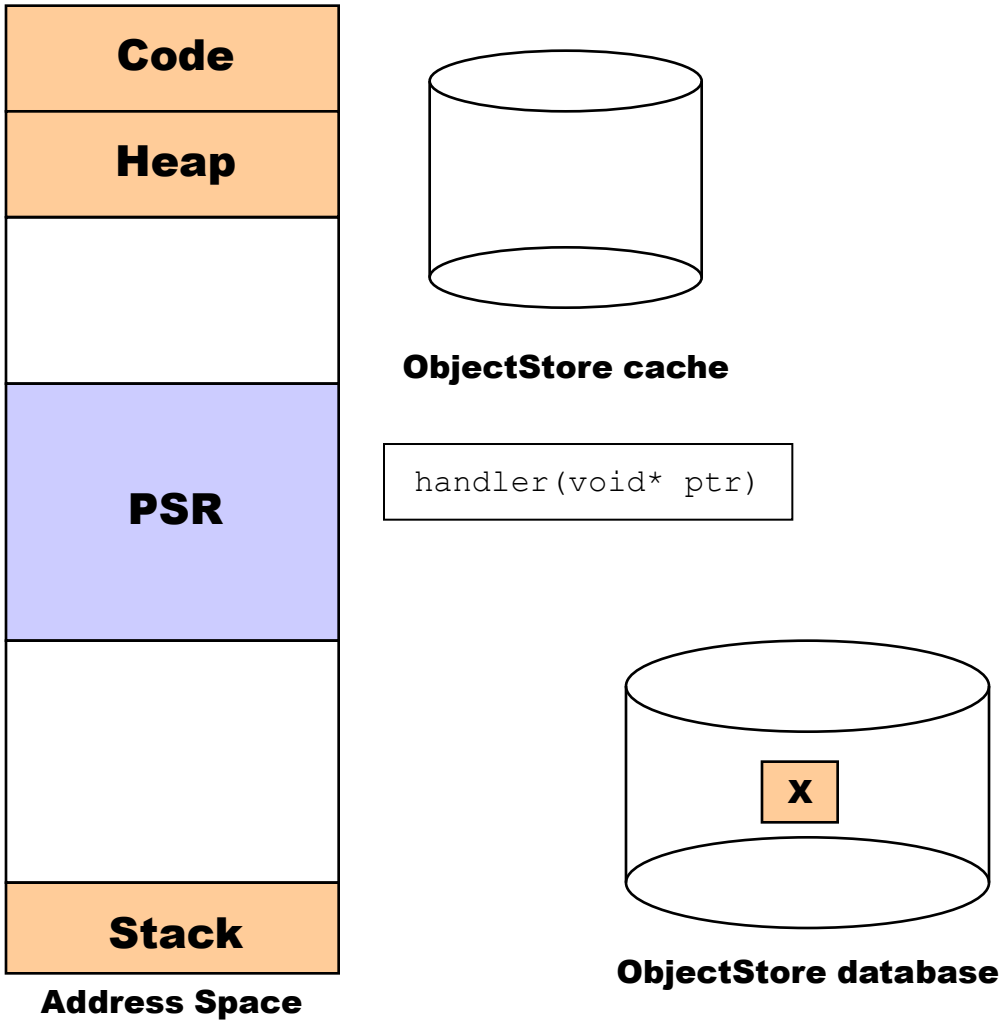
Fetching and Mapping a Page

- Fetching page X



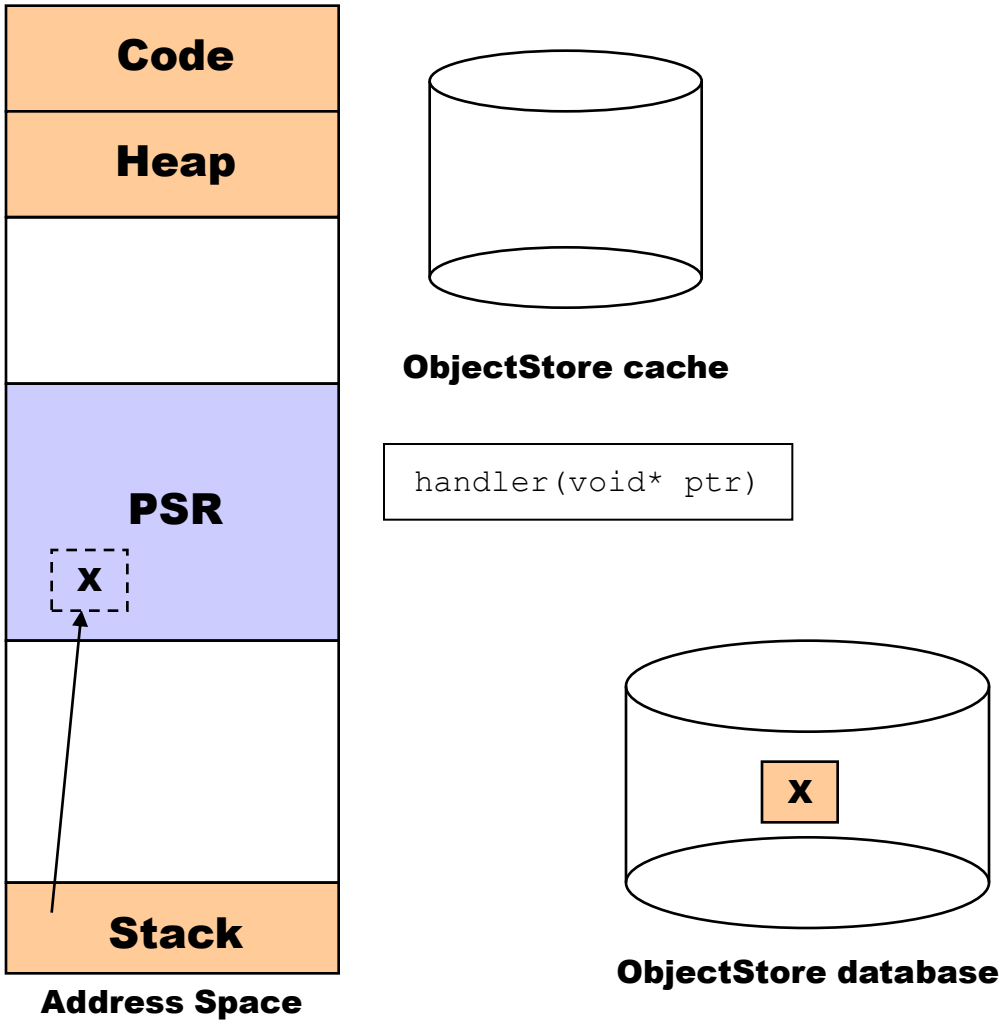
Fetching and Mapping a Page

- Fetching page X
  - ObjectStore installs a SIGSEGV handler



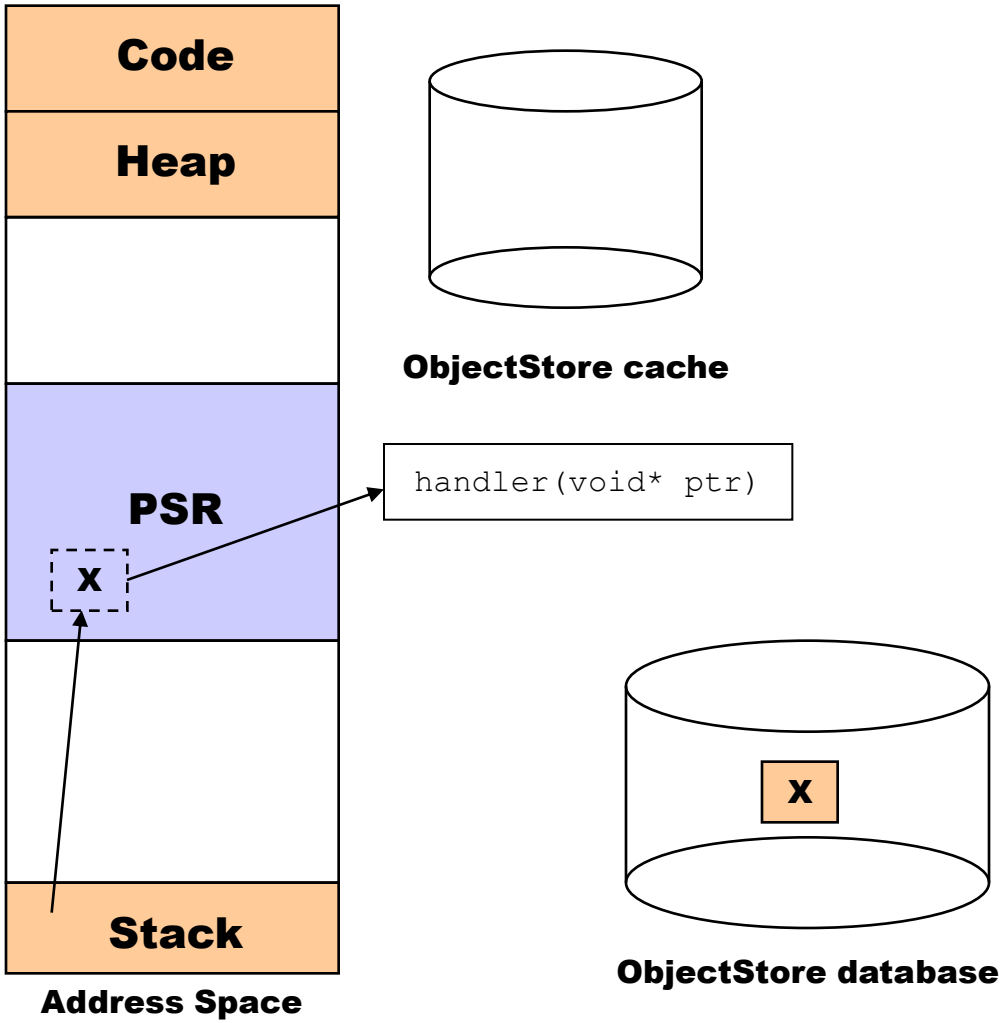
Fetching and Mapping a Page

- Fetching page X
  - ObjectStore installs SIGSEGV handler
  - Program obtains pointer to object on page X



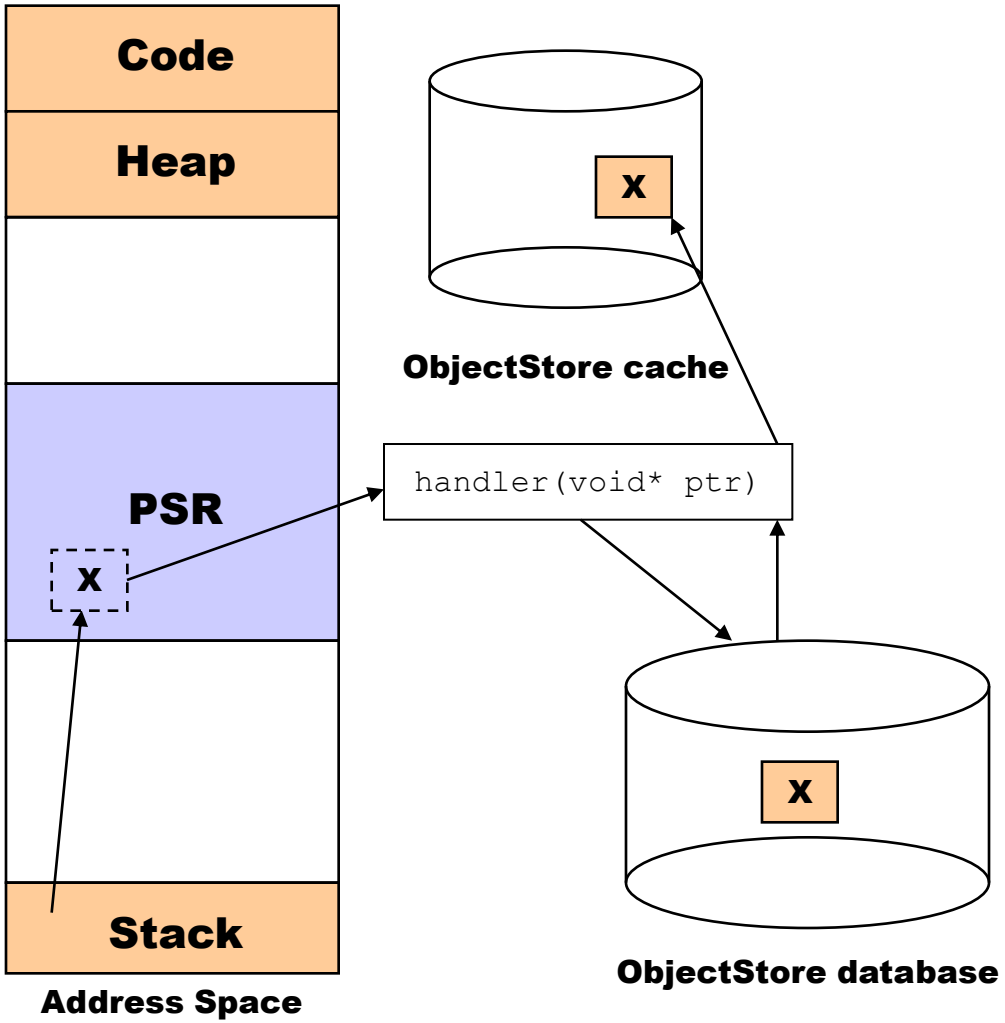
Fetching and Mapping a Page

- Fetching page X
  - ObjectStore installs SIGSEGV handler
  - Program obtains pointer to object on page X
  - Program dereferences pointer, causing SIGSEGV handler to be called



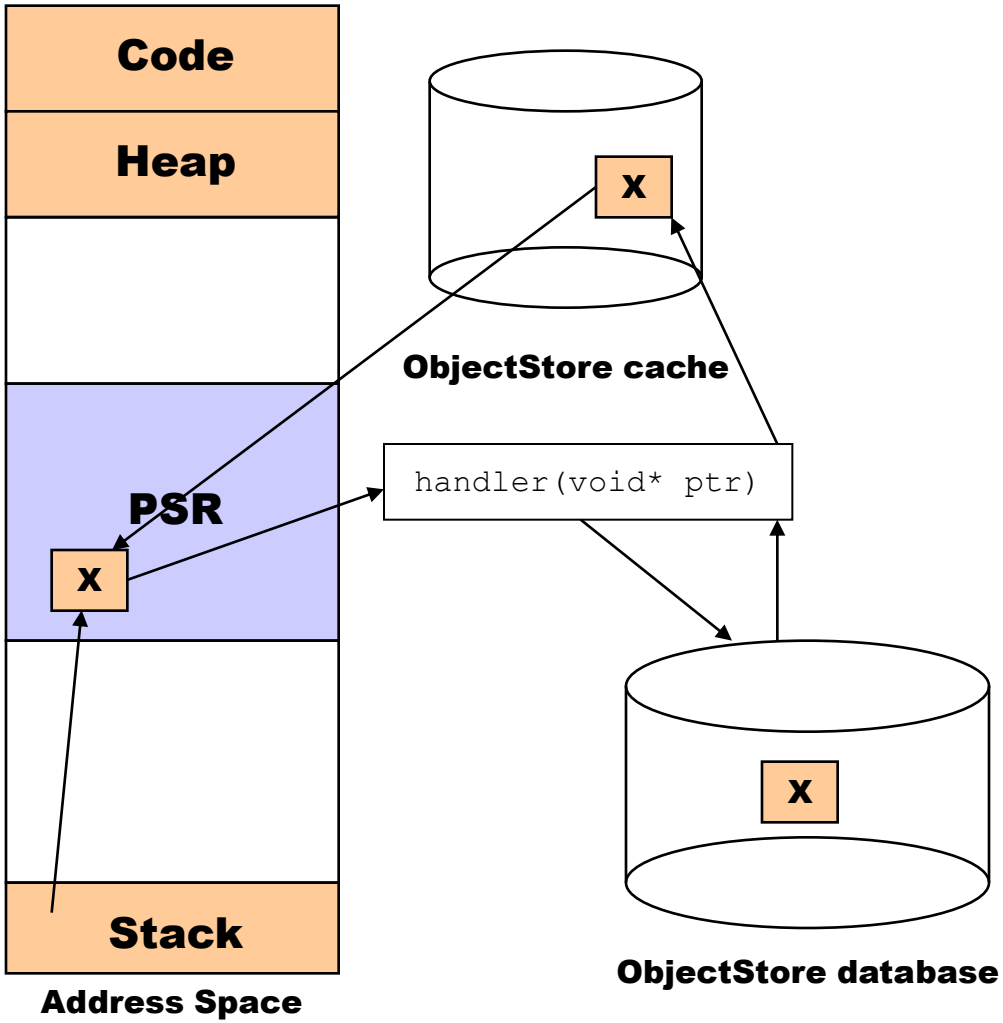
Fetching and Mapping a Page

- Fetching page X
  - ObjectStore installs SIGSEGV handler
  - Program obtains pointer to object on page X
  - Program dereferences pointer, causing SIGSEGV handler to be called
  - Virtual mapping table is consulted; page is fetched from server and stored in the cache



Fetching and Mapping a Page

- Fetching page X
  - ObjectStore installs SIGSEGV handler
  - Program obtains pointer to object on page X
  - Program dereferences pointer, causing SIGSEGV handler to be called
  - Virtual mapping table is consulted; page is fetched from server and stored in the cache
  - Page X is mapped to the address space, and execution continues



Agenda

- What is ObjectStore?
- ObjectStore Components
- Fetching and Mapping a Page
- ObjectStore Programming
- Opening and Closing a database
- Using Transactions
- Creating and Deleting Persistent Objects
- Finding Initial Persistent Objects
- Segments and Clusters
- Writing and Using a Persistent Class
- Conclusion

ObjectStore Programming

- Programmer uses the ObjectStore libraries
- We cover the following classes…
  - `objectstore`
  - `os_database`
  - `os_transaction`
  - `os_typespec`
  - `os_database_root`
  - `os_segment`
  - `os_cluster`

ObjectStore Programming

- Programmer uses the ObjectStore libraries
- We cover the following classes…
  - `objectstore`
  - `os_database`
  - `os_transaction`
  - `os_typespec`
  - `os_database_root`
  - `os_segment`
  - `os_cluster`

Along with useful C++ macros and templates …

ObjectStore Programming

```
#include <ostore/ostore.hh>
int main (int argc, char** argv)
{
  objectstore::initialize()
  OS_ESTABLISH_FAULT_HANDLER

    // ... your code ...

  OS_END_FAULT_HANDLER
  objectstore::shutdown();
  return 0;
}
```

ObjectStore Programming

```
#include <ostore/ostore.hh>
int main (int argc, char** argv)
{
  objectstore::initialize()
  OS_ESTABLISH_FAULT_HANDLER

    // ... your code ...

  OS_END_FAULT_HANDLER
  objectstore::shutdown();
  return 0;
}
```

Include the header file

ObjectStore Programming

```
#include <ostore/ostore.hh>
int main (int argc, char** argv)
{
  objectstore::initialize()
  OS_ESTABLISH_FAULT_HANDLER

    // ... your code ...

  OS_END_FAULT_HANDLER
  objectstore::shutdown();
  return 0;
}
```

Initialize ObjectStore

ObjectStore Programming

```
#include <ostore/ostore.hh>
int main (int argc, char** argv)
{
  objectstore::initialize()
  OS_ESTABLISH_FAULT_HANDLER

    // ... your code ...

  OS_END_FAULT_HANDLER
  objectstore::shutdown();
  return 0;
}
```

Set a fault handler

ObjectStore Programming

```
#include <ostore/ostore.hh>
int main (int argc, char** argv)
{
  objectstore::initialize()
  OS_ESTABLISH_FAULT_HANDLER

    // ... your code ...

  OS_END_FAULT_HANDLER
  objectstore::shutdown();
  return 0;
}
```

Remove the handler

ObjectStore Programming

```
#include <ostore/ostore.hh>
int main (int argc, char** argv)
{
  objectstore::initialize()
  OS_ESTABLISH_FAULT_HANDLER

    // ... your code ...

  OS_END_FAULT_HANDLER
  objectstore::shutdown();
  return 0;
}
```

Shut down

Agenda

- What is ObjectStore?
- ObjectStore Components
- Fetching and Mapping a Page
- ObjectStore Programming
- Opening and Closing a database
- Using Transactions
- Creating and Deleting Persistent Objects
- Finding Initial Persistent Objects
- Segments and Clusters
- Writing and Using a Persistent Class
- Conclusion

Opening and Closing a DB

```
#include <ostore/ostore.hh>
int main (int argc, char** argv)
{
  objectstore::initialize()
  OS_ESTABLISH_FAULT_HANDLER
    os_database* db = os_database::open(
            "d:/x.odb", 0, 0664);

    // ...your code
    db->close();
  OS_END_FAULT_HANDLER
  objectstore::shutdown();
  return 0;
}
```

Opening and Closing a DB

```
#include <ostore/ostore.hh>
int main (int argc, char** argv)
{
  objectstore::initialize()
  OS_ESTABLISH_FAULT_HANDLER
    os_database* db = os_database::open(
            "d:/x.odb", 0, 0664);

    // ...your code
    db->close();
  OS_END_FAULT_HANDLER
  objectstore::shutdown();
  return 0;
}
```

File path

Opening and Closing a DB

```
#include <ostore/ostore.hh>
int main (int argc, char** argv)
{
  objectstore::initialize()
  OS_ESTABLISH_FAULT_HANDLER
    os_database* db = os_database::open(
            "d:/x.odb", 0, 0664);

    // ...your code
    db->close();
  OS_END_FAULT_HANDLER
  objectstore::shutdown();
  return 0;
}
```

Create mode

Open mode

Opening and Closing a DB

```
#include <ostore/ostore.hh>
int main (int argc, char** argv)
{
  objectstore::initialize()
  OS_ESTABLISH_FAULT_HANDLER
    os_database* db = os_database::open(
            "d:/x.odb", 0, 0664);

    // ...your code
    db->close();
  OS_END_FAULT_HANDLER
  objectstore::shutdown();
  return 0;
}
```

Closing the database invalidates pages in the cache

Agenda

- What is ObjectStore?
- ObjectStore Components
- Fetching and Mapping a Page
- ObjectStore Programming
- Opening and Closing a database
- Using Transactions
- Creating and Deleting Persistent Objects
- Finding Initial Persistent Objects
- Segments and Clusters
- Writing and Using a Persistent Class
- Conclusion

ObjectStore Transactions

- A transaction is a group of operations that act on persistent data
  - *All* code that accesses persistent data must execute within a transaction
    - both read and write operations
  - You cannot access persistent data outside a transaction

ACID Properties of Transactions

- Transactions are fundamental to database integrity. They have the following properties:
  - ***Atomic****:* All work on persistent data either succeeds completely or fails completely.
  - ***Consistent***: You are responsible for defining consistent transaction boundaries.
  - ***Isolated***: No changes to persistent data are visible until the transaction commits.
  - ***Durable***: The server ensures that all committed data is safe on disk and recoverable

ObjectStore Transaction Types

- **Read** or **Write**
  - Read txn 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 txn
- **Lexical** or **Dynamic**
  - Lexical txns automatically retry on deadlock
  - Lexical must start and end in same code block
  - Lexical txns are always thread-local
  - Dynamic txns are the lower level `os_transaction` class
  - Better suited to multi-threaded applications

Using Lexical Transactions

```
os_database* db = os_database::open(
   "d:/x.odb", 0, 0664);

OS_BEGIN_TXN(use_case0, 0, os_transaction::read_only)
{
   // Create, read and
   // update persistent
   // objects...
}
OS_END_TXN(use_case0)

//... other code ...

db->close();
```

Using Lexical Transactions

```
os_database* db = os_database::open(
   "d:/x.odb", 0, 0664);

OS_BEGIN_TXN(use_case0, 0, os_transaction::read_only)
{
   // Create, read and
   // update persistent
   // objects...
}
OS_END_TXN(use_case0)

//... other code ...

db->close();
```

Txn Type

Using Lexical Transactions

```
os_database* db = os_database::open(
   "d:/x.odb", 0, 0664);

OS_BEGIN_TXN(use_case1, 0, os_transaction::update)
{
   // Create, read and
   // update persistent
   // objects...
}
OS_END_TXN(use_case1)

//... other code ...

db->close();
```

Txn Type

Using Lexical Transactions

```
os_database* db = os_database::open(
    "d:/x.odb", 0, 0664);

OS_BEGIN_TXN(use_case1, 0, os_transaction::update)
{
    // Create, read and
    // update persistent
    // objects...
}
OS_END_TXN(use_case1)

//... other code ...

db->close();
```

Txn Name

Using Lexical Transactions

```
os_database* db = os_database::open(
    "d:/x.odb", 0, 0664);

OS_BEGIN_TXN(use_case1, 0, os_transaction::update)
{
    // Create, read and
    // update persistent
    // objects...
}
OS_END_TXN(use_case1)

//... other code ...

db->close();
```

Run through end macro to commit

Nested Transactions

```
os_database* db = os_database::open("d:/x.odb);
OS_BEGIN_TXN(use_case1, 0, os_transaction::update)
{
   OS_BEGIN_TXN(sub1,0,os_transaction::update)
   {
   }
   OS_END_TXN(sub1)

   OS_BEGIN_TXN(sub2,0,os_transaction::update)
   {
      OS_BEGIN_TXN(sub3,0,os_transaction::read_only)
      {
      }
      OS_END_TXN(sub3)
   }
   OS_END_TXN(sub2)
}
OS_END_TXN(use_case1)

//... other code ...
```

Nested Transactions

```
os_database* db = os_database::open("d:/x.odb);
OS_BEGIN_TXN(use_case1, 0, os_transaction::update)
{
   OS_BEGIN_TXN(sub1,0,os_transaction::update)
   {
   }
   OS_END_TXN(sub1)

   OS_BEGIN_TXN(sub2,0,os_transaction::update)
   {
      OS_BEGIN_TXN(sub3,0,os_transaction::read_only)
      {
      }
      OS_END_TXN(sub3)
   }
   OS_END_TXN(sub2)
}
OS_END_TXN(use_case1)

//... other code ...
```

Can freely nest different txn types

Nested Transactions

```
os_database* db = os_database::open("d:/x.odb);
OS_BEGIN_TXN(use_case1, 0, os_transaction::update)
{
   OS_BEGIN_TXN(sub1,0,os_transaction::update)
   {
   }
   OS_END_TXN(sub1)

   OS_BEGIN_TXN(sub2,0,os_transaction::update)
   {
      OS_BEGIN_TXN(sub3,0,os_transaction::read_only)
      {
      }
      OS_END_TXN(sub3)
   }
   OS_END_TXN(sub2)
}
OS_END_TXN(use_case1)

//... other code ...
```

…to arbitrary nesting levels

Agenda

- What is ObjectStore?
- ObjectStore Components
- Fetching and Mapping a Page
- ObjectStore Programming
- Opening and Closing a database
- Using Transactions
- Creating and Deleting Persistent Objects
- Finding Initial Persistent Objects
- Segments and Clusters
- Writing and Using a Persistent Class
- Conclusion

Creating a Persistent Object

- Persistent objects are created using an overloaded `new` operator
  - It is an overloading of C++ ‘placement’ new that returns an address in the PSR
  - Must be used within a transaction

Creating a Persistent Object

```
os_database* db = os_database::open("d:/x.odb");

OS_BEGIN_TXN(use_case1, 0, os_transaction::update)
{
   Foo* f = new (db, ts<Foo>()) Foo();

   //...other code...
}
OS_END_TXN(use_case1)

//... other code ...
```

Creating a Persistent Object

```
os_database* db = os_database::open("d:/x.odb");

OS_BEGIN_TXN(use_case1, 0, os_transaction::update)
{
    Foo* f = new (db, ts<Foo>()) Foo();

    //...other code...
}
OS_END_TXN(use_case1)

//... other code ...
```

Standard no argument Foo ctor

Creating a Persistent Object

```
os_database* db = os_database::open("d:/x.odb");

OS_BEGIN_TXN(use_case1, 0, os_transaction::update)
{
    Foo* f = new (db, ts<Foo>()) Foo();

    //...other code...
}
OS_END_TXN(use_case1)

//... other code ...
```

'Placement' new

Creating a Persistent Object

```
os_database* db = os_database::open("d:/x.odb");

OS_BEGIN_TXN(use_case1, 0, os_transaction::update)
{
    Foo* f = new (db, ts<Foo>()) Foo();

    //...other code...
}
OS_END_TXN(use_case1)

//... other code ...
```

Create object in this database

Creating a Persistent Object

```
os_database* db = os_database::open("d:/x.odb");

OS_BEGIN_TXN(use_case1, 0, os_transaction::update)
{
   Foo* f = new (db, ts<Foo>()) Foo();

   //...other code...
}
OS_END_TXN(use_case1)

//... other code ...
```

Template function identifies the class of the persistent object

The TypeSpec Template Function

- Template function returns an instance of `os_typespec`
  - Essentially an integer used by ObjectStore to identify a persistent type
- The `ts<>()` function returns the correct `os_typespec` for all types
  - very efficiently
  - using a consistent syntax e.g.

```
ts<int>()          ts< vector<int> >()
ts<char*>()        ts< map<int,char*> >()
ts<double**>()     ts< T >()
ts<Foo>()          ts< T* >()
ts<Foo*>()         ts< T,U* >()
```

The TypeSpec Template Function

- Template function returns an instance of `os_typespec`
  - Essentially an integer used by ObjectStore to identify a persistent type
- The `ts<>()` function returns the correct `os_typespec` for all types
  - very efficiently
  - using a consistent syntax e.g.

```
ts<int>()
ts<char*>()
ts<double**>()
ts<Foo>()
ts<Foo*>()
```

```
ts< vector<int> >()
ts< map<int,char*> >()
ts< T >()
ts< T* >()
ts< T,U* >()
```

It even works on template arguments

Creating a Persistent Templated Object

```
os_database* db = os_database::open("d:/x.odb");

OS_BEGIN_TXN(use_case1, 0, os_transaction::update)
{
   os_List<T*>* list = new
      (db, ts< os_List<T*> >()) os_List<T*>();

   //...other code...
}
OS_END_TXN(use_case1)

//... other code ...
```

Creating a Persistent Templated Object

```
os_database* db = os_database::open("d:/x.odb");

OS_BEGIN_TXN(use_case1, 0, os_transaction::update)
{
   os_List<T*>* list = new
      (db, ts< os_List<T*> >()) os_List<T*>();

   //...other code...
}
OS_END_TXN(use_case1)

//... other code ...
```

Standard os_List ctor

Creating a Persistent Templated Object

```
os_database* db = os_database::open("d:/x.odb");

OS_BEGIN_TXN(use_case1, 0, os_transaction::update)
{
   os_List<T*>* list = new
      (db, ts< os_List<T*> >()) os_List<T*>();

   //...other code...
}
OS_END_TXN(use_case1)

//... other code ...
```

'Placement' new

Creating a Persistent Templated Object

```
os_database* db = os_database::open("d:/x.odb");

OS_BEGIN_TXN(use_case1, 0, os_transaction::update)
{
   os_List<T*>* list = new
      (db, ts< os_List<T*> >()) os_List<T*>();

}
OS_END_TXN(use_case1)

//... other code ...
```

Database that will contain this list

Creating a Persistent Templated Object

```
os_database* db = os_database::open("d:/x.odb");

OS_BEGIN_TXN(use_case1, 0, os_transaction::update)
{
   os_List<T*>* list = new
      (db, ts< os_List<T*> >()) os_List<T*>();

   //...other code...
}
OS_END_TXN(use_case1)

//... other code ...
```

Template function used as before

Creating a Persistent C++ Array

```
os_database* db = os_database::open("d:/x.odb");

OS_BEGIN_TXN(use_case1, 0, os_transaction::update)
{
   Foo* fooArr = new(db, ts<Foo>(), 9) Foo[9];

   //...other code...
}
OS_END_TXN(use_case1)

//... other code ...
```

Creating a Persistent C++ Array

```
os_database* db = os_database::open("d:/x.odb");

OS_BEGIN_TXN(use_case1, 0, os_transaction::update)
{
    Foo* fooArr = new(db, ts<Foo>(), 9) Foo[9];

    //...other code...
}
OS_END_TXN(use_case1)

//... other code ...
```

Standard Foo object array ctor

Creating a Persistent C++ Array

```
os_database* db = os_database::open("d:/x.odb");

OS_BEGIN_TXN(use_case1, 0, os_transaction::update)
{
    Foo* fooArr = new(db, ts<Foo>(), 9) Foo[9];

    //...other code...
}
OS_END_TXN(use_case1)

//... other code ...
```

'Placement' new

Creating a Persistent C++ Array

```
os_database* db = os_database::open("d:/x.odb");

OS_BEGIN_TXN(use_case1, 0, os_transaction::update)
{
   Foo* fooArr = new(db, ts<Foo>(), 9) Foo[9];

   //...other code...
}
OS_END_TXN(use_case1)

//... other code ...
```

The database…

Creating a Persistent C++ Array

```
os_database* db = os_database::open("d:/x.odb");

OS_BEGIN_TXN(use_case1, 0, os_transaction::update)
{
   Foo* fooArr = new(db, ts<Foo>(), 9) Foo[9];

   //...other code...
}
OS_END_TXN(use_case1)

//... other code ...
```

..the typespec…

Creating a Persistent C++ Array

```
os_database* db = os_database::open("d:/x.odb");

OS_BEGIN_TXN(use_case1, 0, os_transaction::update)
{
   Foo* fooArr = new(db, ts<Foo>(), 9) Foo[9];

   //...other code...
}
OS_END_TXN(use_case1)

//... other code ...
```

…the array size

Creating Persistent C++ Native Types

```
os_database* db = os_database::open("d:/x.odb");

OS_BEGIN_TXN(use_case1, 0, os_transaction::update)
{
   int* x = new(db, ts<int>()) int;

   //...other code...
}
OS_END_TXN(use_case1)

//... other code ...
```

Creating Persistent C++ Native Types

```
os_database* db = os_database::open("d:/x.odb");

OS_BEGIN_TXN(use_case1, 0, os_transaction::update)
{
    int* x = new(db, ts<int>()) int;

    //...other code...
}
OS_END_TXN(use_case1)

//... other code ...
```

A single persistent integer

Creating Persistent C++ Native Types

```
os_database* db = os_database::open("d:/x.odb");

OS_BEGIN_TXN(use_case1, 0, os_transaction::update)
{
   int* x = new(db, ts<int>) int;
   double* dArr = new(db, ts<double>(), 1000)
      double[1000];

   //...other code...
}
OS_END_TXN(use_case1)

//... other code ...
```

Creating Persistent C++ Native Types

```
os_database* db = os_database::open("d:/x.odb");

OS_BEGIN_TXN(use_case1, 0, os_transaction::update)
{
   int* x = new(db, ts<int>) int;
   double* dArr = new(db, ts<double>(), 1000)
      double[1000];

   //...other code...
}
OS_END_TXN(use_case1)

//... other code ...
```

A persistent array of doubles

Deleting Persistent Objects

```
os_database* db = os_database::open("d:/x.odb");

OS_BEGIN_TXN(use_case1, 0, os_transaction::update)
{
   Foo* f = new(db, ts<Foo>()) Foo();
   Bar* barArr = new(db, ts<Bar>(), 100) Bar[100];

   //...other code...

   delete f;
   delete [] barArr;
}
OS_END_TXN(use_case1)

//... other code ...
```

Deleting Persistent Objects

```
os_database* db = os_database::open("d:/x.odb");

OS_BEGIN_TXN(use_case1, 0, os_transaction::update)
{
   Foo* f = new(db, ts<Foo>()) Foo();
   Bar* barArr = new(db, ts<Bar>(), 100) Bar[100];

   //...other code...

   delete f;
   delete [] barArr;
}
OS_END_TXN(use_case1)

//... other code ...
```

Use the correct delete operator for single objects and arrays

Deleting Persistent Objects

```
os_database* db = os_database::open("d:/x.odb");

OS_BEGIN_TXN(use_case1, 0, os_transaction::update)
{
   Foo* f = new(db, ts<Foo>()) Foo();
   Bar* barArr = new(db, ts<Bar>(), 100) Bar[100];

   //...other code...

   delete f;
   delete [] barArr;
}
OS_END_TXN(use_case1)

//... other code ...
```

Only visible in the database if the transaction commits

Agenda

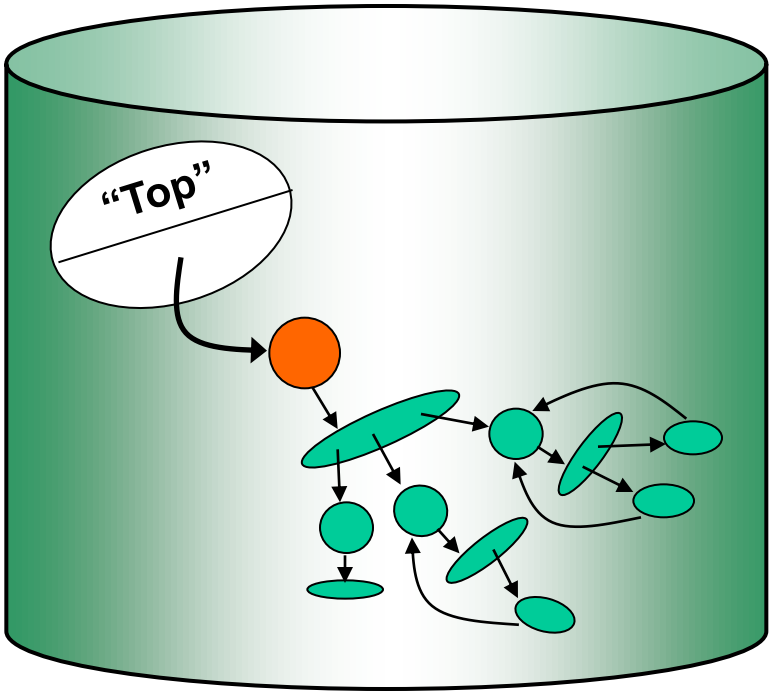
- What is ObjectStore?
- ObjectStore Components
- Fetching and Mapping a Page
- ObjectStore Programming
- Opening and Closing a database
- Using Transactions
- Creating and Deleting Persistent Objects
- Finding Initial Persistent Objects
- Segments and Clusters
- Writing and Using a Persistent Class
- Conclusion

Database Roots

- Object navigation requires some form of initial object
  - This could be the result of a query
  - Or we can use database roots
- Database roots are persistent objects which have been labelled with a well-known-name

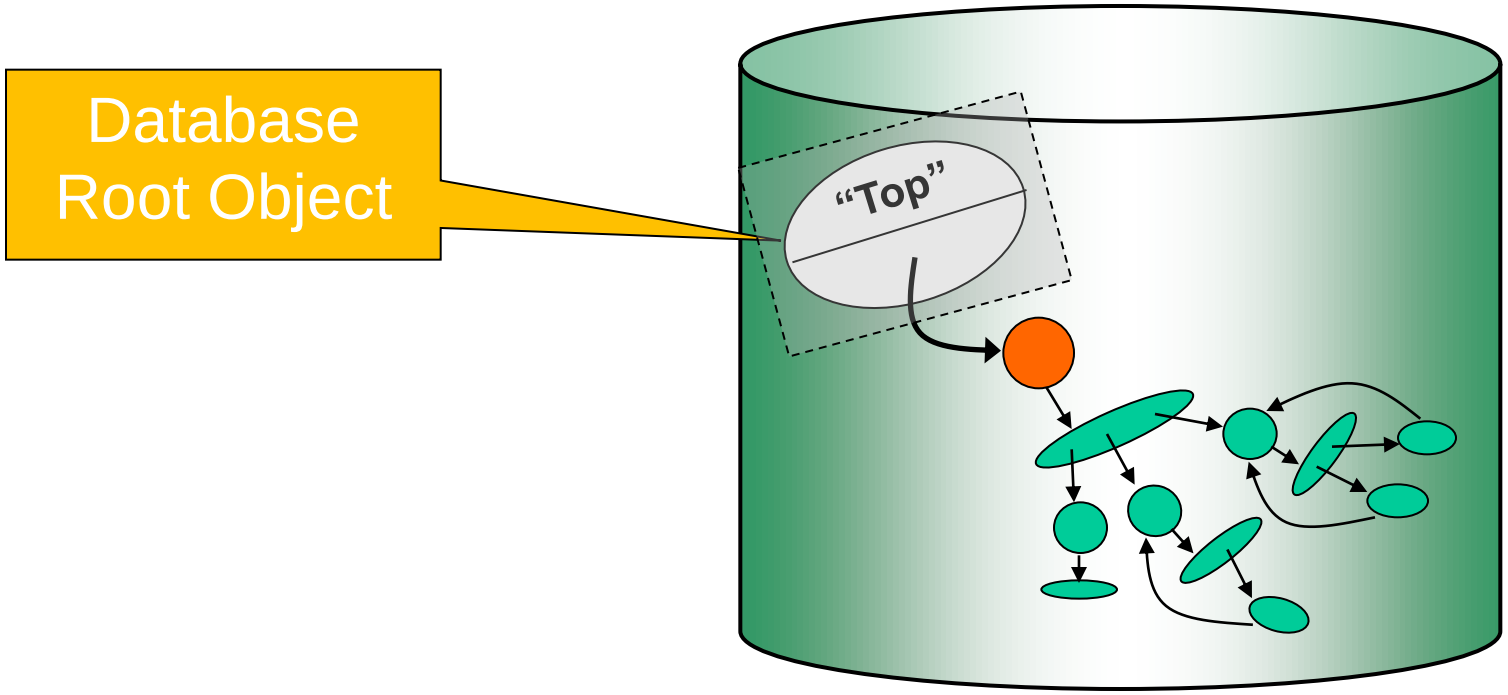
Database Roots

- Database roots consist of two parts
  - A root name held as a `char*`
  - A pointer to the object of interest held as a `void*`



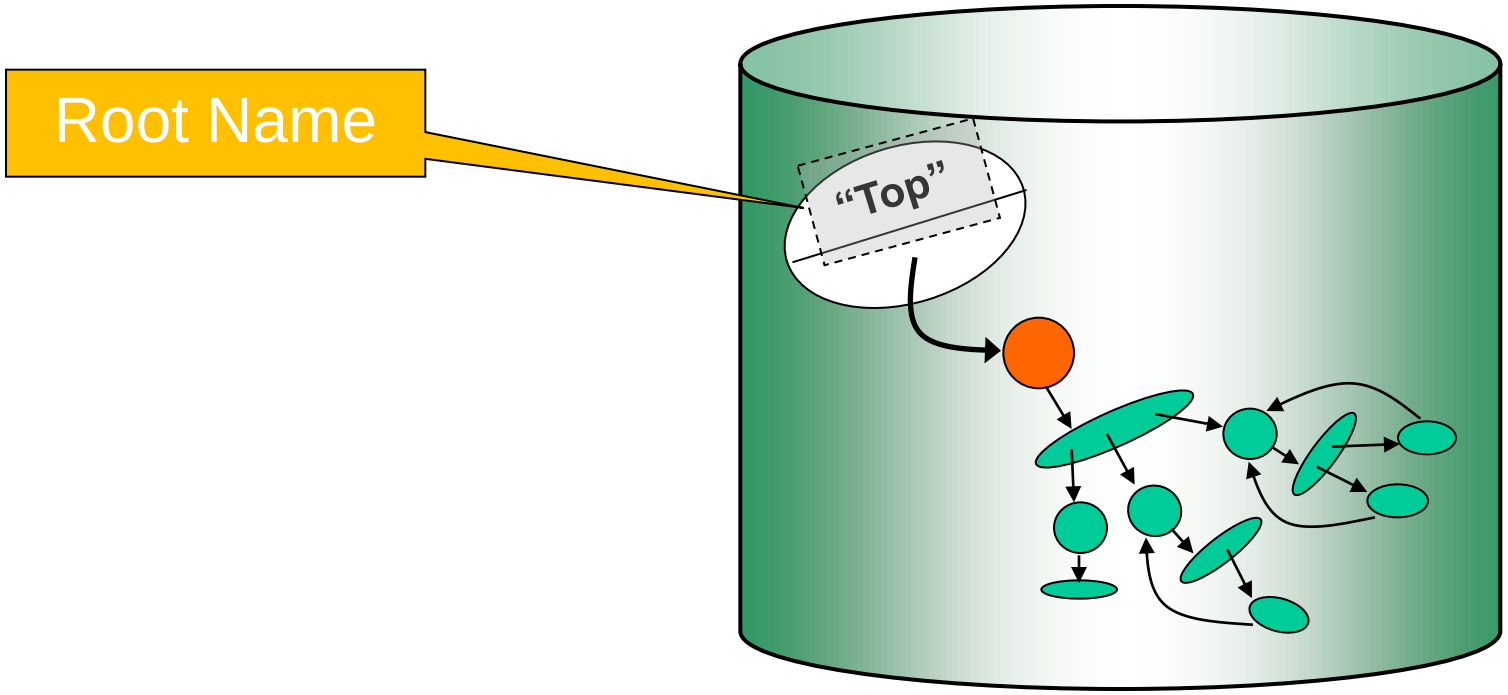
Database Roots

- Database roots consist of two parts
  - A root name held as a `char*`
  - A pointer to the object of interest held as a `void*`



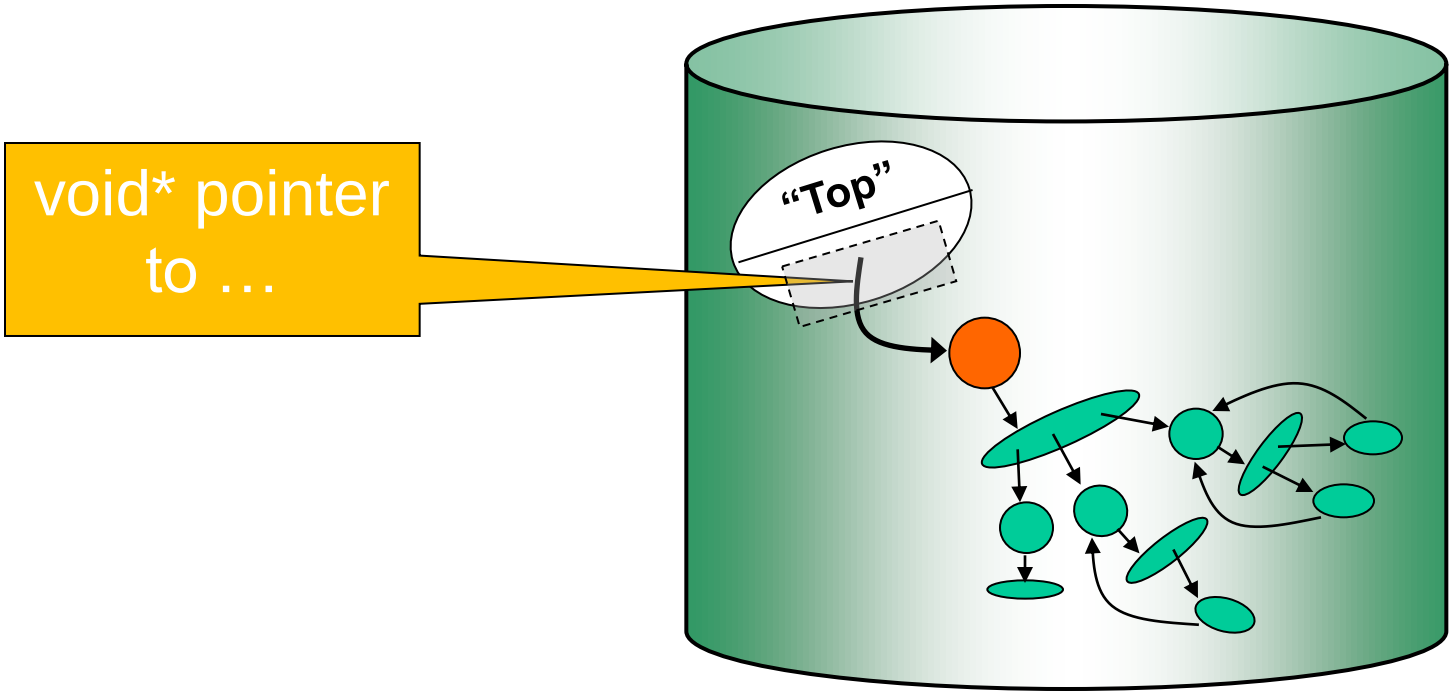
Database Roots

- Database roots consist of two parts
  - A root name held as a `char*`
  - A pointer to the object of interest held as a `void*`



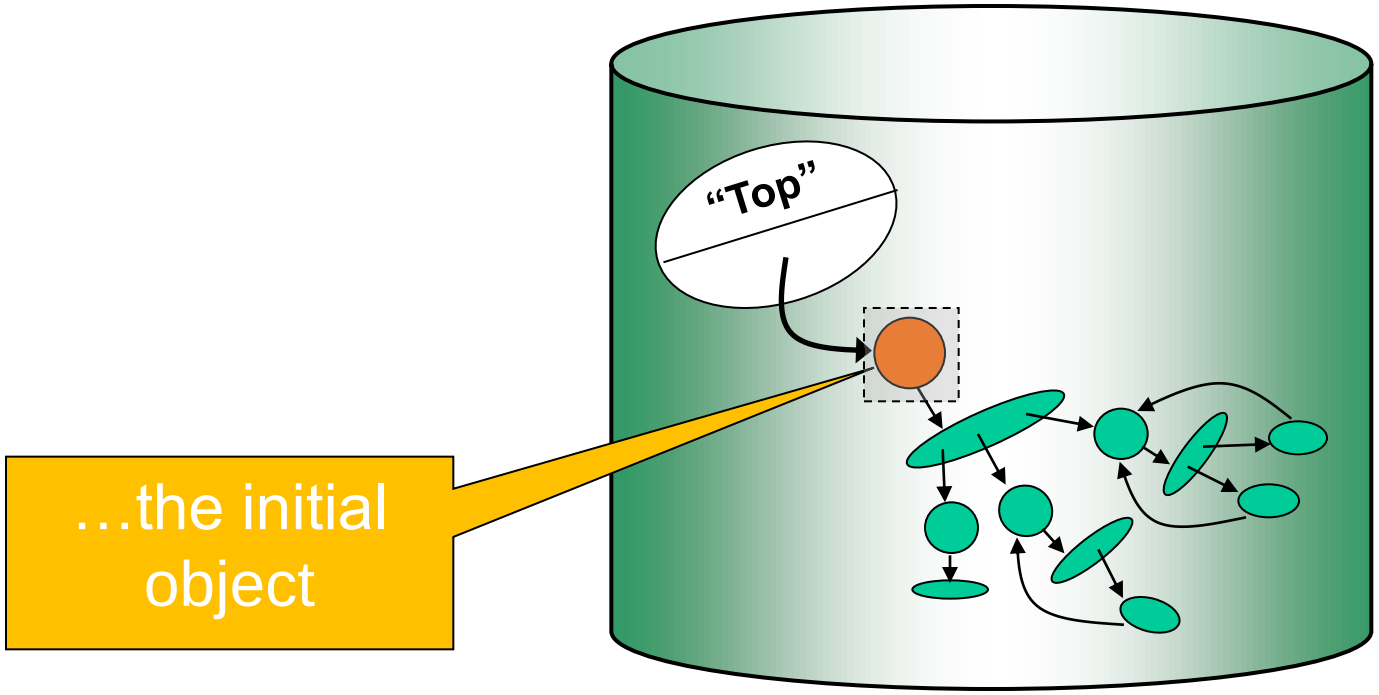
Database Roots

- Database roots consist of two parts
  - A root name held as a `char*`
  - A pointer to the object of interest held as a `void*`



Database Roots

- Database roots consist of two parts
  - A root name held as a `char*`
  - A pointer to the object of interest held as a `void*`



Creating a Database Root

```
os_database* db = os_database::open("d:/x.odb");
static char* buf = "Foo Root";

OS_BEGIN_TXN(use_case1, 0, os_transaction::update)
{
   Foo* f = new (db, ts<Foo>()) Foo();
   os_database_root* root =
         db->create_root(buf);
   root->set_value(f);
}
OS_END_TXN(use_case1)

//... other code ...
```

Creating a Database Root

```
os_database* db = os_database::open("d:/x.odb");
static char* buf = "Foo Root";

OS_BEGIN_TXN(use_case1, 0, os_transactio
{
    Foo* f = new (db, ts<Foo>()) Foo();
    os_database_root* root =
          db->create_root(buf);
    root->set_value(f);
}
OS_END_TXN(use_case1)

//... other code ...
```

Here's our root name…

Creating a Database Root

```
os_database* db = os_database::open("d:/x.odb");
static char* buf = "Foo Root";

OS_BEGIN_TXN(use_case1, 0, os_transaction::update)
{
   Foo* f = new (db, ts<Foo>()) Foo();
   os_database_root* root =
         db->create_root(buf);
   root->set_value(f);
}
OS_END_TXN(use_case1)

//... other code ...
```

Construct the initial object…

Creating a Database Root

```
os_database* db = os_database::open("d:/x.odb");
static char* buf = "Foo Root";

OS_BEGIN_TXN(use_case1, 0, os_transaction::update)
{
   Foo* f = new (db, ts<Foo>()) Foo();
   os_database_root* root =
         db->create_root(buf);
   root->set_value(f);
}
OS_END_TXN(use_case1)

//... other code ...
```

…create the root object…

Creating a Database Root

```
os_database* db = os_database::open("d:/x.odb");
static char* buf = "Foo Root";

OS_BEGIN_TXN(use_case1, 0, os_transaction::update)
{
   Foo* f = new (db, ts<Foo>()) Foo();
   os_database_root* root =
         db->create_root(buf);
   root->set_value(f);
}
OS_END_TXN(use_case1)

//... other code ...
```

…and set its value

Finding a Database Root

```
os_database* db = os_database::open("d:/x.odb");
static char* buf = "Foo Root";

OS_BEGIN_TXN(use_case2, 0, os_transaction::read_only)
{
   Foo* f = 0;
   os_database_root* root = db->find_root(buf);
   if(root)
   {
      f = (Foo*) root->get_value();
   }

   //...other code...
}
OS_END_TXN(use_case2)
```

Finding a Database Root

```
os_database* db = os_database::open("d:/x.odb");
static char* buf = "Foo Root";

OS_BEGIN_TXN(use_case2, 0, os_transaction:
{
    Foo* f = 0;
    os_database_root* root = db->find_root(buf);
    if(root)
    {
        f = (Foo*) root->get_value();
    }

    //...other code...
}
OS_END_TXN(use_case2)
```

Use the same root name…

Finding a Database Root

```
os_database* db = os_database::open("d:/x.odb");
static char* buf = "Foo Root";

OS_BEGIN_TXN(use_case2, 0, os_transaction::read_only)
{
    Foo* f = 0;
    os_database_root* root = db->find_root(buf);
    if(root)
    {
        f = (Foo*) root->get_value();
    }

    //...other code...
}
OS_END_TXN(use_case2)
```

…find the root from the database pointer

Finding a Database Root

```
os_database* db = os_database::open("d:/x.odb");
static char* buf = "Foo Root";

OS_BEGIN_TXN(use_case2, 0, os_transaction::read_only)
{
    Foo* f = 0;
    os_database_root* root = db->find_root(buf);
    if(root)
    {
        f = (Foo*)
    }

    //...other code...
}
OS_END_TXN(use_case2)
```

…check for null before we use it…

Finding a Database Root

```
os_database* db = os_database::open("d:/x.odb");
static char* buf = "Foo Root";

OS_BEGIN_TXN(use_case2, 0, os_transaction::read_only)
{
    Foo* f = 0;
    os_database_root* root = db->find_root(buf);
    if(root)
    {
        f = (Foo*) root->get_value();
    }

    //...other code...
}
OS_END_TXN(use_case2)
```

…get a pointer to the initial object…

Finding a Database Root

```
os_database* db = os_database::open("d:/x.odb");
static char* buf = "Foo Root";

OS_BEGIN_TXN(use_case2, 0, os_transaction::read_only)
{
    Foo* f = 0;
    os_database_root* root = db->find_root(buf);
    if(root)
    {
        f = (Foo*) root->get_value();
    }

    //...other code...
}
OS_END_TXN(use_case2)
```

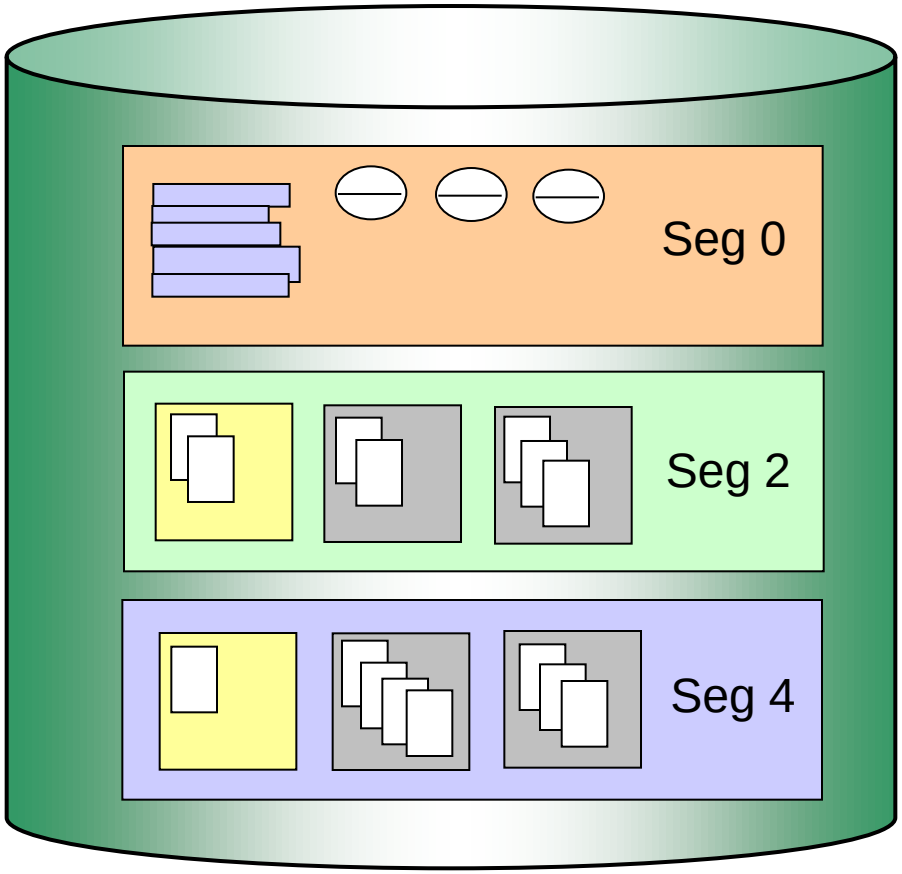
…and cast to the correct type

Agenda

- What is ObjectStore?
- ObjectStore Components
- Fetching and Mapping a Page
- ObjectStore Programming
- Opening and Closing a database
- Using Transactions
- Creating and Deleting Persistent Objects
- Finding Initial Persistent Objects
- Segments and Clusters
- Writing and Using a Persistent Class
- Conclusion

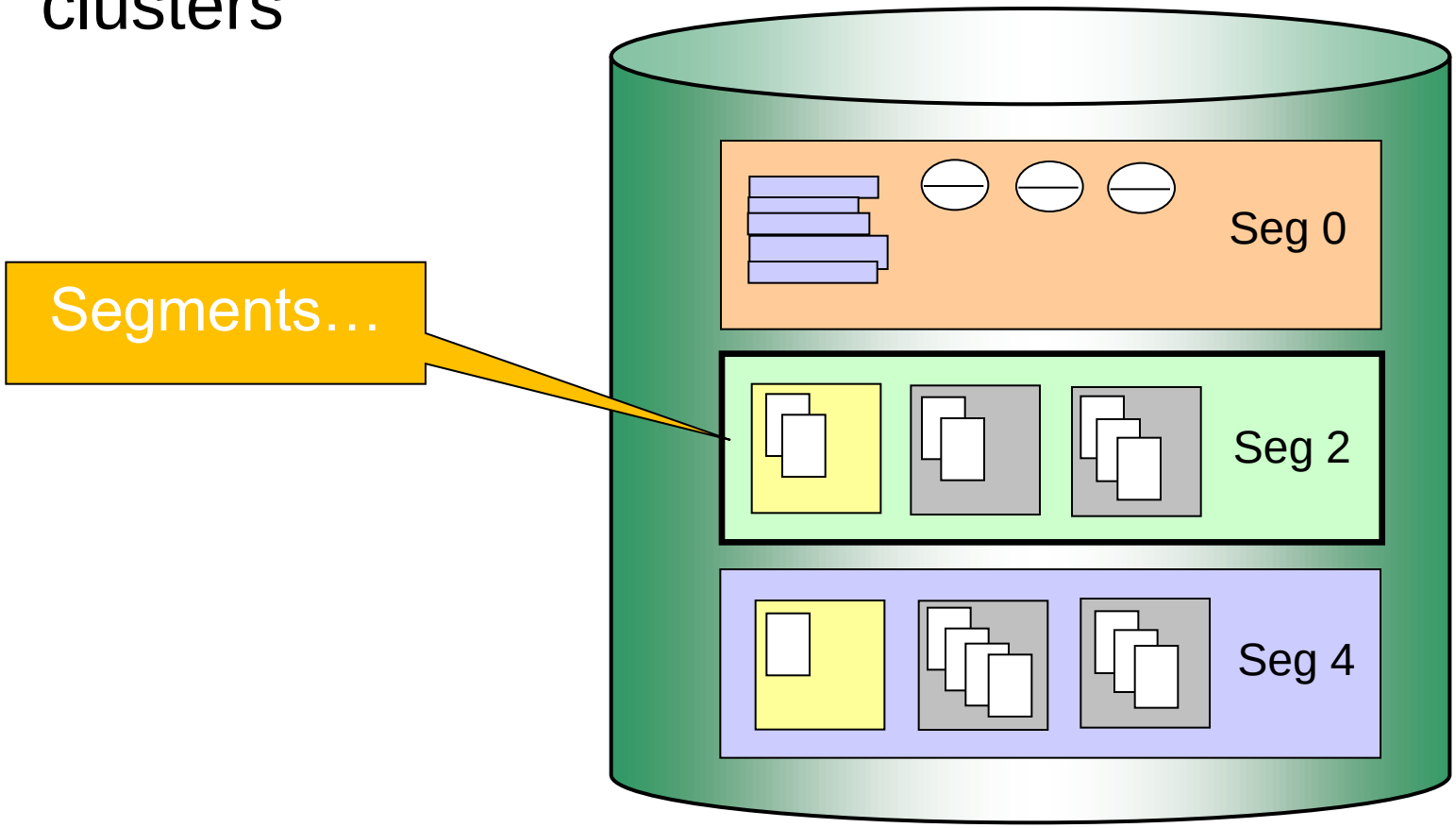
Segments & Clusters

- ObjectStore databases contain pages of memory held in a hierarchy of segments and clusters



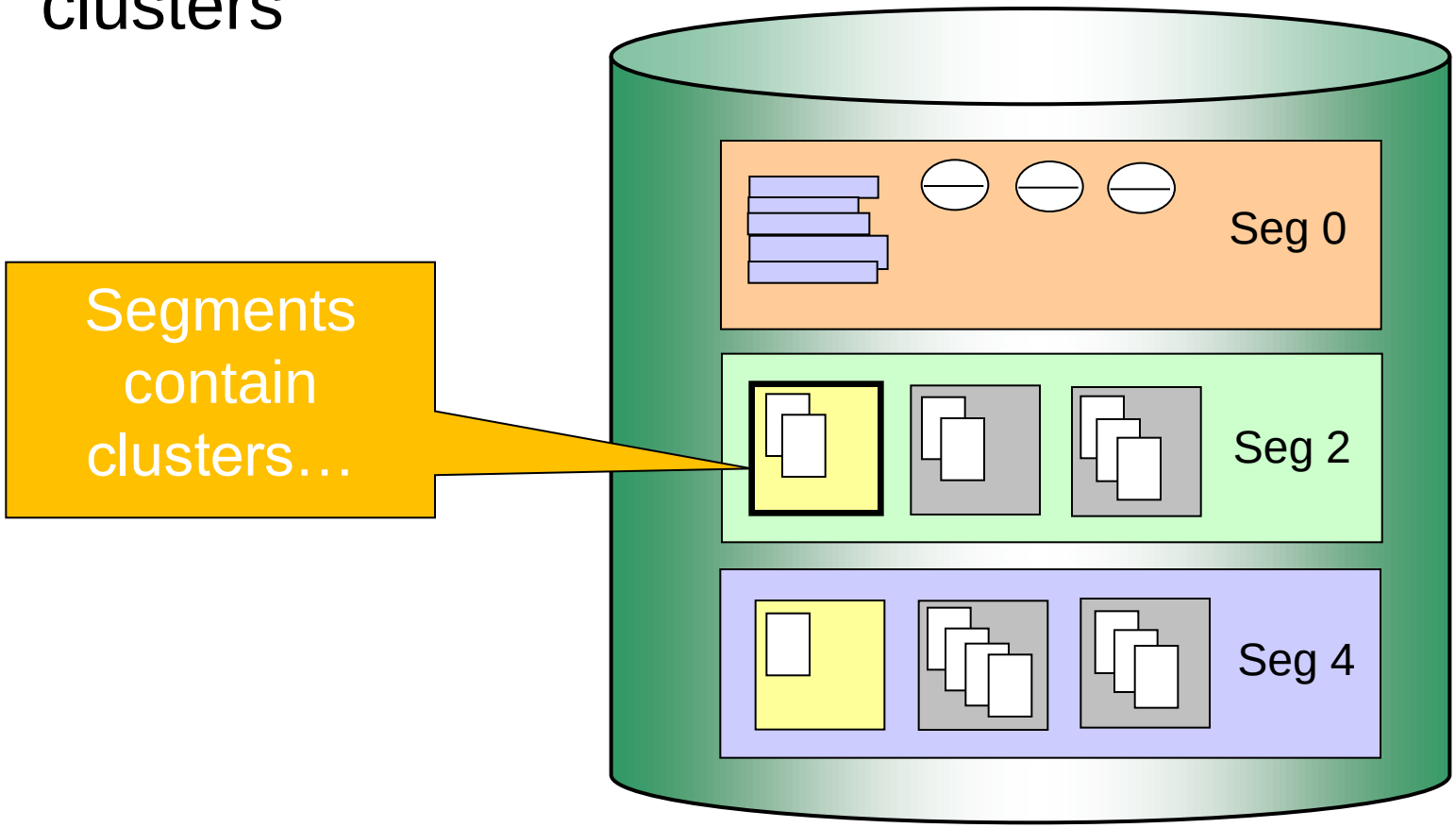
Segments & Clusters

- ObjectStore databases contain pages of memory held in a hierarchy of segments and clusters



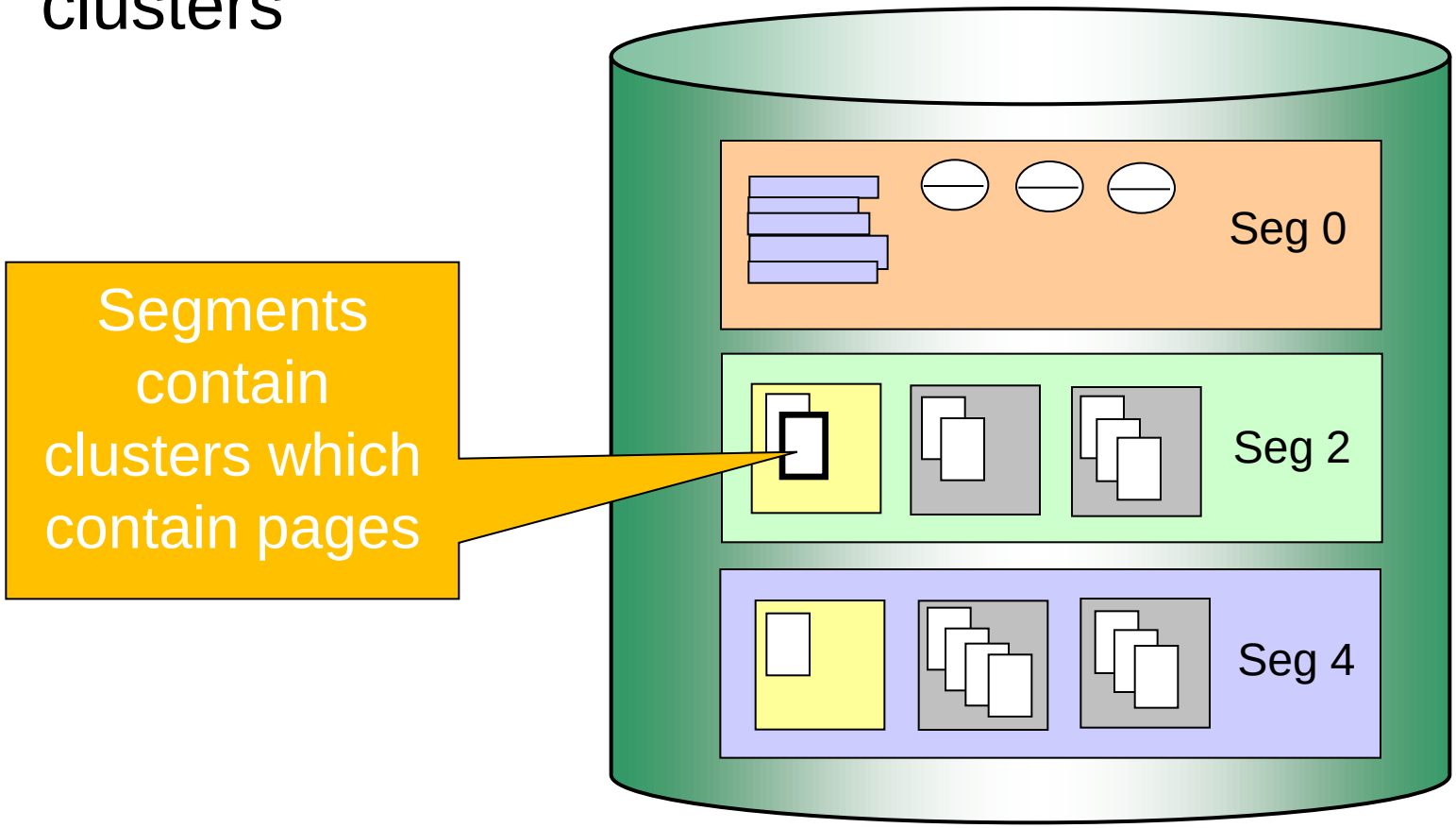
Segments & Clusters

- ObjectStore databases contain pages of memory held in a hierarchy of segments and clusters



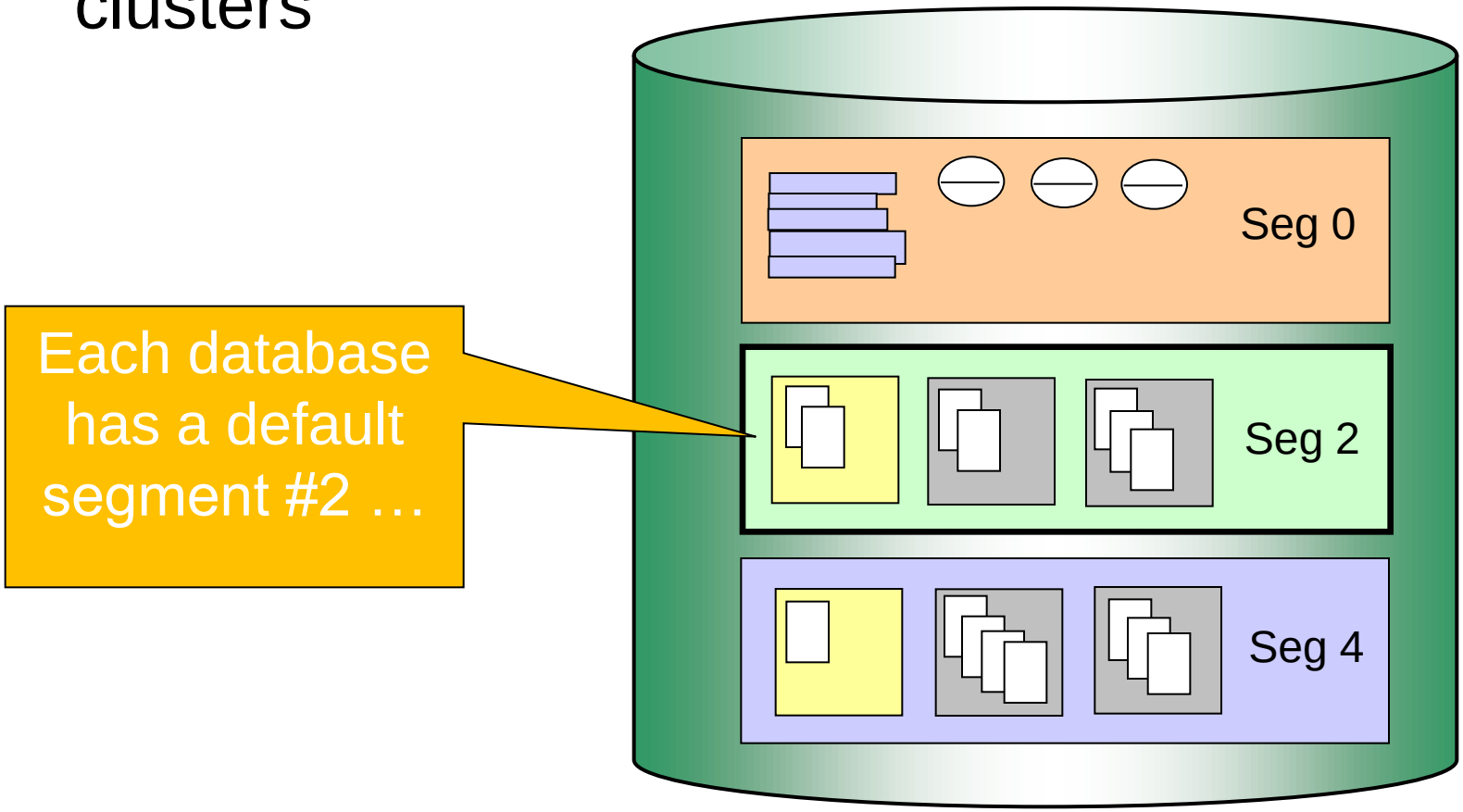
Segments & Clusters

- ObjectStore databases contain pages of memory held in a hierarchy of segments and clusters



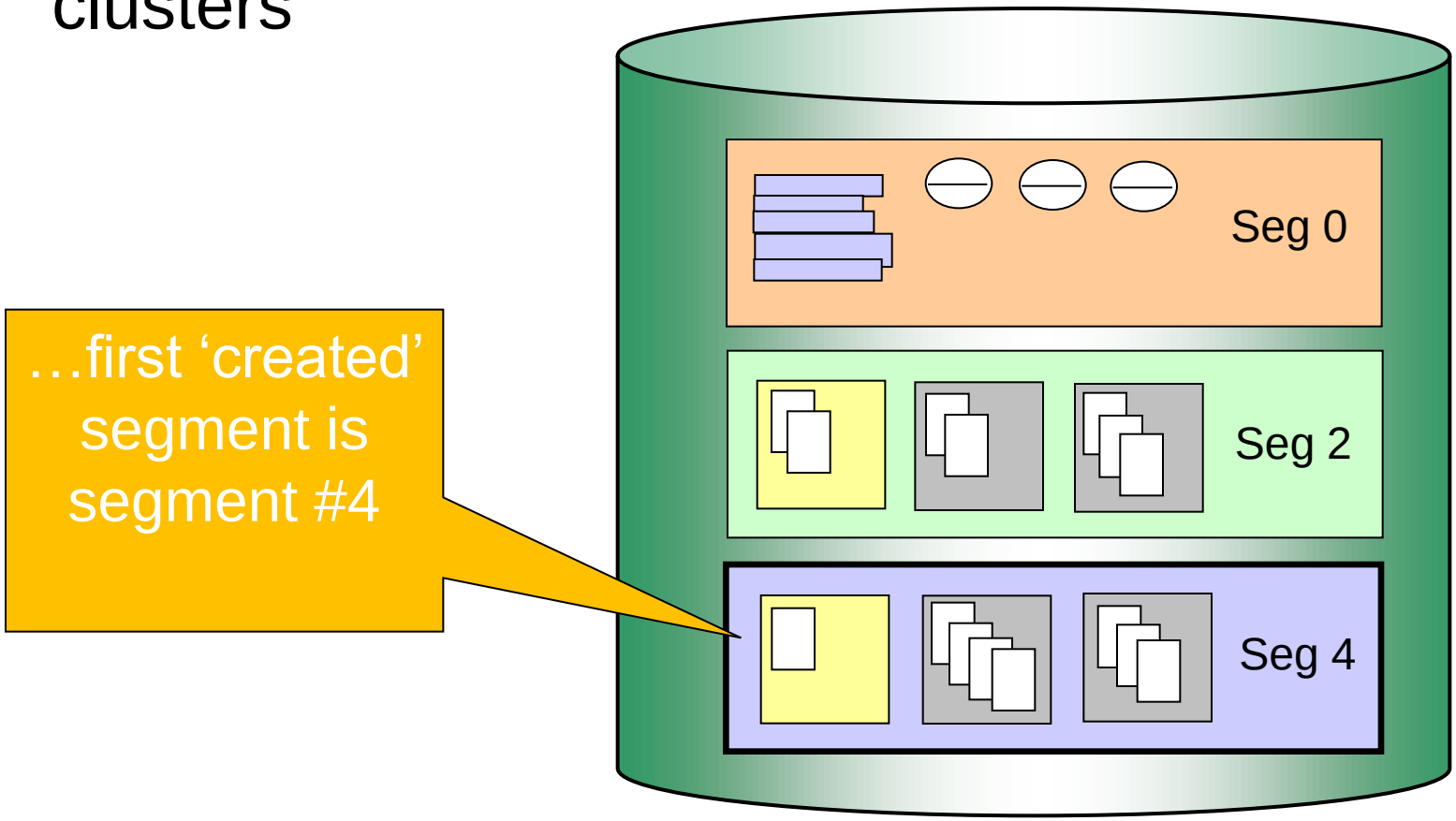
Segments & Clusters

- ObjectStore databases contain pages of memory held in a hierarchy of segments and clusters



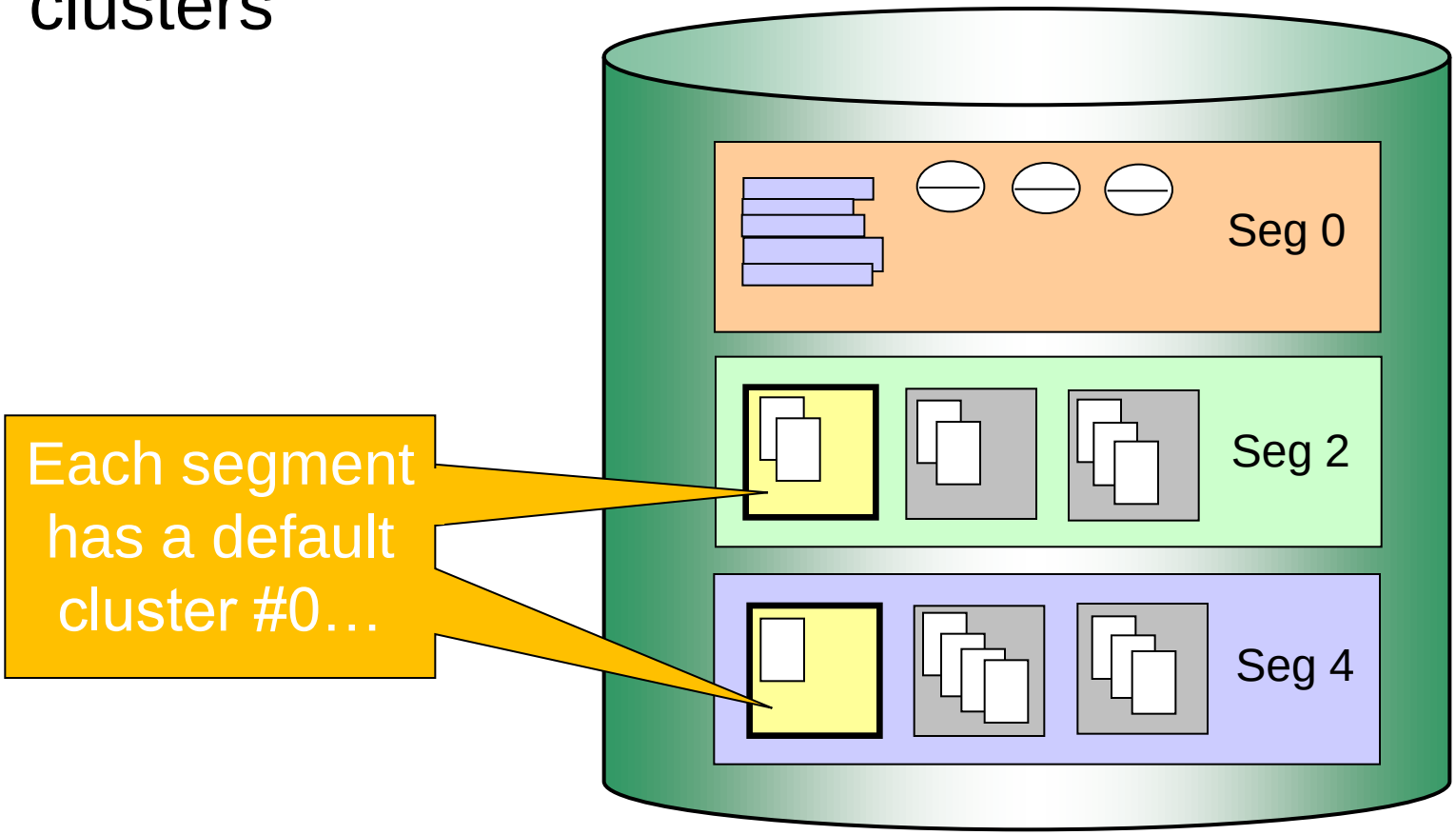
Segments & Clusters

- ObjectStore databases contain pages of memory held in a hierarchy of segments and clusters



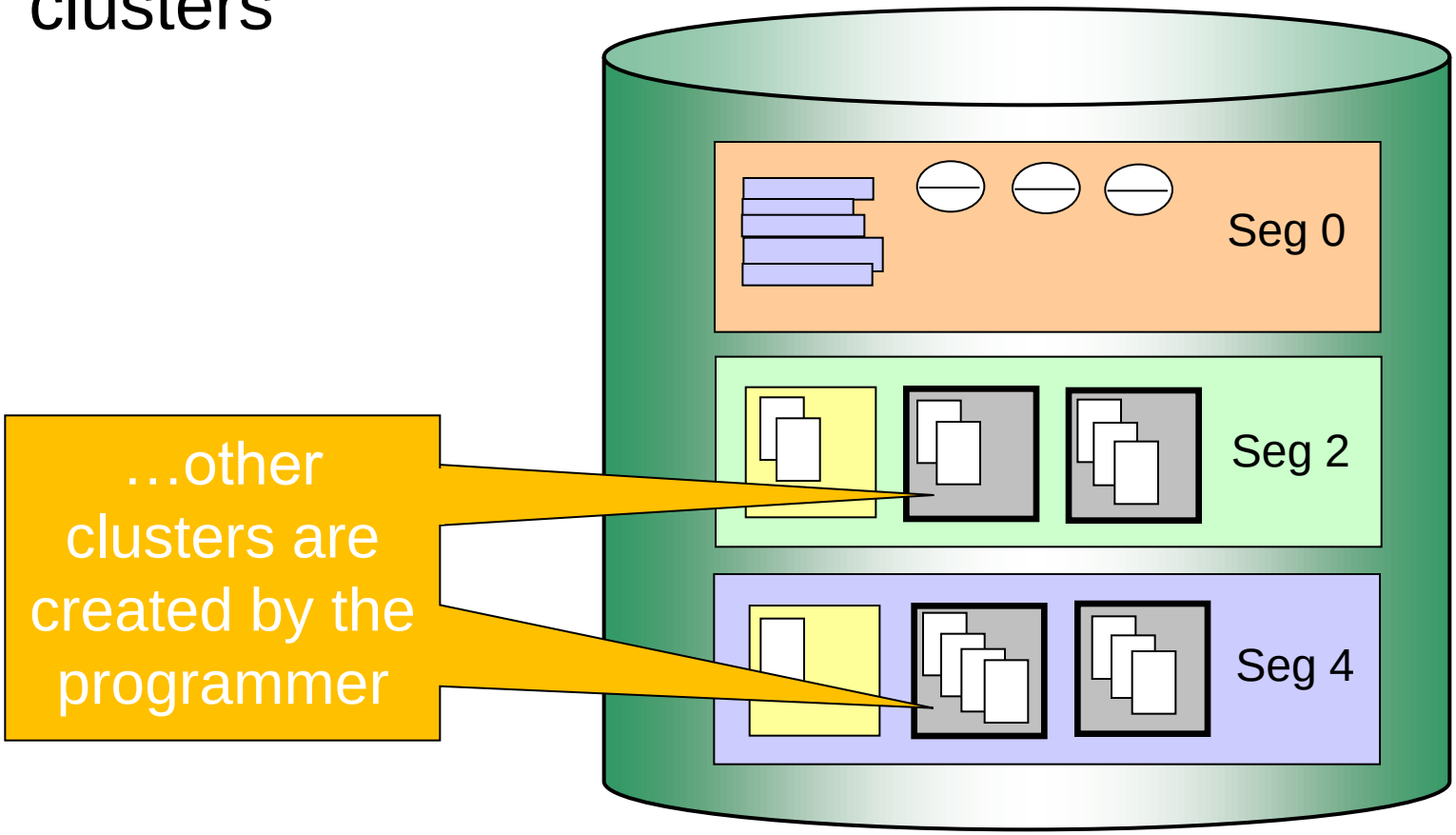
Segments & Clusters

- ObjectStore databases contain pages of memory held in a hierarchy of segments and clusters



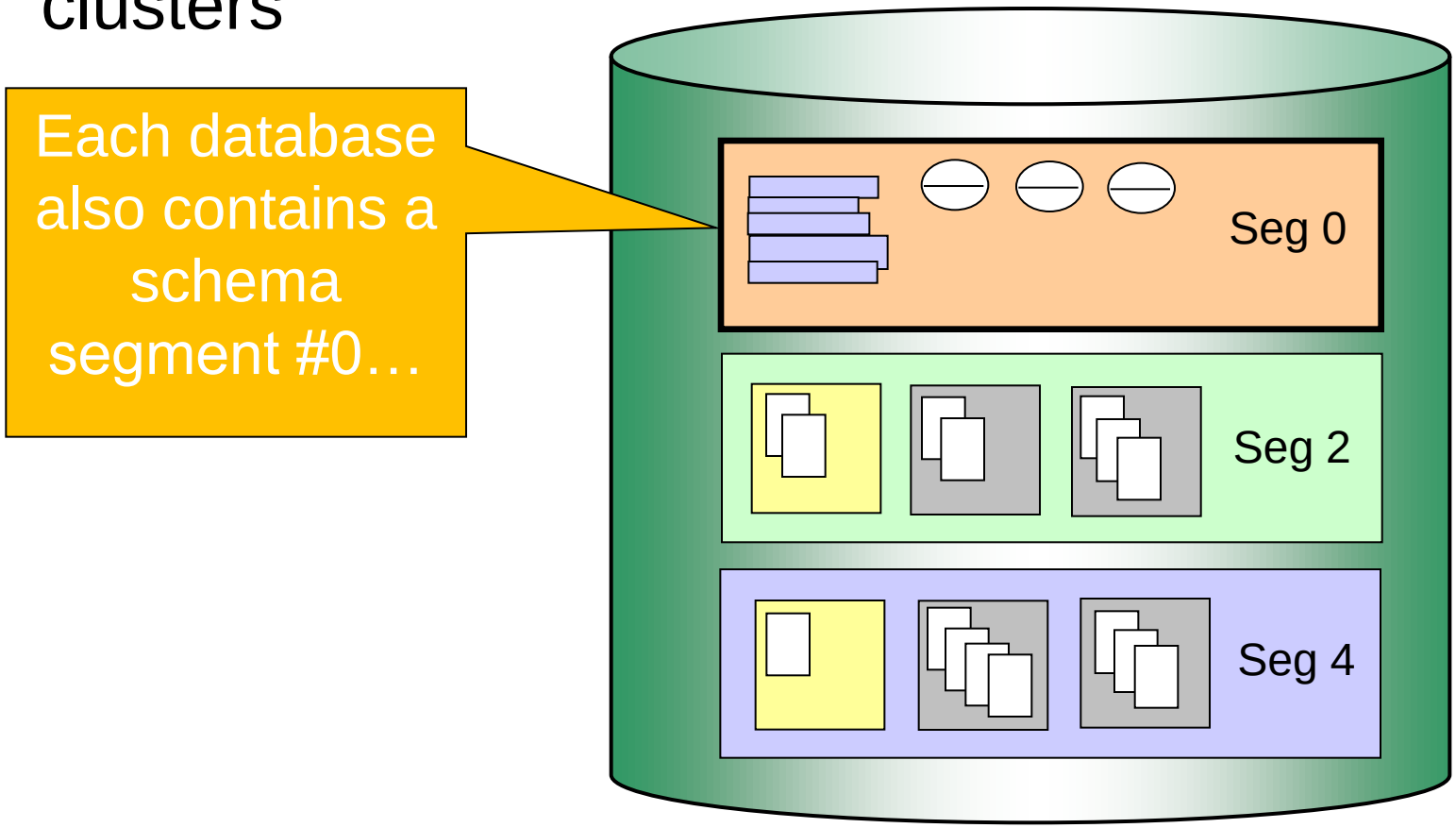
Segments & Clusters

- ObjectStore databases contain pages of memory held in a hierarchy of segments and clusters



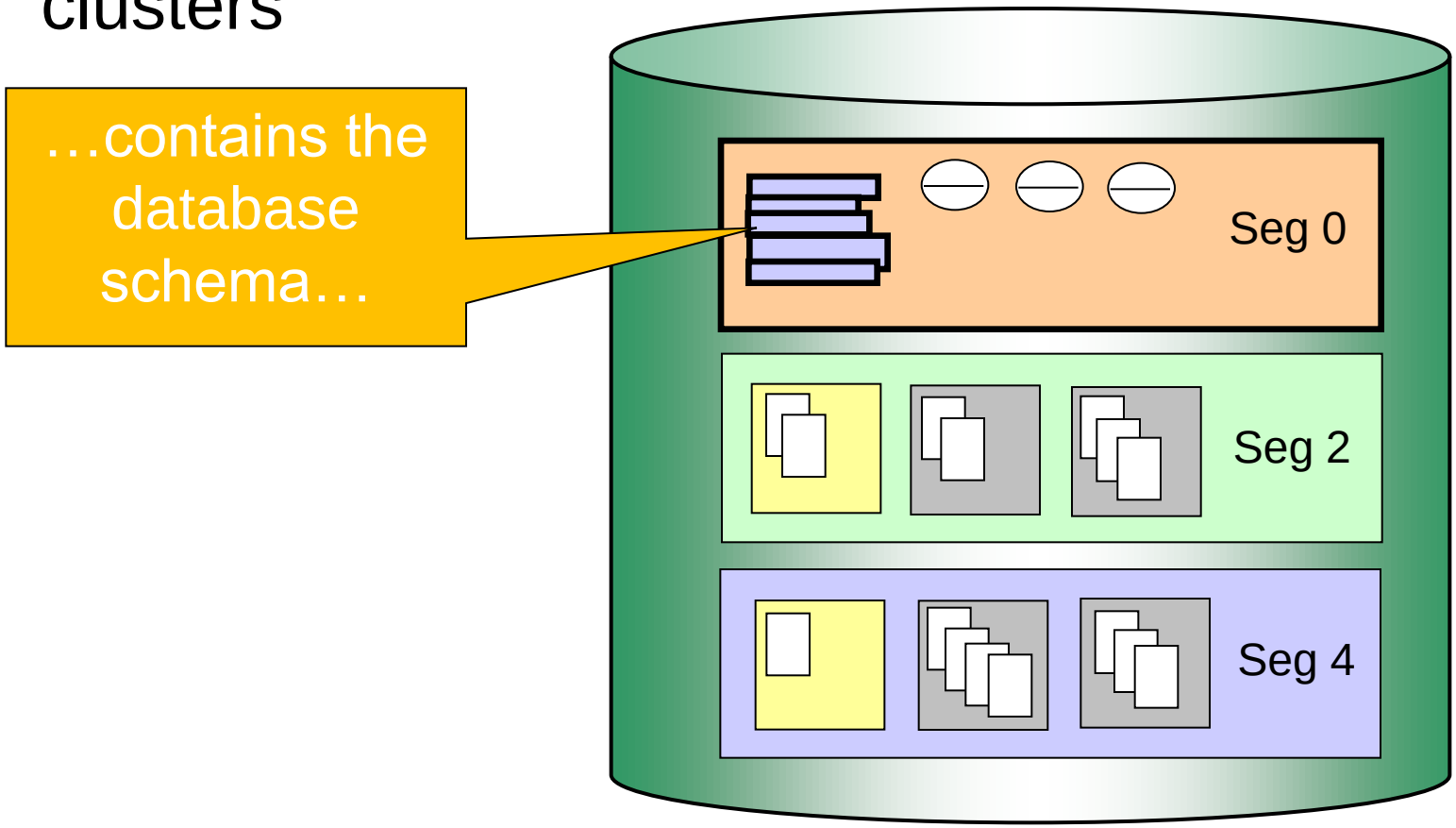
Segments & Clusters

- ObjectStore databases contain pages of memory held in a hierarchy of segments and clusters



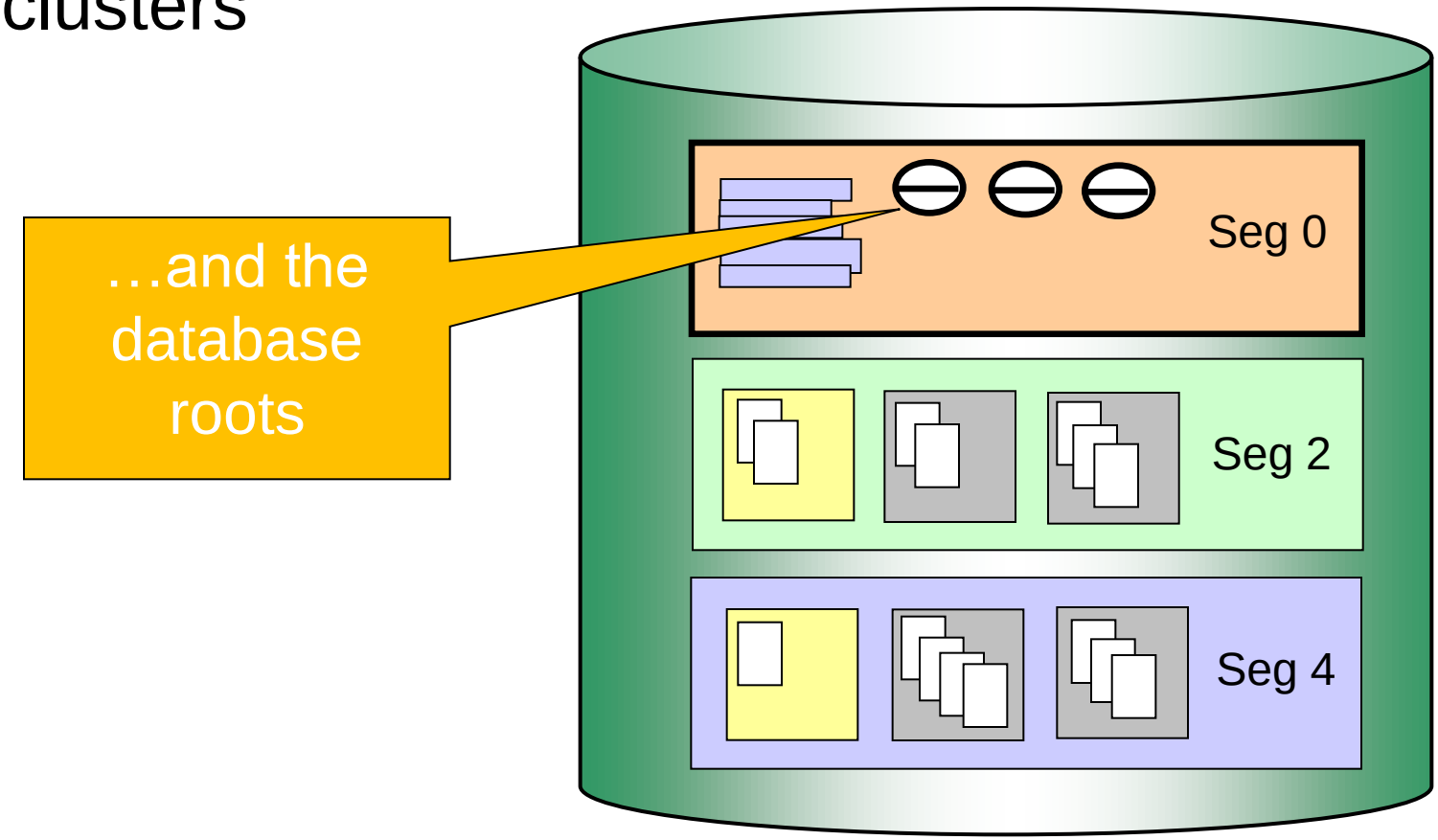
Segments & Clusters

- ObjectStore databases contain pages of memory held in a hierarchy of segments and clusters



Segments & Clusters

- ObjectStore databases contain pages of memory held in a hierarchy of segments and clusters



Segment Creation

```
os_database* db = os_database::open("d:/x.odb");

OS_BEGIN_TXN(use_case1, 0, os_transaction::update)
{
   os_segment* seg = db->create_segment();
   seg->set_comment( "Foo Segment" );
   Foo* f = new(seg, ts<Foo>()) Foo();

   // ...other code ...

}
OS_END_TXN(use_case1)

//... other code ...
```

Segment Creation

```
os_database* db = os_database::open("d:/x.odb");

OS_BEGIN_TXN(use_case1, 0, os_transaction::update)
{
    os_segment* seg = db->create_segment();
    seg->set_comment( "Foo Segment" );
    Foo* f = new(seg, ts<Foo>()) Foo();

    // ...other code ...

}
OS_END_TXN(use_case1)

//... other code ...
```

Segment created using database pointer…

Segment Creation

```
os_database* db = os_database::open("d:/x.odb");

OS_BEGIN_TXN(use_case1, 0, os_transaction::update)
{
    os_segment* seg = db->create_segment();
    seg->set_comment( "Foo Segment" );
    Foo* f = new(seg, ts<Foo>()) Foo();

    // ...other code ...

}
OS_END_TXN(use_case1)

//... other code ...
```

…optionally set a comment

Segment Creation

```
os_database* db = os_database::open("d:/x.odb");

OS_BEGIN_TXN(use_case1, 0, os_transaction::update)
{
    os_segment* seg = db->create_segment();
    seg->set_comment( "Foo Segment" );
    Foo* f = new(seg, ts<Foo>()) Foo();

    // ...other code ...

}
OS_END_TXN(use_case1)

//... other code ...
```

…create an object in the segment.

Creating Clusters

```
os_database* db = os_database::open("d:/x.odb");

OS_BEGIN_TXN(use_case1, 0, os_transaction::update)
{
   os_segment* seg = db->create_segment();
   seg->set_comment( "Foo Segment" );

   os_cluster* clr = seg->create_cluster();
   Foo* f = new(clr, ts<Foo>()) Foo();

   // ...other code ...
}
OS_END_TXN(use_case1)

//... other code ...
```

Creating Clusters

```
os_database* db = os_database::open("d:/x.odb");

OS_BEGIN_TXN(use_case1, 0, os_transaction::update)
{
   os_segment* seg = db->create_segment();
   seg->set_comment( "Foo Segment" );

   os_cluster* clr = seg->create_cluster();
   Foo* f = new(clr, ts<Foo>()) Foo();

   // ...other code ...
}
OS_END_TXN(use_case1)

//... other code ...
```

Cluster created in a segment…

Creating Clusters

```
os_database* db = os_database::open("d:/x.odb");

OS_BEGIN_TXN(use_case1, 0, os_transaction::update)
{
   os_segment* seg = db->create_segment();
   seg->set_comment( "Foo Segment" );

   os_cluster* clr = seg->create_cluster();
   Foo* f = new(clr, ts<Foo>()) Foo();

   // ...other code ..
}
OS_END_TXN(use_case1)

//... other code ...
```

…object allocated in the cluster

Using Segments & Clusters

```
OS_BEGIN_TXN(use_case1, 0, os_transaction::update)
{
   // ...other code ...

   Foo* f = //...
   os_segment* seg = os_segment::of(f);
   cout << "Seg# = " << seg->get_number();

   os_cluster* clr = os_cluster::of(f);
   Foo* f2 = new(clr, ts<Foo>()) Foo();

   // ...other code ...
}
OS_END_TXN(use_case1)

//... other code ...
```

Using Segments & Clusters

```
OS_BEGIN_TXN(use_case1, 0, os_transac
{
   // ...other code ...

   Foo* f = //...
   os_segment* seg = os_segment::of(f);
   cout << "Seg# = " << seg->get_number();

   os_cluster* clr = os_cluster::of(f);
   Foo* f2 = new(clr, ts<Foo>()) Foo();

   // ...other code ...
}
OS_END_TXN(use_case1)

//... other code ...
```

Foo object read from database earlier…

Using Segments & Clusters

```
OS_BEGIN_TXN(use_case1, 0, os_transac
{
   // ...other code ...

   Foo* f = //...
   os_segment* seg = os_segment::of(f);
   cout << "Seg# = " << seg->get_number();

   os_cluster* clr = os_cluster::of(f);
   Foo* f2 = new(clr, ts<Foo>()) Foo();

   // ...other code ...
}
OS_END_TXN(use_case1)

//... other code ...
```

Get pointer to segment containing Foo

Using Segments & Clusters

```
OS_BEGIN_TXN(use_case1, 0, os_transact
{
   // ...other code ...

   Foo* f = //...
   os_segment* seg = os_segment::of(f);
   cout << "Seg# = " << seg->get_number();

   os_cluster* clr = os_cluster::of(f);
   Foo* f2 = new(clr, ts<Foo>()) Foo();

   // ...other code ...
}
OS_END_TXN(use_case1)

//... other code ...
```

Print out its segment number

Using Segments & Clusters

```
OS_BEGIN_TXN(use_case1, 0, os_transaction::update)
{
   // ...other code ...

   Foo* f = //...
   os_segment* seg = os_segment::of(f);
   cout << "Seg# = " << seg->get_number();

   os_cluster* clr = os_cluster::of(f);
   Foo* f2 = new(clr, ts<Foo>()) Foo();

   // ...other code ...
}
OS_END_TXN(use_case1)

//... other code ...
```

Get pointer to cluster containing Foo

Using Segments & Clusters

```
OS_BEGIN_TXN(use_case1, 0, os_transaction::update)
{
    // ...other code ...

    Foo* f = //...
    os_segment* seg = os_segment::of(f);
    cout << "Seg# = " << seg->get_number();

    os_cluster* clr = os_cluster::of(f);
    Foo* f2 = new(clr, ts<Foo>()) Foo();

    // ...other code ...
}
OS_END_TXN(use_case1)

//... other code ...
```

Allocate another Foo into the same cluster

Agenda

- What is ObjectStore?
- ObjectStore Components
- Fetching and Mapping a Page
- ObjectStore Programming
- Opening and Closing a database
- Using Transactions
- Creating and Deleting Persistent Objects
- Finding Initial Persistent Objects
- Segments and Clusters
- Writing and Using a Persistent Class
- Conclusion

Writing a Persistent Class

- Now we examine
  - The steps necessary to write a persistent class
  - How this differs to a transient class
- We use a persistent `MyString` class by way of illustration

Writing a Persistent Class (.h)

```
class MyString
{
public:

    // Ctors.
    MyString();
    MyString(char* str);
    // Accessors
    char* getString() const{
        return _str;
    }
    void setString(char* str);

    friend ostream& operator<<
        (ostream& out, const MyString& obj);
private:
    char* _str;
};

// Forward declaration
ostream& operator<< (ostream& out, const MyString& obj);
```

Writing a Persistent Class (.h)

```
class MyString
{
public:

    // Ctors.
    MyString();
    MyString(char* str);
    // Accessors
    char* getString() const{
        return _str;
    }
    void setString(char* str);

    friend ostream& operator<<
        (ostream& out, const MyString& obj);
private:
    char* _str;
};

// Forward declaration
ostream& operator<< (ostream& out, const MyString& obj);
```

Identical to transient class

Writing a Persistent Class (.cpp)

```
MyString::MyString()
:_str(0)
{}

MyString::MyString(char* str)
:_str(0)
{
    setString(str);
}

ostream& operator<< (ostream& out, const MyString& obj)
{
    out << obj._str;
    return out;
}
```

Writing a Persistent Class (.cpp)

```
MyString::MyString()
:_str(0)
{}

MyString::MyString(char* str)
:_str(0)
{
    setString(str);
}

ostream& operator<< (ostream& out, const MyString& obj)
{
    out << obj._str;
    return out;
}
```

Identical to transient class

Writing a Persistent Class (.cpp)

```
void MyString::setString(char* str)
{
    delete [] _str;
    _str = 0;
    if(str)
    {
        int len = strlen(str)+1;
        _str = new
            (os_cluster::of(this), ts<char>(), len)
            char[len];
        strcpy(_str, str);
        _str[len]=0;
    }
}
```

Writing a Persistent Class (.cpp)

```
void MyString::setString(char* str)
{
    delete [] _str;
    _str = 0;
    if(str)
    {
        int len = strlen(str)+1;
        _str = new
            (os_cluster::of(this), ts<char>(), len)
            char[len];
        strcpy(_str, str);
        _str[len]=0;
    }
}
```

Use ObjectStore new to allocate the character array…

Writing a Persistent Class (.cpp)

```
void MyString::setString(char* str)
{
    delete [] _str;
    _str = 0;
    if(str)
    {
        int len = strlen(str)+1;
        _str = new
            (os_cluster::of(this), ts<char>(), len)
            char[len];
        strcpy(_str, str);
        _str[len]=0;
    }
}
```

Returns the cluster of MyString instance…

… this is the transient cluster if instance is allocated on stack or heap

Writing a Persistent Class (building)

- Indicate that `MyString` class will be used persistently in a separate ‘schema file’

```
#include <ostore\manschem.hh>
#include <ostore\ostore.hh>

#include "MyString.h"
OS_MARK_SCHEMA_TYPE(MyString)
```

Writing a Persistent Class (building)

- Indicate that `MyString` class will be used persistently in the schema file

```
#include <ostore\manschem.hh>
#include <ostore\ostore.hh>

#include "MyString.h"
OS_MARK_SCHEMA_TYPE(MyString)
```

Schema file contains a ‘list’ of all your persistent classes. Called ‘schema.osg’ by convention

Writing a Persistent Class (building)

- Indicate that `MyString` class will be used persistently in the schema file

```
#include <ostore\manschem.hh>
#include <ostore\ostore.hh>

#include "MyString.h"
OS_MARK_SCHEMA_TYPE(MyString)
```

- Compile `schema.osg` file to a `.cpp` file using the ObjectStore schema generator

```
ossg schema.osg ...
```

Writing a Persistent Class (building)

- Indicate that `MyString` class will be used persistently in the schema file

```
#include <ostore\manschem.hh>
#include <ostore\ostore.hh>

#include "MyString.h"
OS_MARK_SCHEMA_TYPE(MyString)
```

- Compile `schema.osg` file to a `.cpp` file using the ObjectStore schema ge

```
ossg schema.osg ...
```

Your header files are parsed by ossg. Your header files ***define*** the persistent schema

Writing a Persistent Class (building)

- Indicate that `MyString` class will be used persistently in the schema file

```
#include <ostore\manschem.hh>
#include <ostore\ostore.hh>

#include "MyString.h"
OS_MARK_SCHEMA_TYPE(MyString)
```

- Compile `schema.osg` file to a `.cpp` file using the ObjectStore schema generator

```
ossg schema.osg ...
```

- Then compile and link application as normal

Using the Persistent Class

```
MyString s1( "This is on the stack" );
cout << "s1 = " << s1 << std::endl;

OS_BEGIN_TXN(txn1, 0, os_transaction::update)
{
        MyString* s2 = new (db, ts<MyString>())
                MyString( "This is in the database" );
        db->create_root( "MyString Root" )->set_value(s2);
}
OS_END_TXN(txn1)

MyString* s3 = new MyString( "This is on the heap" );
cout << "s3 = " << *s3 << std::endl;

OS_BEGIN_TXN(txn2, 0, os_transaction::read_only)
{
        os_database_root* root = db->find_root( "MyString Root" );
        MyString* s4 = (MyString*) root->get_value();
        cout << "Retrieved: s4 = " << *s4 << std::endl;
}
OS_END_TXN(txn2)
```

Using the Persistent Class

- The output showing the same class used on the stack, the heap and in the database

```
s1 = This is on the stack
s3 = This is on the heap
Retrieved: s4 = This is in the database
```

Agenda

- What is ObjectStore?
- ObjectStore Components
- Fetching and Mapping a Page
- ObjectStore Programming
- Opening and Closing a database
- Using Transactions
- Creating and Deleting Persistent Objects
- Finding Initial Persistent Objects
- Segments and Clusters
- Writing and Using a Persistent Class
- Conclusion

Conclusion

- Presented an overview of ObjectStore and introduced its various components
- Looked at the page fetch mechanism and its similarity to virtual memory
- Covered ObjectStore coding fundamentals
  - How to open and close databases
  - How to use ObjectStore transactions
  - How to create and delete persistent objects, and find them initially using DB roots
  - How to create and use segments and clusters
  - How to create your own persistent classes

Conclusion

- ObjectStore is best conceptualized as ***"a persistent, transactional, heap that can be shared between many programs"***
- It is *language transparent*; programmers just code against the C++ library
- Developers can write bespoke index structures to optimally support their particular use-cases
- When properly implemented ObjectStore based systems can run *very, very* fast!!!!

For More Information

- Papers
  - Under **web.progress.com/docs/whitepapers/public/**
    - **/objectstore/objectstore_db_architecture.pdf**
    - **/objectstore/persistent_oodb_patterns.pdf**
  - Under **web.progress.com/docs/media-coverage/**
    - **bespokeprog.pdf**
- Contact Us
  - Adrian Marriott, Independent Consultant
    - adrian@logos-software.co.uk
  - Jeff Wagner, Product Manager
    - jwagner@progress.com

www.progress.com/objectstore

PROGRESS
SOFTWARE

Questions ?