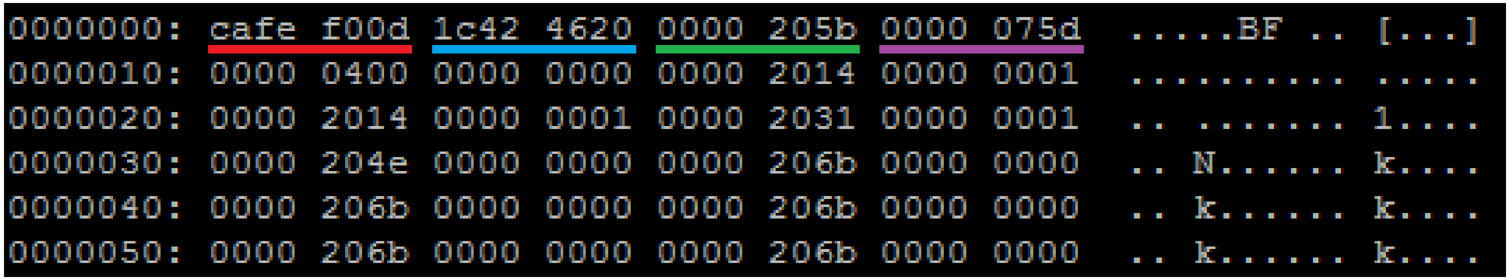
CSE333 SECTION 7

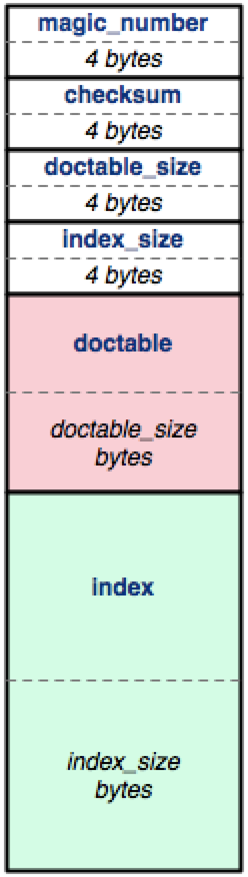
Midterm Debrief

Hex View

1. Find a hex editor.
2. Learn 'goto offset' command.
3. See HW3 pictures.



```
0000000: cafe f00d 1c42 4620 0000 205b 0000 075d  .....BF .. [...]
0000010: 0000 0400 0000 0000 0000 2014 0000 0001  .......... .....
0000020: 0000 2014 0000 0001 0000 2031 0000 0001  .. ....... 1....
0000030: 0000 204e 0000 0000 0000 206b 0000 0000  .. N...... k....
0000040: 0000 206b 0000 0000 0000 206b 0000 0000  .. k...... k....
0000050: 0000 206b 0000 0000 0000 206b 0000 0000  .. k...... k....
```



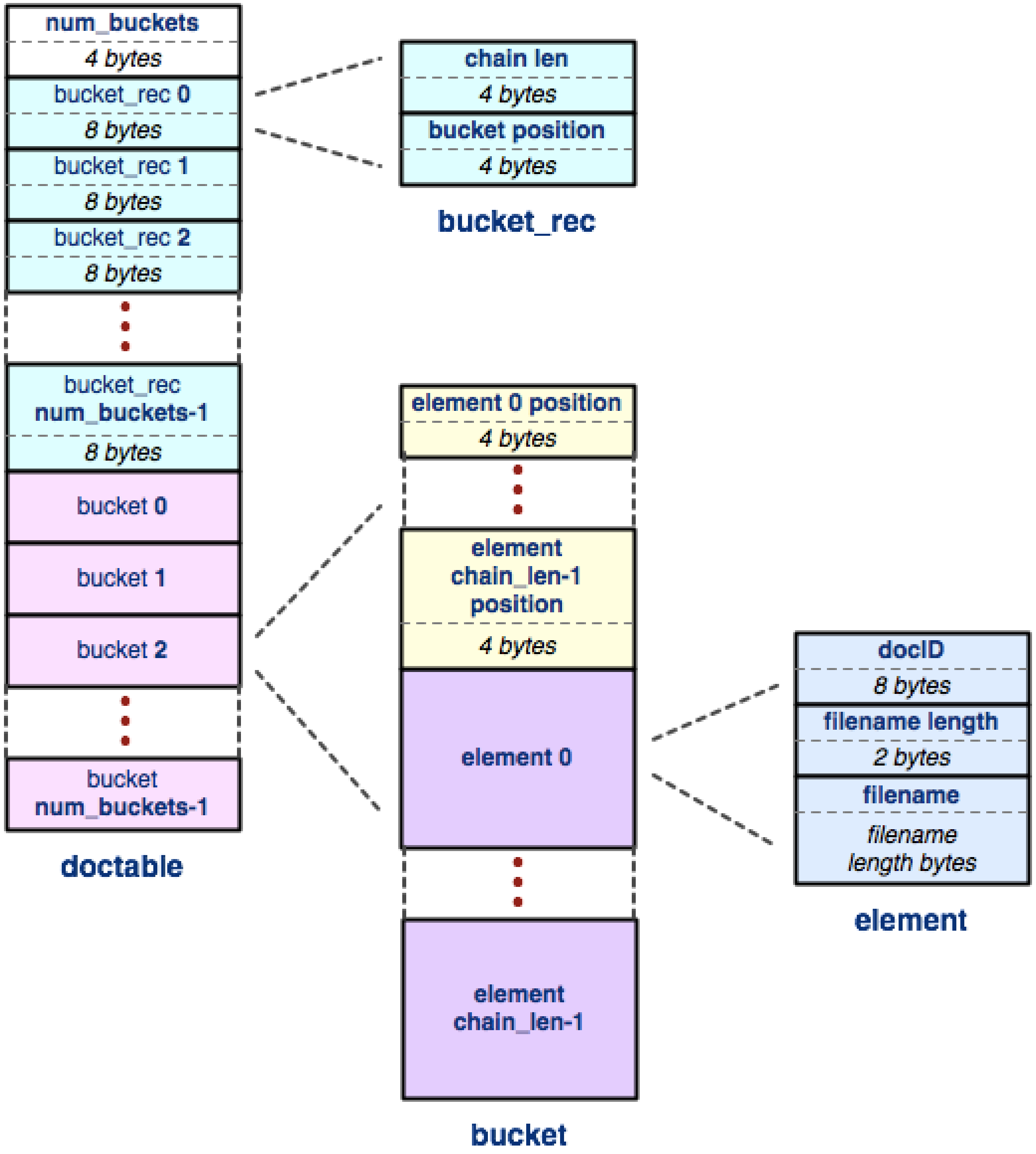
magic_number
4 bytes
checksum
4 bytes
doctable_size
4 bytes
index_size
4 bytes
doctable
doctable_size bytes
index
index_size bytes

index file

The header:

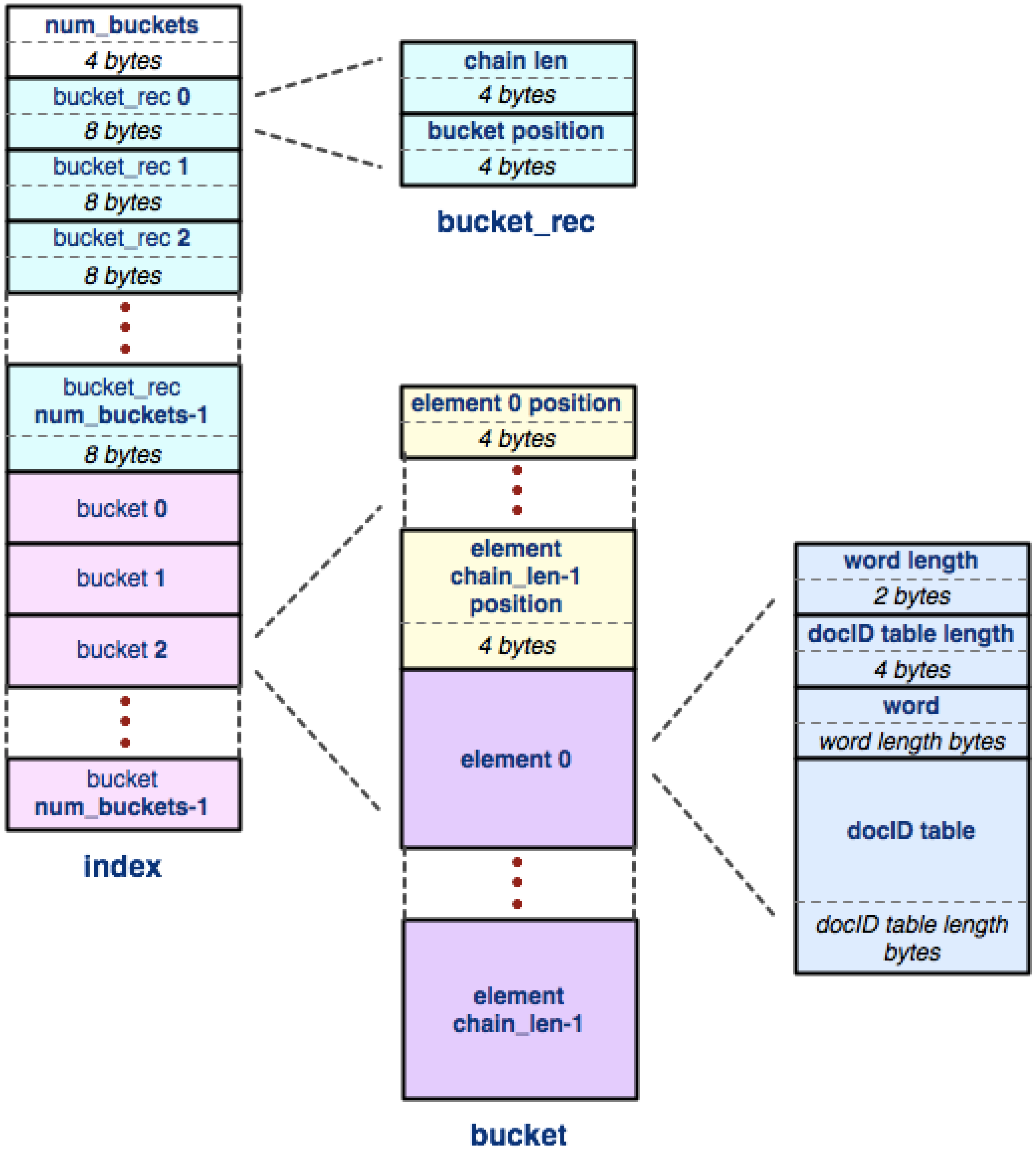
Magic word   Checksum   Doctable size   Index size

Hex View



The doctable

Hex View



num_buckets
4 bytes
bucket_rec 0
8 bytes
bucket_rec 1
8 bytes
bucket_rec 2
8 bytes
bucket_rec num_buckets-1
8 bytes
bucket 0
bucket 1
bucket 2
bucket num_buckets-1

index

chain len
4 bytes
bucket position
4 bytes

bucket_rec

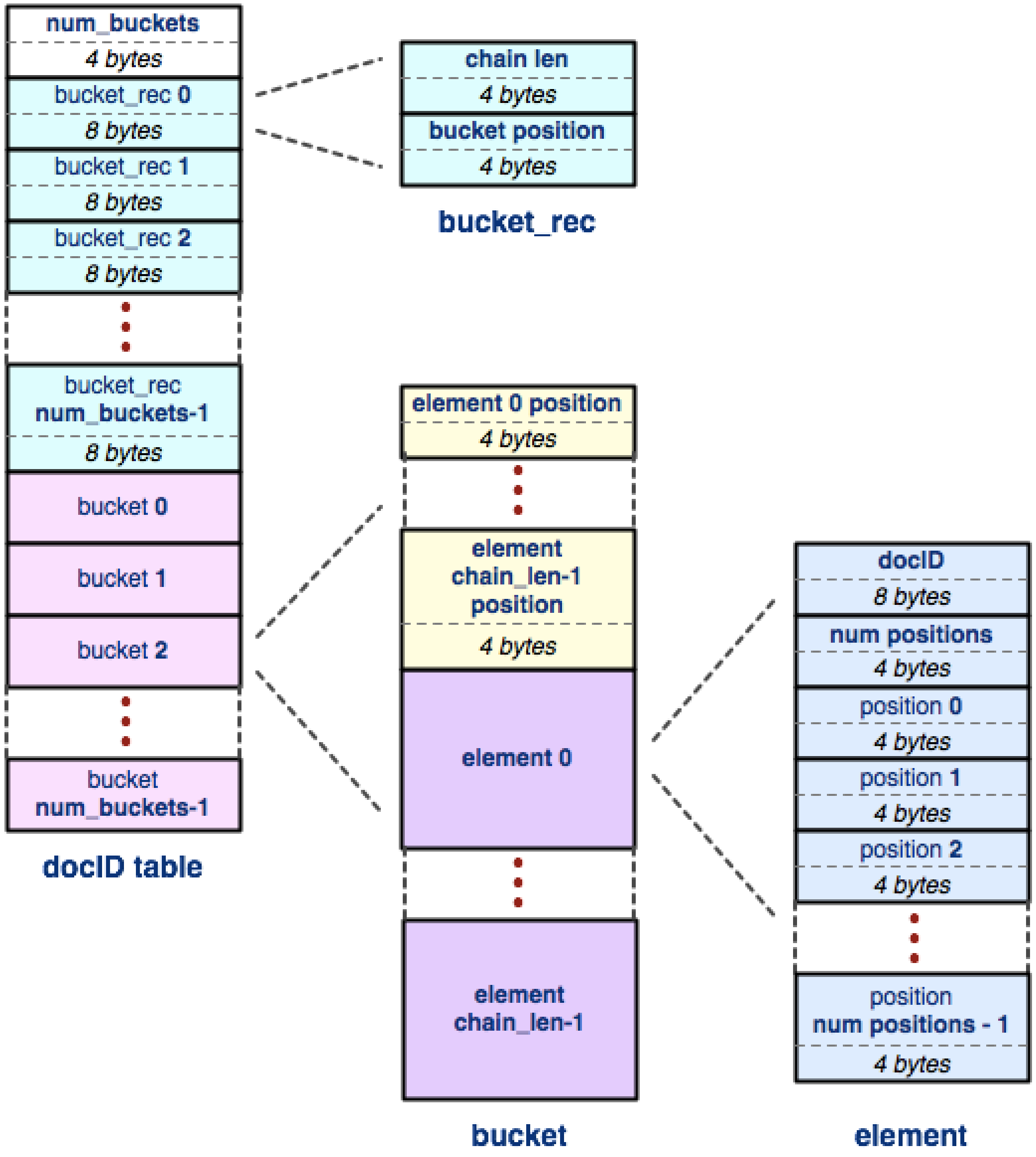
element 0 position
4 bytes
element chain_len-1 position
4 bytes
element 0
element chain_len-1

bucket

word length
2 bytes
docID table length
4 bytes
word
word length bytes
docID table
docID table length bytes

The index

Hex View



num_buckets
4 bytes
bucket_rec 0
8 bytes
bucket_rec 1
8 bytes
bucket_rec 2
8 bytes
bucket_rec num_buckets-1
8 bytes
bucket 0
bucket 1
bucket 2
bucket num_buckets-1

docID table

chain len
4 bytes
bucket position
4 bytes

bucket_rec

element 0 position
4 bytes
element chain_len-1 position
4 bytes
element 0
element chain_len-1

bucket

docID
8 bytes
num positions
4 bytes
position 0
4 bytes
position 1
4 bytes
position 2
4 bytes
position num positions - 1
4 bytes

element

The docID table

Templates

```
class IntPair {
public:
  IntPair(const int first, const int second)
    : first_(first), second_(second) { }
  int first() const { return first_; }
  int second() const { return second_; }
private:
  int first_;
  int second_;
};

class DoublePair {
public:
  DoublePair(const double first, const double second)
    : first_(first), second_(second) { }
  double first() const { return first_; }
  double second() const { return second_; }
private:
  double first_;
  double second_;
};
```

Templates

```
class FooPair {
public:
  FooPair(const Foo& first, const Foo& second)
    : first_(first), second_(second) { }
  Foo first() const { return first_; }
  Foo second() const { return second_; }
private:
  Foo first_;
  Foo second_;
};
```

- This is really repetitive!

Templates

```
template <typename T>
class Pair {
public:
  Pair(const T& first, const T& second)
    : first_(first), second_(second) { }
  T first() const { return first_; }
  T second() const { return second_; }
private:
  T first_;
  T second_;
};
```

Templates

- Functions can be templated too
- Each “type” of template class/function generates distinct code
  - `Pair<int>` and `Pair<Foo>` are two distinct classes with code located in two distinct regions of the binary
- Templates are generated at compile time
  - Compiler needs to know how each template will be used
  - Full definitions of templated code must be included in translation unit

Standard Template Library

- C++ comes with a rich set of templated collections
  - cplusplus.com
  - cppreference.com
- All collections pass by value (copy), *not* by reference
- Automatic resizing of a collection can trigger multiple copy operations
  - One way to make this more efficient: move semantics
    - Outside the scope of this class, but ask Sunjay about it any time
  - Another way to avoid this: pass in pointers to data
    - Memory management gets messy
      - Use smart pointers!

Smart Pointers

- Encapsulate memory management through ctors/dtors
- Wraps a “normal” pointer
- Automatically calls delete when lifetime is over
- Three types:
    - unique_ptr ensures only one pointer to underlying data
        - Does this by disallowing copy construction/assignment
        - You can still use it in STL containers though (move semantics!)
    - shared_ptr keeps a reference count
        - Only deletes wrapped pointer when reference count hits zero
    - weak_ptr does not contribute to the reference count
        - Think circular linked lists, you’d want a weak_ptr at the end of the list to ensure the reference count to the front can go down to 0.
        - Very rarely used otherwise

Smart Pointer Examples

- unique_ptr.cc
- shared_ptr_leaky.cc
- shared_ptr_good.cc

Inheritance Constructors/Destructors

- The derived class:
  - Does not inherit any constructors.
  - MUST call their base class constructor.
    - Omission == calling the default constructor.

- Constructors resolve from base to derived.

- Destructors should be virtual !

Inheritance Examples

- Example:
- destructex.cc
  - This code compiles with no warnings so it must be right?

Vtables

- Dynamic dispatch
- All virtual functions are stored in a “virtual function table”
    - Each class has its own vtable
- Each instance contains an extra “field”
    - Pointer to class vtable
    - Only exists if class has virtual methods
- Derived classes have functions in same order as base class
    - Overriding functions replace base functions at same indices

Vtable Example

```
class Base {
  virtual void other_fn();
  virtual void overridden();
};

class Derived {
  void overridden() override;
};
```

class Base vtable
Base::other_fn()
Base::overridden()

class Derived vtable
Base::other_fn()
Derived::overridden()

Vtable Example

- Example:
- vtable.cc
  - Poke around this code with objdump or gdb!