CSE 333 Section 6

Smart Pointers, Casting, and Inheritance

https://me.me/i/when-you-mistype-a-keyword-in-c-static-cat-dynamic-2bcd3ae67b1343f58403678461cec7b9

UNIVERSITY of WASHINGTON

Logistics

- **Exercise 8**
  - Due **Friday** (7/29) @ 11:00am
  - Streams and STL containers
- **HW3**
  - Due **Thursday** (8/4) @ 11:59pm
  - Start early!

HW3 (Quick) Overview

Homework 3

- Main Idea: Building on HW2 `searchshell` by storing built data structures to disk.
  - Same functionality as `searchshell` (e.g., taking in queries, giving results)
  - Can now read information from file instead of crawling through directories again

- What does this mean?
  - Part A: **Writing** our MemIndex struct to a file
  - Part B: **Reading** from the index file we just created
  - Part C: Write a new version of `searchshell` that can accept index files

Part A: Writing an index file

Key points:

- Memory is expensive and volatile, so we'll be working on storing our search engine index on disk
- **Most of our data structures are hash tables**, so we can create a more general function to write hash tables with different elements
- Host vs disk order (it'll come back when we get to the network…)

Part B: Reading an index file

Key points:

- (Sort of) reverse of what we did in part A, we will read from the index file we just created
- Almost entirely in C++
- Need to check if the file is valid and not corrupt (i.e. magic number, checksum)
- Handling various lookups in the file (remembering that we are mostly working with hashtables, hence we use the HashTableReader baseclass)
- Ultimately implement the QueryProcessor, which powers the actual queries

Part C: `filesearchshell`

Key points:

- Basically just search shell but you have different argument now, the index files!
- Need to check args, build list of index files, then loop and take user queries
- Most of the heavy lifting is done by your work in part B (QueryProcessor)

Smart Pointers!

Review: Smart Pointers

- **`std::shared_ptr`** (Documentation) – Uses reference counting to determine when to delete a managed raw pointer
  - **`std::weak_ptr`** (Documentation) – Used in conjunction with `shared_ptr` but does **not** contribute to reference count

- **`std::unique_ptr`** (Documentation) – Uniquely manages a raw pointer
  - Used when you want to declare unique ownership of a pointer
  - Disabled `cctor` and `op=`

Using Smart Pointers

- Treat a smart pointer like a **normal (raw) pointer**, except now you **won't** have to use `delete` to deallocate memory!
  - You can use `*`, `->`, `[]` as you would with a raw pointer!

- **Initialize** a smart pointer by passing in a pointer to **heap memory**:

```
unique_ptr<int[]> u_ptr(new int[3]);
```

  - For **`shared_ptr`** and **`weak_ptr`**, you can use `cctor` and `op=` to get a copy

```
shared_ptr<int[]> s_ptr(another_shared_ptr);
```

Using Smart Pointers cont.

- Want to transfer ownership from one `unique_ptr` to another ?
```
unique_ptr<T> V = std::move(unique_ptr<T> U);
```

- Want to convert your `weak_ptr` to a `shared_ptr`?
```
std::shared_ptr s = w.lock();
```

- Want to get the reference count of a `shared_ptr`?
```
int count = s.use_count();
```

Exercise 1

Exercise 1

```
#include <memory>
using std::shared_ptr;

struct IntNode {
  IntNode(int* val, IntNode* node): value(val), next(node) {}

  ~IntNode() { delete val; }

  int* value;
  IntNode* next;
};
```

Exercise 1 Solution

```
#include <memory>
using std::shared_ptr;

struct IntNode {
  IntNode(int* val, IntNode* node) :
   value(shared_ptr<int>(val)), next(shared_ptr<IntNode>(node)) {}

  ~IntNode() { delete value; }

  shared_ptr<int> value;
  shared_ptr<IntNode> next;
};
```

Exercise 1 Solution

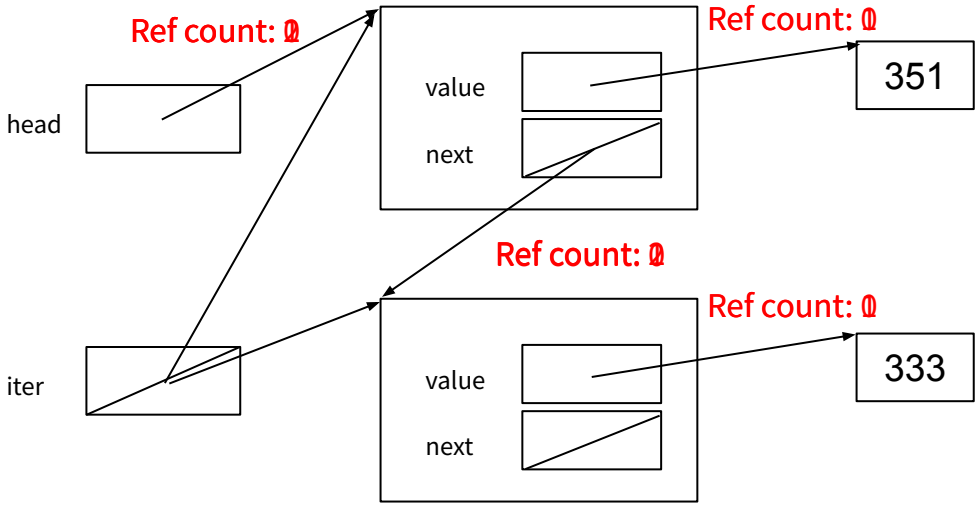
```
#include <memory>
using std::shared_ptr;

struct IntNode {
  IntNode(int* val, IntNode* node) :
   value(shared_ptr<int>(val)), next(shared_ptr<IntNode>(node)) {}

  ~IntNode() { delete value; }

  shared_ptr<int> value;
  shared_ptr<IntNode> next;
};
```

Ex1: Client Code




```
#include <iostream>

using std::cout;
using std::endl;

int main() {
  shared_ptr<IntNode> head(new IntNode(new int(351), nullptr));
  head->next = shared_ptr<IntNode>(new IntNode(new int(333), nullptr));
  shared_ptr<IntNode> iter = head;
  while (iter != nullptr) {
    cout << *(iter->value) << endl;
    iter = iter->next;
  }
}
```

Ex1: Client Code

Look at the section recording for a memory diagram walkthrough.

At the end of this code, nothing should be on the heap!

```
#include <iostream>

using std::cout;
using std::endl;

int main() {
  shared_ptr<IntNode> head(new IntNode(new int(351), nullptr));
  head->next = shared_ptr<IntNode>(new IntNode(new int(333), nullptr));
  shared_ptr<IntNode> iter = head;
  while (iter != nullptr) {
    cout << *(iter->value) << endl;
    iter = iter->next;
  }
}
```

Casting

Motivations

- C++ casting provides **explicit casting** when converting from one type to another
  - Creates **less ambiguity** of an operation
  - Allows for more **visibility** for casting (even though they are longer to write)
  - See: https://google.github.io/styleguide/cppguide.html#Casting

Different Flavors of Casting

- `static_cast<type_to>(expression);`
  Casting between related types

- `dynamic_cast<type_to>(expression);`
  Casting pointers of similar types (only used with inheritance)

- `const_cast<type_to>(expression);`
  Adding or removing **const**-ness of a type

- `reinterpret_cast<type_to>(expression);`
  Casting between incompatible types of the **same size** (doesn't do float conversion)

Tips with Casting

- Style: Use C++ style casting in C++
  - Tradeoff: Extra programming overhead, but provides **clarity** to your programs
  - Be **explicit as possible** with your casting! This means if you notice multiple operations in an implicit cast, you should explicitly write out each cast!

- Read documentation of casting on which casting to use
  - Documentation: https://www.cplusplus.com/articles/iG3hAqkS/
  - Consider the purpose of C++ casting is to be less ambiguous with what casting you use in C

Exercise 2

```
class Base {
 public:
  int x;
};
```

```
class Derived : public Base {
 public:
  int y;
}
```

```
int64_t x = 0x7ffffffffe870;
char* str = ________________________ (x);

void foo(Base* b) {
  Derived* d = ______________________ (b);
  // additional code omitted
}

Derived* d = new Derived;
Base* b = __________________________ (d);

double x = 64.382;
int64_t y = ________________________ (x);
```

```
class Base {
 public:
  int x;
};
```

```
class Derived : public Base {
 public:
  int y;
}
```

```
int64_t x = 0x7ffffffe870;
char* str = ______________________________ (x);

void foo(Base* b) {
  Derived* d = ___________________________ (b);
  // additional code omitted
}

Derived* d = new Derived;
Base* b = _________________________________ (d);

double x = 64.382;
int64_t y = ______________________________ (x);
```

```
class Base {
 public:
  int x;
};
```

```
class Derived : public Base {
 public:
  int y;
}
```

```
int64_t x = 0x7ffffffffe870;
char* str = reinterpret_cast<char*>(x);

void foo(Base* b) {
  Derived* d = dynamic_cast<Derived*>(b);
  // additional code omitted
}

Derived* d = new Derived;
Base* b = static_cast<Base*>(d);

double x = 64.382;
int64_t y = static_cast<int64_t>(x);
```

Inheritance

Inheritance

- Motivation: Better modularize our code for similar classes!

- A derived class inherits all **non-private** member variables and functions (**except** for `ctor`, `cctor`, `dtor`, `op=`) from its base class
  - *Similar to*: A subclass inherits from a superclass

- Aside: We will be only using **public, single** inheritance in CSE 333

HW3 Data Structures

- **DocTable**: A HashTable that maps document ID's to document names
- **Index**: A HashTable of words to DocID Tables (which will contain…)
- **DocIDTable**: (Like `WordPositions` struct) A HashTable that maps documents to offsets in a file (where a specific word is)

That's a lot of HashTables!

HW3 Data Structures: Lots of HashTables

Key	Value
5	"test_tree/README.TXT"
1	"test_tree/books/ulysses.txt"
4	"test_tree/bash-4.2/trap.c"
2	"test_tree/enron_email/2."
3	"test_tree/example.txt"

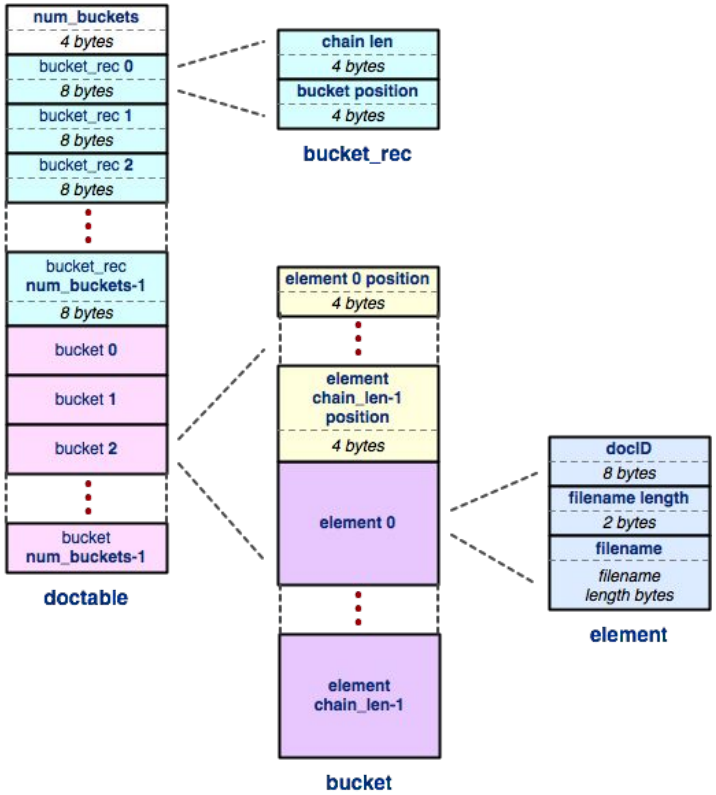
docid_to_docname

Key	Value
FNV64("test_tree/README.TXT")	(DocID_t) 5
FNV64("test_tree/example.txt")	(DocID_t) 3
FNV64("test_tree/enron_email/2.")	(DocID_t) 2
FNV64("test_tree/bash-4.2/trap.c")	(DocID_t) 4
FNV64("test_tree/books/ulysses.txt")	(DocID_t) 1

docname_to_docid

Write File (Part A)

Read File (Part B)



Motivation: HW3 Index Reading (Part B)

- Base Class:
  - `HashTableReader`
  - Example `protected` members:
    - `LookupElementPositions(HTKey_t hash_val) const;`
    - `FILE* file_;`
    - There are more! Take a look at the .h file for `HashTableReader`!
- Derived Classes (can use the fields of the base class):
  - `IndexTableReader` – Reads the index table
  - `DocIDTableReader` – Reads the DocID Table
  - `DocTableReader` – Reads the DocTable
  - `FileIndexReader` – Reads the Index from the file

Polymorphism: Dynamic Dispatch

- **Polymorphism** allows for you to access objects of related types (base and derived classes) – Allows interface usage instead of class implementation

- **Dynamic dispatch**: Implementation is determined *at* **runtime** via lookup
  - Allows you to call the **most-derived** version of the actual type of an object
  - Generally want to use this when you have a derived class

- `virtual` replaces the class's default **static dispatch** with **dynamic dispatch**
  - Static dispatch determines implementation at compile time
  - Meaning it does **not** perform polymorphism (just calls its function)

Example Base Class: Abstract Classes

- Pure `virtual` Functions – Functions without any implementation
  - Declaration Example: `virtual int foo() = 0;`
  - Used for creating an interface of a function

- **Abstract Classes** are those with one or more `virtual` functions
  - Creates an interface for the client to use without knowing its details
  - **Requires** a derived class to implement its functionality (cannot itself be instantiated)

- Often used like an interface!

  Usage Example: `AbstractClass* a = new DerivedClass(params);`

Dynamic Dispatch: Style Considerations

- Defining Dynamic Dispatch in your code base
  - Use `virtual` **only once** when first defined in the base class
  - All derived classes of a base class should use `override` to check at compile time that a function uses dynamic dispatch

- Call dtors of a base class as `virtual` – Guarantees all derived classes will use dynamic dispatch for their destructors

Example Abstract Class/Derived Class

```
using std::string;

class Fruit {
 public:
  Fruit() = default;
  virtual ~Fruit() {}

  // A fun fact
  virtual string FunFact() = 0;
};
```

```
using std::string;

class Banana : public Fruit {
 public:
  Banana() = default;
  virtual ~Banana() = default;

  string FunFact() override {
    return “It’s a berry”;
  }
};
```

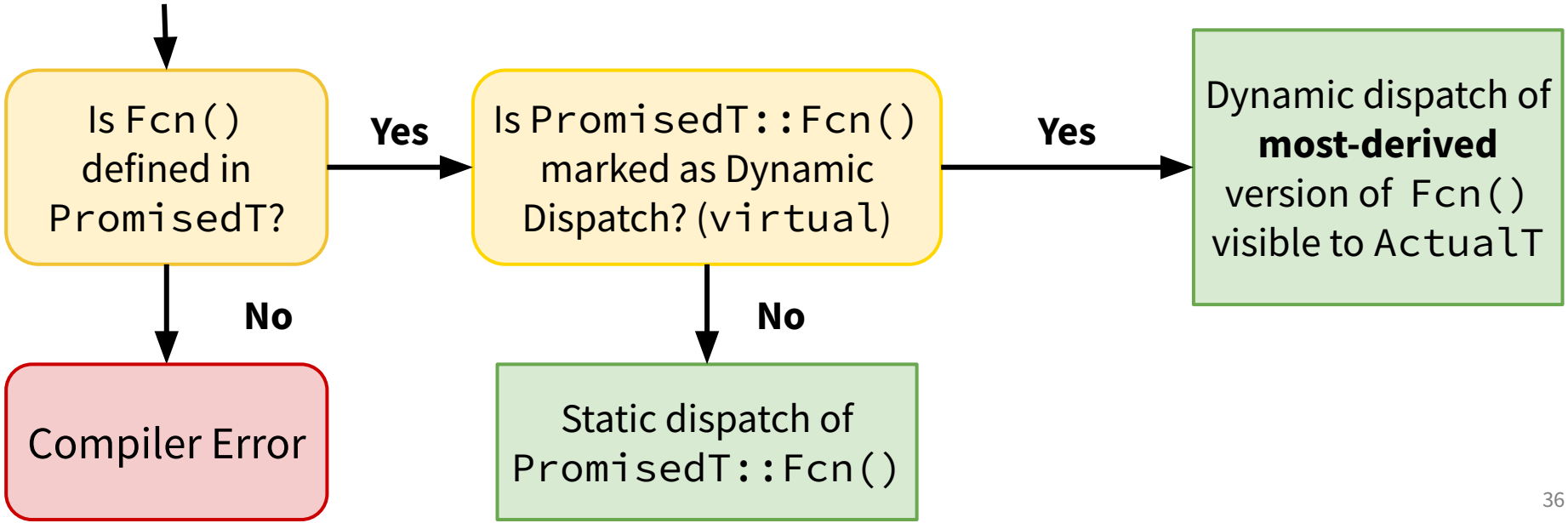
Practice: static, dynamic, or error?

```
class Base {
  void Foo();              //Static Dispatch
  void Bar();              //Static Dispatch
  virtual void Baz();      //  Dynamic Dispatch
};

class Derived : public Base {
  virtual void Foo();      //
  void Bar() override;     //Dynamic Dispatch (for more derived)
  void Baz();              //  Compiler Error!!
};                         //  Dynamic Dispatch
```

Dispatch Decision Tree

```
PromisedT* ptr = new ActualT();
ptr->Fcn();  // which version is called?
```



Fixes: static, dynamic, or error?

```
class Base {
  void Foo();             // static dispatch
  void Bar();             // static dispatch
  virtual void Baz();     // dynamic dispatch
};

class Derived : public Base {
  virtual void Foo();     // now dynamic (for more derived)
  void Bar();             // static dispatch
  void Baz() override;    // still dynamic (sticky!)
};
```

Exercise 3

Exercise 3A: Abstract Animals

Create an `Animal` Abstract class. It should have a protected member `legs` variable and a public `num_legs` pure virtual function. Try to use good style!

Exercise 3A: Abstract Animals

Create an `Animal` Abstract class. It should have a protected member `legs` variable and a public num_legs pure virtual function. Try to use good style!

```
class Animal {
 public:
  Animal() = default;
  virtual ~Animal() {}
  virtual int num_legs() const = 0;

 protected:
  int    legs;
};
```

Exercise 3B: Create an Animal Derived class

Now that you have made an abstract `Animal` class, try to make a implementation with a derived class of `Animal`.

This is an open-ended question, so you are free to be imaginative with your implementation of the abstract `Animal` class!

Exercise 3B: Create an Animal Derived class

```
class Dog : public Animal {
 public:
  Dog(int legs, string breed) : legs(legs), breed(breed) {}
  ~Dog() {}
  int num_legs() const override {
    return legs;
  }
  virtual int get_breed() const {
    return breed;
  }

 protected:
  string breed;
};
```