

# CSE 333

## Section 7

HW3, Smart Pointers, and Inheritance



# Logistics

- **Exercise 14**
  - Due **Friday (5/13) @ 10:00am**
- **HW3**
  - Due next **Thursday (5/19) @ 11:00pm**



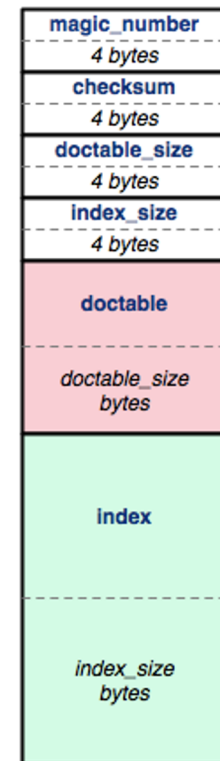
# HW 3 Overview!



# Index File

Crawling the whole file tree takes a long time!

To save time we'll write the completed DocTable and MemIndex into a file!



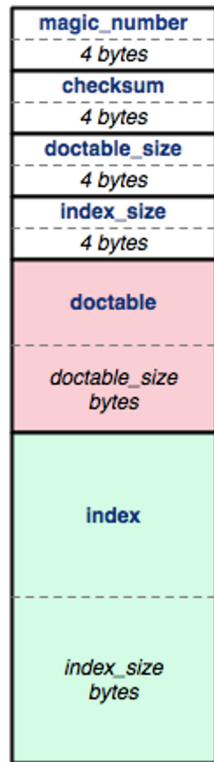
**index file**

# Byte Ordering and Endianness

- Network (Disk) Byte Order (Big Endian)
  - The most significant byte is stored in the highest address
- Host byte order
  - Might be big or little endian, depending on the hardware
- To convert between orderings, we can use
  - `uint32_t htonl (uint32_t hostlong); // host to network`
  - `uint32_t ntohl (uint32_t netlong); // network to host`
- Pro-tip:

The structs in HW3 have `toDiskFormat()` and `toHostFormat()` functions that will convert endianness for you.

# Index File Components



**index file**

Header (metadata)

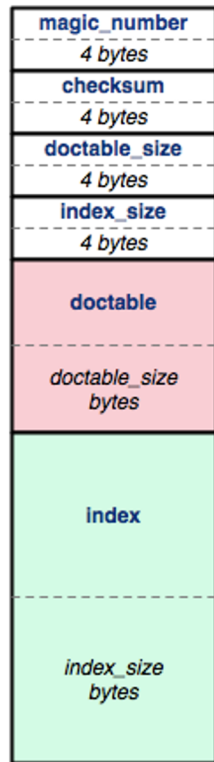
---

DocTable

---

MemIndex

# Index File Header



index file

- magic\_number: 0xCAFEF00D
- checksum: mathematical signature
- doctable\_size: in bytes
- index\_size: in bytes

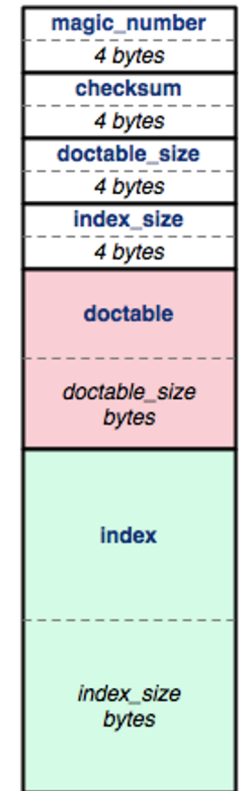
# Index File Header - HEX

1. Find a hex editor/viewer of your choice
  - `xxd <indexfile>`
  - `hexdump -vC <indexfile>`
  - Pipe the output into a file or into `less` to view

```
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....
```

The header:

**Magic word**   **Checksum**   **Doctable size**   **Index size**



index file



# Hex View

- emacs – “M-x hexl-mode”

```
File Edit Options Buffers Tools Hexl Help
87654321 0011 2233 4455 6677 8899 aabb ccdd eeff 0123456789abcdef
00000000: cafe f00d ff48 a0a1 0000 006a 0000 024e .....H.....j...N
00000010: 0000 0001 0000 0002 0000 001c 0000 0024 .....$
00000020: 0000 0054 0000 0000 0000 0002 0026 2e2f ...T.....&./
00000030: 7465 7374 5f74 7265 652f 7469 6e79 2f68 test_tree/tiny/h
00000040: 6f6d 652d 6f6e 2d74 6865 2d72 616e 6765 ome-on-the-range
00000050: 2e74 7874 0000 0000 0000 0001 001c 2e2f .txt...../
```

- vim – “:%!xxd”

```
00000000: cafe f00d ff48 a0a1 0000 006a 0000 024e .....H.....j...N
00000010: 0000 0001 0000 0002 0000 001c 0000 0024 .....$
00000020: 0000 0054 0000 0000 0000 0002 0026 2e2f ...T.....&./
00000030: 7465 7374 5f74 7265 652f 7469 6e79 2f68 test_tree/tiny/h
00000040: 6f6d 652d 6f6e 2d74 6865 2d72 616e 6765 ome-on-the-range
00000050: 2e74 7874 0000 0000 0000 0001 001c 2e2f .txt...../
```

# Hex View

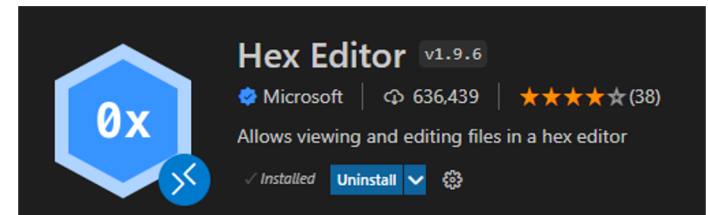
- emacs – “M-x hexl-mode”

```
File Edit Options Buffers Tools Hexl Help
87654321 0011 2233 4455 6677 8899 aabb ccdd eeff 0123456789abcdef
00000000: cafe f00d ff48 a0a1 0000 006a 0000 024e .....H.....j...N
00000010: 0000 0001 0000 0002 0000 001c 0000 0024 .....$
00000020: 0000 0054 0000 0000 0000 0002 0026 2e2f ...T.....&./
00000030: 7465 7374 5f74 7265 652f 7469 6e79 2f68 test_tree/tiny/h
00000040: 6f6d 652d 6f6e 2d74 6865 2d72 616e 6765 ome-on-the-range
00000050: 2e74 7874 0000 0000 0000 0001 001c 2e2f .txt...../
```

- vim – “:%!xxd”

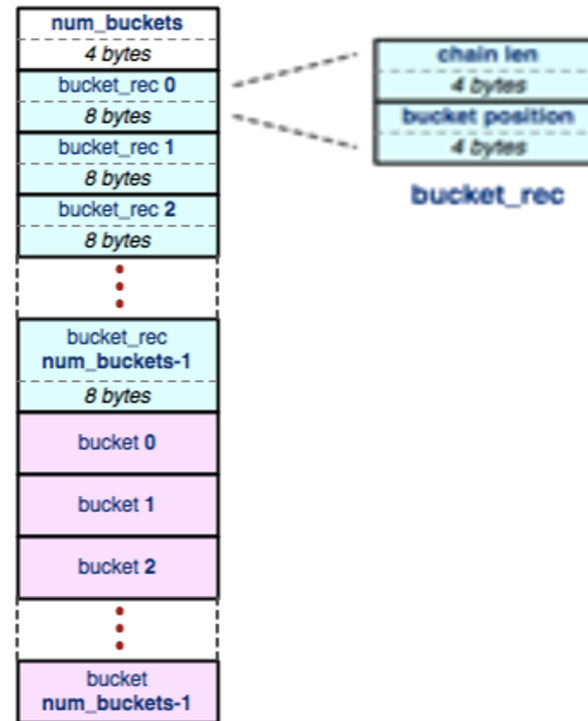
```
00000000: cafe f00d ff48 a0a1 0000 006a 0000 024e .....H.....j...N
00000010: 0000 0001 0000 0002 0000 001c 0000 0024 .....$
00000020: 0000 0054 0000 0000 0000 0002 0026 2e2f ...T.....&./
00000030: 7465 7374 5f74 7265 652f 7469 6e79 2f68 test_tree/tiny/h
00000040: 6f6d 652d 6f6e 2d74 6865 2d72 616e 6765 ome-on-the-range
00000050: 2e74 7874 0000 0000 0000 0001 001c 2e2f .txt...../
```

For those working in VSCode...



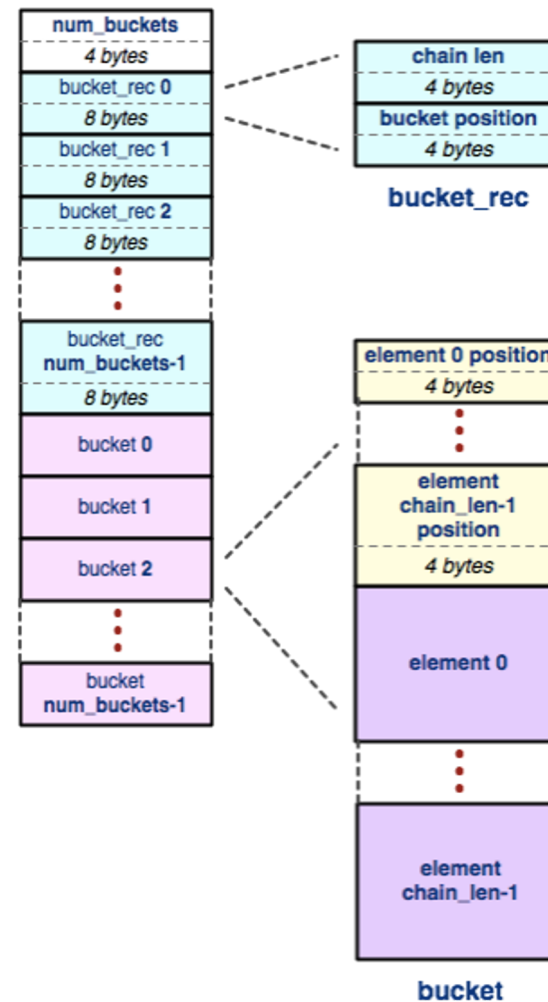
# HashTable

- HashTable can have varying amount of buckets, so start with num\_buckets.
- Buckets can be of varying lengths. To know the offset, we store some bucket records.

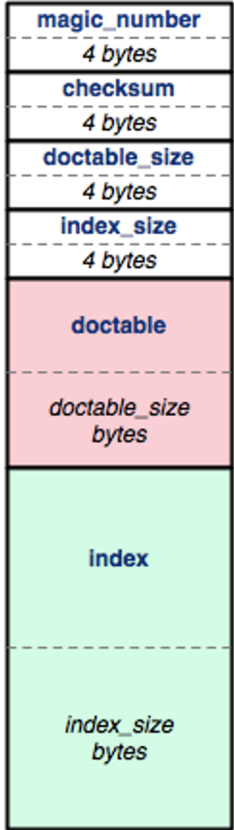


# Buckets

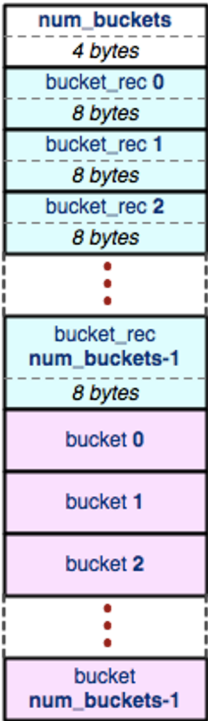
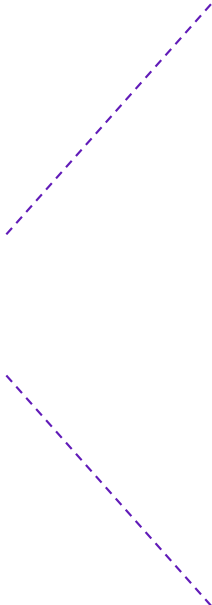
- A bucket is a list that contains elements in the table. Offset to a bucket is found in a bucket record.
- Elements can be of various sizes, so we need to store element positions to know where each element is.



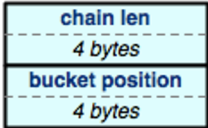
# DocTable



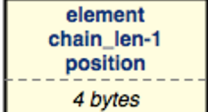
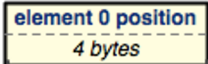
index file



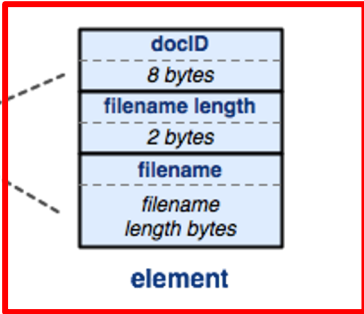
doctable



bucket\_rec

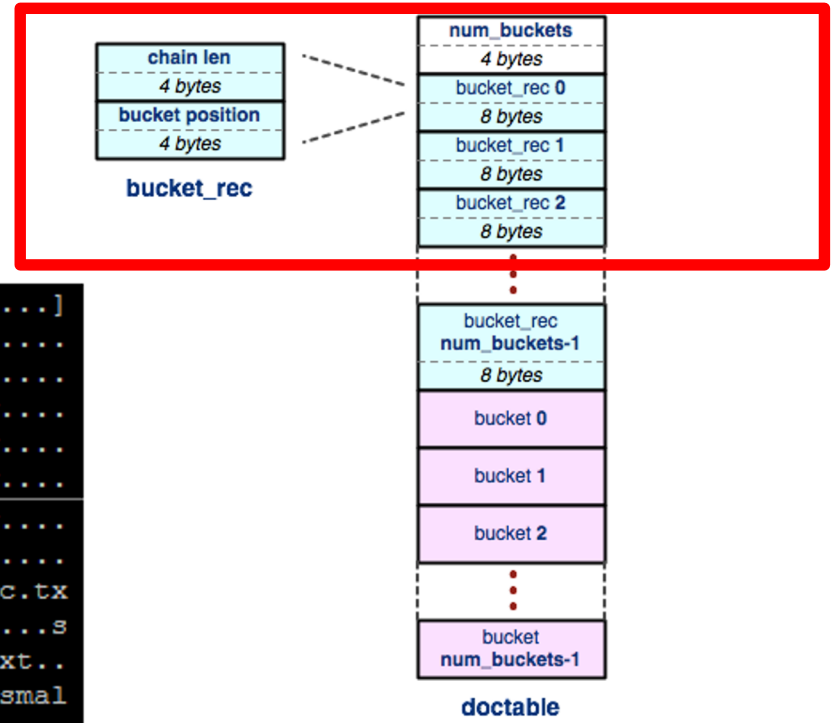


bucket



element

# DocTable (Hex)



```

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....

0002000: 0000 206b 0000 0000 0000 206b 0000 0000  .. k..... k....
0002010: 0000 206b 0000 2018 0000 0000 0000 0001  .. k.. .....
0002020: 000f 736d 616c 6c5f 6469 722f 632e 7478  ..small_dir/c.tx
0002030: 7400 0020 3500 0000 0000 0000 0200 0f73  t.. 5.....s
0002040: 6d61 6c6c 5f64 6972 2f62 2e74 7874 0000  mall_dir/b.txt..
0002050: 2052 0000 0000 0000 0003 000f 736d 616c  R.....smal
0002060: 6c5f 6469 722f 612e 7478 7400 0000 8000  l_dir/a.txt....
0002070: 0000 0000 0024 6f00 0000 0000 0024 6f00  ....$o.....$o.
  
```

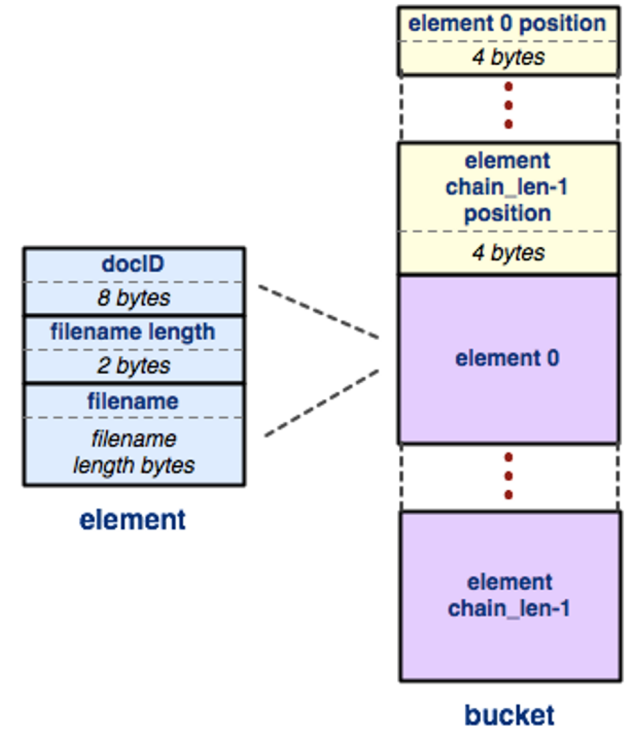
The header

Num buckets ( Chain len Bucket offset )\*

# DocTable

```

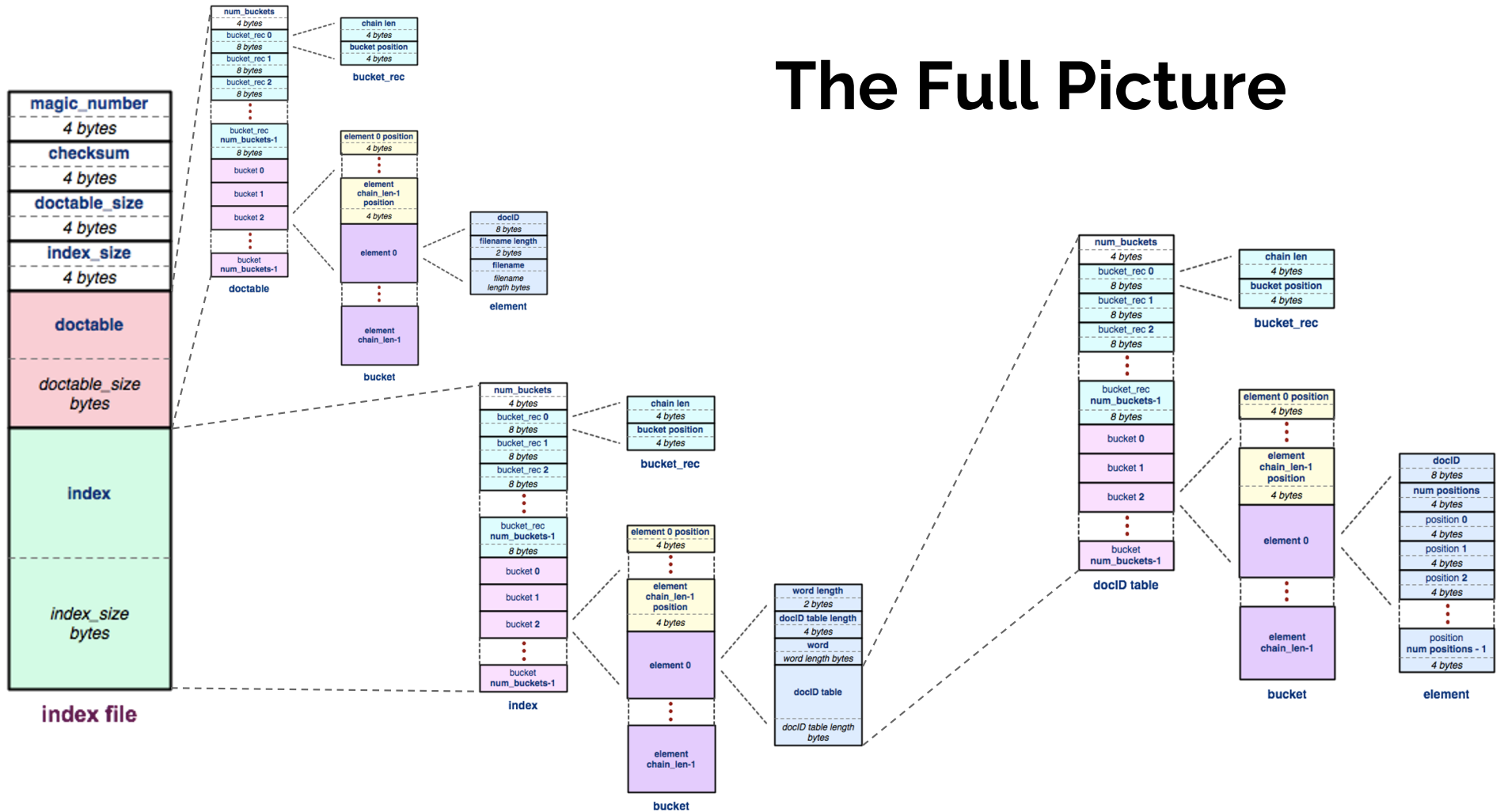
0002000: 0000 206b 0000 0000 0000 206b 0000 0000 .. k..... k....
0002010: 0000 206b 0000 2018 0000 0000 0000 0001 .. k.. .....
0002020: 000f 736d 616c 6c5f 6469 722f 632e 7478 ..small_dir/c.tx
0002030: 7400 0020 3500 0000 0000 0000 0200 0f73 t.. 5.....s
0002040: 6d61 6c6c 5f64 6972 2f62 2e74 7874 0000 mall_dir/b.txt..
  
```



The buckets: where n is equal to the number of elements

$$((\text{Element offset})^n (\text{DocID} \text{ Filename len} \text{ Filename})^n)^*$$

# The Full Picture





# HW Tips

- When Writing, you should (almost) always:
  1. `.toDiskFormat()`
  2. `fseek()`
  3. `fwrite()`
- When Reading, you should (almost) always:
  1. `fseek()`
  2. `fread()`
  3. `.toHostFormat()`
- The most common bugs in the HW involve forgetting to change byte ordering, or forgetting to `fseek()`.

# Hex View Exercise

- Take a look at <https://courses.cs.washington.edu/courses/cse333/22sp/sections/sec07.idx>
  - Log into attu, use wget to download the file, then look into it.
- Try to figure out:
  - How many documents are in this index?
  - Which words are in each document?

# Hex View Exercise

- Take a look at <https://courses.cs.washington.edu/courses/cse333/22sp/sections/sec07.idx>
  - Log into attu, use wget to download the file, then look into it.
- Try to figure out:
  - How many documents are in this index?
  - Which words are in each document?

Answer: This index file was built off of test\_tree/tiny

# 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 `ctor` 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; }

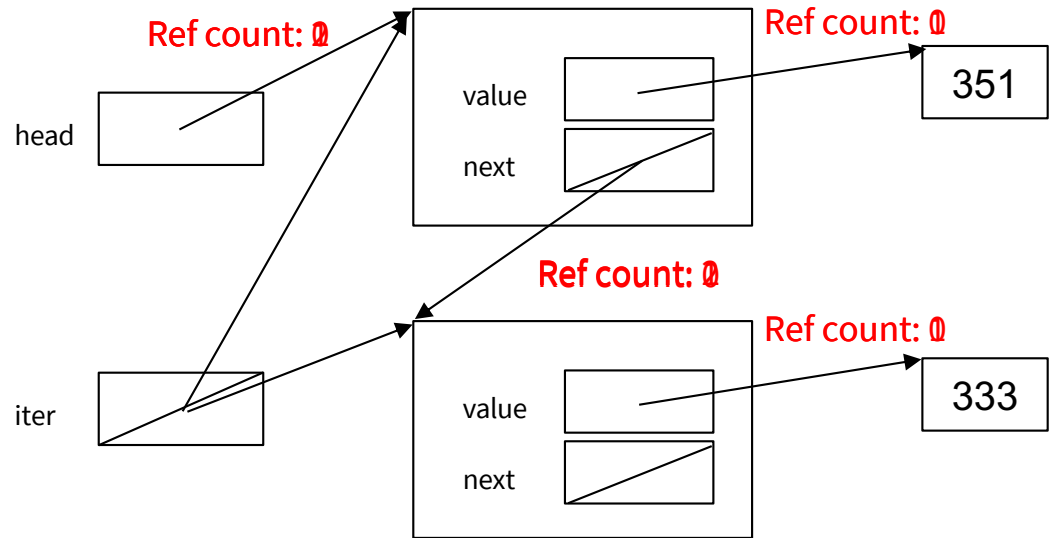
    shared_ptr<int> value;
    shared_ptr<IntNode> next;
};
```

# 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

Nothing left 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;
    }
}
```

# Inheritance



# Inheritance

- Motivation: Better modularize our code for similar classes!
- The public interface of 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

# 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** use dynamic dispatch (just calls its function)

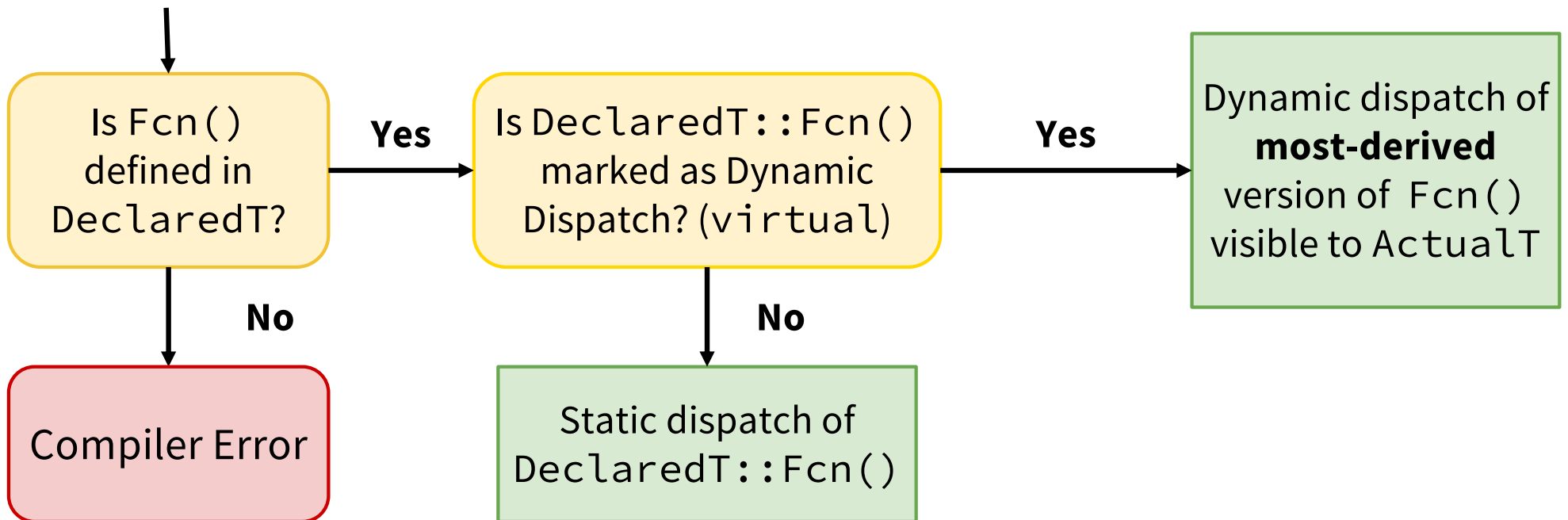


# Dynamic Dispatch: Style Considerations

- 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 get the compiler to check that a function overrides a virtual function from a base class
- Use `virtual` for destructors of a base class – Guarantees all derived classes will use dynamic dispatch to ensure use of appropriate destructors

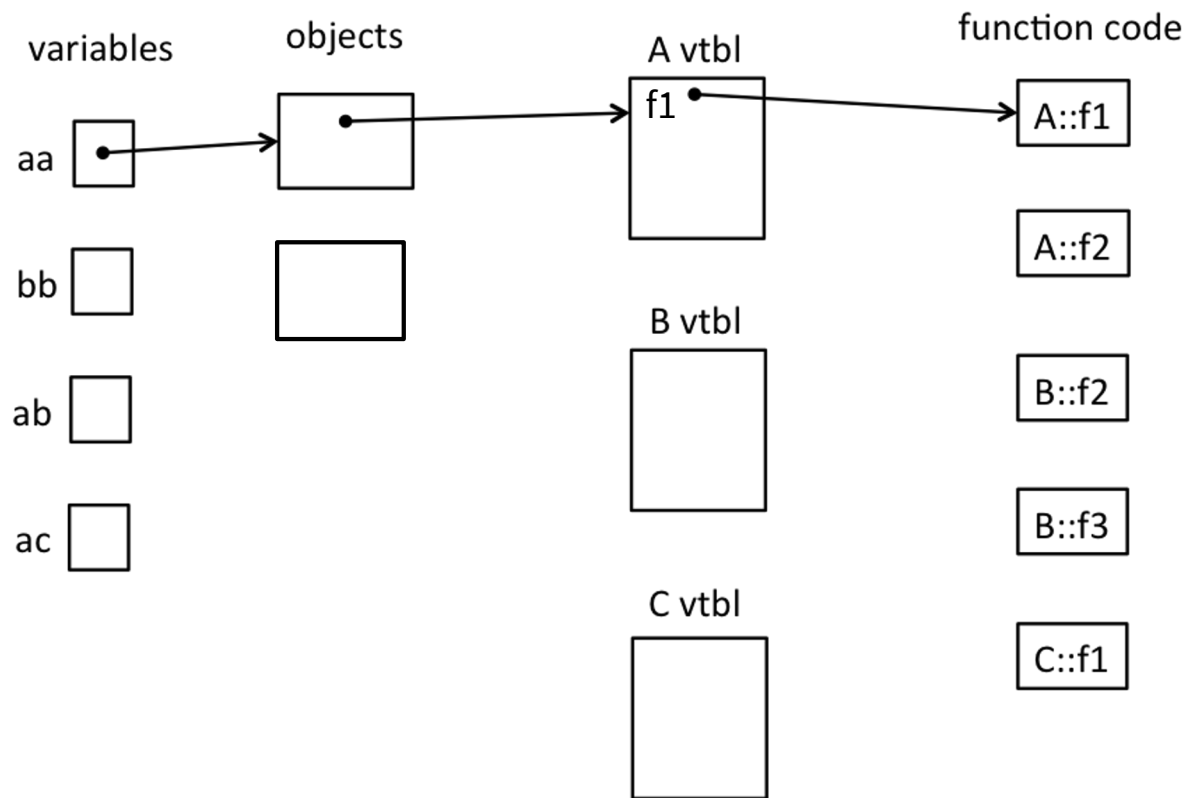
# Dispatch Decision Tree

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



# Exercise 2

## Exercise 2 (Drawing vtable diagram)



# Exercise 2 Solution (pointers)

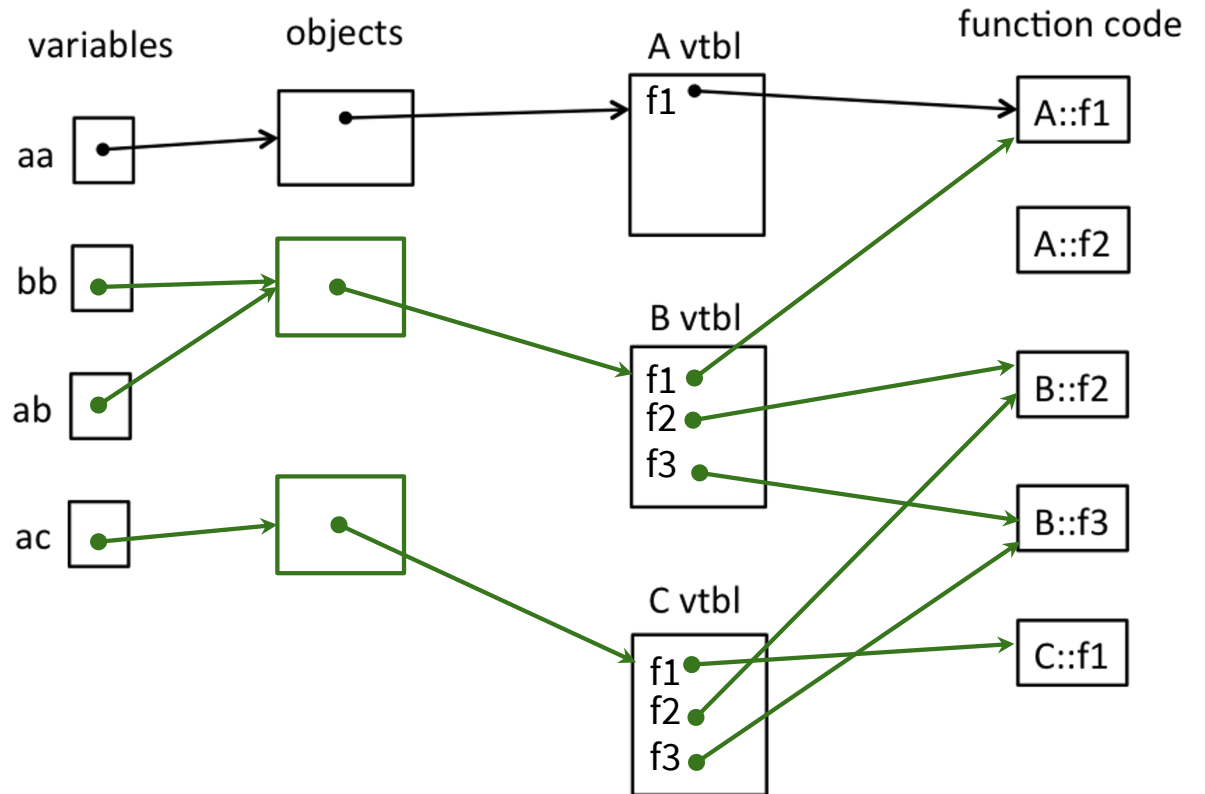
```
#include <iostream>
using namespace std;
```

```
class A {
public:
    virtual void f1() { f2(); cout << "A::f1" << endl; }
    void f2() { cout << "A::f2" << endl; }
};
```

```
class B: public A {
public:
    virtual void f3() { f1(); cout << "B::f3" << endl; }
    virtual void f2() { cout << "B::f2" << endl; }
};
```

```
class C: public B {
public:
    void f1() { f2(); cout << "C::f1" << endl; }
};
```

```
int main() {
    A* aa = new A();
    B* bb = new B();
    A* ab = bb;
    A* ac = new C();
}
```



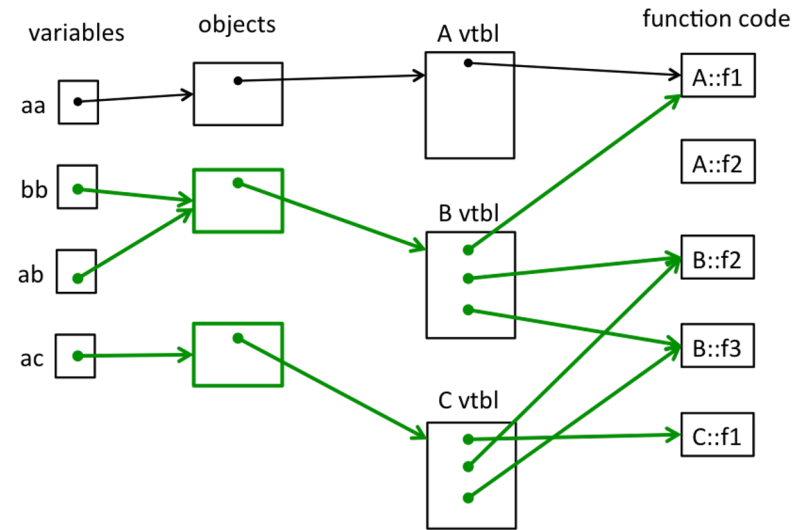
# Exercise 2 Solution (output)

```
#include <iostream>
using namespace std;

class A {
public:
    virtual void f1() { f2(); cout << "A::f1" << endl; }
    void f2() { cout << "A::f2" << endl; }
};

class B: public A {
public:
    virtual void f3() { f1(); cout << "B::f3" << endl; }
    virtual void f2() { cout << "B::f2" << endl; }
};

class C: public B {
public:
    void f1() { f2(); cout << "C::f1" << endl; }
};
```



```
A* aa = new A();
aa->f1();
```

A	B	C	D
B::f2	A::f2	A::f2	B::f2
A::f1	C::f1	A::f1	C::f1

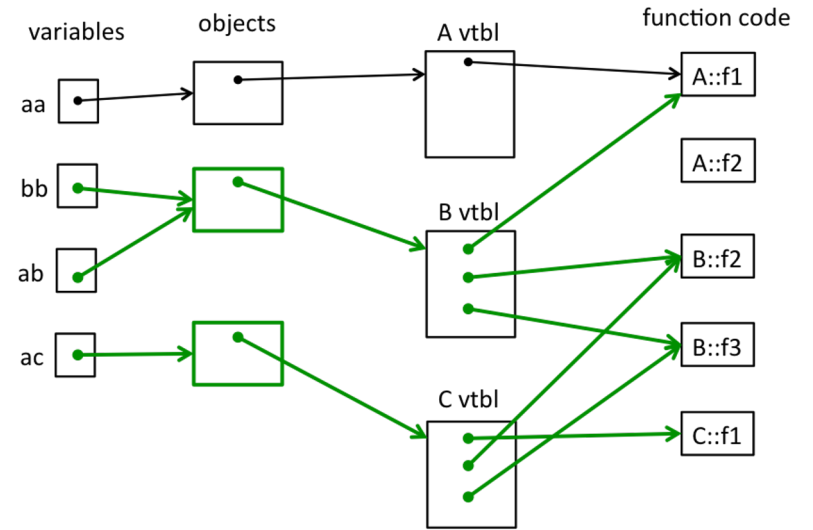
# Exercise 2 Solution (output)

```
#include <iostream>
using namespace std;

class A {
public:
    virtual void f1() { f2(); cout << "A::f1" << endl; }
    void f2() { cout << "A::f2" << endl; }
};

class B: public A {
public:
    virtual void f3() { f1(); cout << "B::f3" << endl; }
    virtual void f2() { cout << "B::f2" << endl; }
};

class C: public B {
public:
    void f1() { f2(); cout << "C::f1" << endl; }
};
```



```
B* bb = new B();
```

```
bb->f1();
```

A	B	C	D
B::f2	A::f2	A::f2	B::f2
A::f1	C::f1	A::f1	C::f1

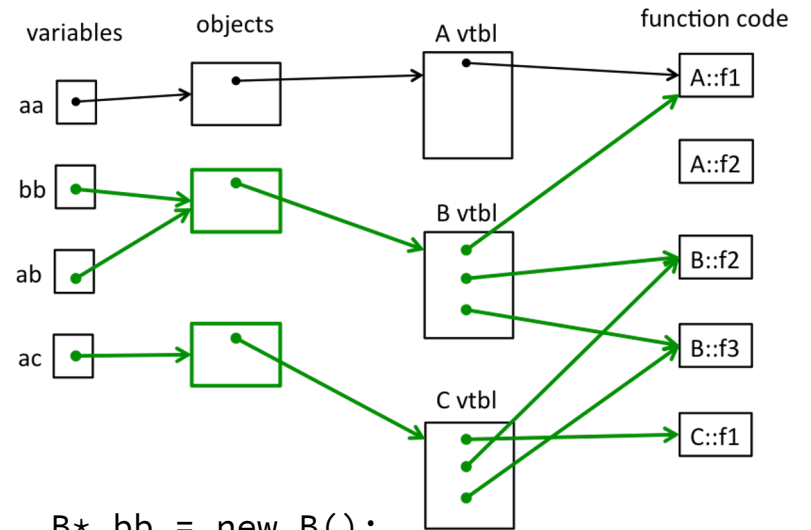
# Exercise 2 Solution (output)

```
#include <iostream>
using namespace std;

class A {
public:
    virtual void f1() { f2(); cout << "A::f1" << endl; }
    void f2() { cout << "A::f2" << endl; }
};

class B: public A {
public:
    virtual void f3() { f1(); cout << "B::f3" << endl; }
    virtual void f2() { cout << "B::f2" << endl; }
};

class C: public B {
public:
    void f1() { f2(); cout << "C::f1" << endl; }
};
```



```
B* bb = new B();
A* ab = bb;

bb->f2();
cout << "----" << endl;
ab->f2();
```

A	B	C	D
B::f2	A::f2	B::f2	A::f2
----	----	----	----
B::f2	B::f2	A::f2	A::f2



# Extension



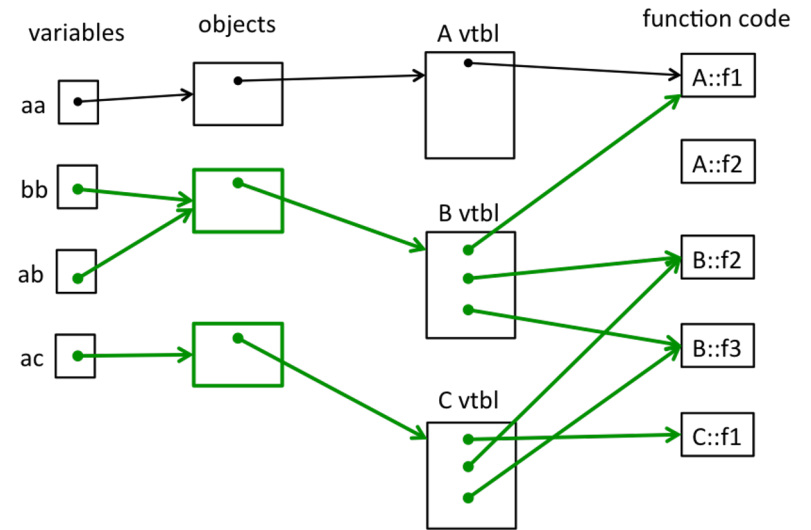
# Exercise 2 Solution (output)

```
#include <iostream>
using namespace std;

class A {
public:
    virtual void f1() { f2(); cout << "A::f1" << endl; }
    void f2() { cout << "A::f2" << endl; }
};

class B: public A {
public:
    virtual void f3() { f1(); cout << "B::f3" << endl; }
    virtual void f2() { cout << "B::f2" << endl; }
};

class C: public B {
public:
    void f1() { f2(); cout << "C::f1" << endl; }
};
```



```
B* bb = new B();
```

```
bb->f3();
```

A	B	C	D
B::f2	A::f2	A::f2	B::f2
A::f1	A::f1	C::f1	C::f1
B::f3	B::f3	B::f3	B::f3

# Exercise 2 Solution (output)

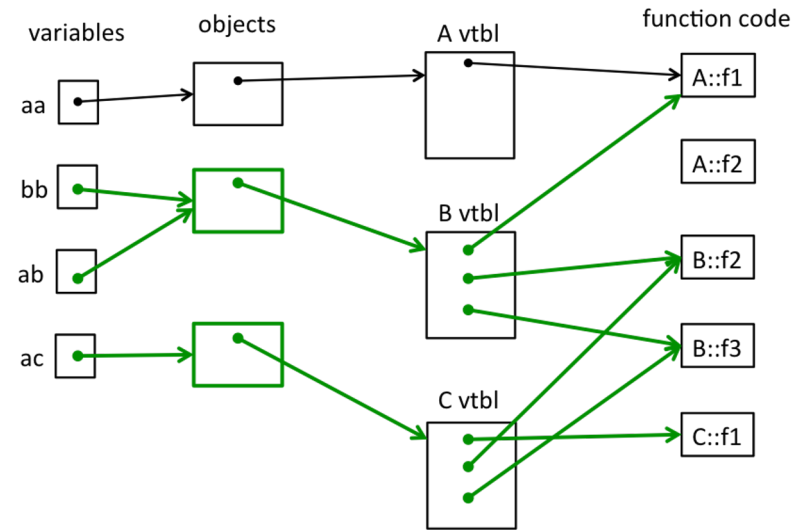
```

#include <iostream>
using namespace std;

class A {
public:
    virtual void f1() { f2(); cout << "A::f1" << endl; }
    void f2() { cout << "A::f2" << endl; }
};

class B: public A {
public:
    virtual void f3() { f1(); cout << "B::f3" << endl; }
    virtual void f2() { cout << "B::f2" << endl; }
};

class C: public B {
public:
    void f1() { f2(); cout << "C::f1" << endl; }
};
    
```



```
A* ac = new C();
```

```
ac->f1();
```

A	B	C	D
B::f2	A::f2	A::f2	B::f2
A::f1	C::f1	A::f1	C::f1

**Thanks for  
coming to section!**