

# CSE 333

## Section 4

### **C++ Const & References; Makefiles; HW2**

*Some slides referenced from CSE 333 Winter 2020*

# Logistics

---

- **Exercise 6** is out
  - due **Monday** (April 26), 11:00 PDT
- **HW2** is out
  - due next **Thursday** (April 29), 23:59 PDT

# Section Plan

— — —

- C++ const & References
  - Example (Exercise 1)
  - MultChoice class (Exercise 3)
- Makefiles practice (Exercise 5)
- HW2 Q&A

# Pointers, References, and Const

# Example

---

- Consider the following code:

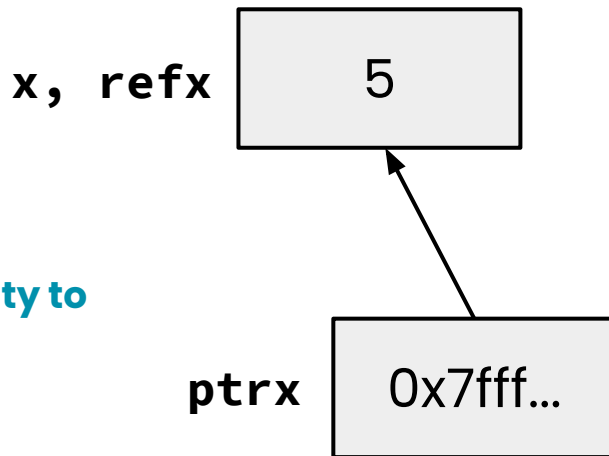
```
int x = 5;
```

```
int &refx = x;
```

```
int *ptrx = &x;
```

Note syntactic similarity to  
pointer declaration

Still the address-of operator!



*What are some tradeoffs to using pointers vs references?*

# Pointers Versus References

---

## Pointers

Can move to different data via reassignment/pointer arithmetic

Can be initialized to **NULL**

Useful for output parameters:  
`MyClass* output`

## References

References the same data for its entire lifetime - can't reassign

No sensible "default reference," must be an alias

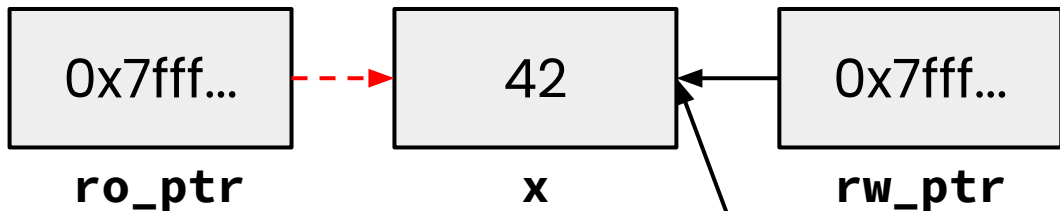
Useful for input parameters:  
`const MyClass& input`

# Pointers, References, and Parameters

---

- When would you prefer:
  - `void func(int &arg)` vs. `void func(int *arg)`
- Use **references** when you don't want to deal with pointer semantics
  - Allows real pass-by-reference
  - Can make intentions clearer in some cases
- Style wise, we want to use **references for input parameters** and **pointers for output parameters**, with the output parameters declared last
  - Note: A reference can't be NULL

# Const



## Legend

**Red** = "can't change box it's next to"

**Black** = "read and write"

- Mark a variable with `const` to make a compile time check that a variable is never reassigned
- Does not change the underlying write-permissions for this variable

```
int x = 42;
```

```
// Read only
```

```
const int *ro_ptr = &x;
```

```
// Can still modify x with rw_ptr!
```

```
int *rw_ptr = &x;
```

```
// Only ever points to x
```

```
int *const ptr = &x;
```

# Exercise 1

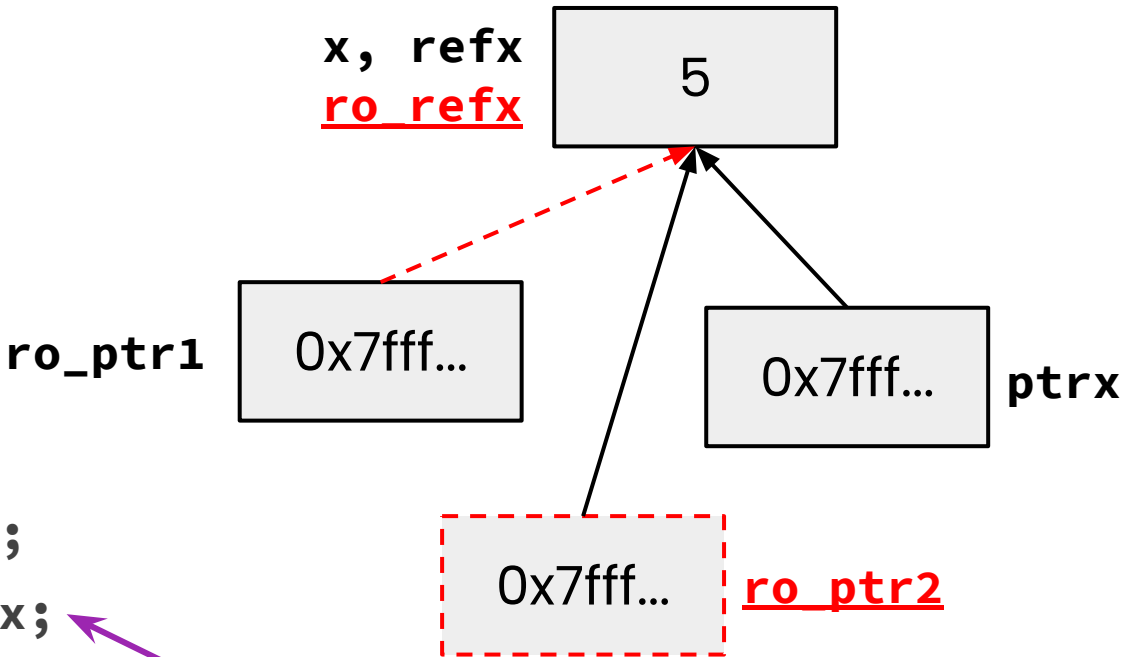
----

```
int x = 5;
int &refx = x;
int *ptrx = &x;
const int &ro_refx = x;
const int *ro_ptr1 = &x;
int *const ro_ptr2 = &x;
```

“Const pointer to an int”

“Pointer to a const int”

**Tip:** Read the declaration “right-to-left”



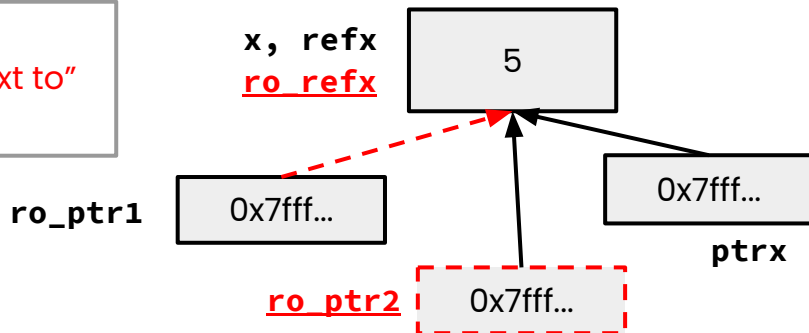
## Legend

**Red** = “can’t change box  
it’s next to”

**Black** = “read and write”

# Exercise 1

**Legend**  
**Red** = "can't change box it's next to"  
**Black** = "read and write"



```
void foo(const int &arg);  
void bar(int &arg);
```

```
int x = 5;  
int &refx = x;  
int *ptrx = &x;  
const int &ro_refx = x;  
const int *ro_ptr1 = &x;  
int *const ro_ptr2 = &x;
```

## Which result in a compiler error?

✓ OK

✗ ERROR

- ✓ bar(refx);
- ✗ bar(ro\_refx); *ro\_refx is const*
- ✓ foo(refx);
- ✓ ro\_ptr1 = (int\*) 0xDEADBEEF;
- ✗ ptrx = &ro\_refx; *ro\_refx is const*
- ✗ ro\_ptr2 = ro\_ptr2 + 2; *ro\_ptr2 is const*
- ✗ \*ro\_ptr1 = \*ro\_ptr1 + 1; *(\*ro\_ptr1) is const*

# Objects and Const Methods

```
#ifndef POINT_H_
#define POINT_H_

class Point {
public:
    Point(const int x, const int y);
    int get_x() const { return x_; }
    int get_y() const { return y_; }
    double Distance(const Point &p) const;
    void SetLocation(const int x, const int y);

private:
    int x_;
    int y_;
}; // class Point

#endif // POINT_H_
```

**Cannot** mutate the object it's called on.

**Trying to change x\_ or y\_ inside will throw a compiler error!**

A **const** class object can only call member functions that have been declared as **const**

# Exercise 3

# Exercise 3

Which *lines* of the snippets of code below would cause compiler errors?

✓ OK

✗ ERROR

---  
✓ `int z = 5;`  
✓ `const int *x = &z;`  
✓ `int *y = &z;`  
✓ `x = y;`  
✗ `*x = *y;`

-----  
✓ `int z = 5;`  
✓ `int *const w = &z;`  
✓ `const int *const v = &z;`  
✗ `*v = *w;`  
✓ `*w = *v;`

# Exercise 3

---

Which *lines* of the snippets of code below would cause compiler errors?

✓ OK     ✗ ERROR

```
class MultChoice {
public:
    MultChoice(int q, char resp) : q_(q), resp_(resp) { } // 2-arg ctor
    int get_q() const { return q_; }
    char get_resp() { return resp_; }
    bool Compare(MultChoice &mc) const; // do these MultChoice's match?

private:
    int q_; // question number
    char resp_; // response: 'A','B','C','D', or 'E'
}; // class MultChoice
```

✓ `const MultChoice m1(1, 'A');`  
✓ `MultChoice m2(2, 'B');`  
✗ `cout << m1.get_resp();`  
✓ `cout << m2.get_q();`

✓ `const MultChoice m1(1, 'A');`  
✓ `MultChoice m2(2, 'B');`  
✓ `m1.Compare(m2);`  
✗ `m2.Compare(m1);`

# What would you change about the class declaration to make it better?

```
class MultChoice {
public:
    MultChoice(int q, char resp) : q_(q), resp_(resp) { } // 2-arg ctor
    int get_q() const { return q_; }
    char get_resp() { return resp_; }
    bool Compare(MultChoice &mc) const; // do these MultChoice's match?

private:
    int q_; // question number
    char resp_; // response: 'A','B','C','D', or 'E'
}; // class MultChoice
```

# Makefiles

```
target: src1 src2 ... srcN  
command/commands
```

Makefiles are used to manage project recompilation. Project structure and dependencies can be represented as a directed acyclic graph (DAG), which a Makefile encodes to recursively build the minimum number of files for a specified target.

---

# Exercise 5

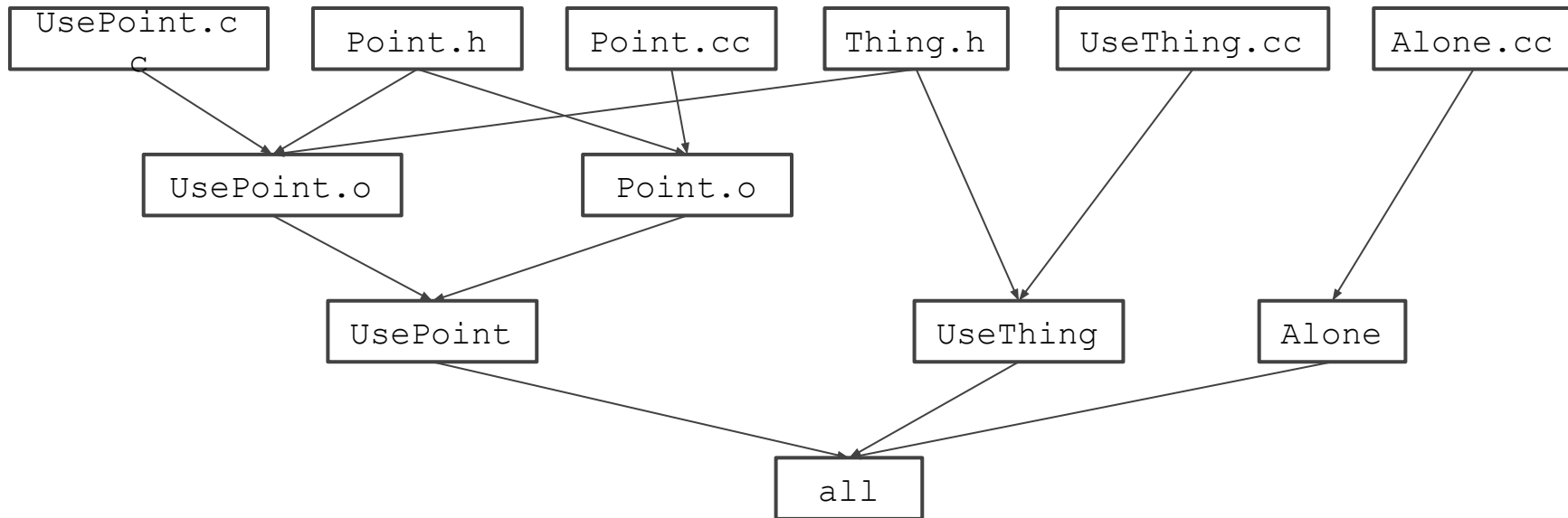
---	Point.h	<pre>class Point { ... };</pre>	Point.cc	<pre>#include "Point.h" // defs of methods</pre>
	UsePoint.cc	<pre>#include "Point.h" #include "Thing.h" int main( ... ) { ... }</pre>	Thing.h	<pre>struct Thing { ... }; // full struct def here</pre>
	UseThing.cc	<pre>#include "Thing.h" int main( ... ) { ... }</pre>	Alone.cc	<pre>int main( ... ) { ... }</pre>

1. Draw out the DAG
  - o The direction of the arrows are not important, but be consistent
2. Write a suitable Makefile

# DAG

Point.h	<pre>class Point { ... };</pre>
UsePoint.cc	<pre>#include "Point.h" #include "Thing.h" int main( ... ) { ... }</pre>
UseThing.cc	<pre>#include "Thing.h" int main( ... ) { ... }</pre>

Point.cc	<pre>#include "Point.h" // defs of methods</pre>
Thing.h	<pre>struct Thing { ... }; // full struct def here</pre>
Alone.cc	<pre>int main( ... ) { ... }</pre>



# Makefile

---

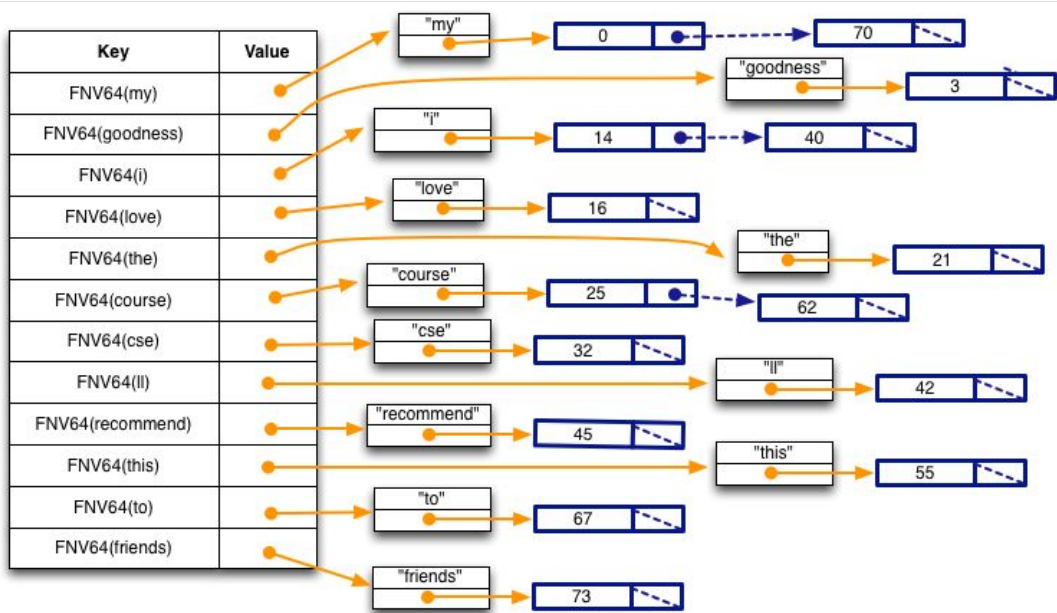
Variable

Phony target  
Note: all first

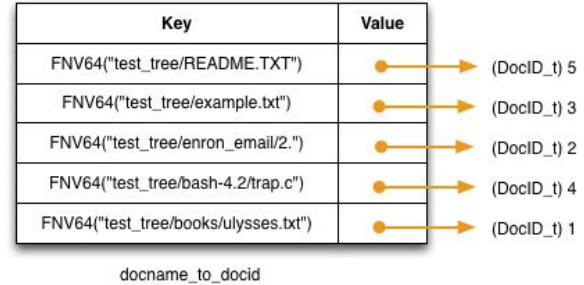
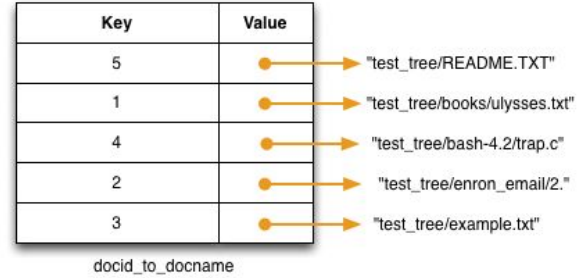
```
CFLAGS = -Wall -g -std=c++17
all: UsePoint UseThing Alone
UsePoint: UsePoint.o Point.o
g++ $(CFLAGS) -o UsePoint UsePoint.o Point.o
UsePoint.o: UsePoint.cc Point.h Thing.h
g++ $(CFLAGS) -c UsePoint.cc
Point.o: Point.cc Point.h
g++ $(CFLAGS) -c Point.cc
UseThing: UseThing.cc Thing.h
g++ $(CFLAGS) -o UseThing UseThing.cc
Alone: Alone.cc
g++ $(CFLAGS) -o Alone Alone.cc
clean:
rm UsePoint UseThing Alone *.o *~
```

# Homework 2

# Document Parsing



The parser produces these WordPosition structures for each file, which are stored in a hash table by word.



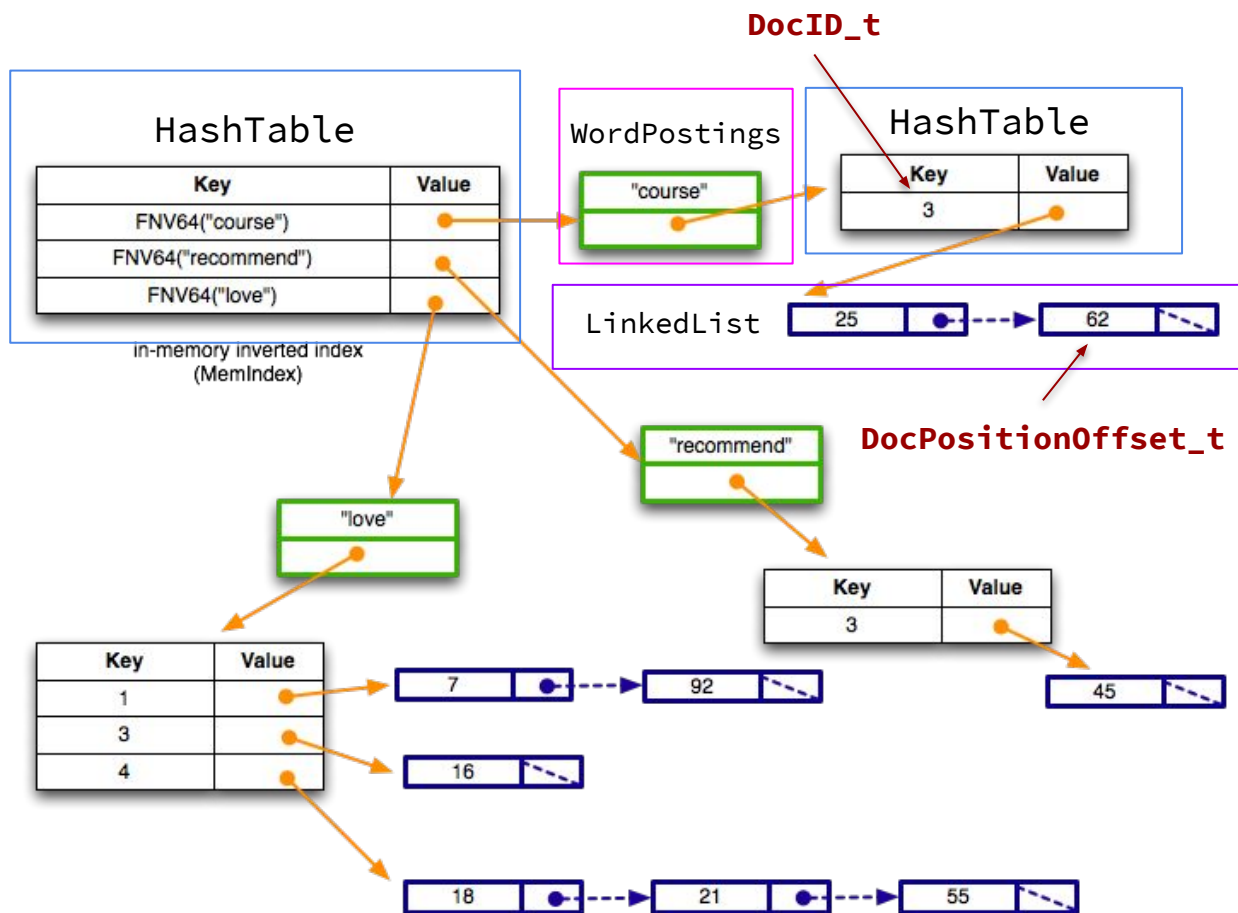
Map document names to IDs.  
FNV64 is a hash function.

# MemIndex

The file crawler builds up a MemIndex structure, which is just a hash table of "word postings."

A word posting is a pair of a word and a hash table with DocID keys and list of positions as a value.

Based on the figure, you can see that the word "course" appeared in a single document with docID 3, at byte offsets 25 and 62 from the start of file. Similarly, the word "love" appears in three documents: docID 1 at positions 7 and 92, docID 3 at position 16, and docID 4 at positions 18, 21, and 55.



**Q&A**

**Thanks for Attending Section!**