CSE 333 Section 4

C++ Const & References; C++ Classes

Some slides referenced from CSE 333 Winter 2020/Spring 2021

Logistics

- **Exercise 6** is out
  - due **Monday** (July 19th), 11:00 PDT
- **HW2** is out
  - due next **Thursday** (July 22nd), 23:59 PDT

Section Plan

- C++ `const` & References
  - Example (Exercise 1)
  - MultChoice class (Exercise 3)
- C++ classes
  - Constructors, Destructors, Initialization Lists

Pointers, References, and Const

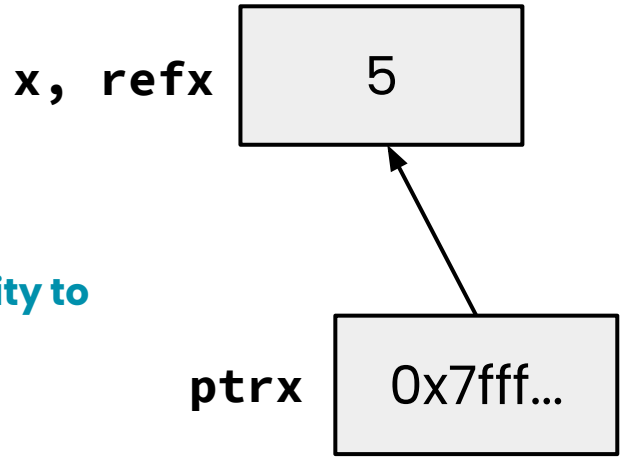
Example

- 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	**References**
Can move to different data via reassignment/pointer arithmetic	References the same data for its entire lifetime - *can't reassign*
Can be initialized to **`NULL`**	No sensible "default reference," must be an alias
Useful for output parameters: `MyClass* output`	Useful for input parameters: `const MyClass& input`

Pointers, References, and Parameters

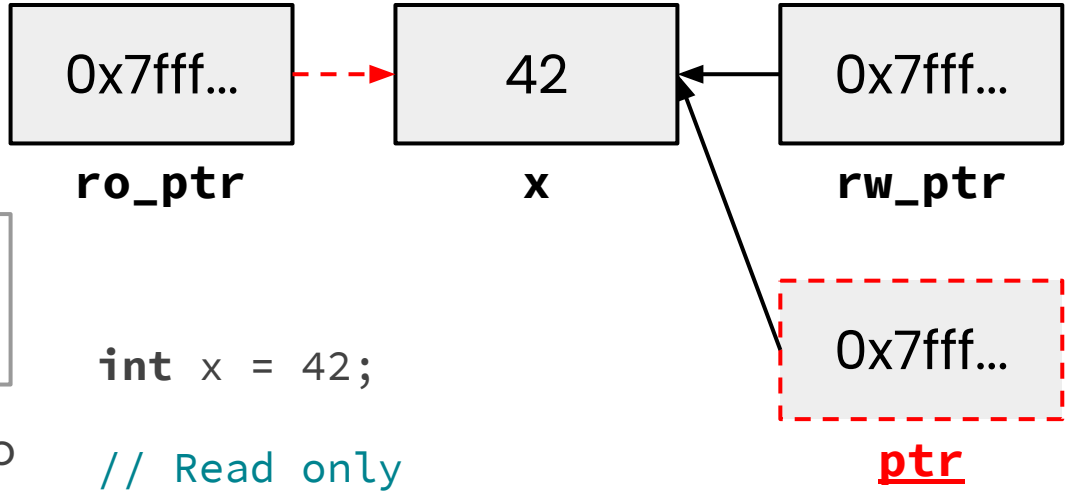
– – –

- 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”

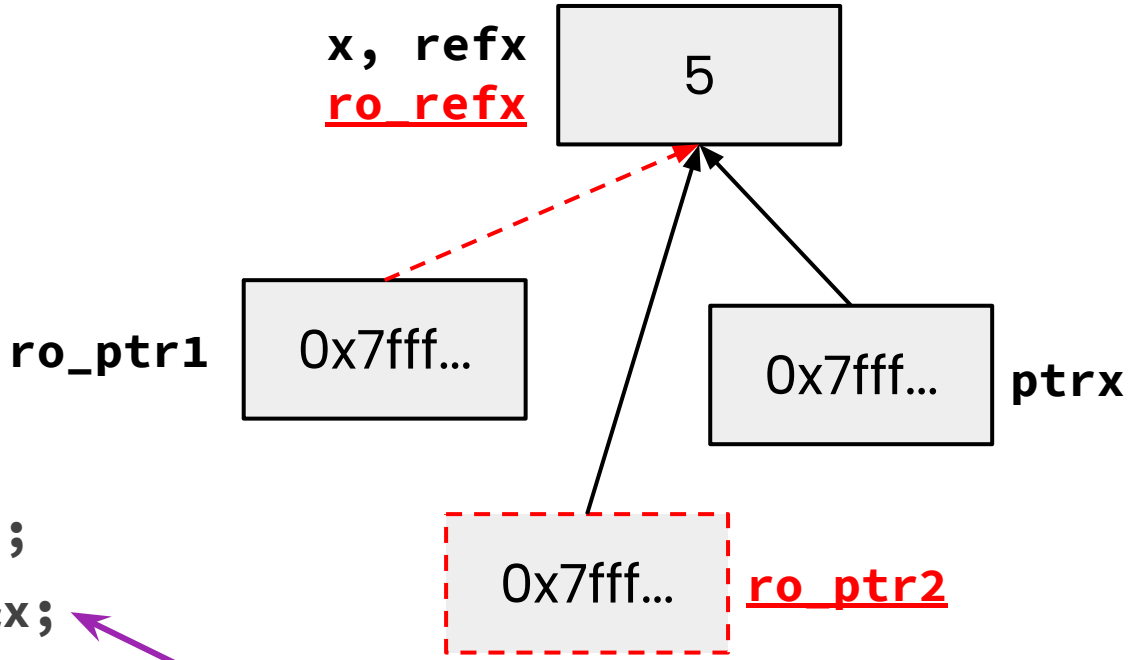
- Mark a variable with `const` to make a compile time check that a variable is never reassigned
- Does not change the underlying write-permissions for this variable




```
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"

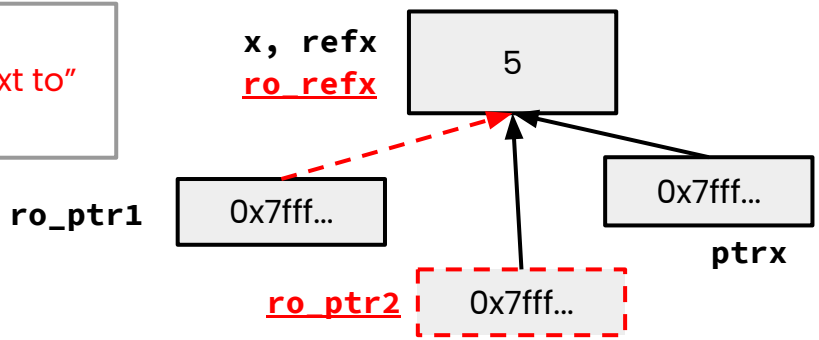
"Pointer to a const int"

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

Exercise 3 (extra practice)

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;`

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 4

Exercise 4

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
```

C++ Classes~~

Preview Questions

- What do the following access modifiers mean?

  `public:` Member is accessible by anyone

  `protected:` Member is accessible by this class and any derived classes

  `private:` Member is only accessible by this class

  `friend:` Allows access of private/protected members to *foreign* functions and/or classes where this modifier is present

- What is the default access modifier for a `struct` in C++?

  A `struct` can be thought of as a class where all members are default public instead of default private. In C++, it is also possible to give member functions (such as a constructor) to a `struct`

Constructors Revisited

```
class Int {
 public:
   Int() { ival = 17; cout << "default(" << ival << ")" << endl; }
   Int(int n) { ival = n; cout << "ctor(" << ival << ")" << endl; }
   Int(const Int &n) {
     ival = n.ival;
     cout << "cctor(" << ival << ")" << endl;
   }
   ~Int() { cout << "dtor(" << ival << ")" << endl; }
};
```

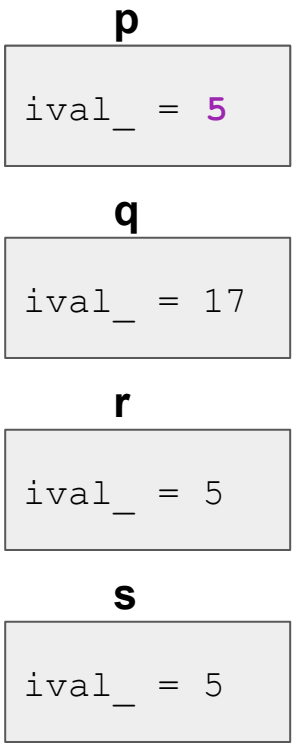
Constructor (ctor): Can define any number as long as they have different parameters. Constructs a new instance of the class.

Copy Constructor (cctor): Creates a new instance based on another instance (must take a reference!). Invoked when passing/returning a **non-reference** object to/from a function.

Destructor (dtor): Cleans up the class instance. Deletes dynamically allocated memory (if any).

What is getting called here?

```
int main() {
  Int p;       // 1. default ctor
  Int q(p);    // 2. copy ctor
  Int r(5);    // 3. 1 arg ctor
  Int s = r;   // 4. copy ctor (cctor)
  p = s;       // 5. assignment operator
}
```



Design Considerations

- What happens if you don't define a copy constructor? Or an assignment operator? Or a destructor? Why might this be bad?
  - In C++, if you don't define any of these, one will be synthesized for you
  - The synthesized copy constructor does a shallow copy of all fields
  - The synthesized assignment operator does a shallow copy of all fields
  - The synthesized destructor calls the default destructors of any fields that have them

- How can you disable the copy constructor/assignment operator/destructor?

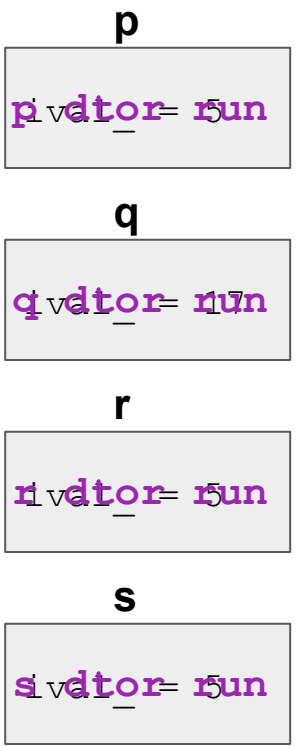
  Set their prototypes equal to the keyword "delete": `~SomeClass() = delete;`

Destructors Review

- When are destructors invoked? In what order are they invoked when multiple objects are getting destructed?
  - An object's destructor is run when it falls out of scope, or when the `delete` keyword is used on heap allocated objects constructed with `new`
  - Invoked in reverse order of construction
- What happens when a destructor actually executes? (Hint: what happens if a dtor body doesn't destruct all of its members?)
  - Destructors are run in reverse order of construction: (1) run destructor body (2) destruct remaining members in reverse order of declaration

When are these destructors run?

```
---

int main() {
  Int p;
  Int q(p);
  Int r(5);
  Int s = r;
  p = s;
}
```



Initialization Lists

– – –

- When is the initialization list of a constructor run, and in what order are data members initialized?

  The initialization list is run before the body of the ctor, and data members are initialized in the order that they are defined in the class, not by initialization list ordering.

- What happens if data members are not included in the initialization list?

  Data members that don't appear in the initialization list are *default initialized/constructed* before ctor body is executed.

Steps for Construction and Destruction

Construction:

1. Construct/initialize members in order of declaration:
    - If: member appears in initialization list, apply initialization
    - Else: default initialize
2. Run constructor body

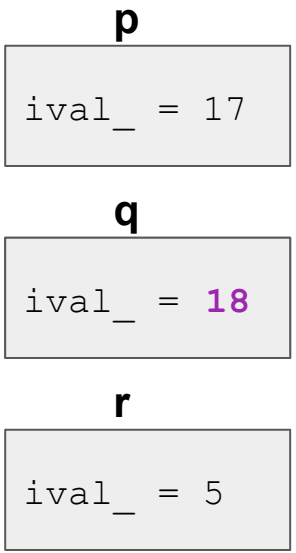
Destruction:

1. Run destructor body
2. Destruct remaining members in reverse order of member declaration

Exercise 6: Constructors and Destructors

```
int main(int argc, char **argv)
{
  Int p;
  Int q(p);
  Int r(5);
  q.set(p.get()+1);
  return EXIT_SUCCESS;
}
```

```
Output:
default(17)
cctor(17)
ctor(5)
get(17)
set(18)
dtor(5)
dtor(18)
dtor(17)
```



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.

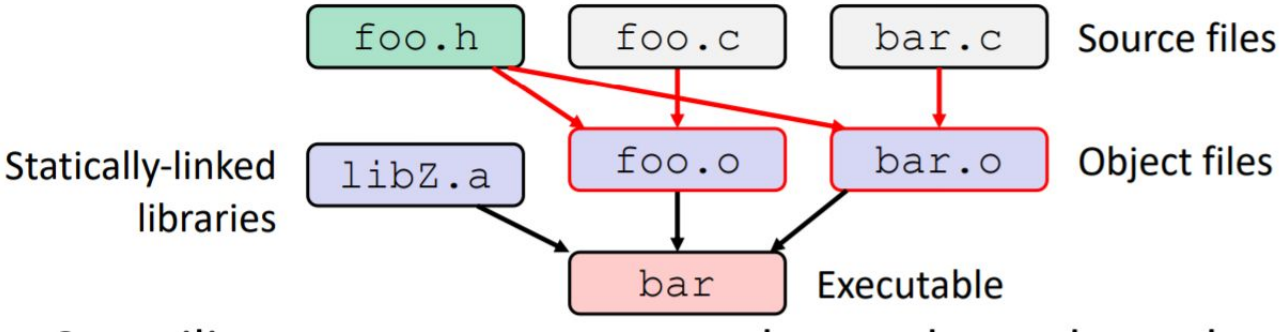
description

Build dag

Build commands

Relate commands to makefile (targets)

clean/all



Exercise 5

Point.h

```
class Point { … };
```

UsePoint.cc

```
#include "Point.h"
#include "Thing.h"
int main( … ) { … }
```

UseThing.cc

```
#include "Thing.h"
int main( … ) { … }
```

Point.cc

```
#include "Point.h"
// defs of methods
```

Thing.h

```
struct Thing { … };
// full struct def here
```

Alone.cc

```
int main( … ) { … }
```

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

https://courses.cs.washington.edu/courses/cse333/21sp/lectures/08/08-make-21sp.pdf#page=24

DAG

Point.h

```
class Point { … };
```

UsePoint.cc

```
#include "Point.h"
#include "Thing.h"
int main( … ) { … }
```

UseThing.cc

```
#include "Thing.h"
int main( … ) { … }
```

Point.cc

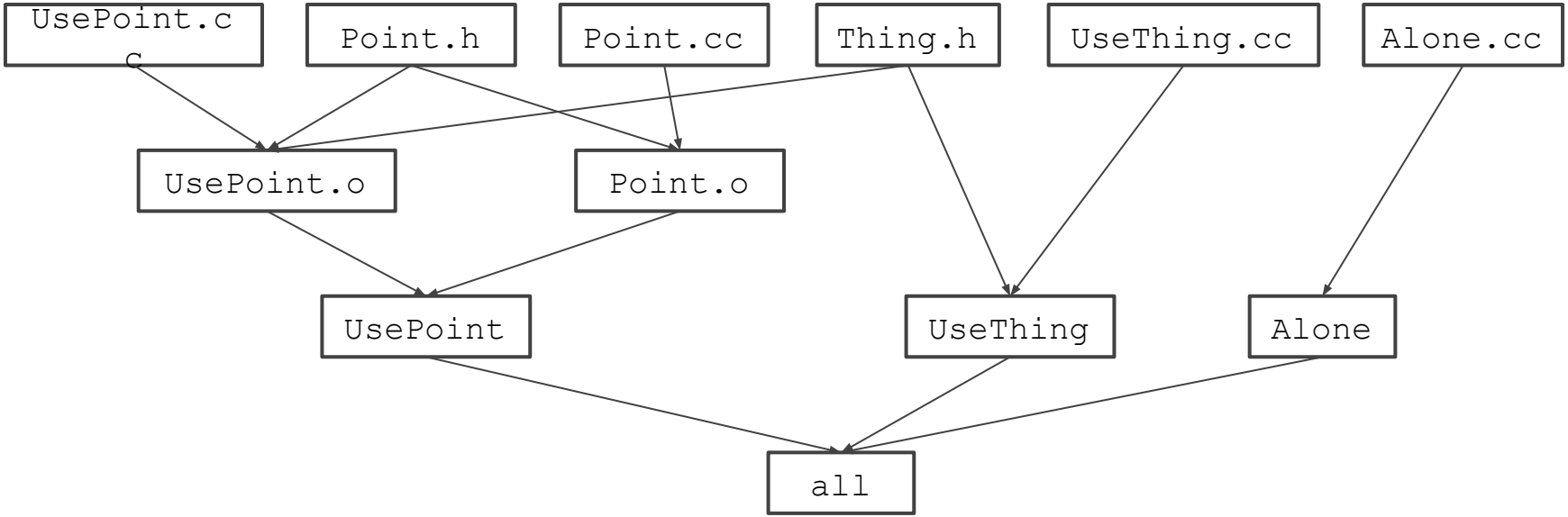
```
#include "Point.h"
// defs of methods
```

Thing.h

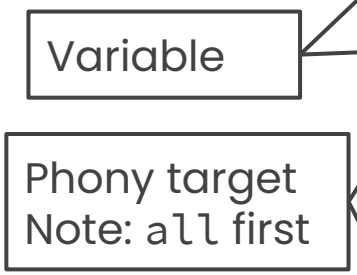
```
struct Thing { … };
// full struct def here
```

Alone.cc

```
int main( … ) { … }
```



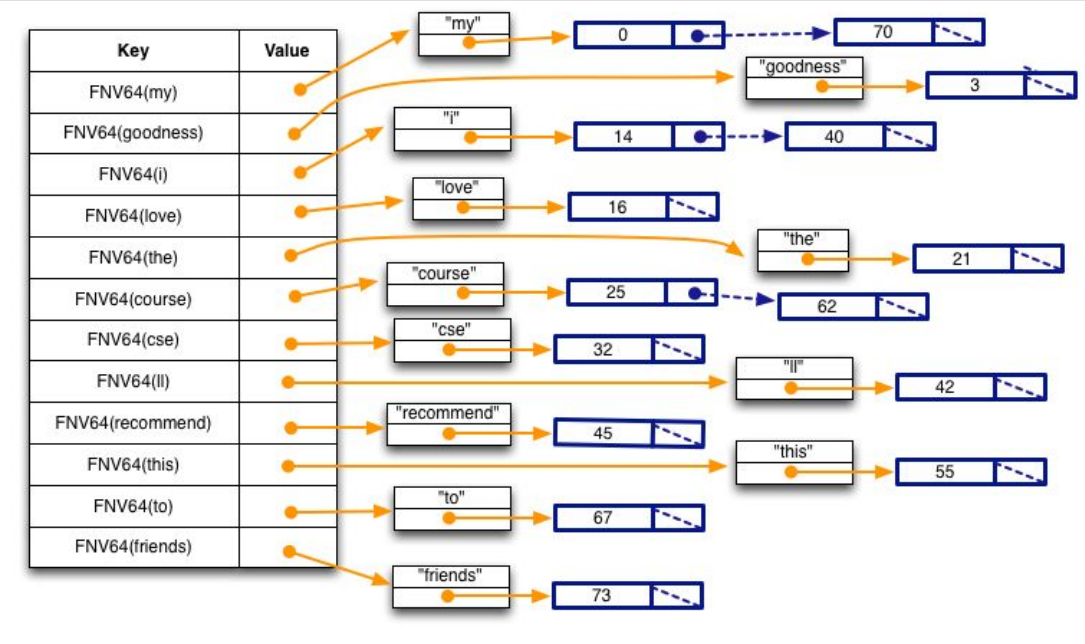
Makefile




```
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.

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

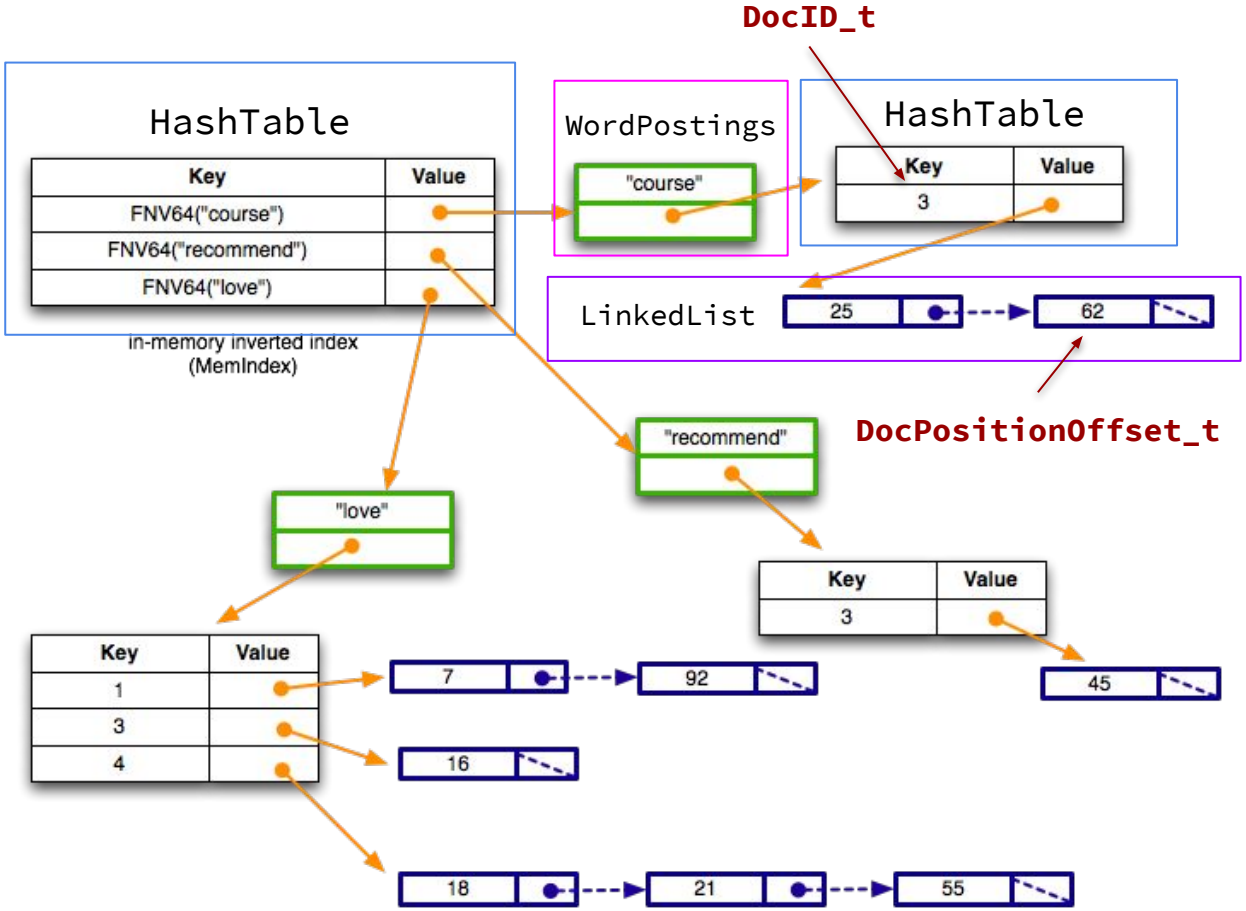
Map document names to IDs. FNV64 is a hash function.

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!