CSE 333 Section AC

Const, References & Make! (w/ Farrell & Travis)

Logistics

Due Friday:

Exercise 8 @ 11 am

Due Monday:

Exercise 9 @ 11 am

Due Thursday 10/24:

Homework 2 @ 9 pm

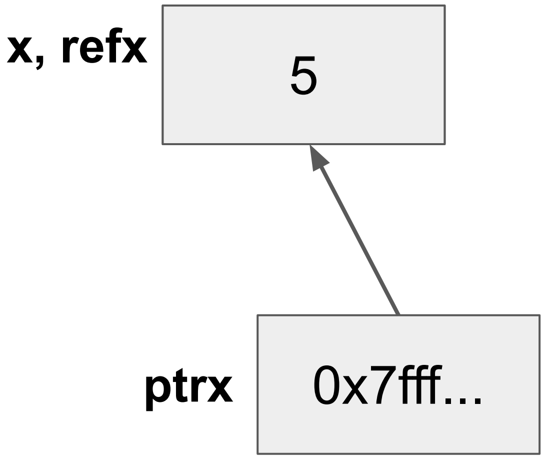
References & Const review

Example

- Consider the following code:

```
int x = 5;
int &refx = x;
int *ptrx = &x;
```

Similar in syntax to the * in pointer declarations



Legend

Red Thing = “can’t change the box it’s next to”
Black = “writeable/readable”

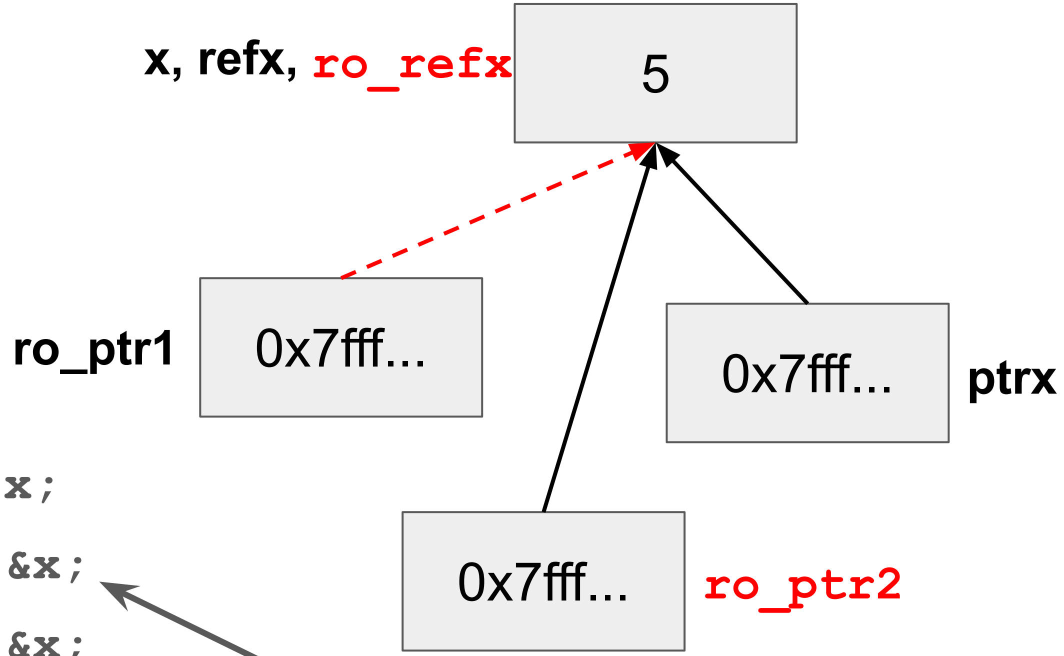
Example

- Consider the following code:

```
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"

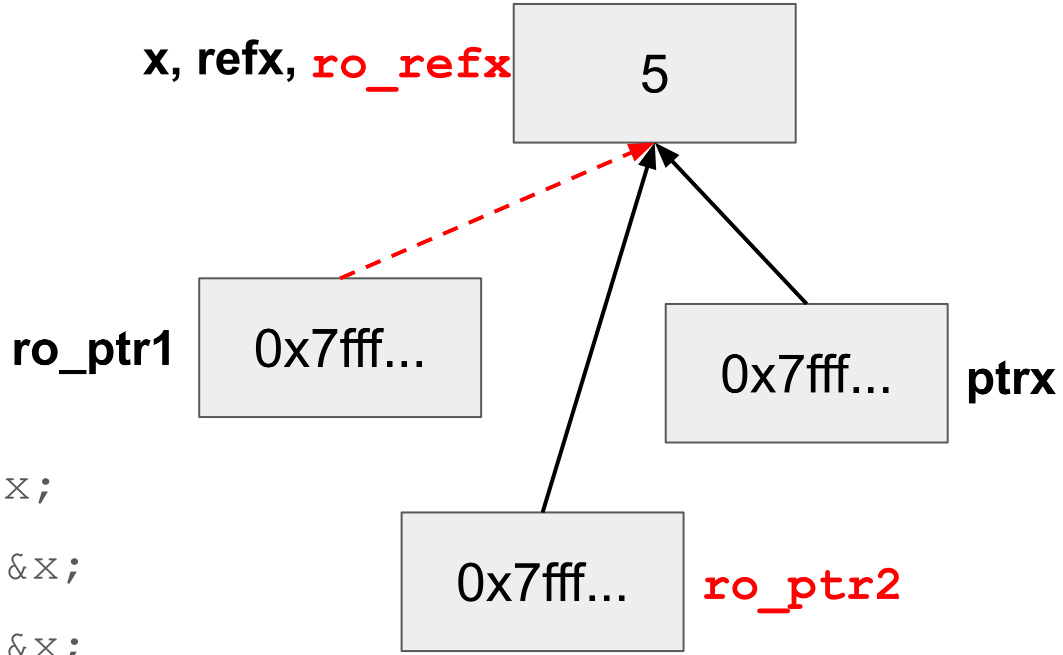
Legend

Red Thing = "can't change the box it's next to"
Black = "writeable/readable"

Example

- Consider the following code:

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



When would you prefer this...

```
void func(int &arg);
```

...to this? Vice-Versa?

```
void func(int *arg);
```

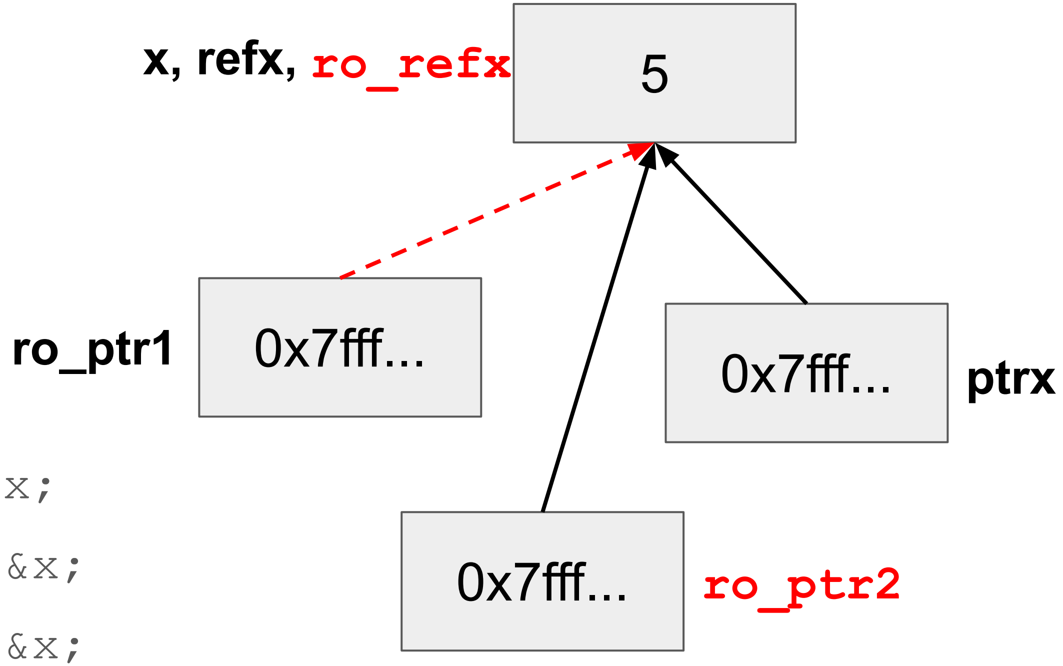
Legend

Red Thing = "can't change the box it's next to"
Black = "writeable/readable"

Example

- Consider the following code:

```
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 results in a compiler error?

```
bar(refx);
bar(ro_refx);
foo(refx);
```

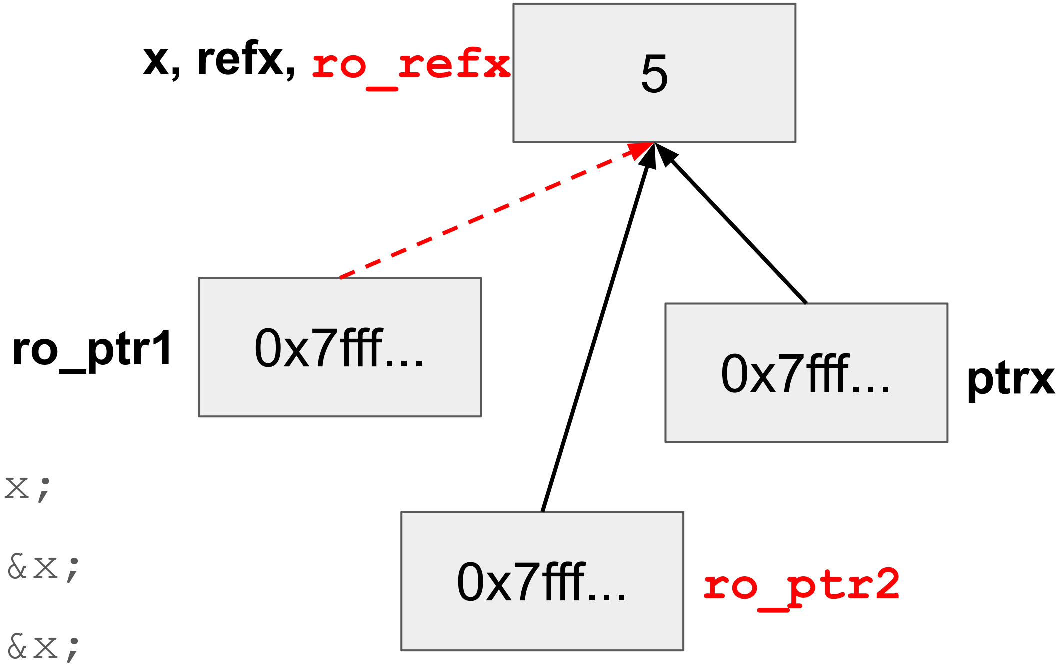
Legend

Red Thing = "can't change the box it's next to"
Black = "writeable/readable"

Example

- Consider the following code:

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



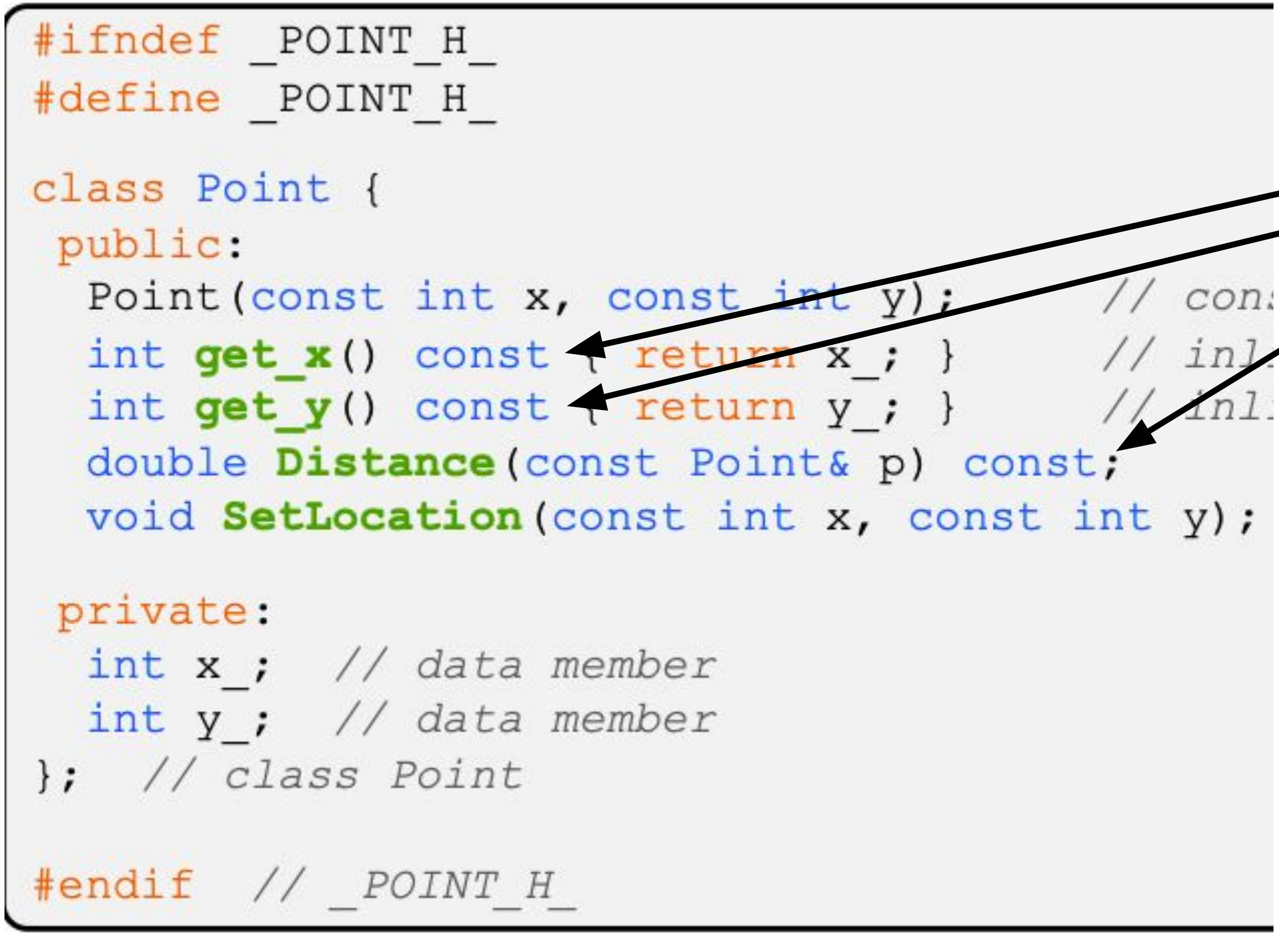
Which results in a compiler error?

```
ro_ptr1 = (int*)0xDEADBEEF;
ptrx = &ro_refx;
ro_ptr2 = ro_ptr2 + 2;
*ro_ptr1 = *ro_ptr1 + 1;
```

Legend

Red Thing = "can't change the box it's next to"
Black = "writeable/readable"

What about “const” object methods?



```
#ifndef _POINT_H_
#define _POINT_H_

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

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

#endif  // _POINT_H_
```

Cannot mutate the object it's called on!

Summary

- Pointers vs. References:

Pointers	References
Can move to different data via reassignment/pointer arithmetic	References the same data for its entire lifetime
Can be initialized to NULL	No sensible "default reference"
"datatype *const ptr" is good style for output parameters within functions *(Unchangeable pointers pointing to changeable data)*	"const datatype &ref" is good style for passing in input values to a function *(Read-only values without copying memory)*

- Const:
    - **Tip:** Read the declaration "right-to-left".
    - Prevent yourself (and clients) from changing data that doesn't make sense to change!

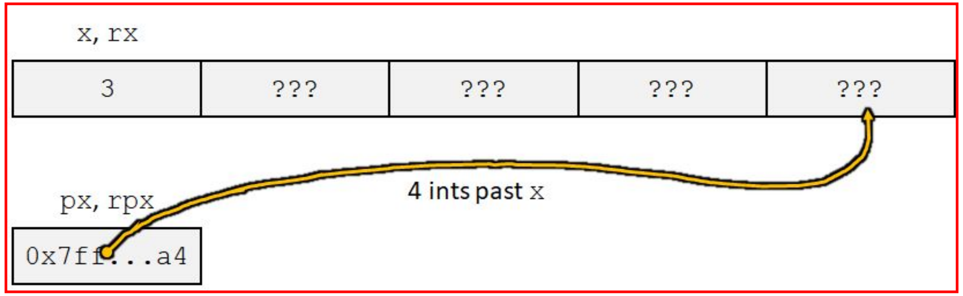
Worksheet Time

2) What does the following program print out? Hint: box-and-arrow diagram!

```
int main(int argc, char** argv) {
  int x = 1;        // assume &x = 0x7ff...94
  int& rx = x;
  int* px = &x;
  int*& rpx = px;

    rx = 2;
  *rpx = 3;
   px += 4;
  cout << "  x: " <<   x << endl;
  cout << " rx: " <<  rx << endl;
  cout << "*px: " << *px << endl;
  cout << " &x: " <<  &x << endl;
  cout << "rpx: " << rpx << endl;
  cout << "*rpx: " << *rpx << endl;

  return 0;
}
```

2) What does the following program print out? Hint: box-and-arrow diagram!

```
int main(int argc, char** argv) {
  int x = 1;       // assume &x = 0x7ff...94
  int& rx = x;
  int* px = &x;
  int*& rpx = px;

    rx = 2;
  *rpx = 3;
   px += 4;
  cout << "  x: " <<   x << endl;    //   x: 3
  cout << " rx: " <<  rx << endl;    //  rx: 3
  cout << "*px: " << *px << endl;    // *px: ??? (garbage)
  cout << " &x: " <<  &x << endl;    //  &x: 0x7ff...94
  cout << "rpx: " << rpx << endl;    // rpx: 0x7ff...a4
  cout << "*rpx: " << *rpx << endl;  // *rpx = *px: ??? (garbage)

  return 0;
}
```



```
struct Thing {
  int a;
  bool b;
};

void PrintThing(const Thing& t) {
  cout << boolalpha << "Thing:  " << t.a << ", " << t.b << endl;
}

int main() {
  Thing foo = {5, true};
  cout << "(0) ";
  PrintThing(foo);

  cout << "(1) ";
  __???__(foo);   // mystery 1
  PrintThing(foo);

  cout << "(2) ";
  __???__(&foo);  // mystery 2
  PrintThing(foo);

  cout << "(3) ";
  __???__(foo);   // mystery 3
  PrintThing(foo);

  return 0;
}
```

Possible Functions:

```
  void f1(Thing t);
  void f2(Thing &t);
  void f3(Thing *t);
  void f4(const Thing &t);
  void f5(const Thing t);
```

Program Output:

```
  (0) Thing:  5, true
  (1) Thing:  6, false
  (2) Thing:  3, true
  (3) Thing:  3, true
```

```
struct Thing {
  int a;
  bool b;
};

void PrintThing(const Thing& t) {
  cout << boolalpha << "Thing:  " << t.a << ", " << t.b << endl;
}

int main() {
  Thing foo = {5, true};
  cout << "(0) ";
  PrintThing(foo);

  cout << "(1) ";
  __???__(foo);   // mystery 1 :   f2
  PrintThing(foo);

  cout << "(2) ";
  __???__(&foo);  // mystery 2 :   f3
  PrintThing(foo);

  cout << "(3) ";
  __???__(foo);   // mystery 3 :  f1, f2, f4, or f5
  PrintThing(foo);

  return 0;
}
```

Possible Functions:

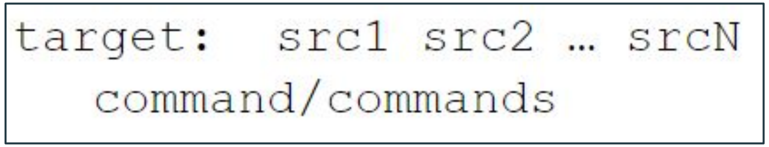
```
  void f1(Thing t);
  void f2(Thing &t);
  void f3(Thing *t);
  void f4(const Thing &t);
  void f5(const Thing t);
```

Program Output:

```
  (0) Thing:  5, true
  (1) Thing:  6, false
  (2) Thing:  3, true
  (3) Thing:  3, true
```

Makefiles, how do they work?

MakeFile Format:



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

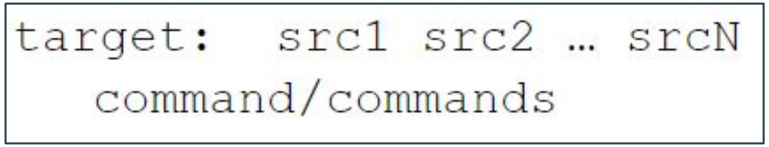
Can type “make <target>” it will attempt to build the target.

// attempts to build by running the supplied commands

- If the target file doesn’t exist, it is rebuilt.
- If a sources are “older” than the target, it will not be rebuilt.
- If a source doesn’t exist or has been updated, target is rebuilt.
- Make will recursively check that sources are up to date.

Makefiles, Phony targets

MakeFile Format:



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

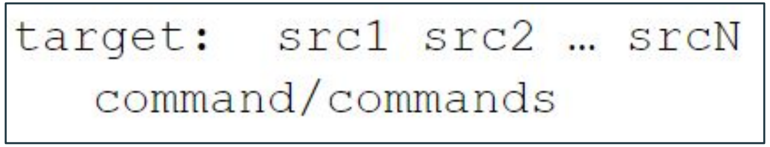
Phony Target: If we list a target, but the command provided doesn’t make a file with the target’s name

Examples:

```
all: <List all executables>
      // no need to provide a command

clean:
      rm <all files we want to delete>
```

Makefiles

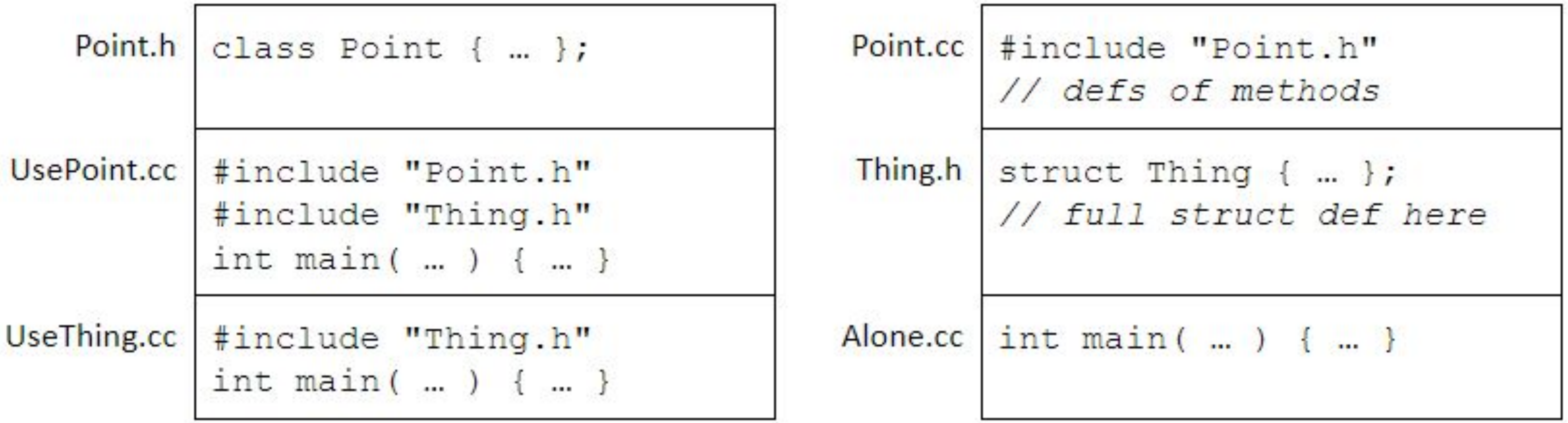
MakeFile Format:



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

The most important part is drawing the dependencies

- .cc files and .h are sources, should not be targets
- .o files are compiled from .cc files. Depends on the source .cc and
  included .h files
- Executables need intermediate .o files if using multiple source .cc files
  Otherwise, can be compiled directly from sources.



Point.h

```
class Point { … };
```

UsePoint.cc

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

UseThing.cc

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

Point.cc

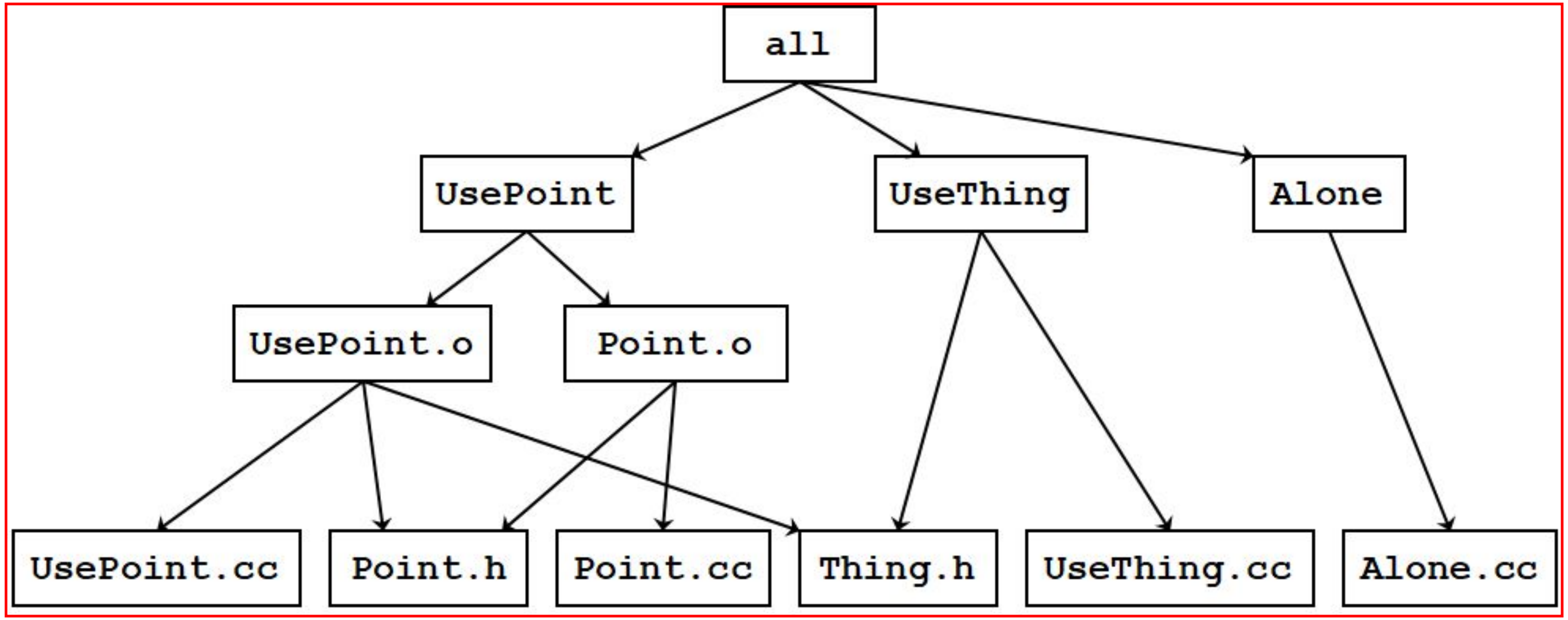
```
#include "Point.h"
// defs of methods
```

Thing.h

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

Alone.cc

```
int main( … ) { … }
```


all
UsePoint
UseThing
Alone
UsePoint.o
Point.o
UsePoint.cc
Point.h
Point.cc
Thing.h
UseThing.cc
Alone.cc