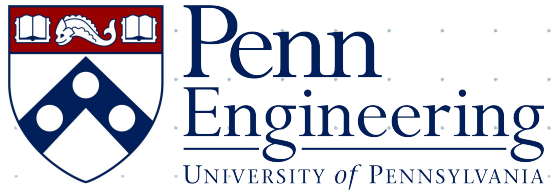
Penn Engineering
University of Pennsylvania

CIS 3990 Recitation 01

References

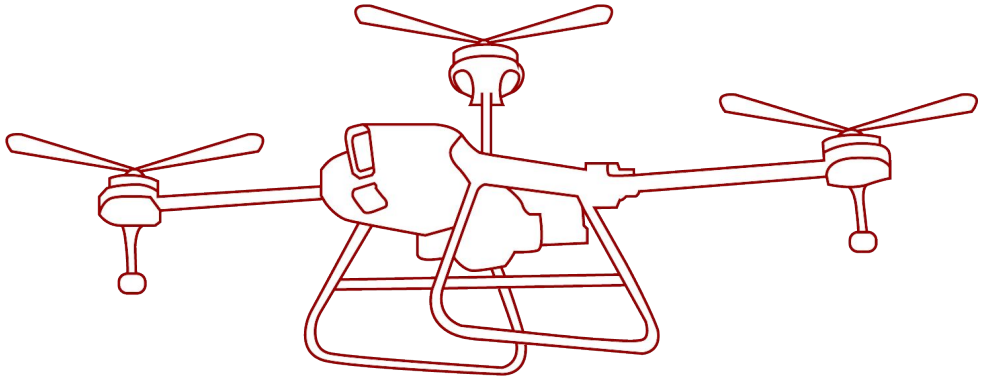
2025-09-04

Agenda

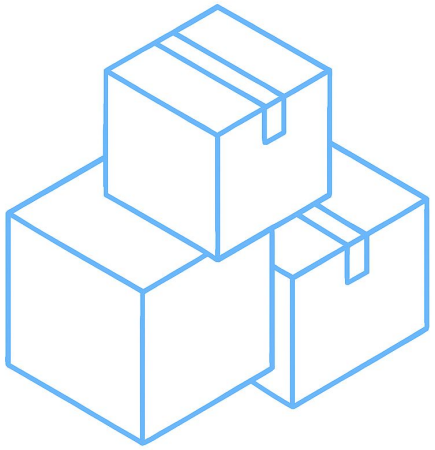


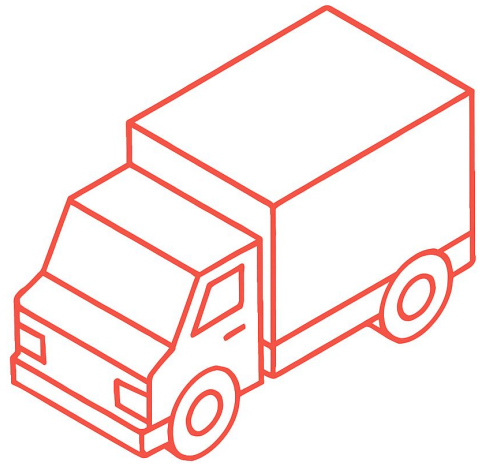
01	Logistics
02	C & C++ don’t lie
03	Pointers vs Refs
04	Const
05	Valgrind demo
06	Ed time

Penn Engineering
University of Pennsylvania



Logistics





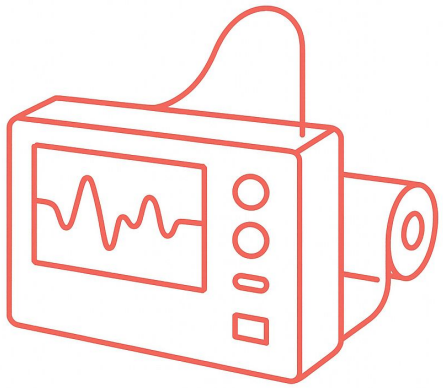
Logistics

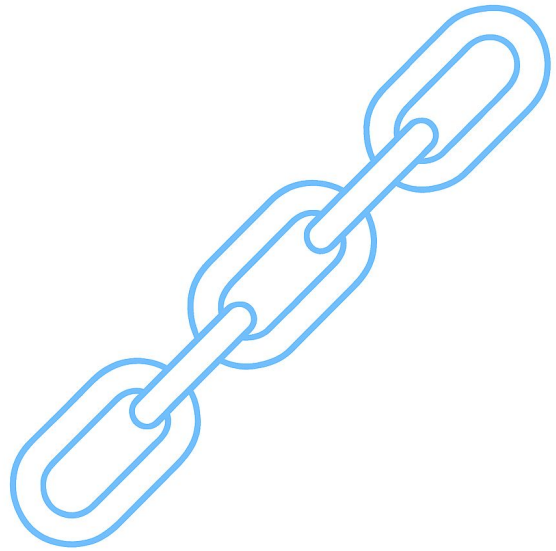
- HW00 past-due - extended to 11:59pm on Tues, Sep 09 - via Gradescope
- HW01 out - due 11:59pm on Tues, Sep 09 - via Gradescope
- Survey00 out - due 11:59pm on Fri, Sep 12 - via Canvas
- If you haven’t, set up github and docker!!
- Github setup - automatic via Gradescope
- Oct 22 - Exam 0 - Please let us know ASAP if you have conflicts
- OH Tomorrow: will be from 4pm to 6:30pm. Waiting on confirmation for permanent reschedule to 2:30pm - 6:30pm

Penn Engineering
University of Pennsylvania

C and C++ don't lie

i.e. Value Semantics





Data types from OOP pov

Penn Engineering
University of Pennsylvania

- Primitive data type - store the actual data we care about

```
int a;
bool b;
float c;
```

- Referential data types - store information about how to get to the data we care about

```
int* p = &a;
char* str;
int& x = a;
```

Python & Java lie

- Higher-level object-oriented languages often give you the impression that you're working with a primitive, when you're actually passing around references

- It is your duty to know what data types are passed by value and what data types are passed by reference

```
1   def i_lie(y):
2       y.append(16)
3
4   z = [20]
5
6   print(z)
7
8   i_lie(z)
9
10  print(z)
```

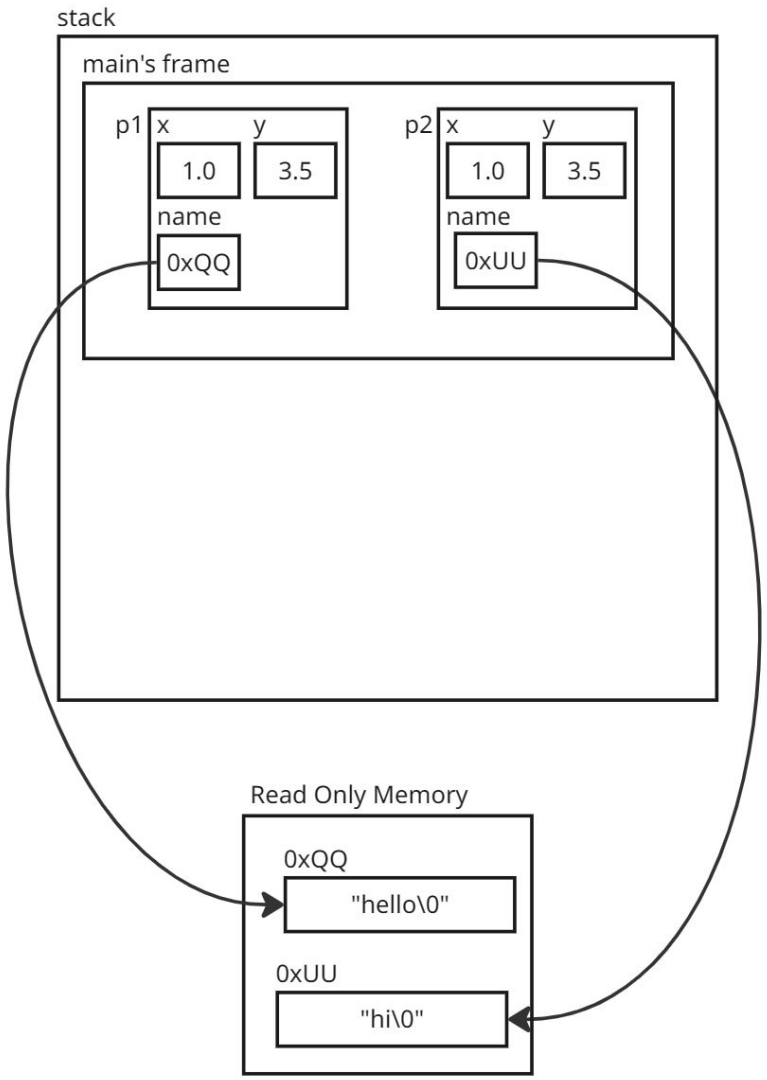
```
Theodor-ThinkPadT15% python lies.py
[20]
[20, 16]
```

In C/C++ you have to ask

- In C and C++, even when we make our own data types via structs they are still primitives
- You specifically have to declare something as a pointer or reference
- A struct may have referential types within it, but those fields are passed by value

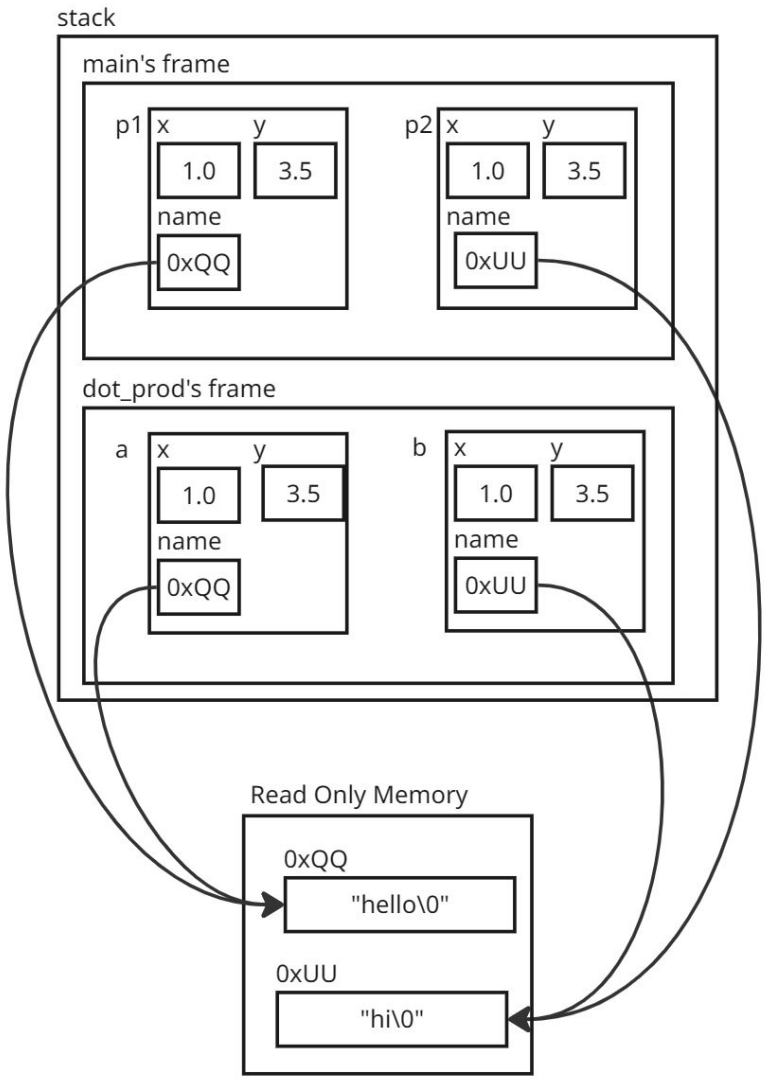
```
struct Point {
  double x;
  double y;
  char* name;
};

double dot_prod(Point a, Point b) {
  return a.x * b.x + a.y * b.y;
}

int main() {
  Point p1 = {1.0, 3.5, "hello"};
  Point p2 = {2.0, 7.5, "hi"};

  cout << dot_prod(p1, p2) << endl;
  exit(EXIT_SUCCESS);
}
```

Structs are (like) primitives!



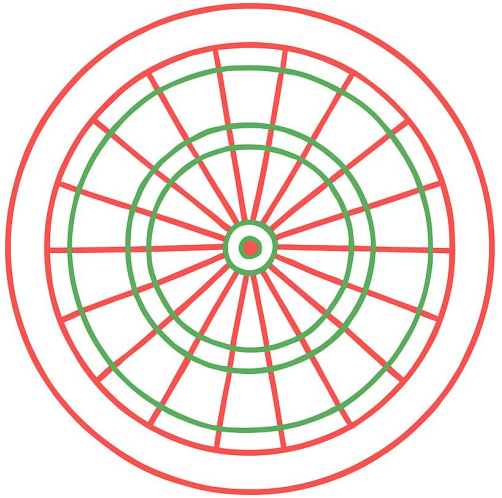

```
struct Point {
  double x;
  double y;
  char* name;
};

double dot_prod(Point a, Point b) {
  return a.x * b.x + a.y * b.y;
}

int main() {
  Point p1 = {1.0, 3.5, "hello"};
  Point p2 = {2.0, 7.5, "hi"};

  cout << dot_prod(p1, p2) << endl;
  exit(EXIT_SUCCESS);
}
```

Structs are (like) primitives!

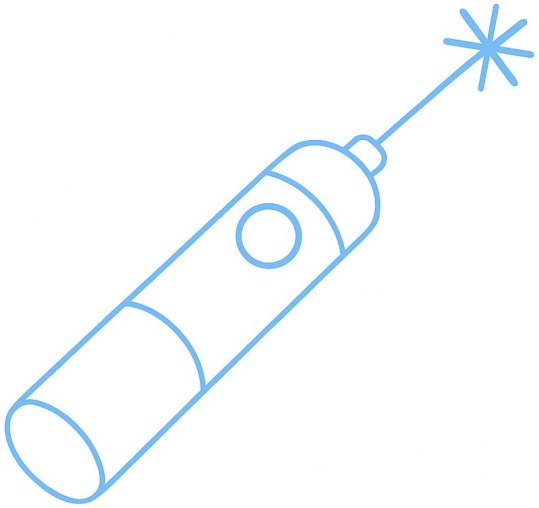


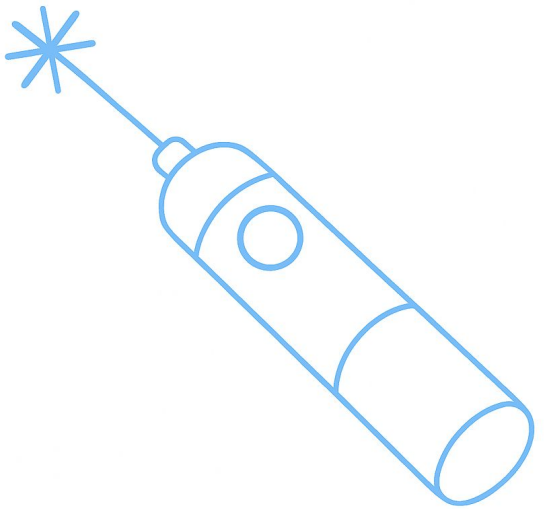

```
struct Point {
  double x;
  double y;
  char* name;
};

double dot_prod(Point a, Point b) {
  return a.x * b.x + a.y * b.y;
}

int main() {
  Point p1 = {1.0, 3.5, "hello"};
  Point p2 = {2.0, 7.5, "hi"};

  cout << dot_prod(p1, p2) << endl;
  exit(EXIT_SUCCESS);
}
```

Penn Engineering
University of Pennsylvania



Pointers & References





Comparison

Penn Engineering
University of Pennsylvania

Pointers

- Referential type that **stores the address** of the data we are interested in
- Data can be read and modified via dereferencing
- The pointer variable itself can be reassigned to point to something else, including null

References

- Referential type that **aliases** the data we are interested in
- Data can be read and modified via “normal access”
- The reference **cannot** be reassigned to alias another piece of data

Okay but like what?

- “A reference is the object, just with another name. It is neither a pointer to the object, nor a copy of the object. It **is** the object. There is no C++ syntax that lets you operate on the reference itself separate from the object to which it refers.”

- Source: https://isocpp.org/wiki/faq/references

What if I have C brain rot

- You are technically correct, since a reference is often implemented using an address in the underlying assembly language, but please do not think of a reference as a funny looking pointer to an object.

- Think of it as the object, the variable, the thing itself, because functionally, in every sense, it is that thing.

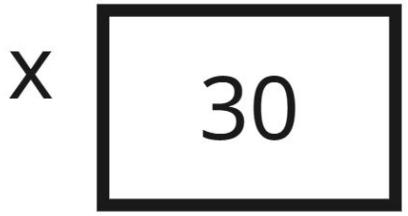
Short demo

- One of the most convenient ways to understand references is to implement a swap function
- How you would have done this in C?

```
24  void my_swap(int & a, int & b) {
25    int c = a;
26    a = b;
27    b = c;
28  }
29
30  int main() {
31    int x = 30;
32    int y = 40;
33    int &z = x;
34
35    my_swap(x, y);
36
37    cout << "x: "<< x << " y: "<< y << " z: " << z;
38    cout << endl;
39    z = 50;
40
41    cout << "x: "<< x << " y: "<< y << " z: " << z;
42    cout << endl;
43  }
```

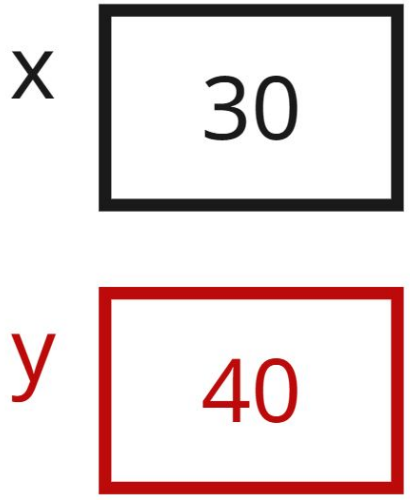
```
PROBLEMS 4   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS   SERIAL MO

moya@5ad86bf379f8:~/shared/ta_3990/rec_code/rec_01$ ./live
x: 40 y: 30 z: 40
x: 50 y: 30 z: 50
moya@5ad86bf379f8:~/shared/ta_3990/rec_code/rec_01$
```

How do I draw this thing



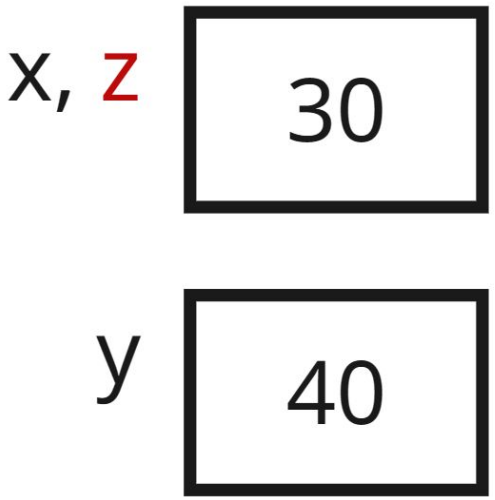

```
void my_swap(int & a, int & b) {
  int c = a;
  a = b;
  b = c;
}

int main() {
  int x = 30;
  int y = 40;
  int &z = x;

  my_swap(x, y);

  cout << "x: "<< x << " y: "<< y << " z: " << z;
  cout << endl;
  z = 50;

  cout << "x: "<< x << " y: "<< y << " z: " << z;
  cout << endl;
}
```

How do I draw this thing



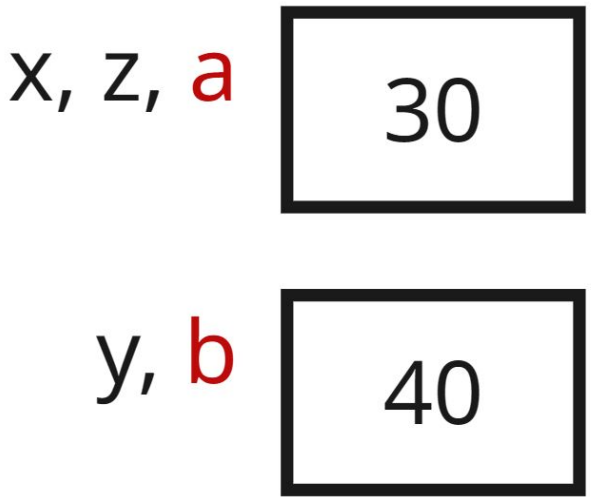

```
void my_swap(int & a, int & b) {
  int c = a;
  a = b;
  b = c;
}

int main() {
  int x = 30;
  int y = 40;
  int &z = x;

  my_swap(x, y);

  cout << "x: "<< x << " y: "<< y << " z: " << z;
  cout << endl;
  z = 50;

  cout << "x: "<< x << " y: "<< y << " z: " << z;
  cout << endl;
}
```

How do I draw this thing



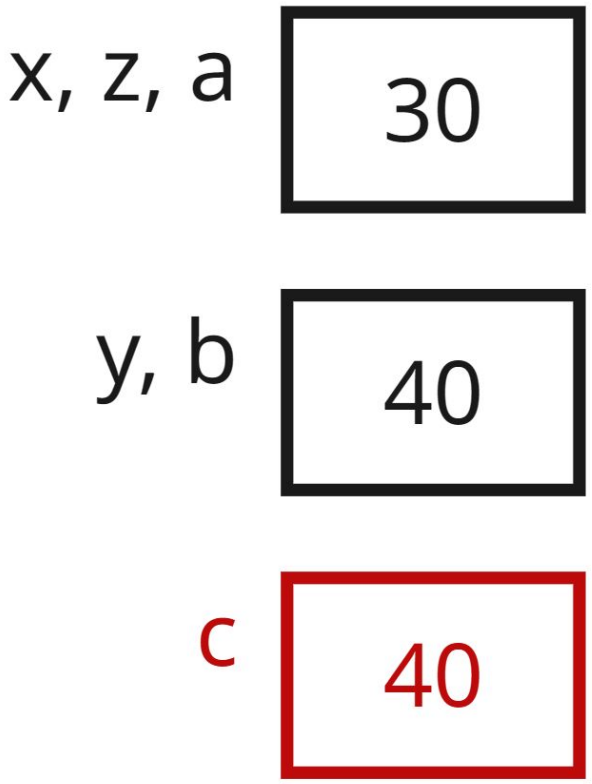

```
void my_swap(int & a, int & b) {
  int c = a;
  a = b;
  b = c;
}

int main() {
  int x = 30;
  int y = 40;
  int &z = x;

  my_swap(x, y);

  cout << "x: "<< x << " y: "<< y << " z: " << z;
  cout << endl;
  z = 50;

  cout << "x: "<< x << " y: "<< y << " z: " << z;
  cout << endl;
}
```

How do I draw this thing



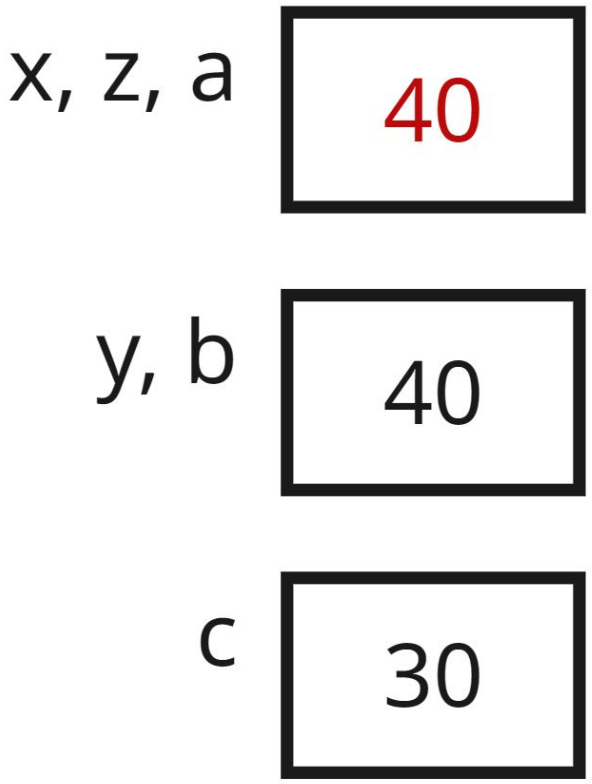

```
void my_swap(int & a, int & b) {
  int c = a;
  a = b;
  b = c;
}

int main() {
  int x = 30;
  int y = 40;
  int &z = x;

  my_swap(x, y);

  cout << "x: "<< x << " y: "<< y << " z: " << z;
  cout << endl;
  z = 50;

  cout << "x: "<< x << " y: "<< y << " z: " << z;
  cout << endl;
}
```

How do I draw this thing



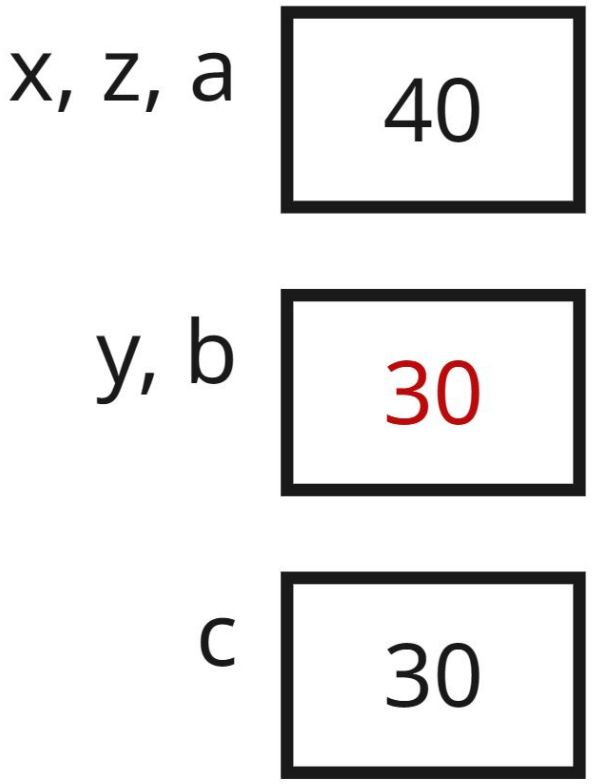

```
void my_swap(int & a, int & b) {
  int c = a;
  a = b;
  b = c;
}

int main() {
  int x = 30;
  int y = 40;
  int &z = x;

  my_swap(x, y);

  cout << "x: "<< x << " y: "<< y << " z: " << z;
  cout << endl;
  z = 50;

  cout << "x: "<< x << " y: "<< y << " z: " << z;
  cout << endl;
}
```

How do I draw this thing



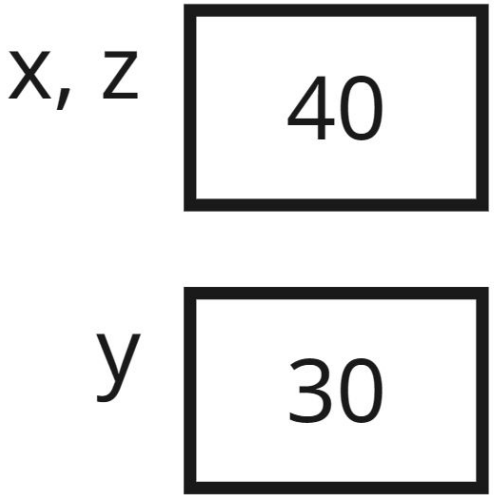

```
void my_swap(int & a, int & b) {
  int c = a;
  a = b;
  b = c;
}

int main() {
  int x = 30;
  int y = 40;
  int &z = x;

  my_swap(x, y);

  cout << "x: "<< x << " y: "<< y << " z: " << z;
  cout << endl;
  z = 50;

  cout << "x: "<< x << " y: "<< y << " z: " << z;
  cout << endl;
}
```

How do I draw this thing




```
void my_swap(int & a, int & b) {
  int c = a;
  a = b;
  b = c;
}

int main() {
  int x = 30;
  int y = 40;
  int &z = x;

  my_swap(x, y);

  cout << "x: "<< x << " y: "<< y << " z: " << z;
  cout << endl;
  z = 50;

  cout << "x: "<< x << " y: "<< y << " z: " << z;
  cout << endl;
}
```

How do I draw this thing

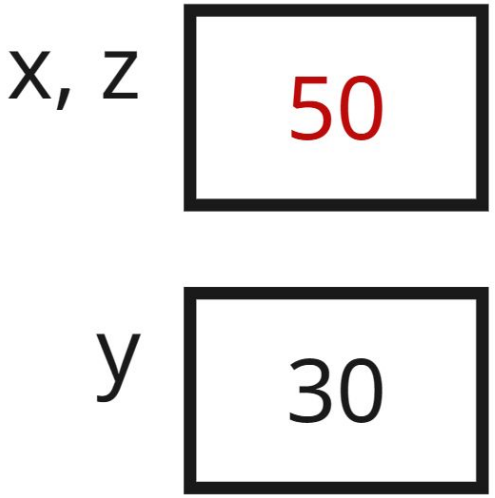


out:

x: 40 y: 30 z: 40

```
void my_swap(int & a, int & b) {
  int c = a;
  a = b;
  b = c;
}

int main() {
  int x = 30;
  int y = 40;
  int &z = x;

  my_swap(x, y);

  cout << "x: "<< x << " y: "<< y << " z: " << z;
  cout << endl;
  z = 50;

  cout << "x: "<< x << " y: "<< y << " z: " << z;
  cout << endl;
}
```

How do I draw this thing



out:

x: 40 y: 30 z: 40

x: 50 y: 30 z: 50

```
void my_swap(int & a, int & b) {
  int c = a;
  a = b;
  b = c;
}

int main() {
  int x = 30;
  int y = 40;
  int &z = x;

  my_swap(x, y);

  cout << "x: "<< x << " y: "<< y << " z: " << z;
  cout << endl;
  z = 50;

  cout << "x: "<< x << " y: "<< y << " z: " << z;
  cout << endl;
}
```

It’s really not a pointer

- We cannot use a pointer to make it “look somewhere else”

```
58  int main() {
59      int x = 20;
60      int & y = x;
61      int * z = &y;
62
63      (*z) = (*z) + 1;
64
65      cout << "x: "<< x << " y: "<< y << endl;
66  }
```

PROBLEMS 4 OUTPUT DEBUG CONSOLE TERMINAL PORTS SERIAL MC

```
moya@5ad86bf379f8:~/shared/ta_3990/rec_code/rec_01$ ./live
x: 21 y: 21
moya@5ad86bf379f8:~/shared/ta_3990/rec_code/rec_01$
```

It’s really not a pointer

- We cannot use a pointer to make it “look somewhere else”

```
58  int main() {
59    int x = 20;
60    int & y = x;
61    int * z = &y;
62
63    (*z) = (*z) + 1;
64
65    cout << "x: "<< x << " y: "<< y << endl;
66  }
```

```
PROBLEMS 4  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  SERIAL MO

moya@5ad86bf379f8:~/shared/ta_3990/rec_code/rec_01$ ./live
x: 21 y: 21
moya@5ad86bf379f8:~/shared/ta_3990/rec_code/rec_01$
```

- Normal pointers allow that

```
58  int main() {
59    int x = 20;
60    int * y = &x;
61    int * z = (int*) (&y);
62
63    (*z) = (*z) + 1;
64
65    cout << "x: "<< x << " y: "<< *y << endl;
66  }
```

```
PROBLEMS 4  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  SERIAL MO

moya@5ad86bf379f8:~/shared/ta_3990/rec_code/rec_01$ ./live
x: 20 y: 16777216
moya@5ad86bf379f8:~/shared/ta_3990/rec_code/rec_01$
```

When do I use one or the other?

Penn Engineering
University of Pennsylvania

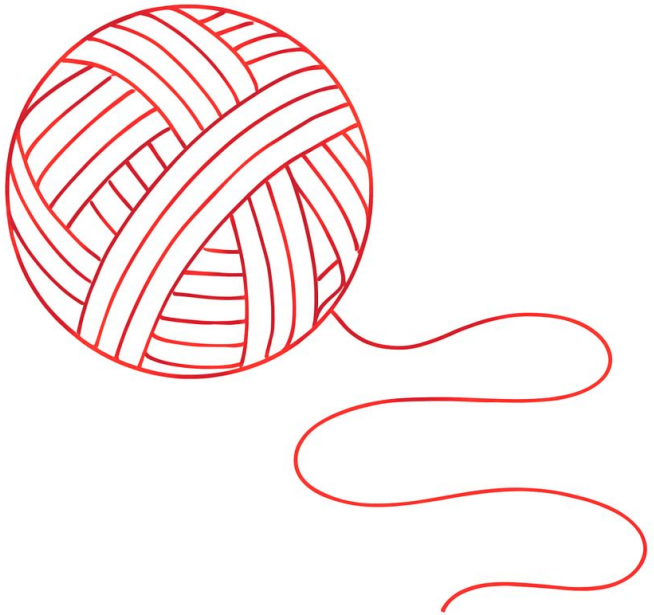
References

- Whenever you can. They solve the problem of having to copy around data without introducing pointer syntax
- It is EXTREMELY hard to shoot yourself in the foot
- Most often in function definitions and operator overloads

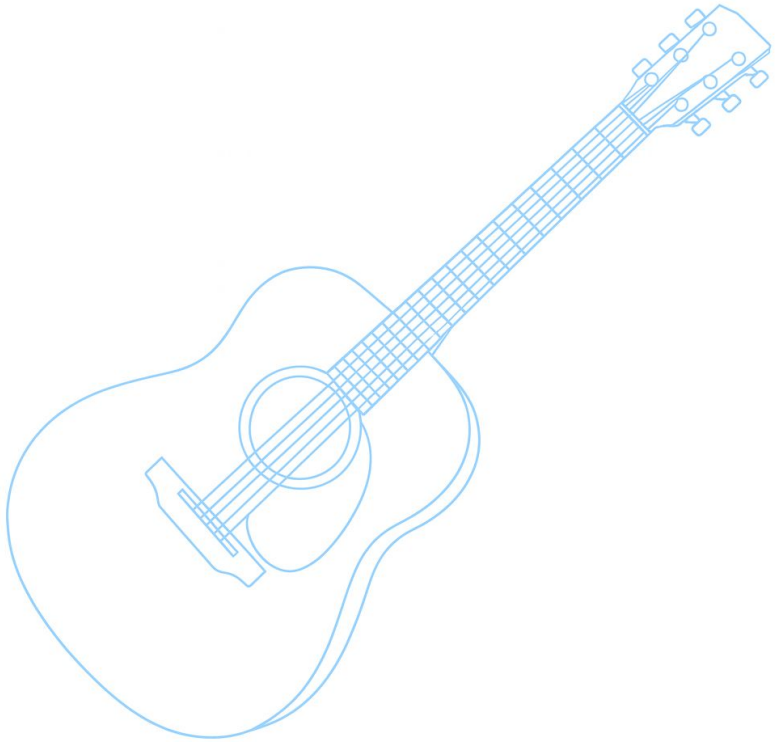
Pointers

- When there is no way to use references *(or smarter C++ stuff)*. A common case is when you need to point to different things during the lifetime of the program, like in a linked list
- When you need nullability, which is not supported by references

Penn Engineering
University of Pennsylvania



Const



Remember from class

- When we want to tell the compiler to not let us modify something, we use **const**
- It will cause compilation errors IF the compiler detects it
- The keyword can be placed in many spots

```
const int a = 20;

int const b = 30;
```

Remember from class

- When we want to tell the compiler to not let us modify something, we use **const**
- It will cause compilation errors IF the compiler detects it
- The keyword can be placed in many spots

```
const int a = 20;
int const b = 30;
```

you cannot change a

you cannot change b

Remember from class

```
const int a = 20;
int const b = 30;

int const * const p = &b;
const int * ptr = &a;
ptr = &b;
```

you **cannot** change what address p is storing and you **cannot** change the data at that address

you **can** change what address ptr is storing but you **cannot** change the data at that address

Remember from class

```
const int a = 20;
int const b = 30;

int const * const p = &b;
const int * ptr = &a;
ptr = &b;

int c = 50;
int * const pc = &c;
(*pc) = (*pc) + 40;
```

you **cannot** change what address pc is storing

you **can** change the data at that address

I feel like I've heard this before

- You can change the data at the address but not the address you're looking at? **Isn't that like a reference?**

```
int c = 50;
int * const pc = &c;
(*pc) = (*pc) + 40;
```

- Yes, it is, but please have them separate in your head
- Also, if you use a constant pointer, you still have to use pointer syntax 🤮

Hw Question

Valgrind demo!

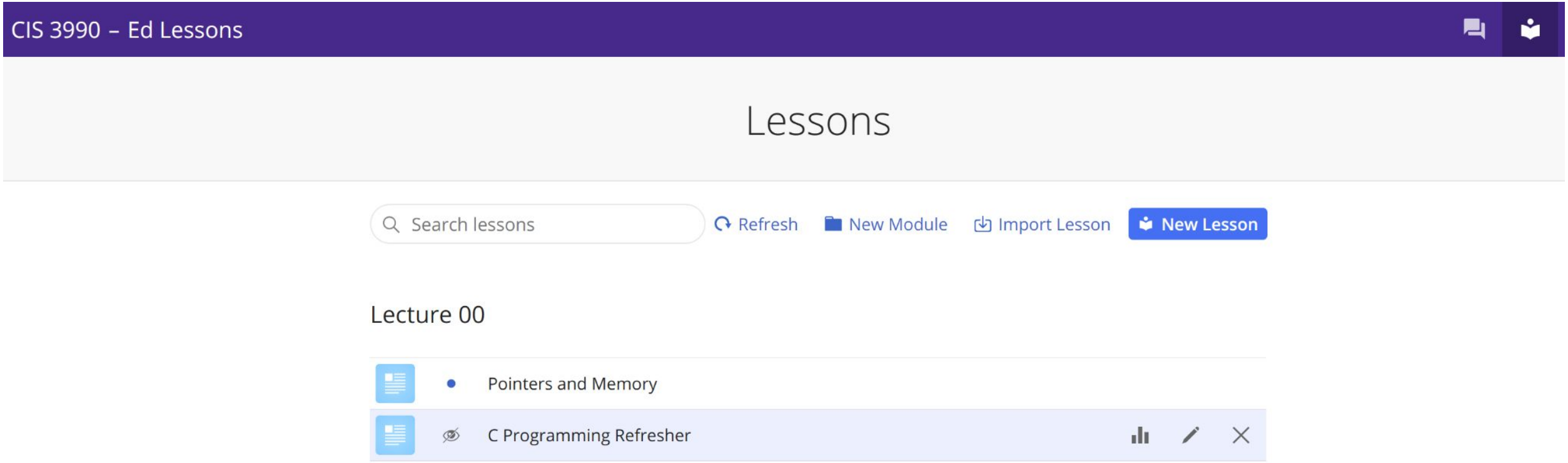
Penn Engineering
University of Pennsylvania

```
1   #include <iostream>
2   #include <cstdlib>
3
4   using namespace std;
5
6   struct Leaky {
7     int x;
8   };
9
10
11  int main(int argc, char **argv) {
12    Leaky **lkyptr = new Leaky *;
13    Leaky *lky = new Leaky;
14
15    cout << "x: " << lky->x << endl;
16
17    lky->x = 30;
18    *lkyptr = lky;
19
20    cout << "lkyptr: " << lkyptr << endl;
21    cout << "lky: " << lky << endl;
22    cout << "x: " << lky->x << endl;
23
24    delete lkyptr;
25    return EXIT_SUCCESS;
26  }
```

- valgrind --leak-check=full ./errors
- valgrind --leak-check=full --track-origins=yes ./errors

Ed practice

Penn Engineering
University of Pennsylvania




That's all Folks!