Day 14
Move Semantics 

Preston Seay && Emily Ai

Special Member Functions Recap

Q: Which functions run when a **`Photo`** is created, copied, or destroyed?

```
1  Photo(int width, int height);              // constructor
2  Photo(const Photo& other);                 // copy constructor
3  Photo& operator=(const Photo& other);      // copy assignment
4  ~Photo();                                  // destructor
```

The compiler writes them for us... unless we manage our own memory!

Warm-up - Photo Dump

https://www.106l.com/live/move-semantics/warm-up?key=fMyJfgqynINZbfadyQGR5zSYTtl_WMO2lINps5tGUw8 ↗

Announcements

- Make sure to check for any pending assignments on 106l.com.

Plan

1. **Motivation: copying costs energy**
2. Move semantics
3. `std::move`
4. The Pirate Code: Rules of 0 and 5

What Drains Your Phone Battery? 🔋

1. Connecting to WiFi
2. Powering the display
3. **Copying data?!**

Yes! Moving bytes around memory is one of the most expensive things a chip does.

Memory Costs Far More Energy Than Math

Adding two `ints`

≈ 0.1 pJ

Reading one `int` from memory

≈ 640 pJ

Reading memory costs about **6,400×** more energy than doing math on it

Source: M. Horowitz, "Computing's Energy Problem," ISSCC 2014 (45 nm chip)

Our Example: The Captain's Camera 📷

```
1  class Photo {
2  public:
3      Photo(int width, int height);
4      Photo(const Photo& other);
5      Photo& operator=(const Photo& other);
6      ~Photo();
7
8  private:
9      int width;
10     int height;
11     int* data;  // the pixels
12 };
```

One photo: `1080 × 1080` = **1,166,400** pixels

Copying it reads **and** writes every pixel: over **2 million** memory accesses

Constructor & Destructor

```
1  Photo::Photo(int width, int height)
2      : width(width),
3        height(height),
4        data(new int[width * height])
         {}
```

Dig up a brand-new treasure chest of pixels

```
1  Photo::~Photo() {
2      delete[] data;
3  }
```

Free the pixels when the photo is destroyed

Copy Constructor & Copy Assignment

```
1  Photo::Photo(const Photo& other)
2      : width(other.width),
3        height(other.height),
4        data(new int[width * height]) {
5      std::copy(other.data,
6                other.data + width * height,
7                data);
8  }
```

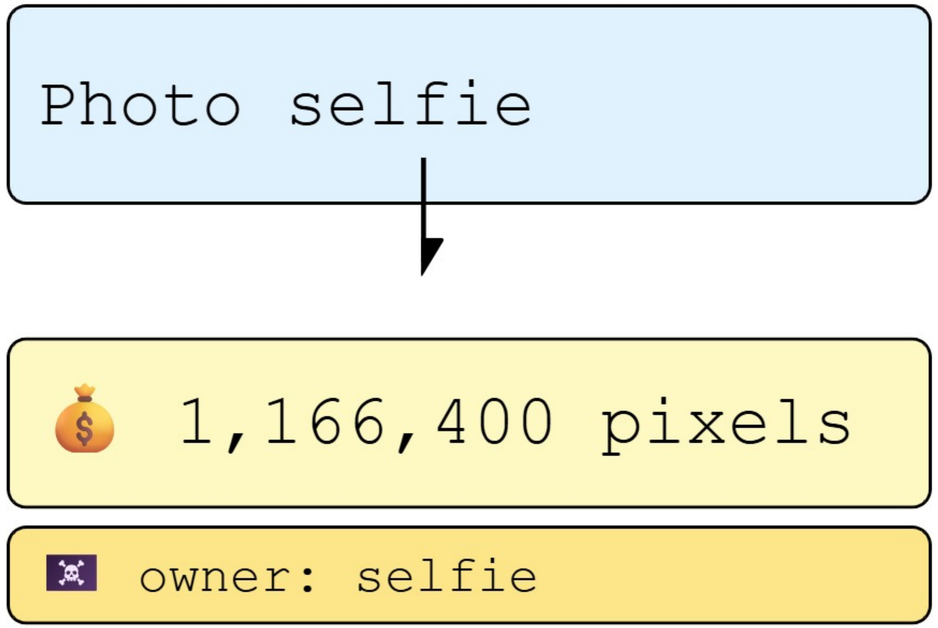
New pixels for a new photo, then copy every one

```
1  Photo& Photo::operator=(
2          const Photo& other) {
3      if (this == &other) return *this;
4      delete[] data;
5      width = other.width;
6      height = other.height;
7      data = new int[width * height];
8      std::copy(other.data,
9          other.data + width * height,
10         data);
11     return *this;
12 }
```

Free our old pixels, then copy every one of theirs

Key Idea: One Treasure, One Owner ☠️

A `Photo` **owns** its pixels: they're its treasure 💰, and only its destructor may `delete[]` them.



Two photos must **never** own the same pixels. Otherwise both destructors would `delete[]` them!

Think About It With a Partner!

Which special member functions run at **(A)** and **(B)**?

```
1  Photo takePhoto();
2
3  int main() {
4      Photo selfie = takePhoto();  // (A)
5
6      Photo retake(0, 0);
7      retake = takePhoto();        // (B)
8  }
```

(A) nothing! The compiler builds it in place (**copy elision**)

(B) copy assignment, then the destructor

The Problem: A Wasted Copy

```
1  retake = takePhoto();
```

takePhoto() (temporary)

```
width = 1080
height = 1080
data = 0x1024
```

💰 1,166,400 pixels

☠️ owner: takePhoto()

retake

```
width = 1080
height = 1080
data = 0x7f3a
```

💰 1,166,400 pixels (copy)

☠️ owner: retake

Copy assignment duplicates every pixel... then the temporary is destroyed!

What If We Just Took the Treasure?

The temporary is about to be destroyed anyway. Nobody will ever look at it again...

Instead of copying its pixels, `retake` could plunder them: just take the pointer!


TED Talk
How to Be a Pirate
with Preston Seay
TED

Step 1: Take the Pointer

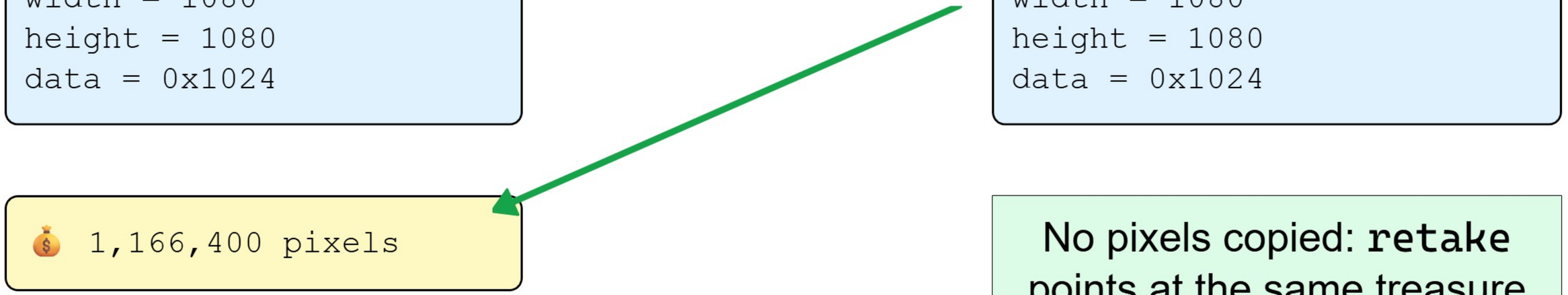
```
1  retake = takePhoto();
```

takePhoto() (temporary)

```
width = 1080
height = 1080
data = 0x1024
```

retake

```
width = 1080
height = 1080
data = 0x1024
```



No pixels copied: `retake` points at the same treasure

But now **two** photos point to one treasure. Who owns it?

Step 2: Leave the Old Owner Empty-Handed

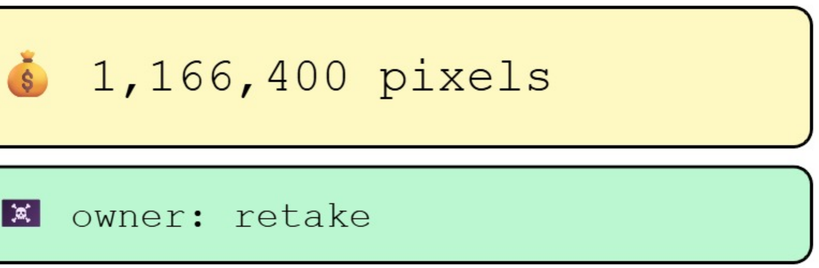
```
1  retake = takePhoto();
```

takePhoto() (temporary)

```
width = 1080
height = 1080
data = nullptr
```

retake

```
width = 1080
height = 1080
data = 0x1024
```



One treasure, one owner, and **zero** pixels copied. That's a **move**!

Even Pirates Have a Code

```
1 void post(Photo whoAmI) {
2     Photo selfie = whoAmI;  // plunder??
3     whoAmI.getPixel(21, 24);
4 }
```

`whoAmI` is used on the next line!

Only plunder objects nobody will use again: copy persistent things, **move temporaries**

Recap: lvalues, rvalues & &&

lvalue: has a location (address) and name.

(yar)**rvalue**: a readable, temporary value. (PLUNDERABLE)

```
1  void upload(Photo& pic);   // lvalues land here
2  void upload(Photo&& pic);  // rvalues land here
3
4  upload(selfie);        // calls upload(Photo&)
5  upload(takePhoto());   // calls upload(Photo&&)
```

Overload on & and &&, and the compiler sorts the loot for you!

Activity: Keep or Toss?

106l.com → Move Semantics → slide 2

Plan

1. Motivation: copying costs energy
2. **Move semantics**
3. `std::move`
4. The Pirate Code: Rule of 0 and 5

Copy Constructor vs. Move Constructor

```
1  Photo::Photo(const Photo& other)
2      : width(other.width),
3        height(other.height),
4        data(new int[width * height]) {
5      std::copy(other.data,
6                other.data + width * height,
7                data);
8  }
```

New treasure for a new owner: copies every pixel

```
1  Photo::Photo(Photo&& other)
2      : width(other.width),
3        height(other.height),
4        data(other.data) {
5      other.data = nullptr;
6  }
```

Take the treasure, leave `other` empty-handed

Copy Assignment vs. Move Assignment

```
1  Photo& Photo::operator=(
2          const Photo& other) {
3      if (this == &other) return *this;
4      delete[] data;
5      width = other.width;
6      height = other.height;
7      data = new int[width * height];
8      std::copy(other.data,
9          other.data + width * height,
10         data);
11     return *this;
12 }
```

```
1  Photo& Photo::operator=(
2          Photo&& other) {
3      if (this == &other) return *this;
4      delete[] data;
5      width = other.width;
6      height = other.height;
7      data = other.data;
8      other.data = nullptr;
9      return *this;
10 }
```

Toss our old treasure overboard first!

Activity: Steal the Pixels

106l.com → Move Semantics → slides 3 & 4

Plan

1. Motivation: copying costs energy
2. Move semantics
3. **`std::move`**
4. The Pirate Code: Rules of 0 and 5

Forcing Move Semantics: `std::move`

What if **we** know an lvalue won't be used again?

```
1  void insert(std::vector<Photo>& album, const Photo& pic, int pos) {
2      album.push_back(Photo(1, 1));
3      for (int i = album.size() - 1; i > pos; i--)
4          album[i] = album[i - 1];  // copies every photo!
5      album[pos] = pic;
6  }
```

```
1          album[i] = std::move(album[i - 1]);  // plunders instead
```

`std::move` says "I'm done with this one." Now it's an rvalue!

`std::move` Doesn't Move Anything!

- `std::move` is just a **cast** from lvalue to rvalue
- The move constructor or move assignment does the plundering
- After a move, the old owner is empty-handed: only **assign to** or **destroy** it
- Use it only when you're sure the object won't be used again

```
1  Photo selfie = std::move(whoAmI);
2  whoAmI.getPixel(21, 24);  // 💥 use-after-move
```

Plan

1. Motivation: copying costs energy
2. Move semantics
3. `std::move`
4. **The Pirate Code: Rules of 0 and 5**

The Pirate Code: Rules of 0 and 5

- **Rule of Zero:** no treasure of your own to guard? Write **none** of the special member functions
- **Rule of Five:** wrote those three? Add **both moves** too, or every move becomes a slow copy

Most classes should follow the rule of zero: let `std::vector` and `std::string` guard the treasure!

Final Thoughts 

1. Copying data costs time **and** energy
2. Every resource should have **one owner**. Copying makes a new one; moving hands it over
3. `&&` lets functions (and SMFs!) spot temporaries and plunder them
4. `std::move` casts an lvalue to an rvalue. Use it carefully!
5. Rule of zero when you can, rule of five when you must