Copying data is expensive

Quantifying the Energy Cost of Data Movement for Emerging Smart Phone Workloads on Mobile Platforms

Dhinakaran Pandiyan and Carole-Jean Wu
School of Computing, Informatics, and Decision Systems Engineering
Arizona State University
Tempe, Arizona 85281
Email: {dpandiya,carole-jean.wu}@asu.edu

moving data for a wide range of popular smart phone workloads. We find that a considerable amount of total device energy is spent in data movement (an average of 35% of the total device energy). Our results also indicate a relatively high stalled cycle

[source]

Copying data is expensive

Quantifying the Energy Cost of Data Movement in Scientific Applications

Gokcen Kestor*, Roberto Gioiosa*, Darren J. Kerbyson*, Adolfy Hoisie*
* Pacific Northwest National Laboratory
{gokcen.kestor, roberto.gioiosa, darren.kerbyson, adolfy.hoisie}@pnnl.gov

exascale systems. Projections show that the cost of moving data from memory is two orders of magnitudes higher than the cost of computing a double-precision register-to-register floating point operation. These

[source]

Last Time

- Special member functions handle the class lifecycle
  - Copy constructor `Type::Type(const Type& other);`
  - Copy assignment operator `Type& Type::operator=(const Type& other);`
  - Destructor `Type::~Type();`
- Compiler creates these for us
  - But… if we’re managing memory, we need to override

Introducing... the Photo class

```cpp
class Photo {
public:
   Photo(int width, int height);
   Photo(const Photo& other);
   Photo& operator=(const Photo& other);
   ~Photo();
private:
   int width;
   int height;
   int* data;
};
```

Photo Constructor

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

Photo SMF: Copy Constructor

```
Photo::Photo(const Photo& other)
  : width(other.width)
  , height(other.height)
  , data(new int[width * height])
{
  std::copy(other.data, other.data + width * height, data);
}
```

Photo SMF: Copy Assignment

```
Photo &Photo::operator=(const Photo& other) {
   // Check for self assignment
   if (this == &other) return *this;

   delete[] data; // Clean up old pixels!

   // Copy over new pixels!
   width = other.width;
   height = other.height;
   data = new int[width * height];
   std::copy(other.data, other.data + width * height, data);
   return *this;
}
```

Photo SMF: Destructor

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

Your Turn

What special member functions get called at **(A)** and **(B)** below?

```
Photo takePhoto();

int main() {
   Photo coolestSelfie = takePhoto(); // (A) Copy Destruct   RVO
   Photo retake(0, 0);
   retake = takePhoto();              // (B) Assign Destruct
}
```

RVO = Return Value Optimization (compiler optimization)

Your Turn

What happened?!

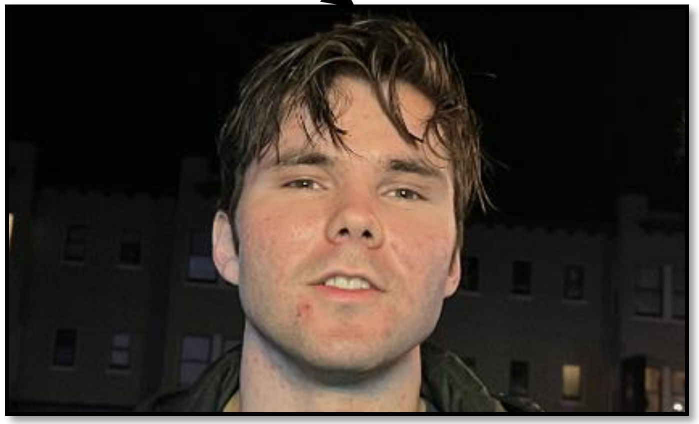
```
Photo takePhoto();

int main() {
   Photo coolestSelfie = takePhoto(); // (A) ~~Copy Destruct~~ RVO
   Photo retake(0, 0);
   retake = takePhoto();             // (B) Assign Destruct
}
```

The Problem

takePhoto()

Photo

- width = 3840
- height = 2160
- data = 0x1024c3bd



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

The Problem

takePhoto()

Photo
- width = 3840
- height = 2160
- data = 0x1024c3bd

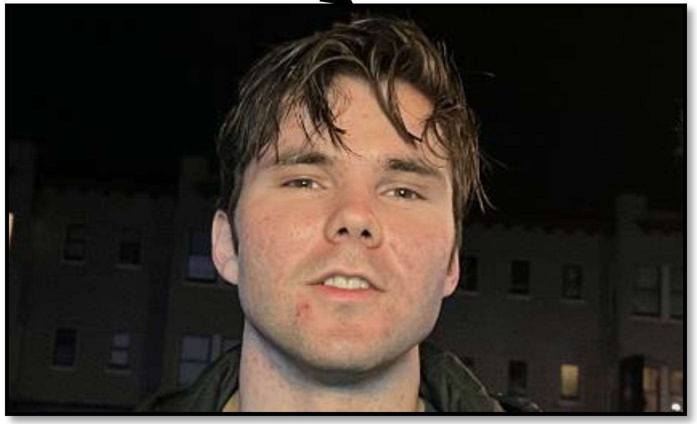


Photo (retake)
- width = 0
- height = 0
- data = 0x153713f1

```
retake = takePhoto(); // Copy assignment
```

The Problem

takePhoto()

Photo
- width = 3840
- height = 2160
- data = 0x1024c3bd

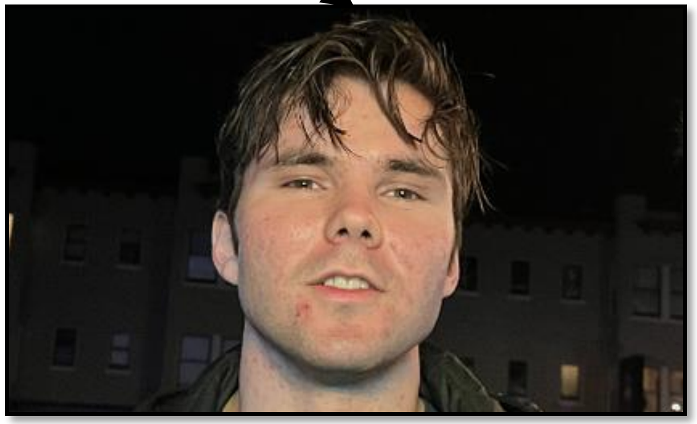
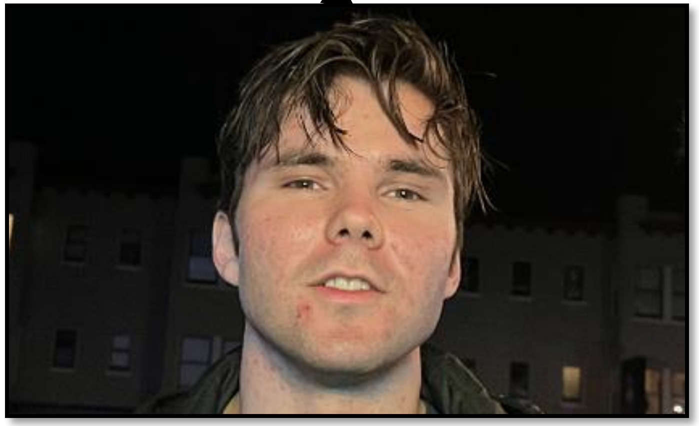


Photo (retake)
- width = 3840
- height = 2160
- data = 0x133210f1



```
retake = takePhoto(); // Copy assignment
```

The Problem

takePhoto()

☠ Photo
- width = 3840
- height = 2160
- data = 0x1024c3bd

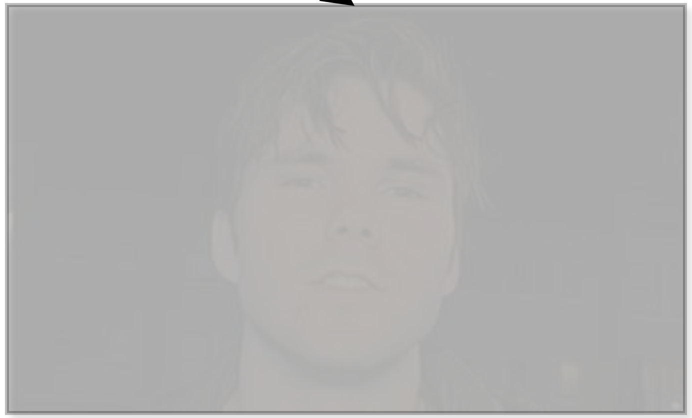
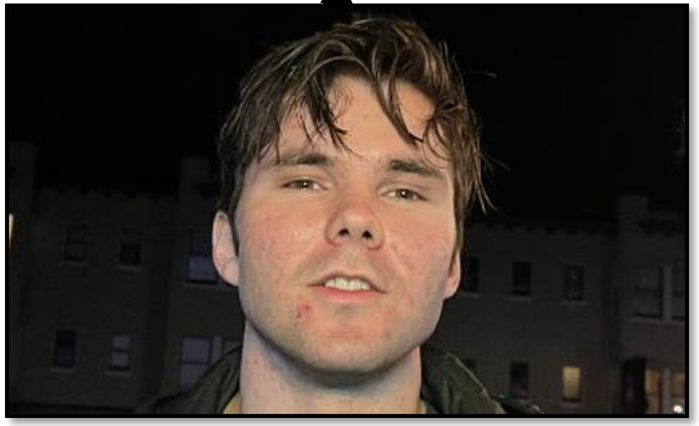


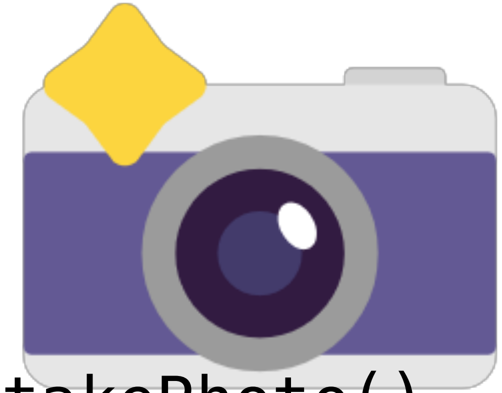
Photo (retake)
- width = 3840
- height = 2160
- data = 0x133210f1



```
retake = takePhoto(); // Destructor
```

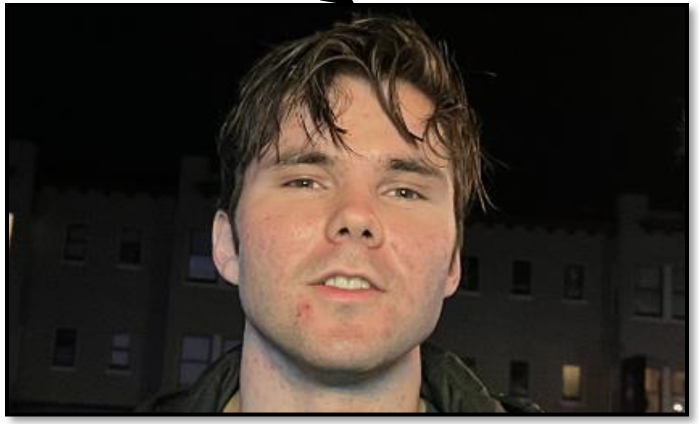
What if we could reuse the memory instead?

The Solution: Move Semantics



takePhoto()

Photo
- width = 3840
- height = 2160
- data = 0x1024c3bd



The Problem

takePhoto()

Photo
- width = 3840
- height = 2160
- data = 0x1024c3bd

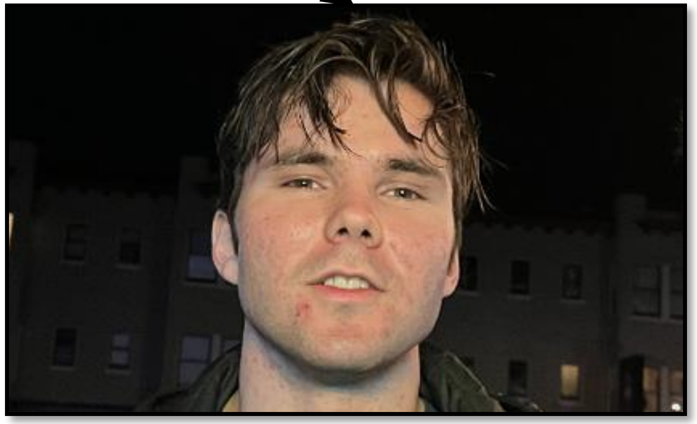


Photo (retake)
- width = 0
- height = 0
- data = 0x133210f1

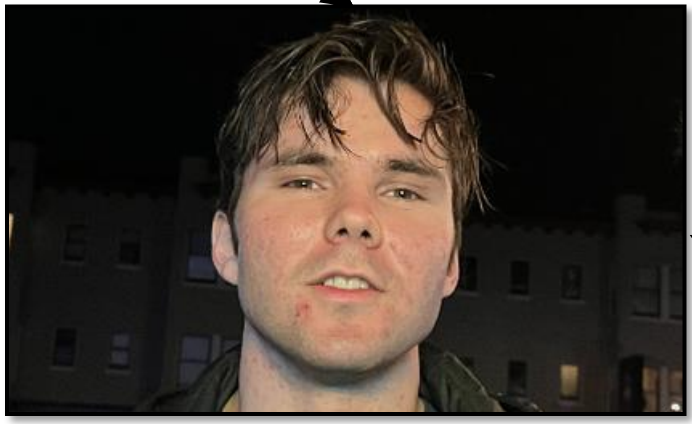
```
retake = takePhoto(); // Move assignment
```

The Problem

takePhoto()

Photo
- width = 3840
- height = 2160
- data = 0x1024c3bd

Photo (retake)
- width = 3840
- height = 2160
- data = 0x1024c3bd



retake = takePhoto(); // ~~Copy~~ Move assignment

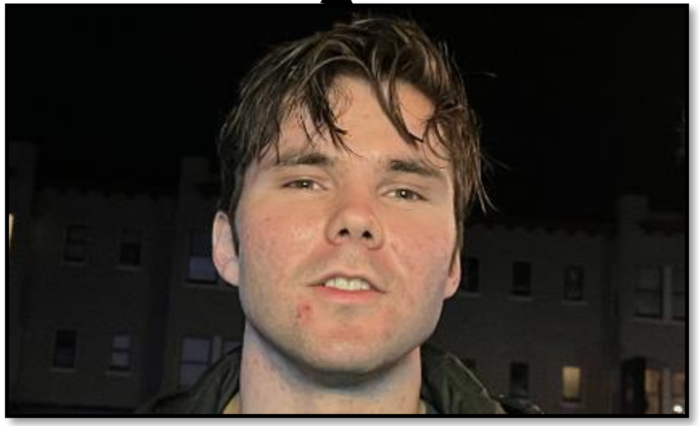
The Problem

takePhoto()

Photo
- width = 3840
- height = 2160
- data = 0x0

nullptr

Photo (retake)
- width = 3840
- height = 2160
- data = 0x1024c3bd



retake = takePhoto(); // ~~Copy~~ Move assignment

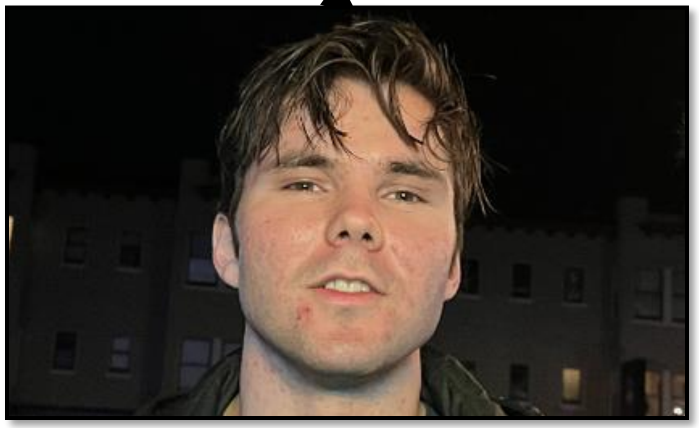
The Problem

takePhoto()

☠ **Photo**
- width = 3840
- height = 2160
- data = 0x0

nullptr

Photo (retake)
- width = 3840
- height = 2160
- data = 0x1024c3bd



```
retake = takePhoto(); // Destructor
```

We move `takePhoto()` because it's temporary

Move vs. Copy Semantics

`takePhoto()` is temporary, so we can steal its resources!

```
Photo takePhoto();

int main() {
   Photo retake(0,0);
   retake = takePhoto();  // Move takePhoto()
                          // because it's temporary!
}
```

Move vs. Copy Semantics

Is it always safe to move objects? Assume `get_pixel` accesses data

```
Photo takePhoto();

void foo(Photo whoAmI) {
   Photo selfie = whoAmI;       // What if we move here?
   whoAmI.get_pixel(21, 24);    // ???
}
```

What will happen if we try to run this code?

Move vs. Copy Semantics

❌ Since `selfie` stole `whoAmI`'s data, we end up dereferencing `nullptr`

```
Photo takePhoto();

void foo(Photo whoAmI) {
  Photo selfie = whoAmI;      // What if we move here?
  whoAmI.get_pixel(21, 24);   // ❌ use-after-move
}
```

Move vs. Copy Semantics

```
selfie1 = selfie2;
// copy persistent objects (e.g. variables)

retake = takePhoto();
// move temporary objects (e.g return values)
```

How does the compiler know whether to move or copy?

lvalues & rvalues

lvalues and rvalues generalize the idea of “temporariness” in C++

```
void foo(Photo pic) {
  Photo beReal = pic;
  Photo insta = takePhoto();
}
```

`pic` is an lvalue!

`takePhoto()` is an rvalue!

lvalues & rvalues

lvalues have a definite address, **rvalues** do not!

```
void foo(Photo pic) {
  Photo* p1 = &pic;
  Photo* p2 = &takePhoto(); // ❌ Doesn't work!
}
```

`pic` is an lvalue!

We can take its address!

`takePhoto()` is an rvalue!

We **cannot** take its address!

An lvalue can appear on either side of an =

```
x = y;
y = 5;
```

An rvalue can appear only right of an =

```
x = 5;
```

~~5 = y;~~

Your Turn

Which of the following right hand assignments are rvalues?

- Hint: which ones have a definite address?

```
int          a = 4;                       rvalue
int&         b = a;                       lvalue
vector<int>  c = {1, 2, 3};               rvalue
int          d = c[1];                    lvalue
int*         e = &c[2];                   rvalue
size_t       f = c.size();                rvalue
int          g = static_cast<int>(f);     lvalue
```

An lvalue's lifetime is until the end of scope

An rvalue's lifetime is until the end of line

Working towards move semantics

If we have an `lvalue`, how can we avoid copying its memory?

```
void uploadToInsta(Photo pic);

int main() {
   Photo selfie = takePhoto(); // selfie is lvalue
   uploadToInsta(selfie); // 🤦 Unnecessary copy is made here
}
```

Working towards move semantics

We can pass by reference! 🥳

```
void uploadToInsta(Photo& pic);

int main() {
  Photo selfie = takePhoto(); // selfie is lvalue
  uploadToInsta(selfie); // ✅ No copy is made here
}
```

Working towards move semantics

- How can we avoid copying **rvalues**?
- What happens if we try to pass by reference?

```
void uploadToInsta(Photo& pic);

int main() {
  uploadToInsta(takePhoto()); // Does this work?
}
```

❌ `candidate function not viable: expects lvalue as 1st argument`

lvalue reference

```
void upload(Photo& pic);

int main() {
  Photo selfie = takePhoto();
  upload(selfie);
}
```

rvalue reference

```
void upload(Photo&& pic);

int main() {
  upload(takePhoto());
}
```

We can do whatever we want with **Photo&&** pic, it's temporary!

A few important points

- **lvalue** references
  - Syntax: Type&
  - Persistent, must keep object in valid state after function terminates
- **rvalue** references
  - Syntax: Type&&
  - Temporary, we can steal (move) its resources
  - Object might end up in an invalid state, but that's okay! It's temporary!

```cpp
//Hello. I want to take your Widget and play with it. It may be in a
//different state than when you gave it to me, but it'll still be yours
//when I'm finished. Trust me!
void foo(Widget& w);
```

```cpp
//Hello. Ooh, I like that Widget you have. You're not going to use it
//anymore, are you? Please just give it to me. Thank you! It's my
//responsibility now, so don't worry about it anymore, m'kay?
void foo(Widget&& w);
```

[source]

Key Idea: Overloading & and && parameters distinguish lvalue and rvalue references

lvalue/rvalue overloading

```
void upload(Photo& pic);

int main() {
   Photo selfie = takePhoto();
   upload(selfie);
}
```

```
void upload(Photo&& pic);

int main() {
   upload(takePhoto());
}
```

Compiler decides which version of **upload** to call depending on whether argument is lvalue or rvalue!

What we want!

Photo&

```
Photo selfie1 = selfie2;
// copy persistent objects (e.g. variables)
```

Photo&&

```
retake = takePhoto();
// move temporary objects (e.g return values)
```

Let’s overload the special member functions!

Copy constructor

```
Photo::Photo(const Photo& other)
    : width(other.width)
    , height(other.height)
    , data(new int[width * height])
{
    std::copy(
        other.data,
        other.data + width * height,
        data
    );
}
```

Move constructor

```
Photo::Photo(Photo&& other)
: width(other.width)
, height(other.height)
{
    // other is temporary
    // Let’s steal its
    // resources since we know
    // it’s about to be gone!
}
```

```
Photo::Photo(Photo&& other)
  : width(other.width)
  , height(other.height)
  , data(other.data)
{
  log() << "Photo(Photo&&)" << std::endl;
  other.data = nullptr;
}
```

Let’s overload the special member functions!

Copy assignment operator

```
Photo& Photo::operator=(const Photo& other) {
    if (this == &other) return *this;
    delete[] data;
    width = other.width;
    height = other.height;
    data = new int[width * height];
    std::copy(other.data, other.data +
    width * height, data);
    return *this;
}
```

Move assignment operator

```
Photo&
Photo::operator=(Photo&& other)
{
    // other is temporary
    // Let’s steal its
    // resources since we know
    // it’s about to be gone!
}
```

```
Photo& Photo::operator=(Photo&& other)
{
  log() << "Photo::operator=(Photo&&)" << std::endl;

  // Clean up our data before assigning into this
  delete[] data;

  width = other.width;
  height = other.height;
  data = other.data;
  other.data = nullptr;

  return *this;
}
```

Forcing Move Semantics

- Usually, we let the compiler decide between **&** and **&&**
- Is that always the most efficient choice?
  - E.g. what if we know that an lvalue will never be used again?

Forcing Move Semantics

Line 3 *copies* each element into its new spot, even though the original value is never used again

```
1 void PhotoCollection::insert(const Photo& pic, int pos) {
2     for (int i = size(); i > pos; i--)
3         elems[i] = elems[i - 1]; // Shuffle elements down
4     elems[i] = pic;
5 }
```

Forcing Move Semantics

Solution: use move semantics

```
void PhotoCollection::insert(const Photo& pic, int pos) {
    for (int i = size(); i > pos; i--)
        elems[i] = std::move(elems[i - 1]);
    elems[i] = pic;
}
```

std::move casts an lvalue to an rvalue, allowing compiler to select correct overload

Be wary of `std::move`

If we move an lvalue, what happens to it afterwards?

```
Photo takePhoto();

void foo(Photo whoAmI)
   Photo selfie = std::move(whoAmI);
   whoAmI.get_pixel(21, 24); // ???
}
```

❌ If we move, `whoAmI` ends up in an unknown state!

Use `std::move` to implement move operations!

```
class Photo {
public:
  Photo::Photo(Photo&& other) {
    keywords = other.keywords;
  }

private:
  std::vector<string> keywords;
};
```

We know that `other` is temporary! So do we *really* need to make a copy of `other.keywords`?

Use `std::move` to implement move operations!

```
class Photo {
public:
  Photo::Photo(Photo&& other) {
    keywords = std::move(other.keywords);
  }

private:
  std::vector<string> keywords;
};
```

Solution: force move semantics by using `std::move`

`std::move` doesn't do anything special!

- `std::move` just type casts an lvalue to an rvalue

Return value

```
static_cast<typename std::remove_reference<T>::type&&>(t)
```

- Like `const_cast`, we "opt in" to potentially error-prone behaviour
  - What if we try to use an object after it's been moved! 🚨 🆘 🚨
- Try to avoid explicitly using `std::move` unless you have good reason!
  - E.g. performance really matters, you know for sure the object won't be used!

So many SMFs... 😩
Do I need to define them all!?


No! Rule of zero, three, and five!

Rule of Zero

- If a class doesn't manage memory (or another external resource), the compiler generated versions of the SMFs are sufficient!
- **Example:** Compiler generated SMFs of `Post` will call SMFs of `Photo` and `std::string`

```
struct Post {
    Photo photo;
    std::string user;
};
```

Rule of Three

- If a class manages external resources, we must define **copy assignment/constructor**
- If we don't, compiler-generated SMF won't copy underlying resource
  - This will lead to bugs, e.g. two `Photo`'s referring to the same underlying data

```
Rule of Three: If you need any one of these, you need them all:
    • Destructor
    • Copy Assignment
    • Copy Constructor
```

Rule of Five

- If we defined **copy constructor/assignment** and **destructor**, we should also define **move constructor/assignment**
- This is not required, but our code will be slower as it involves unnecessary copying

```
Rule of Five: If you need any of these, you probably want them all:
    • Destructor
    • Copy Assignment
    • Copy Constructor
    • Move Assignment    (Optional)
    • Move Constructor   (Optional)
```

What we covered

- SMFs Recap
  - SMFs manage the lifetime of a class instance.
- The Problem
  - The copy SMFs make a copy of every object, including temporary ones!
- `lvalues` and `rvalues`
  - An lvalue is a persistent object. An rvalue is a temporary one.
- Move Semantics
  - We can define “move” overloads of the copy constructor/assignment operator
- `std::move` and SMFs
  - `std::move` opts into move semantics, but be careful! Rule of zero/three/five