

Welcome back! Link to Attendance Form ↓





## Low Battery

10% battery remaining.

[Low Power Mode](#)

[Close](#)

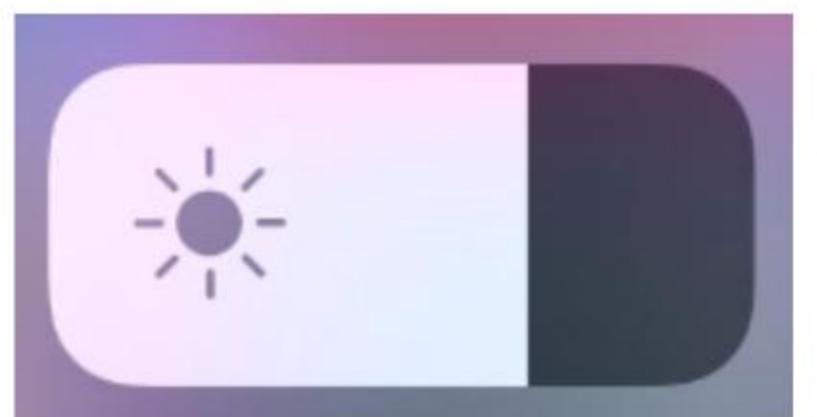
# **Things that drain battery life**

# Things that drain battery life

Connecting to WiFi



Powering the display

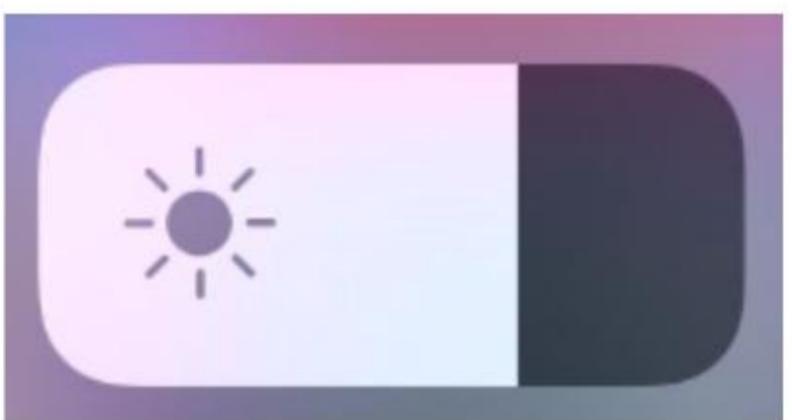


# Things that drain battery life

Connecting to WiFi



Powering the display



Performing computations

$$f(x) = a_0 + \sum_{n=1}^{\infty} \left( a_n \cos \frac{n\pi x}{L} + b_n \sin \frac{n\pi x}{L} \right)$$

Copying data?? 🤔

```
110101110101011101  
010101010111010101  
010100100010101010
```

# 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\*

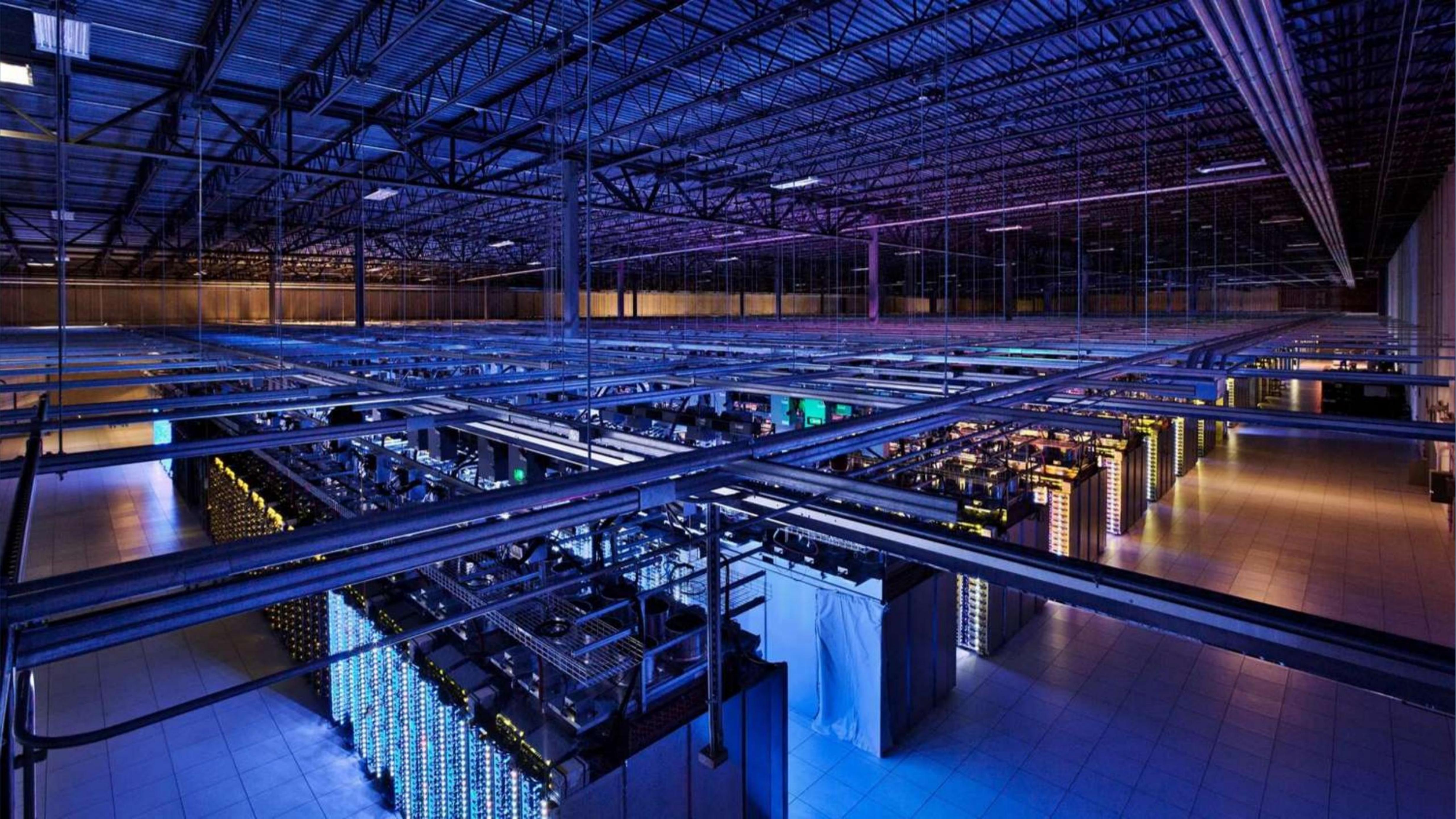
\* Pacific Northwest National Laboratory

{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

[\[source\]](#)

**Why does it matter?**



**How can we avoid needlessly copying data?**

# **Lecture 13:**

# **Move Semantics**

CS106L, Winter 2025

# **Today's Agenda**

# Today's Agenda

- SMFs Recap
  - What is a special member function?
- The Problem
  - How do our SMFs cause unnecessary copies?
- lvalues and rvalues
  - How does C++ distinguish between persistent and temporary objects?

# Today's Agenda

- SMFs Recap
  - What is a special member function?
- The Problem
  - How do our SMFs cause unnecessary copies?
- lvalues and rvalues
  - How does C++ distinguish between persistent and temporary objects?
- Move Semantics
  - How can we avoid making unnecessary copies? And a code demo!
- `std::move` and SMFs
  - How can we "opt-in" to move semantics? Which SMFs should I define?

# What questions do you have?



bjarne\_about\_to\_raise\_hand

# **SMFs Recap**

# Last Time

- Special member functions handle the class lifecycle

# Last Time

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

# Introducing... the Photo class

```
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)
```

# Photo SMF: Copy Constructor

```
Photo::Photo(const Photo& other)
: width(other.width)
, height(other.height)
```

Creates a **new photo** from an **existing one**, creating a copy of its data!

# Photo SMF: Copy Assignment

```
Photo &Photo::operator=(const Photo& other) {  
    // Implementation  
    return *this;  
}
```

# Photo SMF: Copy Assignment

```
Photo &Photo::operator=(const Photo& other) {  
  
    return *this;  
}
```

**Replaces a photo's  
contents with the  
contents of another,  
cleaning up its own  
data before copying  
the new one!**

# Photo SMF: Copy Assignment

```
Photo &Photo::operator=(const Photo& other) {  
  
    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;  
}
```

**Replaces a photo's contents with the contents of another, cleaning up its own data before copying the new one!**

# Photo SMF: Destructor

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

# What questions do you have?



bjarne\_about\_to\_raise\_hand

# Your Turn

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

```
Photo takePhoto();  
  
int main() {  
    Photo selfie = takePhoto();      // (A)  
    Photo retake(0, 0);  
    retake = takePhoto();           // (B)  
}
```

# Your Turn

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

```
Photo takePhoto();  
  
int main() {  
    Photo selfie = takePhoto(); // (A) Copy  
    Photo retake(0, 0);  
    retake = takePhoto(); // (B)  
}
```

# Your Turn

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

```
Photo takePhoto();
```

```
int main() {
```

```
    Photo selfie = takePhoto();
```

// (A)

**Copy Destruct**

```
    Photo retake(0, 0);
```

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

// (B)

**Assign**

```
}
```

# A Small Aside: Return Value Optimization

- This line

```
Photo selfie = takePhoto();
```

might not actually call **copy-constructor + destructor**

# A Small Aside: Return Value Optimization

- This line

```
Photo selfie = takePhoto();
```

might not actually call **copy-constructor + destructor**

- This is due to a compiler optimization called return-value optimization (RVO)
- For the purposes of this lecture, we will pretend that it does!

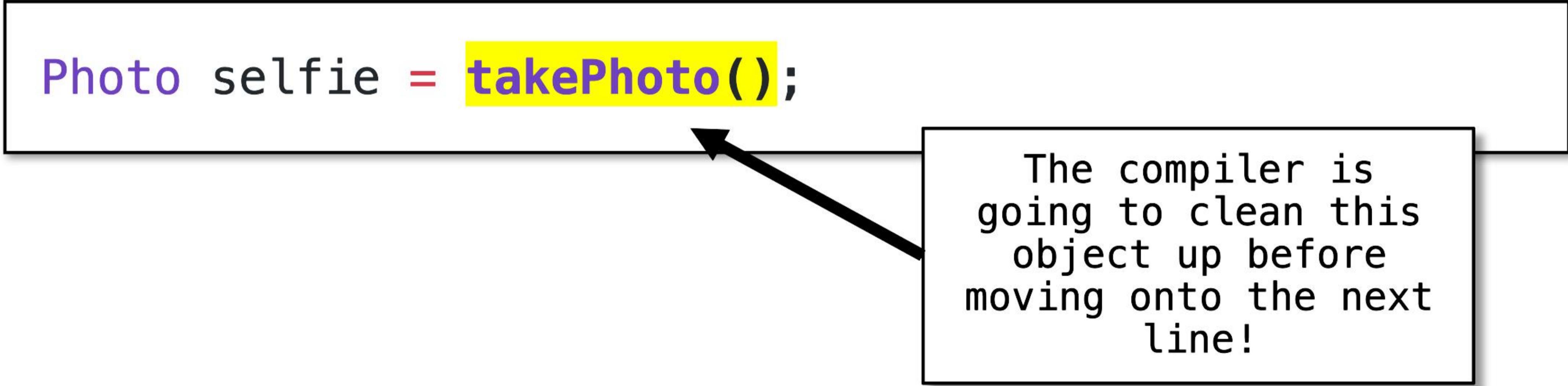
**Key Idea:** The **return value** of a function is **temporary**  
(it's destroyed before the next line)

**Key Idea:** The **return value** of a function is **temporary** (it's destroyed before the next line)

```
Photo selfie = takePhoto();
```

**Key Idea:** The **return value** of a function is **temporary** (it's destroyed before the next line)

```
Photo selfie = takePhoto();
```



The compiler is going to clean this object up before moving onto the next line!

# What questions do you have?



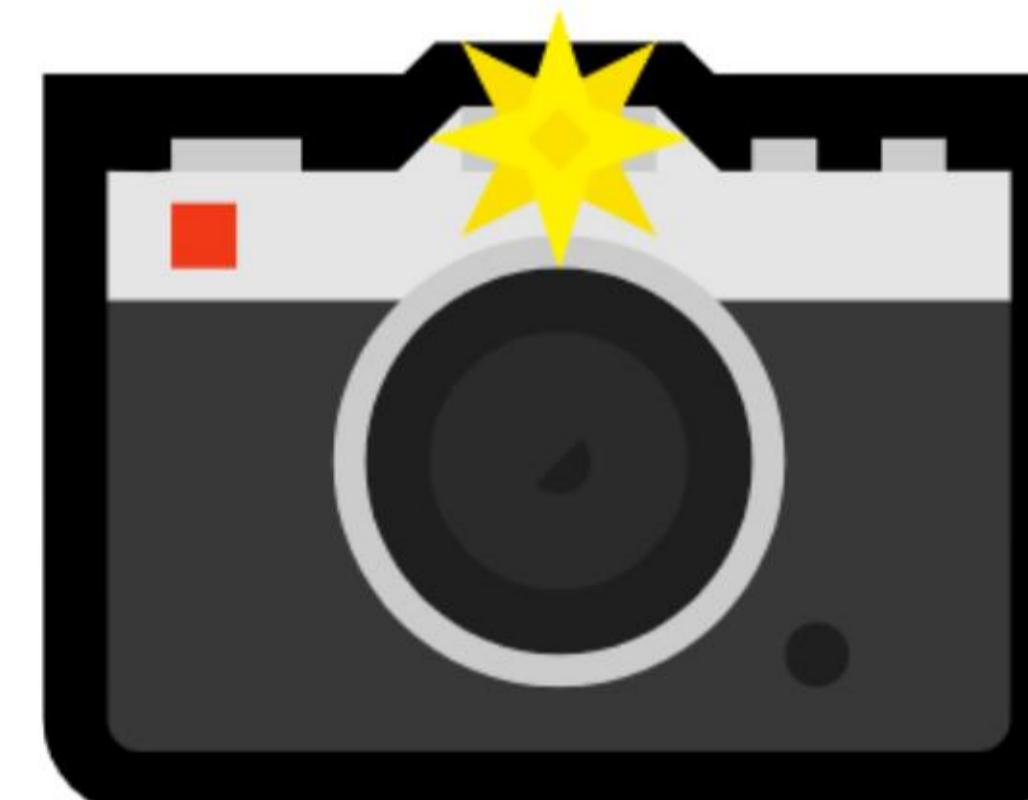
bjarne\_about\_to\_raise\_hand

# The Problem

# The Problem

```
Photo selfie = takePhoto();
```

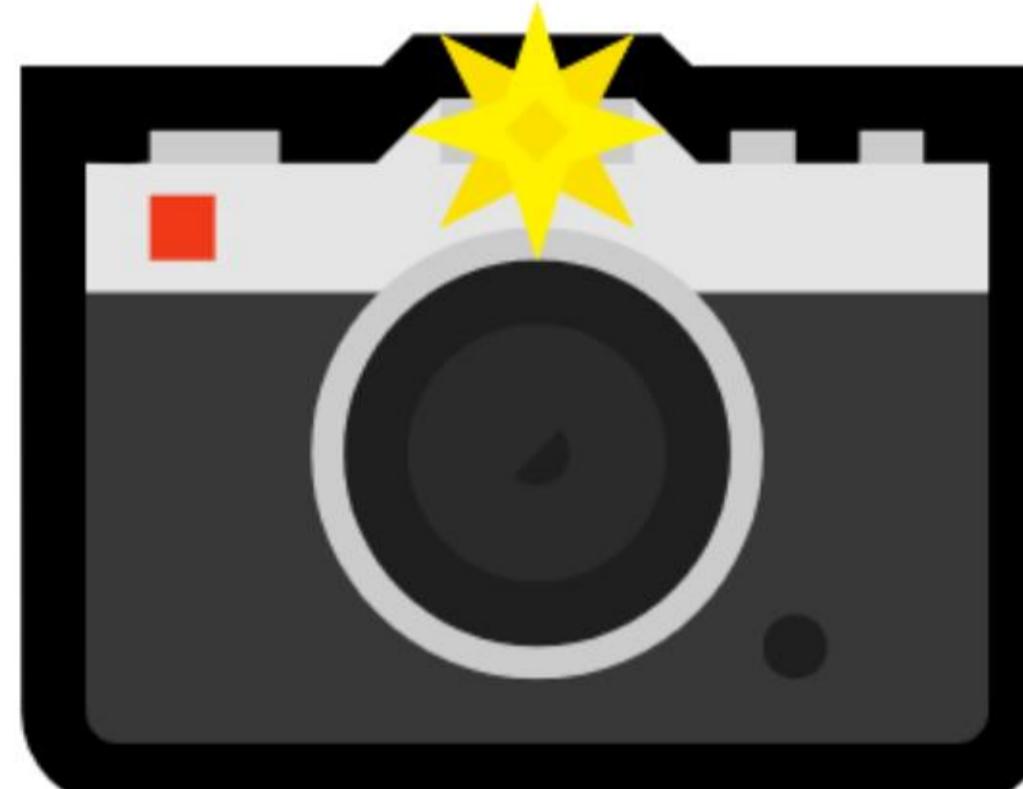
# The Problem



takePhoto()

Photo selfie = takePhoto();

# The Problem



`takePhoto()`



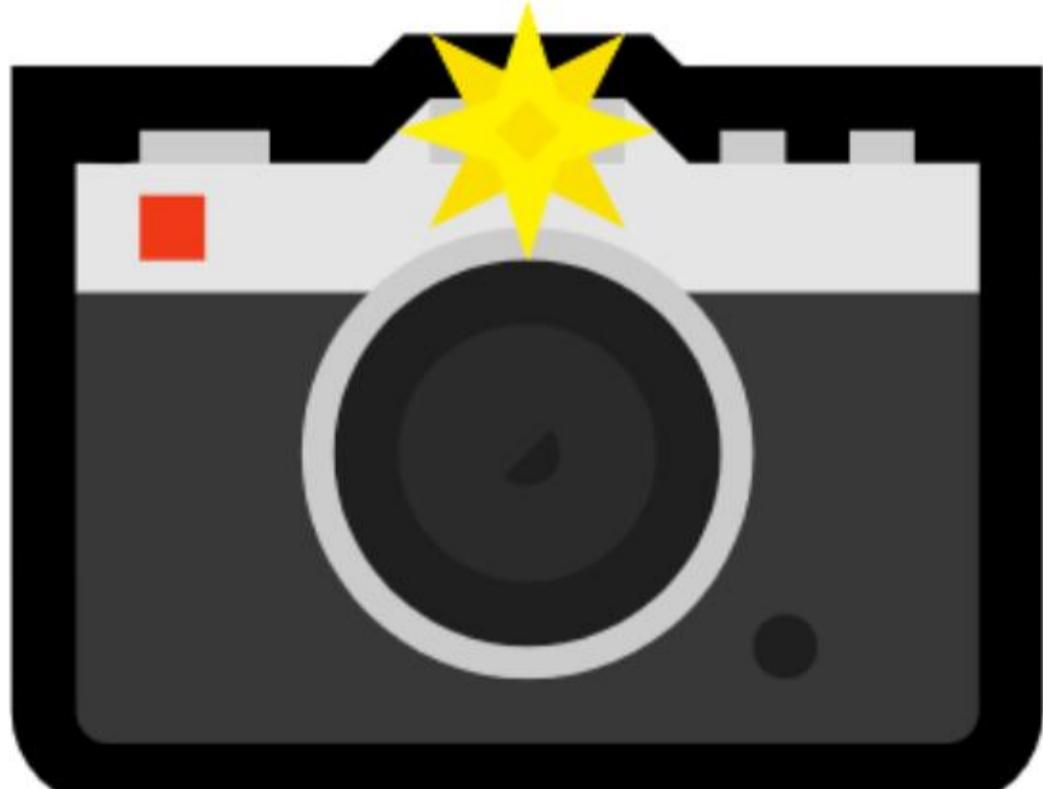
## Photo

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



`Photo selfie = takePhoto();`

# The Problem



takePhoto()

**Photo**

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



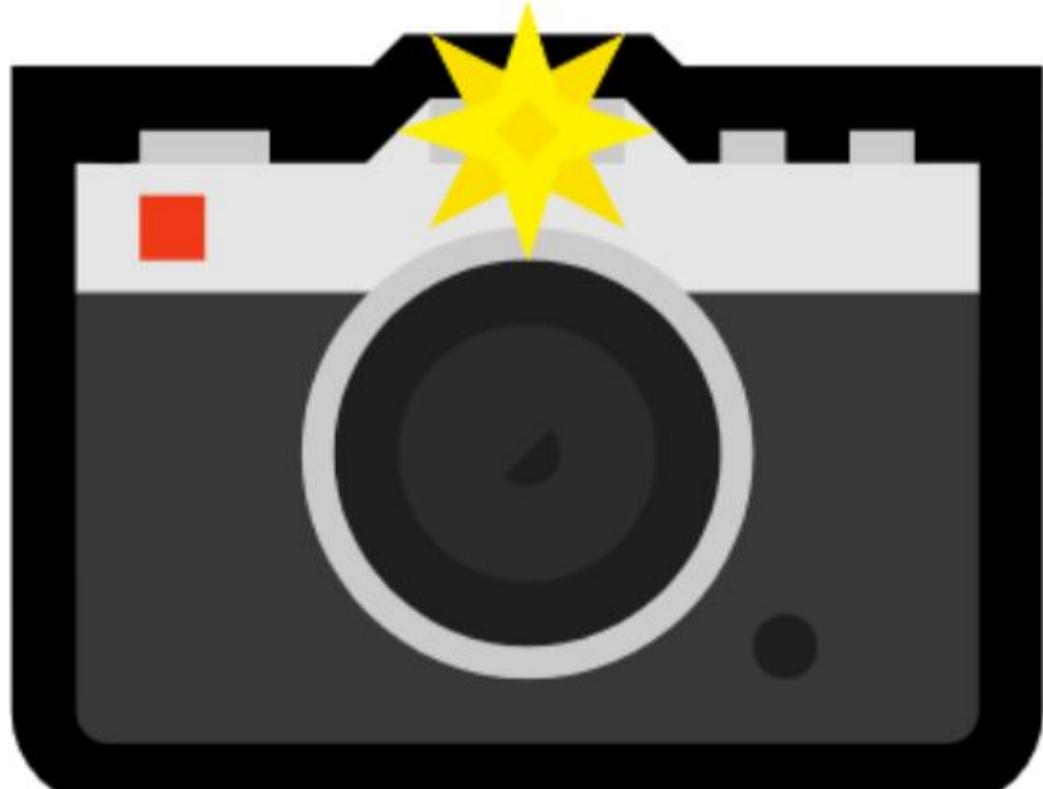
**Photo (selfie)**

- width = 3840
- height = 2160
- data = 0x133210f1



**Photo** selfie = takePhoto(); // Copy constructor

# The Problem



`takePhoto()`

**Photo**  
• width = 3840  
• height = 2160  
• data = 0x1024c3bd

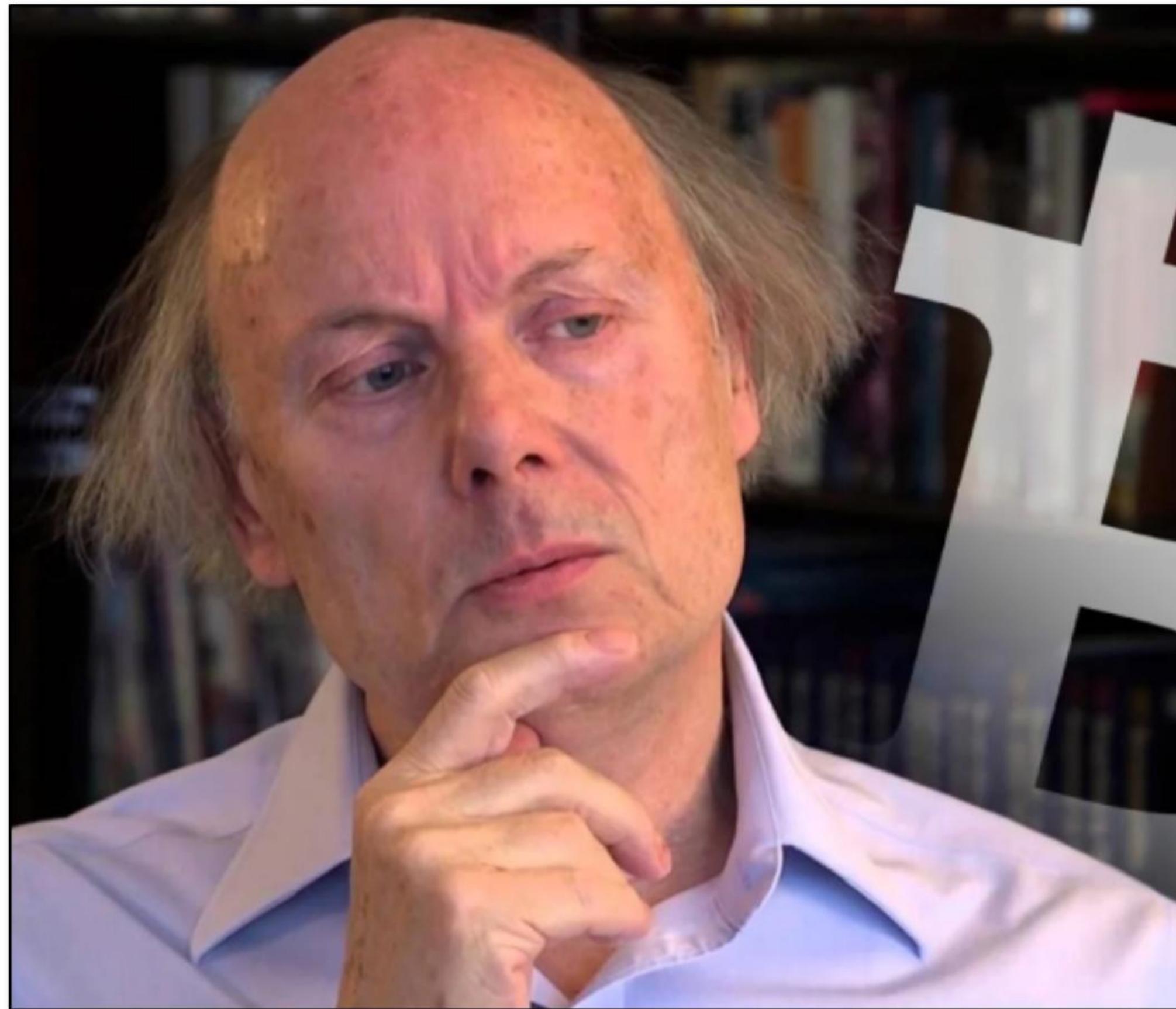


**Photo (selfie)**  
• width = 3840  
• height = 2160  
• data = 0x133210f1



```
Photo selfie = takePhoto(); // Destructor
```

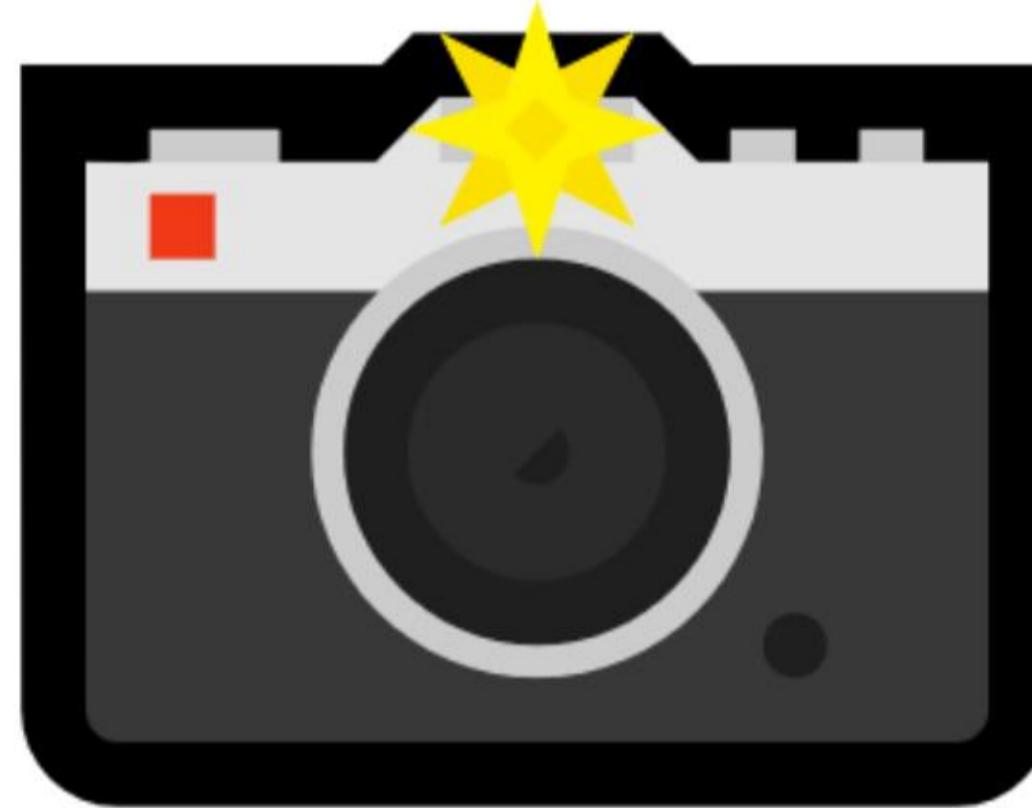
# The Problem



concerned\_bjarne

**What if we could reuse the memory instead?**

# The Solution: Move Semantics



`takePhoto()`



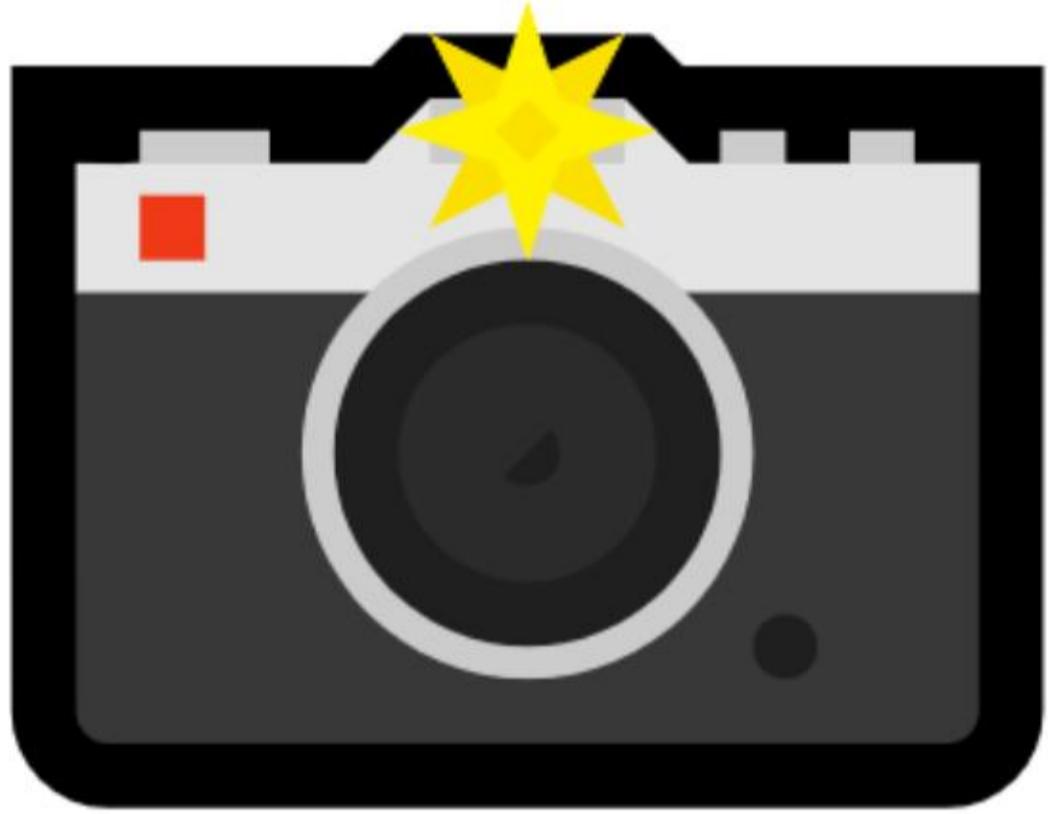
## Photo

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



`Photo selfie = takePhoto();`

# The Problem



takePhoto()

**Photo**

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

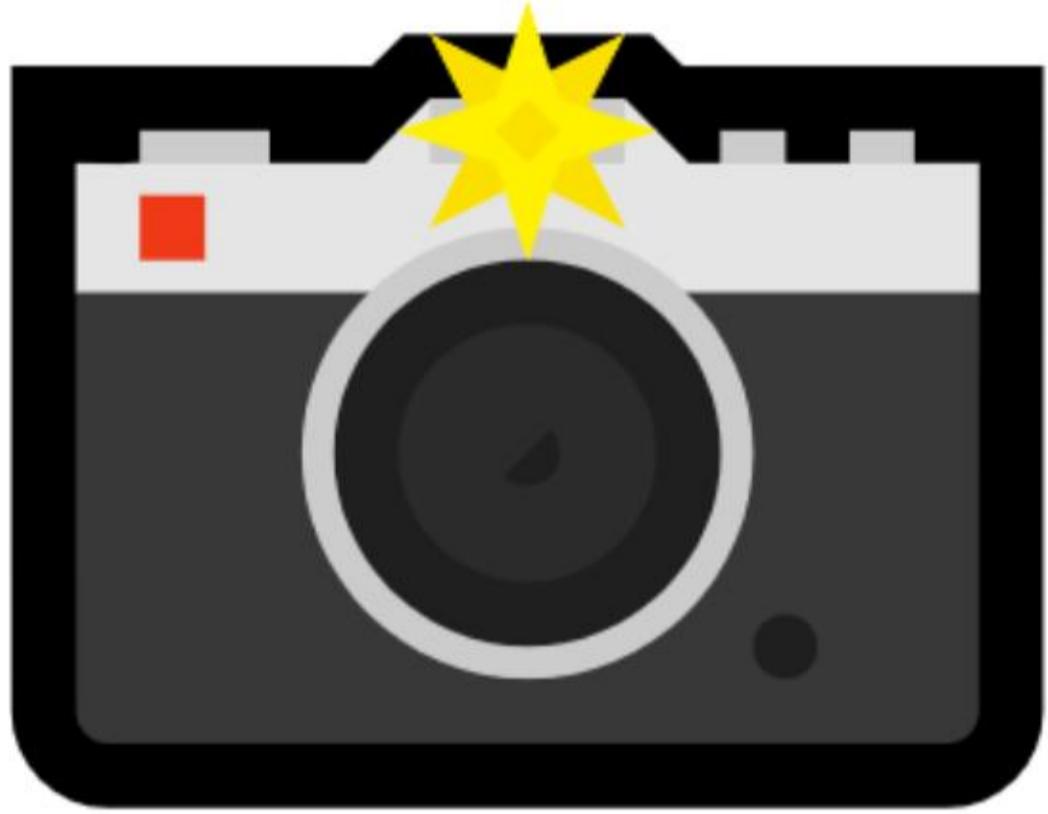


**Photo (selfie)**

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

Photo selfie = takePhoto(); // ~~Copy~~ Move constructor

# The Problem



`takePhoto()`

**Photo**

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

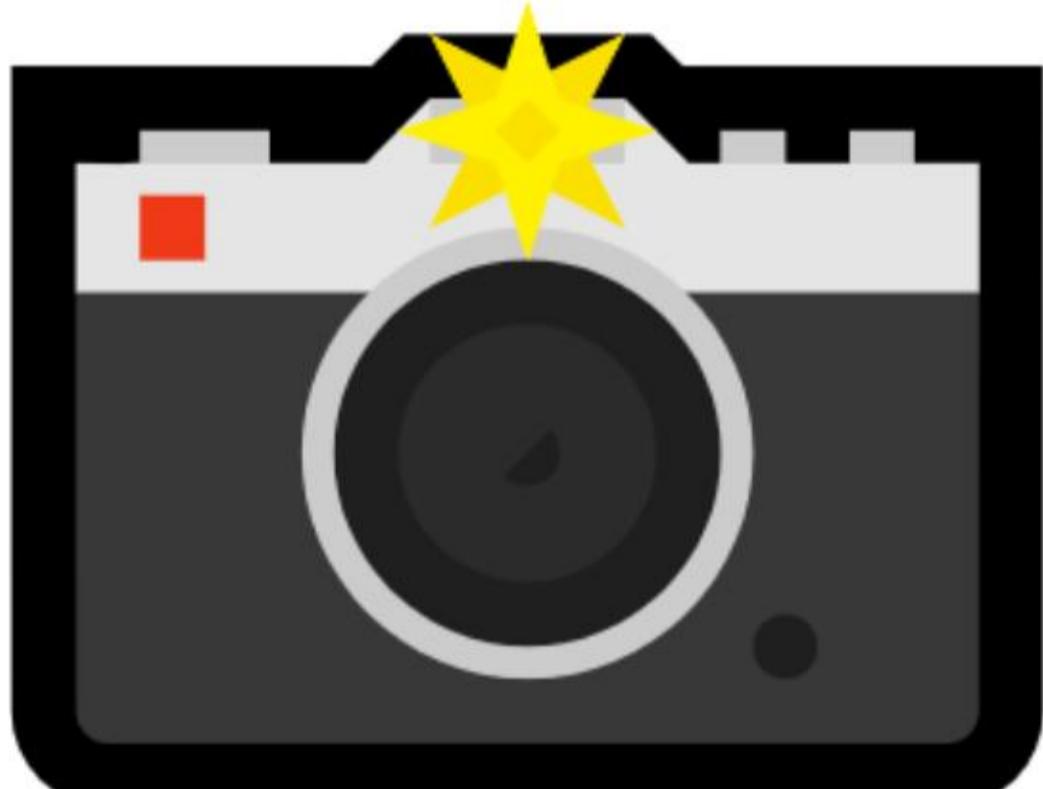


**Photo (selfie)**

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

```
Photo selfie = takePhoto(); // Destructor
```

# The Problem



takePhoto()

☠ **Photo**

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

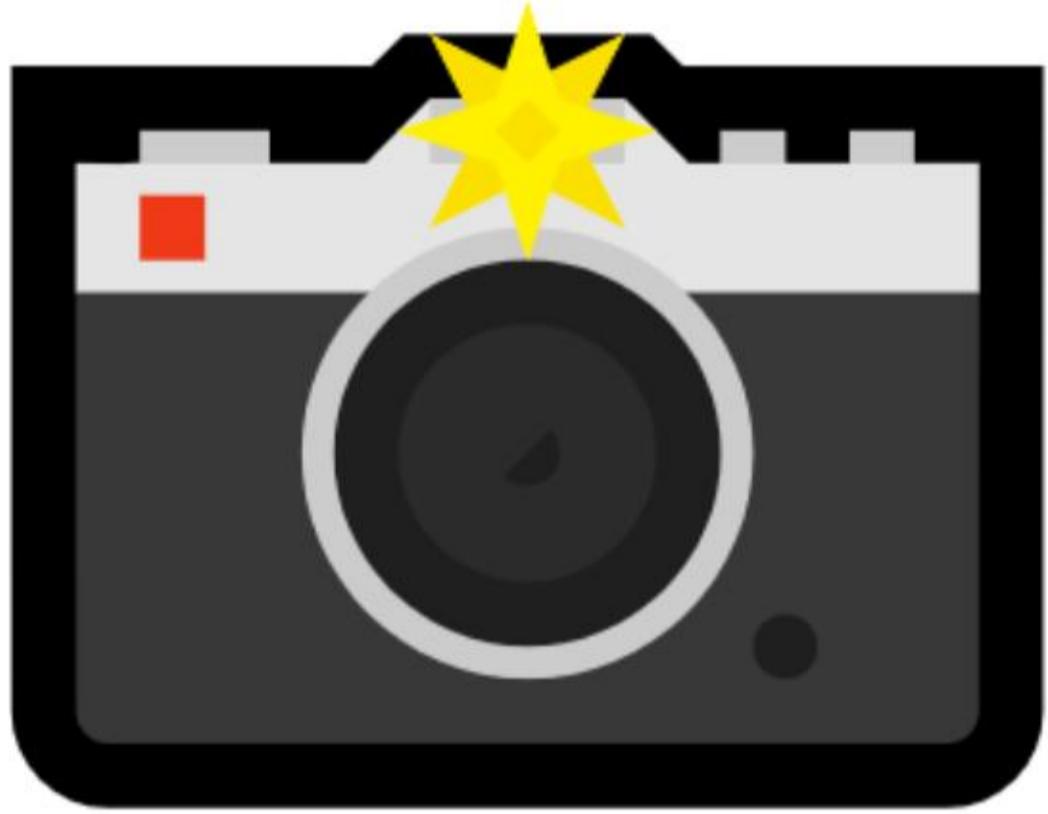


**Photo (selfie)**

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

```
Photo selfie = takePhoto(); // Destructor
```

# The Problem



takePhoto()

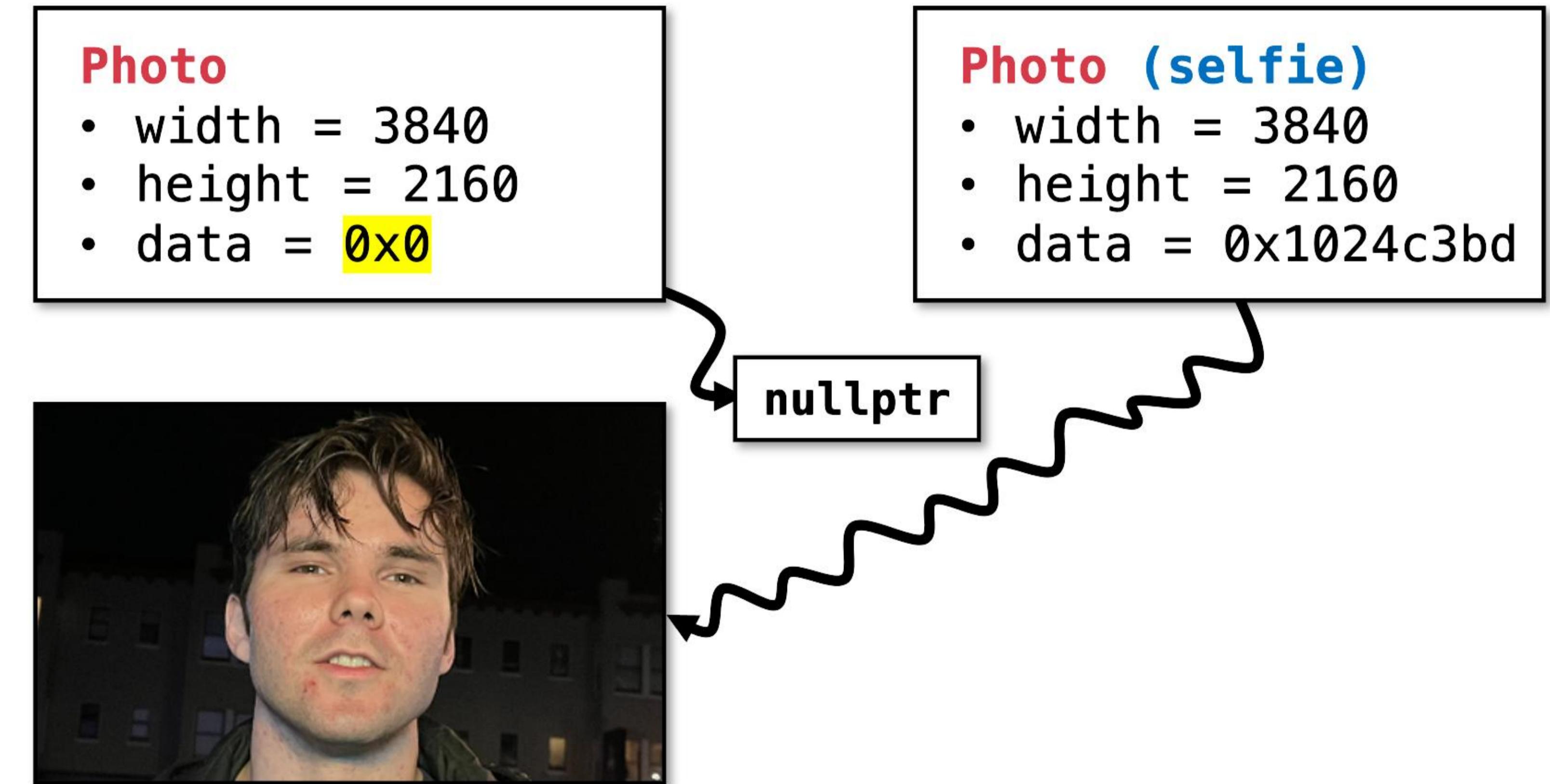
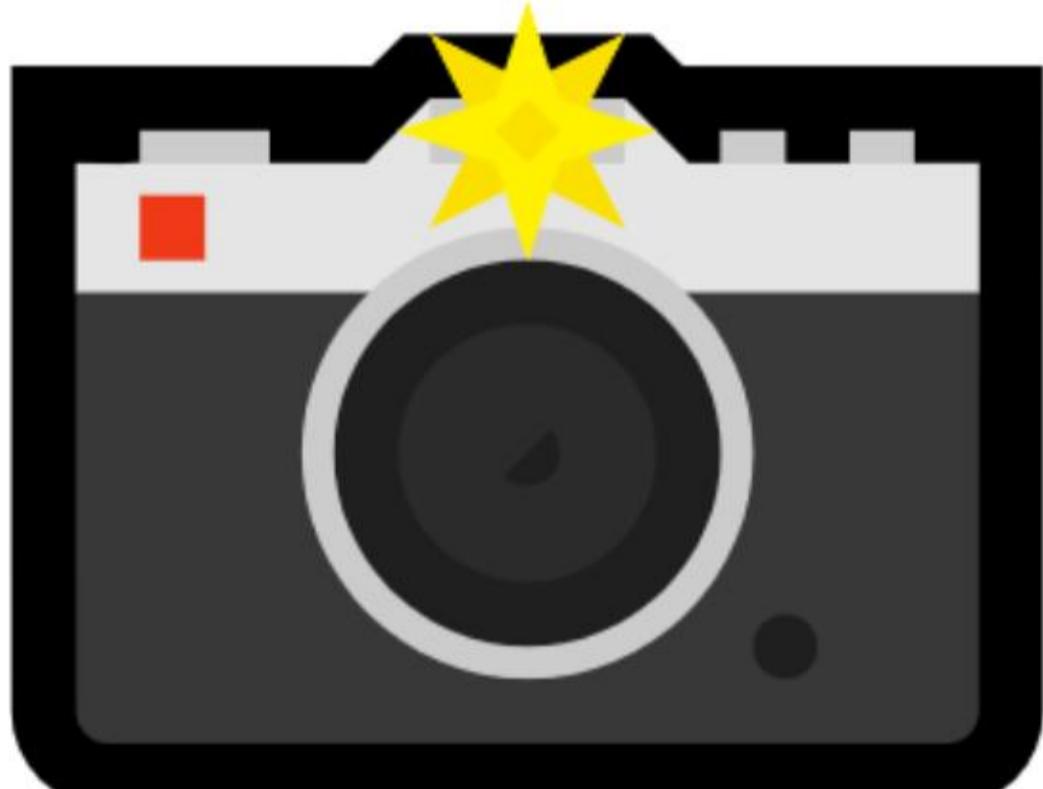
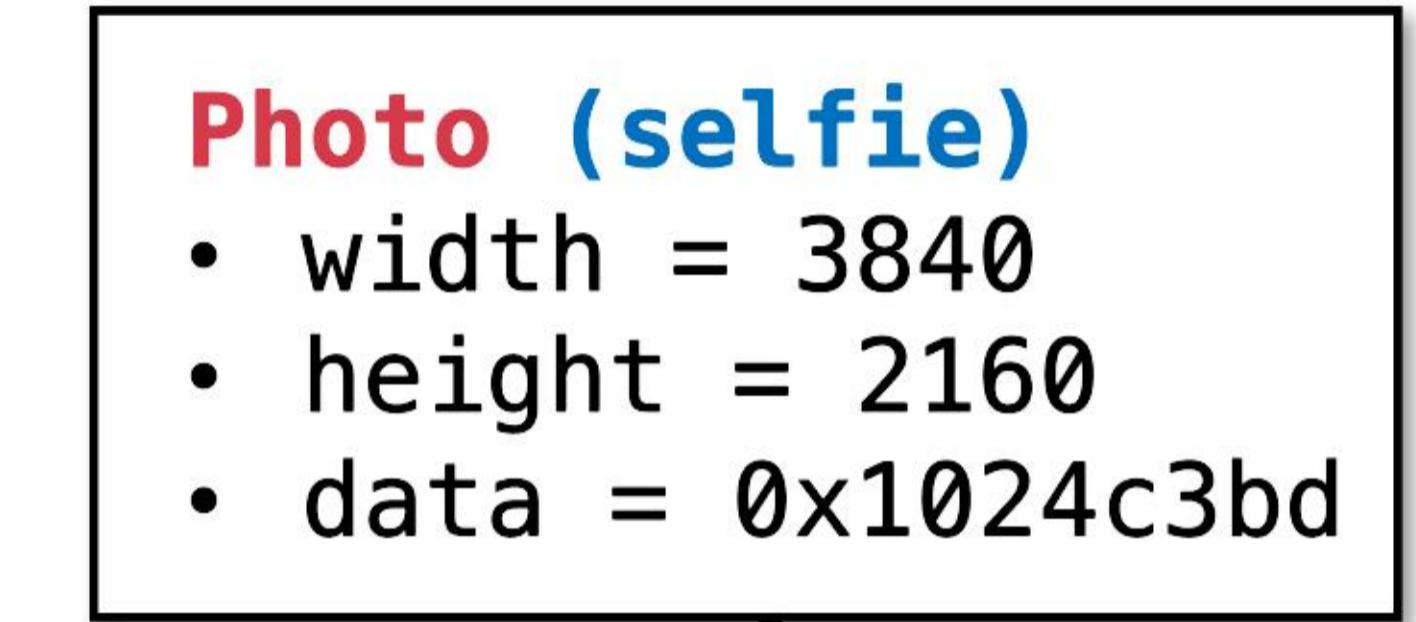
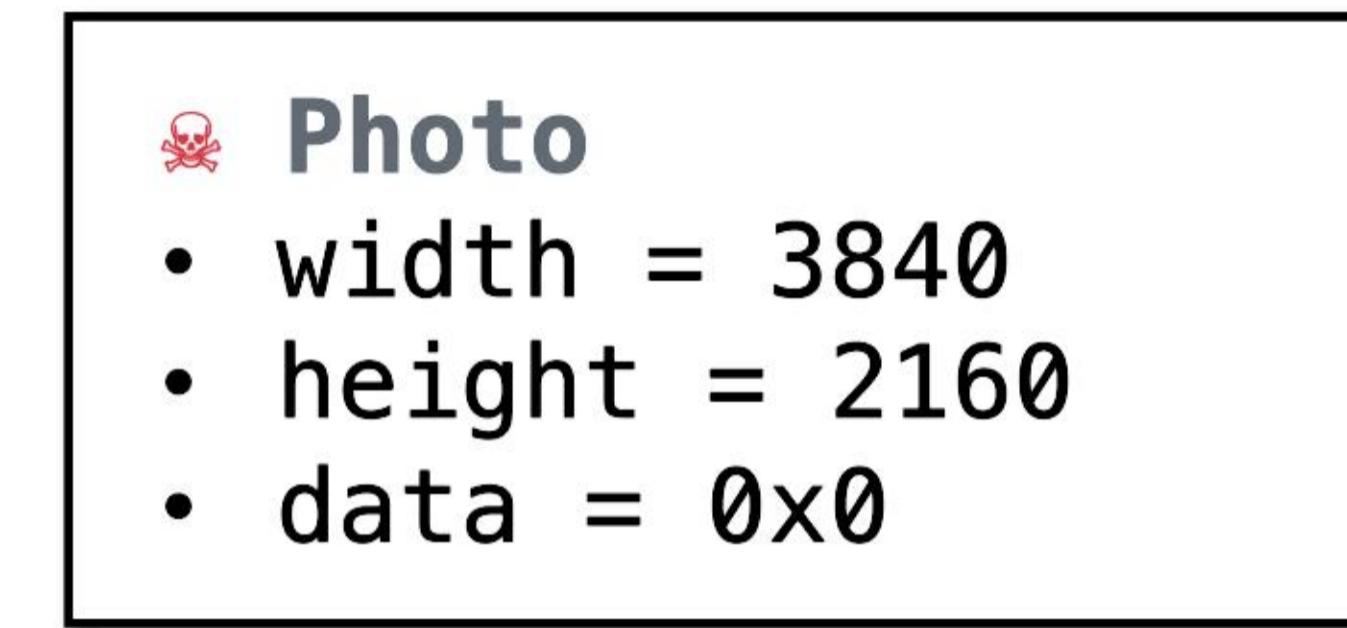


Photo selfie = takePhoto(); // ~~Copy~~ Move constructor

# The Problem



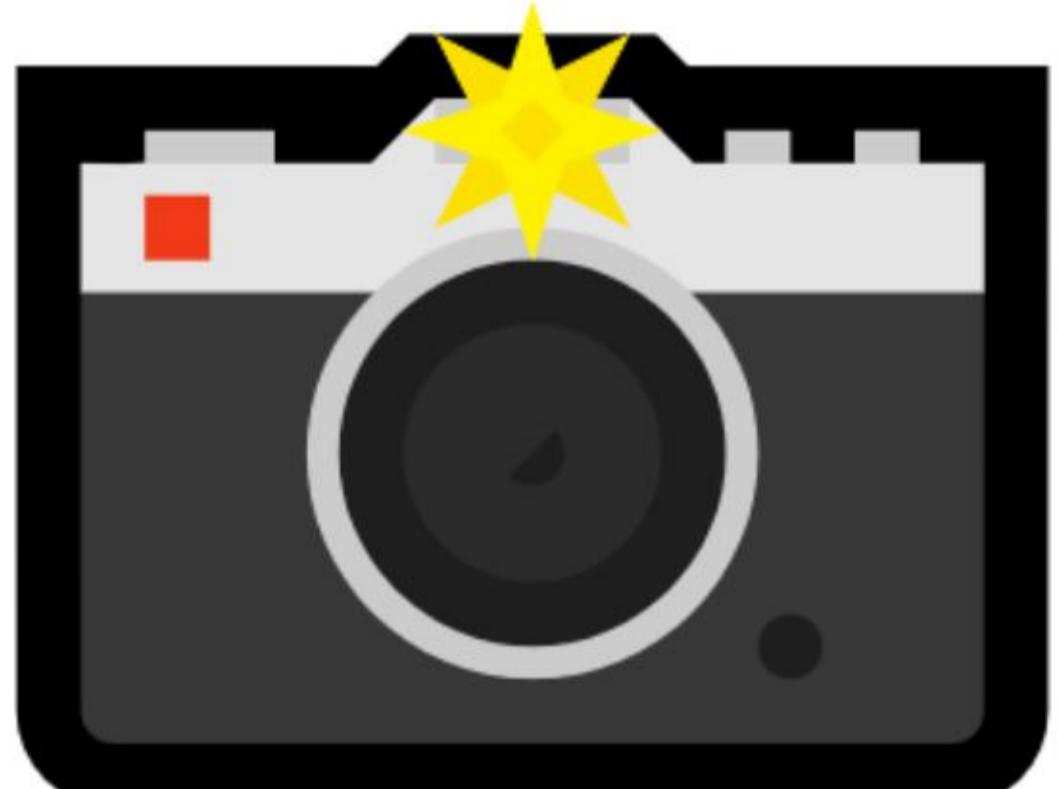
takePhoto()



nullptr

```
Photo selfie = takePhoto(); // Destructor
```

# The Problem



`takePhoto()`

## **Photo**

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

`nullptr`

## **Photo (selfie)**

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



```
Photo selfie = takePhoto(); // Destructor
```

We created a new Photo without any copying! 💪

**But... is it always safe to do this?**

# Move vs. Copy Semantics

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

# Move vs. Copy Semantics

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

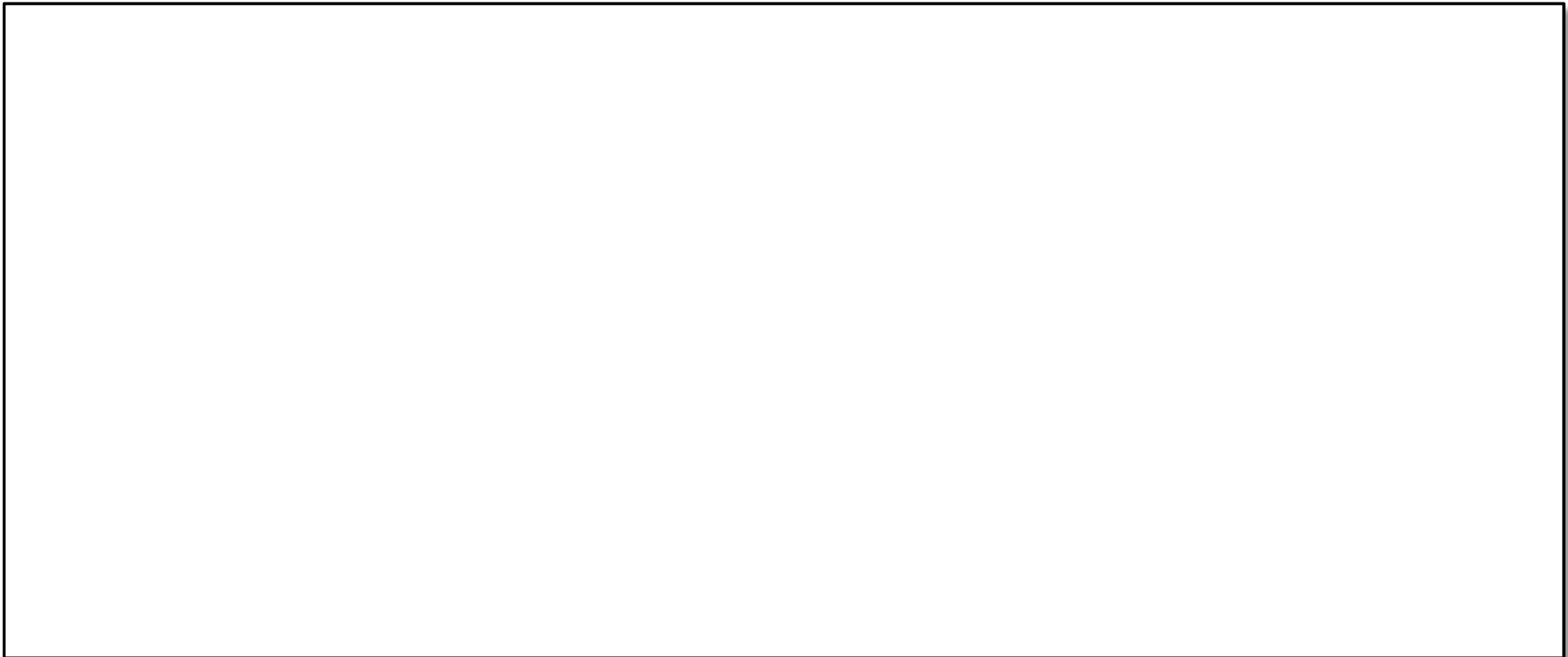
```
Photo takePhoto();  
  
void foo(Photo whoAmI) {  
    Photo selfie = whoAmI;           // What if we move here?  
    whoAmI.get_pixel(21, 24);        // ???  
}
```

# 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



# Move vs. Copy Semantics

```
Photo selfie = pic;  
// make copies of persistent objects (e.g. variables)  
// that might get used in the future
```

```
Photo selfie = takePhoto();  
// move temporary objects (e.g return values)  
// since we no longer need to use them
```

# What questions do you have?



bjarne\_about\_to\_raise\_hand

# Move vs. Copy Semantics

```
Photo selfie = pic;  
// make copies of persistent objects (e.g. variables)  
// that might get used in the future
```

```
Photo selfie = takePhoto();  
// move temporary objects (e.g return values)  
// since we no longer need to use them
```

# **lvalues & rvalues**

# lvalues & rvalues

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

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

# 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

Generally speaking, **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 **Ivalue** 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?

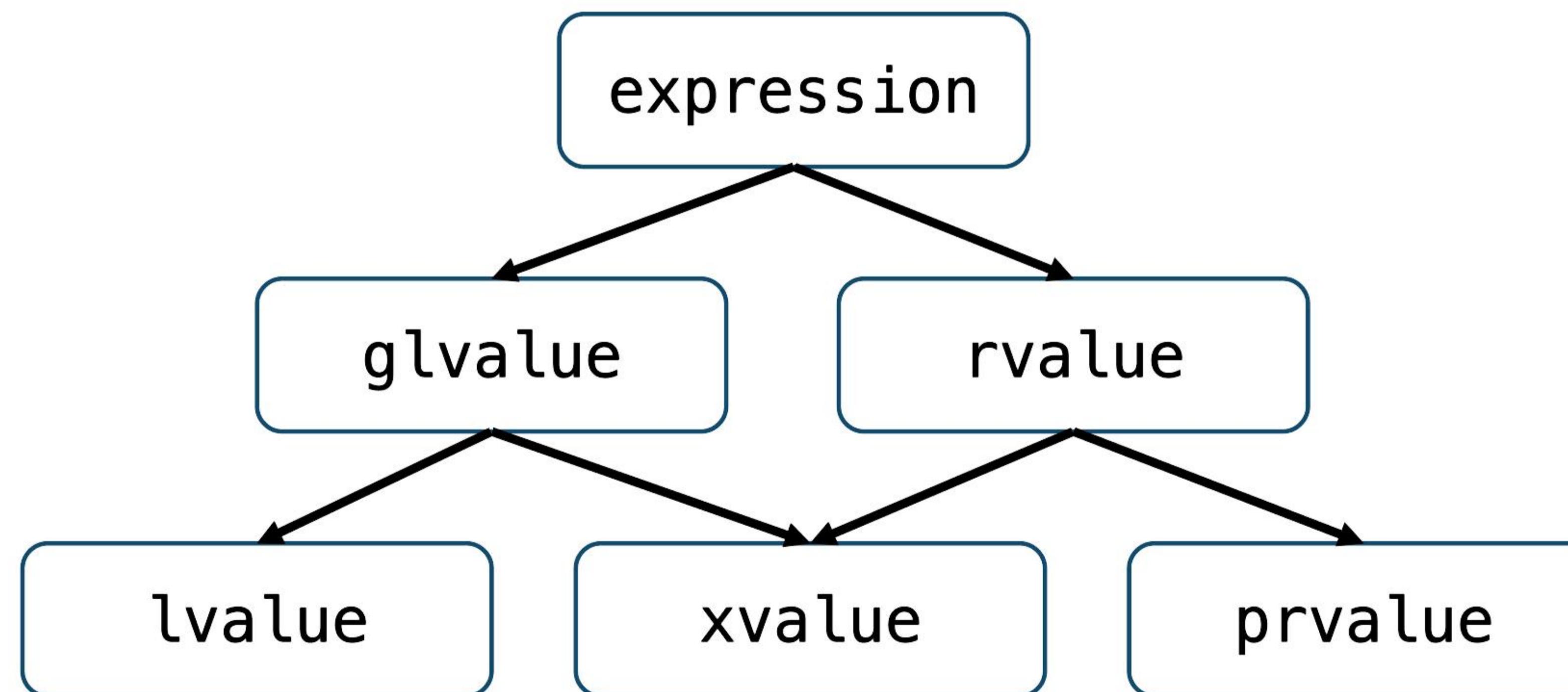
- Hint: which ones have a definite address?

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

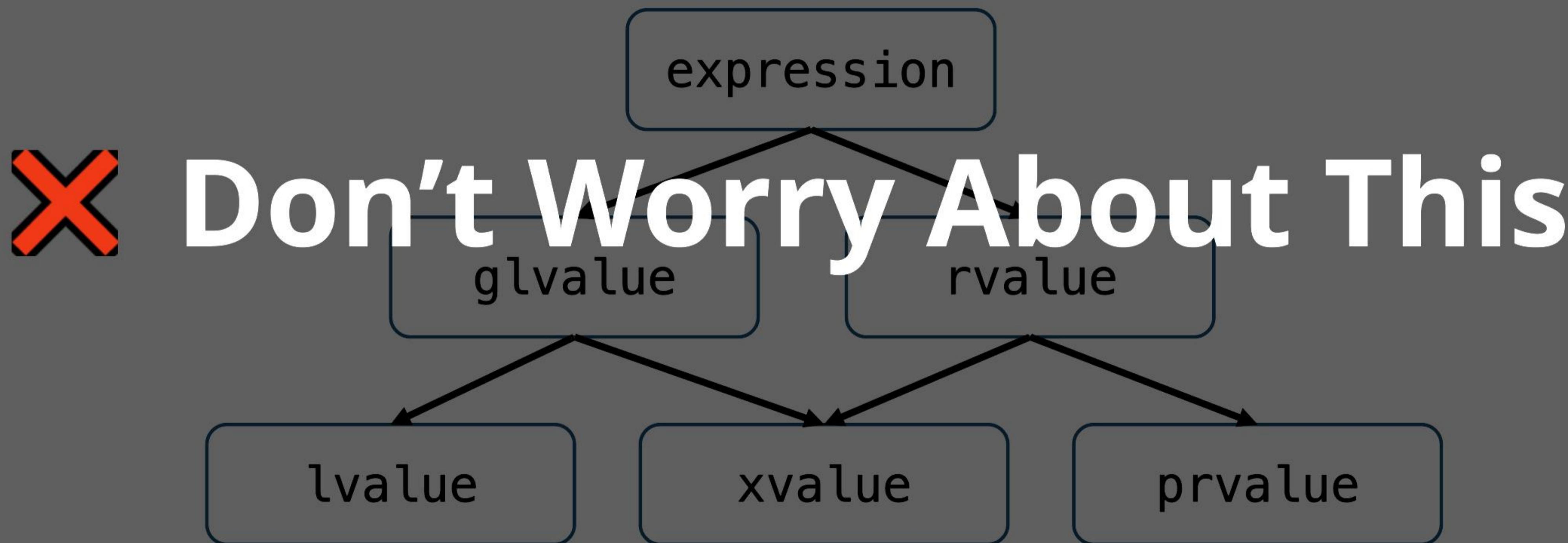
An **I**value's lifetime is until the end of scope

An **Ivalue** is persistent

# Quick Note: It's more complicated than this!



**Quick Note: It's more complicated than this!**



# Working towards move semantics

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

# 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);

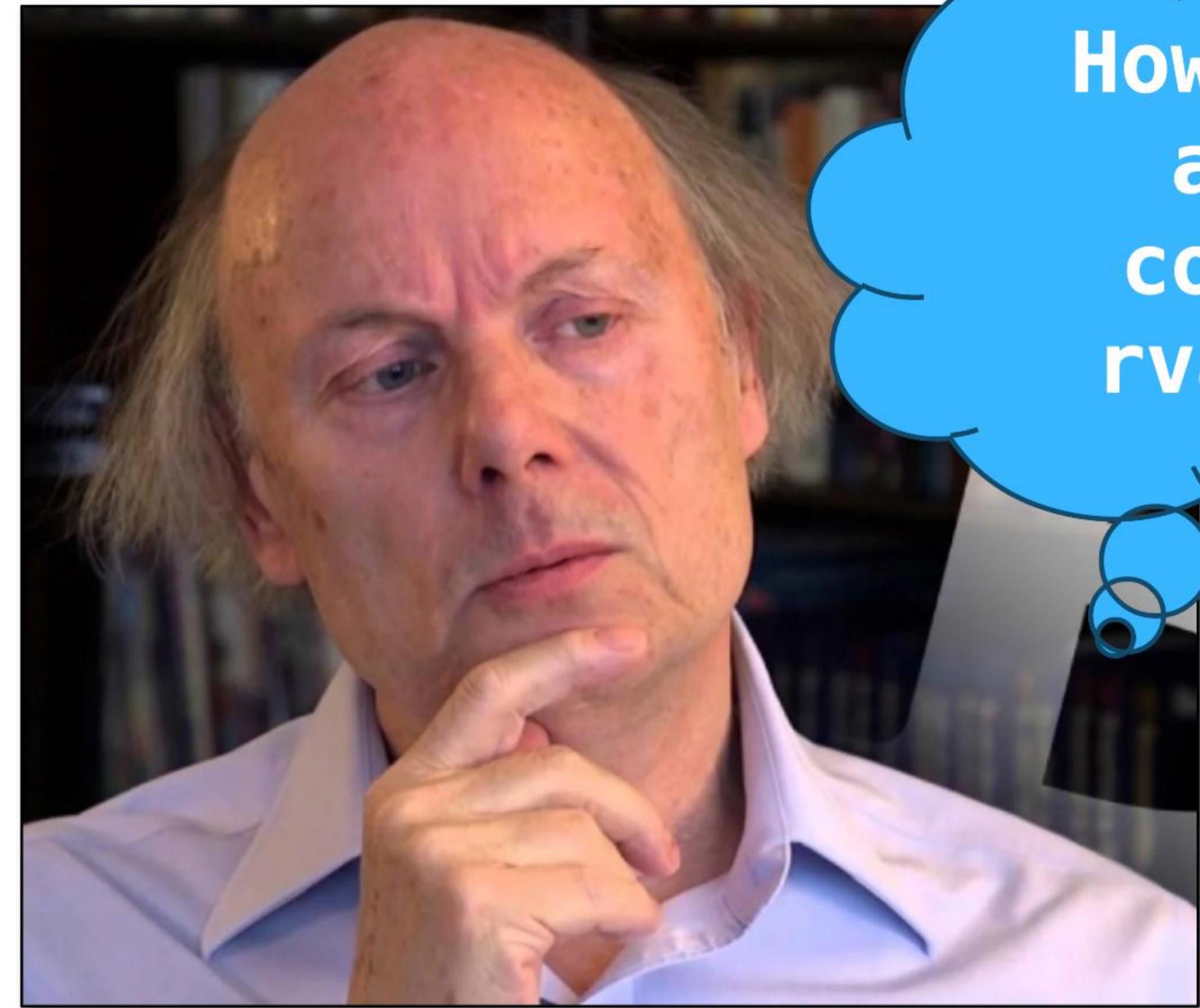
int main() {
    Photo selfie = takePhoto(); // selfie is lvalue
    uploadToInsta(selfie); // ✅ No copy is made here
}
```

# 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?
}
```



How do we  
avoid  
copying  
rvalues?

thinking\_bjarne

# lvalue reference

```
void upload(Photo& pic);

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

# Ivalue reference

```
void upload(Photo& pic);

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

# rvalue reference

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

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

# **A few important points**

# 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

```
//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);
```

[\[source\]](#)

```
//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);
```

```
//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 questions do you have?



bjarne\_about\_to\_raise\_hand

# **Move Semantics**

# What we want!

```
Photo selfie = pic;  
// copy persistent objects (e.g. variables)
```

```
Photo selfie = 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
    );
}
```

# 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!
}
```

# 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;
}
```

# Let's overload the special member functions!

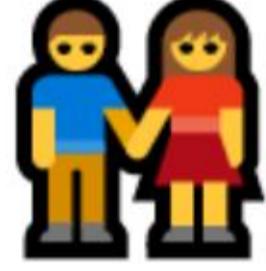
## Copy assignment operator

```
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!  
}
```

**Let's code this up!**

**Let's code this together** 

**1061.vercel.app/movesem**

# Two new special member functions!

- Move constructor
  - `Type::(Type&& other)`
- Move assignment operator
  - `Type& Type::operator=(Type&& other)`

# What questions do you have?



bjarne\_about\_to\_raise\_hand

# **std::move and SMFs**

# Forcing Move Semantics

# 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

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

# 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); // ???
}
```

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

```
class Photo {  
public:  
    Photo::Photo(Photo&& other) {  
        keywords = other.keywords;  
    }  
  
private:  
    std::vector<string> keywords;  
};
```

# 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;  
};
```

# **std::move doesn't do anything special!**

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

# **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)
```

# **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! 🚨 SOS 🚨
  - Try to avoid explicitly using **std::move** unless you have good reason!

# What questions do you have?



bjarne\_about\_to\_raise\_hand

# We have two new SMFs!

- Type::Type(const Type& other);
- Type& Type::operator=(const Type& other);
- Type::Type(Type&& other);
- Type& Type::operator=(Type&& other);
- ~Type::Type();

**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!

# 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 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 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 questions do you have?



bjarne\_about\_to\_raise\_hand

End of slide show, click to exit.