Welcome back! Link to Attendance Form \





Things that drain battery life

Connecting to WiFi



Performing computations

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

Powering the display



Copying data?? 👺



1101011101010111101 .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 of the total device energy). Our results also indicate a relatively high stalled cycle

Copying data is expensive

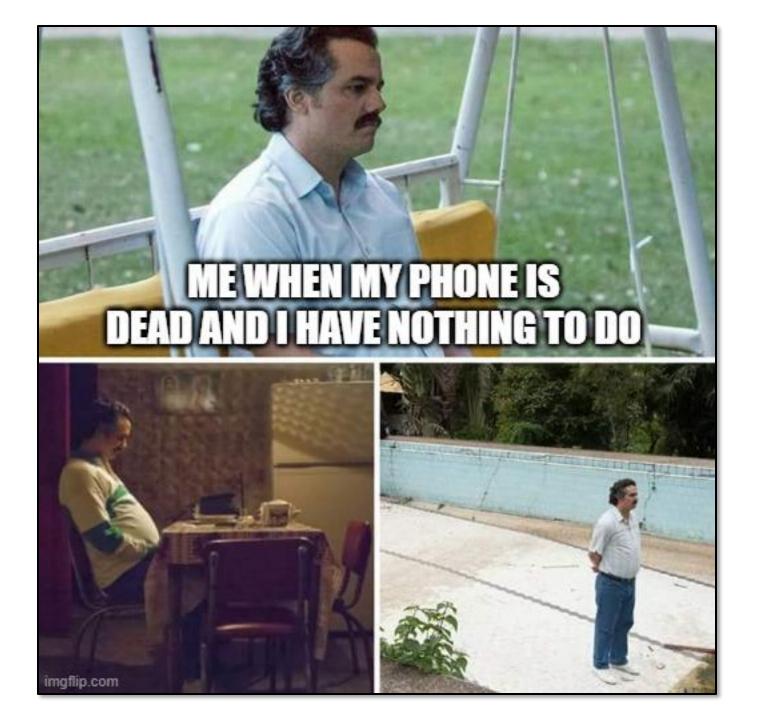
Quantifying the Energy Cost of Data Movement in Scientific Applications

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



Why does it matter?





Lecture 15: Move Semantics

CS106L, Spring 2025

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?



bjarne_about_to_raise_hand

SMFs Recap

Last Time

Special member functions handle the class lifecycle

- 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])
{}
```

Creates a **brand new photo** and allocates
memory for its pixels!
Photo photo(500, 500);

Photo SMF: Copy Constructor

```
Creates a new photo
                                            from an existing
Photo::Photo(const Photo& other)
                                          one, creating a copy
  : width(other.width)
                                              of its data!
                                            Photo p = photo;
    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& ot
  // Check for self assignment
                                                 Replaces a photo's
   if (this == &other) return *this;
                                                  contents with the
                                                 contents of another,
   delete[] data; // Clean up old pixels!
                                                 cleaning up its own
                                                 data before copying
  // Copy over new pixels!
                                                     the new one!
  width = other.width;
                                                      p = photo;
   height = other.height;
   data = new int[width * height];
   std::copy(other.data, other.data + width * height, data);
   return *this;
```

```
E.g. if we did p = p;
```

Photo SMF: Destructor

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

Cleans up this photo's data so we don't leak memory!



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)
                                           Copy Destruct
  Photo retake(0, 0);
  retake = takePhoto();
                                           Assign Destruct
                               // (B)
```

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 <u>return-value optimization (RVO)</u>
- 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)

Photo selfie = takePhoto();
The compiler is
going to clean this
object up before
moving onto the next
line!



bjarne_about_to_raise_hand



Photo selfie = takePhoto();

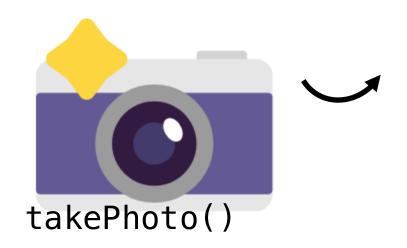




- width = 3840
- height = 2160
- data = $0 \times 1024 c3bd$



Photo selfie = takePhoto();



Photo

- width = 3840
- height = 2160
- data = $0 \times 1024 c3bd$

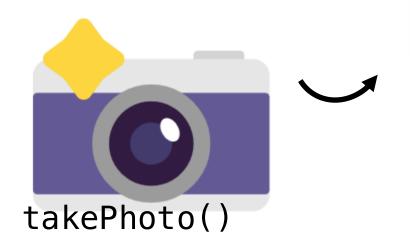


Photo (selfie)

- width = 3840
- height = 2160
- data = $0 \times 133210f1$



Photo selfie = takePhoto(); // Copy constructor



Photo

- width = 3840
- height = 2160
- data = $0 \times 1024 c3bd$

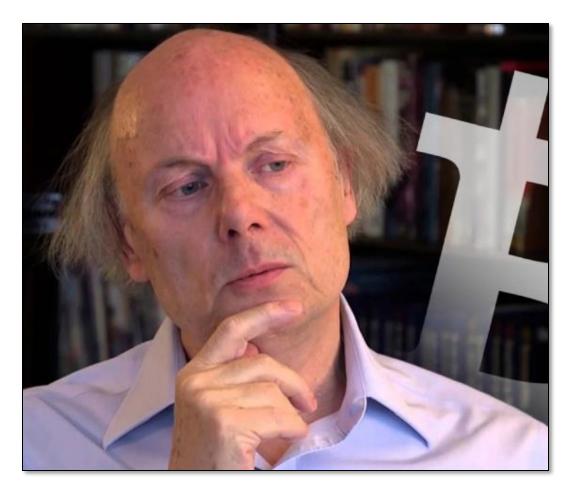


Photo (selfie)

- width = 3840
- height = 2160
- data = $0 \times 133210f1$



Photo selfie = takePhoto(); // Destructor



concerned_bjarne

What if we could reuse the memory instead?

The Solution: Move Semantics

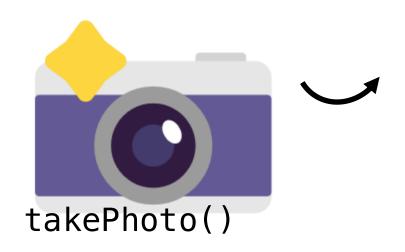




- width = 3840
- height = 2160
- data = $0 \times 1024 c3bd$



Photo selfie = takePhoto();



Photo

- width = 3840
- height = 2160
- data = $0 \times 1024 c3bd$

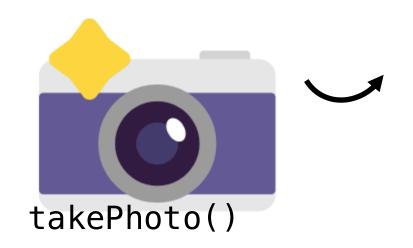


- width = 3840
- height = 2160
- data = $0 \times 1024 \times 3bd$



Instead of copying
data, let's steal
it!

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



Photo

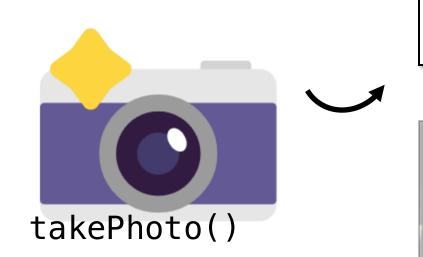
- width = 3840
- height = 2160
- data = $0 \times 1024 c3bd$

Photo (selfie)

- width = 3840
- height = 2160
- data = $0 \times 1024 c3bd$



Photo selfie = takePhoto(); // Destructor



Photo

- width = 3840
- height = 2160

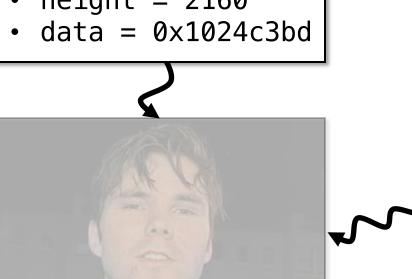
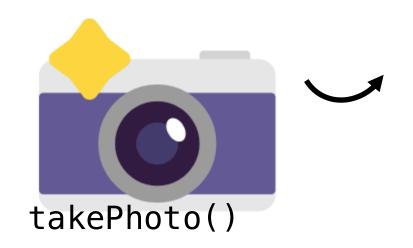


Photo (selfie)

- width = 3840
- height = 2160
- data = $0 \times 1024 c3bd$

Oh no... the destructor of takePhoto() deletes our stolen data

Photo selfie = takePhoto(); // Destructor



Photo

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



Photo (selfie)

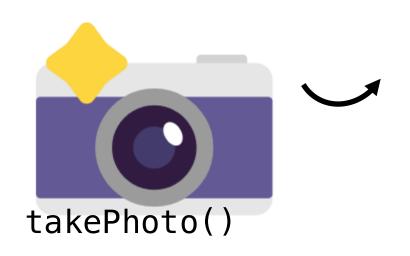
- width = 3840
- height = 2160

nullptr

• data = $0 \times 1024 c3bd$

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

The Problem



Photo

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



Photo (selfie)

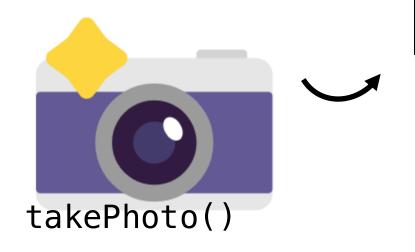
- width = 3840
- height = 2160
- data = $0 \times 1024 c3bd$

Photo destructor calls delete on nullptr, which does nothing!

Photo selfie = takePhoto(); // Destructor

nullptr

The Problem



Photo

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

nullptr

Photo (selfie)

- width = 3840
- height = 2160
- data = $0 \times 1024 c3bd$



We created a new Photo without any copying! 🌭



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

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

```
Photo takePhoto();
int main() {
  Photo selfie = takePhoto(); // Move takePhoto()
                              // since it's temporary!
```

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?
```

X 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); // X use-after-move
```

Copy Semantics Move Semantics

Building a new computer

Copy semantics

Move semantics

"I still want to use my old computer"







```
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
```



bjarne_about_to_raise_hand

```
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
How does the compiler know whether to move or copy?
```

Ivalues & rvalues

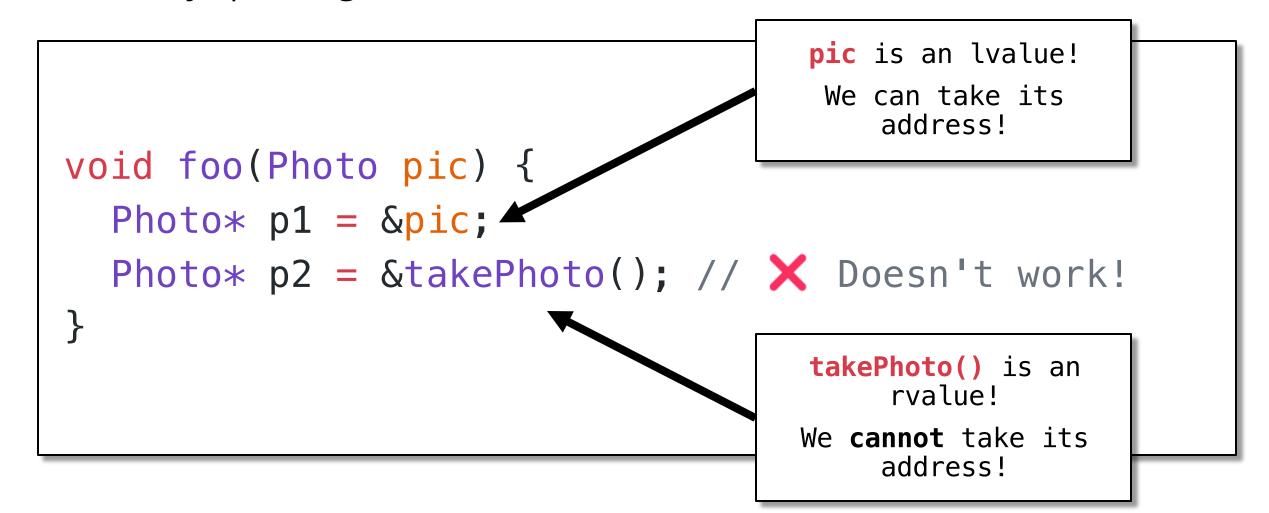
Ivalues & rvalues

Ivalues and rvalues generalize the idea of "temporariness" in C++

```
pic is an lvalue!
void foo(Photo pic) {
  Photo beReal = pic;
  Photo insta = takePhoto();
                                      takePhoto() is an
                                           rvalue!
```

Ivalues & rvalues

Generally speaking, Ivalues have a definite address, rvalues do not!



An Ivalue can appear on either side of an =

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

An rvalue can appear only right of an =

```
x = 5;

<del>5 = y;</del>
```

Your Turn

Which of the following <u>right-hand assignments</u> are rvalues?

Hint: which ones have a definite address?

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

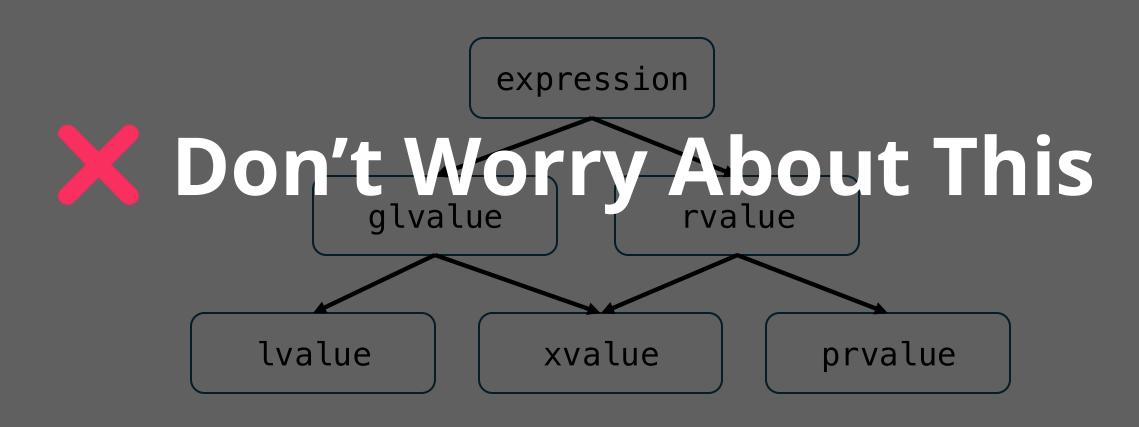
An Ivalue's lifetime is until the end of scope

An rvalue's lifetime is until the end of line

An Ivalue is persistent

An rvalue is temporary

Quick Note: It's more complicated than this!



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); // <a> Unnecessary copy is made here
```

Working towards move semantics

We can pass by reference! 😚

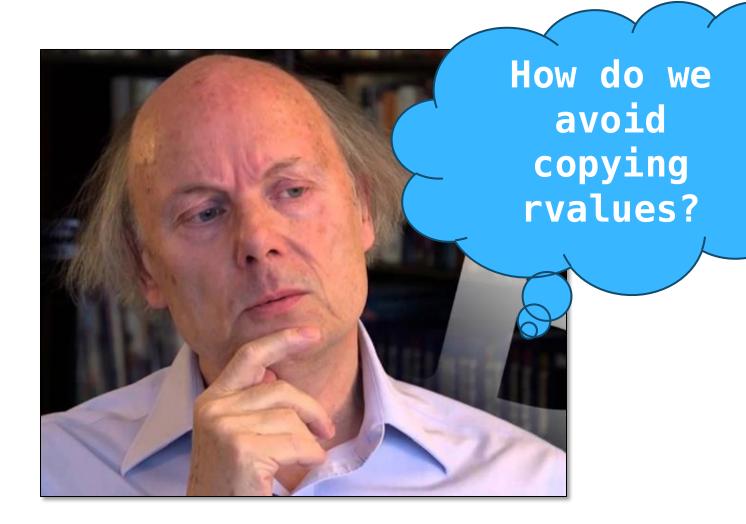
```
void uploadToInsta(Photo& pic);
int main() {
  Photo selfie = takePhoto(); // selfie is lvalue
  uploadToInsta(selfie); // ✓ No copy is made here
```

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?
}
```

X candidate function not viable: expects lvalue as 1st argument



thinking_bjarne

Value reference

rvalue reference

```
void upload(Photo& pic);
int main() {
  Photo selfie = takePhoto();
  upload(selfie);
```

```
void upload(Photo&& pic);
int main() {
  upload(takePhoto());
```

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

A few important points

- **Ivalue** 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!

```
//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);
```

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

Ivalue/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!



bjarne_about_to_raise_hand

Move Semantics

What we want!

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

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

Move 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;
```

```
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)



bjarne_about_to_raise_hand

std::move and SMFs

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 Ivalue 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

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

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;
```

Be wary of std::move

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

```
Photo takePhoto();
void foo(Photo whoAmI)
  Photo selfie = std::move(whoAmI);
  whoAmI_get_pixel(21, 24); // ???
```

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

std::move doesn't do anything special!

std::move just type casts an Ivalue 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!

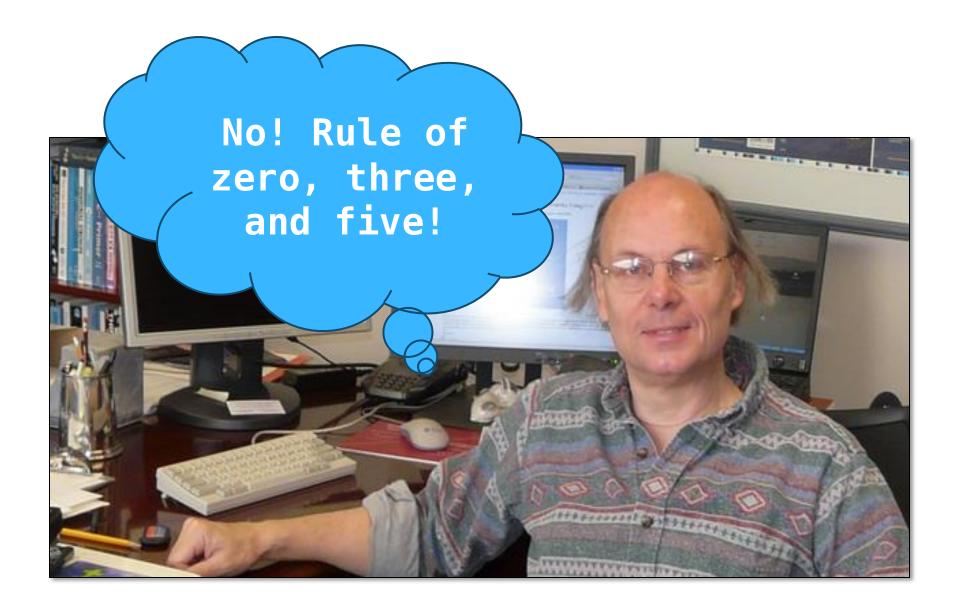


bjarne_about_to_raise_hand

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!?



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 caption;
};
```

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)



bjarne_about_to_raise_hand