CS106L Lecture 13:

Move Semantics (C++11)

Winter 2024

Fabio Ibanez, Haven Whitney

Attendance

http://tinyurl.com/moveW24

Plan

1. Lvalues, Rvalues review
2. Why do we need move semantics?
3. `std::move()`
4. Move constructor and move assignment operator
5. Rule of Zero, Three, and Five

There are six special member functions!

These functions are generated only when they're called (and before any are explicitly defined by you):

```
class Widget {
 public:
   Widget();                                // default constructor
   Widget (const Widget& w);                // copy constructor
   Widget& operator = (const Widget& w);    // copy assignment operator
   ~Widget();                               // destructor
   Widget (Widget&& rhs);                   // move constructor
   Widget& operator = (Widget&& rhs);       // move assignment operator
}
```

There are six special member functions!

These functions are generated only when they're called (and before any are explicitly defined by you):

```
class Widget {
  public:
    Widget();                                  // default constructor
    Widget (const Widget& w);                  // copy constructor
    Widget& operator = (const Widget& w);      // copy assignment operator
    ~Widget();                                 // destructor
    Widget (Widget&& rhs);                     // move constructor
    Widget& operator = (Widget&& rhs);         // move assignment operator
}
```

Before any of that

An l-value

An **l-value** can be to the left ***or*** the right of an equal sign!

What's an example?

`x` can be an l-value for instance because you can have something like:

```
int y = x
```

✅ **AND** ✅

```
x = 344
```

An r-value

An **r-value** can be ⭐***ONLY***⭐ to the right of an equal sign!

What's an example?

`21` can be an r-value for instance because you can have something like:

```
int y = 21
```

❌ **BUT NOT** ❌

```
21 = x
```

L-value & R-value lifetime

L-values live until the end of the scope

R-values live until the end of the line

L-value & R-value lifetime

L-values live until the end of the scope

R-values live until the end of the line

L-value & R-value examples

```
int x = 3;
int *ptr = 0x02248837;
vector<int> v1{1, 2, 3};
size_t size = v.size();
v1[1] = 4*i;
ptr = &x;
v1[2] = *ptr;
MyClass obj;
x = obj.public_member_variable;
```

Credit to Sarah McCarthy

L-value & R-value examples

```
int x = 3;                //3 is an r-value
int *ptr = 0x02248837;
vector<int> v1{1, 2, 3};
size_t size = v.size();
v1[1] = 4*i;
ptr = &x;
v1[2] = *ptr;
MyClass obj;
x = obj.public_member_variable;
```

Credit to Sarah McCarthy

L-value & R-value examples

```
int x = 3;                    //3 is an r-value
int *ptr = 0x02248837;        //0x02248837 is an r-value
vector<int> v1{1, 2, 3};
size_t size = v.size();
v1[1] = 4*i;
ptr = &x;
v1[2] = *ptr;
MyClass obj;
x = obj.public_member_variable;
```

Credit to Sarah McCarthy

L-value & R-value examples

```
int x = 3;                  //3 is an r-value
int *ptr = 0x02248837;      //0x02248837 is an r-value
vector<int> v1{1, 2, 3};    //{1, 2, 3} is an r-value,v1 is an l-value
size_t size = v.size();
v1[1] = 4*i;
ptr = &x;
v1[2] = *ptr;
MyClass obj;
x = obj.public_member_variable;
```

Credit to Sarah McCarthy

L-value & R-value examples

```
int x = 3;                      //3 is an r-value
int *ptr = 0x02248837;          //0x02248837 is an r-value
vector<int> v1{1, 2, 3};        //{1, 2, 3} is an r-value,v1 is an l-value
size_t size = v.size();
v1[1] = 4*i;
ptr = &x;
v1[2] = *ptr;
MyClass obj;
x = obj.public_member_variable;
```

Credit to Sarah McCarthy

L-value & R-value examples

```
int x = 3;                   //3 is an r-value
int *ptr = 0x02248837;       //0x02248837 is an r-value
vector<int> v1{1, 2, 3};     //{1, 2, 3} is an r-value,v1 is an l-value
size_t size = v.size();      //v.size()is an r-value
v1[1] = 4*i;
ptr = &x;
v1[2] = *ptr;
MyClass obj;
x = obj.public_member_variable;
```

Credit to Sarah McCarthy

L-value & R-value examples

```
int x = 3;                      //3 is an r-value
int *ptr = 0x02248837;          //0x02248837 is an r-value
vector<int> v1{1, 2, 3};        //{1, 2, 3} is an r-value,v1 is an l-value
size_t size = v.size();         //v.size()is an r-value
v1[1] = 4*i;                    //4*i is an r-value, v1[1] is an l-value
ptr = &x;
v1[2] = *ptr;
MyClass obj;
x = obj.public_member_variable;
```

Credit to Sarah McCarthy

L-value & R-value examples

```
int x = 3;                  //3 is an r-value
int *ptr = 0x02248837;      //0x02248837 is an r-value
vector<int> v1{1, 2, 3};    //{1, 2, 3} is an r-value,v1 is an l-value
size_t size = v.size();     //v.size()is an r-value
v1[1] = 4*i;                //4*i is an r-value, v1[1] is an l-value
ptr = &x;                   //&x is an r-value
v1[2] = *ptr;
MyClass obj;
x = obj.public_member_variable;
```

Credit to Sarah McCarthy

L-value & R-value examples

```
int x = 3;                      //3 is an r-value
int *ptr = 0x02248837;          //0x02248837 is an r-value
vector<int> v1{1, 2, 3};        //{1, 2, 3} is an r-value,v1 is an l-value
size_t size = v.size();         //v.size()is an r-value
v1[1] = 4*i;                    //4*i is an r-value, v1[1] is an l-value
ptr = &x;                       //&x is an r-value
v1[2] = *ptr;                   //*ptr is an l-value
MyClass obj;
x = obj.public_member_variable;
```

Credit to Sarah McCarthy

L-value & R-value examples

```
int x = 3;                        //3 is an r-value
int *ptr = 0x02248837;            //0x02248837 is an r-value
vector<int> v1{1, 2, 3};          //{1, 2, 3} is an r-value,v1 is an l-value
size_t size = v.size();           //v.size()is an r-value
v1[1] = 4*i;                      //4*i is an r-value, v1[1] is an l-value
ptr = &x;                         //&x is an r-value
v1[2] = *ptr;                     //*ptr is an l-value
MyClass obj;                      //obj is an l-value
x = obj.public_member_variable;
```

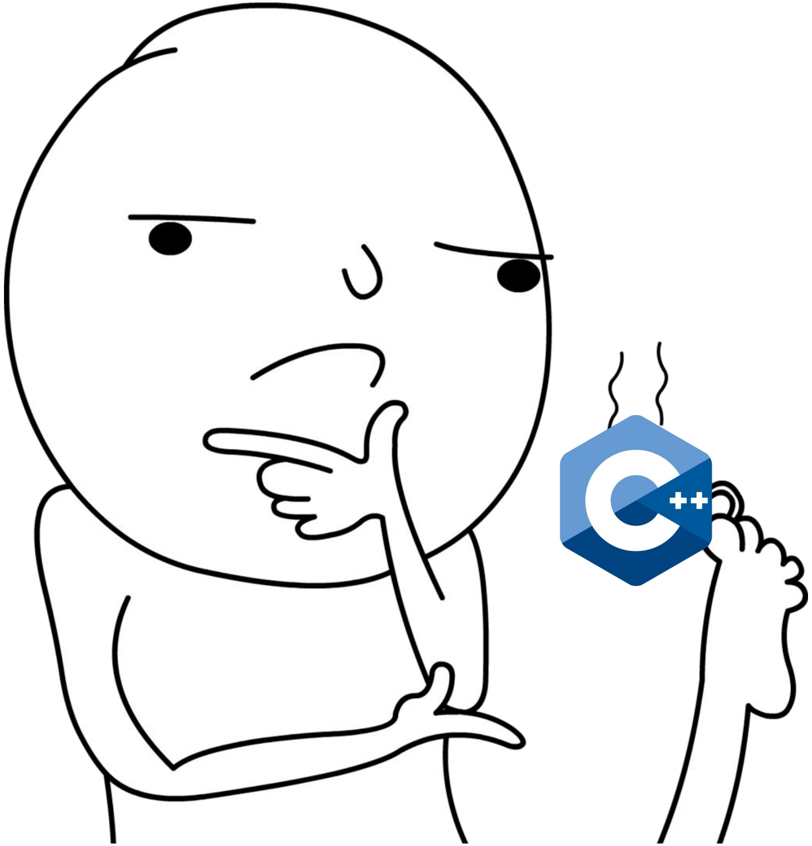
Credit to Sarah McCarthy

L-value & R-value examples

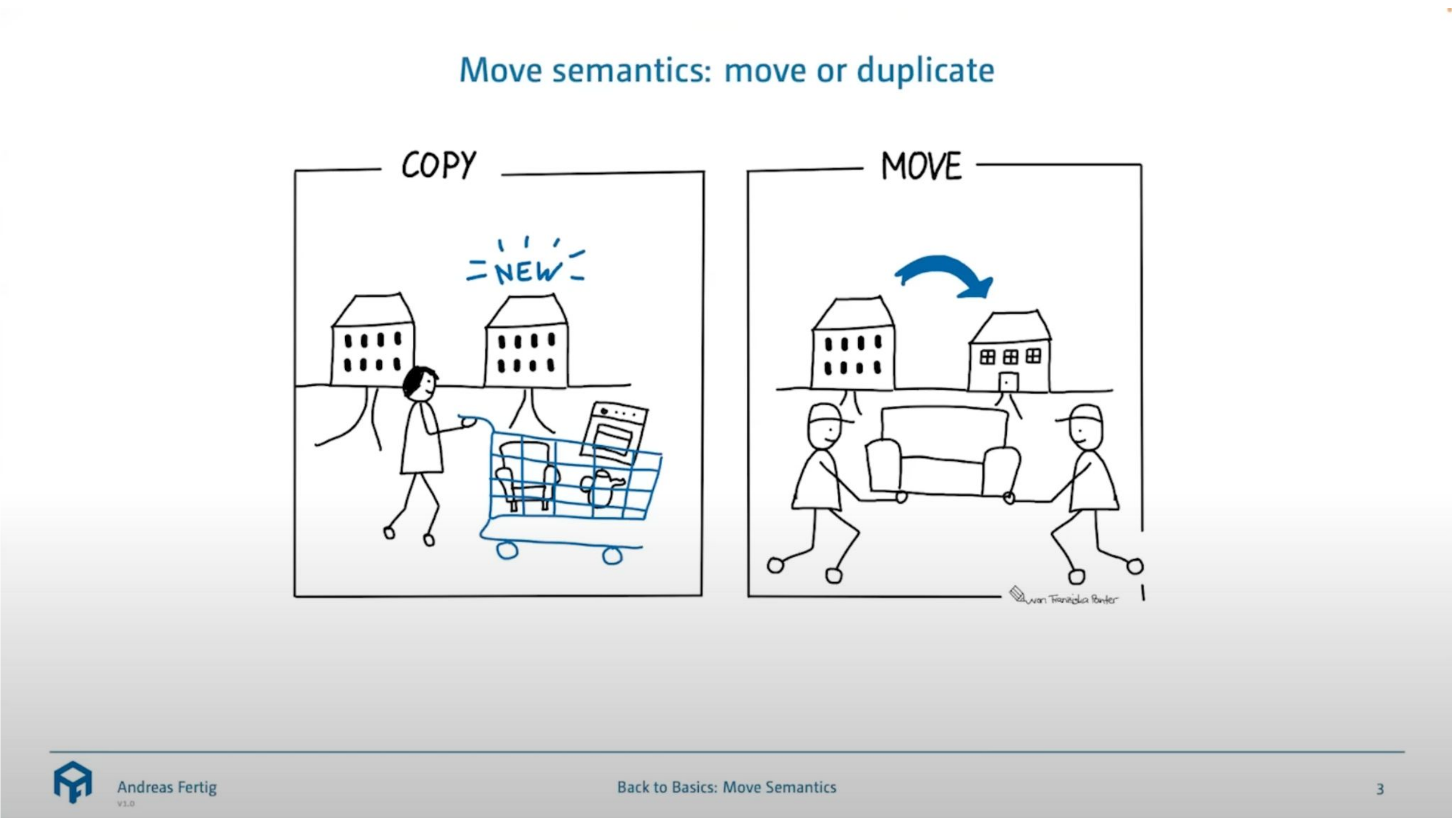
```
int x = 3;                          //3 is an r-value
int *ptr = 0x02248837;              //0x02248837 is an r-value
vector<int> v1{1, 2, 3};            //{1, 2, 3} is an r-value,v1 is an l-value
size_t size = v.size();             //v.size()is an r-value
v1[1] = 4*i;                        //4*i is an r-value, v1[1] is an l-value
ptr = &x;                           //&x is an r-value
v1[2] = *ptr;                       //*ptr is an l-value
MyClass obj;                        //obj is an l-value
x = obj.public_member_variable;     //obj.public_member_variable is l-value
```

Credit to Sarah McCarthy

What questions do we have?



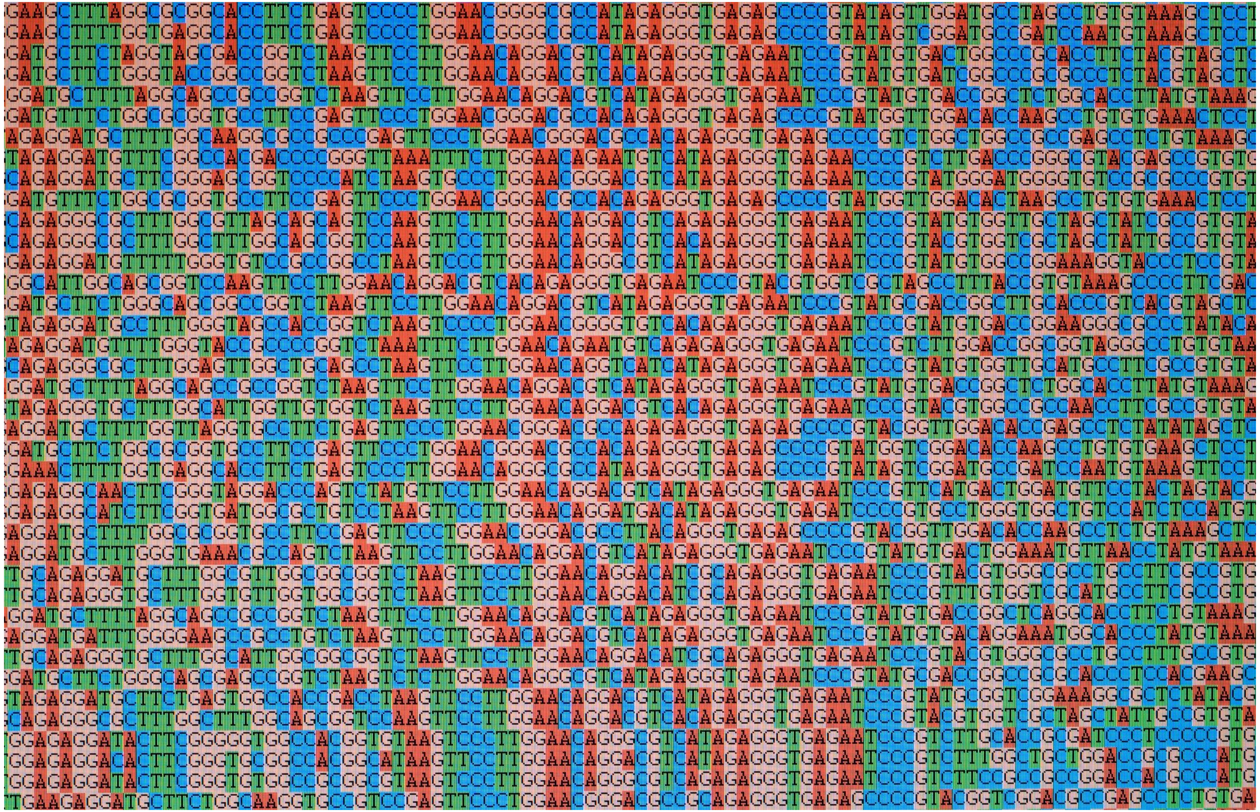
A good way to prime move semantics



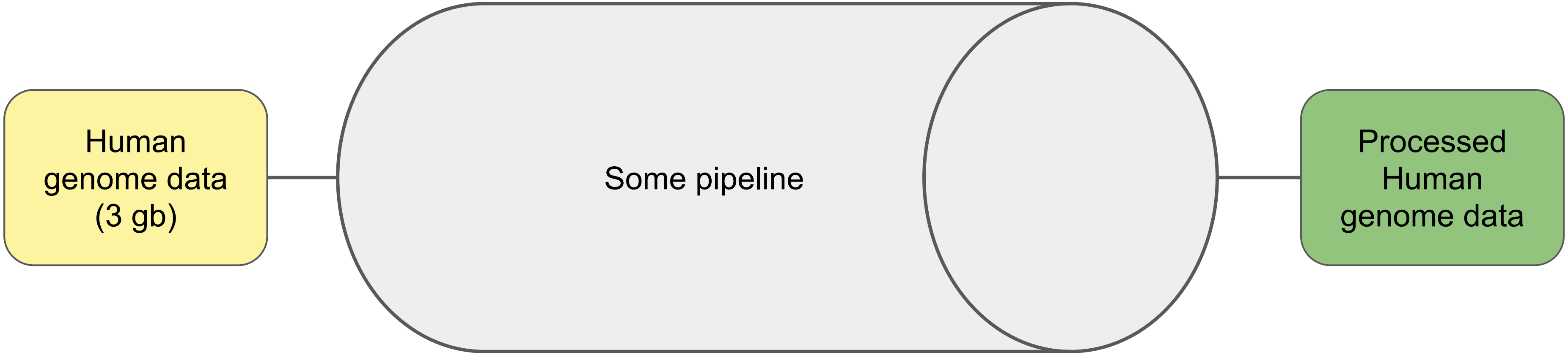
I really like this way of thinking about move semantics:

Watch the full video here

Motivation

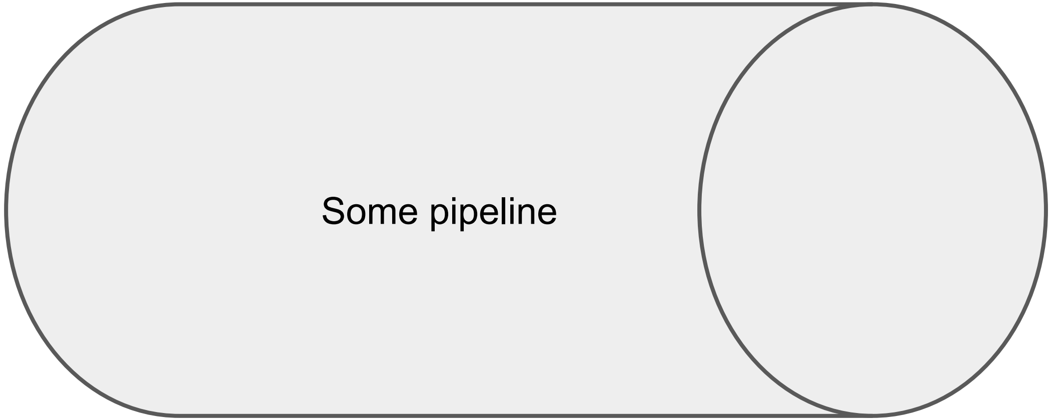


Motivation

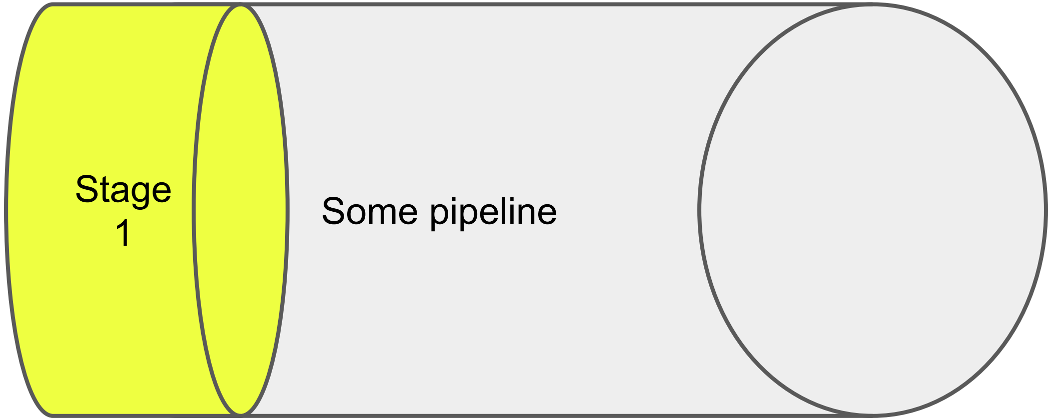


Human genome data (3 gb)
Some pipeline
Processed Human genome data

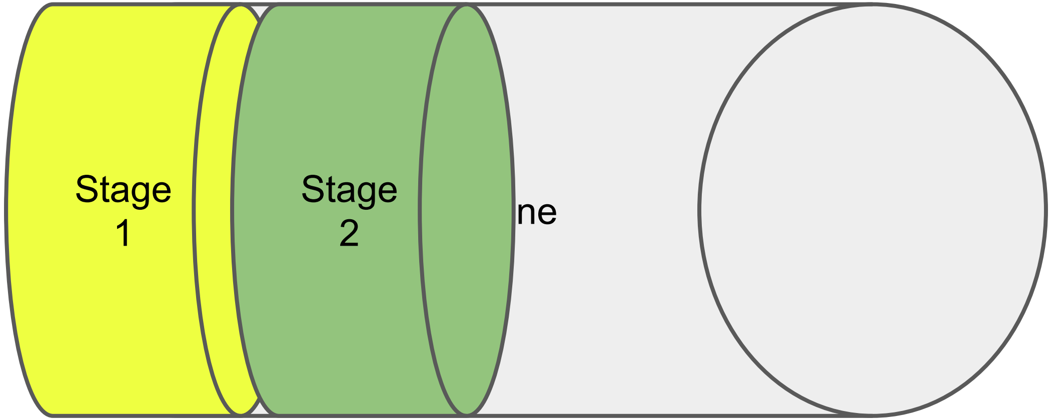
Motivation



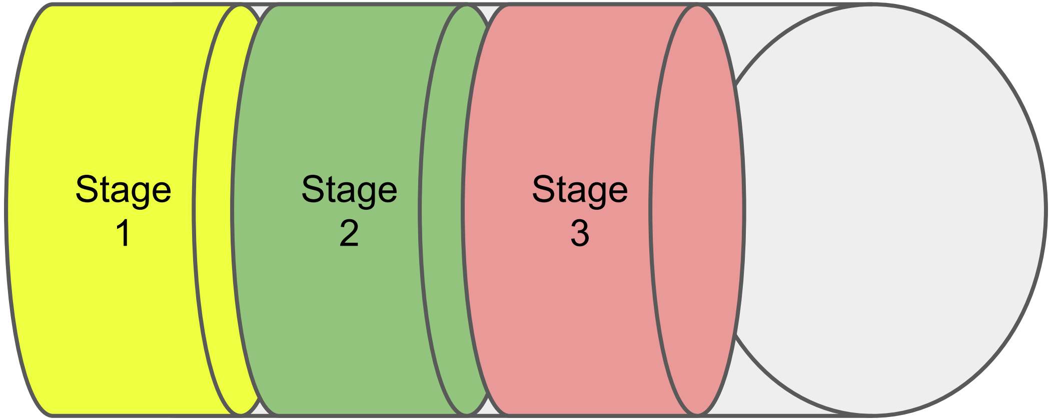
Motivation



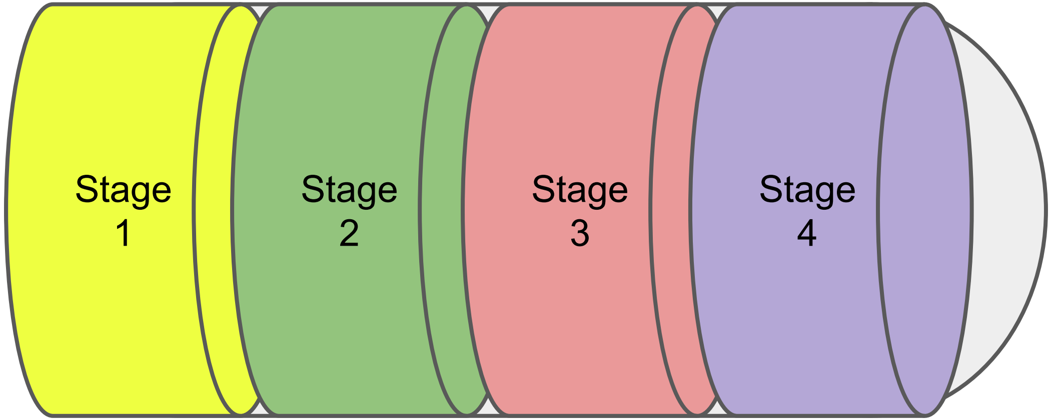
Motivation



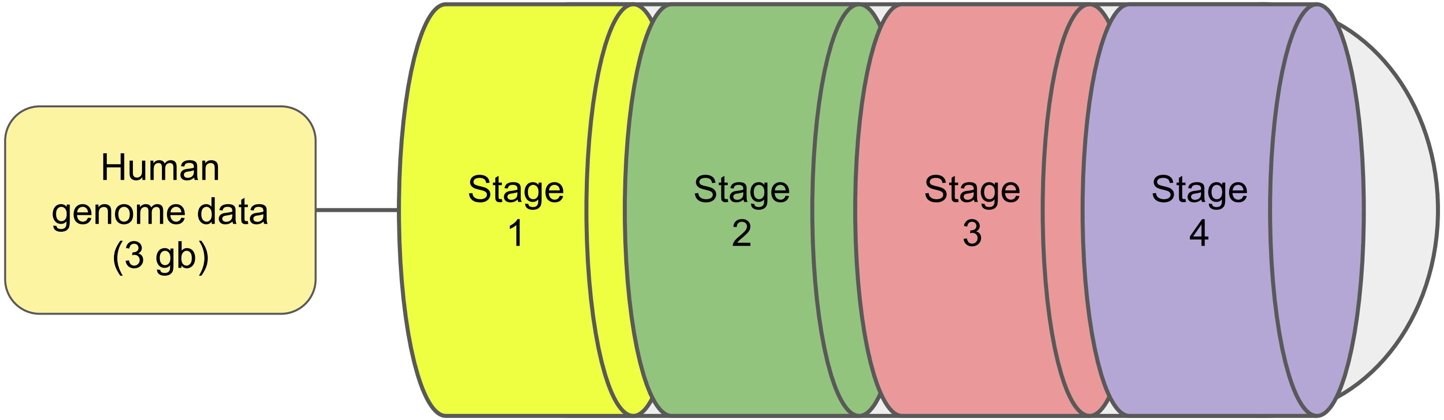
Motivation



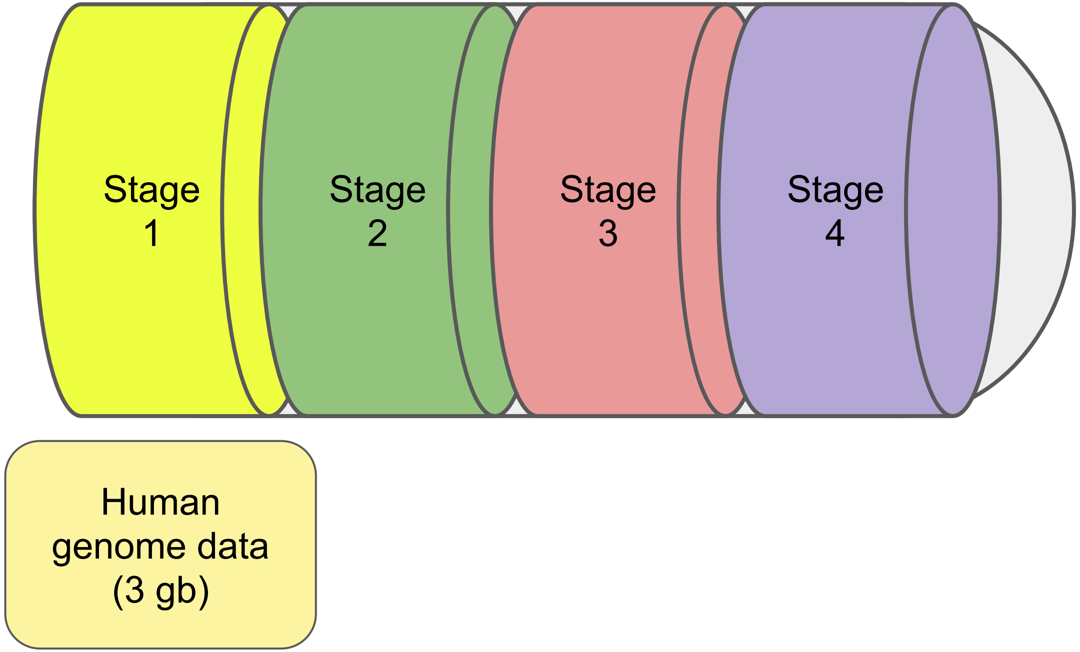
Motivation



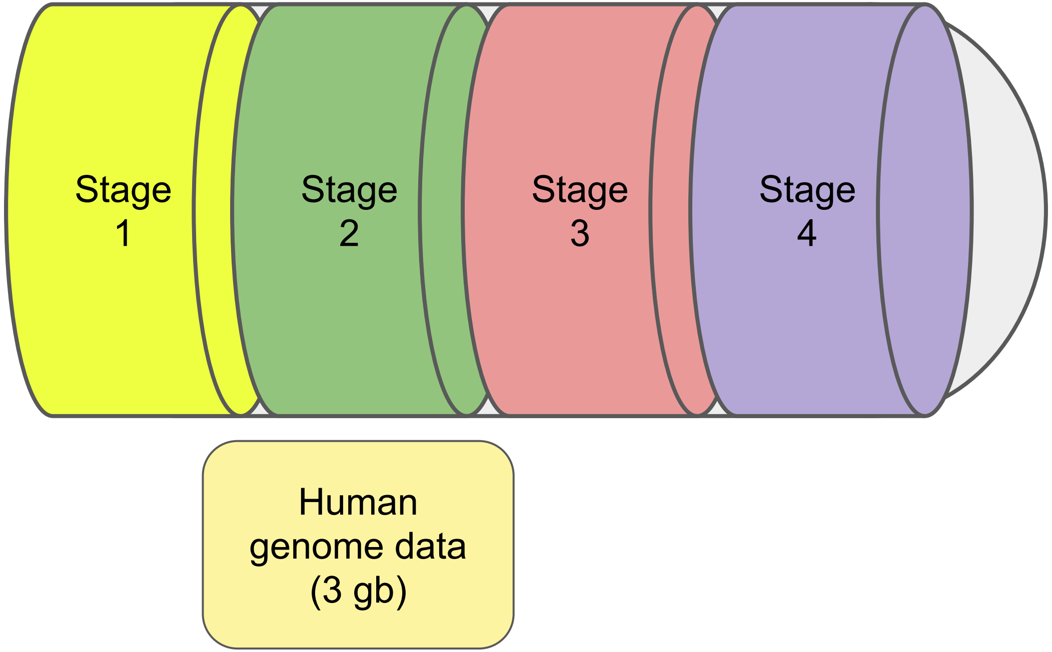
Motivation



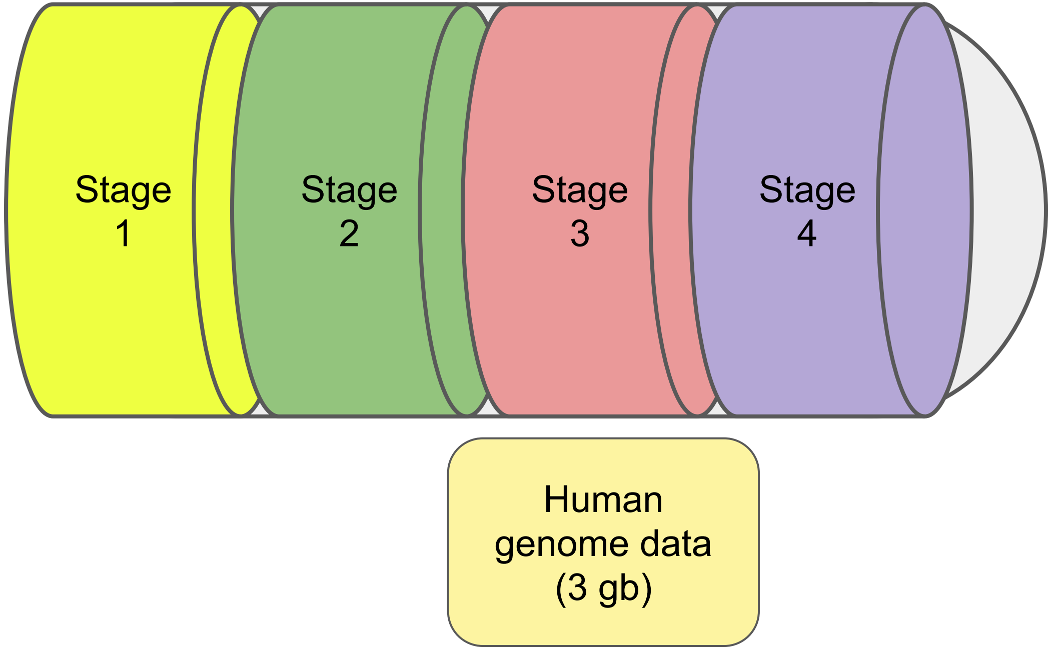
Motivation



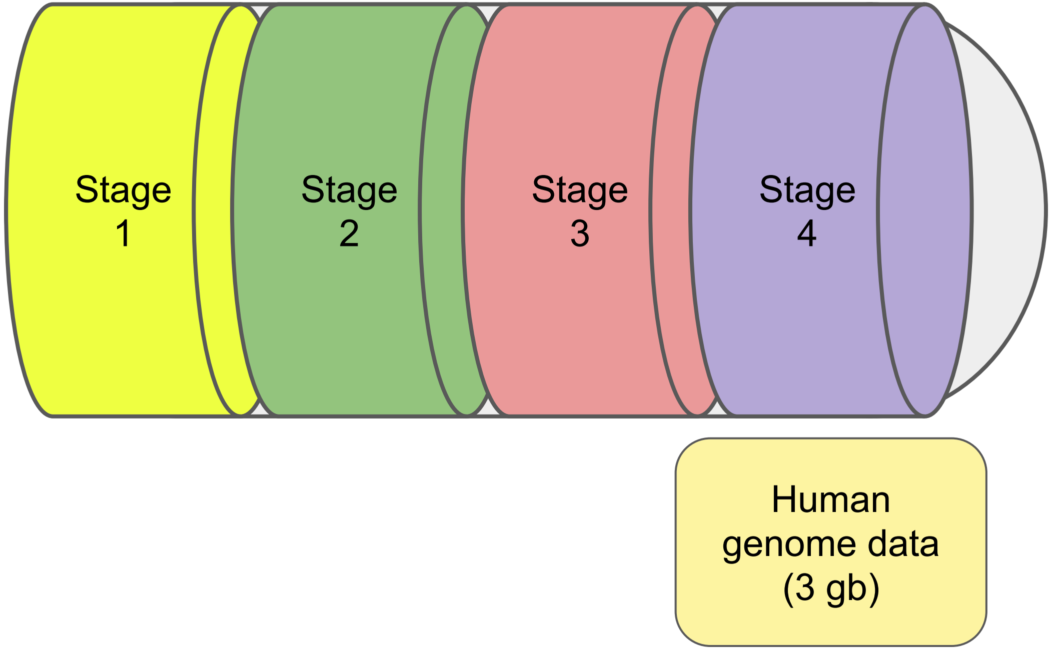
Motivation



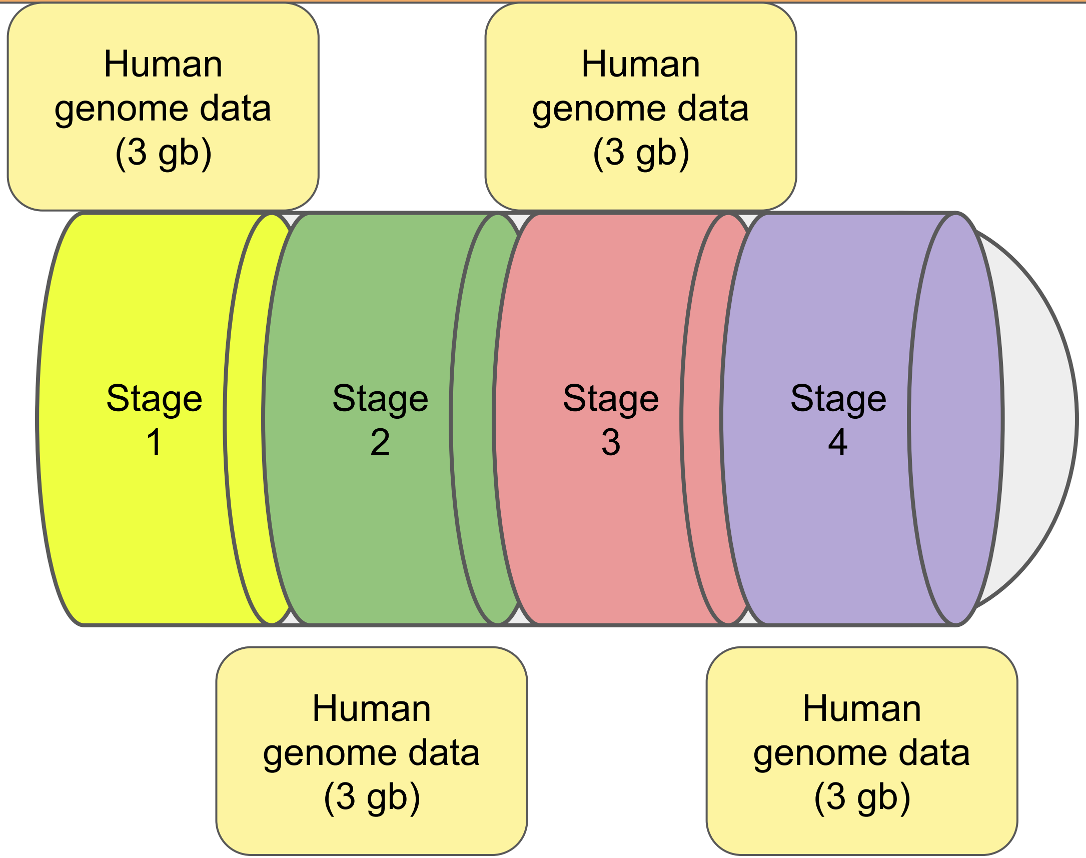
Motivation



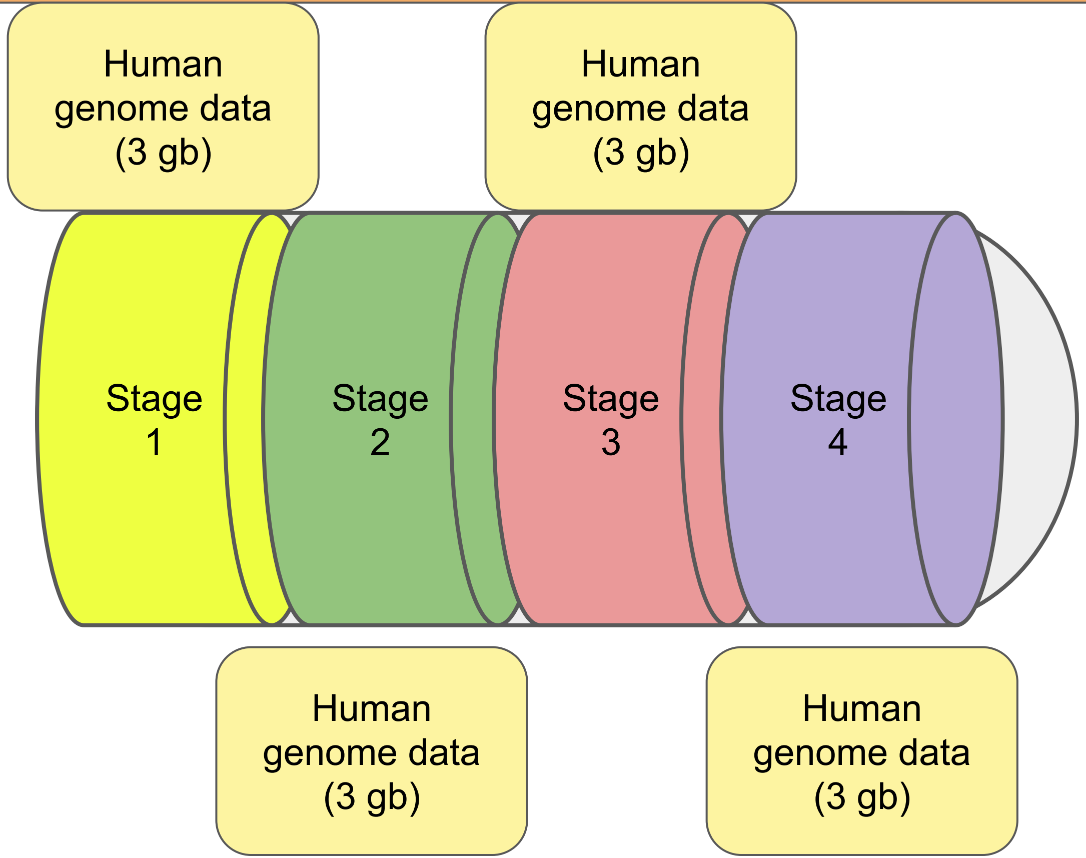
Motivation



Imagine this



Imagine this

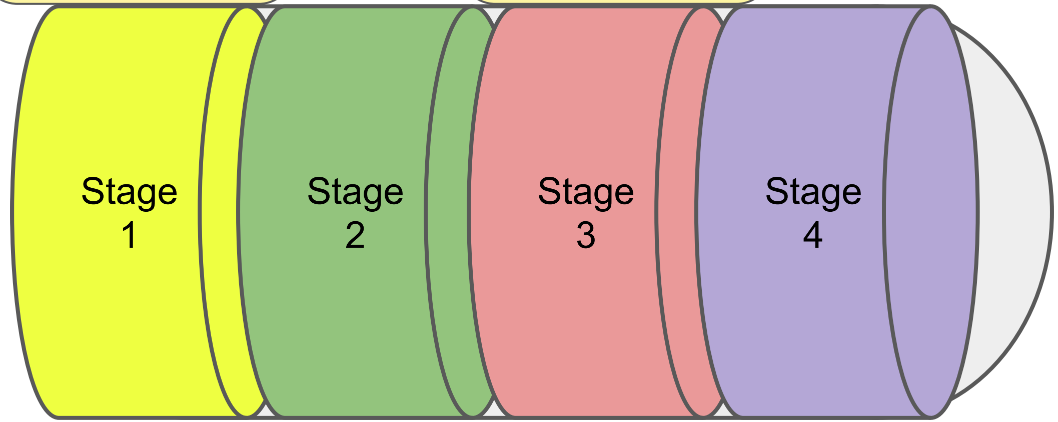


Anyone see a problem here? 🤔

Imagine this

Human genome data (3 gb)

Human genome data (3 gb)

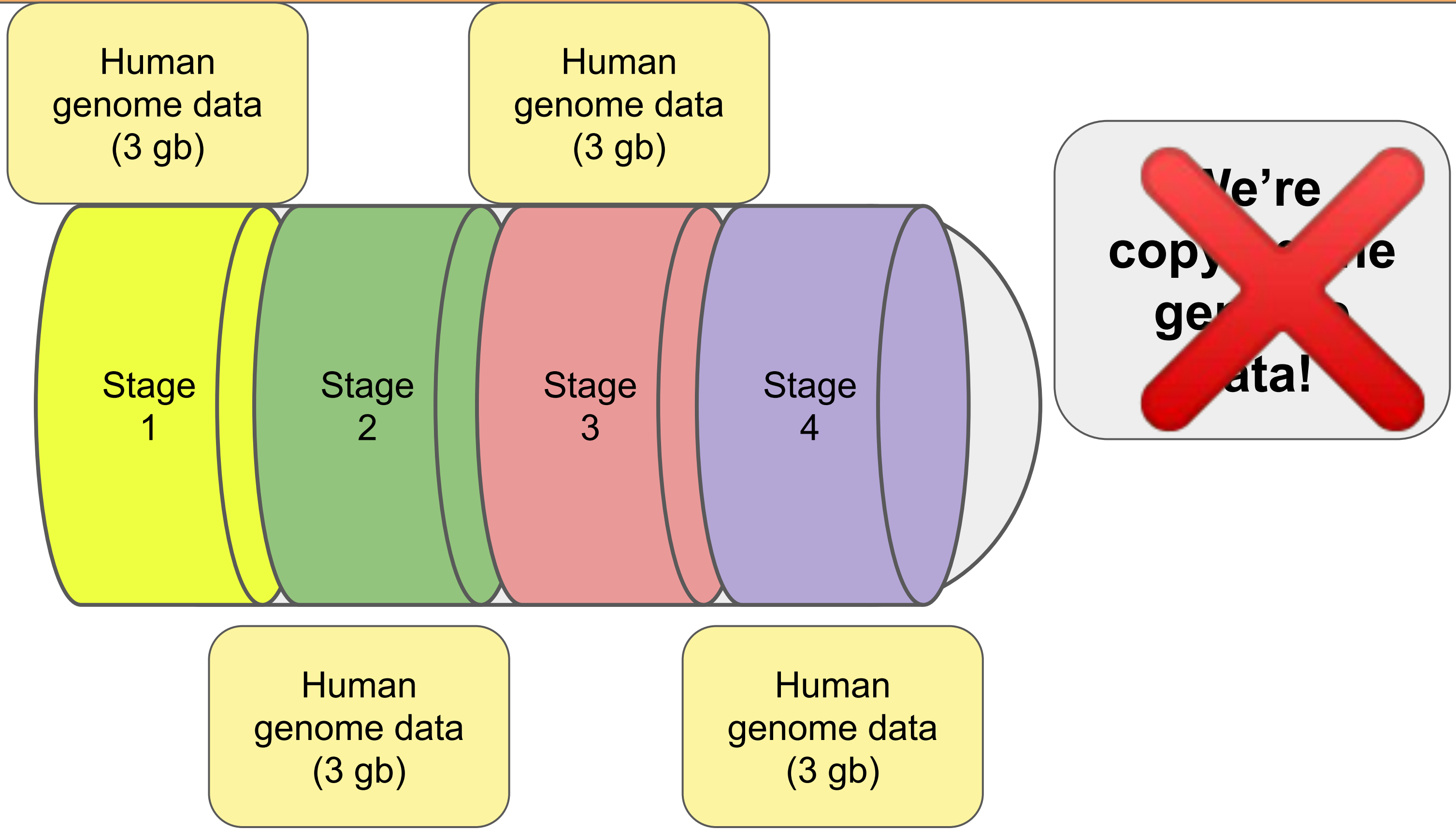


Human genome data (3 gb)

Human genome data (3 gb)

We're copying the genome data!

Imagine this



What does this look like in code?

```
class HumanGenome {
private:
    std::vector<char> data;
public:
    HumanGenome() = default;

    HumanGenome(size_t size): data(size) {
        std::fill(data.begin(), data.end(), ‘A’);
    }
}
```

A new type of constructor

```
class HumanGenome {
private:
    std::vector<char> data;
public:
    // move constructor
    HumanGenome(HumanGenome&& other) noexcept :
    data(std::move(other.data)) {
        std::cout << "HumanGenome moved into stage." << std::endl;
    }
}
```

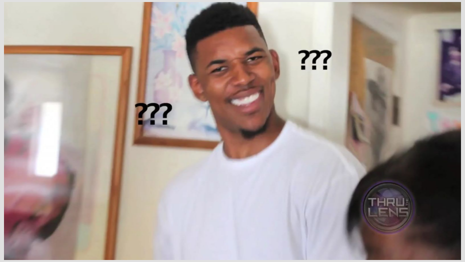
A new type of constructor

```
class HumanGenome {
private:
    std::vector<char> data;
public:
    // move constructor
    HumanGenome(HumanGenome&& other) noexcept
    data(std::move(other.data)) {
        std::cout << "HumanGenome moved into stage." << std::endl;
    }
}
```

This basically says *"hey I guarantee not to throw an exception"*

A new type of constructor

```
class HumanGenome {
private:
    std::vector<char> data;
public:
    // move constructor
    HumanGenome(HumanGenome&& other) noexcept :
    data(std::move(other.data)) {
        std::cout << "HumanGenome moved into stage." << std::endl;
    }
}
```



A new type of constructor

```
class HumanGenome {
private:
    std::vector<char> data;
public:
    // move constructor
    HumanGenome(HumanGenome&& other) noexcept :
    data(std::move(other.data)) {
        std::cout << "HumanGenome moved into stage." << std::endl;
    }
}
```

This basically says *"I'm gonna yank this thing's resource, I will treat it as an r-value"*

When would this be used?

```
HumanGenome stage1(HumanGenome genome) {
    genome.process(); // assume some process function exists in HumanGenome
    return genome;
}
HumanGenome stage2(HumanGenome genome) {
    genome.process();
    return genome;
}
HumanGenome stage3(HumanGenome genome) {
    genome.process();
    return genome;
 }
```

When would this be used?

```
HumanGenome stage1(HumanGenome genome) {
    genome.process(); // assume some process function exists in HumanGenome
    return genome;
}
HumanGenome stage2(HumanGenome genome) {
    genome.process();
    return genome;
}
HumanGenome stage3(HumanGenome genome) {
    genome.process();
    return genome;
 }
```

Does anyone remember what special member function is called here?

When would this be used?

```
HumanGenome stage1(HumanGenome genome) {
    genome.process(); // assume some process function exists in HumanGenome
    return genome;
}
HumanGenome stage2(HumanGenome genome) {
    genome.process();
    return genome;
}
HumanGenome stage3(HumanGenome genome) {
    genome.process();
    return genome;
 }
```

Copy constructor!

When would this be used?

```cpp
std::vector<char> initialData = {'A', 'T', 'G', 'C'};

HumanGenome genome(initialData);

/// pipelines are independent of each other
genome = stage1(genome);
genome = stage2(genome);
genome = stage3(genome);
```

Here we're not making use of our move semantics!

When would this be used?

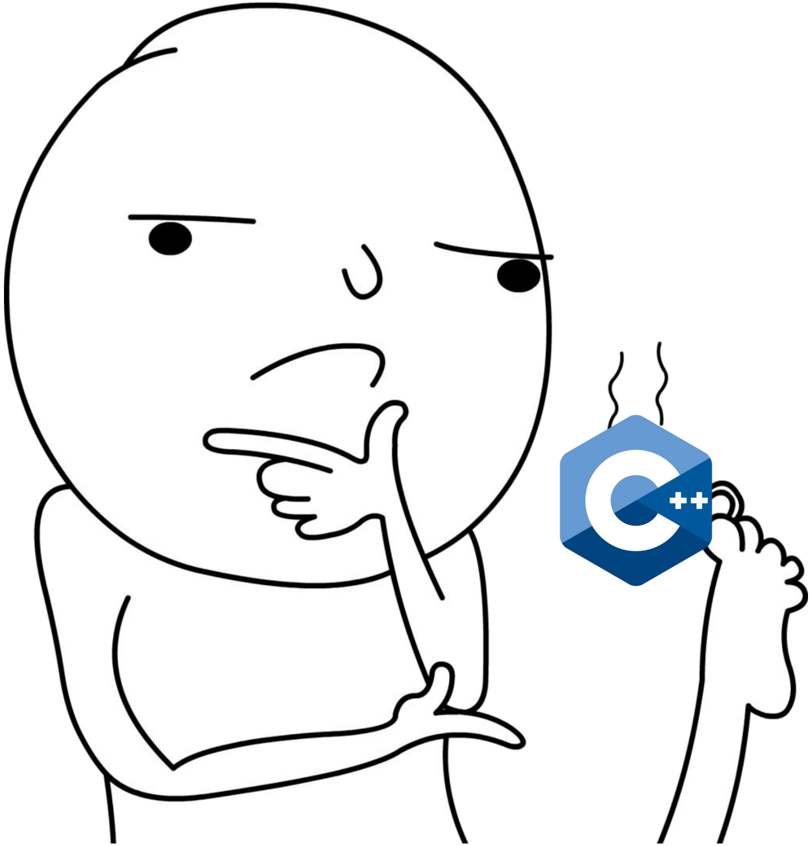
```
std::vector<char> initialData = {'A', 'T', 'G', 'C'};

HumanGenome genome(initialData);

/// pipelines are independent of each other
genome = stage1(std::move(genome));
genome = stage2(std::move(genome));
genome = stage3(std::move(genome));
```

Explicitly moving the genome object

What questions do we have?



A new type of operator

```
class HumanGenome {
private:
    std::vector<char> data;
public:
    // Move assignment operator
    HumanGenome& operator=(HumanGenome&& other) noexcept {
        if (this != &other) {
            data = other.data;
            std::cout << "HumanGenome moved within stage." << std::endl;
        }
        return *this;
    }
}
```

A new type of operator

```
class HumanGenome {
private:
    std::vector<char> data;
public:
    // Move assignment operator
    HumanGenome& operator=(HumanGenome&& other) noexcept {
        if (this != &other) {
            data = other.data;
            std::cout << "HumanGenome moved within stage." << std::endl;
        }
        return *this;
    }
}
```

Does anyone see a problem here though?

A new type of operator

```
class HumanGenome {
private:
    std::vector<char> data;
public:
    // Move assignment operator
    HumanGenome& operator=(HumanGenome&& other) noex
        if (this != &other) {
            data = other.data;
            std::cout << "HumanGenome moved within s                ;
        }
        return *this;
    }
}
```

This is actually performing a copy! This defeats the purpose of move

A new type of operator

```cpp
class HumanGenome {
private:
    std::vector<char> data;
public:
    // Move assignment operator
    HumanGenome& operator=(HumanGenome&& other) noexcept {
        if (this != &other) {
            data = std::move(other.data);
            std::cout << "HumanGenome moved within stage." << std::endl;
        }
        return *this;
    }
}
```

std::move

```
class HumanGenome {
private:
    std::vector<char> data;
public:
    // Move assignment operator
    HumanGenome& operator=(HumanGenome&& other) noexcept {
        if (this != &other) {
            data = std::move(other.data);
            std::cout << "HumanGenome moved with
        }
        return *this;
    }
}
```

It turns out that `other.data` is an l-value

std::move

```
class HumanGenome {
private:
    std::vector<char> data;
public:
    // Move assignment operator
    HumanGenome& operator=(HumanGenome&& other) noexcept {
        if (this != &other) {
            data = std::move(other.data);
            std::cout << "HumanGenome moved with
        }
        return *this;
    }
}
```

`std::move()` changes an l-value to an x-value

x-value

> You can plunder me, **move** anything I'm holding and use it elsewhere (since I'm going to be destroyed soon anyway)".

[Check this out if you're interested!]

x-value

> You can plunder me, **move** any… g a
> going to be destroyed soon any…

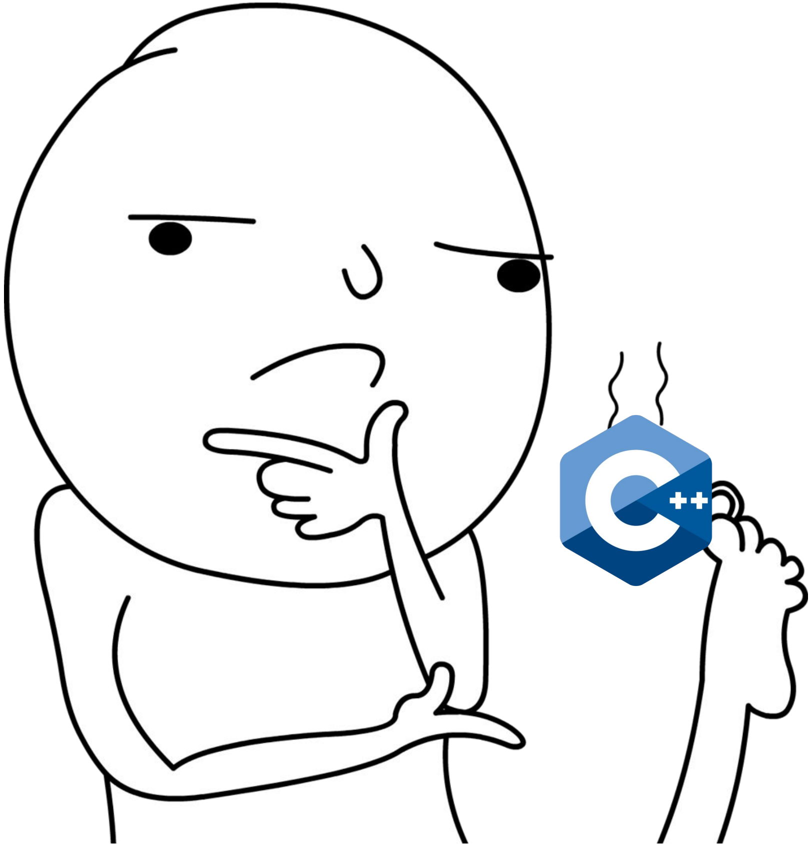
Don’t worry about this too much! This is an aside. Just understand what `std::move()` is doing on a *philosophical* level.

[k this out if you’re intere]

std::move()

Whenever the original object is no longer needed you can use **std::move()** to transfer as opposed to copy

What questions do we have?



std::move

```
int main() {
    HumanGenome genome_one;
    HumanGenome genome_two;

    // add a base to genome_one; assume add_base method exists
    genome_one.add_base('A');
    genome_two = genome_one;
    genome_one.add_base('T');
}
```

Is there an issue here? 🤔

We need both a copy and move constructor!

```
int main() {
    HumanGenome genome_one;
    HumanGenome genome_two;

    // add a base to genome_one; assume add_base method exists
    genome_one.add_base('A');
    genome_two = genome_one;
    genome_one.add_base('T');
}
```

If we don't have a copy constructor we are doing an illegal **`add_base`**!

Operator overloading

Operator Overloading

http://web.stanford.edu/class/cs106l/

What operators can we overload?

Most of them, actually!

```
+  -  *  /  %  ^  &  |  ~  !  ,  =  <  >  <=  >=
++  --  <<  >>  ==  !=  &&  ||  +=  -=  *=
/=  %=  ^=  &=  |=  <<=  >>=  []  ()  ->
->*  new  new[]  delete  delete[]
```

Operator overloading

Operator Overloading

http://web.stanford.edu/class/cs106l/

What operators can we overload?

Most of them, actually!

You can overload the assignment operator!

```
+   -   *   /   %   ^   &   |   ~   !   ,   =   <   >   <=   >=
++   --   <<   >>   ==   !=   &&   ||   +=   -=   *=
/=   %=   ^=   &=   |=   <<=   >>=   []   ()   ->
->*   new   new[]   delete   delete[]
```

Operator overloading

Copy assignment

```
HumanGenome& operator=(const HumanGenome&
other) {
     if (&other == this) return *this;
     data = other.data;

     return *this;
}
```

Move assignment

```
HumanGenome& operator=(HumanGenome&&
other) noexcept {
        if (this != &other) {
            data = std::move(other.data);
            std::cout << "HumanGenome
moved within stage." << std::endl;
        }
        return *this;
 }
```

We need both a copy and move constructor!

```
int main() {
    HumanGenome genome_one;
    HumanGenome genome_two;

    // add a base to genome_one; assume add_base method exists
    genome_one.add_base('A');
    genome_two = genome_one;
    genome_one.add_base('T');
}
```

Happy Bjarne, this works now!

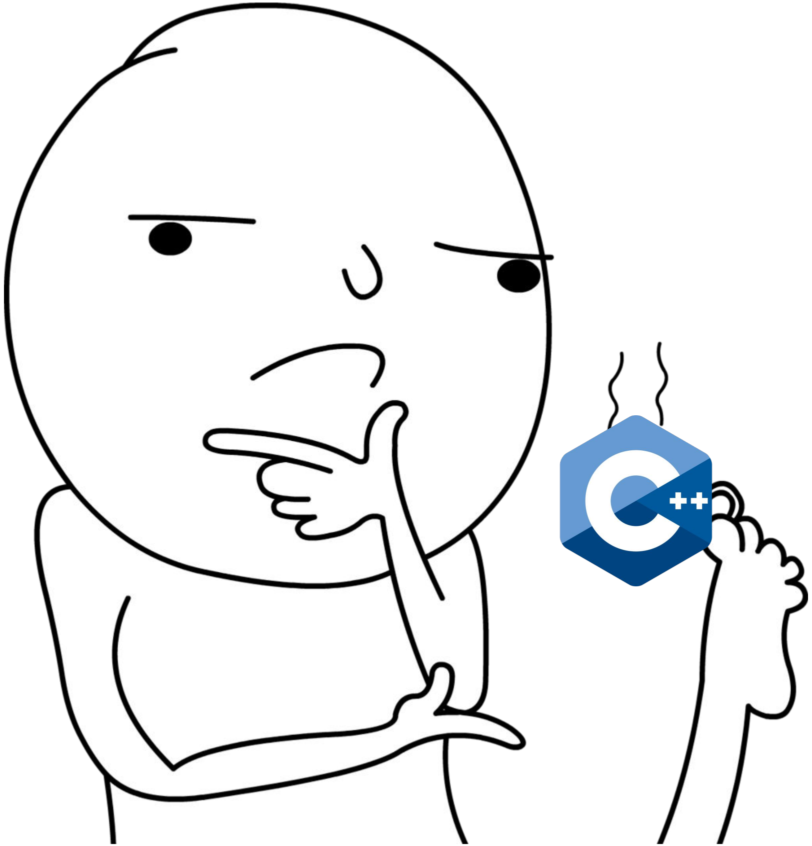
We need both a copy and move constructor!

```
int main() {
    HumanGenome genome_one;
    HumanGenome genome_two;

    // add a base to genome_one; assume add_base method exists
    genome_one.add_base('A');
    genome_two = genome_one; // uses the copy assignment operator!
    genome_one.add_base('T');
}
```

Happy Bjarne, this works now!

What questions do we have?



Circling back to `std::move()`

- You should use this when you're assigning some l-value that is no longer needed where it is previously stored

Circling back to `std::move()`

- You should use this when you're assigning some l-value that is no longer needed where it is previously stored
- Generally, we want to avoid using **`std::move()`** in application code. Use it in class definitions, like constructors and operators.
    - The compiler can do much of the optimizations without you needing to do **`std::move()`** if you define the move constructor and move assignment operator.

Why?

```
int main() {
    vector<string> vec1 = {"hello", "world"}
    vector<string> vec2 = std::move(vec1);
    vec1.push_back("Sure hope vec2 doesn't see this!")
}
```

Credit to Sarah McCarthy

Why?

```
int main() {
    vector<string> vec1 = {“hello”, “world”}
    vector<string> vec2 = std::move(vec1);
    vec1.push_back(“Sure hope vec2 doesn’t see this!”)
}
```

Credit to Sarah McCarthy

Why?

```
int main() {
    vector<string> vec1 = {“hello”, “world”}
    vector<string> vec2 = std::move(vec1);
    vec1.push_back(“Sure hope vec2 doesn’t see this!”)
}
```

In application code we might make a mistake like this and try to **`push_back()`** to a moved object.

Credit to Sarah McCarthy

Why?

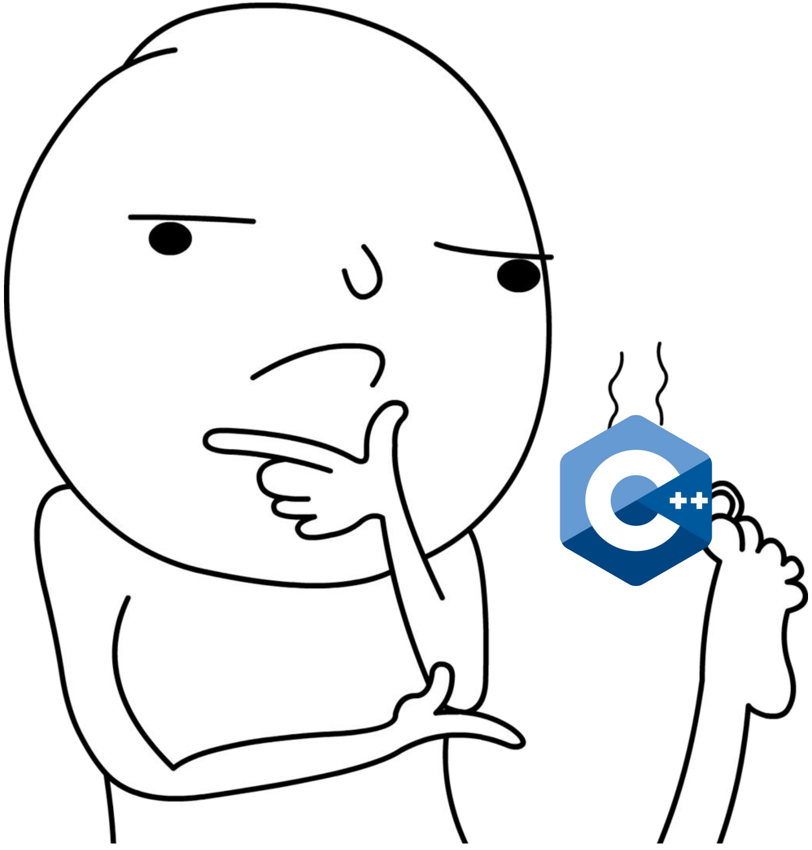
```
int main() {
    vector<string> vec1 = {“hello”, “world”}
    vector<string> vec2 = std::move(vec1);
    vec1.push_back(“Sure hope vec2 doesn’t see this!”)
}
```

In application code we might make a mistake like this and try to **`push_back()`** to a moved object.

Credit to Sarah McCarthy


use std::move()
in application code
Just let the
compiler decide which
class method to use
JUST LET THE
COMPILER CHOOSE BRO
JAKE-CLARK.TUMBLR
imgflip.com

What questions do we have?



Summarizing move semantics

- If your class has copy constructor and copy assignment defined, you should also define a move constructor and move assignment

Credit to Sarah McCarthy

Summarizing move semantics

- If your class has copy constructor and copy assignment defined, you should also define a move constructor and move assignment
- Define these by overloading your copy constructor and assignment to be defined for `Type&& other` as well as `Type& other`

Credit to Sarah McCarthy

Summarizing move semantics

- If your class has copy constructor and copy assignment defined, you should also define a move constructor and move assignment
- Define these by overloading your copy constructor and assignment to be defined for `Type&& other` as well as `Type& other`
- Use `std::move` to force the use of other types' move assignments and constructors

Credit to Sarah McCarthy

Summarizing move semantics

- If your class has copy constructor and copy assignment defined, you should also define a move constructor and move assignment
- Define these by overloading your copy constructor and assignment to be defined for `Type&& other` as well as `Type& other`

- Use `std::move` to force the use of other types' move assignments and constructors
- All `std::move(x)` does is cast `x` as an ~~rvalue~~ xvalue

Credit to Sarah McCarthy

Summarizing move semantics

- If your class has copy constructor and copy assignment defined, you should also define a move constructor and move assignment
- Define these by overloading your copy constructor and assignment to be defined for `Type&& other` as well as `Type& other`

- Use `std::move` to force the use of other types' move assignments and constructors
- All `std::move(x)` does is cast `x` as an ~~rvalue~~ xvalue
- Be wary of `std::move(x)` in main function code!

Credit to Sarah McCarthy

At this point:

You know about:

1. **Default constructor:** Initializes an object to a default state
2. **Copy constructor:** Creates a new object by copying an existing object
3. **Move constructor:** Creates a new object by moving the resources of an existing object
4. **Copy Assignment Operator:** Assigns the contents of one object to another object
5. **Move Assignment Operator:** Moves the resources of one object to another object
6. **Destructor**: Frees any dynamically allocated resources owned by an object when it is destroyed

Some philosophy!

Some philosophy about SMFs!

There are these three guiding principles we follow for special member functions (SMFs):

1. Rule of Zero
2. Rule of Three
3. Rule of Five

Rule of Zero

If you don’t need a constructor or a destructor or copy assignment etc. Then simply don’t use it!

If your class relies on objects/classes that already have these SMFs implemented, then there’s no need to reimplement this logic!

Rule of Zero

If you don't need a constructor or a destructor or copy assignment etc. Then simply don't use it!

If your class relies on objects/classes that already have these SMFs implemented, then there's no need to reimplement this logic!

```
class a_string_with_an_id() {
    public:
        /// getter and setter methods for our private variables
    private:
        int id;
        std::string str;
}
```

Rule of Zero

If you don't need a constructor or a destructor or copy assignment etc. Then simply don't use it!

If your class relies on objects/classes that already have these SMFs implemented, then there's no need to reimplement this logic!

```
class a_string_with_an_id() {
    public:
        /// getter and setter methods for our private variables
    private:
        int id;
        std::string str;
}
```

Our class a_string_with_an_id has self managing variables.

Rule of Zero

If you don’t need a constructor or a destructor or copy assignment etc. Then simply don’t use it!

If your class relies on objects/classes that already have these SMFs implemented, then there’s no need to reimplement this logic!

```
class a_string_with_an_id() {
    public:
        /// getter and setter methods for our private variables
    private:
        int id;
        std::string str;
}
```

`std::string` ***already*** has copy constructor, copy assignment, move constructor, and move assignment!

Rule of Three

If you need a custom destructor, then you also probably ***need*** to define a copy constructor and a copy assignment operator for your class

Rule of Three

If you need a custom destructor, then you also probably ***need*** to define a copy constructor and a copy assignment operator for your class

Why is this the case?

If you use a destructor, that often means that you are manually dealing with dynamic memory allocation/are generally just handling your own memory.

Rule of Three

If you need a custom destructor, then you also probably ***need*** to define a copy constructor and a copy assignment operator for your class

Why is this the case?

If you use a destructor, that often means that you are manually dealing with dynamic memory allocation/are generally just handling your own memory.

If this is the case:

The compiler will not be able to automatically generate these for you, because of the manual memory management.

Rule of Five

If you define a copy constructor or copy assignment operator, then you ***should*** define a move constructor and a move assignment operator as well.

Rule of Five

If you define a copy constructor or copy assignment operator, then you ***should*** define a move constructor and a move assignment operator as well.

Why?

Copies = Slow

This is less about correctness, unlike the rule of three, and more about efficiency.

What questions do we have?