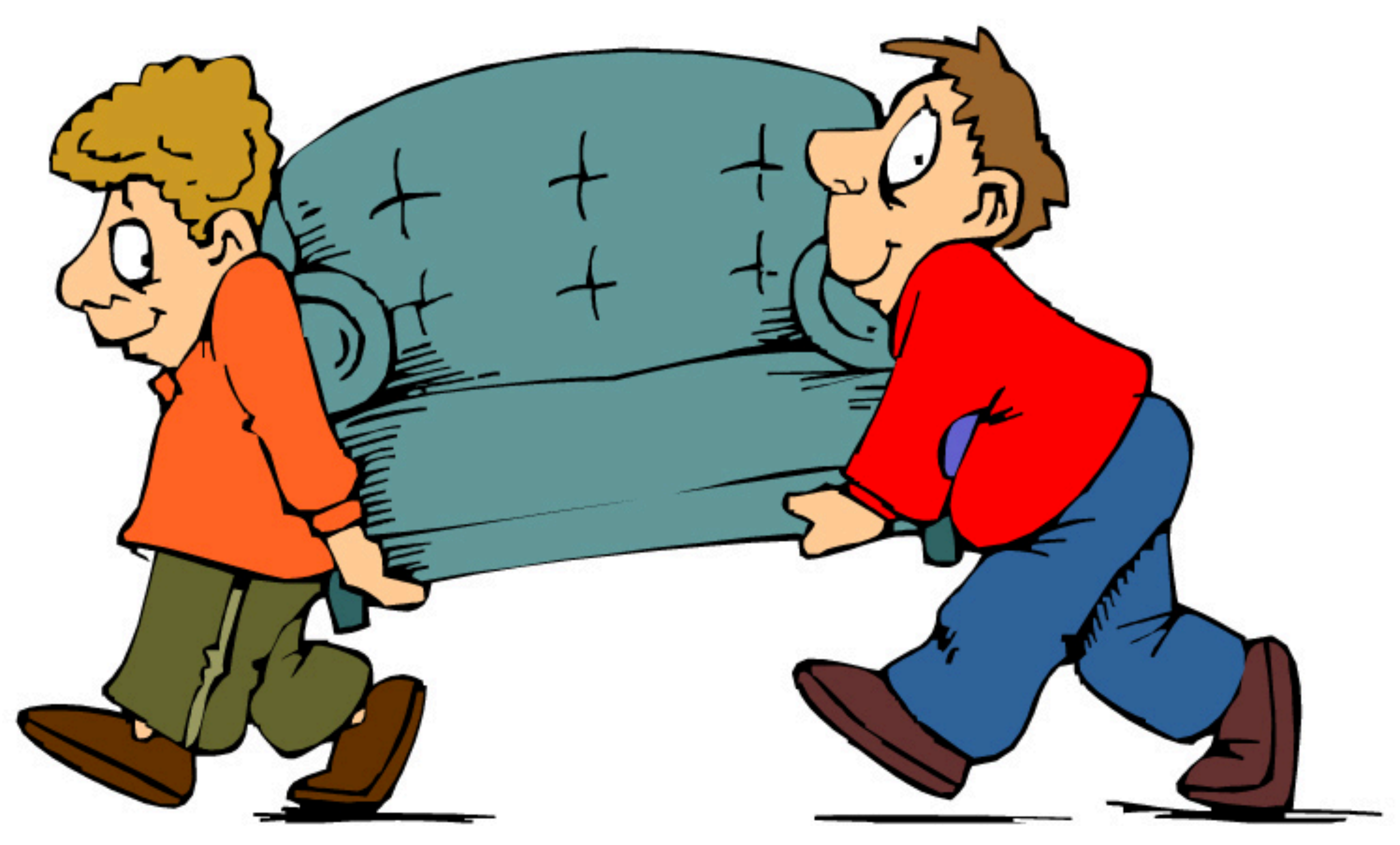
Modern C++ Explained : Move Semantics

by Olve Maudal



February 5, 2018

In this session I will try to explain **rvalue references** (aka **refref)** and **move semantics**. Here is a code snippet to get us started…

```
#include <iostream>
#include <string>

void analyze(std::string seq)
{
    std::cout << seq << std::endl;
    // ...
}

int main()
{
    std::string seq = "ACTTCTGTATTGGGTCTTTAATAG";
    analyze(seq);
}
```

In this session I will try to explain **rvalue references** (aka **refref)** and **move semantics**. Here is a code snippet to get us started…

```
#include <iostream>
#include <string>

void analyze(std::string seq)
{
    std::cout << seq << std::endl;
    // ...
}

int main()
{
    std::string seq = "ACTTCTGTATTGGGTCTTTAATAG";
    analyze(seq);
}
```

In this session I will try to explain **rvalue references** (aka **refref)** and **move semantics**. Here is a code snippet to get us started…

```
$ c++ tour.cpp && ./a.out
ACTTCTGTATTGGGTCTTTAATAG
$
```

```
#include <iostream>
#include <string>

void analyze(std::string seq)
{
    std::cout << seq << std::endl;
    // ...
}

int main()
{
    std::string seq = "ACTTCTGTATTGGGTCTTTAATAG";
    analyze(seq);
}
```

In this session I will try to explain **rvalue references** (aka **refref)** and **move semantics**. Here is a code snippet to get us started…

We have a sequence of letters and send it off to a function to be analyzed. In this case, we pass the string **by value** and the function will be working on a **copy** of the string.

```
$ c++ tour.cpp && ./a.out
ACTTCTGTATTGGGTCTTTAATAG
$
```

```
#include <iostream>
#include <string>

void analyze(std::string seq)
{
    std::cout << seq << std::endl;
    // ...
}

int main()
{
    std::string seq = "ACTTCTGTATTGGGTCTTTAATAG";
    analyze(seq);
}
```

In this session I will try to explain **rvalue references** (aka **refref)** and **move semantics**. Here is a code snippet to get us started…

We have a sequence of letters and send it off to a function to be analyzed. In this case, we pass the string **by value** and the function will be working on a **copy** of the string.

For large objects, taking a copy might be inefficient, so passing the object by a **reference to a constant** (aka **const ref**) is often recommend.

```
$ c++ tour.cpp && ./a.out
ACTTCTGTATTGGGTCTTTAATAG
$
```

```cpp
#include <iostream>
#include <string>

void analyze(std::string seq)
{
    std::cout << seq << std::endl;
    // ...
}

int main()
{
    std::string seq = "ACTTCTGTATTGGGTCTTTAATAG";
    analyze(seq);
}
```

```cpp
#include <iostream>
#include <string>

void analyze(std::string seq)
{
    std::cout << seq << std::endl;
    // ...
}

int main()
{
    std::string seq = "ACTTCTGTATTGGGTCTTTAATAG";
    analyze(seq);
}
```

```cpp
#include <iostream>
#include <string>

void analyze(const std::string & seq)
{
    std::cout << seq << std::endl;
    // ...
}

int main()
{
    std::string seq = "ACTTCTGTATTGGGTCTTTAATAG";
    analyze(seq);
}
```

```cpp
#include <iostream>
#include <string>

void analyze(const std::string & seq)
{
    std::cout << seq << std::endl;
    // ...
}

int main()
{
    std::string seq = "ACTTCTGTATTGGGTCTTTAATAG";
    analyze(seq);
}
```

```
#include <iostream>
#include <string>

void analyze(const std::string & seq)
{
    std::cout << seq << std::endl;
    // ...
}

int main()
{
    std::string seq = "ACTTCTGTATTGGGTCTTTAATAG";
    analyze(seq);
}
```

And this is probably fine, until the `analyze()` function needs to modify the sequence. For example…

```
#include <iostream>
#include <string>

void analyze(const std::string & seq)
{
    std::cout << seq << std::endl;
    // ...
}

int main()
{
    std::string seq = "ACTTCTGTATTGGGTCTTTAATAG";
    analyze(seq);
}
```

```cpp
#include <iostream>
#include <string>

void analyze(const std::string & seq)
{
    std::cout << seq << std::endl;
    // ...
}

int main()
{
    std::string seq = "ACTTCTGTATTGGGTCTTTAATAG";
    analyze(seq);
}
```

```cpp
#include <iostream>
#include <string>

void analyze(const std::string & seq)
{
    std::cout << seq << std::endl;
    // ...
    std::string seq2(seq);
    if (std::size_t pos = seq2.find("CTG"); pos != seq2.npos)
        seq2.replace(pos, 3, "...");
    std::cout << seq2 << std::endl;
    // ...
}

int main()
{
    std::string seq = "ACTTCTGTATTGGGTCTTTAATAG";
    analyze(seq);
}
```

```cpp
#include <iostream>
#include <string>

void analyze(const std::string & seq)
{
    std::cout << seq << std::endl;
    // ...
    std::string seq2(seq);
    if (std::size_t pos = seq2.find("CTG"); pos != seq2.npos)
        seq2.replace(pos, 3, "...");
    std::cout << seq2 << std::endl;
    // ...
}

int main()
{
    std::string seq = "ACTTCTGTATTGGGTCTTTAATAG";
    analyze(seq);
}
```

```
$ c++ tour.cpp && ./a.out
ACTTCTGTATTGGGTCTTTAATAG
ACTT...TATTGGGTCTTTAATAG
$
```

```cpp
#include <iostream>
#include <string>

void analyze(const std::string & seq)
{
    std::cout << seq << std::endl;
    // ...
    std::string seq2(seq);
    if (std::size_t pos = seq2.find("CTG"); pos != seq2.npos)
        seq2.replace(pos, 3, "...");
    std::cout << seq2 << std::endl;
    // ...
}

int main()
{
    std::string seq = "ACTTCTGTATTGGGTCTTTAATAG";
    analyze(seq);
}
```

Here we take a copy of the sequence and then change the copy. Creating a copy could be expensive for large objects.

```
$ c++ tour.cpp && ./a.out
ACTTCTGTATTGGGTCTTTAATAG
ACTT...TATTGGGTCTTTAATAG
$
```

```
#include <iostream>
#include <string>

void analyze(const std::string & seq)
{
    std::cout << seq << std::endl;
    // ...
    std::string seq2(seq);
    if (std::size_t pos = seq2.find("CTG"); pos != seq2.npos)
        seq2.replace(pos, 3, "...");
    std::cout << seq2 << std::endl;
    // ...
}

int main()
{
    std::string seq = "ACTTCTGTATTGGGTCTTTAATAG";
    analyze(seq);
}
```

Here we take a copy of the sequence and then change the copy. Creating a copy could be expensive for large objects.

But what if the caller is fine with the idea of letting the `analyze()` function do whatever it wants with the object we pass to it?

```
$ c++ tour.cpp && ./a.out
ACTTCTGTATTGGGTCTTTAATAG
ACTT...TATTGGGTCTTTAATAG
$
```

```
#include <iostream>
#include <string>

void analyze(const std::string & seq)
{
    std::cout << seq << std::endl;
    // ...
    std::string seq2(seq);
    if (std::size_t pos = seq2.find("CTG"); pos != seq2.npos)
        seq2.replace(pos, 3, "...");
    std::cout << seq2 << std::endl;
    // ...
}

int main()
{
    std::string seq = "ACTTCTGTATTGGGTCTTTAATAG";
    analyze(seq);
}
```

Here we take a copy of the sequence and then change the copy. Creating a copy could be expensive for large objects.

But what if the caller is fine with the idea of letting the `analyze()` function do whatever it wants with the object we pass to it?

Then, in this case, taking the argument as a **non-const ref** might be a better solution.

```
$ c++ tour.cpp && ./a.out
ACTTCTGTATTGGGTCTTTAATAG
ACTT...TATTGGGTCTTTAATAG
$
```

```
#include <iostream>
#include <string>

void analyze(const std::string & seq)
{
    std::cout << seq << std::endl;
    // ...
    std::string seq2(seq);
    if (std::size_t pos = seq2.find("CTG"); pos != seq2.npos)
        seq2.replace(pos, 3, "...");
    std::cout << seq2 << std::endl;
    // ...
}

int main()
{
    std::string seq = "ACTTCTGTATTGGGTCTTTAATAG";
    analyze(seq);
}
```

```
#include <iostream>
#include <string>

void analyze(const std::string & seq)
{
    std::cout << seq << std::endl;
    // ...
    std::string seq2(seq);
    if (std::size_t pos = seq2.find("CTG"); pos != seq2.npos)
        seq2.replace(pos, 3, "...");
    std::cout << seq2 << std::endl;
    // ...
}

int main()
{
    std::string seq = "ACTTCTGTATTGGGTCTTTAATAG";
    analyze(seq);
}
```

```
#include <iostream>
#include <string>

void analyze(std::string & seq)
{
    std::cout << seq << std::endl;
    // ...
    std::string seq2(seq);
    if (std::size_t pos = seq2.find("CTG"); pos != seq2.npos)
        seq2.replace(pos, 3, "...");
    std::cout << seq2 << std::endl;
    // ...
}

int main()
{
    std::string seq = "ACTTCTGTATTGGGTCTTTAATAG";
    analyze(seq);
}
```

```cpp
#include <iostream>
#include <string>

void analyze(std::string & seq)
{
    std::cout << seq << std::endl;
    // ...
    std::string seq2(seq);
    if (std::size_t pos = seq2.find("CTG"); pos != seq2.npos)
        seq2.replace(pos, 3, "...");
    std::cout << seq2 << std::endl;
    // ...
}

int main()
{
    std::string seq = "ACTTCTGTATTGGGTCTTTAATAG";
    analyze(seq);
}
```

```
#include <iostream>
#include <string>

void analyze(std::string & seq)
{
    std::cout << seq << std::endl;
    // ...
    std::string seq2(seq);
    if (std::size_t pos = seq2.find("CTG"); pos != seq2.npos)
        seq2.replace(pos, 3, "...");
    std::cout << seq2 << std::endl;
    // ...
}

int main()
{
    std::string seq = "ACTTCTGTATTGGGTCTTTAATAG";
    analyze(seq);
}
```

```
#include <iostream>
#include <string>

void analyze(std::string & seq)
{
    std::cout << seq << std::endl;
    // ...
    std::string seq2(seq);
    if (std::size_t pos = seq2.find("CTG"); pos != seq2.npos)
        seq2.replace(pos, 3, "...");
    std::cout << seq2 << std::endl;
    // ...
}

int main()
{
    std::string seq = "ACTTCTGTATTGGGTCTTTAATAG";
    analyze(seq);
}
```

```cpp
#include <iostream>
#include <string>

void analyze(std::string & seq)
{
    std::cout << seq << std::endl;
    // ...

    if (std::size_t pos = seq.find("CTG"); pos != seq.npos)
        seq.replace(pos, 3, "...");
    std::cout << seq << std::endl;
    // ...
}

int main()
{
    std::string seq = "ACTTCTGTATTGGGTCTTTAATAG";
    analyze(seq);
}
```

```cpp
#include <iostream>
#include <string>

void analyze(std::string & seq)
{
    std::cout << seq << std::endl;
    // ...

    if (std::size_t pos = seq.find("CTG"); pos != seq.npos)
        seq.replace(pos, 3, "...");
    std::cout << seq << std::endl;
    // ...
}

int main()
{
    std::string seq = "ACTTCTGTATTGGGTCTTTAATAG";
    analyze(seq);
}
```

```cpp
#include <iostream>
#include <string>

void analyze(std::string & seq)
{
    std::cout << seq << std::endl;
    // ...
    if (std::size_t pos = seq.find("CTG"); pos != seq.npos)
        seq.replace(pos, 3, "...");
    std::cout << seq << std::endl;
    // ...
}

int main()
{
    std::string seq = "ACTTCTGTATTGGGTCTTTAATAG";
    analyze(seq);
}
```

```
#include <iostream>
#include <string>

void analyze(std::string & seq)
{
    std::cout << seq << std::endl;
    // ...
    if (std::size_t pos = seq.find("CTG"); pos != seq.npos)
        seq.replace(pos, 3, "...");
    std::cout << seq << std::endl;
    // ...
}

int main()
{
    std::string seq = "ACTTCTGTATTGGGTCTTTAATAG";
    analyze(seq);
}
```

Of course, we should be sceptical about functions that modifies its arguments like this - but in this case, for very big data structures, it might be exactly the design solution for the problem we are trying to solve.

```cpp
#include <iostream>
#include <string>

void analyze(std::string & seq)
{
    std::cout << seq << std::endl;
    // ...
    if (std::size_t pos = seq.find("CTG"); pos != seq.npos)
        seq.replace(pos, 3, "...");
    std::cout << seq << std::endl;
    // ...
}

int main()
{
    std::string seq = "ACTTCTGTATTGGGTCTTTAATAG";
    analyze(seq);
}
```

Of course, we should be sceptical about functions that modifies its arguments like this - but in this case, for very big data structures, it might be exactly the design solution for the problem we are trying to solve.

However…

```cpp
#include <iostream>
#include <string>

void analyze(std::string & seq)
{
    std::cout << seq << std::endl;
    // ...
    if (std::size_t pos = seq.find("CTG"); pos != seq.npos)
        seq.replace(pos, 3, "...");
    std::cout << seq << std::endl;
    // ...
}
```

```cpp
int main()
{
    std::string seq = "ACTTCTGTATTGGGTCTTTAATAG";
    analyze(seq);
}
```

```
#include <iostream>
#include <string>

void analyze(std::string & seq)
{
    std::cout << seq << std::endl;
    // ...
    if (std::size_t pos = seq.find("CTG"); pos != seq.npos)
        seq.replace(pos, 3, "...");
    std::cout << seq << std::endl;
    // ...
}
```

```
int main()
{
    std::string seq = "ACTTCTGTATTGGGTCTTTAATAG";
    analyze(seq);
}
```

```
#include <iostream>
#include <string>

void analyze(std::string & seq)
{
    std::cout << seq << std::endl;
    // ...
    if (std::size_t pos = seq.find("CTG"); pos != seq.npos)
        seq.replace(pos, 3, "...");
    std::cout << seq << std::endl;
    // ...
}

int main()
{
    std::string seq = "ACTTCTGTATTGGGTCTTTAATAG";
    analyze(seq);
}
```

```
#include <iostream>
#include <string>

void analyze(std::string & seq)
{
    std::cout << seq << std::endl;
    // ...
    if (std::size_t pos = seq.find("CTG"); pos != seq.npos)
        seq.replace(pos, 3, "...");
    std::cout << seq << std::endl;
    // ...
}

std::string extract()
{
    // ...
    return "ACTTCTGTATTGGGTCTTTAATAG";
}

int main()
{
    std::string seq = extract();
    analyze(seq);
}
```

```cpp
#include <iostream>
#include <string>

void analyze(std::string & seq)
{
    std::cout << seq << std::endl;
    // ...
    if (std::size_t pos = seq.find("CTG"); pos != seq.npos)
        seq.replace(pos, 3, "...");
    std::cout << seq << std::endl;
    // ...
}

std::string extract()
{
    // ...
    return "ACTTCTGTATTGGGTCTTTAATAG";
}

int main()
{
    std::string seq = extract();
    analyze(seq);
}
```

```cpp
#include <iostream>
#include <string>

void analyze(std::string & seq)
{
    std::cout << seq << std::endl;
    // ...
    if (std::size_t pos = seq.find("CTG"); pos != seq.npos)
        seq.replace(pos, 3, "...");
    std::cout << seq << std::endl;
    // ...
}

std::string extract()
{
    // ...
    return "ACTTCTGTATTGGGTCTTTAATAG";
}

int main()
{
    std::string seq = extract();
    analyze(seq);
}
```

```
$ c++ tour.cpp && ./a.out
ACTTCTGTATTGGGTCTTTAATAG
ACTT...TATTGGGTCTTTAATAG
$
```

```cpp
#include <iostream>
#include <string>

void analyze(std::string & seq)
{
    std::cout << seq << std::endl;
    // ...
    if (std::size_t pos = seq.find("CTG"); pos != seq.npos)
        seq.replace(pos, 3, "...");
    std::cout << seq << std::endl;
    // ...
}

std::string extract()
{
    // ...
    return "ACTTCTGTATTGGGTCTTTAATAG";
}

int main()
{
    std::string seq = extract();
    analyze(seq);
}
```

```cpp
#include <iostream>
#include <string>

void analyze(std::string & seq)
{
    std::cout << seq << std::endl;
    // ...
    if (std::size_t pos = seq.find("CTG"); pos != seq.npos)
        seq.replace(pos, 3, "...");
    std::cout << seq << std::endl;
    // ...
}

std::string extract()
{
    // ...
    return "ACTTCTGTATTGGGTCTTTAATAG";
}

int main()
{
    std::string seq = extract();
    analyze(seq);
}
```

```cpp
#include <iostream>
#include <string>

void analyze(std::string & seq)
{
    std::cout << seq << std::endl;
    // ...
    if (std::size_t pos = seq.find("CTG"); pos != seq.npos)
        seq.replace(pos, 3, "...");
    std::cout << seq << std::endl;
    // ...
}

std::string extract()
{
    // ...
    return "ACTTCTGTATTGGGTCTTTAATAG";
}

int main()
{
    analyze(extract());
}
```

```cpp
#include <iostream>
#include <string>

void analyze(std::string & seq)
{
    std::cout << seq << std::endl;
    // ...
    if (std::size_t pos = seq.find("CTG"); pos != seq.npos)
        seq.replace(pos, 3, "...");
    std::cout << seq << std::endl;
    // ...
}

std::string extract()
{
    // ...
    return "ACTTCTGTATTGGGTCTTTAATAG";
}

int main()
{
    analyze(extract());
}
```

```
#include <iostream>
#include <string>

void analyze(std::string & seq)
{
    std::cout << seq << std::endl;
    // ...
    if (std::size_t pos = seq.find("CTG"); pos != seq.npos)
        seq.replace(pos, 3, "...");
    std::cout << seq << std::endl;
    // ...
}

std::string extract()
{
    // ...
    return "ACTTCTGTATTGGGTCTTTAATAG";
}

int main()
{
    analyze(extract());
}
```

Now we get an error.

```
#include <iostream>
#include <string>

void analyze(std::string & seq)
{
    std::cout << seq << std::endl;
    // ...
    if (std::size_t pos = seq.find("CTG"); pos != seq.npos)
        seq.replace(pos, 3, "...");
    std::cout << seq << std::endl;
    // ...
}

std::string extract()
{
    // ...
    return "ACTTCTGTATTGGGTCTTTAATAG";
}

int main()
{
    analyze(extract());
}
```

Now we get an error.

```
$ c++ tour.cpp && ./a.out
error: no matching function for call to 'analyze'
candidate function expects an l-value
$
```

```
#include <iostream>
#include <string>

void analyze(std::string & seq)
{
    std::cout << seq << std::endl;
    // ...
    if (std::size_t pos = seq.find("CTG"); pos != seq.npos)
        seq.replace(pos, 3, "...");
    std::cout << seq << std::endl;
    // ...
}

std::string extract()
{
    // ...
    return "ACTTCTGTATTGGGTCTTTAATAG";
}

int main()
{
    analyze(extract());
}
```

Now we get an error.

Because we try to pass an unnamed value to a function that needs to “borrow” the object.

```
$ c++ tour.cpp && ./a.out
error: no matching function for call to 'analyze'
candidate function expects an l-value
$
```

```
#include <iostream>
#include <string>

void analyze(std::string & seq)
{
    std::cout << seq << std::endl;
    // ...
    if (std::size_t pos = seq.find("CTG"); pos != seq.npos)
        seq.replace(pos, 3, "...");
    std::cout << seq << std::endl;
    // ...
}

std::string extract()
{
    // ...
    return "ACTTCTGTATTGGGTCTTTAATAG";
}

int main()
{
    analyze(extract());
}
```

Now we get an error.

Because we try to pass an unnamed value to a function that needs to "borrow" the object.

A slightly better (but still imprecise*) way of saying this is that the `analyze()` function expects an **lvalue** (something that can appear on the left hand side of an assignment), while we are trying to pass it an **rvalue** (something that needs to be on the right hand side of an assignment statement).

(*) the legalese for this includes discussions about five, partly overlapping value categories: glvalues, prvalues, xvalues, lvalues and rvalues... a nice topic for later presentations

```
$ c++ tour.cpp && ./a.out
error: no matching function for call to 'analyze'
candidate function expects an l-value
$
```

```
#include <iostream>
#include <string>

void analyze(std::string & seq)
{
    std::cout << seq << std::endl;
    // ...
    if (std::size_t pos = seq.find("CTG"); pos != seq.npos)
        seq.replace(pos, 3, "...");
    std::cout << seq << std::endl;
    // ...
}

std::string extract()
{
    // ...
    return "ACTTCTGTATTGGGTCTTTAATAG";
}

int main()
{
    analyze(extract());
}
```

Now we get an error.

Because we try to pass an unnamed value to a function that needs to "borrow" the object.

A slightly better (but still imprecise*) way of saying this is that the `analyze()` function expects an **lvalue** (something that can appear on the left hand side of an assignment), while we are trying to pass it an **rvalue** (something that needs to be on the right hand side of an assignment statement).

(*) the legalese for this includes discussions about five, partly overlapping value categories: glvalues, prvalues, xvalues, lvalues and rvalues... a nice topic for later presentations

Is there a way to let the `analyze()` function take an **rvalue**?

```
$ c++ tour.cpp && ./a.out
error: no matching function for call to 'analyze'
candidate function expects an l-value
$
```

```
#include <iostream>
#include <string>

void analyze(std::string & seq)
{
    std::cout << seq << std::endl;
    // ...
    if (std::size_t pos = seq.find("CTG"); pos != seq.npos)
        seq.replace(pos, 3, "...");
    std::cout << seq << std::endl;
    // ...
}

std::string extract()
{
    // ...
    return "ACTTCTGTATTGGGTCTTTAATAG";
}

int main()
{
    analyze(extract());
}
```

Now we get an error.

Because we try to pass an unnamed value to a function that needs to "borrow" the object.

A slightly better (but still imprecise*) way of saying this is that the `analyze()` function expects an **lvalue** (something that can appear on the left hand side of an assignment), while we are trying to pass it an **rvalue** (something that needs to be on the right hand side of an assignment statement).

(*) the legalese for this includes discussions about five, partly overlapping value categories: glvalues, prvalues, xvalues, lvalues and rvalues... a nice topic for later presentations

Is there a way to let the `analyze()` function take an **rvalue**?

Yes! And that feature was introduced in **C++11**

```
$ c++ tour.cpp && ./a.out
error: no matching function for call to 'analyze'
candidate function expects an l-value
$
```

```cpp
#include <iostream>
#include <string>

void analyze(std::string & seq)
{
    std::cout << seq << std::endl;
    // ...
    if (std::size_t pos = seq.find("CTG"); pos != seq.npos)
        seq.replace(pos, 3, "...");
    std::cout << seq << std::endl;
    // ...
}

std::string extract()
{
    // ...
    return "ACTTCTGTATTGGGTCTTTAATAG";
}

int main()
{
    analyze(extract());
}
```

```
#include <iostream>
#include <string>

void analyze(std::string & seq)
{
    std::cout << seq << std::endl;
    // ...
    if (std::size_t pos = seq.find("CTG"); pos != seq.npos)
        seq.replace(pos, 3, "...");
    std::cout << seq << std::endl;
    // ...
}

std::string extract()
{
    // ...
    return "ACTTCTGTATTGGGTCTTTAATAG";
}

int main()
{
    analyze(extract());
}
```

```
#include <iostream>
#include <string>

void analyze(std::string & seq)
{
    std::cout << seq << std::endl;
    // ...
    if (std::size_t pos = seq.find("CTG"); pos != seq.npos)
        seq.replace(pos, 3, "...");
    std::cout << seq << std::endl;
    // ...
}

std::string extract()
{
    // ...
    return "ACTTCTGTATTGGGTCTTTAATAG";
}

int main()
{
    analyze(extract());
}
```

```cpp
#include <iostream>
#include <string>

void analyze(std::string && seq)
{
    std::cout << seq << std::endl;
    // ...
    if (std::size_t pos = seq.find("CTG"); pos != seq.npos)
        seq.replace(pos, 3, "...");
    std::cout << seq << std::endl;
    // ...
}

std::string extract()
{
    // ...
    return "ACTTCTGTATTGGGTCTTTAATAG";
}

int main()
{
    analyze(extract());
}
```

```cpp
#include <iostream>
#include <string>

void analyze(std::string && seq)
{
    std::cout << seq << std::endl;
    // ...
    if (std::size_t pos = seq.find("CTG"); pos != seq.npos)
        seq.replace(pos, 3, "...");
    std::cout << seq << std::endl;
    // ...
}

std::string extract()
{
    // ...
    return "ACTTCTGTATTGGGTCTTTAATAG";
}

int main()
{
    analyze(extract());
}
```

```
$ c++ tour.cpp && ./a.out
ACTTCTGTATTGGGTCTTTAATAG
ACTT...TATTGGGTCTTTAATAG
$
```

```cpp
#include <iostream>
#include <string>

void analyze(std::string && seq)
{
    std::cout << seq << std::endl;
    // ...
    if (std::size_t pos = seq.find("CTG"); pos != seq.npos)
        seq.replace(pos, 3, "...");
    std::cout << seq << std::endl;
    // ...
}

std::string extract()
{
    // ...
    return "ACTTCTGTATTGGGTCTTTAATAG";
}

int main()
{
    analyze(extract());
}
```

So now we have an `analyze()` function that can take reference to a so called **rvalue**.

```
$ c++ tour.cpp && ./a.out
ACTTCTGTATTGGGTCTTTAATAG
ACTT...TATTGGGTCTTTAATAG
$
```

```
#include <iostream>
#include <string>

void analyze(std::string && seq)
{
    std::cout << seq << std::endl;
    // ...
    if (std::size_t pos = seq.find("CTG"); pos != seq.npos)
        seq.replace(pos, 3, "...");
    std::cout << seq << std::endl;
    // ...
}

std::string extract()
{
    // ...
    return "ACTTCTGTATTGGGTCTTTAATAG";
}

int main()
{
    analyze(extract());
}
```

So now we have an `analyze()` function that can take reference to a so called **rvalue**.

However, can this same function also take **lvalues**?

```
$ c++ tour.cpp && ./a.out
ACTTCTGTATTGGGTCTTTAATAG
ACTT...TATTGGGTCTTTAATAG
$
```

```
#include <iostream>
#include <string>

void analyze(std::string && seq)
{
    std::cout << seq << std::endl;
    // ...
    if (std::size_t pos = seq.find("CTG"); pos != seq.npos)
        seq.replace(pos, 3, "...");
    std::cout << seq << std::endl;
    // ...
}

std::string extract()
{
    // ...
    return "ACTTCTGTATTGGGTCTTTAATAG";
}

int main()
{
    analyze(extract());
}
```

So now we have an `analyze()` function that can take reference to a so called **rvalue**.

However, can this same function also take **lvalues**?

The answer is NO.

```
$ c++ tour.cpp && ./a.out
ACTTCTGTATTGGGTCTTTAATAG
ACTT...TATTGGGTCTTTAATAG
$
```

```cpp
#include <iostream>
#include <string>

void analyze(std::string && seq)
{
    std::cout << seq << std::endl;
    // ...
    if (std::size_t pos = seq.find("CTG"); pos != seq.npos)
        seq.replace(pos, 3, "...");
    std::cout << seq << std::endl;
    // ...
}

std::string extract()
{
    // ...
    return "ACTTCTGTATTGGGTCTTTAATAG";
}

int main()
{
    analyze(extract());
}
```

So now we have an `analyze()` function that can take reference to a so called **rvalue**.

However, can this same function also take **lvalues**?

The answer is NO.

Let's demonstrate...

```
$ c++ tour.cpp && ./a.out
ACTTCTGTATTGGGTCTTTAATAG
ACTT...TATTGGGTCTTTAATAG
$
```

```cpp
#include <iostream>
#include <string>

void analyze(std::string && seq)
{
    std::cout << seq << std::endl;
    // ...
    if (std::size_t pos = seq.find("CTG"); pos != seq.npos)
        seq.replace(pos, 3, "...");
    std::cout << seq << std::endl;
    // ...
}

std::string extract()
{
    // ...
    return "ACTTCTGTATTGGGTCTTTAATAG";
}

int main()
{
    analyze(extract());
}
```

```cpp
#include <iostream>
#include <string>

void analyze(std::string && seq)
{
    std::cout << seq << std::endl;
    // ...
    if (std::size_t pos = seq.find("CTG"); pos != seq.npos)
        seq.replace(pos, 3, "...");
    std::cout << seq << std::endl;
    // ...
}

std::string extract()
{
    // ...
    return "ACTTCTGTATTGGGTCTTTAATAG";
}

int main()
{
    analyze(extract());
}
```

```cpp
#include <iostream>
#include <string>

void analyze(std::string && seq)
{
    std::cout << seq << std::endl;
    // ...
    if (std::size_t pos = seq.find("CTG"); pos != seq.npos)
        seq.replace(pos, 3, "...");
    std::cout << seq << std::endl;
    // ...
}

std::string extract()
{
    // ...
    return "ACTTCTGTATTGGGTCTTTAATAG";
}

int main()
{
    std::string seq = extract();
    analyze(seq);
}
```

```cpp
#include <iostream>
#include <string>

void analyze(std::string && seq)
{
    std::cout << seq << std::endl;
    // ...
    if (std::size_t pos = seq.find("CTG"); pos != seq.npos)
        seq.replace(pos, 3, "...");
    std::cout << seq << std::endl;
    // ...
}

std::string extract()
{
    // ...
    return "ACTTCTGTATTGGGTCTTTAATAG";
}

int main()
{
    std::string seq = extract();
    analyze(seq);
}
```

```
$ c++ tour.cpp && ./a.out
error: no matching function for call to ‘analyze’
no known conversion from ‘std::string' to 'std::string &&’
$
```

```cpp
#include <iostream>
#include <string>

void analyze(std::string && seq)
{
    std::cout << seq << std::endl;
    // ...
    if (std::size_t pos = seq.find("CTG"); pos != seq.npos)
        seq.replace(pos, 3, "...");
    std::cout << seq << std::endl;
    // ...
}

std::string extract()
{
    // ...
    return "ACTTCTGTATTGGGTCTTTAATAG";
}

int main()
{
    std::string seq = extract();
    analyze(seq);
}
```

Now we get an error because the function expected an **rvalue** but we try to call it with an **lvalue**.

```
$ c++ tour.cpp && ./a.out
error: no matching function for call to ‘analyze’
no known conversion from ‘std::string' to 'std::string &&’
$
```

```cpp
#include <iostream>
#include <string>

void analyze(std::string && seq)
{
    std::cout << seq << std::endl;
    // ...
    if (std::size_t pos = seq.find("CTG"); pos != seq.npos)
        seq.replace(pos, 3, "...");
    std::cout << seq << std::endl;
    // ...
}

std::string extract()
{
    // ...
    return "ACTTCTGTATTGGGTCTTTAATAG";
}

int main()
{
    std::string seq = extract();
    analyze(seq);
}
```

```cpp
#include <iostream>
#include <string>

void analyze(std::string && seq)
{
    std::cout << seq << std::endl;
    // ...
    if (std::size_t pos = seq.find("CTG"); pos != seq.npos)
        seq.replace(pos, 3, "...");
    std::cout << seq << std::endl;
    // ...
}

std::string extract()
{
    // ...
    return "ACTTCTGTATTGGGTCTTTAATAG";
}

int main()
{
    std::string seq = extract();
    analyze(seq);
}
```

It is possible to cast an **lvalue** into a **rvalue** by using `std::move()`

```cpp
#include <iostream>
#include <string>

void analyze(std::string && seq)
{
    std::cout << seq << std::endl;
    // ...
    if (std::size_t pos = seq.find("CTG"); pos != seq.npos)
        seq.replace(pos, 3, "...");
    std::cout << seq << std::endl;
    // ...
}

std::string extract()
{
    // ...
    return "ACTTCTGTATTGGGTCTTTAATAG";
}

int main()
{
    std::string seq = extract();
    analyze(seq);
}
```

It is possible to cast an **lvalue** into a **rvalue** by using `std::move()`

```cpp
#include <iostream>
#include <string>

void analyze(std::string && seq)
{
    std::cout << seq << std::endl;
    // ...
    if (std::size_t pos = seq.find("CTG"); pos != seq.npos)
        seq.replace(pos, 3, "...");
    std::cout << seq << std::endl;
    // ...
}

std::string extract()
{
    // ...
    return "ACTTCTGTATTGGGTCTTTAATAG";
}

int main()
{
    std::string seq = extract();
    analyze(seq);
}
```

It is possible to cast an **lvalue** into a **rvalue** by using `std::move()`

```
#include <iostream>
#include <string>

void analyze(std::string && seq)
{
    std::cout << seq << std::endl;
    // ...
    if (std::size_t pos = seq.find("CTG"); pos != seq.npos)
        seq.replace(pos, 3, "...");
    std::cout << seq << std::endl;
    // ...
}

std::string extract()
{
    // ...
    return "ACTTCTGTATTGGGTCTTTAATAG";
}

int main()
{
    std::string seq = extract();
    analyze(std::move(seq));
}
```

```cpp
#include <iostream>
#include <string>

void analyze(std::string && seq)
{
    std::cout << seq << std::endl;
    // ...
    if (std::size_t pos = seq.find("CTG"); pos != seq.npos)
        seq.replace(pos, 3, "...");
    std::cout << seq << std::endl;
    // ...
}

std::string extract()
{
    // ...
    return "ACTTCTGTATTGGGTCTTTAATAG";
}

int main()
{
    std::string seq = extract();
    analyze(std::move(seq));
}
```

```
$ c++ tour.cpp && ./a.out
ACTTCTGTATTGGGTCTTTAATAG
ACTT...TATTGGGTCTTTAATAG
$
```

```cpp
#include <iostream>
#include <string>

void analyze(std::string && seq)
{
    std::cout << seq << std::endl;
    // ...
    if (std::size_t pos = seq.find("CTG"); pos != seq.npos)
        seq.replace(pos, 3, "...");
    std::cout << seq << std::endl;
    // ...
}

std::string extract()
{
    // ...
    return "ACTTCTGTATTGGGTCTTTAATAG";
}

int main()
{
    std::string seq = extract();
    analyze(std::move(seq));
}
```

The use of `std::move` here expresses the idea that "you can take the object. It's yours. I will clean up the mess, but I promise not to assume anything about the object after you are done."

```
$ c++ tour.cpp && ./a.out
ACTTCTGTATTGGGTCTTTAATAG
ACTT...TATTGGGTCTTTAATAG
$
```

```
#include <iostream>
#include <string>

void analyze(std::string && seq)
{
    std::cout << seq << std::endl;
    // ...
    if (std::size_t pos = seq.find("CTG"); pos != seq.npos)
        seq.replace(pos, 3, "...");
    std::cout << seq << std::endl;
    // ...
}

std::string extract()
{
    // ...
    return "ACTTCTGTATTGGGTCTTTAATAG";
}

int main()
{
    std::string seq = extract();
    analyze(std::move(seq));
}
```

The use of `std::move` here expresses the idea that "you can take the object. It's yours. I will clean up the mess, but I promise not to assume anything about the object after you are done."

Having said that… it is important to understand that there is nothing magical with `std::move`, it is just a simple cast that generates no code, but tells the compiler that it is ok to deal with the object (or more generally, the expression) as if it was an **rvalue**. (At some point I guess it was considered to call it `std::rvalue_cast` instead)

```
$ c++ tour.cpp && ./a.out
ACTTCTGTATTGGGTCTTTAATAG
ACTT...TATTGGGTCTTTAATAG
$
```

```
#include <iostream>
#include <string>

void analyze(std::string && seq)
{
    std::cout << seq << std::endl;
    // ...
    if (std::size_t pos = seq.find("CTG"); pos != seq.npos)
        seq.replace(pos, 3, "...");
    std::cout << seq << std::endl;
    // ...
}

std::string extract()
{
    // ...
    return "ACTTCTGTATTGGGTCTTTAATAG";
}

int main()
{
    std::string seq = extract();
    analyze(std::move(seq));
}
```

The use of `std::move` here expresses the idea that "you can take the object. It's yours. I will clean up the mess, but I promise not to assume anything about the object after you are done."

Having said that… it is important to understand that there is nothing magical with `std::move`, it is just a simple cast that generates no code, but tells the compiler that it is ok to deal with the object (or more generally, the expression) as if it was an **rvalue**. (At some point I guess it was considered to call it `std::rvalue_cast` instead)

When moving an object like this, the only thing you can do afterwards is to delete it or give it a new state or invoke member functions that do not have any assumption of its internal state. If you don't know exactly what you are doing, it is probably best to not touch the object at all, just make sure that the object eventually gets destroyed.

```
$ c++ tour.cpp && ./a.out
ACTTCTGTATTGGGTCTTTAATAG
ACTT...TATTGGGTCTTTAATAG
$
```

```cpp
#include <iostream>
#include <string>

void analyze(std::string && seq)
{
    std::cout << seq << std::endl;
    // ...
    if (std::size_t pos = seq.find("CTG"); pos != seq.npos)
        seq.replace(pos, 3, "...");
    std::cout << seq << std::endl;
    // ...
}

std::string extract()
{
    // ...
    return "ACTTCTGTATTGGGTCTTTAATAG";
}

int main()
{
    std::string seq = extract();
    analyze(std::move(seq));
}
```

```cpp
#include <iostream>
#include <string>

void analyze(std::string && seq)
{
    std::cout << seq << std::endl;
    // ...
    if (std::size_t pos = seq.find("CTG"); pos != seq.npos)
        seq.replace(pos, 3, "...");
    std::cout << seq << std::endl;
    // ...
}

std::string extract()
{
    // ...
    return "ACTTCTGTATTGGGTCTTTAATAG";
}

int main()
{
    std::string seq = extract();
    analyze(std::move(seq));
}
```

```
#include <iostream>
#include <string>

void analyze(std::string && seq)
{
    std::cout << seq << std::endl;
    // ...
    if (std::size_t pos = seq.find("CTG"); pos != seq.npos)
        seq.replace(pos, 3, "...");
    std::cout << seq << std::endl;
    // ...
}

std::string extract()
{
    // ...
    return "ACTTCTGTATTGGGTCTTTAATAG";
}

int main()
{
    std::string seq = extract();
    analyze(std::move(seq));
}
```

```cpp
#include <iostream>
#include <string>

void analyze(std::string && seq)
{
    std::cout << seq << std::endl;
    // ...
    if (std::size_t pos = seq.find("CTG"); pos != seq.npos)
        seq.replace(pos, 3, "...");
    std::cout << seq << std::endl;
    // ...
}

std::string extract()
{
    // ...
    return "ACTTCTGTATTGGGTCTTTAATAG";
}

int main()
{
    std::string seq = extract();
    analyze(std::move(seq));
}
```

```
#include <iostream>
#include <string>

void analyze(std::string && seq)
{
    std::cout << seq << std::endl;
    // ...
    if (std::size_t pos = seq.find("CTG"); pos != seq.npos)
        seq.replace(pos, 3, "...");
    std::cout << seq << std::endl;
    // ...
}

std::string extract()
{
    // ...
    return "ACTTCTGTATTGGGTCTTTAATAG";
}

int main()
{
    std::string seq = extract();
    analyze(std::move(seq));
}
```

```cpp
#include <iostream>
#include <string>

void analyze(std::string && seq)
{
    std::cout << seq << std::endl;
    // ...
}

std::string extract()
{
    // ...
    return "ACTTCTGTATTGGGTCTTTAATAG";
}

int main()
{
    std::string seq = extract();
    analyze(std::move(seq));
}
```

```
#include <iostream>
#include <string>

void analyze(std::string && seq)
{
    std::cout << seq << std::endl;
    // ...
}

std::string extract()
{
    // ...
    return "ACTTCTGTATTGGGTCTTTAATAG";
}

int main()
{
    std::string seq = extract();
    analyze(std::move(seq));
}
```

We have seen **rvalue reference** and **move semantics** from a users point of view. I will now show how to implement a class that supports move semantics.

```
#include <iostream>
#include <string>

void analyze(std::string && seq)
{
    std::cout << seq << std::endl;
    // ...
}

std::string extract()
{
    // ...
    return "ACTTCTGTATTGGGTCTTTAATAG";
}

int main()
{
    std::string seq = extract();
    analyze(std::move(seq));
}
```

We have seen **rvalue reference** and **move semantics** from a users point of view. I will now show how to implement a class that supports move semantics.

Let's replace `std::string` with a `class Contig` that can hold a sequence of these letters (representing the nucleobases adenine, guanine, thymine and cytosine).

```
#include <iostream>
#include <string>

void analyze(std::string && seq)
{
    std::cout << seq << std::endl;
    // ...
}

std::string extract()
{
    // ...
    return "ACTTCTGTATTGGGTCTTTAATAG";
}

int main()
{
    std::string seq = extract();
    analyze(std::move(seq));
}
```

```
#include <iostream>
#include <string>







void analyze(std::string && seq)
{
    std::cout << seq << std::endl;
    // ...
}

std::string extract()
{
    // ...
    return "ACTTCTGTATTGGGTCTTTAATAG";
}

int main()
{
    std::string seq = extract();
    analyze(std::move(seq));
}
```

```cpp
#include <iostream>
#include <algorithm>
#include <cstring>
#include <iterator>

class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
}

Contig extract()
{
    // ...
    return Contig("ACTTCTGTATTGGGTCTTTAATAG");
}

int main()
{
    Contig seq = extract();
    analyze(std::move(seq));
}
```

```cpp
#include <iostream>
#include <algorithm>
#include <cstring>
#include <iterator>

class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
}

Contig extract()
{
    // ...
    return Contig("ACTTCTGTATTGGGTCTTTAATAG");
}

int main()
{
    Contig seq = extract();
    analyze(std::move(seq));
}
```

```cpp
#include <algorithm>
#include <cstring>
#include <iostream>
#include <iterator>

class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
}

Contig extract()
{
    // ...
    return Contig("ACTTCTGTATTGGGTCTTTAATAG");
}

int main()
{
    Contig seq = extract();
    analyze(std::move(seq));
}
```

```
#include <algorithm>
#include <cstring>
#include <iostream>
#include <iterator>

class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
}

Contig extract()
{
    // ...
    return Contig("ACTTCTGTATTGGGTCTTTAATAG");
}

int main()
{
    Contig seq = extract();
    analyze(std::move(seq));
}
```

```cpp
#include <algorithm>
#include <cstring>
#include <iostream>
#include <iterator>

class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
}

Contig extract()
{
    // ...
    return Contig("ACTTCTGTATTGGGTCTTTAATAG");
}

int main()
{
    Contig seq = extract();
    analyze(std::move(seq));
}
```

```
$ c++ tour.cpp && ./a.out
ACTTCTGTATTGGGTCTTTAATAG
$
```

```cpp
#include <algorithm>
#include <cstring>
#include <iostream>
#include <iterator>

class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
}

Contig extract()
{
    // ...
    return Contig("ACTTCTGTATTGGGTCTTTAATAG");
}

int main()
{
    Contig seq = extract();
    analyze(std::move(seq));
}
```

Looks good, but we are not done yet. There is some essential stuff missing.

```
$ c++ tour.cpp && ./a.out
ACTTCTGTATTGGGTCTTTAATAG
$
```

```
#include <algorithm>
#include <cstring>
#include <iostream>
#include <iterator>

class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
}

Contig extract()
{
    // ...
    return Contig("ACTTCTGTATTGGGTCTTTAATAG");
}

int main()
{
    Contig seq = extract();
    analyze(std::move(seq));
}
```

Looks good, but we are not done yet. There is some essential stuff missing.

```
$ c++ tour.cpp && ./a.out
ACTTCTGTATTGGGTCTTTAATAG
$
```

```
#include <algorithm>
#include <cstring>
#include <iostream>
#include <iterator>

class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
}

Contig extract()
{
    // ...
    return Contig("ACTTCTGTATTGGGTCTTTAATAG");
}

int main()
{
    Contig seq = extract();
    analyze(std::move(seq));
}
```

Looks good, but we are not done yet. There is some essential stuff missing.

First, have a look at what happens if we forget to write our own destructor for this class.

```
$ c++ tour.cpp && ./a.out
ACTTCTGTATTGGGTCTTTAATAG
$
```

```
#include <algorithm>
#include <cstring>
#include <iostream>
#include <iterator>

class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    // ~Contig() {}
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
}

Contig extract()
{
    // ...
    return Contig("ACTTCTGTATTGGGTCTTTAATAG");
}

int main()
{
    Contig seq = extract();
    analyze(std::move(seq));
}
```

```cpp
#include <algorithm>
#include <cstring>
#include <iostream>
#include <iterator>

class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    // ~Contig() {}
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
}

Contig extract()
{
    // ...
    return Contig("ACTTCTGTATTGGGTCTTTAATAG");
}

int main()
{
    Contig seq = extract();
    analyze(std::move(seq));
}
```

```
$ c++ -fsanitize=leak,address tour.cpp && ./a.out
ACTTCTGTATTGGGTCTTTAATAG
detected memory leaks
$
```

```cpp
#include <algorithm>
#include <cstring>
#include <iostream>
#include <iterator>

class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    // ~Contig() {}
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
}

Contig extract()
{
    // ...
    return Contig("ACTTCTGTATTGGGTCTTTAATAG");
}

int main()
{
    Contig seq = extract();
    analyze(std::move(seq));
}
```

If we do not implement a **user defined destructor** for this class, an empty **implicit destructor** will be created for us and using the class might result in memory leakage.

```
$ c++ -fsanitize=leak,address tour.cpp && ./a.out
ACTTCTGTATTGGGTCTTTAATAG
detected memory leaks
$
```

```cpp
#include <algorithm>
#include <cstring>
#include <iostream>
#include <iterator>

class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    // ~Contig() {}
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
}

Contig extract()
{
    // ...
    return Contig("ACTTCTGTATTGGGTCTTTAATAG");
}

int main()
{
    Contig seq = extract();
    analyze(std::move(seq));
}
```

If we do not implement a **user defined destructor** for this class, an empty **implicit destructor** will be created for us and using the class might result in memory leakage.

However, the destructor is not the only **special member** that will be implicitly created if we don't specify them explicitly. There are four more special members we need to consider.

```
$ c++ -fsanitize=leak,address tour.cpp && ./a.out
ACTTCTGTATTGGGTCTTTAATAG
detected memory leaks
$
```

```
#include <algorithm>
#include <cstring>
#include <iostream>
#include <iterator>

class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    // ~Contig() {}
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
}

Contig extract()
{
    // ...
    return Contig("ACTTCTGTATTGGGTCTTTAATAG");
}

int main()
{
    Contig seq = extract();
    analyze(std::move(seq));
}
```

```cpp
#include <algorithm>
#include <cstring>
#include <iostream>
#include <iterator>

class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    // ~Contig() {}
    // Contig(const Contig & other) : size(other.size), data(other.data) {}
    // Contig & operator=(const Contig & other) { size = other.size; data = other.data; return *this; }
    // Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {}
    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
}

Contig extract()
{
    // ...
    return Contig("ACTTCTGTATTGGGTCTTTAATAG");
}

int main()
{
    Contig seq = extract();
    analyze(std::move(seq));
}
```

```
#include <algorithm>
#include <cstring>
#include <iostream>
#include <iterator>

class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    // ~Contig() {}
    // Contig(const Contig & other) : size(other.size), data(other.data) {}
    // Contig & operator=(const Contig & other) { size = other.size; data = other.data; return *this; }
    // Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {}
    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
}

Contig extract()
{
    // ...
    return Contig("ACTTCTGTATTGGGTCTTTAATAG");
}

int main()
{
    Contig seq = extract();
    analyze(std::move(seq));
}
```

By default, this is approximately what we get for our class. They are all doing the "wrong" thing

```
#include <algorithm>
#include <cstring>
#include <iostream>
#include <iterator>

class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    // ~Contig() {}
    // Contig(const Contig & other) : size(other.size), data(other.data) {}
    // Contig & operator=(const Contig & other) { size = other.size; data = other.data; return *this; }
    // Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {}
    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
}

Contig extract()
{
    // ...
    return Contig("ACTTCTGTATTGGGTCTTTAATAG");
}

int main()
{
    Contig seq = extract();
    analyze(std::move(seq));
}
```

By default, this is approximately what we get for our class. They are all doing the “wrong” thing

For classes that manages resources, you need to be aware of all the **special member functions** that the compiler might automatically generate for you if you don’t specify them explicitly:

- destructor
- copy constructor
- copy assignment
- move constructor
- move assignment

```cpp
#include <algorithm>
#include <cstring>
#include <iostream>
#include <iterator>

class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    // ~Contig() {}
    // Contig(const Contig & other) : size(other.size), data(other.data) {}
    // Contig & operator=(const Contig & other) { size = other.size; data = other.data; return *this; }
    // Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {}
    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
}

Contig extract()
{
    // ...
    return Contig("ACTTCTGTATTGGGTCTTTAATAG");
}

int main()
{
    Contig seq = extract();
    analyze(std::move(seq));
}
```

Let's zoom in on the class and the `analyze()` function. Then implement the special member functions one by one after first illustrating the problem and then providing a fix.

```
#include <algorithm>
#include <cstring>
#include <iostream>
#include <iterator>

class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    // ~Contig() {}
    // Contig(const Contig & other) : size(other.size), data(other.data) {}
    // Contig & operator=(const Contig & other) { size = other.size; data = other.data; return *this; }
    // Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {}
    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
}

Contig extract()
{
    // ...
    return Contig("ACTTCTGTATTGGGTCTTTAATAG");
}

int main()
{
    Contig seq = extract();
    analyze(std::move(seq));
}
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    // ~Contig() {}
    // Contig(const Contig & other) : size(other.size), data(other.data) {}
    // Contig & operator=(const Contig & other) { size = other.size; data = other.data; return *this; }
    // Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {}
    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
}
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    // ~Contig() {}
    // Contig(const Contig & other) : size(other.size), data(other.data) {}
    // Contig & operator=(const Contig & other) { size = other.size; data = other.data; return *this; }
    // Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {}
    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
}
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    // ~Contig() {}
    // Contig(const Contig & other) : size(other.size), data(other.data) {}
    // Contig & operator=(const Contig & other) { size = other.size; data = other.data; return *this; }
    // Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {}
    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
}
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    // ~Contig() {}
    // Contig(const Contig & other) : size(other.size), data(other.data) {}
    // Contig & operator=(const Contig & other) { size = other.size; data = other.data; return *this; }
    // Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {}
    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
}
```

```cpp
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    // ~Contig() {}
    // Contig(const Contig & other) : size(other.size), data(other.data) {}
    // Contig & operator=(const Contig & other) { size = other.size; data = other.data; return *this; }
    // Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {}
    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
}
```

```cpp
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    // ~Contig() {}
    // Contig(const Contig & other) : size(other.size), data(other.data) {}


    // Contig & operator=(const Contig & other) { size = other.size; data = other.data; return *this; }





    // Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {}



    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }




private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
}
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    // ~Contig() {}
    // Contig(const Contig & other) : size(other.size), data(other.data) {}


    // Contig & operator=(const Contig & other) { size = other.size; data = other.data; return *this; }






    // Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {}



    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }




private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
}
```

```cpp
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    // ~Contig() {}
    // Contig(const Contig & other) : size(other.size), data(other.data) {}


    // Contig & operator=(const Contig & other) { size = other.size; data = other.data; return *this; }





    // Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {}



    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }




private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
}
```

```
$ c++ -fsanitize=leak,address tour.cpp && ./a.out
ACTTCTGTATTGGGTCTTTAATAG
detected memory leaks
$
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    // ~Contig() {}
    // Contig(const Contig & other) : size(other.size), data(other.data) {}


    // Contig & operator=(const Contig & other) { size = other.size; data = other.data; return *this; }






    // Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {}



    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }




private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
}
```

The first one is obvious. If you grab a resource in the constructor, it is usually a good idea to release it in the destructor.

```
$ c++ -fsanitize=leak,address tour.cpp && ./a.out
ACTTCTGTATTGGGTCTTTAATAG
detected memory leaks
$
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    // ~Contig() {}
    // Contig(const Contig & other) : size(other.size), data(other.data) {}

    // Contig & operator=(const Contig & other) { size = other.size; data = other.data; return *this; }

    // Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {}

    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }

private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
}
```

The first one is obvious. If you grab a resource in the constructor, it is usually a good idea to release it in the destructor.

```
$ c++ -fsanitize=leak,address tour.cpp && ./a.out
ACTTCTGTATTGGGTCTTTAATAG
detected memory leaks
$
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    // Contig(const Contig & other) : size(other.size), data(other.data) {}


    // Contig & operator=(const Contig & other) { size = other.size; data = other.data; return *this; }





    // Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {}



    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }




private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
}
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    // Contig(const Contig & other) : size(other.size), data(other.data) {}


    // Contig & operator=(const Contig & other) { size = other.size; data = other.data; return *this; }






    // Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {}



    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }




private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
}
```

```
$ c++ -fsanitize=leak,address tour.cpp && ./a.out
ACTTCTGTATTGGGTCTTTAATAG
$
```

```cpp
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    // Contig(const Contig & other) : size(other.size), data(other.data) {}


    // Contig & operator=(const Contig & other) { size = other.size; data = other.data; return *this; }






    // Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {}



    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }




private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
}
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    // Contig(const Contig & other) : size(other.size), data(other.data) {}

    // Contig & operator=(const Contig & other) { size = other.size; data = other.data; return *this; }

    // Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {}

    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }

private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
}
```

The problem with the implicit copy constructor can be illustrated by creating a temporary copy in the `analyze()` function. Eg…

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    // Contig(const Contig & other) : size(other.size), data(other.data) {}

    // Contig & operator=(const Contig & other) { size = other.size; data = other.data; return *this; }

    // Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {}

    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }

private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
}
```

The problem with the implicit copy constructor can be illustrated by creating a temporary copy in the `analyze()` function. Eg…

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    // Contig(const Contig & other) : size(other.size), data(other.data) {}


    // Contig & operator=(const Contig & other) { size = other.size; data = other.data; return *this; }





    // Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {}



    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }




private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
}
```

The problem with the implicit copy constructor can be illustrated by creating a temporary copy in the `analyze()` function. Eg…

```cpp
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    // Contig(const Contig & other) : size(other.size), data(other.data) {}


    // Contig & operator=(const Contig & other) { size = other.size; data = other.data; return *this; }





    // Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {}



    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }




private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
    Contig tmp(seq);

    // ...
}
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    // Contig(const Contig & other) : size(other.size), data(other.data) {}


    // Contig & operator=(const Contig & other) { size = other.size; data = other.data; return *this; }





    // Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {}



    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }




private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
    Contig tmp(seq);

    // ...
}
```

```
$ c++ -fsanitize=leak,address tour.cpp && ./a.out
ACTTCTGTATTGGGTCTTTAATAG
attempting double-free
$
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    // Contig(const Contig & other) : size(other.size), data(other.data) {}


    // Contig & operator=(const Contig & other) { size = other.size; data = other.data; return *this; }





    // Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {}



    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }




private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
    Contig tmp(seq);

    // ...
}
```

The implicit default copy constructor makes a shallow copy of the object so we end up with two pointers to the same memory object. When `tmp` and `seq` are destroyed both of them will try to free the same allocated resource.

```
$ c++ -fsanitize=leak,address tour.cpp && ./a.out
ACTTCTGTATTGGGTCTTTAATAG
attempting double-free
$
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    // Contig(const Contig & other) : size(other.size), data(other.data) {}


    // Contig & operator=(const Contig & other) { size = other.size; data = other.data; return *this; }





    // Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {}


    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }




private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
    Contig tmp(seq);

    // ...
}
```

The implicit default copy constructor makes a shallow copy of the object so we end up with two pointers to the same memory object. When `tmp` and `seq` are destroyed both of them will try to free the same allocated resource.

We fix this by writing code so that the copy constructor allocates a new memory array and copies data into it.

```
$ c++ -fsanitize=leak,address tour.cpp && ./a.out
ACTTCTGTATTGGGTCTTTAATAG
attempting double-free
$
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    // Contig(const Contig & other) : size(other.size), data(other.data) {}


    // Contig & operator=(const Contig & other) { size = other.size; data = other.data; return *this; }





    // Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {}



    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }




private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
    Contig tmp(seq);

    // ...
}
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    // Contig(const Contig & other) : size(other.size), data(other.data) {}


    // Contig & operator=(const Contig & other) { size = other.size; data = other.data; return *this; }






    // Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {}



    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }




private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
    Contig tmp(seq);

    // ...
}
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    // Contig & operator=(const Contig & other) { size = other.size; data = other.data; return *this; }





    // Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {}



    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }




private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
    Contig tmp(seq);

    // ...
}
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    // Contig & operator=(const Contig & other) { size = other.size; data = other.data; return *this; }





    // Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {}



    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }




private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
    Contig tmp(seq);

    // ...
}
```

```
$ c++ -fsanitize=leak,address tour.cpp && ./a.out
ACTTCTGTATTGGGTCTTTAATAG
$
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    // Contig & operator=(const Contig & other) { size = other.size; data = other.data; return *this; }





    // Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {}



    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }




private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
    Contig tmp(seq);

    // ...
}
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    // Contig & operator=(const Contig & other) { size = other.size; data = other.data; return *this; }




    // Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {}



    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }




private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
    Contig tmp(seq);

    // ...
}
```

The next problem can be illustrated by assigning a `Contig` object to another `Contig` object. Eg...

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    // Contig & operator=(const Contig & other) { size = other.size; data = other.data; return *this; }




    // Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {}



    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }




private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
    Contig tmp(seq);

    // ...
}
```

The next problem can be illustrated by assigning a `Contig` object to another `Contig` object. Eg...

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    // Contig & operator=(const Contig & other) { size = other.size; data = other.data; return *this; }

    // Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {}

    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }

private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
    Contig tmp(seq);

    // ...
}
```

The next problem can be illustrated by assigning a `Contig` object to another `Contig` object. Eg...

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    // Contig & operator=(const Contig & other) { size = other.size; data = other.data; return *this; }





    // Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {}



    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }




private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
    Contig tmp(seq);
    seq = tmp;
    // ...
}
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    // Contig & operator=(const Contig & other) { size = other.size; data = other.data; return *this; }





    // Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {}



    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }




private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
    Contig tmp(seq);
    seq = tmp;
    // ...
}
```

```
$ c++ -fsanitize=leak,address tour.cpp && ./a.out
ACTTCTGTATTGGGTCTTTAATAG
attempting double-free
$
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    // Contig & operator=(const Contig & other) { size = other.size; data = other.data; return *this; }




    // Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {}



    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }




private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
    Contig tmp(seq);
    seq = tmp;
    // ...
}
```

It is basically the same problem as before. We get a shallow copy by default. We need to write the code to reallocate our array and then copy the elements from the other object. Here we use a so-called **copy-swap idiom** to implement the copy assignment operator.

```
$ c++ -fsanitize=leak,address tour.cpp && ./a.out
ACTTCTGTATTGGGTCTTTAATAG
attempting double-free
$
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    // Contig & operator=(const Contig & other) { size = other.size; data = other.data; return *this; }



    // Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {}


    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }



private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
    Contig tmp(seq);
    seq = tmp;
    // ...
}
```

It is basically the same problem as before. We get a shallow copy by default. We need to write the code to reallocate our array and then copy the elements from the other object. Here we use a so-called **copy-swap idiom** to implement the copy assignment operator.

```
$ c++ -fsanitize=leak,address tour.cpp && ./a.out
ACTTCTGTATTGGGTCTTTAATAG
attempting double-free
$
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        Contig tmp(other);
        std::swap(size, tmp.size);
        std::swap(data, tmp.data);
        return *this;
    }
    // Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {}



    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }




private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
    Contig tmp(seq);
    seq = tmp;
    // ...
}
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        Contig tmp(other);
        std::swap(size, tmp.size);
        std::swap(data, tmp.data);
        return *this;
    }
    // Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {}



    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }




private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
    Contig tmp(seq);
    seq = tmp;
    // ...
}
```

```
$ c++ -fsanitize=leak,address tour.cpp && ./a.out
ACTTCTGTATTGGGTCTTTAATAG
$
```

```cpp
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        Contig tmp(other);
        std::swap(size, tmp.size);
        std::swap(data, tmp.data);
        return *this;
    }
    // Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {}



    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }




private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
    Contig tmp(seq);
    seq = tmp;
    // ...
}
```

Now, let's try to invoke the implicit move constructor.

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        Contig tmp(other);
        std::swap(size, tmp.size);
        std::swap(data, tmp.data);
        return *this;
    }
    // Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {}



    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }




private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
    Contig tmp(seq);
    seq = tmp;
    // ...
}
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        Contig tmp(other);
        std::swap(size, tmp.size);
        std::swap(data, tmp.data);
        return *this;
    }
    // Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {}



    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }




private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
    Contig tmp(seq);
    seq = tmp;
    // ...
}
```

Now, let's try to invoke the implicit move constructor.

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        Contig tmp(other);
        std::swap(size, tmp.size);
        std::swap(data, tmp.data);
        return *this;
    }
    // Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {}



    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }




private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
    Contig tmp(seq);
    seq = tmp;
    // ...
}
```

Now, let's try to invoke the implicit move constructor.

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        Contig tmp(other);
        std::swap(size, tmp.size);
        std::swap(data, tmp.data);
        return *this;
    }
    // Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {}



    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }




private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
    Contig tmp(std::move(seq));

    // ...
}
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        Contig tmp(other);
        std::swap(size, tmp.size);
        std::swap(data, tmp.data);
        return *this;
    }
    // Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {}



    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }




private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
    Contig tmp(std::move(seq));

    // ...
}
```

```
$ c++ -fsanitize=leak,address tour.cpp && ./a.out
ACTTCTGTATTGGGTCTTTAATAG
$
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        Contig tmp(other);
        std::swap(size, tmp.size);
        std::swap(data, tmp.data);
        return *this;
    }
    // Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {}



    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }




private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
    Contig tmp(std::move(seq));

    // ...
}
```

```
$ c++ -fsanitize=leak,address tour.cpp && ./a.out
ACTTCTGTATTGGGTCTTTAATAG
$
```

Huh? No error?

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        Contig tmp(other);
        std::swap(size, tmp.size);
        std::swap(data, tmp.data);
        return *this;
    }
    // Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {}



    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }




private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
    Contig tmp(std::move(seq));

    // ...
}
```

Huh? No error?

When there is a user-declared destructor and/or a user-declared copy constructor, then no default move constructor will be provided, and the fallback is to invoke the copy constructor.

```
$ c++ -fsanitize=leak,address tour.cpp && ./a.out
ACTTCTGTATTGGGTCTTTAATAG
$
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        Contig tmp(other);
        std::swap(size, tmp.size);
        std::swap(data, tmp.data);
        return *this;
    }
    // Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {}


    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }




private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
    Contig tmp(std::move(seq));

    // ...
}
```

Huh? No error?

When there is a user-declared destructor and/or a user-declared copy constructor, then no default move constructor will be provided, and the fallback is to invoke the copy constructor.

We can ask for a (faulty) default move constructor to be generated like this...

```
$ c++ -fsanitize=leak,address tour.cpp && ./a.out
ACTTCTGTATTGGGTCTTTAATAG
$
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        Contig tmp(other);
        std::swap(size, tmp.size);
        std::swap(data, tmp.data);
        return *this;
    }
    // Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {}


    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }




private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
    Contig tmp(std::move(seq));

    // ...
}
```

Huh? No error?

When there is a user-declared destructor and/or a user-declared copy constructor, then no default move constructor will be provided, and the fallback is to invoke the copy constructor.

We can ask for a (faulty) default move constructor to be generated like this...

```
$ c++ -fsanitize=leak,address tour.cpp && ./a.out
ACTTCTGTATTGGGTCTTTAATAG
$
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        Contig tmp(other);
        std::swap(size, tmp.size);
        std::swap(data, tmp.data);
        return *this;
    }
    // Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {}
    Contig(Contig && other) = default;


    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }




private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
    Contig tmp(std::move(seq));

    // ...
}
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        Contig tmp(other);
        std::swap(size, tmp.size);
        std::swap(data, tmp.data);
        return *this;
    }
    // Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {}
    Contig(Contig && other) = default;


    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }




private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
    Contig tmp(std::move(seq));

    // ...
}
```

```
$ c++ -fsanitize=leak,address tour.cpp && ./a.out
ACTTCTGTATTGGGTCTTTAATAG
attempting double-free
$
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        Contig tmp(other);
        std::swap(size, tmp.size);
        std::swap(data, tmp.data);
        return *this;
    }
    // Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {}
    Contig(Contig && other) = default;


    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }




private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
    Contig tmp(std::move(seq));

    // ...
}
```

And now it fails, as expected, since the default move constructor for this class is also just doing a shallow copy of the data members. Remember that `std::move` is not a magical function, think about it as a cast that is not generating any code.

```
$ c++ -fsanitize=leak,address tour.cpp && ./a.out
ACTTCTGTATTGGGTCTTTAATAG
attempting double-free
$
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        Contig tmp(other);
        std::swap(size, tmp.size);
        std::swap(data, tmp.data);
        return *this;
    }
    // Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {}
    Contig(Contig && other) = default;


    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }




private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
    Contig tmp(std::move(seq));

    // ...
}
```

And now it fails, as expected, since the default move constructor for this class is also just doing a shallow copy of the data members. Remember that `std::move` is not a magical function, think about it as a cast that is not generating any code.

You can also ask for the move constructor to be deleted. In this case we get a compilation error because we still have a user-declared move constructor (even if it is deleted), and therefor there will be no fallback to the copy ctor.

```
$ c++ -fsanitize=leak,address tour.cpp && ./a.out
ACTTCTGTATTGGGTCTTTAATAG
attempting double-free
$
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        Contig tmp(other);
        std::swap(size, tmp.size);
        std::swap(data, tmp.data);
        return *this;
    }
    // Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {}
    Contig(Contig && other) = default;

    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }



private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
    Contig tmp(std::move(seq));

    // ...
}
```

And now it fails, as expected, since the default move constructor for this class is also just doing a shallow copy of the data members. Remember that `std::move` is not a magical function, think about it as a cast that is not generating any code.

You can also ask for the move constructor to be deleted. In this case we get a compilation error because we still have a user-declared move constructor (even if it is deleted), and therefor there will be no fallback to the copy ctor.

```
$ c++ -fsanitize=leak,address tour.cpp && ./a.out
ACTTCTGTATTGGGTCTTTAATAG
attempting double-free
$
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        Contig tmp(other);
        std::swap(size, tmp.size);
        std::swap(data, tmp.data);
        return *this;
    }
    // Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {}
    Contig(Contig && other) = default;

    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }



private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
    Contig tmp(std::move(seq));

    // ...
}
```

And now it fails, as expected, since the default move constructor for this class is also just doing a shallow copy of the data members. Remember that `std::move` is not a magical function, think about it as a cast that is not generating any code.

You can also ask for the move constructor to be deleted. In this case we get a compilation error because we still have a user-declared move constructor (even if it is deleted), and therefor there will be no fallback to the copy ctor.

```
$ c++ -fsanitize=leak,address tour.cpp && ./a.out
ACTTCTGTATTGGGTCTTTAATAG
attempting double-free
$
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        Contig tmp(other);
        std::swap(size, tmp.size);
        std::swap(data, tmp.data);
        return *this;
    }
    // Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {}
    Contig(Contig && other) = delete;


    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }




private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
    Contig tmp(std::move(seq));

    // ...
}
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        Contig tmp(other);
        std::swap(size, tmp.size);
        std::swap(data, tmp.data);
        return *this;
    }
    // Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {}
    Contig(Contig && other) = delete;


    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }




private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
    Contig tmp(std::move(seq));

    // ...
}
```

```
$ c++ -fsanitize=leak,address tour.cpp && ./a.out
call to deleted constructor of 'Contig'
$
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        Contig tmp(other);
        std::swap(size, tmp.size);
        std::swap(data, tmp.data);
        return *this;
    }
    // Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {}
    Contig(Contig && other) = delete;


    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }




private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
    Contig tmp(std::move(seq));

    // ...
}
```

or we can implement it properly

```
$ c++ -fsanitize=leak,address tour.cpp && ./a.out
call to deleted constructor of 'Contig'
$
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        Contig tmp(other);
        std::swap(size, tmp.size);
        std::swap(data, tmp.data);
        return *this;
    }
    // Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {}
    Contig(Contig && other) = delete;

    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }

private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
    Contig tmp(std::move(seq));

    // ...
}
```

or we can implement it properly

```
$ c++ -fsanitize=leak,address tour.cpp && ./a.out
call to deleted constructor of 'Contig'
$
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        Contig tmp(other);
        std::swap(size, tmp.size);
        std::swap(data, tmp.data);
        return *this;
    }
    Contig(Contig && other) : size(0), data(nullptr) {
        std::swap(size, other.size);
        std::swap(data, other.data);
    }
    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }




private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
    Contig tmp(std::move(seq));

    // ...
}
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        Contig tmp(other);
        std::swap(size, tmp.size);
        std::swap(data, tmp.data);
        return *this;
    }
    Contig(Contig && other) : size(0), data(nullptr) {
        std::swap(size, other.size);
        std::swap(data, other.data);
    }
    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }




private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
    Contig tmp(std::move(seq));

    // ...
}
```

```
$ c++ -fsanitize=leak,address tour.cpp && ./a.out
ACTTCTGTATTGGGTCTTTAATAG
$
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        Contig tmp(other);
        std::swap(size, tmp.size);
        std::swap(data, tmp.data);
        return *this;
    }
    Contig(Contig && other) : size(0), data(nullptr) {
        std::swap(size, other.size);
        std::swap(data, other.data);
    }
    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }



private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
    Contig tmp(std::move(seq));

    // ...
}
```

finally...

```
$ c++ -fsanitize=leak,address tour.cpp && ./a.out
ACTTCTGTATTGGGTCTTTAATAG
$
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        Contig tmp(other);
        std::swap(size, tmp.size);
        std::swap(data, tmp.data);
        return *this;
    }
    Contig(Contig && other) : size(0), data(nullptr) {
        std::swap(size, other.size);
        std::swap(data, other.data);
    }
    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }




private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
    Contig tmp(std::move(seq));

    // ...
}
```

finally...

```
$ c++ -fsanitize=leak,address tour.cpp && ./a.out
ACTTCTGTATTGGGTCTTTAATAG
$
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        Contig tmp(other);
        std::swap(size, tmp.size);
        std::swap(data, tmp.data);
        return *this;
    }
    Contig(Contig && other) : size(0), data(nullptr) {
        std::swap(size, other.size);
        std::swap(data, other.data);
    }
    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }




private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
    Contig tmp("");
    tmp = std::move(seq);
    // ...
}
```

```cpp
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        Contig tmp(other);
        std::swap(size, tmp.size);
        std::swap(data, tmp.data);
        return *this;
    }
    Contig(Contig && other) : size(0), data(nullptr) {
        std::swap(size, other.size);
        std::swap(data, other.data);
    }
    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }




private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
    Contig tmp("");
    tmp = std::move(seq);
    // ...
}
```

```
$ c++ -fsanitize=leak,address tour.cpp && ./a.out
ACTTCTGTATTGGGTCTTTAATAG
$
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        Contig tmp(other);
        std::swap(size, tmp.size);
        std::swap(data, tmp.data);
        return *this;
    }
    Contig(Contig && other) : size(0), data(nullptr) {
        std::swap(size, other.size);
        std::swap(data, other.data);
    }
    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }



private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
    Contig tmp("");
    tmp = std::move(seq);
    // ...
}
```

It is the copy assignment that is called now.

```
$ c++ -fsanitize=leak,address tour.cpp && ./a.out
ACTTCTGTATTGGGTCTTTAATAG
$
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        Contig tmp(other);
        std::swap(size, tmp.size);
        std::swap(data, tmp.data);
        return *this;
    }
    Contig(Contig && other) : size(0), data(nullptr) {
        std::swap(size, other.size);
        std::swap(data, other.data);
    }
    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }



private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
    Contig tmp("");
    tmp = std::move(seq);
    // ...
}
```

It is the copy assignment that is called now.

```
$ c++ -fsanitize=leak,address tour.cpp && ./a.out
ACTTCTGTATTGGGTCTTTAATAG
$
```

```cpp
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        Contig tmp(other);
        std::swap(size, tmp.size);
        std::swap(data, tmp.data);
        return *this;
    }
    Contig(Contig && other) : size(0), data(nullptr) {
        std::swap(size, other.size);
        std::swap(data, other.data);
    }
    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }
    Contig & operator=(Contig && other) = default;



private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
    Contig tmp("");
    tmp = std::move(seq);
    // ...
}
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        Contig tmp(other);
        std::swap(size, tmp.size);
        std::swap(data, tmp.data);
        return *this;
    }
    Contig(Contig && other) : size(0), data(nullptr) {
        std::swap(size, other.size);
        std::swap(data, other.data);
    }
    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }
    Contig & operator=(Contig && other) = default;



private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
    Contig tmp("");
    tmp = std::move(seq);
    // ...
}
```

```
$ c++ -fsanitize=leak,address tour.cpp && ./a.out
ACTTCTGTATTGGGTCTTTAATAG
attempting double-free
$
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        Contig tmp(other);
        std::swap(size, tmp.size);
        std::swap(data, tmp.data);
        return *this;
    }
    Contig(Contig && other) : size(0), data(nullptr) {
        std::swap(size, other.size);
        std::swap(data, other.data);
    }
    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }
    Contig & operator=(Contig && other) = default;



private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
    Contig tmp("");
    tmp = std::move(seq);
    // ...
}
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        Contig tmp(other);
        std::swap(size, tmp.size);
        std::swap(data, tmp.data);
        return *this;
    }
    Contig(Contig && other) : size(0), data(nullptr) {
        std::swap(size, other.size);
        std::swap(data, other.data);
    }
    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }
    Contig & operator=(Contig && other) = default;



private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
    Contig tmp("");
    tmp = std::move(seq);
    // ...
}
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        Contig tmp(other);
        std::swap(size, tmp.size);
        std::swap(data, tmp.data);
        return *this;
    }
    Contig(Contig && other) : size(0), data(nullptr) {
        std::swap(size, other.size);
        std::swap(data, other.data);
    }
    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }
    Contig & operator=(Contig && other) = delete;



private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
    Contig tmp("");
    tmp = std::move(seq);
    // ...
}
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        Contig tmp(other);
        std::swap(size, tmp.size);
        std::swap(data, tmp.data);
        return *this;
    }
    Contig(Contig && other) : size(0), data(nullptr) {
        std::swap(size, other.size);
        std::swap(data, other.data);
    }
    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }
    Contig & operator=(Contig && other) = delete;



private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
    Contig tmp("");
    tmp = std::move(seq);
    // ...
}
```

```
$ c++ -fsanitize=leak,address tour.cpp && ./a.out
overload resolution selected deleted operator '='
$
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        Contig tmp(other);
        std::swap(size, tmp.size);
        std::swap(data, tmp.data);
        return *this;
    }
    Contig(Contig && other) : size(0), data(nullptr) {
        std::swap(size, other.size);
        std::swap(data, other.data);
    }
    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }
    Contig & operator=(Contig && other) = delete;



private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
    Contig tmp("");
    tmp = std::move(seq);
    // ...
}
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        Contig tmp(other);
        std::swap(size, tmp.size);
        std::swap(data, tmp.data);
        return *this;
    }
    Contig(Contig && other) : size(0), data(nullptr) {
        std::swap(size, other.size);
        std::swap(data, other.data);
    }
    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }
    Contig & operator=(Contig && other) = delete;



private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
    Contig tmp("");
    tmp = std::move(seq);
    // ...
}
```

An acceptable implementation might look like this.

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        Contig tmp(other);
        std::swap(size, tmp.size);
        std::swap(data, tmp.data);
        return *this;
    }
    Contig(Contig && other) : size(0), data(nullptr) {
        std::swap(size, other.size);
        std::swap(data, other.data);
    }
    // Contig & operator=(Contig && other) { size = std::move(other.size); data = std::move(other.data); return *this; }
    Contig & operator=(Contig && other) = delete;


private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
    Contig tmp("");
    tmp = std::move(seq);
    // ...
}
```

An acceptable implementation might look like this.

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        Contig tmp(other);
        std::swap(size, tmp.size);
        std::swap(data, tmp.data);
        return *this;
    }
    Contig(Contig && other) : size(0), data(nullptr) {
        std::swap(size, other.size);
        std::swap(data, other.data);
    }
    Contig & operator=(Contig && other) {
        std::swap(size, other.size);
        std::swap(data, other.data);
        return *this;
    }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
    Contig tmp("");
    tmp = std::move(seq);
    // ...
}
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        Contig tmp(other);
        std::swap(size, tmp.size);
        std::swap(data, tmp.data);
        return *this;
    }
    Contig(Contig && other) : size(0), data(nullptr) {
        std::swap(size, other.size);
        std::swap(data, other.data);
    }
    Contig & operator=(Contig && other) {
        std::swap(size, other.size);
        std::swap(data, other.data);
        return *this;
    }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};

void analyze(Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
    Contig tmp("");
    tmp = std::move(seq);
    // ...
}
```

```
$ c++ -fsanitize=leak,address tour.cpp && ./a.out
ACTTCTGTATTGGGTCTTTAATAG
$
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        Contig tmp(other);
        std::swap(size, tmp.size);
        std::swap(data, tmp.data);
        return *this;
    }
    Contig(Contig && other) : size(0), data(nullptr) {
        std::swap(size, other.size);
        std::swap(data, other.data);
    }
    Contig & operator=(Contig && other) {
        std::swap(size, other.size);
        std::swap(data, other.data);
        return *this;
    }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        Contig tmp(other);
        std::swap(size, tmp.size);
        std::swap(data, tmp.data);
        return *this;
    }
    Contig(Contig && other) : size(0), data(nullptr) {
        std::swap(size, other.size);
        std::swap(data, other.data);
    }
    Contig & operator=(Contig && other) {
        std::swap(size, other.size);
        std::swap(data, other.data);
        return *this;
    }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};
```

So now we have provided all the 5 special members and we have a class that supports copy and move properly.

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        Contig tmp(other);
        std::swap(size, tmp.size);
        std::swap(data, tmp.data);
        return *this;
    }
    Contig(Contig && other) : size(0), data(nullptr) {
        std::swap(size, other.size);
        std::swap(data, other.data);
    }
    Contig & operator=(Contig && other) {
        std::swap(size, other.size);
        std::swap(data, other.data);
        return *this;
    }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};
```

So now we have provided all the 5 special members and we have a class that supports copy and move properly.

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        Contig tmp(other);
        std::swap(size, tmp.size);
        std::swap(data, tmp.data);
        return *this;
    }
    Contig(Contig && other) : size(0), data(nullptr) {
        std::swap(size, other.size);
        std::swap(data, other.data);
    }
    Contig & operator=(Contig && other) {
        std::swap(size, other.size);
        std::swap(data, other.data);
        return *this;
    }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};
```

So now we have provided all the 5 special members and we have a class that supports copy and move properly.

You can even take this idea further, and reduce code duplication by implementing and calling a free-standing `swap` function that can swap two `Contig` objects. But we are not going to show that here.

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        Contig tmp(other);
        std::swap(size, tmp.size);
        std::swap(data, tmp.data);
        return *this;
    }
    Contig(Contig && other) : size(0), data(nullptr) {
        std::swap(size, other.size);
        std::swap(data, other.data);
    }
    Contig & operator=(Contig && other) {
        std::swap(size, other.size);
        std::swap(data, other.data);
        return *this;
    }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        Contig tmp(other);
        std::swap(size, tmp.size);
        std::swap(data, tmp.data);
        return *this;
    }
    Contig(Contig && other) : size(0), data(nullptr) {
        std::swap(size, other.size);
        std::swap(data, other.data);
    }
    Contig & operator=(Contig && other) {
        std::swap(size, other.size);
        std::swap(data, other.data);
        return *this;
    }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        Contig tmp(other);
        std::swap(size, tmp.size);
        std::swap(data, tmp.data);
        return *this;
    }
    Contig(Contig && other) : size(0), data(nullptr) {
        std::swap(size, other.size);
        std::swap(data, other.data);
    }
    Contig & operator=(Contig && other) {
        std::swap(size, other.size);
        std::swap(data, other.data);
        return *this;
    }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};
```

Another way to implement the move assignment is to copy the content of the other object, and then reset it's state. I call this "gutting" the other object.

```cpp
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        Contig tmp(other);
        std::swap(size, tmp.size);
        std::swap(data, tmp.data);
        return *this;
    }
    Contig(Contig && other) : size(0), data(nullptr) {
        std::swap(size, other.size);
        std::swap(data, other.data);
    }
    Contig & operator=(Contig && other) {
        std::swap(size, other.size);
        std::swap(data, other.data);



        return *this;
    }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        Contig tmp(other);
        std::swap(size, tmp.size);
        std::swap(data, tmp.data);
        return *this;
    }
    Contig(Contig && other) : size(0), data(nullptr) {
        std::swap(size, other.size);
        std::swap(data, other.data);
    }
    Contig & operator=(Contig && other) {
        delete[] data;
        size = std::move(other.size);
        data = std::move(other.data);
        other.size = 0;
        other.data = nullptr;
        return *this;
    }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        Contig tmp(other);
        std::swap(size, tmp.size);
        std::swap(data, tmp.data);
        return *this;
    }
    Contig(Contig && other) : size(0), data(nullptr) {
        std::swap(size, other.size);
        std::swap(data, other.data);
    }
    Contig & operator=(Contig && other) {
        delete[] data;
        size = std::move(other.size);
        data = std::move(other.data);
        other.size = 0;
        other.data = nullptr;
        return *this;
    }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};
```

```
$ c++ -fsanitize=leak,address tour.cpp && ./a.out
ACTTCTGTATTGGGTCTTTAATAG
$
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        Contig tmp(other);
        std::swap(size, tmp.size);
        std::swap(data, tmp.data);
        return *this;
    }
    Contig(Contig && other) : size(0), data(nullptr) {
        std::swap(size, other.size);
        std::swap(data, other.data);
    }
    Contig & operator=(Contig && other) {
        delete[] data;
        size = std::move(other.size);
        data = std::move(other.data);
        other.size = 0;
        other.data = nullptr;
        return *this;
    }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        Contig tmp(other);
        std::swap(size, tmp.size);
        std::swap(data, tmp.data);
        return *this;
    }
    Contig(Contig && other) : size(0), data(nullptr) {
        std::swap(size, other.size);
        std::swap(data, other.data);
    }
    Contig & operator=(Contig && other) {
        delete[] data;
        size = std::move(other.size);
        data = std::move(other.data);
        other.size = 0;
        other.data = nullptr;
        return *this;
    }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        Contig tmp(other);
        std::swap(size, tmp.size);
        std::swap(data, tmp.data);
        return *this;
    }
    Contig(Contig && other) : size(0), data(nullptr) {
        std::swap(size, other.size);
        std::swap(data, other.data);
    }
    Contig & operator=(Contig && other) {
        delete[] data;
        size = std::move(other.size);
        data = std::move(other.data);
        other.size = 0;
        other.data = nullptr;
        return *this;
    }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};
```

Similarly, in the move constructor, we can just do a shallow copy of the other object, but then reset the other object to an empty state.

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        Contig tmp(other);
        std::swap(size, tmp.size);
        std::swap(data, tmp.data);
        return *this;
    }
    Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {
        other.size = 0;
        other.data = nullptr;
    }
    Contig & operator=(Contig && other) {
        delete[] data;
        size = std::move(other.size);
        data = std::move(other.data);
        other.size = 0;
        other.data = nullptr;
        return *this;
    }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};
```

```cpp
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        Contig tmp(other);
        std::swap(size, tmp.size);
        std::swap(data, tmp.data);
        return *this;
    }
    Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {
        other.size = 0;
        other.data = nullptr;
    }
    Contig & operator=(Contig && other) {
        delete[] data;
        size = std::move(other.size);
        data = std::move(other.data);
        other.size = 0;
        other.data = nullptr;
        return *this;
    }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};
```

```
$ c++ -fsanitize=leak,address tour.cpp && ./a.out
ACTTCTGTATTGGGTCTTTAATAG
$
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        Contig tmp(other);
        std::swap(size, tmp.size);
        std::swap(data, tmp.data);
        return *this;
    }
    Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {
        other.size = 0;
        other.data = nullptr;
    }
    Contig & operator=(Contig && other) {
        delete[] data;
        size = std::move(other.size);
        data = std::move(other.data);
        other.size = 0;
        other.data = nullptr;
        return *this;
    }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        Contig tmp(other);
        std::swap(size, tmp.size);
        std::swap(data, tmp.data);
        return *this;
    }
    Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {
        other.size = 0;
        other.data = nullptr;
    }
    Contig & operator=(Contig && other) {
        delete[] data;
        size = std::move(other.size);
        data = std::move(other.data);
        other.size = 0;
        other.data = nullptr;
        return *this;
    }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        Contig tmp(other);
        std::swap(size, tmp.size);
        std::swap(data, tmp.data);
        return *this;
    }
    Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {
        other.size = 0;
        other.data = nullptr;
    }
    Contig & operator=(Contig && other) {
        delete[] data;
        size = std::move(other.size);
        data = std::move(other.data);
        other.size = 0;
        other.data = nullptr;
        return *this;
    }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};
```

It is worth pointing out that `std::move()` does not have any effect here. Since we are working with primitive types we could have just skipped the "cast" here, and used regular assignment instead.

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        Contig tmp(other);
        std::swap(size, tmp.size);
        std::swap(data, tmp.data);
        return *this;
    }
    Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {
        other.size = 0;
        other.data = nullptr;
    }
    Contig & operator=(Contig && other) {
        delete[] data;
        size = std::move(other.size);
        data = std::move(other.data);
        other.size = 0;
        other.data = nullptr;
        return *this;
    }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};
```

```cpp
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        Contig tmp(other);
        std::swap(size, tmp.size);
        std::swap(data, tmp.data);
        return *this;
    }
    Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {
        other.size = 0;
        other.data = nullptr;
    }
    Contig & operator=(Contig && other) {
        delete[] data;
        size = std::move(other.size);
        data = std::move(other.data);
        other.size = 0;
        other.data = nullptr;
        return *this;
    }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};
```

```cpp
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        Contig tmp(other);
        std::swap(size, tmp.size);
        std::swap(data, tmp.data);
        return *this;
    }
    Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {
        other.size = 0;
        other.data = nullptr;
    }
    Contig & operator=(Contig && other) {
        delete[] data;
        size = std::move(other.size);
        data = std::move(other.data);
        other.size = 0;
        other.data = nullptr;
        return *this;
    }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};
```

Finally, we must consider whether the copy-swap idiom is an acceptable design decision for our job. It is true that this idiom gives you strong exception guarantees, but if are working with really large data structures, then making a copy first does not make sense.

Here is an alternative implementation that is worth considering. Requires less memory, it might be faster, but the behavior in case of a memory allocation error is different (and unexpected?).

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        Contig tmp(other);
        std::swap(size, tmp.size);
        std::swap(data, tmp.data);
        return *this;
    }
    Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {
        other.size = 0;
        other.data = nullptr;
    }
    Contig & operator=(Contig && other) {
        delete[] data;
        size = std::move(other.size);
        data = std::move(other.data);
        other.size = 0;
        other.data = nullptr;
        return *this;
    }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        Contig tmp(other);
        std::swap(size, tmp.size);
        std::swap(data, tmp.data);





        return *this;
    }
    Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {
        other.size = 0;
        other.data = nullptr;
    }
    Contig & operator=(Contig && other) {
        delete[] data;
        size = std::move(other.size);
        data = std::move(other.data);
        other.size = 0;
        other.data = nullptr;
        return *this;
    }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        if (&other == this)
            return *this;
        delete[] data;
        data = nullptr;
        size = 0;
        data = new int[other.size];
        size = other.size;
        std::copy(other.data, other.data + size, data);
        return *this;
    }
    Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {
        other.size = 0;
        other.data = nullptr;
    }
    Contig & operator=(Contig && other) {
        delete[] data;
        size = std::move(other.size);
        data = std::move(other.data);
        other.size = 0;
        other.data = nullptr;
        return *this;
    }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};
```

```cpp
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        if (&other == this)
            return *this;
        delete[] data;
        data = nullptr;
        size = 0;
        data = new int[other.size];
        size = other.size;
        std::copy(other.data, other.data + size, data);
        return *this;
    }
    Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {
        other.size = 0;
        other.data = nullptr;
    }
    Contig & operator=(Contig && other) {
        delete[] data;
        size = std::move(other.size);
        data = std::move(other.data);
        other.size = 0;
        other.data = nullptr;
        return *this;
    }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};
```

```
$ c++ -fsanitize=leak,address tour.cpp && ./a.out
ACTTCTGTATTGGGTCTTTAATAG
$
```

```cpp
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        if (&other == this)
            return *this;
        delete[] data;
        data = nullptr;
        size = 0;
        data = new int[other.size];
        size = other.size;
        std::copy(other.data, other.data + size, data);
        return *this;
    }
    Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {
        other.size = 0;
        other.data = nullptr;
    }
    Contig & operator=(Contig && other) {
        delete[] data;
        size = std::move(other.size);
        data = std::move(other.data);
        other.size = 0;
        other.data = nullptr;
        return *this;
    }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};
```

```
$ c++ -fsanitize=leak,address tour.cpp && ./a.out
ACTTCTGTATTGGGTCTTTAATAG
$
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        if (&other == this)
            return *this;
        delete[] data;
        data = nullptr;
        size = 0;
        data = new int[other.size];
        size = other.size;
        std::copy(other.data, other.data + size, data);
        return *this;
    }
    Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {
        other.size = 0;
        other.data = nullptr;
    }
    Contig & operator=(Contig && other) {
        delete[] data;
        size = std::move(other.size);
        data = std::move(other.data);
        other.size = 0;
        other.data = nullptr;
        return *this;
    }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};
```

This is also an acceptable implementation of copy- and move-semantics for our `Contig` class. It all depends on the type of problem that you are trying to solve.

Note: If you objects are small and you don't have strict performance requirements then the copy-swap-idiom and move-swap-idiom might be a good idea to get clean code and avoid code duplication.

I just wanted to show that there are alternative ways to implement copy and move semantics - and in C++ you can choose depending on what you need.

```
$ c++ -fsanitize=leak,address tour.cpp && ./a.out
ACTTCTGTATTGGGTCTTTAATAG
$
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        if (&other == this)
            return *this;
        delete[] data;
        data = nullptr;
        size = 0;
        data = new int[other.size];
        size = other.size;
        std::copy(other.data, other.data + size, data);
        return *this;
    }
    Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {
        other.size = 0;
        other.data = nullptr;
    }
    Contig & operator=(Contig && other) {
        delete[] data;
        size = std::move(other.size);
        data = std::move(other.data);
        other.size = 0;
        other.data = nullptr;
        return *this;
    }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        if (&other == this)
            return *this;
        delete[] data;
        data = nullptr;
        size = 0;
        data = new int[other.size];
        size = other.size;
        std::copy(other.data, other.data + size, data);
        return *this;
    }
    Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {
        other.size = 0;
        other.data = nullptr;
    }
    Contig & operator=(Contig && other) {
        delete[] data;
        size = std::move(other.size);
        data = std::move(other.data);
        other.size = 0;
        other.data = nullptr;
        return *this;
    }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};
```

Perhaps you decide that copying `Contig` objects should not be allowed. Then you can consider...

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) : size(other.size), data(new int[size]) {
        std::copy(other.data, other.data + size, data);
    }
    Contig & operator=(const Contig & other) {
        if (&other == this)
            return *this;
        delete[] data;
        data = nullptr;
        size = 0;
        data = new int[other.size];
        size = other.size;
        std::copy(other.data, other.data + size, data);
        return *this;
    }
    Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {
        other.size = 0;
        other.data = nullptr;
    }
    Contig & operator=(Contig && other) {
        delete[] data;
        size = std::move(other.size);
        data = std::move(other.data);
        other.size = 0;
        other.data = nullptr;
        return *this;
    }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};
```

Perhaps you decide that copying `Contig` objects should not be allowed. Then you can consider...

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) = delete;


    Contig & operator=(const Contig & other) = delete;










    Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {
        other.size = 0;
        other.data = nullptr;
    }
    Contig & operator=(Contig && other) {
        delete[] data;
        size = std::move(other.size);
        data = std::move(other.data);
        other.size = 0;
        other.data = nullptr;
        return *this;
    }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) = delete;
    Contig & operator=(const Contig & other) = delete;
    Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {
        other.size = 0;
        other.data = nullptr;
    }
    Contig & operator=(Contig && other) {
        delete[] data;
        size = std::move(other.size);
        data = std::move(other.data);
        other.size = 0;
        other.data = nullptr;
        return *this;
    }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) = delete;
    Contig & operator=(const Contig & other) = delete;
    Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {
        other.size = 0;
        other.data = nullptr;
    }
    Contig & operator=(Contig && other) {
        delete[] data;
        size = std::move(other.size);
        data = std::move(other.data);
        other.size = 0;
        other.data = nullptr;
        return *this;
    }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) = delete;
    Contig & operator=(const Contig & other) = delete;
    Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {
        other.size = 0;
        other.data = nullptr;
    }
    Contig & operator=(Contig && other) {
        delete[] data;
        size = std::move(other.size);
        data = std::move(other.data);
        other.size = 0;
        other.data = nullptr;
        return *this;
    }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};
```

So this is an example of stuff you need to do to support move semantics, if you are managing a resource yourself.

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) = delete;
    Contig & operator=(const Contig & other) = delete;
    Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {
        other.size = 0;
        other.data = nullptr;
    }
    Contig & operator=(Contig && other) {
        delete[] data;
        size = std::move(other.size);
        data = std::move(other.data);
        other.size = 0;
        other.data = nullptr;
        return *this;
    }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};
```

So this is an example of stuff you need to do to support move semantics, if you are managing a resource yourself.

... if you are managing a resource yourself ... ?

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) = delete;
    Contig & operator=(const Contig & other) = delete;
    Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {
        other.size = 0;
        other.data = nullptr;
    }
    Contig & operator=(Contig && other) {
        delete[] data;
        size = std::move(other.size);
        data = std::move(other.data);
        other.size = 0;
        other.data = nullptr;
        return *this;
    }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};
```

So this is an example of stuff you need to do to support move semantics, if you are managing a resource yourself.

... if you are managing a resource yourself ... ?

Of course, in this case, we both can and probably should let a **smart pointer** or a **standard container** do the **resource management** for us. Then a lot of this complexity will just disappear.

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) = delete;
    Contig & operator=(const Contig & other) = delete;
    Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {
        other.size = 0;
        other.data = nullptr;
    }
    Contig & operator=(Contig && other) {
        delete[] data;
        size = std::move(other.size);
        data = std::move(other.data);
        other.size = 0;
        other.data = nullptr;
        return *this;
    }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};
```

So this is an example of stuff you need to do to support move semantics, if you are managing a resource yourself.

... if you are managing a resource yourself ... ?

Of course, in this case, we both can and probably should let a **smart pointer** or a **standard container** do the **resource management** for us. Then a lot of this complexity will just disappear.

Let's rewrite this code using `unique_ptr` instead.

```cpp
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) = delete;
    Contig & operator=(const Contig & other) = delete;
    Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {
        other.size = 0;
        other.data = nullptr;
    }
    Contig & operator=(Contig && other) {
        delete[] data;
        size = std::move(other.size);
        data = std::move(other.data);
        other.size = 0;
        other.data = nullptr;
        return *this;
    }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) = delete;
    Contig & operator=(const Contig & other) = delete;
    Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {
        other.size = 0;
        other.data = nullptr;
    }
    Contig & operator=(Contig && other) {
        delete[] data;
        size = std::move(other.size);
        data = std::move(other.data);
        other.size = 0;
        other.data = nullptr;
        return *this;
    }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};
```

First change the calculation of the begin/end iterators

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, data);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) = delete;
    Contig & operator=(const Contig & other) = delete;
    Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {
        other.size = 0;
        other.data = nullptr;
    }
    Contig & operator=(Contig && other) {
        delete[] data;
        size = std::move(other.size);
        data = std::move(other.data);
        other.size = 0;
        other.data = nullptr;
        return *this;
    }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(seq.data, seq.data + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};
```

First change the calculation of the begin/end iterators

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, &data[0]);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) = delete;
    Contig & operator=(const Contig & other) = delete;
    Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {
        other.size = 0;
        other.data = nullptr;
    }
    Contig & operator=(Contig && other) {
        delete[] data;
        size = std::move(other.size);
        data = std::move(other.data);
        other.size = 0;
        other.data = nullptr;
        return *this;
    }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(&seq.data[0], &seq.data[0] + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};
```

```cpp
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, &data[0]);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) = delete;
    Contig & operator=(const Contig & other) = delete;
    Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {
        other.size = 0;
        other.data = nullptr;
    }
    Contig & operator=(Contig && other) {
        delete[] data;
        size = std::move(other.size);
        data = std::move(other.data);
        other.size = 0;
        other.data = nullptr;
        return *this;
    }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(&seq.data[0], &seq.data[0] + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, &data[0]);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) = delete;
    Contig & operator=(const Contig & other) = delete;
    Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {
        other.size = 0;
        other.data = nullptr;
    }
    Contig & operator=(Contig && other) {
        delete[] data;
        size = std::move(other.size);
        data = std::move(other.data);
        other.size = 0;
        other.data = nullptr;
        return *this;
    }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(&seq.data[0], &seq.data[0] + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};
```

Then change to a `std::unique_ptr` and therefore also no need for calling delete[] explicitly.

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, &data[0]);
    }
    ~Contig() { delete[] data; }
    Contig(const Contig & other) = delete;
    Contig & operator=(const Contig & other) = delete;
    Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {
        other.size = 0;
        other.data = nullptr;
    }
    Contig & operator=(Contig && other) {
        delete[] data;
        size = std::move(other.size);
        data = std::move(other.data);
        other.size = 0;
        other.data = nullptr;
        return *this;
    }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(&seq.data[0], &seq.data[0] + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    int * data;
};
```

Then change to a `std::unique_ptr` and therefore also no need for calling delete[] explicitly.

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, &data[0]);
    }
    ~Contig() {                }
    Contig(const Contig & other) = delete;
    Contig & operator=(const Contig & other) = delete;
    Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {
        other.size = 0;
        other.data = nullptr;
    }
    Contig & operator=(Contig && other) {

        size = std::move(other.size);
        data = std::move(other.data);
        other.size = 0;
        other.data = nullptr;
        return *this;
    }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(&seq.data[0], &seq.data[0] + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    std::unique_ptr<int[]> data;
};
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, &data[0]);
    }
    ~Contig() {}
    Contig(const Contig & other) = delete;
    Contig & operator=(const Contig & other) = delete;
    Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {
        other.size = 0;
        other.data = nullptr;
    }
    Contig & operator=(Contig && other) {
        size = std::move(other.size);
        data = std::move(other.data);
        other.size = 0;
        other.data = nullptr;
        return *this;
    }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(&seq.data[0], &seq.data[0] + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    std::unique_ptr<int[]> data;
};
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, &data[0]);
    }
    ~Contig() {}
    Contig(const Contig & other) = delete;
    Contig & operator=(const Contig & other) = delete;
    Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {
        other.size = 0;
        other.data = nullptr;
    }
    Contig & operator=(Contig && other) {
        size = std::move(other.size);
        data = std::move(other.data);
        other.size = 0;
        other.data = nullptr;
        return *this;
    }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(&seq.data[0], &seq.data[0] + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    std::unique_ptr<int[]> data;
};
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, &data[0]);
    }
    ~Contig() {}
    Contig(const Contig & other) = delete;
    Contig & operator=(const Contig & other) = delete;
    Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {
        other.size = 0;
        other.data = nullptr;
    }
    Contig & operator=(Contig && other) {
        size = std::move(other.size);
        data = std::move(other.data);
        other.size = 0;
        other.data = nullptr;
        return *this;
    }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(&seq.data[0], &seq.data[0] + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    std::unique_ptr<int[]> data;
};
```

And now, the default destructor is fine. Also the move members are now doing more or less the same as the default implicit move members will do, so we can simplify further.

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, &data[0]);
    }
    ~Contig() {}
    Contig(const Contig & other) = delete;
    Contig & operator=(const Contig & other) = delete;
    Contig(Contig && other) : size(std::move(other.size)), data(std::move(other.data)) {
        other.size = 0;
        other.data = nullptr;
    }
    Contig & operator=(Contig && other) {
        size = std::move(other.size);
        data = std::move(other.data);
        other.size = 0;
        other.data = nullptr;
        return *this;
    }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(&seq.data[0], &seq.data[0] + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    std::unique_ptr<int[]> data;
};
```

And now, the default destructor is fine. Also the move members are now doing more or less the same as the default implicit move members will do, so we can simplify further.

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, &data[0]);
    }
    ~Contig() {}
    Contig(const Contig & other) = delete;
    Contig & operator=(const Contig & other) = delete;
    Contig(Contig && other) = default;



    Contig & operator=(Contig && other) = default;






private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(&seq.data[0], &seq.data[0] + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    std::unique_ptr<int[]> data;
};
```

```cpp
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, &data[0]);
    }
    ~Contig() {}
    Contig(const Contig & other) = delete;
    Contig & operator=(const Contig & other) = delete;
    Contig(Contig && other) = default;



    Contig & operator=(Contig && other) = default;







private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(&seq.data[0], &seq.data[0] + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    std::unique_ptr<int[]> data;
};
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, &data[0]);
    }
    ~Contig() {}
    Contig(const Contig & other) = delete;
    Contig & operator=(const Contig & other) = delete;
    Contig(Contig && other) = default;



    Contig & operator=(Contig && other) = default;







private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(&seq.data[0], &seq.data[0] + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    std::unique_ptr<int[]> data;
};
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, &data[0]);
    }
    ~Contig() = default;
    Contig(const Contig & other) = delete;
    Contig & operator=(const Contig & other) = delete;
    Contig(Contig && other) = default;



    Contig & operator=(Contig && other) = default;







private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(&seq.data[0], &seq.data[0] + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    std::unique_ptr<int[]> data;
};
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, &data[0]);
    }
    ~Contig() = default;
    Contig(const Contig & other) = delete;
    Contig & operator=(const Contig & other) = delete;
    Contig(Contig && other) = default;



    Contig & operator=(Contig && other) = default;






private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(&seq.data[0], &seq.data[0] + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    std::unique_ptr<int[]> data;
};
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, &data[0]);
    }
    ~Contig() = default;
    Contig(const Contig & other) = delete;
    Contig & operator=(const Contig & other) = delete;
    Contig(Contig && other) = default;
    Contig & operator=(Contig && other) = default;
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(&seq.data[0], &seq.data[0] + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    std::unique_ptr<int[]> data;
};
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, &data[0]);
    }
    ~Contig() = default;
    Contig(const Contig & other) = delete;
    Contig & operator=(const Contig & other) = delete;
    Contig(Contig && other) = default;
    Contig & operator=(Contig && other) = default;
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(&seq.data[0], &seq.data[0] + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    std::unique_ptr<int[]> data;
};
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, &data[0]);
    }
    ~Contig() = default;
    Contig(const Contig & other) = delete;
    Contig & operator=(const Contig & other) = delete;
    Contig(Contig && other) = default;
    Contig & operator=(Contig && other) = default;
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(&seq.data[0], &seq.data[0] + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    std::unique_ptr<int[]> data;
};
```

Since we are using a `std::unique_ptr`, the copy members are disabled implicitly by default. So we can really just remove the whole thing if we want to, and we get exactly the behavior we expect and hope for.

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, &data[0]);
    }
    ~Contig() = default;
    Contig(const Contig & other) = delete;
    Contig & operator=(const Contig & other) = delete;
    Contig(Contig && other) = default;
    Contig & operator=(Contig && other) = default;
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(&seq.data[0], &seq.data[0] + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    std::unique_ptr<int[]> data;
};
```

Since we are using a `std::unique_ptr`, the copy members are disabled implicitly by default. So we can really just remove the whole thing if we want to, and we get exactly the behavior we expect and hope for.

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, &data[0]);
    }
    ~Contig() = default;
    Contig(const Contig & other) = delete;
    Contig & operator=(const Contig & other) = delete;
    Contig(Contig && other) = default;
    Contig & operator=(Contig && other) = default;
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(&seq.data[0], &seq.data[0] + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    std::unique_ptr<int[]> data;
};
```

Since we are using a `std::unique_ptr`, the copy members are disabled implicitly by default. So we can really just remove the whole thing if we want to, and we get exactly the behavior we expect and hope for.

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, &data[0]);
    }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(&seq.data[0], &seq.data[0] + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    std::unique_ptr<int[]> data;
};
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, &data[0]);
    }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(&seq.data[0], &seq.data[0] + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    std::unique_ptr<int[]> data;
};
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, &data[0]);
    }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(&seq.data[0], &seq.data[0] + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    std::unique_ptr<int[]> data;
};
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(new int[size]) {
        std::copy(str, str + size, &data[0]);
    }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(&seq.data[0], &seq.data[0] + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    std::unique_ptr<int[]> data;
};
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(std::make_unique<int[]>(size)) {
        std::copy(str, str + size, &data[0]);
    }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(&seq.data[0], &seq.data[0] + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    std::unique_ptr<int[]> data;
};
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(std::make_unique<int[]>(size)) {
        std::copy(str, str + size, &data[0]);
    }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(&seq.data[0], &seq.data[0] + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    std::unique_ptr<int[]> data;
};
```

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(std::make_unique<int[]>(size)) {
        std::copy(str, str + size, &data[0]);
    }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(&seq.data[0], &seq.data[0] + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    std::unique_ptr<int[]> data;
};
```

Or, perhaps you would like to use a standard container instead. Eg, `std::vector<int>`, then you might end up with something even simpler. Standard containers in modern C++ supports both copy and move semantics properly out of the box.

```
class Contig {
public:
    explicit Contig(const char * str) : size(std::strlen(str)), data(std::make_unique<int[]>(size)) {
        std::copy(str, str + size, &data[0]);
    }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(&seq.data[0], &seq.data[0] + seq.size, std::ostream_iterator<char>(out));
        return out;
    }
    std::size_t size;
    std::unique_ptr<int[]> data;
};
```

Or, perhaps you would like to use a standard container instead. Eg, `std::vector<int>`, then you might end up with something even simpler. Standard containers in modern C++ supports both copy and move semantics properly out of the box.

```
class Contig {
public:
    explicit Contig(const char * str) : data(str, str + std::strlen(str)) {

    }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(std::cbegin(seq.data), std::cend(seq.data), std::ostream_iterator<char>(out));
        return out;
    }

    std::vector<int> data;
};
```

```
class Contig {
public:
    explicit Contig(const char * str) : data(str, str + std::strlen(str)) {

    }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(std::cbegin(seq.data), std::cend(seq.data), std::ostream_iterator<char>(out));
        return out;
    }

    std::vector<int> data;
};
```

```
class Contig {
public:
    explicit Contig(const char * str) : data(str, str + std::strlen(str)) {}


private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(std::cbegin(seq.data), std::cend(seq.data), std::ostream_iterator<char>(out));
        return out;
    }
    std::vector<int> data;
};
```

```
class Contig {
public:
    explicit Contig(const char * str) : data(str, str + std::strlen(str)) {}


private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(std::cbegin(seq.data), std::cend(seq.data), std::ostream_iterator<char>(out));
        return out;
    }
    std::vector<int> data;
};
```

```
class Contig {
public:
    explicit Contig(const char * str) : data(str, str + std::strlen(str)) {}

private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(std::cbegin(seq.data), std::cend(seq.data), std::ostream_iterator<char>(out));
        return out;
    }
    std::vector<int> data;
};
```

```
class Contig {
public:
    explicit Contig(const char * str) : data(str, str + std::strlen(str)) {}
    auto begin() const { return std::cbegin(data); }
    auto end() const { return std::cend(data); }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(std::cbegin(seq), std::cend(seq), std::ostream_iterator<char>(out));
        return out;
    }
    std::vector<int> data;
};
```

```
class Contig {
public:
    explicit Contig(const char * str) : data(str, str + std::strlen(str)) {}
    auto begin() const { return std::cbegin(data); }
    auto end() const { return std::cend(data); }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(std::cbegin(seq), std::cend(seq), std::ostream_iterator<char>(out));
        return out;
    }
    std::vector<int> data;
};
```

```
class Contig {
public:
    explicit Contig(const char * str) : data(str, str + std::strlen(str)) {}
    auto begin() const { return std::cbegin(data); }
    auto end() const { return std::cend(data); }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(std::cbegin(seq), std::cend(seq), std::ostream_iterator<char>(out));
        return out;
    }
    std::vector<int> data;
};
```

```
class Contig {
public:
    explicit Contig(const char * str) : data(str, str + std::strlen(str)) {}
    auto begin() const { return std::cbegin(data); }
    auto end() const { return std::cend(data); }
private:
    friend std::ostream & operator<<(std::ostream & out, const Contig & seq) {
        std::copy(std::cbegin(seq), std::cend(seq), std::ostream_iterator<char>(out));
        return out;
    }
    std::vector<int> data;
};
```

```
class Contig {
public:
    explicit Contig(const char * str) : data(str, str + std::strlen(str)) {}
    auto begin() const { return std::cbegin(data); }
    auto end() const { return std::cend(data); }
private:
    std::vector<int> data;
};

std::ostream & operator<<(std::ostream & out, const Contig & seq) {
    std::copy(std::cbegin(seq), std::cend(seq), std::ostream_iterator<char>(out));
    return out;
}
```

```
class Contig {
public:
    explicit Contig(const char * str) : data(str, str + std::strlen(str)) {}
    auto begin() const { return std::cbegin(data); }
    auto end() const { return std::cend(data); }
private:
    std::vector<int> data;
};

std::ostream & operator<<(std::ostream & out, const Contig & seq) {
    std::copy(std::cbegin(seq), std::cend(seq), std::ostream_iterator<char>(out));
    return out;
}
```

```
class Contig {
public:
    explicit Contig(const char * str) : data(str, str + std::strlen(str)) {}
    auto begin() const { return std::cbegin(data); }
    auto end() const { return std::cend(data); }
private:
    std::vector<int> data;
};

std::ostream & operator<<(std::ostream & out, const Contig & seq) {
    std::copy(std::cbegin(seq), std::cend(seq), std::ostream_iterator<char>(out));
    return out;
}
```

And if you want to make sure that Contig objects can only be moved around (never copied), then you can add this:

```
class Contig {
public:
    explicit Contig(const char * str) : data(str, str + std::strlen(str)) {}
    auto begin() const { return std::cbegin(data); }
    auto end() const { return std::cend(data); }
private:
    std::vector<int> data;
};

std::ostream & operator<<(std::ostream & out, const Contig & seq) {
    std::copy(std::cbegin(seq), std::cend(seq), std::ostream_iterator<char>(out));
    return out;
}
```

And if you want to make sure that Contig objects can only be moved around (never copied), then you can add this:

```
class Contig {
public:
    explicit Contig(const char * str) : data(str, str + std::strlen(str)) {}
    Contig(Contig && other) = default;
    Contig & operator=(Contig && other) = default;
    auto begin() const { return std::cbegin(data); }
    auto end() const { return std::cend(data); }
private:
    std::vector<int> data;
};

std::ostream & operator<<(std::ostream & out, const Contig & seq) {
    std::copy(std::cbegin(seq), std::cend(seq), std::ostream_iterator<char>(out));
    return out;
}
```

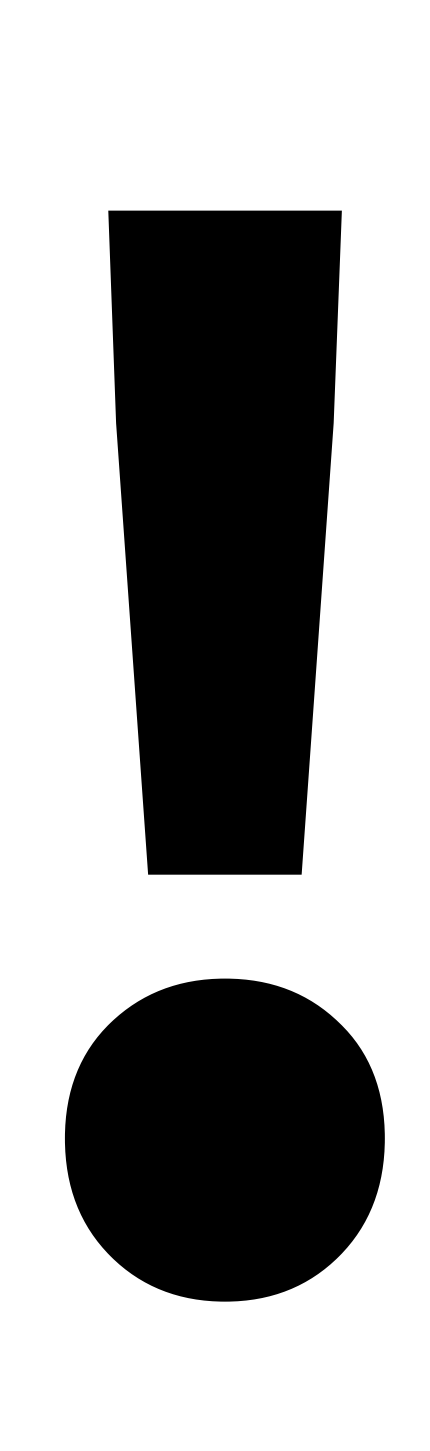
```
class Contig {
public:
    explicit Contig(const char * str) : data(str, str + std::strlen(str)) {}
    Contig(Contig && other) = default;
    Contig & operator=(Contig && other) = default;
    auto begin() const { return std::cbegin(data); }
    auto end() const { return std::cend(data); }
private:
    std::vector<int> data;
};

std::ostream & operator<<(std::ostream & out, const Contig & seq) {
    std::copy(std::cbegin(seq), std::cend(seq), std::ostream_iterator<char>(out));
    return out;
}
```

```
class Contig {
public:
    explicit Contig(const char * str) : data(str, str + std::strlen(str)) {}
    Contig(Contig && other) = default;
    Contig & operator=(Contig && other) = default;
    auto begin() const { return std::cbegin(data); }
    auto end() const { return std::cend(data); }
private:
    std::vector<int> data;
};

std::ostream & operator<<(std::ostream & out, const Contig & seq) {
    std::copy(std::cbegin(seq), std::cend(seq), std::ostream_iterator<char>(out));
    return out;
}
```

Nice!



mylib.cpp

```
#include "mylib.hpp"

#include <cstring>
#include <iostream>

namespace mylib {

Contig::Contig(const char * str)
    : data_(str, str + std::strlen(str))
{
}

Contig::const_iterator Contig::begin() const
{
    return std::cbegin(data_);
}

Contig::const_iterator Contig::end() const
{
    return std::cend(data_);
}

Contig::iterator Contig::begin()
{
    return std::begin(data_);
}

Contig::iterator Contig::end()
{
    return std::end(data_);
}

}

std::ostream & operator<<(
    std::ostream & out,
    const mylib::Contig & seq)
{
    for (char e : seq)
        out << e;
    return out;
}
```

mylib.hpp

```
#ifndef MYLIB_HPP_INCLUDED
#define MYLIB_HPP_INCLUDED

#include <iosfwd>
#include <vector>

namespace mylib {

class Contig {
public:
    using iterator = std::vector<int>::iterator;
    using const_iterator = std::vector<int>::const_iterator;
    explicit Contig(const char * str);
    Contig(Contig & other) = delete;
    Contig & operator=(const Contig & other) = delete;
    Contig(Contig && other) = default;
    Contig & operator=(Contig && other) = default;
    const_iterator begin() const;
    const_iterator end() const;
    iterator begin();
    iterator end();
private:
    std::vector<int> data_;
};

}

std::ostream & operator<<(
    std::ostream & out, const mylib::Contig & seq);

#endif
```

demo.cpp

```
#include "mylib.hpp"

#include <iostream>
#include <utility>
#include <vector>

namespace {

void analyze(mylib::Contig && seq)
{
    std::cout << seq << std::endl;
    // ...
    mylib::Contig sub("CTG");
    mylib::Contig pad("...");
    auto at = std::search(
        std::begin(seq), std::end(seq),
        std::cbegin(sub), std::cend(sub));
    if (at != std::end(seq))
        std::copy(std::cbegin(pad), std::cend(pad), at);
    std::cout << seq << std::endl;
    // ...
}

mylib::Contig extract()
{
    // ...
    return mylib::Contig("ACTTCTGTATTGGGTCTTTAATAG");
}

}

int main()
{
    analyze(extract());
}
```

```
$ c++ -c mylib.cpp
$ c++ -c demo.cpp
$ c++ -o demo mylib.o demo.o
$ ./demo
ACTTCTGTATTGGGTCTTTAATAG
ACTT...TATTGGGTCTTTAATAG
$
```

Example of a more complete Contig class

Special Members

compiler implicitly declares

user declares

	default constructor	destructor	copy constructor	copy assignment	move constructor	move assignment
Nothing	defaulted	defaulted	defaulted	defaulted	defaulted	defaulted
Any constructor	not declared	defaulted	defaulted	defaulted	defaulted	defaulted
default constructor	user declared	defaulted	defaulted	defaulted	defaulted	defaulted
destructor	defaulted	user declared	defaulted	defaulted	not declared	not declared
copy constructor	not declared	defaulted	user declared	defaulted	not declared	not declared
copy assignment	defaulted	defaulted	defaulted	user declared	not declared	not declared
move constructor	not declared	defaulted	deleted	deleted	user declared	not declared
move assignment	defaulted	defaulted	deleted	deleted	not declared	user declared

It is hard-ish to remember when and what a compiler declares implicitly or not. And once you know it by heart, you can be sure that one of your damn colleagues has forgotten it. Probably a good idea to be explicit and verbose when doing these things.

At the ACCU 2014 keynote, I seem to recall that Howard Hinnant admitted that he kept a copy of this particular table next to his workstation. (Howard was the lead designer and implementor of rvalue references and move semantics in the C++11 standard.)

https://www.slideshare.net/ripplelabs/howard-hinnant-accu2014

6.10 Lvalues and rvalues [basic.lval]

1 Expressions are categorized according to the taxonomy in Figure 1.

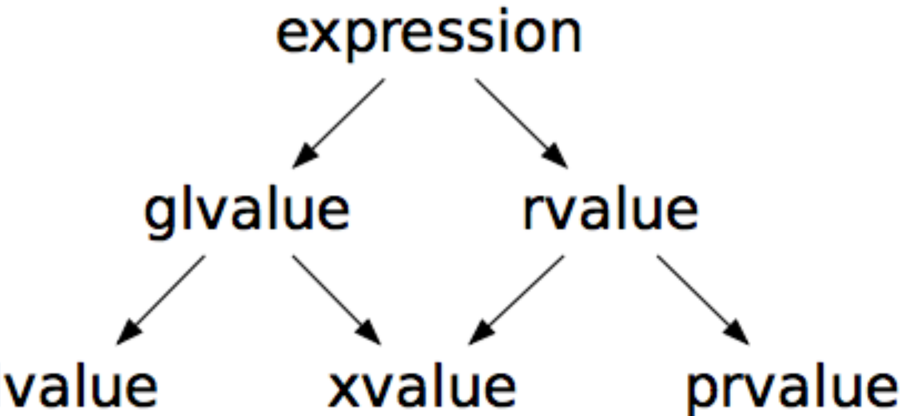


Figure 1 — Expression category taxonomy

(1.1) — A *glvalue* is an expression whose evaluation determines the identity of an object, bit-field, or function.

(1.2) — A *prvalue* is an expression whose evaluation initializes an object or a bit-field, or computes the value of the operand of an operator, as specified by the context in which it appears.

(1.3) — An *xvalue* is a glvalue that denotes an object or bit-field whose resources can be reused (usually because it is near the end of its lifetime). [*Example:* Certain kinds of expressions involving rvalue references (11.3.2) yield xvalues, such as a call to a function whose return type is an rvalue reference or a cast to an rvalue reference type. — *end example*]

(1.4) — An *lvalue* is a glvalue that is not an xvalue.

(1.5) — An *rvalue* is a prvalue or an xvalue.

Historically, lvalues and rvalues were so-called because they could appear on the left- and right-hand side of an assignment (although this is no longer generally true); glvalues are "generalized" lvalues, prvalues are "pure" rvalues, and xvalues are "eXpiring" lvalues. Despite their names, these terms classify expressions, not values.

“Every block of stone has a statue inside it and it is the task of the sculptor to discover it.” Michelangelo

“As a programmer, your main job is to reduce flexibility and options in a computer system until the point that it becomes useful and valuable for others.” Olve