The Big-5

Rob Hackman and Xiao-bo Li

Winter 2024

University of Alberta

Table of Contents

Porting our List ADT to C++

Let's implement our Linked List ADT in C++.

```
class List {
  struct Node {
    int data;
    Node *next;
  }
  int len;
  Node *head;
 public:
  List() : len{0}, head{nullptr} {}
  List &addToFront(int n) {
    head = new Node{n, head};
    ++len;
    return *this;
  }
  void setIth(int i, int val) {...}
  void getIth(int i) {...}
};
```

Nested classes

In the code above `Node` is a *nested class* within class `List`.

Since the class `Node` only exists for the implementation of `List` we put its definition inside of `List`, we've also defined it within the private members of `List`, this means that only `List` methods can create and use the `Node` type.

It also means the *fully qualified* when referring to this type is `List::Node`, as it is the type `Node` defined *within* the `List` scope. We'll need that to implement `Node` methods outside of the class definition.

Using our `List` ADT — problem one

So, we try to use our `List` ADT, but we notice a problem with the following program:

```
int main() {
  List l;
  l.addToFront(1).addToFront(2).addToFront(3);
}
```

It leaks memory! But how? We didn't allocate any memory in this code!

Yes, we did — the code for `addToFront` allocates new `Node` objects to add to our `List`, but we never free these! Also we, the client, cannot free them as we can't access the `head` field of a `List`, it's private!

How to free resources inside an object

The client can't free the resources of a `List` when they're done with it as they can't access the fields. Nor should the client have to! The point of an ADT is that it abstracts away the details.

In C we solved this by providing a function to delete our `List` and everything it contained. Such a solution can work, but it's prone to errors as clients may misuse the method or forget to use it at all. We have a better solution in `C++`

Table of Contents

Destructors

In C++ each class has a special method called the `destructor` which runs automatically when objects of that type are destroyed. For statically-allocated objects that is when the lifetime of their enclosing scope ends, and for dynamically-allocated objects that is when the programmer frees them with `delete`.

Destructors are methods of a class, much like constructors do not have names and are called implicitly the same is true of the destructor. We denote the destructor with the class name preceeded by the tilde (˜) character.

Phases of Object Destruction

Just like the phases of object creation there are phases of object destruction.

1. Destructor body runs
2. Fields that are objects are destroyed
3. Space is deallocated

List **destructor**

We attempt to write our List destructor like so, which compiles just fine.

```
class List {
  ...
public:
  ...
  ~List() {
    delete head;
  }
};
```

List destructor

We attempt to write our List destructor like so, which compiles just fine.

However, this code still leaks! How come?

```
class List {
  ...
public:
  ...
  ~List() {
    delete head;
  }
};
```

List **destructor**

We attempt to write our List destructor like so, which compiles just fine.

However, this code still leaks! How come?

Our destructor only frees the *first* Node, all Nodes after that our leaked. How to solve?

```
class List {
  ...
public:
  ...
  ~List() {
    delete head;
  }
};
```

`List` destructor

However, this code still leaks! How come?

Our destructor only frees the *first* `Node`, all `Node`s after that our leaked. How to solve?

We could write the code to iterate through the `Node`s deleting them inside the `List` destructor, or we could recognize that `Node` itself is a class, and can have it's own destructor!

```cpp
class List {
    ...
public:
    ...
    ~List() {
        delete head;
    }
};
```

`Node` destructor

Is this correct? It only calls deletes on the `Node` immediately after this one, won't that only get the second `Node` in the list?

```
class List {
  struct Node {
    ...
    ~Node() {
      delete next;
    }
  };
  ...
};
```

`Node` destructor

It is correct! Remember, an object's `destructor` gets called when it is destroyed.

Since our next `Node` is an object when we delete it *its* destructor also runs which deletes the node after it — so on and so forth until our last node whose destructor calls `delete` on the null pointer which is a no-op in C++

```
class List {
  struct Node {
    ...
    ~Node() {
      delete next;
    }
  };
  ...
};
```

Table of Contents

Another problem with List

We continue to use our List ADT and write a program like the following, only to notice a problem.

```
int main() {
  List l1;
  l2.addToFront(1).addToFront(2).addToFront(3);
  List l2 = l1;
  l2.setIth(0,10);
  cout << l1 << endl;
  cout << l2 << endl;
}
```

In this program the change to l2 has also affected l1 — probably not what we intended. To make matters worse our program crashes with a double free error, why?

Copy Constructor

When an object A is initialized with the value of another object B of the same type the *copy constructor* is used to construct object A. Every class comes with a built-in copy constructor which simply copy-initializes all fields.

Since we have not implemented a copy constructor the built-in copy constructor is being used, which is simply copying each field. This means `l2` has a copy of the same pointer as `l1` — they're *shallow copies.* This means mutating the nodes of one will affect the other, as they're pointers to the same `Node` objects. Additionally, this is causing our double free errors as the destructors of both `Lists` are trying to free the same `Nodes`!

Writing your own Copy Constructor

For most types that manage resources we do not want a shallow copy, so we must write our own copy constructor. We will do the same thing we did for our destructor and implement both `List's` copy constructor and `Node's` copy constructor to help.

The copy constructor is the constructor that takes *one* parameter, and the type of that parameter is an lvalue reference to the same type. The parameter *must* be a reference, it cannot be pass-by-value!

Why? How would the compiler initialize the local paramater copy?

Copy Constructor `List` implementation

```cpp
class List {
  struct Node {
    ...
    Node(const Node &o) : data{o.data}
      head{o.head ? new Node{*o.head} : nullptr} {}
  };
  ...
 public:
  ...
  List(const List &o) : len{o.len},
    head{o.head ? new Node{*o.head} : nullptr} {}
};
```

Copy Constructor `List` implementation

Just like our destructor our copy constructor is using recursion for its implementation.

Not all destructors and copy constructors will be written recursively, a linked list is just a recursive data type, so it makes sense to do so in this case.

When is the copy constructor called?

The copy constructor is called whenever an object of a given type is initialized with an object of the same type. The main times this happens are:

- When directly initializing an object with another of the same type (example above)
- When passing an object into a function by value (to initialize the parameter)
- When returning an object by value

There are exceptions to all three of these which we will discuss shortly.

Copy constructor — result

```
int main() {
  List l1;
  l2.addToFront(1).addToFront(2).addToFront(3);
  List l2 = l1;
  l2.setIth(0,10);
  cout << l1 << endl;
  cout << l2 << endl;
}
```

Our code above now runs as we expect! `l2` becomes a *deep copy* of `l1` and changes to `l2` do not affect `l1` as they do not share any pointers. Our program also no longer crashes as the destructors for `l1` and `l2` are not trying to free the same list anymore.

Table of Contents

Another problem...

After working with our `List` ADT a little bit longer we observe a new problem:

```
int main() {
  List l1;
  l1.addToFront(1).addToFront(2).addToFront(3);
  List l2; // Default ctor
  l2.addToFront(5).addToFront(6);
  l2 = l1; // Not a construction... what is it?
  l2.setIth(0, 10);
  cout << l1 << endl;
  cout << l2 << endl;
}
```

This code once again exhibits aliasing between `l1` and `l2`, and the double free error is back! Why is this a shallow copy?

Reminder - initalization only happens once!

A quick reminder, initialization of an object only happens *once* when it is created, you can't "initialize" something after it's already been created — that's just changing it's value! Constructors are only called when objects are first created and initialized.

```
List l2; // Default ctor
l2 = l1; // Not a construction... what is it?
```

In these lines from our `main` function on the previous slide the variable `l2` is not being copy constructed from `l1`. The variable `l2` is default constructed and then *assigned* to the variable `l1`.

The assignment operator is just that, however, an operator. As an operator it can be overloaded!

Signature of the copy assignment operator

We call the assignment operator when overloaded between two objects of the same type the *copy assignment operator (CAO)*.

Since both operands, most importantly the first operand, are of class `List` we can overload the operator as a method and don't have to make it an external `friend` function.

What should the type signature be? As it's a member function the first operand is implicitly the object being assigned itself (pointed at by the `this` parameter). The second operand is the object we'd like to copy, so it should be a `const` lvalue reference to a `List`. What about the return type?

Return type of assignment

You've likely used the return value of an assignment expression without thinking about it before, consider the following code:

```
int x = 1, y = 2, z = 3;
x = y = z;
```

What do you expect to be the values of `x`, `y`, and `z` after line 2 runs? You may know from experience or intuition they should all be 3, but what is the real behaviour of the language here? Line 2 above is equivalent to:

```
x = (y = z);
```

`x` is really being assigned to the result of evaluating the expression `y = z`. So what should assignment return? The agreed upon convention is *the value that was assigned*, so in the expression `y = z` it should return a reference to `y`.

Overloading the copy assignment operator — attempt 1

Now that we know the signature of our CAO we can overload it.

```
class List {
  ...
  public:
  ...
  List &operator=(const List &o) {
    theHead = o.theHead ? new Node{*o.theHead} : nullptr;
    len = o.len;
    return *this; // Return the object assigned.
    // Must dereference this (*this)
    // otherwise we're returning a pointer
    // which is not the object itself!
  }
};
```

Overloading the copy assignment operator — problem 1

There are several problems with our first attempt at the CAO. First, we compile our program (with the `main`) from earlier) and run it. We see that the aliasing no longer occurs and the double free error is gone! We may assume everything is working!

Overloading the copy assignment operator — problem 1

There are several problems with our first attempt at the CAO. First, we compile our program (with the `main`) from earlier) and run it. We see that the aliasing no longer occurs and the double free error is gone! We may assume everything is working!

However, upon running our program through valgrind we see that there are memory leaks... how come?

Overloading the copy assignment operator — problem 1

There are several problems with our first attempt at the CAO. First, we compile our program (with the `main`) from earlier) and run it. We see that the aliasing no longer occurs and the double free error is gone! We may assume everything is working!

However, upon running our program through valgrind we see that there are memory leaks... how come?

We forgot, when implementing the copy assignment operator, that the object we were assigning to is not a new object being constructed. It already exists, and as such may have it's own memory already allocated.

Overloading the copy assignment operator — problem 1

However, upon running our program through valgrind we see that there are memory leaks... how come?

We forgot, when implementing the copy assignment operator, that the object we were assigning to is not a new object being constructed. It already exists, and as such may have it's own memory already allocated.

On line 6 of the code showing our CAO we overwrote our `theHead` pointer with a new pointer without considering that our pointer may already point at memory — this is a memory leak! We must also free our object's old memory.

Overloading the copy assignment operator — attempt 2

Now that we know the signature of our CAO we can overload it.

```
class List {
  ...
  public:
  ...
  List &operator=(const List &o) {
    delete theHead;
    theHead = o.theHead ? new Node{*o.theHead} : nullptr;
    len = o.len;
    return *this;
  }
};
```

Overloading the copy assignment operator — problem 2

We try our new assignment operator with the `main` shown and it seems the problems are fixed: there is no aliasing, no double free error due to the aliasing, and no memory leak! However, what if we try our ADT with the following `main`

```
1   int main() {
2     List l;
3     l.addToFront(3).addToFront(2).addToFront(1);
4     cout << l << endl;
5     l = l;
6     cout << l << endl;
7   }
```

On line 6 it prints out garbage data! What happened?

The self assignment problem

The previous main prints out garbage values when trying to print out `l` the second time because we had just assigned `l` to itself. Let's consider what happens in our CAO when `*this` and `o` are the same object

```
1   List &List::operator=(const List &o) {
2     delete theHead;
3     theHead = o.theHead ? new Node{*o.theHead} : nullptr;
4     len = o.len;
5     return *this;
6   }
```

On line 2 we delete `theHead` which deletes the entire linked list of nodes, then on line 3 we immediately try and copy construct that whole list we just deleted since `o.theHead` is the same as `this->theHead` in a self assignment scenario. So we're accessing a dangling pointer, how to solve this problem?

Self assignment — do we care?

Why would a programmer ever assign an object to itself, is this really something worth concerning ourselves with?

Yes! In complex programs there will often be references to objects or pointers to objects produced by functions or other mechanisms. So the programmer may not know they're doing a self assignment. Even in simple programs it could occur.

Accidental self assignment example

Consider Foo in the following example is a complex ADT that manages resources

```
Foo &max(Foo *arr, size_t len) {
  size_t maxInd = 0;
  for (size_t i = 0; i < len; ++i) {
    if (arr[i] > arr[maxInd]) maxInd = i;
  }
  return arr[maxInd];
}

int main() {
  Foo arr[10] = ...;
  // If first item is also the max
  // this is self assignment!
  arr[0] = max(arr, 10);
}
```

The self assignment problem — one solution

If the client asks for self assignment one possible solution is to simply do nothing, as nothing needs to be done to assign an object to itself.

How do we write the code that does nothing if self assignment is occurring? More specifically how do we check if self assignment is occurring to know we should do nothing?

How do we know when two objects are really the *same* object and don't just store the same value?

The self assignment problem — one solution

If the client asks for self assignment one possible solution is to simply do nothing, as nothing needs to be done to assign an object to itself.

How do we write the code that does nothing if self assignment is occurring? More specifically how do we check if self assignment is occurring to know we should do nothing?

How do we know when two objects are really the *same* object and don't just store the same value?

Observation: two objects are the exact same object if they exist at the same place in memory!

Overloading the copy assignment operator — attempt 3

Now that we know the signature of our CAO we can overload it.

```
class List {
  ...
  public:
  ...
  List &operator=(const List &o) {
    if (this == &o) return *this;
    delete theHead;
    theHead = o.theHead ? new Node{*o.theHead} : nullptr;
    len = o.len;
    return *this;
  }
};
```

Overloading the copy assignment operator — problem 3

Our new CAO now solves the self assignment problem by simply doing nothing when self assignment occurs. There is another problem with our CAO — though one we won't see often and can't replicate easily.

The `new` operator can fail (as could `malloc`). If the `new` operator fails and we can't copy our `List` we'd prefer that the assignment left our object unchanged, rather than with deleted memory[1]. Fix? Only delete our old memory after we know `new` succeeded.

[1]This ties in to the important concept of *exceptions* and *exception safety* in C++ which unfortunately we won't cover in this class. A good C++ programmer must be aware of this though!

Overloading the copy assignment operator — attempt 4

Here is a final, and working, copy of our CAO.

```
class List {
  ...
  public:
  ...
  List &operator=(const List &o) {
    if (this == &o) return *this;
    Node *oldHead = theHead;
    theHead = o.theHead ? new Node{*o.theHead} : nullptr;
    delete oldHead;
    len = o.len;
    return *this;
  }
};
```

Copying is copying... why double work?

The work to make a copy in our CAO is not hard, since we're just using `Node's` copy constructor which works recursively. For some ADTs the copy code might be more complex.

But... we already wrote the code to make a copy of our object in our copy constructor. Why write the code twice? Code duplication is after all a code smell.

Suggestion: use your copy constructor to implement your CAO using the *copy-and-swap idiom*

Copy-and-swap idiom

The copy-and-swap idiom is a simple idiom for implementing copy assignment operators. The idiom is given that you've already implemented your copy constructor your CAO can be implemented easily by providing a `swap` method, and simply swapping with a copy.

The `swap` method should be a private helper method that swaps all the data of your ADT with the data of another object of the same type.

Copy-and-swap idiom implementation

```
#include <utility>
class List {
  ...
  void swap(List &o) {
    std::swap(theHead, o.theHead);
    std::swap(len, o.len);
  }
  public:
  ...
  List &operator=(const List &o) {
    List cpy{o};
    swap(cpy);
    return *this;
  }
};
```

Copy-and-swap idiom implementation notes

- In the `<utility>` the function `std::swap` is provided which can be used for swapping built-in types, we use it to swap our pointer and int fields
- Note that since `swap` is a method we still have the implicit `this` parameter, so we're swapping the fields of the object pointed at by `this` with the fields of thge parameter `o`
- In the code for our CAO we just wrote `swap(cpy)` — what is the object being swapped with?
    - Just like accessing fields within a member function accessing methods also uses the implied `this`
    - So in the CAO the code `swap(cpy)` is equivalent to `this->swap(cpy)`.

Copy-and-swap idiom implementation niceties

The copy and swap idiom implicitly solves all the problems we talked about with our earlier CAO implementations!

1. How do we free our old memory? It's swapped into the local variable cpy which goes out of scope when the function returns... so it gets destroyed and its destructor frees our old data for us

Copy-and-swap idiom implementation niceties

The copy and swap idiom implicitly solves all the problems we talked about with our earlier CAO implementations!

2. How do we handle self assignment? Since we make a full copy first and swap with it even if we self assign we haven't destroyed our data.

Copy-and-swap idiom implementation niceties

The copy and swap idiom implicitly solves all the problems we talked about with our earlier CAO implementations!

2. How do we handle self assignment? Since we make a full copy first and swap with it even if we self assign we haven't destroyed our data.
    - Some may take offense to the fact that we're wasting time making a deep copy of ourselves in self assignment, and that's a reasonable complaint! You can still add the self assignment check to this CAO to do no work when self assignment occurs.

Copy-and-swap idiom implementation niceties

The copy and swap idiom implicitly solves all the problems we talked about with our earlier CAO implementations!

3. How do we handle `new` failing? If `new` fails it is when we're creating the local copy, before we've made any changes to our object. So if `new` fails we will not leave our object in an invalid state (like having a dangling pointer).

Copy-and-swap idiom implementation niceties

The copy and swap idiom implicitly solves all the problems we talked about with our earlier CAO implementations!

3. How do we handle `new` failing? If `new` fails it is when we're creating the local copy, before we've made any changes to our object. So if `new` fails we will not leave our object in an invalid state (like having a dangling pointer).
    - If our swap fails halfway through then we might still have an invalid object (an object that doesn't follow its own invariants). Again this is a problem to consider for exception safety — the solution is to make sure `swap` can never fail which can be achieved easily by following the *Pointer to Implementation (pImpl) idiom* — look it up if you're curious!

Table of Contents

An efficiency problem

Our `List` ADT now *works*, but it still has flaws that could be improved. For example consider the following program:

```
List inc(List l) {
  for (int i = 0; i < l.length(); ++i) {
    l.setIth(i, l.getIth(i) + 1);
  }
  return l;
}

int main() {
  List l1;
  l1.addToFront(1).addToFront(2).addToFront(3);
  List l2{inc(l1)};
}
```

Add a print statement to our `List` copy constructor, how many copy constructions do we see?

Where did the copies come from?

Compiling the above program we see two copy constructor calls — but are they both necessary?

The first copy constructor call is occurring when we're initializing the parameter `l` of the function `inc` when we pass the value of `l1` as an argument. This copy construction is necessary, we must construct the parameters value and we *want* it to be a copy not a reference as we don't want to mutate the given argument.

The second copy constructor call is when the value of `l` is returned to the caller and used to construct `l2`. Does this make sense though?

Unnecessary copies

When a function returns we know all of its local varaibles go out of scope and are popped off the stack and destroyed, for objects this means their destructor runs.

So when `inc` returns `l` for us to copy construct `l2` with `l` is immediately going to be destroyed anyways.

We wrote our copy constructor (and CAO) to do deep copies so that our objects aren't shallow copies of eachother, but when `inc` returns `l` is about to be destroyed anyways — is there any reason to deep copy all that data just to destroy the original copy?

Unnecessary copies

When a function returns we know all of its local varaibles go out of scope and are popped off the stack and destroyed, for objects this means their destructor runs.

So when `inc` returns `l` for us to copy construct `l2` with `l` is immediately going to be destroyed anyways.

We wrote our copy constructor (and CAO) to do deep copies so that our objects aren't shallow copies of eachother, but when `inc` returns `l` is about to be destroyed anyways — is there any reason to deep copy all that data just to destroy the original copy?

Not really! In fact it seems quite wasteful to deep copy an entire list and then throw out the original!

lvalues and rvalues

This is where the concept of *rvalues* becomes helfpul.

We already seen a basic concept of this, we've said all our references so far have been lvalue references, and observed if we had a function that took a non-const lvalue reference parameter we could not pass it an argument of a literal or the result of an expression that calculated a new value (see slides 15-17 of references slides). That was because those arguments are rvalues.

So what is an lvalue and an rvalue?

lvalues and rvalues — what are they?

Originally lvalues and rvalues actually did mean “left-values” and “right-values” to mean “values that can appear on the left hand side of an assignment operator” and “values that can **only** appear on the right hand side of the assignment operator”, for example:

```
1    int x;
2    int arr[10];
3    x = 2; // fine
4    5 = 10; // ???
5    arr[3] = 1; // fine
6    x+3 = 10; // ???
```

The simple concept is shown in the examples above, the assignments on line 3 and 5 make sense and work, the assignments on line 4 and 6 are nonsensical (what’s even being assigned to?) and do not compile.

lvalues and rvalues — not just “left” and “right”

The simplification of lvalues and rvalues described in the previous slide does not fully capture the concept, and has problems — so lvalue and rvalue do not actually stand for this! In fact C++ has more value categories than just lvalue and rvalue but we’ll just discuss these.

The simplest counterpoint to the “left” and “right” definitions is that a `const` variable cannot appear on the left-hand side of an assignment, but is absolutely an lvalue.

It is better to say an rvalue is a temporary value and and an lvalue is anything not temporary.

lvalues and rvalues described

The formal definition is very long, so we'll do an informal description that doesn't match the exact details. If you're curious about the details read up about value categories in the C++ standard.

- an lvalue (a non-temporary)
  - Anything with an address (can be read with the address of operator)
  - Has a lifetime longer than the end of the expression its used in or calculated from
- an rvalue (a temporary)
  - Does not necessarily have an address (cannot be read with address of operator)
  - Has a lifetime only as long as the expression in which it is used or calculated from

Returnining local data by value

We've known since C that returning by value means copying data to the callers stack frame, if the value being returned is the value of data local to that function then its copying data from the functions stackframe to the callers.

Consider our parameter `l` in our `inc` function — it's not an rvalue, it exists for longer than one expression and it has an address we can read. However, when our function is returning `l` its lifetime is only until the end of that statement, as once the `return` occurs the variable is popped off the stack.

So, when a function is returning a local variable by value C++ considers this to be returning an rvalue — how can we use this to our advantage?

rvalue references

When we first learned about references it was mentioned that you cannot take the reference of a reference, e.g. `int &&`. It was also mentioned that syntax was valid but meant something different in C++ — it means an *rvalue reference*.

The double ampersands in a type indicate an rvalue reference given this information we can implement a constructor which takes an rvalue reference instead of an lvalue reference like the copy constructor. We call this constructor which has a single parameter that is an rvalue reference to an objet of the same type the *move constructor*.

The Move Constructor

```
class List {
  public:
  ...
  List(List &&o) : theHead{o.theHead}, len{o.len} {
    o.theHead = nullptr;
    o.len = 0;
  }
};
```

Here we have defined the move constructor. In the implementation of the move constructor since our parameter is a temporary we know it is safe to simply *steal* their data rather than fully deep copy it.

The Move Constructor notes

In our move constructor we simply copied the pointer our parameter `o` had, rather than deep copying the list. Since `o` is a reference to a temporary that temporary is about to be destroyed, so why bother deep copying all its data?

Important — since we know our parameter is a temporary and about to be destroyed we must also be careful it doesn't destroy the data we just stole along with itself! This is why in our move constructor body we set `o.theHead` to `nullptr` to make sure when the temporary is destroyed its destructor doesn't delete the pointer we just stole.

The Move Constructor notes

It's not exactly necessary that we set `o.len` to 0, but the rule of thumb is that *moved from* object should be left in a *valid but unspecified state*.

What that means is that clients cannot rely on that temporary having a specific value after we've move constructed an object with it, however it should be left in a logically consistent state that doesn't violate any of the class invariants, since if it violated the class invariants it wouldn't be in a *valid* state.

So, since we've removed all items from the `List` by setting its `theHead` pointer to null we also set its `len` to 0 so the invariant that the `len` variable always represents the length of the list still holds.

When is the move constructor called?

The move constructor is called whenever an object is initialized with an rvalue of the same type, the compiler is able to differentiate between rvalues and lvalues and thus calls the copy constructor or move constructor appropriately based on the arguments.

The most common time we have rvalues is when we have functions or operators that return an object by value, the value produced by those functions and operators will be an rvalue in the expression in which its used.

The move constructor benefits

After implementing our move constructor if we run the following program we see one copy construction and one move construction (add print statements to move/copy ctors to see this).

```
List inc(List l) {
  for (int i = 0; i < l.length(); ++i) {
    l.setIth(i, l.getIth(i) + 1);
  }
  return l;
}

int main() {
  List l1;
  l1.addToFront(1).addToFront(2).addToFront(3);
  List l2{inc(l1)};
}
```

Assignment and rvalues

However, we must also be consider the case where we *assign* an object to an rvalue object of the same type.

```
List inc(List l) {
  for (int i = 0; i < l.length(); ++i) {
    l.setIth(i, l.getIth(i) + 1);
  }
  return l;
}

int main() {
  List l1;
  l1.addToFront(1).addToFront(2).addToFront(3);
  List l2;
  l2.addToFront(7).addToFront(8);
  l2 = inc(l1); // Assignment operator!
}
```

The Move assignment operator

In the code above the copy assignment operator is called to construct `l2` from the result of the `inc` function — once again this is a deep copy that is unnecessary! What can we do?

The Move assignment operator

In the code above the copy assignment operator is called to construct `l2` from the result of the `inc` function — once again this is a deep copy that is unnecessary! What can we do?

Just like writing the move constructor to handle the situation where our objects are constructed with rvalues of the same type we can write the *move assignment operator (MAO)* to be executed when our objects are assigned to rvalues of the same type.

The Move assignment operator

In the code above the copy assignment operator is called to construct `l2` from the result of the `inc` function — once again this is a deep copy that is unnecessary! What can we do?

Just like writing the move constructor to handle the situation where our objects are constructed with rvalues of the same type we can write the *move assignment operator (MAO)* to be executed when our objects are assigned to rvalues of the same type.

The move assignemnt operator is the overloaded assignment operator where the right-hand operand is an rvalue reference to the same type as the left-hand operand.

`List` move assignment operator

```cpp
class List {
  public:
  ...
  List &operator=(List &&o) {
    swap(o);
    return *this;
  }
};
```

Here is the MAO for our `List` class — notice that we simply *swap* our existing data with the temporary — why?

Move assignment operator swap implementation

Reusing the `swap` we wrote for our copy-and-swap CAO is a common and neat solution to implementing our MAO.

- If we simply steal our parameter's data and null it out like in our move constructor, then we still must remember to free our old data
- We already wrote the code to free all our resources, in the destructor. The temporary is going to be destroyed anyways... why not give it our old data to clean up for us?
- We already wrote the swap function for our copy-and-swap CAO, it elegantly solves all the problems our MAO must solve — might as well use it!

Table of Contents

Constructor calls — when do they happen?

Consider class `Foo` that has a constructor that takes one integer as well as a copy constructor, move constructor, CAO, and MAO all of which print out when they're called. Given that `Foo` consider the following code:

```
 Foo makeFoo(int n) {
  Foo ret{n % 2 ? n*3 +1 : n/2};
  return ret;
}

int main() {
  Foo f{makeFoo(5)};
}
```

Which constructors do you expect to see called?

A note on constructor calls

We might think the above program prints out that the int parameter constructor is called (to construct `ret` on the functions stack frame), and that the move constructor is called to construct `f` from the return value.

A note on constructor calls

We might think the above program prints out that the int parameter constructor is called (to construct `ret` on the functions stack frame), and that the move constructor is called to construct `f` from the return value.

Upon compiling and running the code we see that only the int parameter constructor is called... what happened? How is `f` initialized?

Copy/Move elision

This is an exampe of move/copy *elision*. An optimization where the compiler is allowed to omit constructor calls (even if they have side effects like printing!) when able. In C++17 and above the compiler is even *required* to do so in some cases.

How does `f` get constructed? The compiler simply constructs `ret` directly in the memory of object `f`, instead of constructing it on the functions stack frame, skipping the need to move it out of the functions stackframe.

Stopping Copy/Move elision

To turn off Copy/Move elision you can compile your program with the flag `-fno-elide-constructors`, if we compile the above program with this flag what do we see?

Stopping Copy/Move elision

To turn off Copy/Move elision you can compile your program with the flag `-fno-elide-constructors`, if we compile the above program with this flag what do we see?

We see the
int parameter constructor and two move constructors called — why?

Stopping Copy/Move elision

To turn off Copy/Move elision you can compile your program with the flag `-fno-elide-constructors`, if we compile the above program with this flag what do we see?

We see the int parameter constructor and two move constructors called — why?

If we go back and compile our previous example for `List` with the `-fno-elide-constructors` we see the same! So elision was happening even then!

Copies/Moves without elision

When the function `makeFoo` returns `ret` without any optimization there are actually two objects that need to be constructed (and both are move constructed).

First, the “return value” of the function must be created on the caller’s stack frame — this is the value that the function call expression evalutes to. This value is move constructed from `ret` since at that point `ret` is an rvalue.

Second, the variable `f` must be constructed with this return value — this is the second move construction

In this example the compiler is able to elide both constructors by constructing the value directly into `ret’s` memory.

Return value optimization (RVO)

In our `List` example when we constructed `l2` with the result of `inc` we still saw one move construction — which one was it?

It was the second one mentioned in the previous slide, that is the actual construction of `l2`. Whenever possible the compiler will skip the step of creating the temporary “return value” object that just represents the evaluation of a function call. This is called *return value optimization* and is a specific case of elision.

When can elision occur?

The exact details of when elision can occur are very detailed, we do not expect you to know *when* exactly elision can occur only that it *can* occur.

Table of Contents

Closing comments

We've now learned about the destructor, the copy constructor, the move constructor, the copy assignment operator, and the move assignment operator. Collectively these are called *the big 5* methods of a class.

The general rule of thumb involving these methods is the "Rule of 5" that states *If you need any one of the big 5, you probably need all 5*. This rule should be followed!

When do you need one of the big 5? Typically anytime your class contains/manages resources such as memory, network sockets, files, etc. In this course we really only consider memory. More generally you must make that decision for yourself!