

TDDD38/726G82:

Adv. Programming in C++

Advanced Constructs I

Christoffer Holm

Department of Computer and information science

- 1 Forwarding references
- 2 Smart pointers
- 3 `std::unique_ptr`
- 4 `std::shared_ptr`

- 1 Forwarding references
- 2 Smart pointers
- 3 `std::unique_ptr`
- 4 `std::shared_ptr`

Forwarding references

Opportunity for optimization

```
1  template <typename T>
2  void add(vector<T>& v,
3          T value)
4  {
5      v.push_back(value);
6  }
```

```
1  vector<string> w;
2  add(w, "hello"s);
3
4  string s { "bye" };
5  add(w, s);
```

Forwarding references

Opportunity for optimization

```
1 template <typename T>
2 void add(vector<T>& v,
3         T value)
4 {
5     v.push_back(value);
6 }
```

```
1 vector<string> w;
2 add(w, "hello"); create!
3
4 string s { "bye" };
5 add(w, s);
```

Forwarding references

Opportunity for optimization

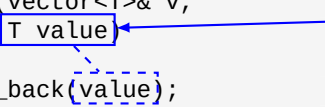
```
1 template <typename T>
2 void add(vector<T>& v,
3         copy! T value)
4 {
5     v.push_back(value);
6 }
```

```
1 vector<string> w;
2 add(w, "hello"); create!
3
4 string s { "bye" };
5 add(w, s);
```

Forwarding references

Opportunity for optimization

```
1 template <typename T>
2 void add(vector<T>& v,
3         copy! T value)
4 {
5     v.push_back(value);
6 }
```



```
1 vector<string> w;
2 add(w, "hello"); create!
3
4 string s { "bye" };
5 add(w, s);
```

Forwarding references

Opportunity for optimization

```

1  template <typename T>
2  void add(vector<T>& v,
3         copy! T value)
4  {
5  [v] push_back(value);
6  }

```

```

1  vector<string> w;
2  add(w, "hello"); create!
3
4  string s { "bye" };
5  add(w, s);

```

copy!

Forwarding references

Opportunity for optimization

```
1  template <typename T>
2  void add(vector<T>& v,
3          T value)
4  {
5      v.push_back(value);
6  }
```

```
1  vector<string> w;
2  add(w, "hello"s);
3
4  string s { "bye" };
5  add(w, s);
```

rvalue \Rightarrow 3 created

Forwarding references

Opportunity for optimization

```
1 template <typename T>
2 void add(vector<T>& v,
3         destroy T value)
4 {
5     v.push_back(value);
6 }
```

```
1 vector<string> w;
2 add(w, "hello"s);
3
4 string s { "bye" };
5 add(w, s);
```

rvalue \Rightarrow 3 created

Forwarding references

Opportunity for optimization

```

1  template <typename T>
2  void add(vector<T>& v,
3         destroy T value)
4  {
5      v.push_back(value);
6  }
```

```

1  vector<string> w;
2  add(w, "hello");
3
4  string s { "bye" };
5  add(w, s);
```

rvalue \Rightarrow 3 created

Forwarding references

Opportunity for optimization

```
1 template <typename T>
2 void add(vector<T>& v,
3         T value)
4 {
5     v.push_back(value);
6 }
```

```
1 vector<string> w;
2 add(w, "hello"s);
3
4 string s { "bye" };
5 add(w, s);
```

rvalue \Rightarrow 3 created, 2 destroyed

Forwarding references

Opportunity for optimization

```

1  template <typename T>
2  void add(vector<T>& v,
3          T value)
4  {
5      v.push_back(value);
6  }
```

```

1  vector<string> w;
2  add(w, "hello"s);
3          create!
4  string s { "bye" };
5  add(w, s);
```

rvalue \Rightarrow 3 created, 2 destroyed

Forwarding references

Opportunity for optimization

```

1  template <typename T>
2  void add(vector<T>& v,
3          T value)
4  {
5      v.push_back(value);
6  }
```

```

1  vector<string> w;
2  add(w, "hello"s);
3
4  string s { "bye" };
5  add(w, s);
```

Annotations in the code above: "hello"s is red, "create!" is blue above line 4, and a blue box highlights "s { \"bye\" }" on line 4. A blue arrow points from the boxed "s" to the "s" in line 5. A dashed line is below the "s" in line 5.

rvalue \Rightarrow 3 created, 2 destroyed

Forwarding references

Opportunity for optimization

```

1  template <typename T>
2  void add(vector<T>& v,
3         copy! T value)
4  {
5      v.push_back(value);
6  }
```

```

1  vector<string> w;
2  add(w, "hello"s);
3
4  string s { "bye" };
5  add(w, s);
```

rvalue \Rightarrow 3 created, 2 destroyed

Forwarding references

Opportunity for optimization

```

1  template <typename T>
2  void add(vector<T>& v,
3         copy! T value)
4  {
5      v.push_back(value);
6  }

```

```

1  vector<string> w;
2  add(w, "hello"s);
3
4  string s { "bye" };
5  add(w, s);

```

rvalue \Rightarrow 3 created, 2 destroyed

Forwarding references

Opportunity for optimization

```

1  template <typename T>
2  void add(vector<T>& v,
3         copy! T value)
4  {
5  [v] push_back([value]);
6  }

```

```

1  vector<string> w;
2  add(w, "hello"s);
3
4  string s { "bye" };
5  add(w, s);

```

copy! rvalue \Rightarrow 3 created, 2 destroyed

Forwarding references

Opportunity for optimization

```

1  template <typename T>
2  void add(vector<T>& v,
3          T value)
4  {
5      v.push_back(value);
6  }
```

```

1  vector<string> w;
2  add(w, "hello"s);
3
4  string s { "bye" };
5  add(w, s);
```

rvalue \Rightarrow 3 created, 2 destroyed

lvalue \Rightarrow 3 created

Forwarding references

Opportunity for optimization

```

1  template <typename T>
2  void add(vector<T>& v,
3         destroy T value)
4  {
5     v.push_back(value);
6  }
```

```

1  vector<string> w;
2  add(w, "hello"s);
3
4  string s { "bye" };
5  add(w, s);
```

rvalue \Rightarrow 3 created, 2 destroyed

lvalue \Rightarrow 3 created

Forwarding references

Opportunity for optimization

```

1  template <typename T>
2  void add(vector<T>& v,
3         destroy T value)
4  {
5      v.push_back(value);
6  }
```

```

1  vector<string> w;
2  add(w, "hello"s);
3
4  string s { "bye" };
5  add(w, s);
```

rvalue \Rightarrow 3 created, 2 destroyed

lvalue \Rightarrow 3 created

Forwarding references

Opportunity for optimization

```

1  template <typename T>
2  void add(vector<T>& v,
3         destroy T value)
4  {
5      v.push_back(value);
6  }
```

```

1  vector<string> w;
2  add(w, "hello"s);
3         destroy
4  string s { "bye" };
5  add(w, s);
```

rvalue \Rightarrow 3 created, 2 destroyed

lvalue \Rightarrow 3 created

Forwarding references

Opportunity for optimization

```

1  template <typename T>
2  void add(vector<T>& v,
3          T value)
4  {
5      v.push_back(value);
6  }
```

```

1  vector<string> w;
2  add(w, "hello"s);
3
4  string s { "bye" };
5  add(w, s);
```

rvalue \Rightarrow 3 created, 2 destroyed

lvalue \Rightarrow 3 created, 2 destroyed

Forwarding references

Opportunity for optimization

```
1 template <typename T>
2 void add(vector<T>& v,
3         T value)
4 {
5     v.push_back(value);
6 }
```

```
1 vector<string> w;
2 add(w, "hello"s);
3
4 string s { "bye" };
5 add(w, s);
```

rvalue \Rightarrow 3 created, 2 destroyed

lvalue \Rightarrow 3 created, 2 destroyed

Must be possible to reduce!

Forwarding references

Attempt #1

```
1 template <typename T>
2 void add(vector<T>& v,
3         T const& value)
4 {
5     v.push_back(value);
6 }
```

```
1 vector<string> w;
2 add(w, "hello"s);
3
4 string s { "bye" };
5 add(w, s);
```

Forwarding references

Attempt #1

```
1 template <typename T>
2 void add(vector<T>& v,
3         T const& value)
4 {
5     v.push_back(value);
6 }
```

```
1 vector<string> w;
2 add(w, "hello"); create!
3
4 string s { "bye" };
5 add(w, s);
```

Forwarding references

Attempt #1

```
1 template <typename T>
2 void add(vector<T>& v,
3         T const& value)
4 {
5     v.push_back(value);
6 }
```

```
1 vector<string> w;
2 add(w, "hello"); create!
3
4 string s { "bye" };
5 add(w, s);
```

Forwarding references

Attempt #1

```
1 template <typename T>
2 void add(vector<T>& v,
3         T const& value)
4 {
5     v.push_back(value);
6 }
```

```
1 vector<string> w;
2 add(w, "hello"); create!
3
4 string s { "bye" };
5 add(w, s);
```

Forwarding references

Attempt #1

```

1  template <typename T>
2  void add(vector<T>& v,
3          T const& value)
4  {
5      v.push_back(value);
6  }

```

copy!

```

1  vector<string> w;
2  add(w, "hello"); create!
3
4  string s { "bye" };
5  add(w, s);

```

Forwarding references

Attempt #1

```
1 template <typename T>
2 void add(vector<T>& v,
3         T const& value)
4 {
5     v.push_back(value);
6 }
```

```
1 vector<string> w;
2 add(w, "hello"s);
3
4 string s { "bye" };
5 add(w, s);
```

rvalue \Rightarrow 2 created

Forwarding references

Attempt #1

```
1 template <typename T>
2 void add(vector<T>& v,
3         T const& value)
4 {
5     v.push_back(value);
6 }
```

```
1 vector<string> w;
2 add(w, "hello"s);
3
4 string s { "bye" };
5 add(w, s);
```

rvalue \Rightarrow 2 created

Forwarding references

Attempt #1

```

1  template <typename T>
2  void add(vector<T>& v,
3          T const& value)
4  {
5      v.push_back(value);
6  }
```

```

1  vector<string> w;
2  add(w, "hello");
3
4  string s { "bye" };
5  add(w, s);
```

destroy

rvalue \Rightarrow 2 created

Forwarding references

Attempt #1

```
1 template <typename T>
2 void add(vector<T>& v,
3         T const& value)
4 {
5     v.push_back(value);
6 }
```

```
1 vector<string> w;
2 add(w, "hello"s);
3
4 string s { "bye" };
5 add(w, s);
```

rvalue \Rightarrow 2 created, 1 destroyed

Forwarding references

Attempt #1

```
1 template <typename T>
2 void add(vector<T>& v,
3         T const& value)
4 {
5     v.push_back(value);
6 }
```

```
1 vector<string> w;
2 add(w, "hello"s);
3         create!
4 string s { "bye" };
5 add(w, s);
```

rvalue \Rightarrow 2 created, 1 destroyed

Forwarding references

Attempt #1

```

1  template <typename T>
2  void add(vector<T>& v,
3          T const& value)
4  {
5      v.push_back(value);
6  }

```

```

1  vector<string> w;
2  add(w, "hello"s);
3
4  string s { "bye" };
5  add(w, s);

```

rvalue \Rightarrow 2 created, 1 destroyed

Forwarding references

Attempt #1

```

1  template <typename T>
2  void add(vector<T>& v,
3          T const& value)
4  {
5      v.push_back(value);
6  }

```

```

1  vector<string> w;
2  add(w, "hello"s);
3
4  string s { "bye" };
5  add(w, s);

```

rvalue \Rightarrow 2 created, 1 destroyed

Forwarding references

Attempt #1

```

1  template <typename T>
2  void add(vector<T>& v,
3          T const& value)
4  {
5      v.push_back(value);
6  }

```

```

1  vector<string> w;
2  add(w, "hello"s);
3
4  string s { "bye" };
5  add(w, s);

```

rvalue \Rightarrow 2 created, 1 destroyed

Forwarding references

Attempt #1

```

1  template <typename T>
2  void add(vector<T>& v,
3          T const& value)
4  {
5      v.push_back(value);
6  }

```

```

1  vector<string> w;
2  add(w, "hello"s);
3
4  string s { "bye" };
5  add(w, s);

```

copy! rvalue \Rightarrow 2 created, 1 destroyed

Forwarding references

Attempt #1

```
1 template <typename T>
2 void add(vector<T>& v,
3         T const& value)
4 {
5     v.push_back(value);
6 }
```

```
1 vector<string> w;
2 add(w, "hello"s);
3
4 string s { "bye" };
5 add(w, s);
```

rvalue \Rightarrow 2 created, 1 destroyed

lvalue \Rightarrow 2 created

Forwarding references

Attempt #1

```
1 template <typename T>
2 void add(vector<T>& v,
3         T const& value)
4 {
5     v.push_back(value);
6 }
```

```
1 vector<string> w;
2 add(w, "hello"s);
3
4 string s { "bye" };
5 add(w, s);
```

rvalue \Rightarrow 2 created, 1 destroyed

lvalue \Rightarrow 2 created

Forwarding references

Attempt #1

```

1  template <typename T>
2  void add(vector<T>& v,
3          T const& value)
4  {
5      v.push_back(value);
6  }
```

```

1  vector<string> w;
2  add(w, "hello"s);
3
4  string s { "bye" };
5  add(w, s);
```

rvalue \Rightarrow 2 created, 1 destroyed

lvalue \Rightarrow 2 created

Forwarding references

Attempt #1

```

1  template <typename T>
2  void add(vector<T>& v,
3          T const& value)
4  {
5      v.push_back(value);
6  }

```

```

1  vector<string> w;
2  add(w, "hello"s);
3
4  string s { "bye" };
5  add(w, s);

```

rvalue \Rightarrow 2 created, 1 destroyed

lvalue \Rightarrow 2 created

Forwarding references

Attempt #1

```
1 template <typename T>
2 void add(vector<T>& v,
3         T const& value)
4 {
5     v.push_back(value);
6 }
```

```
1 vector<string> w;
2 add(w, "hello"s);
3
4 string s { "bye" };
5 add(w, s);
```

rvalue \Rightarrow 2 created, 1 destroyed

lvalue \Rightarrow 2 created, 1 destroyed

Forwarding references

Attempt #1

```
1 template <typename T>
2 void add(vector<T>& v,
3         T const& value)
4 {
5     v.push_back(value);
6 }
```

```
1 vector<string> w;
2 add(w, "hello"s);
3
4 string s { "bye" };
5 add(w, s);
```

rvalue \Rightarrow 2 created, 1 destroyed

lvalue \Rightarrow 2 created, 1 destroyed

The rvalue case should probably use move!

Forwarding references

Attempt #2

```
1  template <typename T>
2  void add(vector<T>& v, T const& value) // #1
3  {
4      v.push_back(value);
5  }
6
7  template <typename T>
8  void add(vector<T>& v, T&& value) // #2
9  {
10     v.push_back(std::move(value));
11 }
```

```
1  vector<string> w;
2  string s { "bye" };
3
4
5  add(w, "hello"s);
6
7
8  add(w, s);
```

Forwarding references

Attempt #2

```
1  template <typename T>
2  void add(vector<T>& v, T const& value) // #1
3  {
4      v.push_back(value);
5  }
6
7  template <typename T>
8  void add(vector<T>& v, T&& value) // #2
9  {
10     v.push_back(std::move(value));
11 }
```

```
1  vector<string> w;
2  string s { "bye" };
3
4  // chooses #2!
5  add(w, "hello"s);
6
7
8  add(w, s);
```

Forwarding references

Attempt #2

```
1  template <typename T>
2  void add(vector<T>& v, T const& value) // #1
3  {
4      v.push_back(value);
5  }
6
7  template <typename T>
8  void add(vector<T>& v, T&& value) // #2
9  {
10     v.push_back(std::move(value));
11 }
```

```
1  vector<string> w;
2  string s { "bye" };
3
4  // chooses #2!
5  add(w, "hello"s);
6
7  // chooses #1!
8  add(w, s);
```

Forwarding references

Attempt #2

```

1  template <typename T>
2  void add(vector<T>& v, T const& value) // #1
3  {
4      v.push_back(value);
5  }
6
7  template <typename T>
8  void add(vector<T>& v, T&& value) // #2
9  {
10     v.push_back(std::move(value));
11 }

```

```

1  vector<string> w;
2  string s { "bye" };
3
4  // chooses #2!
5  add(w, "hello"s);
6
7  // chooses #1!
8  add(w, s);

```

Cool!

Forwarding references

Attempt #2

```

1  template <typename T>
2  void add(vector<T>& v, T const& value) // #1
3  {
4      v.push_back(value);
5  }
6
7  template <typename T>
8  void add(vector<T>& v, T&& value) // #2
9  {
10     v.push_back(std::move(value));
11 }

```

```

1  vector<string> w;
2  string s { "bye" };
3
4  // chooses #2!
5  add(w, "hello"s);
6
7  // chooses #1!
8  add(w, s);

```

Cool!

But what about the duplication?

Forwarding references

Attempt #3

```
1  template <typename T, typename U>  
2  void add(vector<T>& v, U&& value)  
3  {  
4      v.push_back(std::move(value));  
5  }
```

```
1  vector<string> w;  
2  string s { "bye" };  
3  
4  
5  add(w, "hello"s);  
6  
7  
8  add(w, s);
```

Forwarding references

Attempt #3

```
1  template <typename T, typename U>  
2  void add(vector<T>& v, U&& value)  
3  {  
4      v.push_back(std::move(value));  
5  }
```

```
1  vector<string> w;  
2  string s { "bye" };  
3  
4  // Works (as expected)!  
5  add(w, "hello"s);  
6  
7  
8  add(w, s);
```

Forwarding references

Attempt #3

```
1  template <typename T, typename U>
2  void add(vector<T>& v, U&& value)
3  {
4      v.push_back(std::move(value));
5  }
```

```
1  vector<string> w;
2  string s { "bye" };
3
4  // Works (as expected)!
5  add(w, "hello"s);
6
7  // Also works?!
8  add(w, s);
```

Forwarding references

Attempt #3

```
1  template <typename T, typename U>
2  void add(vector<T>& v, U&& value)
3  {
4      v.push_back(std::move(value));
5  }
```

```
1  vector<string> w;
2  string s { "bye" };
3
4  // Works (as expected)!
5  add(w, "hello"s);
6
7  // But s is now empty?!
8  add(w, s);
```

Forwarding references

What is going on?

```
1 vector<string> w;  
2 string s { "bye" };  
3  
4  
5 add(w, "hello"s);  
6  
7  
8 add(w, s);
```

```
1 template <typename T, typename U>  
2 void add(vector<T>& v, U&& value);  
3  
4  
5  
6  
7  
8
```

Forwarding references

What is going on?

```
1 vector<string> w;  
2 string s { "bye" };  
3  
4 // T = string  
5 add(w, "hello"s);  
6  
7  
8 add(w, s);
```

```
1 template <typename T, typename U>  
2 void add(vector<T>& v, U&& value);  
3  
4  
5  
6  
7  
8
```

Forwarding references

What is going on?

```
1 vector<string> w;  
2 string s { "bye" };  
3  
4 // T = string, U = string  
5 add(w, "hello"s);  
6  
7  
8 add(w, s);
```

```
1 template <typename T, typename U>  
2 void add(vector<T>& v, U&& value);  
3  
4  
5  
6  
7  
8
```

Forwarding references

What is going on?

```
1 vector<string> w;  
2 string s { "bye" };  
3  
4 // T = string, U = string  
5 add(w, "hello"s);  
6  
7  
8 add(w, s);
```

```
1 template <typename T, typename U>  
2 void add(vector<T>& v, U&& value);  
3  
4 void add(vector<string>& v,  
5         string&& value);  
6  
7  
8
```

Forwarding references

What is going on?

```
1 vector<string> w;  
2 string s { "bye" };  
3  
4 // T = string, U = string  
5 add(w, "hello"s);  
6  
7 // T = string  
8 add(w, s);
```

```
1 template <typename T, typename U>  
2 void add(vector<T>& v, U&& value);  
3  
4 void add(vector<string>& v,  
5         string&& value);  
6  
7  
8
```

Forwarding references

What is going on?

```
1 vector<string> w;  
2 string s { "bye" };  
3  
4 // T = string, U = string  
5 add(w, "hello"s);  
6  
7 // T = string, U = string& (???)  
8 add(w, s);
```

```
1 template <typename T, typename U>  
2 void add(vector<T>& v, U&& value);  
3  
4 void add(vector<string>& v,  
5         string&& value);  
6  
7  
8
```

Forwarding references

What is going on?

```
1 vector<string> w;  
2 string s { "bye" };  
3  
4 // T = string, U = string  
5 add(w, "hello"s);  
6  
7 // T = string, U = string& (???)  
8 add(w, s);
```

```
1 template <typename T, typename U>  
2 void add(vector<T>& v, U&& value);  
3  
4 void add(vector<string>& v,  
5         string&& value);  
6  
7 void add(vector<string>& v,  
8         string&&& value); // ???
```

Forwarding references

Reference collapsing

T	T = U	\Rightarrow U
T&	T = U	\Rightarrow U&
T&&	T = U	\Rightarrow U&&
<hr/>		
T	T = U&	\Rightarrow U&
T&	T = U&	\Rightarrow U&
T&&	T = U&	\Rightarrow U&
<hr/>		
T	T = U&&	\Rightarrow U&&
T&	T = U&&	\Rightarrow U&
T&&	T = U&&	\Rightarrow U&&

Forwarding references

Reference collapsing

T	T = U	\Rightarrow U
T&	T = U	\Rightarrow U&
T&&	T = U	\Rightarrow U&&
T	T = U&	\Rightarrow U&
T&	T = U&	\Rightarrow U&
T&&	T = U&	\Rightarrow U&
T	T = U&&	\Rightarrow U&&
T&	T = U&&	\Rightarrow U&
T&&	T = U&&	\Rightarrow U&&

Forwarding references

Reference collapsing

<code>T</code>	<code>T = U</code>	$\Rightarrow U$
<code>T&</code>	<code>T = U</code>	$\Rightarrow U\&$
<code>T&&</code>	<code>T = U</code>	$\Rightarrow U\&\&$
<code>T</code>	<code>T = U&</code>	$\Rightarrow U\&$
<code>T&</code>	<code>T = U&</code>	$\Rightarrow U\&$
<code>T&&</code>	<code>T = U&</code>	$\Rightarrow U\&$
<code>T</code>	<code>T = U&&</code>	$\Rightarrow U\&\&$
<code>T&</code>	<code>T = U&&</code>	$\Rightarrow U\&$
<code>T&&</code>	<code>T = U&&</code>	$\Rightarrow U\&\&$

Forwarding references

Reference collapsing

T	T = U	⇒ U
T&	T = U	⇒ U&
T&&	T = U	⇒ U&&
T	T = U&	⇒ U&
T&	T = U&	⇒ U&
T&&	T = U&	⇒ U&
T	T = U&&	⇒ U&&
T&	T = U&&	⇒ U&
T&&	T = U&&	⇒ U&&

Forwarding references

Reference collapsing

T	T = U	⇒ U
T&	T = U	⇒ U&
T&&	T = U	⇒ U&&
T	T = U&	⇒ U&
T&	T = U&	⇒ U&
T&&	T = U&	⇒ U&
T	T = U&&	⇒ U&&
T&	T = U&&	⇒ U&
T&&	T = U&&	⇒ U&&

Forwarding references

Reference collapsing

T	T = U	⇒ U
T&	T = U	⇒ U&
T&&	T = U	⇒ U&&
T	T = U&	⇒ U&
T&	T = U&	⇒ U&
T&&	T = U&	⇒ U&
T	T = U&&	⇒ U&&
T&	T = U&&	⇒ U&
T&&	T = U&&	⇒ U&&

Forwarding references

Reference collapsing

T	T = U	⇒ U
T&	T = U	⇒ U&
T&&	T = U	⇒ U&&
T	T = U&	⇒ U&
T&	T = U&	⇒ U&
T&&	T = U&	⇒ U&
T	T = U&&	⇒ U&&
T&	T = U&&	⇒ U&
T&&	T = U&&	⇒ U&&

Forwarding references

Reference collapsing

T	T = U	⇒ U
T&	T = U	⇒ U&
T&&	T = U	⇒ U&&
T	T = U&	⇒ U&
T&	T = U&	⇒ U&
T&&	T = U&	⇒ U&
T	T = U&&	⇒ U&&
T&	T = U&&	⇒ U&
T&&	T = U&&	⇒ U&&

Forwarding references

Reference collapsing

T	T = U	⇒ U
T&	T = U	⇒ U&
T&&	T = U	⇒ U&&
T	T = U&	⇒ U&
T&	T = U&	⇒ U&
T&&	T = U&	⇒ U&
T	T = U&&	⇒ U&&
T&	T = U&&	⇒ U&
T&&	T = U&&	⇒ U&&

Forwarding references

Reference collapsing

T	T = U	⇒ U
T&	T = U	⇒ U&
T&&	T = U	⇒ U&&
T	T = U&	⇒ U&
T&	T = U&	⇒ U&
T&&	T = U&	⇒ U&
T	T = U&&	⇒ U&&
T&	T = U&&	⇒ U&
T&&	T = U&&	⇒ U&&

Forwarding references

Reference collapsing

Rule of thumb: `&` > `&&` > no reference

Forwarding references

Problem

```
1  template <typename T, typename U>  
2  void add(vector<int>& v, U&& value)  
3  {  
4      v.push_back(std::move(value));  
5  }
```

Forwarding references

Problem

```
1  template <typename T, typename U>  
2  void add(vector<int>& v, U&& value)  
3  {  
4      v.push_back(std::move(value));  
5  }
```

Forwarding references

Problem

```
1  template <typename T, typename U>
2  void add(vector<int>& v, U&& value)
3  {
4      v.push_back(std::move(value)) ← will always move
5  }
```

Forwarding references

Fix by duplication

```
1  template <typename T, typename U>
2  void add(vector<int>& v, U&& value)
3  {
4      if (std::is_lvalue_reference_v<U>)
5          v.push_back(value);
6      else
7          v.push_back(std::move(value));
8  }
```

Forwarding references

Fix by duplication

```
1  template <typename T, typename U>
2  void add(vector<int>& v, U&& value)
3  {
4      if (std::is_lvalue_reference_v<U>)
5          v.push_back(value);
6      else
7          v.push_back(std::move(value));
8  }
```

Forwarding references

Fix by duplication

```
1  template <typename T, typename U>
2  void add(vector<int>& v, U&& value)
3  {
4      if (std::is_lvalue_reference_v<U>)
5          v.push_back(value); ← value is lvalue ⇒ copy!
6      else
7          v.push_back(std::move(value));
8  }
```

Forwarding references

Fix by duplication

```
1  template <typename T, typename U>
2  void add(vector<int>& v, U&& value)
3  {
4      if (std::is_lvalue_reference_v<U>)
5          v.push_back(value); ← value is lvalue ⇒ copy!
6      else
7          v.push_back(std::move(value));
8  }
```

Forwarding references

Fix by duplication

```
1  template <typename T, typename U>
2  void add(vector<int>& v, U&& value)
3  {
4      if (std::is_lvalue_reference_v<U>)
5          v.push_back(value) ← value is lvalue ⇒ copy!
6      else
7          v.push_back(std::move(value)) ← value is rvalue ⇒ move!
8  }
```

Forwarding references

Better fix!

```
1  template <typename T, typename U>  
2  void add(vector<int>& v, U&& value)  
3  {  
4      v.push_back(std::forward<U>(value));  
5  }
```

Forwarding references

Forwarding references

T&& is called a *forwarding reference* **if**:

1. T is an arbitrary type (template parameter or `auto`)
2. T is *deduced* by the compiler

Forwarding references

Examples: Is there a forwarding reference in this example?

```
1  template <typename T>  
2  struct X  
3  {  
4      static void fun(T&&);  
5  };
```

Forwarding references

Examples: Is there a forwarding reference in this example?

```
1  template <typename T>  
2  struct X  
3  {  
4      static void fun(T&&);  
5  };
```

No!

Forwarding references

Examples: Is there a forwarding reference in this example?

```
1  template <typename T>  
2  void fun(T&&);  
3  
4  
5
```

Forwarding references

Examples: Is there a forwarding reference in this example?

```
1  template <typename T>  
2  void fun(T&&);  
3  
4  int x { 5 };  
5  fun(x); // yes!
```

Forwarding references

Examples: Is there a forwarding reference in this example?

```
1  template <typename T>  
2  void fun(T&&);  
3  
4  int x { 5 };  
5  fun<int>(x); // no...
```

Forwarding references

Examples: Is there a forwarding reference in this example?

```
1  template <typename T>
2  void fun(T&&);
3
4  int x { 5 };
5  fun<int>(x); // no...
```

Yes, if T is not specified

Forwarding references

Examples: Is there a forwarding reference in this example?

```
1  template <typename T>  
2  void fun(vector<T>&& v);  
3  
4  
5
```

Forwarding references

Examples: Is there a forwarding reference in this example?

```
1  template <typename T>  
2  void fun(vector<T>&& v);  
3  
4  
5
```

No!

Forwarding references

Examples: Is there a forwarding reference in this example?

```
1 auto&& var = 5;  
2  
3  
4  
5
```

Forwarding references

Examples: Is there a forwarding reference in this example?

```
1 auto&& var = 5;  
2  
3  
4  
5
```

Yes!

Forwarding references

Rule of thumb

Use forwarding references if:

Forwarding references

Rule of thumb

Use forwarding references if:

- a templated parameter is passed to another function

Forwarding references

Rule of thumb

Use forwarding references if:

- a templated parameter is passed to another function, or
- you are writing generic code

Forwarding references

Rule of thumb

Use forwarding references if:

- a templated parameter is passed to another function, or
- you are writing generic code, e.g.: `for (auto&& v : c)`

- 1 Forwarding references
- 2 Smart pointers
- 3 `std::unique_ptr`
- 4 `std::shared_ptr`

Smart pointers

Can we get memory problems here?

```
1  class My_Class
2  {
3  public:
4      My_Class(int x, int y);
5      ~My_Class()
6      {
7          delete p1;
8          delete p2;
9      }
10 private:
11     int* p1;
12     int* p2;
13 };
```

Smart pointers

Can we get memory problems here?

```
1  static int* create(int n)
2  {
3      if (n >= 0)
4      {
5          return new int{n};
6      }
7      throw domain_error{"Negative"};
8  }
9
10 My_Class::My_Class(int x, int y)
11     : p1{create(x)}, p2{create(y)}
12 { }
```

Smart pointers

Yes, there are problems!

```
1 int main()  
2 {  
3     My_Class c{0, -1};  
4 }
```

Smart pointers

Solution?

```
1 My_Class::My_Class(int x, int y) try
2   : p1{create(x)}, p2{create(y)}
3   { }
4 catch (domain_error& e)
5   {
6     delete p1;
7   }
8 int main()
9   {
10    My_Class c{-1, 0};
11  }
```

Smart pointers

Solution?

```
1 My_Class::My_Class(int x, int y) try
2   : p1{create(x)}, p2{create(y)}
3   { }
4 catch (domain_error& e)
5   {
6     delete p1;
7   }
8 int main()
9   {
10    My_Class c{-1, 0};
11  }
```

Smart pointers

Solution?

```
1 My_Class::My_Class(int x, int y)
2   : p1{create(x)}, p2{}
3   {
4     try
5     {
6       p2 = create(y);
7     }
8     catch (domain_error& e)
9     {
10      delete p1;
11      throw;
12    }
13 }
```

Smart pointers

Solution?

```
1 My_Class::My_Class(int x, int y)
2   : p1{create(x)}, p2{}
3   {
4     try
5     {
6       p2 = create(y);
7     }
8     catch (domain_error& e)
9     {
10      delete p1;
11      throw;
12    }
13 }
```

Works...

Smart pointers

Solution?

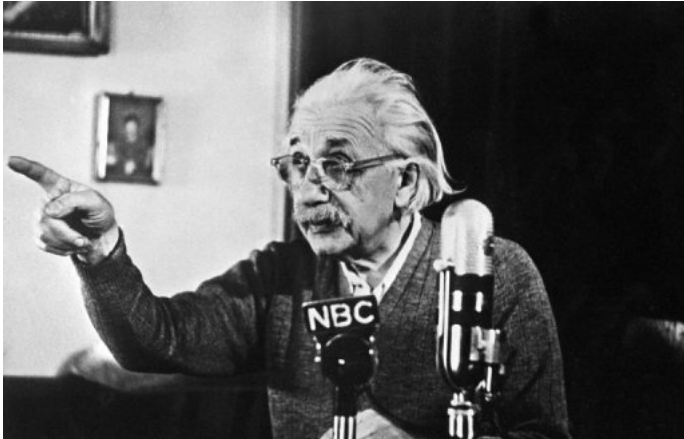
```
1 My_Class::My_Class(int x, int y)
2   : p1{create(x)}, p2{}
3   {
4     try
5     {
6       p2 = create(y);
7     }
8     catch (domain_error& e)
9     {
10      delete p1;
11      throw;
12    }
13 }
```

At what cost?

Smart pointers

It would be nice if pointers
could deallocate
themselves...

Smart pointers



Smart pointers

Let's use RAII!

```
1  template <typename T>
2  struct Smart_Pointer
3  {
4      T* data; // the data
5
6      // automatically delete (and null)
7      ~Smart_Pointer()
8      {
9          delete data;
10         data = nullptr;
11     }
12
13     // access by casting to T*
14     operator T*()
15     {
16         return data;
17     }
18 };
```

Smart pointers

Better solution!

```
1 class My_Class
2 {
3 public:
4     My_Class(int x, int y);
5     ~My_Class() = default;
6 private:
7     Smart_Pointer<int> p1;
8     Smart_Pointer<int> p2;
9 };
```

Smart pointers

Better solution!

```
1 My_Class::My_Class(int x, int y)
2   : p1{create(x)}, p2{create(y)}
3   { }
```

Smart pointers

Let's see another motivating example

Smart pointers

Do you see any issues here?

```
1  struct Image {
2      Color data[2500000000]; // 1 GB
3  };
4  Image load_image(string const& file);
5
6  struct Sprite {
7      // ...
8
9      Image image;
10 };
11 Sprite create_sprite(string const& file) {
12     return Sprite { load_image(file) };
13 }
14
15 void run() {
16
17     auto s1 = create_sprite("image.png");
18     auto s2 = create_sprite("image.png");
19     // use s1 and s2
20 }
```

Smart pointers

Do you see any issues here?

```
1  struct Image
2      Color data[2500000000]; // 1 GB
3  };
4  Image load_image(string const& file);
5
6  struct Sprite {
7      // ...
8
9      Image image;
10 };
11 Sprite create_sprite(string const& file) {
12     return Sprite { load_image(file) };
13 }
14
15 void run() {
16
17     auto s1 = create_sprite("image.png");
18     auto s2 = create_sprite("image.png");
19     // use s1 and s2
20 }
```

Segmentation fault!

Smart pointers

Do you see any issues now?

```
1  struct Image {
2      Color data[2500000000]; // 1 GB
3  };
4  Image* load_image(string const& file);
5
6  struct Sprite {
7      // ...
8
9      Image* image;
10 };
11 Sprite create_sprite(string const& file) {
12     return Sprite { load_image(file) };
13 }
14
15 void run() {
16
17     auto s1 = create_sprite("image.png");
18     auto s2 = create_sprite("image.png");
19     // use s1 and s2
20 }
```

Smart pointers

Do you see any issues now?

```
1 struct Image {
2     Color data[2500000000]; // 1 GB
3 };
4 Image* load_image(string const& file);
5
6 struct Sprite {
7     // ...
8
9     Image* image;
10 };
11 Sprite create_sprite(string const& file) {
12     return Sprite { load_image(file) };
13 }
14
15 void run() {
16
17     auto s1 = create_sprite("image.png");
18     auto s2 = create_sprite("image.png");
19     // use s1 and s2
20 }
```

Memory leaks!

Smart pointers

How about **now**?

```
1  struct Image {
2      Color data[2500000000]; // 1 GB
3  };
4  Image* load_image(string const& file);
5
6  struct Sprite {
7      // ...
8      ~Sprite() { delete image; }
9      Image* image;
10 };
11 Sprite create_sprite(string const& file) {
12     return Sprite { load_image(file) };
13 }
14
15 void run() {
16
17     auto s1 = create_sprite("image.png");
18     auto s2 = create_sprite("image.png");
19     // use s1 and s2
20 }
```

Smart pointers

How about **now**?

```
1 struct Image {
2     Color data[2500000000]; // 1 GB
3 };
4 Image* load_image(string const& file);
5
6 struct Sprite {
7     // ...
8     ~Sprite() { delete image; }
9     Image* image;
10 };
11 Sprite create_sprite(string const& file) {
12     return Sprite { load_image(file) };
13 }
14
15 void run() {
16
17     auto s1 = create_sprite("image.png");
18     auto s2 = create_sprite("image.png");
19     // use s1 and s2
20 }
```

Smart pointers

How about **now**?

```
1  struct Image {
2      Color data[2500000000]; // 1 GB
3  };
4  Image* load_image(string const& file);
5
6  struct Sprite {
7      // ...
8      ~Sprite() { delete image; }
9      Image* image;
10 };
11 Sprite create_sprite(Image* image) {
12     return Sprite { image };
13 }
14
15 void run() {
16     Image* image = load_image("image.png");
17     auto s1 = create_sprite(image);
18     auto s2 = create_sprite(image);
19     // use s1 and s2
20 }
```

Smart pointers

How about **now**?

```
1  struct Image {
2      Color data[2500000000]; // 1 GB
3  };
4  Image* load_image(string const& file);
5
6  struct Sprite {
7      // ...
8      ~Sprite() { delete image; }
9      Image* image;
10 };
11 Sprite create_sprite(Image* image) {
12     return Sprite { image };
13 }
14
15 void run() {
16     Image* image = load_image("image.png");
17     auto s1 = create_sprite(image);
18     auto s2 = create_sprite(image);
19     // use s1 and s2
20 }
```

Double free error!

Smart pointers

Two solutions

This problem can be solved in two ways:

- *unique ownership*
- *shared ownership*

Depending on your perspective

Smart pointers

Important model: Ownership

Whoever is responsible for deallocating a resource is the
owner of said resource

Smart pointers

Important model: Ownership

Whoever is responsible for deallocating a resource is the *owner* of said resource

If done correctly *ownership* can be enforced by the compiler

Smart pointers

Important model: Ownership

Whoever is responsible for deallocating a resource is the *owner* of said resource

If done correctly *ownership* can be enforced by the compiler

In C++ this is done through *smart pointers*

Smart pointers

Different kinds of smart pointers

- Ownership is unique – `std::unique_ptr`
- Ownership is shared – `std::shared_ptr`

Smart pointers

Shared properties of all smart pointers

A smart pointer:

Smart pointers

Shared properties of all smart pointers

A smart pointer:

- Is used like a “normal” pointer

Smart pointers

Shared properties of all smart pointers

A smart pointer:

- Is used like a “normal” pointer
- Deletes the memory when no owner remains

Smart pointers

Shared properties of all smart pointers

A smart pointer:

- Is used like a “normal” pointer
- Deletes the memory when no owner remains
- Sets itself to `nullptr` once it is deleted

- 1 Forwarding references
- 2 Smart pointers
- 3 `std::unique_ptr`
- 4 `std::shared_ptr`

std::unique_ptr

Identifying the *owner*

```
1  struct Image {
2      Color data[2500000000]; // 1 GB
3  };
4  Image* load_image(string const& file);
5
6  struct Sprite {
7      // ...
8      ~Sprite() { delete image; } // Sprite owns it
9      Image* image;
10 };
11 Sprite create_sprite(Image* image) {
12     return Sprite { image };
13 }
14
15 void run() {
16     Image* image = load_image("image.png");
17     auto s1 = create_sprite(image);
18     auto s2 = create_sprite(image);
19     // use s1 and s2
20 }
```

std::unique_ptr

Identifying the *owner*

```
1  struct Image {
2      Color data[2500000000]; // 1 GB
3  };
4  Image* load_image(string const& file);
5
6  struct Sprite {
7      // ...
8      Image* image;
9  };
10 Sprite create_sprite(Image* image) {
11     return Sprite { image };
12 }
13
14 void run() {
15     Image* image = load_image("image.png");
16     auto s1 = create_sprite(image);
17     auto s2 = create_sprite(image);
18     // use s1 and s2
19     delete image; // run() owns it
20 }
```

std::unique_ptr

Incorrect solution

```
1  struct Image {
2      Color data[2500000000]; // 1 GB
3  };
4  Image* load_image(string const& file);
5
6  struct Sprite {
7      // ...
8      ~Sprite() { delete image; } // sprite owns it
9      Image* image;
10 };
11 Sprite create_sprite(Image* image) {
12     return Sprite { image };
13 }
14
15 void run() {
16     Image* image = load_image("image.png");
17     auto s1 = create_sprite(image);
18     auto s2 = create_sprite(image);
19     // use s1 and s2
20     delete image; // run() owns it
21 }
```

std::unique_ptr

Incorrect solution

```
1  struct Image {
2      Color data[250000000]; // 1 GB
3  };
4  Image* load_image(string const& file);
5
6  struct Sprite {
7      //
8      ~Sprite() { delete image; } // sprite owns it
9      Image* image;
10 };
11 Sprite create_sprite(Image* image) {
12     return Sprite { image };
13 }
14
15 void run() {
16     Image* image = load_image("image.png");
17     auto s1 = create_sprite(image);
18     auto s2 = create_sprite(image);
19     // use s1 and s2
20     delete image; // run() owns it
21 }
```

conflicting ownership!

std::unique_ptr

Who owns the memory?

```
1 Data* get_data();  
2  
3 int main()  
4 {  
5     auto data = get_data();  
6  
7     // who is the owner?  
8  
9 }
```

std::unique_ptr

Who owns the memory?

```
1 Data* get_data();  
2  
3 int main()  
4 {  
5     auto data = get_data();  
6  
7     // who is the owner?  
8  
9 }
```

```
1 Data* get_data()  
2 {  
3     static Data data { ... };  
4     return &data;  
5 }
```

std::unique_ptr

Who owns the memory?

```
1 Data* get_data();  
2  
3 int main()  
4 {  
5     auto data = get_data();  
6  
7     // who is the owner?  
8     // get_data()!  
9 }
```

```
1 Data* get_data()  
2 {  
3     static Data data { ... };  
4     return &data;  
5 }
```

std::unique_ptr

Who owns the memory?

```
1 Data* get_data();  
2  
3 int main()  
4 {  
5     auto data = get_data();  
6  
7     // who is the owner?  
8  
9 }
```

```
1 Data* get_data()  
2 {  
3     return new Data { ... };  
4 }
```

std::unique_ptr

Who owns the memory?

```
1 Data* get_data();  
2  
3 int main()  
4 {  
5     auto data = get_data();  
6  
7     // who is the owner?  
8     // main()!  
9 }
```

```
1 Data* get_data()  
2 {  
3     return new Data { ... };  
4 }
```

std::unique_ptr

Who owns the memory?

```
1 Data* get_data();  
2  
3 int main()  
4 {  
5     auto data = get_data();  
6  
7     // who is the owner?  
8     // main()!  
9 }
```

```
1 Data* get_data()  
2 {  
3     return new Data { ... };  
4 }
```

How can we communicate this?

std::unique_ptr

Enter std::unique_ptr!

```
1 // raw pointer => you are NOT the owner
2 Data* get_data()
3 {
4     static Data data { ... };
5     return &data;
6 }
```

```
1 Data* get_data();
2
3 int main()
4 {
5     // I am not owner...
6     auto data = get_data();
7
8     // ... So I don't delete it
9 }
```

std::unique_ptr

Enter std::unique_ptr!

```
1 // smart pointer => you ARE the owner
2 std::unique_ptr<Data> get_data()
3 {
4     return new Data { ... };
5 }
6
```

```
1 std::unique_ptr<Data> get_data();
2
3 int main()
4 {
5     // I am the owner...
6     auto data = get_data();
7     // ... So it dies when I do
8     // => no need to delete!
9 }
```

std::unique_ptr

A big **gotcha** with std::unique_ptr

```
1 int* memory { new int { 5 } };
2 {
3     std::unique_ptr<int> ptr1 { memory };
4     {
5         std::unique_ptr<int> ptr2 { memory };
6     }
7 }
```

std::unique_ptr

A big **gotcha** with std::unique_ptr

```
1 int* memory { new int { 5 } };  
2 {  
3     std::unique_ptr<int> ptr1 { memory };  
4     {  
5         std::unique_ptr<int> ptr2 { memory };  
6     }  
7 }
```

Double free error!

std::unique_ptr

A big **gotcha** with std::unique_ptr

```
1 int* memory { new int { 5 } };  
2 {  
3     std::unique_ptr<int> ptr1 { memory };  
4     {  
5         std::unique_ptr<int> ptr2 { memory };  
6     } // ptr2 deletes the memory  
7 }
```

std::unique_ptr

A big **gotcha** with std::unique_ptr

```
1 int* memory { new int { 5 } };  
2 {  
3     std::unique_ptr<int> ptr1 { memory };  
4     {  
5         std::unique_ptr<int> ptr2 { memory };  
6     } // ptr2 deletes the memory  
7 } // ptr1 TRIES to delete the same memory
```

```
std::unique_ptr
```

Rule of thumb: Don't manually allocate the memory

std::unique_ptr

Solution

```
1 {  
2     auto ptr1 = std::make_unique<int>(5);  
3     {  
4         std::unique_ptr<int> ptr2 { ptr1 };  
5     }  
6 }
```

std::unique_ptr

Solution

```
1 {  
2     auto ptr1 = std::make_unique<int>(5);  
3     {  
4         std::unique_ptr<int> ptr2 { ptr1 };  
5     }  
6 }
```

Compile error!

std::unique_ptr

Why did it not compile?

std::unique_ptr<T>:

- Represents *unique* ownership

std::unique_ptr

Why did it not compile?

std::unique_ptr<T>:

- Represents *unique* ownership
- The ownership can not be *shared*

std::unique_ptr

Why did it not compile?

std::unique_ptr<T>:

- Represents *unique* ownership
- The ownership can not be *shared*
- \Rightarrow you are not allowed to copy it

std::unique_ptr

Why did it not compile?

std::unique_ptr<T>:

- Represents *unique* ownership
- The ownership can not be *shared*
- \Rightarrow you are not allowed to copy it
- You can however *move* the ownership

std::unique_ptr

Solution

```
1 {  
2     auto ptr1 = std::make_unique<int>(5);  
3     {  
4         std::unique_ptr<int> ptr2 { std::move(ptr1) };  
5         assert( *ptr2 == 5 );  
6     }  
7     assert( ptr1 == nullptr );  
8 }
```

std::unique_ptr

Solution

```
1 {  
2     auto ptr1 = std::make_unique<int>(5);  
3     {  
4         std::unique_ptr<int> ptr2 { std::move(ptr1) };  
5         assert( *ptr2 == 5 );  
6     }  
7     assert( ptr1 == nullptr );  
8 }
```

WORKS!

std::unique_ptr

Solving our example

```
1  struct Image {
2      Color data[2500000000]; // 1 GB
3  };
4  unique_ptr<Image> load_image(string const& file);
5
6  struct Sprite {
7      // ...
8      Image const& image;
9  };
10 Sprite create_sprite(Image const& image) {
11     return Sprite { image };
12 }
13
14 void run() {
15     // run() is the owner of the image
16     auto image = load_image("image.png");
17     auto s1 = create_sprite(*image);
18     auto s2 = create_sprite(*image);
19     // use s1 and s2
20 }
```

- 1 Forwarding references
- 2 Smart pointers
- 3 `std::unique_ptr`
- 4 `std::shared_ptr`

std::shared_ptr

Solving our example with *shared* ownership

```
1  struct Image {
2      Color data[250000000]; // 1 GB
3  };
4  shared_ptr<Image> load_image(string const& file);
5
6  struct Sprite {
7      // ...
8      shared_ptr<Image> image;
9  };
10 Sprite create_sprite(shared_ptr<Image> image) {
11     return Sprite { image };
12 }
13
14 void run() {
15     // run() is owner #1 of the image
16     auto image = load_image("image.png");
17     auto s1 = create_sprite(image); // owner #2
18     auto s2 = create_sprite(image); // owner #3
19     // use s1 and s2
20 } // once *all* owners are destroyed, the memory is deallocated
```

std::shared_ptr

Notes on std::shared_ptr

- using std::shared_ptr \neq a garbage collector
- std::shared_ptr cannot handle cycles
- it is *only* used for independent sharing of ownership

std::shared_ptr

Cycles

```
1 struct Node {
2     // make_shared requires a constructor
3     Node(int v, shared_ptr<Node> n = nullptr)
4         : value { v }, next { n } {}
5
6     int value;
7     shared_ptr<Node> next { };
8 };
9
10 int main() {
11     auto root = make_shared<Node>(0);
12     auto next = make_shared<Node>(1);
13
14     next->next = root;
15     root->next = std::move(next);
16     // next is no longer an owner
17 }
```

std::shared_ptr

Cycles

```
1 struct Node {
2     // make_shared requires a constructor
3     Node(int v, shared_ptr<Node> n = nullptr)
4         : value { v }, next { n } {}
5
6     int value;
7     shared_ptr<Node> next { };
8 };
9
10 int main() {
11     auto root = make_shared<Node>(0);
12     auto next = make_shared<Node>(1);
13
14     next->next = root;
15     root->next = std::move(next);
16     // next is no longer an owner
17 }
```

std::shared_ptr

Cycles

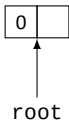
```
1 struct Node {
2     // make_shared requires a constructor
3     Node(int v, shared_ptr<Node> n = nullptr)
4         : value { v }, next { n } {}
5
6     int value;
7     shared_ptr<Node> next { };
8 };
9
10 int main() {
11     auto root = make_shared<Node>(0);
12     auto next = make_shared<Node>(1);
13
14     next->next = root;
15     root->next = std::move(next);
16     // next is no longer an owner
17 }
```

0	
---	--

std::shared_ptr

Cycles

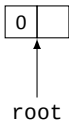
```
1 struct Node {
2     // make_shared requires a constructor
3     Node(int v, shared_ptr<Node> n = nullptr)
4         : value { v }, next { n } {}
5
6     int value;
7     shared_ptr<Node> next { };
8 };
9
10 int main() {
11     auto root = make_shared<Node>(0);
12     auto next = make_shared<Node>(1);
13
14     next->next = root;
15     root->next = std::move(next);
16     // next is no longer an owner
17 }
```



std::shared_ptr

Cycles

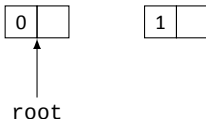
```
1 struct Node {
2     // make_shared requires a constructor
3     Node(int v, shared_ptr<Node> n = nullptr)
4         : value { v }, next { n } {}
5
6     int value;
7     shared_ptr<Node> next { };
8 };
9
10 int main() {
11     auto root = make_shared<Node>(0);
12     auto next = make_shared<Node>(1);
13
14     next->next = root;
15     root->next = std::move(next);
16     // next is no longer an owner
17 }
```



std::shared_ptr

Cycles

```
1 struct Node {
2     // make_shared requires a constructor
3     Node(int v, shared_ptr<Node> n = nullptr)
4         : value { v }, next { n } {}
5
6     int value;
7     shared_ptr<Node> next { };
8 };
9
10 int main() {
11     auto root = make_shared<Node>(0);
12     auto next = make_shared<Node>(1);
13
14     next->next = root;
15     root->next = std::move(next);
16     // next is no longer an owner
17 }
```



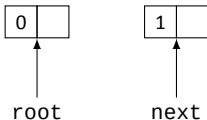
std::shared_ptr

Cycles

```

1 struct Node {
2     // make_shared requires a constructor
3     Node(int v, shared_ptr<Node> n = nullptr)
4         : value { v }, next { n } {}
5
6     int value;
7     shared_ptr<Node> next { };
8 };
9
10 int main() {
11     auto root = make_shared<Node>(0);
12     auto next = make_shared<Node>(1);
13
14     next->next = root;
15     root->next = std::move(next);
16     // next is no longer an owner
17 }

```



std::shared_ptr

Cycles

```

1 struct Node {
2     // make_shared requires a constructor
3     Node(int v, shared_ptr<Node> n = nullptr)
4         : value { v }, next { n } {}
5
6     int value;
7     shared_ptr<Node> next { };
8 };
9
10 int main() {
11     auto root = make_shared<Node>(0);
12     auto next = make_shared<Node>(1);
13
14     next->next = root;
15     root->next = std::move(next);
16     // next is no longer an owner
17 }

```



↑
root



↑
next

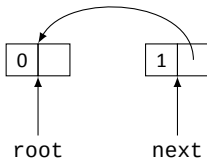
std::shared_ptr

Cycles

```

1 struct Node {
2     // make_shared requires a constructor
3     Node(int v, shared_ptr<Node> n = nullptr)
4         : value { v }, next { n } {}
5
6     int value;
7     shared_ptr<Node> next { };
8 };
9
10 int main() {
11     auto root = make_shared<Node>(0);
12     auto next = make_shared<Node>(1);
13
14     next->next = root;
15     root->next = std::move(next);
16     // next is no longer an owner
17 }

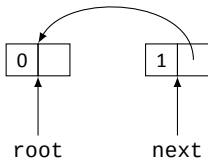
```



std::shared_ptr

Cycles

```
1 struct Node {  
2     // make_shared requires a constructor  
3     Node(int v, shared_ptr<Node> n = nullptr)  
4         : value { v }, next { n } {}  
5  
6     int value;  
7     shared_ptr<Node> next { };  
8 };  
9  
10 int main() {  
11     auto root = make_shared<Node>(0);  
12     auto next = make_shared<Node>(1);  
13  
14     next->next = root;  
15     root->next = std::move(next);  
16     // next is no longer an owner  
17 }
```



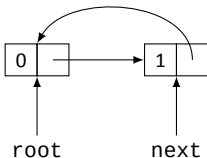
std::shared_ptr

Cycles

```

1 struct Node {
2     // make_shared requires a constructor
3     Node(int v, shared_ptr<Node> n = nullptr)
4         : value { v }, next { n } {}
5
6     int value;
7     shared_ptr<Node> next { };
8 };
9
10 int main() {
11     auto root = make_shared<Node>(0);
12     auto next = make_shared<Node>(1);
13
14     next->next = root;
15     root->next = std::move(next);
16     // next is no longer an owner
17 }

```



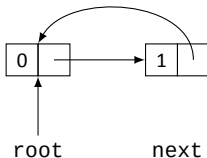
std::shared_ptr

Cycles

```

1 struct Node {
2     // make_shared requires a constructor
3     Node(int v, shared_ptr<Node> n = nullptr)
4         : value { v }, next { n } {}
5
6     int value;
7     shared_ptr<Node> next { };
8 };
9
10 int main() {
11     auto root = make_shared<Node>(0);
12     auto next = make_shared<Node>(1);
13
14     next->next = root;
15     root->next = std::move(next);
16     // next is no longer an owner
17 }

```



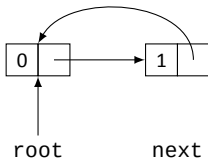
std::shared_ptr

Cycles

```

1 struct Node {
2     // make_shared requires a constructor
3     Node(int v, shared_ptr<Node> n = nullptr)
4         : value { v }, next { n } {}
5
6     int value;
7     shared_ptr<Node> next { };
8 };
9
10 int main() {
11     auto root = make_shared<Node>(0);
12     auto next = make_shared<Node>(1);
13
14     next->next = root;
15     root->next = std::move(next);
16     // next is no longer an owner
17 }

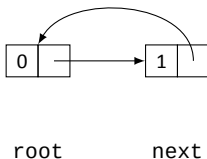
```



std::shared_ptr

Cycles

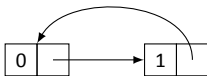
```
1 struct Node {
2     // make_shared requires a constructor
3     Node(int v, shared_ptr<Node> n = nullptr)
4         : value { v }, next { n } {}
5
6     int value;
7     shared_ptr<Node> next { };
8 };
9
10 int main() {
11     auto root = make_shared<Node>(0);
12     auto next = make_shared<Node>(1);
13
14     next->next = root;
15     root->next = std::move(next);
16     // next is no longer an owner
17 }
```



std::shared_ptr

Cycles

```
1 struct Node {
2     // make_shared requires a constructor
3     Node(int v, shared_ptr<Node> n = nullptr)
4         : value { v }, next { n } {}
5
6     int value;
7     shared_ptr<Node> next { };
8 };
9
10 int main() {
11     auto root = make_shared<Node>(0);
12     auto next = make_shared<Node>(1);
13
14     next->next = root;
15     root->next = std::move(next);
16     // next is no longer an owner
17 }
```



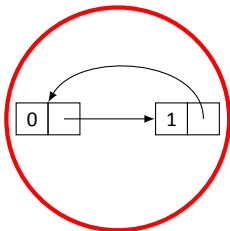
std::shared_ptr

Cycles

```

1 struct Node {
2     // make_shared requires a constructor
3     Node(int v, shared_ptr<Node> n = nullptr)
4         : value { v }, next { n } {}
5
6     int value;
7     shared_ptr<Node> next { };
8 };
9
10 int main() {
11     auto root = make_shared<Node>(0);
12     auto next = make_shared<Node>(1);
13
14     next->next = root;
15     root->next = std::move(next);
16     // next is no longer an owner
17 }

```



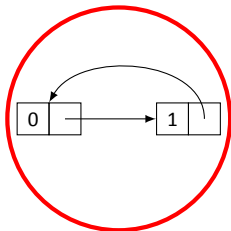
std::shared_ptr

Cycles

```

1 struct Node {
2     // make_shared requires a constructor
3     Node(int v, shared_ptr<Node> n = nullptr)
4         : value { v }, next { n } {}
5
6     int value;
7     shared_ptr<Node> next { };
8 };
9
10 int main() {
11     auto root = make_shared<Node>(0);
12     auto next = make_shared<Node>(1);
13
14     next->next = root;
15     root->next = std::move(next);
16     // next is no longer an owner
17 }

```



Keeping each other alive!

www.liu.se