

OPERATORS AND DESTRUCTORS

LECTURE 09-3

JIM FIX, REED COLLEGE CS2-S20

TODAY'S PLAN

- ▶ **CORRECTION** ABOUT CLASS VARIABLES AND METHODS
- ▶ MY SOLUTION FOR **RATIONAL**
- ▶ (OVERLOADING) OPERATORS (LIKE **+**, *****, **<<**)
- ▶ CONTAINER CLASS
 - STACK EXAMPLE
- ▶ DESTRUCTORS AND THEIR IMPLICIT INVOCATION
- ▶ USING **new** TO HEAP-ALLOCATE OBJECTS
- ▶ INVOKING DESTRUCTOR WITH **delete**

PLAN FOR HOMEWORK 09; MORE LOGISTICS

NOTE: Homework 09 will be published sometime tomorrow

- It will have you devise or modify object classes and their clients.

MENTORS

- Since we've done away with lab, my TAs will be *adopting* each of you.
 - Each of you will be assigned a Homework 09 *mentor*.
 - They will check in with you over the next week.

TUTORING AND OFFICE HOURS

- Canceling "evening drop-in tutoring"...
- ...we'll instead post/email a TA drop-in schedule for next week.
- I'll still hold my office hours Tuesdays and Thursdays. **SLACK too!**

CORRECTION ON LECTURE 09-2

NOTE: My code and slides in the last lecture incorrectly stated that you need to use the static keyword in the definitions of class variables and methods.

▶ This is both true and false:

- You need them in the class definition (normally in the `.hh`)
- You don't want them in the variable and method definitions (i.e. in the `.cc`)

→ Let's correct those slides....

IMPLEMENTING STATIC MEMBERS

```
1. static const double Cmpx::kEpsilon = 0.000001;

3. static const Cmpx Cmpx::I {0.0,1.0};

5. static bool Cmpx::parse(string s, double& rp, double& ip) {
6.     ...
7. }
8. static Cmpx Cmpx::product(Cmpx z1, Cmpx z2) {
9.     ...
10. }
```

When we define what these members mean, we label them as **static**.

- Each of their declared names start with the **class name prefix**.
- Class method code won't access **this**; there is no particular receiver.

IMPLEMENTING STATIC MEMBERS

```
1. static const double Cmpx::kEpsilon = 0.000001;

3. static const Cmpx Cmpx::I {0.0,1.0};

5. static bool Cmpx::parse(string s, double& rp, double& ip) {
6.     ...
7. }
8. static Cmpx Cmpx::product(Cmpx z1, Cmpx z2) {
9.     ...
10. }
```

~~When we define what these members mean, we label them as static.~~

- Each of their declared names start with the class name prefix.
- Class method code won't access **this**; there is no particular receiver.

CORRECTION: IMPLEMENTING CLASS MEMBERS

```
1. const double Cmpx::kEpsilon = 0.000001;

3. const Cmpx Cmpx::I {0.0,1.0};

5. bool Cmpx::parse(string s, double& rp, double& ip) {
6.     ...
7. }
8. Cmpx Cmpx::product(Cmpx z1, Cmpx z2) {
9.     ...
10. }
```

When we define these class members, we **do not** label them as static.

- Each of their declared names start with the **class name prefix**.
- **Class method code** won't access **this**; there is no particular receiver.

CORRECTION: IMPLEMENTING CLASS MEMBERS

```
1. const double Cmpx::kEpsilon = 0.000001;

3. const Cmpx Cmpx::I {0.0,1.0};

5. bool Cmpx::parse(string s, double& rp, double& ip) {
6.     ...
7. }
8. Cmpx Cmpx::product(Cmpx z1, Cmpx z2) {
9.     ...
10. }
```

When we define these class members, we **do not** label them as static.

- Each of their declared names start with the **class name prefix**.
- **Class method code** won't access **this**; there is no particular receiver.

NOTE: My example code got this right. My slides Wednesday didn't.

OPERATORS AND DESTRUCTORS

LECTURE 09-3

JIM FIX, REED COLLEGE CS2-S20

EXERCISE SOLUTION: MY RATIONAL CLASS CLIENT

```
1. #include <iostream>
2. #include "Rational.hh"
3.
4. int main() {
5.     std::string s1,s2;
6.     std::cout << "Enter a rational number: ";
7.     std::cin >> s1;
8.     std::cout << "Enter another rational number: ";
9.     std::cin >> s2;
10.
11.     Rational q1 {s1};
12.     Rational q2 {s2};
13.     Rational sum = q1.plus(q2);
14.     Rational product = q1.times(q2);
15.
16.     std::cout << q1.to_string() << std::endl;
17.     std::cout << q2.to_string() << std::endl;
18.     std::cout << sum.to_string() << std::endl;
19.     std::cout << product.to_string() << std::endl;
20. }
```

MY RATIONAL CLASS SPEC

```
1. class Rational {
2.
3. private:
4.     int num;
5.     int den;
6.
7. public:
8.
9.     // constructors
10.    Rational(void);
11.    Rational(std::string s);
12.    Rational(int n, int d);
13.    Rational(const Rational& q);
14.
15.    // methods
16.    Rational plus(Rational that);
17.    Rational times(Rational that);
18.    std::string to_string(void);
19.};
```

MY RATIONAL CLASS CONSTRUCTORS

```
1. Rational::Rational(int n, int d) {
2.     if (d == 0) {
3.         n = 0;
4.         d = 1;
5.     }
6.     if (d < 0) {
7.         n *= -1;
8.         d *= -1;
9.     }
10.    num = n;
11.    den = d;
12. }
13.
14. Rational::Rational(void) : num {0}, den {1}
15. { }
16.
17. Rational::Rational(const Rational& q) :
18.    num {q.num}, den {q.den}
19. { }
```

MY RATIONAL CLASS CONSTRUCTORS INITIALIZING FIELDS

```
1. Rational::Rational(int n, int d) {
2.     if (d == 0) {
3.         n = 0;
4.         d = 1;
5.     }
6.     if (d < 0) {
7.         n *= -1;
8.         d *= -1;
9.     }
10.    num = n;
11.    den = d;
12. }
13.
14. Rational::Rational(void) : num {0}, den {1}
15. { }
16.
17. Rational::Rational(const Rational& q) :
18.    num {q.num}, den {q.den}
19. { }
```

MY RATIONAL CLASS CONSTRUCTORS CALLING CONSTRUCTORS

```
1. Rational::Rational(int n, int d) {
2.     if (d == 0) {
3.         n = 0;
4.         d = 1;
5.     }
6.     if (d < 0) {
7.         n *= -1;
8.         d *= -1;
9.     }
10.    num = n;
11.    den = d;
12. }
13.
14. Rational::Rational(void) : Rational {0,1}
15. { }
16.
17. Rational::Rational(const Rational& q) :
18.    Rational {q.num,q.den}
19. { }
```

MY RATIONAL CLASS CONSTRUCTORS (CONT'D)

```
1. Rational::Rational(std::string s) { // NOTE: simplified from samples
2.     std::string s_num = "";
3.     std::string s_den = "1";
4.     bool saw_slash = false'
5.
6.     for (int i=0; i<s.length(); i++) {
7.         char c = s[i];
8.         if (c >= '0' && c <= '9') {
9.             if (saw_slash) {
10.                s_den += c;
11.            } else {
12.                s_num += c;
13.            }
14.        } else if (c == '-' && i == 0) {
15.            s_num += c;
16.        } else if (c == '/') {
17.            s_den = "";
18.            saw_slash = true;
19.        }
20.    }
21.
22.    num = std::stoi(s_num);
23.    den = std::stoi(s_den);
24. }
```

parses string

MY RATIONAL CLASS CONSTRUCTORS (CONT'D)

```
1. Rational::Rational(std::string s) { // NOTE: simplified from samples
2.     std::string s_num = "";
3.     std::string s_den = "1";
4.     bool saw_slash = false'
5.
6.     for (int i=0; i<s.length(); i++) {
7.         char c = s[i];
8.         if (c >= '0' && c <= '9') {
9.             if (saw_slash) {
10.                 s_den += c;
11.             } else {
12.                 s_num += c;
13.             }
14.         } else if (c == '-' && i == 0) {
15.             s_num += c;
16.         } else if (c == '/') {
17.             s_den = "";
18.             saw_slash = true;
19.         }
20.     }
21.
22.     num = std::stoi(s_num);
23.     den = std::stoi(s_den);
24. }
```

parses string

initializes the fields

MY RATIONAL CLASS METHODS

```
1. Rational Rational::plus(Rational q) {
2.     return Rational {num*q.den + den*q.num, den*q.den};
3. }
4.
5. Rational Rational::times(Rational q) {
6.     return Rational {num*q.num, den*q.den};
7. }
8.
9. std::string Rational::to_string(void) {
10.    if (den == 1) {
11.        return std::to_string(num);
12.    } else {
13.        return std::to_string(num) + "/" + std::to_string(den);
14.    }
15. }
```

NOTE AGAIN HOW ARITHMETIC IS EXPRESSED

```
1. #include <iostream>
2. #include "Rational.hh"
3.
4. int main() {
5.     std::string s1,s2;
6.     std::cout << "Enter a rational number: ";
7.     std::cin >> s1;
8.     std::cout << "Enter another rational number: ";
9.     std::cin >> s2;
10.
11.     Rational q1 {s1};
12.     Rational q2 {s2};
13.     Rational sum = q1.plus(q2);
14.     Rational product = q1.times(q2);
15.
16.     std::cout << q1.to_string() << std::endl;
17.     std::cout << q2.to_string() << std::endl;
18.     std::cout << sum.to_string() << std::endl;
19.     std::cout << product.to_string() << std::endl;
20. }
```

NOTE AGAIN HOW INPUT AND OUTPUT ARE EXPRESSED

```
1. #include <iostream>
2. #include "Rational.hh"
3.
4. int main() {
5.     std::string s1,s2;
6.     std::cout << "Enter a rational number: ";
7.     std::cin >> s1;
8.     std::cout << "Enter another rational number: ";
9.     std::cin >> s2;
10.
11.     Rational q1 {s1};
12.     Rational q2 {s2};
13.     Rational sum = q1.plus(q2);
14.     Rational product = q1.times(q2);
15.
16.     std::cout << q1.to_string() << std::endl;
17.     std::cout << q2.to_string() << std::endl;
18.     std::cout << sum.to_string() << std::endl;
19.     std::cout << product.to_string() << std::endl;
20. }
```

BUT WHAT IF WE COULD JUST...

```
1. #include <iostream>
2. #include "Rational.hh"
3.
4. int main() {
5.     std::string s1,s2;
6.     std::cout << "Enter a rational number: ";
7.     std::cin >> s1;
8.     std::cout << "Enter another rational number: ";
9.     std::cin >> s2;
10.
11.     Rational q1 {s1};
12.     Rational q2 {s2};
13.     Rational sum = q1.plus(q2);
14.     Rational product = q1.times(q2);
15.
16.     std::cout << q1.to_string() << std::endl;
17.     std::cout << q2.to_string() << std::endl;
18.     std::cout << sum.to_string() << std::endl;
19.     std::cout << product.to_string() << std::endl;
20. }
```

BUT WHAT IF WE COULD JUST... MAKE IT ALL LOOK OFFICIAL

```
1. #include <iostream>
2. #include "Rational.hh"
3.
4. int main() {
5.     Rational q1,q2;
6.     std::cout << "Enter a rational number: ";
7.     std::cin >> q1;
8.     std::cout << "Enter another rational number: ";
9.     std::cin >> q2;
10.
11.
12.
13.     Rational sum = q1+q2;
14.     Rational product = q1*q2;
15.
16.     std::cout << q1 << std::endl;
17.     std::cout << q2 << std::endl;
18.     std::cout << sum << std::endl;
19.     std::cout << product << std::endl;
20. }
```

OVERLOADING OPERATORS TO TAKE RATIONAL VALUES

```
1. #include <iostream>
2. #include "Rational.hh"
3.
4. int main() {
5.     Rational q1;
6.     Rational q2;
7.
8.     std::cout << "Enter a rational number: ";
9.     std::cin >> q1;
10.    std::cout << "Enter another rational number: ";
11.    std::cin >> q2;
12.
13.    std::cout << "The first was " << q1 << "." << std::endl;
14.    std::cout << "The second was " << q2 << "." << std::endl;
15.    std::cout << "Their sum is " << (q1 + q2) << "." << std::endl;
16.    std::cout << "Product is " << (q1 * q2) << "." << std::endl;
17. }
```

MY RATIONAL CLIENT USING OVERLOADED OPERATORS

```
1. #include <iostream>
2. #include "Rational.hh"
3.
4. int main() {
5.     Rational q1;
6.     Rational q2;
7.
8.     std::cout << "Enter a rational number: ";
9.     std::cin >> q1;
10.    std::cout << "Enter another rational number: ";
11.    std::cin >> q2;
12.
13.    std::cout << "The first was " << q1 << "." << std::endl;
14.    std::cout << "The second was " << q2 << "." << std::endl;
15.    std::cout << "Their sum is " << (q1 + q2) << "." << std::endl;
16.    std::cout << "Product is " << (q1 * q2) << "." << std::endl;
17. }
```

ADDITIONS TO RATIONAL.HH FOR OVERLOADED OPERATORS

```
1. class Rational {
2. private:
3.     int num;
4.     int den;
5. public:
6.     ...
7.     // methods
8.     Rational plus(Rational that);
9.     Rational times(Rational that);
10.    std::string to_string(void) const;
11. };
12.
13. Rational operator+(Rational q1, Rational q2);
14. Rational operator*(Rational q1, Rational q2);
15.
16. std::ostream& operator<<(std::ostream& os, const Rational& q);
17. std::istream& operator>>(std::istream& is, Rational& q);
```


ADDITIONS TO RATIONAL.HH FOR OVERLOADED OPERATORS

```
1. class Rational {
2. private:
3.     int num;
4.     int den;
5. public:
6.     ...
7.     // methods
8.     Rational plus(Rational that);
9.     Rational times(Rational that);
10.    std::string to_string(void) const;
11. };
12.
13. Rational operator+(Rational q1, Rational q2);
14. Rational operator*(Rational q1, Rational q2);
15.
16. std::ostream& operator<<(std::ostream& os, const Rational& q);
17. std::istream& operator>>(std::istream& is, Rational& q);
```

USE OF BINARY OPERATIONS

► In C++ code, `+`, `*`, `>>`, `<<` are binary operations.

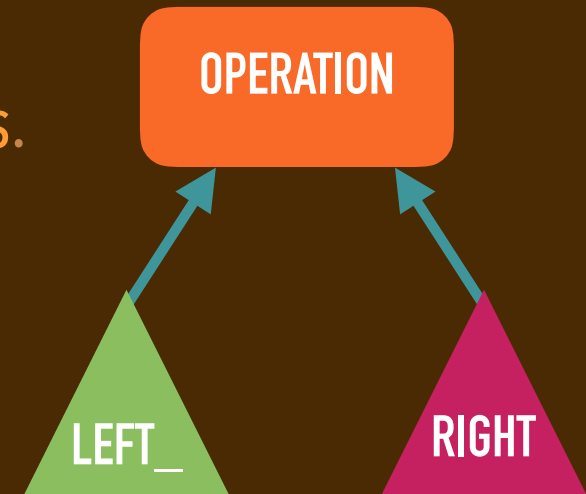
```
1. Rational q1;  
2.     Rational q2;  
3.     std::cin >> q1;  
4.     std::cin >> q2;  
5.     Rational sum = (q1 + q2);  
6.     std::cout << sum;
```



USE OF BINARY OPERATIONS

► In C++ code, `+`, `*`, `>>`, `<<` are binary operations.

```
1. Rational q1;  
2. Rational q2;  
3. std::cin >> q1;  
4. std::cin >> q2;  
5. Rational sum = (q1 + q2);  
6. std::cout << sum;
```

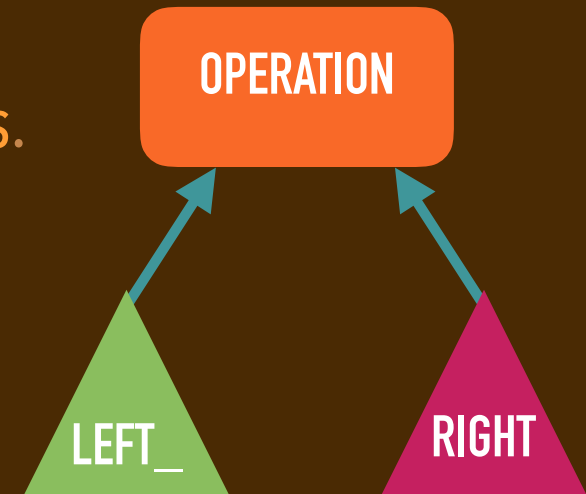


► This means that they are viewed as functions that take two arguments.

USE OF BINARY OPERATIONS

► In C++ code, `+`, `*`, `>>`, `<<` are binary operations.

```
1. Rational q1;  
2.   Rational q2;  
3.   std::cin >> q1;  
4.   std::cin >> q2;  
5.   Rational sum = (q1 + q2);  
6.   std::cout << sum;
```



► This means that they are viewed as functions that take two arguments.

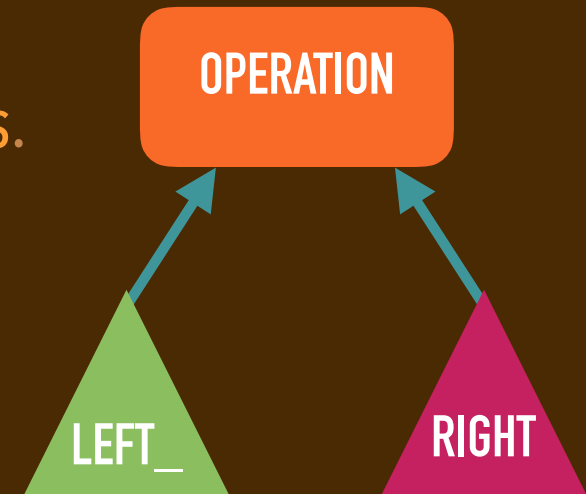
```
Rational operator+(Rational q1, Rational q2);  
Rational operator*(Rational q1, Rational q2);
```

```
std::ostream& operator<<(std::ostream& os, const Rational& q);  
std::istream& operator>>(std::istream& is, Rational& q);
```

USE OF BINARY OPERATIONS

► In C++ code, `+`, `*`, `>>`, `<<` are binary operations.

```
1. Rational q1;  
2.   Rational q2;  
3.   std::cin >> q1;  
4.   std::cin >> q2;  
5.   Rational sum = (q1 + q2);  
6.   std::cout << sum;
```



► This means that they are viewed as functions that take two arguments.

```
Rational operator+(Rational q1, Rational q2);  
Rational operator*(Rational q1, Rational q2);
```

```
std::ostream& operator<<(std::ostream& os, const Rational& q);  
std::istream& operator>>(std::istream& is, Rational& q);
```

LEFT ASSOCIATIVITY OF PLUS AND TIMES

- Addition and multiplication are left associative. This means that these

$q1 + q2 + q3 + q4$

$q1 * q2 * q3 * q4$

are treated as if they were the expressions

$((q1 + q2) + q3) + q4$

$((q1 * q2) * q3) * q4$

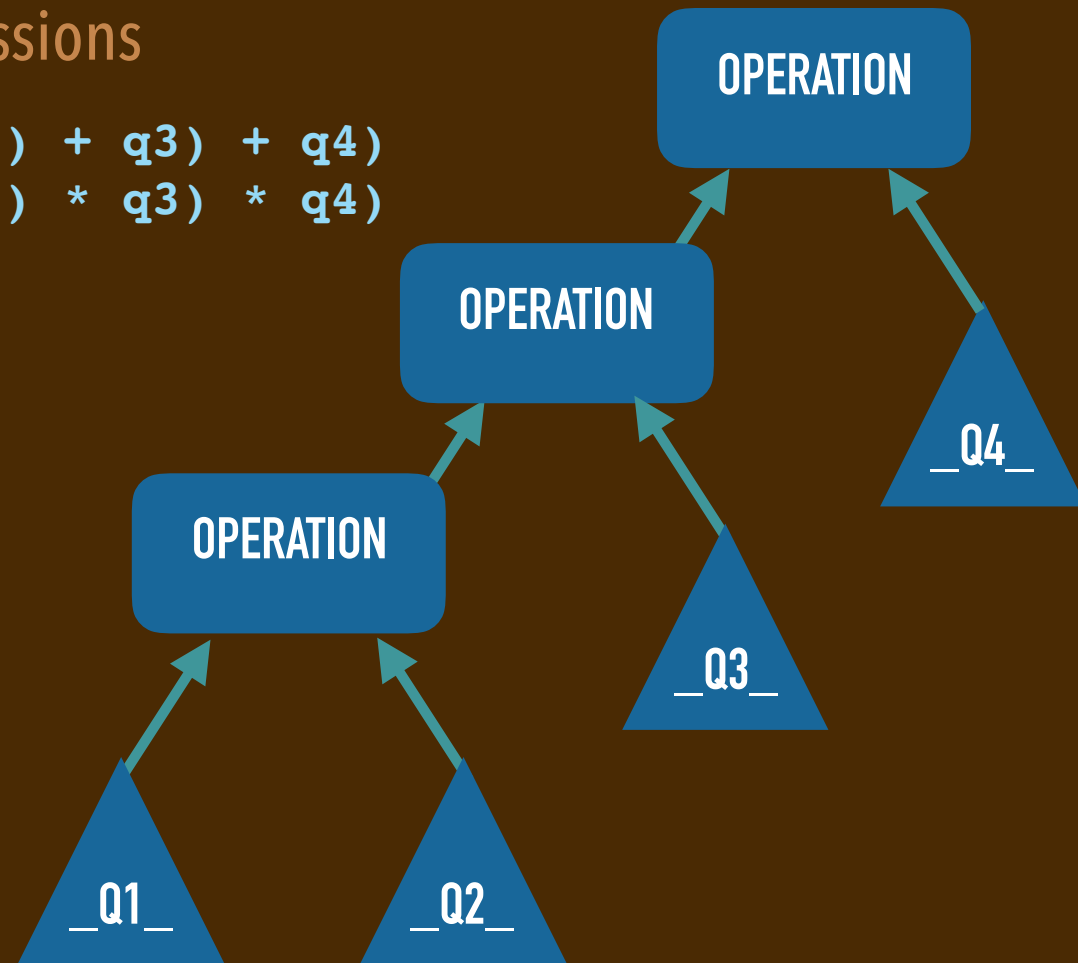
LEFT ASSOCIATIVITY OF PLUS AND TIMES

► Addition and multiplication are left associative. This means that these

$q1 + q2 + q3 + q4$
 $q1 * q2 * q3 * q4$

are treated as if they were the expressions

$((q1 + q2) + q3) + q4$
 $((q1 * q2) * q3) * q4$



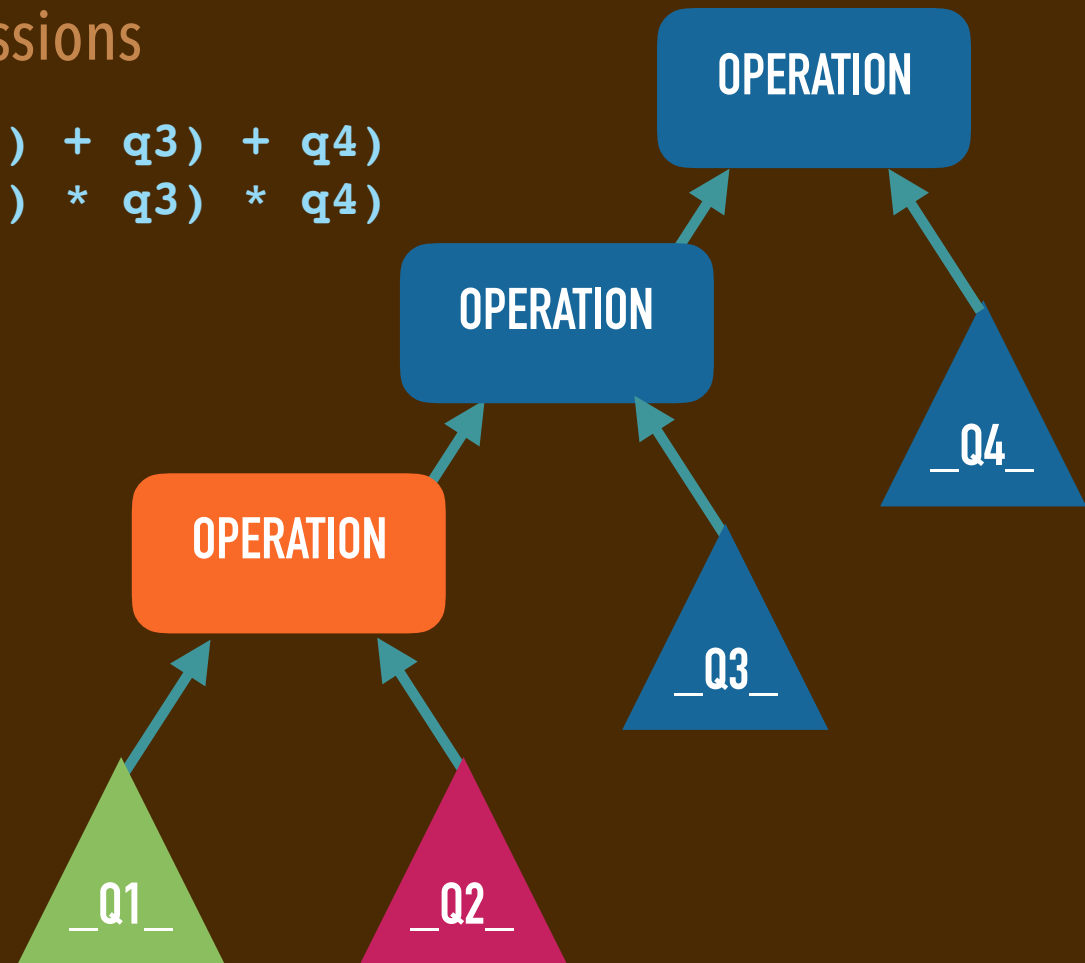
LEFT ASSOCIATIVITY OF PLUS AND TIMES

- Addition and multiplication are left associative. This means that these

$q1 + q2 + q3 + q4$
 $q1 * q2 * q3 * q4$

are treated as if they were the expressions

$((q1 + q2) + q3) + q4$
 $((q1 * q2) * q3) * q4$



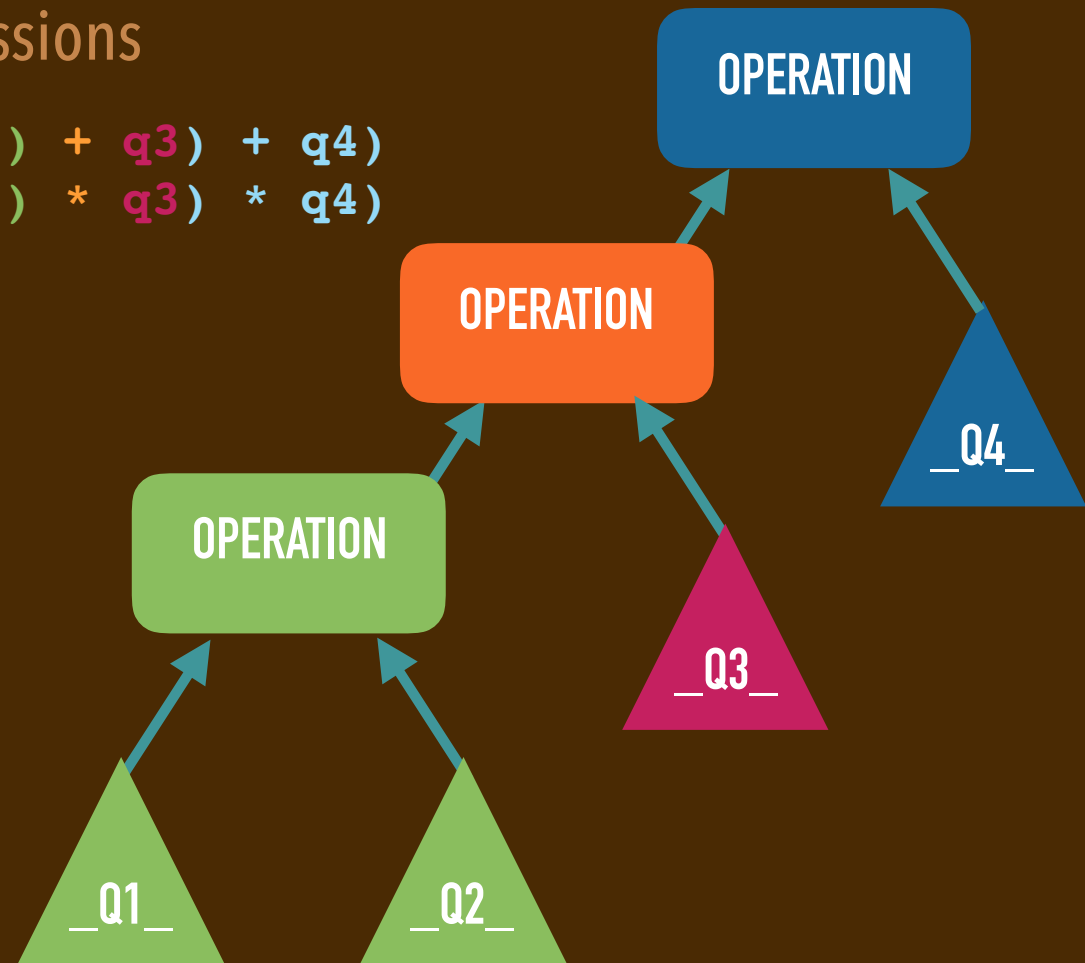
LEFT ASSOCIATIVITY OF PLUS AND TIMES

► Addition and multiplication are left associative. This means that these

$$\begin{array}{ccccccc} q1 & + & q2 & + & q3 & + & q4 \\ q1 & * & q2 & * & q3 & * & q4 \end{array}$$

are treated as if they were the expressions

$$\begin{array}{l} (((q1 + q2) + q3) + q4) \\ (((q1 * q2) * q3) * q4) \end{array}$$



LEFT ASSOCIATIVITY OF INPUT AND OUTPUT

- ▶ The C++ input and output stream operations are left associative, as well.
- ▶ This means that these two statements

```
std::cin >> q1 >> q2;  
std::cout << q1 << q2;
```

are treated as if they were these two statements

```
(std::cin >> q1) >> q2;  
(std::cout << q1) << q2;
```

LEFT ASSOCIATIVITY OF INPUT AND OUTPUT

- ▶ The C++ input and output stream operations are left associative, as well.
- ▶ This means that these two statements

```
std::cin >> q1 >> q2;  
std::cout << q1 << q2;
```

are treated as if they were these two statements

```
(std::cin >> q1) >> q2;  
(std::cout << q1) << q2;
```

- ▶ This means, for example, that **output of q1** produces a stream that gets used for the **output of q2**.

LEFT ASSOCIATIVITY OF INPUT AND OUTPUT

- ▶ The C++ input and output stream operations are left associative, as well.
- ▶ This means that these two statements

```
std::cin >> q1 >> q2;  
std::cout << q1 << q2;
```

are treated as if they were these two statements

```
(std::cin >> q1) >> q2;  
(std::cout << q1) << q2;
```

- ▶ This means, for example, that **output of q1** produces a stream that gets used for the **output of q2**. This explains its signature:

```
std::ostream& operator<<(std::ostream& os, const Rational& q);
```

LEFT ASSOCIATIVITY OF INPUT AND OUTPUT

- ▶ The C++ input and output stream operations are left associative, as well.
- ▶ This means that these two statements

```
std::cin >> q1 >> q2;  
std::cout << q1 << q2;
```

are treated as if they were these two statements

```
(std::cin >> q1) >> q2;  
(std::cout << q1) << q2;
```

- ▶ This means, for example, that **output of q1** produces a stream that gets used for the **output of q2**. This explains its signature:

```
std::ostream& operator<<(std::ostream& os, const Rational& q);
```

IMPLEMENTATION OF THE OPERATORS

```
Rational operator+(Rational q1, Rational q2) {  
    return q1.plus(q2);  
}  
Rational operator*(Rational q1, Rational q2) {  
    return q1.times(q2);  
}  
std::ostream& operator<<(std::ostream& os, const Rational& q) {  
    os << q.to_string();  
    return os;  
}  
std::istream& operator>>(std::istream& is, Rational& q) {  
    std::string s;  
    is >> s;  
    q = Rational {s};  
    return is;  
}
```

IMPLEMENTATION OF THE OPERATORS

```
Rational operator+(Rational q1, Rational q2) {  
    return q1.plus(q2);  
}  
Rational operator*(Rational q1, Rational q2) {  
    return q1.times(q2);  
}  
std::ostream& operator<<(std::ostream& os, const Rational& q) {  
    os << q.to_string();  
    return os;  
}  
std::istream& operator>>(std::istream& is, Rational& q) {  
    std::string s;  
    is >> s;  
    q = Rational {s};  
    return is;  
}
```

►NOTES:

- I've done a bit of work here to use the public interface of Rational.

IMPLEMENTATION OF THE OPERATORS

```
Rational operator+(Rational q1, Rational q2) {  
    return q1.plus(q2);  
}  
Rational operator*(Rational q1, Rational q2) {  
    return q1.times(q2);  
}  
std::ostream& operator<<(std::ostream& os, const Rational& q) {  
    os << q.to_string();  
    return os;  
}  
std::istream& operator>>(std::istream& is, Rational& q) {  
    std::string s;  
    is >> s;  
    q = Rational {s};  
    return is;  
}
```

►NOTES:

- I've done a bit of work here to use the public interface of Rational.
- Most examples of these have them each declared as a friend of the data class.

IMPLEMENTATION OF THE OPERATORS

```
Rational operator+(Rational q1, Rational q2) {  
    return q1.plus(q2);  
}  
Rational operator*(Rational q1, Rational q2) {  
    return q1.times(q2) Rational {q1.num*q2.num,q1.den*q2.den};  
}  
std::ostream& operator<<(std::ostream& os, const Rational& q) {  
    os << q.to_string();  
    return os;  
}  
std::istream& operator>>(std::istream& is, Rational& q) {  
    std::string s;  
    is >> s;  
    q = Rational {s};  
    return is;  
}
```

►NOTES:

- I've done a bit of work here to use the public interface of Rational.
- Most examples of these have them each declared as a friend of the data class.

IMPLEMENTATION OF THE OPERATORS

```
Rational operator+(Rational q1, Rational q2) {  
    return q1.plus(q2);  
}  
Rational operator*(Rational q1, Rational q2) {  
    return q1.times(q2) Rational {q1.num*q2.num,q1.den*q2.den};  
}  
std::ostream& operator<<(std::ostream& os, const Rational& q) {  
    os << "Rational {num=" << q.num << ", den=" << q.den << "}  
    return os;  
}  
std::istream& operator>>(std::istream& is, Rational& q) {  
    std::string s;  
    is >> s;  
    q = Rational {s};  
    return is;  
}
```

►NOTES:

- I've done a bit of work here to use the public interface of Rational.
- Most examples of these have them each declared as a friend of the data class.

IMPLEMENTATION OF THE OPERATORS

```
Rational operator+(Rational q1, Rational q2) {  
    return q1.plus(q2);  
}  
Rational operator*(Rational q1, Rational q2) {  
    return q1.times(q2);  
}  
std::ostream& operator<<(std::ostream& os, const Rational& q) {  
    os << q.to_string();  
    return os;  
}  
std::istream& operator>>(std::istream& is, Rational& q) {  
    std::string s;  
    is >> s;  
    q = Rational {s};  
    return is;  
}
```

►NOTES:

- Both << and >> return the stream that they operate on.

IMPLEMENTATION OF THE OPERATORS

```
Rational operator+(Rational q1, Rational q2) {
    return q1.plus(q2);
}
Rational operator*(Rational q1, Rational q2) {
    return q1.times(q2);
}
std::ostream& operator<<(std::ostream& os, const Rational& q) {
    os << q.to_string();
    return os;
}
std::istream& operator>>(std::istream& is, Rational& q) {
    std::string s;
    is >> s;
    q = Rational {s};
    return is;
}
```

►NOTES:

- Both << and >> return the stream that they operate on.
- There is a lot of other (important) **window dressing** here. We'll discuss soon...

IMPLEMENTATION OF THE OPERATORS

```
Rational operator+(Rational q1, Rational q2) {  
    return q1.plus(q2);  
}  
Rational operator*(Rational q1, Rational q2) {  
    return q1.times(q2);  
}  
std::ostream& operator<<(std::ostream& os, const Rational& q) {  
    os << q.to_string();  
    return os;  
}  
std::istream& operator>>(std::istream& is, Rational& q) {  
    std::string s;  
    is >> s;  
    q = Rational {s};  
    return is;  
}
```

`std::string to_string(void) const;`

►NOTES:

- Both << and >> return the stream that they operate on.
- There is a lot of other (important) **window dressing** here. We'll discuss soon...

WE CAN DEFINE OPERATOR METHODS

```
1. class Rational {  
2. private:  
3.     int num;  
4.     int den;  
5. public:  
6.     ...  
7.     Rational operator-(void);  
8.     int operator[](std::string s);  
9.     ...  
10.};
```

In the above, I'm overloading **unary minus** and **indexing**.

OPERATOR METHODS

```
1. class Rational {
2. private:
3.     int num;
4.     int den;
5. public:
6.     ...
7.     Rational operator-(void);
8.     int operator[](std::string s);
9.     ...
10.};
```

In the above, I'm overloading **unary minus** and **indexing**. Their code:

```
Rational Rational::operator-(void) {
    return Rational {-num,den};
}
int Rational::operator[](std::string s) {
    if (s == "numerator") return num;
    if (s == "denominator") return den;
    return 0;
}
```

OPERATOR METHODS

```
1. class Rational {
2. private:
3.     int num;
4.     int den;
5. public:
6.     ...
7.     Rational operator-(void);
8.     int operator[](std::string s);
9.     ...
10.};
```

In the above, I'm overloading **unary minus** and **indexing**.

→ A client can use these in this way:

```
std::cout << "The first was " << q1 << "." << std::endl;
std::cout << "Its negation is " << -q1 << std::endl;
std::cout << "The second was " << q2 << "." << std::endl;
std::cout << "Its numerator is " << q2["numerator"] << ".\n";
```


SUMMARY OF OPERATORS

- ▶ C++ let's you define and overload binary and unary operators.
 - Examples: `+`, `*`, binary `-`, `/`, `<<`, `>>`, unary `-`, `[]`, and many others
- ▶ Prefix them with **operator** keyword in type signatures and definitions.
- ▶ Some are functions. Others are methods.
 - May need to make it a **friend** of the class, if a function.
- ▶ Some require **const** and **&** annotations to meet their C++ spec.
 - We'll cover their meaning soon.
- ▶ Tricky ones we'll cover later: `==`, `=`, `()`

RECALL: MEMORY MANAGEMENT

- ▶ Structs can live *on the stack* or *in the heap*.
- ▶ Function parameters and local variables live on the stack frame.
 - Allocated upon function entry.
 - De-allocated upon function exit. The stack frame is "taken down."
- ▶ The heap is for "dynamic allocation", usually for longer lifetimes.
 - We use **new** to allocate them on the heap. Gives us a *pointer*.
 - We use **delete** to give back their storage to the heap.
 - ➔ Without an explicit delete, their memory is "*leaked*."
- ▶ **NOTE:** The same is true for objects that are instances of a class.

OBJECT MEMORY MANAGEMENT

- ▶ Objects can also live *on the stack* or *in the heap*.
- ▶ But typically objects have more functionality, behavior, and they are often in relation with other data.
- ▶ Upon construction, we set up the object's data and these relations.
 - In particular, may need allocate subcomponents on the heap.
 - **CONSTRUCTORS** do that work.
- ▶ When done with an object need to perform cleanup...
 - In particular, may need to give back heap-allocated components.
 - Define **DESTRUCTORS** to perform this clean-up.

CONTAINER EXAMPLE: A STACK OBJECT CLASS

```
1. class Stck {  
  
3. private:  
4.     int *elements;  
5.     int num_elements;  
6.     int capacity;  
  
8. public:  
9.     Stck(int capacity);  
10.    bool is_empty();  
11.    void push(int value);  
12.    int pop();  
13.    int top();  
14.    ~Stck();  
15.};
```

CONTAINER EXAMPLE: A STACK OBJECT CLASS

```
1. class Stck {  
  
3. private:  
4.     int *elements; // this will be an array of size capacity  
5.     int num_elements;  
6.     int capacity;  
  
8. public:  
9.     Stck(int capacity);  
10.    bool is_empty();  
11.    void push(int value);  
12.    int pop();  
13.    int top();  
14.    ~Stck();  
15.};
```

CONTAINER EXAMPLE: A STACK OBJECT CLASS

```
1. class Stck {  
  
3. private:  
4.     int *elements;  
5.     int num_elements;  
6.     int capacity;  
  
8. public:  
9.     Stck(int capacity); # This will heap-allocate the array.  
10.    bool is_empty();  
11.    void push(int value);  
12.    int pop();  
13.    int top();  
14.    ~Stck();  
15.};
```

CONTAINER EXAMPLE: A STACK OBJECT CLASS

```
1. class Stck {  
  
3. private:  
4.     int *elements;  
5.     int num_elements;  
6.     int capacity;  
  
8. public:  
9.     Stck(int capacity); # This will heap-allocate the array.  
10.    bool is_empty();  
11.    void push(int value);  
12.    int pop();  
13.    int top();  
14.    ~Stck(); # Destructor. This will "delete" the array.  
15.};
```

IMPLEMENTATION OF THE CONSTRUCTOR

```
1. #include "Stck.hh"
2.
3. Stck::Stck(int capacity) {
4.     this->elements = new int[capacity];
5.     this->num_elements = 0,
6.     this->capacity = capacity;
7. }
```


IMPLEMENTATION USING A CONSTRUCTOR INITIALIZER LIST

```
1. #include "Stck.hh"
2.
3. Stck::Stck(int capacity) :
4.     elements {new int[capacity]},
5.     num_elements {0},
6.     capacity {capacity}
7. { }
```

IMPLEMENTATION OF STACK METHODS

```
9.  bool Stck::is_empty() {
10.    return (num_elements == 0);
11. }
12.
13. void Stck::push(int value) {
14.    assert(num_elements < capacity);
15.    elements[num_elements] = value;
16.    num_elements++;
17. }
18.
19. int Stck::pop() {
20.    assert(!is_empty());
21.    num_elements--;
22.    return elements[num_elements];
23. }
24.
25. int Stck::top() {
26.    assert(!is_empty());
27.    return elements[num_elements-1];
28. }
```

ILLUSTRATION WITH A SIMPLE CLIENT

```
9. #include "Stck.hh"
10. #include <iostream>
11. int main(void) {
12.     Stck s {5};
13.     s.push(7);
14.     s.push(1);
15.     s.push(3);
16.     std::cout << s.pop() << std::endl;
17.     std::cout << s.pop() << std::endl;
18.     s.push(11);
19.     std::cout << s.pop() << std::endl;
20. }
```

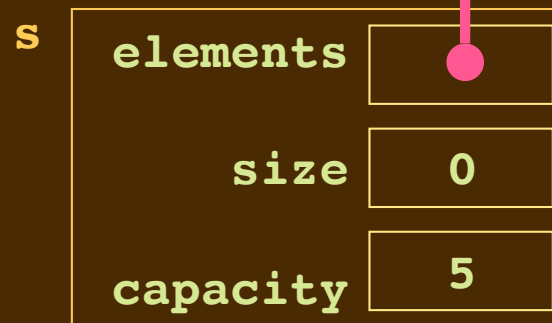
ILLUSTRATION WITH A SIMPLE CLIENT

```
9. #include "Stck.hh"
10. #include <iostream>
11. int main(void) {
12.     Stck s {5};
13.     s.push(7);
14.     s.push(1);
15.     s.push(3);
16.     std::cout << s.pop() << std::endl;
17.     std::cout << s.pop() << std::endl;
18.     s.push(11);
19.     std::cout << s.pop() << std::endl;
20. }
```

CONSOLE

```
1
2
3
4
5
```

STACK FRAME



HEAP MEMORY

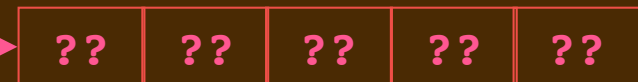


ILLUSTRATION WITH A SIMPLE CLIENT

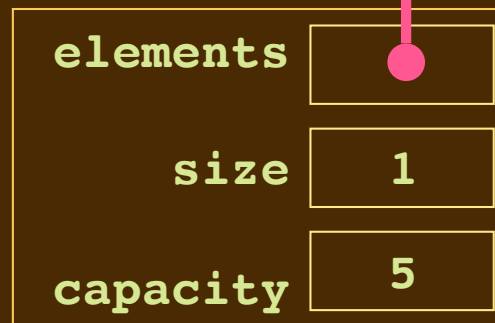
```
9. #include "Stck.hh"
10. #include <iostream>
11. int main(void) {
12.     Stck s {5};
13.     s.push(7);
14.     s.push(1);
15.     s.push(3);
16.     std::cout << s.pop() << std::endl;
17.     std::cout << s.pop() << std::endl;
18.     s.push(11);
19.     std::cout << s.pop() << std::endl;
20. }
```

CONSOLE

```
1
2
3
4
5
```

STACK FRAME

s



HEAP MEMORY

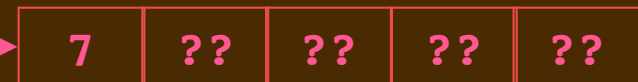


ILLUSTRATION WITH A SIMPLE CLIENT

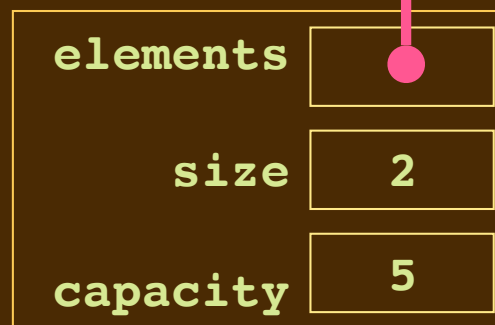
```
9. #include "Stck.hh"
10. #include <iostream>
11. int main(void) {
12.     Stck s {5};
13.     s.push(7);
14.     s.push(1);
15.     s.push(3);
16.     std::cout << s.pop() << std::endl;
17.     std::cout << s.pop() << std::endl;
18.     s.push(11);
19.     std::cout << s.pop() << std::endl;
20. }
```

CONSOLE

```
1
2
3
4
5
```

STACK FRAME

s



HEAP MEMORY

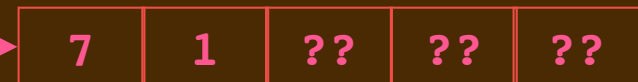


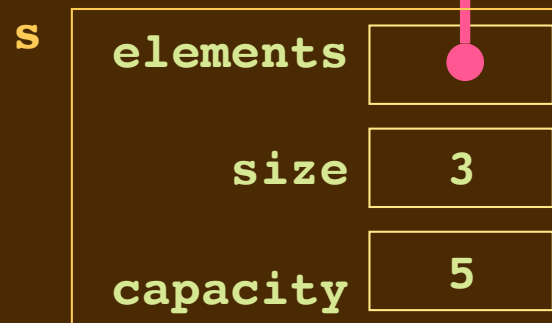
ILLUSTRATION WITH A SIMPLE CLIENT

```
9. #include "Stck.hh"
10. #include <iostream>
11. int main(void) {
12.     Stck s {5};
13.     s.push(7);
14.     s.push(1);
15.     s.push(3);
16.     std::cout << s.pop() << std::endl;
17.     std::cout << s.pop() << std::endl;
18.     s.push(11);
19.     std::cout << s.pop() << std::endl;
20. }
```

CONSOLE

```
1
2
3
4
5
```

STACK FRAME



HEAP MEMORY



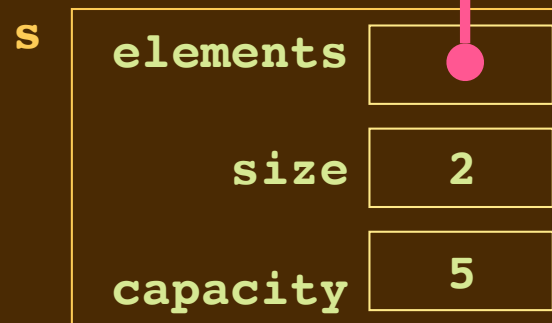
ILLUSTRATION WITH A SIMPLE CLIENT

```
9. #include "Stck.hh"
10. #include <iostream>
11. int main(void) {
12.     Stck s {5};
13.     s.push(7);
14.     s.push(1);
15.     s.push(3);
16.     std::cout << s.pop() << std::endl;
17.     std::cout << s.pop() << std::endl;
18.     s.push(11);
19.     std::cout << s.pop() << std::endl;
20. }
```

CONSOLE

```
1 3
2
3
4
5
```

STACK FRAME



HEAP MEMORY

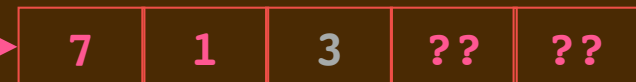


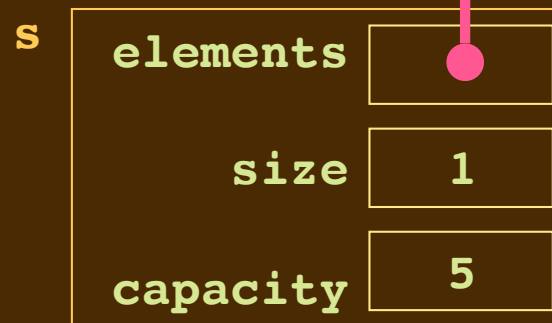
ILLUSTRATION WITH A SIMPLE CLIENT

```
9. #include "Stck.hh"
10. #include <iostream>
11. int main(void) {
12.     Stck s {5};
13.     s.push(7);
14.     s.push(1);
15.     s.push(3);
16.     std::cout << s.pop() << std::endl;
17.     std::cout << s.pop() << std::endl;
18.     s.push(11);
19.     std::cout << s.pop() << std::endl;
20. }
```

CONSOLE

```
1 3
2 1
3
4
5
```

STACK FRAME



HEAP MEMORY

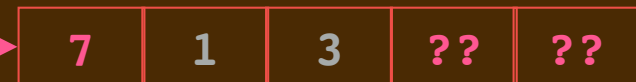


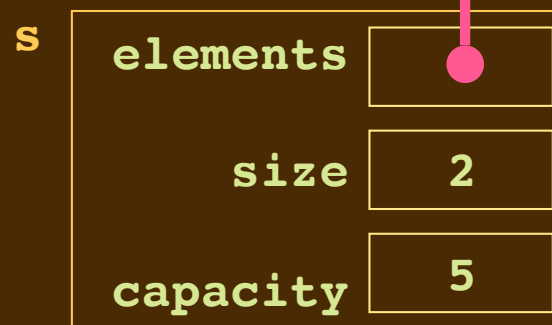
ILLUSTRATION WITH A SIMPLE CLIENT

```
9. #include "Stck.hh"
10. #include <iostream>
11. int main(void) {
12.     Stck s {5};
13.     s.push(7);
14.     s.push(1);
15.     s.push(3);
16.     std::cout << s.pop() << std::endl;
17.     std::cout << s.pop() << std::endl;
18.     s.push(11);
19.     std::cout << s.pop() << std::endl;
20. }
```

CONSOLE

```
1 3
2 1
3
4
5
```

STACK FRAME



HEAP MEMORY

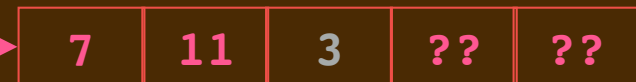


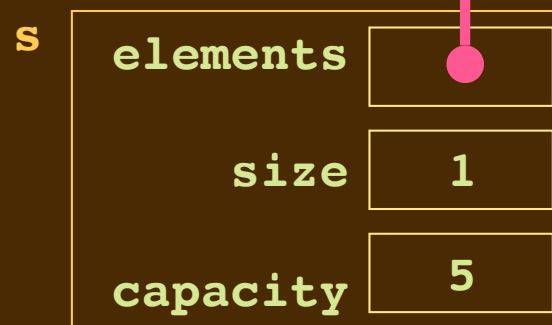
ILLUSTRATION WITH A SIMPLE CLIENT

```
9. #include "Stck.hh"
10. #include <iostream>
11. int main(void) {
12.     Stck s {5};
13.     s.push(7);
14.     s.push(1);
15.     s.push(3);
16.     std::cout << s.pop() << std::endl;
17.     std::cout << s.pop() << std::endl;
18.     s.push(11);
19.     std::cout << s.pop() << std::endl;
20. }
```

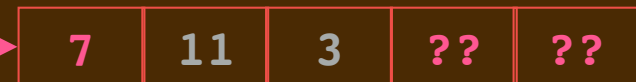
CONSOLE

```
1 3
2 1
3 11
4
5
```

STACK FRAME



HEAP MEMORY



DESTRUCTOR CODE EXECUTION

```
9. #include "Stck.hh"
10. #include <iostream>
11. int main(void) {
12.     Stck s {5};
13.     s.push(7);
14.     s.push(1);
15.     s.push(3);
16.     std::cout << s.pop() << std::endl;
17.     std::cout << s.pop() << std::endl;
18.     s.push(11);
19.     std::cout << s.pop() << std::endl;
20. }
```

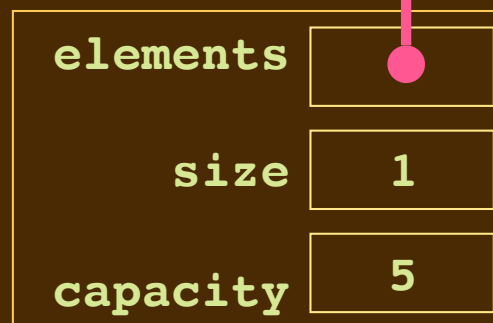
Calls the destructor.

CONSOLE

```
1 3
2 1
3 11
4
5
```

STACK FRAME

s



HEAP MEMORY

7	11	3	??	??
---	----	---	----	----

DESTRUCTOR CODE

- ▶ Destructor code is executed when a stack-allocated object goes out of scope.
- ▶ Here is the code for the `Stck` destructor:

```
Stck::~~Stck() {  
    delete [] elements;  
}
```

DESTRUCTOR CODE

- ▶ Destructor code is executed when a stack-allocated object goes out of scope.
- ▶ Here is the code for the `Stck` destructor:

```
Stck::~~Stck() {  
    delete [] elements;  
}
```

- ▶ In this case, we simply `delete` the pointer to the elements array.
- ▶ If we didn't, we'd have a *memory leak*.
 - The 5 words would be reserved, but the program has no access to them.

DESTRUCTOR CODE

- ▶ Destructor code is executed when a stack-allocated object goes out of scope.
- ▶ Here is the code for the `Stck` destructor:

```
Stck::~~Stck() {  
    delete [] elements;  
}
```

- ▶ In this case, we simply `delete` the pointer to the elements array.
- ▶ If we didn't, we'd have a *memory leak*.
 - The 5 words would be reserved, but the program has no access to them.
- ▶ This just undoes the work of the constructor; gives back the heap storage.

IMPLICIT CALL OF THE DESTRUCTOR

```
9. #include "Stck.hh"
10. #include <iostream>
11. int main(void) {
12.     Stck s {5};
13.     s.push(7);
14.     s.push(1);
15.     s.push(3);
16.     std::cout << s.pop() << std::endl;
17.     std::cout << s.pop() << std::endl;
18.     s.push(11);
19.     std::cout << s.pop() << std::endl;
20. }
```

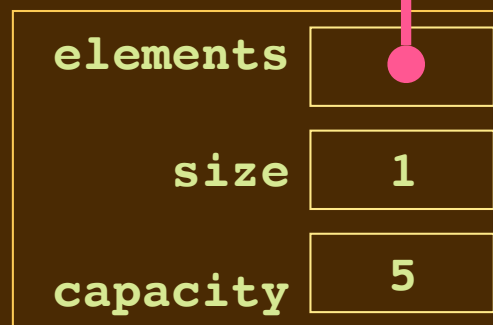
Calls the destructor.

CONSOLE

```
1 3
2 1
3 11
4
5
```

STACK FRAME

s



HEAP MEMORY

7	11	3	??	??
---	----	---	----	----

IMPLICIT CALL OF THE DESTRUCTOR

```
9. #include "Stck.hh"
10. #include <iostream>
11. int main(void) {
12.     Stck s {5};
13.     s.push(7);
14.     s.push(1);
15.     s.push(3);
16.     std::cout << s.pop() << std::endl;
17.     std::cout << s.pop() << std::endl;
18.     s.push(11);
19.     std::cout << s.pop() << std::endl;
20. }
```

Calls the

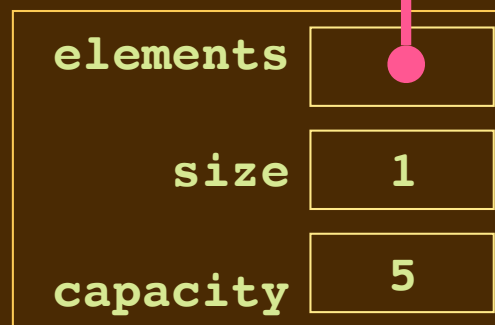
*And the frame gets
taken down.*

CONSOLE

```
1 3
2 1
3 11
4
5
```

STACK FRAME

s



HEAP MEMORY

7	11	3	??	??
---	----	---	----	----

IMPLICIT CALL OF THE DESTRUCTOR

```
9. #include "Stck.hh"
10. #include <iostream>
11. int main(void) {
12.     Stck s {5};
13.     s.push(7);
14.     s.push(1);
15.     s.push(3);
16.     std::cout << s.pop() << std::endl;
17.     std::cout << s.pop() << std::endl;
18.     s.push(11);
19.     std::cout << s.pop() << std::endl;
20. }
```

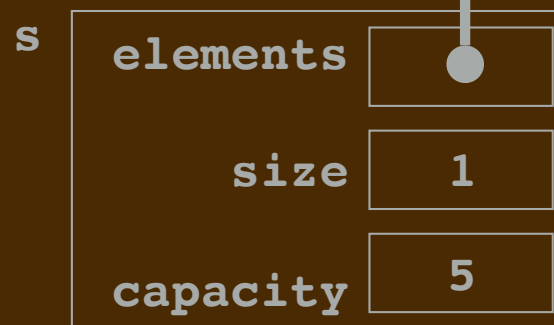
Calls the

*And the frame gets
taken down.*

CONSOLE

```
1 3
2 1
3 11
4
5
```

STACK FRAME



HEAP MEMORY

7	11	3	??	??
---	----	---	----	----

HEAP-ALLOCATED STACK

```
9. #include "Stck.hh"
10. #include <iostream>
11. int main(void) {
12.     Stck* s = new Stck {5};
13.     s->push(7);
14.     s->push(1);
15.     s->push(3);
16.     std::cout << s->pop() << std::endl;
17.     std::cout << s->pop() << std::endl;
18.     s->push(11);
19.     std::cout << s->pop() << std::endl;
20.     delete s;
21. }
```

HEAP-ALLOCATED STACK

```
9. #include "Stck.hh"
10. #include <iostream>
11. int main(void) {
12.     Stck* s = new Stck {5};
13.     s->push(7);
14.     s->push(1);
15.     s->push(3);
16.     std::cout << s->pop() << std::endl;
17.     std::cout << s->pop() << std::endl;
18.     s->push(11);
19.     std::cout << s->pop() << std::endl;
20.     delete s;
21. }
```

► Now **s** is a pointer to a **Stck** instance.

HEAP-ALLOCATED STACK

```
9. #include "Stck.hh"
10. #include <iostream>
11. int main(void) {
12.     Stck* s = new Stck {5};
13.     s->push(7);
14.     s->push(1);
15.     s->push(3);
16.     std::cout << s->pop() << std::endl;
17.     std::cout << s->pop() << std::endl;
18.     s->push(11);
19.     std::cout << s->pop() << std::endl;
20.     delete s;
21. }
```

- ▶ Now **s** can point to a **Stck** instance. Its type is **Stck***
- ▶ We can **construct a new instance** that lives on the heap.

HEAP-ALLOCATED STACK

```
9. #include "Stck.hh"
10. #include <iostream>
11. int main(void) {
12.     Stck* s = new Stck {5};
13.     s->push(7);
14.     s->push(1);
15.     s->push(3);
16.     std::cout << s->pop() << std::endl;
17.     std::cout << s->pop() << std::endl;
18.     s->push(11);
19.     std::cout << s->pop() << std::endl;
20.     delete s;
21. }
```

- ▶ Now **s** can point to a **Stck** instance. Its type is **Stck***
- ▶ We can **construct a new instance** that lives on the heap.
- ▶ And we must explicitly **delete** that pointer.

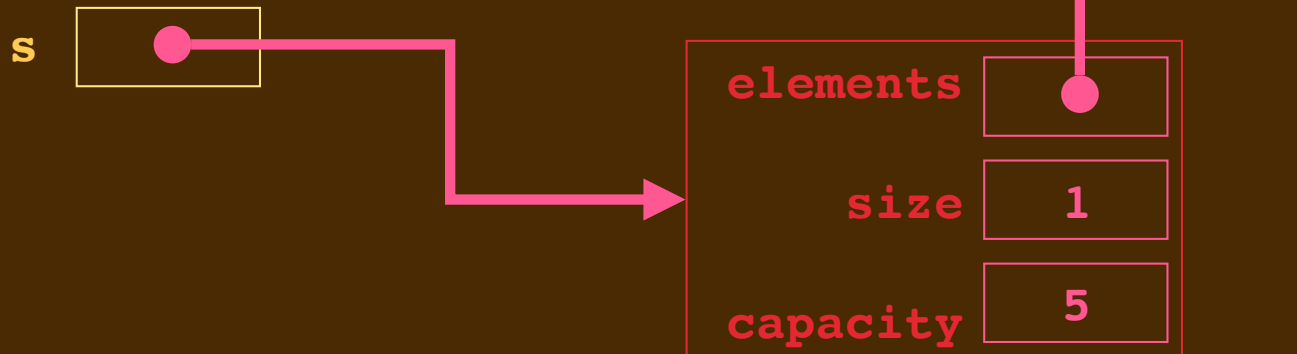
HEAP-ALLOCATED STACK ILLUSTRATED

```
9. #include "Stck.hh"
10. #include <iostream>
11. int main(void) {
12.     Stck* s = new Stck {5};
13.     s->push(7);
14.     s->push(1);
15.     s->push(3);
16.     std::cout << s->pop() << std::endl;
17.     std::cout << s->pop() << std::endl;
18.     s->push(11);
19.     std::cout << s->pop() << std::endl;
20.     delete s;
21. }
```

CONSOLE

```
1
2
3
4
5
```

STACK FRAME



HEAP-ALLOCATED STACK ILLUSTRATED

```
9. #include "Stck.hh"
10. #include <iostream>
11. int main(void) {
12.     Stck* s = new Stck {5};
13.     s->push(7);
14.     s->push(1);
15.     s->push(3);
16.     std::cout << s->pop() << std::endl;
17.     std::cout << s->pop() << std::endl;
18.     s->push(11);
19.     std::cout << s->pop() << std::endl;
20.     delete s;
21. }
```

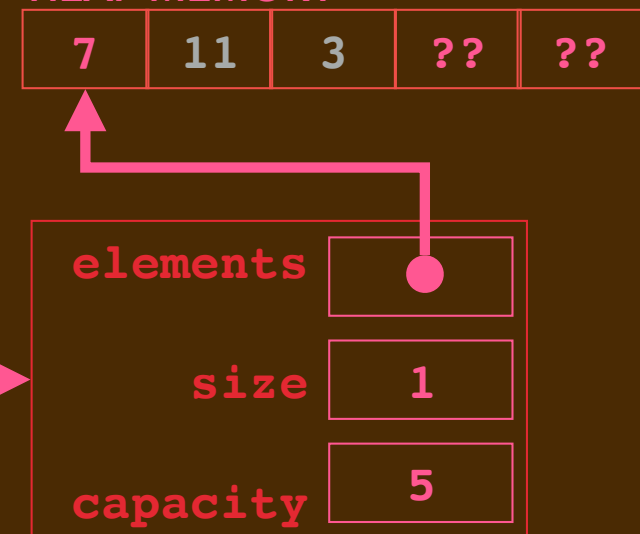
CONSOLE

```
1 3
2 1
3 11
4
5
```

STACK FRAME



HEAP MEMORY



HEAP-ALLOCATED STACK ILLUSTRATED

```
9. #include "Stck.hh"
10. #include <iostream>
11. int main(void) {
12.     Stck* s = new Stck {5};
13.     s->push(7);
14.     s->push(1);
15.     s->push(3);
16.     std::cout << s->pop() << std::endl;
17.     std::cout << s->pop() << std::endl;
18.     s->push(11);
19.     std::cout << s->pop() << std::endl;
20.     delete s;
21. }
```

CONSOLE

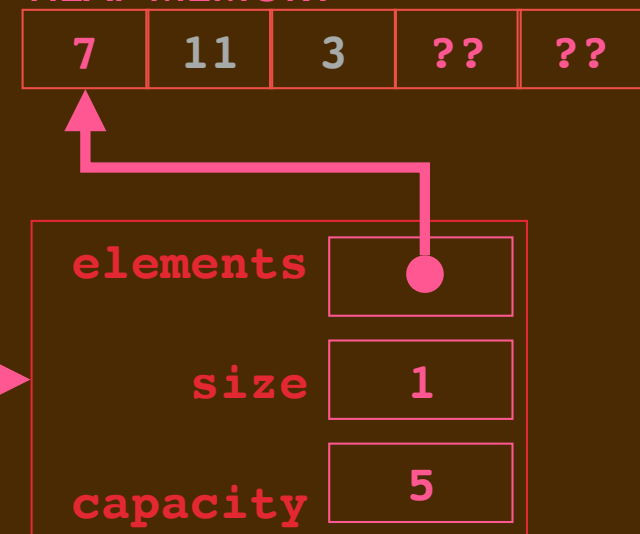
```
1 3
2 1
3 11
4
5
```

STACK FRAME



*The destructor code
gets called with
delete*

HEAP MEMORY



HEAP-ALLOCATED STACK ILLUSTRATED

```
9. #include "Stck.hh"
10. #include <iostream>
11. int main(void) {
12.     Stck* s = new Stck {5};
13.     s->push(7);
14.     s->push(1);
15.     s->push(3);
16.     std::cout << s->pop() << std::endl;
17.     std::cout << s->pop() << std::endl;
18.     s->push(11);
19.     std::cout << s->pop() << std::endl;
20.     delete s;
21. }
```

CONSOLE

```
1 3
2 1
3 11
4
5
```

STACK FRAME

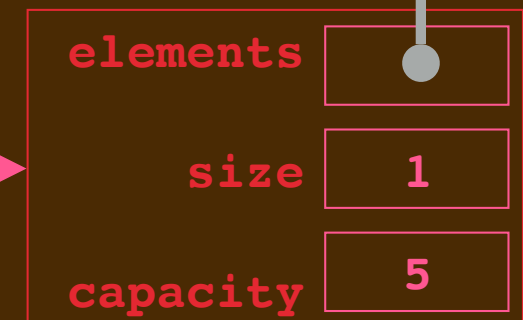


*The destructor code
gets called with*

*...which deletes
s->elements.*

HEAP MEMORY

7	11	3	??	??
---	----	---	----	----



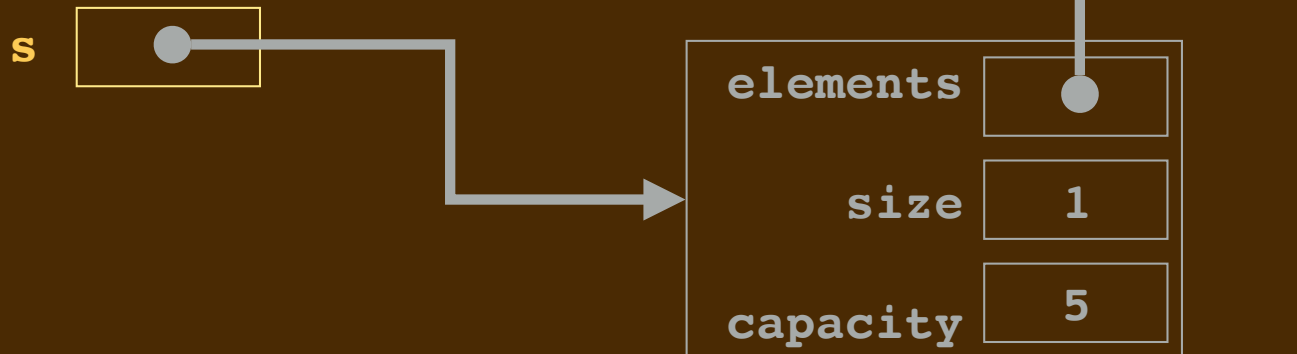
HEAP-ALLOCATED STACK ILLUSTRATED

```
9. #include "Stck.hh"
10. #include <iostream>
11. int main(void) {
12.     Stck* s = new Stck {5};
13.     s->push(7);
14.     s->push(1);
15.     s->push(3);
16.     std::cout << s->pop() << std::endl;
17.     std::cout << s->pop() << std::endl;
18.     s->push(11);
19.     std::cout << s->pop() << std::endl;
20.     delete s;
21. }
```

CONSOLE

```
1 3
2 1
3 11
4
5
```

STACK FRAME



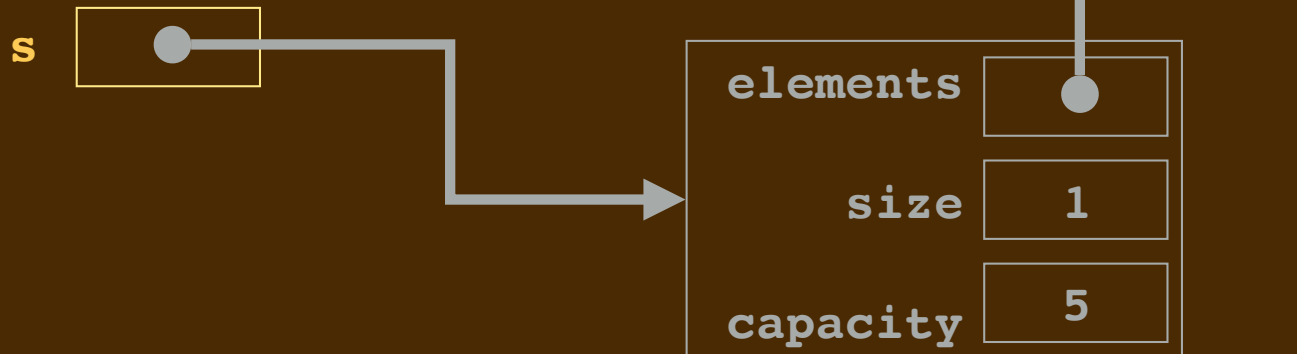
HEAP-ALLOCATED STACK ILLUSTRATED

```
9. #include "Stck.hh"
10. #include <iostream>
11. int main(void) {
12.     Stck* s = new Stck {5};
13.     s->push(7);
14.     s->push(1);
15.     s->push(3);
16.     std::cout << s->pop() << std::endl;
17.     std::cout << s->pop() << std::endl;
18.     s->push(11);
19.     std::cout << s->pop() << std::endl;
20.     delete s;
21. }
```

CONSOLE

```
1 3
2 1
3 11
4
5
```

STACK FRAME



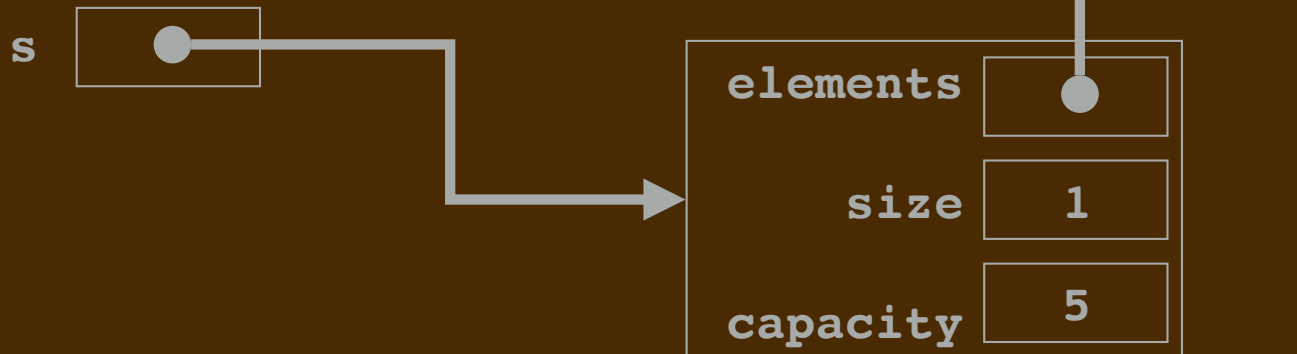
HEAP-ALLOCATED STACK ILLUSTRATED

```
9. #include "Stck.hh"
10. #include <iostream>
11. int main(void) {
12.     Stck* s = new Stck {5};
13.     s->push(7);
14.     s->push(1);
15.     s->push(3);
16.     std::cout << s->pop() << std::endl;
17.     std::cout << s->pop() << std::endl;
18.     s->push(11);
19.     std::cout << s->pop() << std::endl;
20.     delete s;
21. }
```

CONSOLE

```
1 3
2 1
3 11
4
5
```

STACK FRAME



SUMMARY OF CONSTRUCTORS AND DESTRUCTORS

► Constructors

- Code is invoked when an object's struct is allocated
 - within the stack frame, and
 - on the heap using **new**.
- Initialize the instance's variables.

► Destructors

- Code is invoked when an object's struct is de-allocated
 - upon exit from a function when the stack frame is taken down, and
 - upon explicit call of **delete** on a pointer to an instance.
- Typically for giving back heap-allocated components.
 - ✦ (Other use: class-wide accounting.)

MODERN C++ WE COVER

- ▶ BASIC OBJECT-ORIENTATION: CLASSES, METHODS, CON-/DE-STRUCTORS
- ▶ INHERITANCE
- ▶ TEMPLATES
- ▶ SOME NITTY-GRITTY STUFF
 - OPERATOR OVERLOADING
 - REFERENCES **&** ; **const** ; COPY/MOVE CONSTRUCTORS/ASSIGNMENT
- ▶ THE C++ STANDARD TEMPLATE LIBRARY
 - **vector**, **map**, **unordered_map**, ...
- ▶ **lambda**
- ▶ SMART POINTERS, "RAII": **shared_ptr** AND **weak_ptr**

MODERN C++ WE COVER

- ▶ BASIC OBJECT-ORIENTATION: CLASSES, METHODS, CON-/DE-STRUCTORS ✓
- ▶ INHERITANCE
- ▶ TEMPLATES
- ▶ SOME NITTY-GRITTY STUFF
 - OPERATOR OVERLOADING ✓
 - REFERENCES **&** ; **const** ; COPY/MOVE CONSTRUCTORS/ASSIGNMENT
- ▶ THE C++ STANDARD TEMPLATE LIBRARY
 - **vector**, **map**, **unordered_map**, ...
- ▶ **lambda**
- ▶ SMART POINTERS, "RAII": **shared_ptr** AND **weak_ptr**

MODERN C++ WE COVER

- ▶ BASIC OBJECT-ORIENTATION: CLASSES, METHODS, CON-/DE-STRUCTORS ✓
- ▶ INHERITANCE **next week 10**
- ▶ TEMPLATES **week 10 or 11**
- ▶ SOME NITTY-GRITTY STUFF
 - OPERATOR OVERLOADING ✓
 - REFERENCES & ; **const** ; COPY/MOVE CONSTRUCTORS/ASSIGNMENT
- ▶ THE C++ STANDARD TEMPLATE LIBRARY **week 11**
 - **vector**, **map**, **unordered_map**, ... **week 11**
- ▶ **lambda week 12**
- ▶ SMART POINTERS, "RAII": **shared_ptr** AND **weak_ptr week 12**

MODERN C++ WE COVER

- ▶ BASIC OBJECT-ORIENTATION: CLASSES, METHODS, CON-/DE-STRUCTORS ✓
- ▶ INHERITANCE **next week 10**
- ▶ TEMPLATES **week 10 or 11**
- ▶ SOME NITTY-GRITTY STUFF
 - OPERATOR OVERLOADING ✓
 - REFERENCES & ; **const** ; COPY/MOVE CONSTRUCTORS/ASSIGNMENT **10? 11?**
- ▶ THE C++ STANDARD TEMPLATE LIBRARY **week 11**
 - **vector**, **map**, **unordered_map**, ... **week 11**
- ▶ **lambda week 12**
- ▶ SMART POINTERS, "RAII": **shared_ptr** AND **weak_ptr week 12**

LOOK FOR HOMEWORK 09 *COMING SOON*

- ▶ Remember: you'll be assigned a TA mentor
- ▶ Meanwhile: see you on Slack and on Zoom!