OBJECT-ORIENTATION IN C++
OPERATORS AND DESTRUCTORS

LECTURE 11-2

JIM FIX, REED COLLEGE CS2-S20

INVOKING METHODS IN METHODS

- We reference the receiver **`this`** several times.

```
Cmpx Cmpx::conjugate() {
  return Cmpx {this->re,-this->im};
}
double Cmpx::modulus2() {
  return this->times(this->conjugate()).re;
}
Cmpx Cmpx::reciprocal() {
  return this->conjugate().times(1.0 / this->modulus2());
}
Cmpx Cmpx::over(Cmpx that) {
  return this->times(that.reciprocal());
}
```

INVOKING METHODS IN METHODS

- Here instead we touch the receiver's fields and invoke its methods...

```
Cmpx Cmpx::conjugate() {
  return Cmpx {re,-im};
}
double Cmpx::modulus2() {
  return times(conjugate()).re;
}
Cmpx Cmpx::reciprocal() {
  return conjugate().times(1.0 / modulus2());
}
Cmpx Cmpx::over(Cmpx that) {
  return times(that.reciprocal());
}
```

CONSTRUCTOR IMPLEMENTATION

The role of a constructor is to initialize a new object instance's components:

```
Rational::Rational(int n, int d) {
  this->num = n;
  this->den = d;
}
Rational::Rational(int n) {
  this->num = n;
  this->den = 1;
}
Rational::Rational() {
  this->num = 0;
  this->den = 1;
}
Rational::Rational(const Rational& that) {
  this->num = that.num;
  this->den = that.den;
}
```

CONSTRUCTOR IMPLEMENTATION

The role of a constructor is to initialize a new object instance's components:

```
Rational::Rational(int n, int d) {
  num = n;
  den = d;
}
Rational::Rational(int n) {
  num = n;
  den = 1;
}
Rational::Rational() {
  num = 0;
  den = 1;
}
Rational::Rational(const Rational& that) {
  num = that.num;
  den = that.den;
}
```

CONSTRUCTOR IMPLEMENTATION

The initializer notation mimics the initializer list notation of variable declarations:

```
Rational::Rational(int n, int d) :
   num {n}, den {d}
{ }

Rational::Rational(int n) :
   Rational {n, 1}
{ }
Rational::Rational() :
   Rational {0}
{ }

Rational::Rational(const Rational& that) :
   Rational{that.num, that.den}
{ }
```

CONSTRUCTOR IMPLEMENTATION

The initializer notation mimics the initializer list notation of variable declarations:

```
Rational::Rational(int n, int d) {
  num = n / gcd(n,d);
  den = d / gcd(n,d);
}
```

```
Rational::Rational(int n) :
  Rational {n, 1}
{ }
Rational::Rational() :
  Rational {0}
{ }

Rational::Rational(const Rational& that) :
  Rational{that.num, that.den}
{ }
```

CONTROLLING FIELD/METHOD ACCESS

- Might not want clients to use helper methods.
- Might not want clients to directly access instance variables.
- Why?
  - Should isolate the underlying implementation.
  - That way it can be changed by the programmer later.

C++ allows field/method access control with public and private keywords.

CONTROLLING FIELD/METHOD ACCESS

```
1.  class Cmpx {
2.  private:            // Can only be accessed/invoked by methods.
3.    double re;
4.    double im;
5.    Cmpx conjugate(void);
6.    double modulus2(void);
7.    Cmpx reciprocal(void);
8.  public:             // Can be invoked by clients.
9.    Cmpx(void);
10.   Cmpx(std::string);
11.   Cmpx(double rp, double ip);
12.   Cmpx(const Cmpx& that);
13.   Cmpx plus(Cmpx that);
14.   Cmpx times(Cmpx that);
15.   Cmpx over(Cmpx that); // Uses reciprocal.
16.   std::string to_string();
17. };
```

CONTROLLING FIELD/METHOD ACCESS

```
1.  class Cmpx {
2.  private:          // Can only be accessed/invoked by methods.
3.    double re;
4.    double im;
5.    Cmpx conjugate(void);
6.    double modulus2(void);
7.    Cmpx reciprocal(void);
8.  public:           // Can be invoked by clients.
9.    Cmpx(void);
10.   Cmpx(std::string);
11.   Cmpx(double rp, double ip);
12.   Cmpx(const Cmpx& that);
13.   Cmpx plus(Cmpx that);
14.   Cmpx times(Cmpx that);
15.   Cmpx over(Cmpx that); // Uses reciprocal.
16.   std::string to_string();
17. };
```

CONTROLLING FIELD/METHOD ACCESS

```
1.  class Cmpx {
2.  private:            // Can only be accessed/invoked by methods.
3.    double re;
4.    double im;
5.    Cmpx conjugate(void);
6.    double modulus2(void);
7.    Cmpx reciprocal(void);
8.  public:             // Can be invoked by clients.
9.    Cmpx(void);
10.   Cmpx(std::string);
11.   Cmpx(double rp, double ip);
12.   Cmpx(const Cmpx& that);
13.   Cmpx plus(Cmpx that);
14.   Cmpx times(Cmpx that);
15.   Cmpx over(Cmpx that); // Uses reciprocal.
16.   std::string to_string();
17. };
```

CONTROLLING FIELD/METHOD ACCESS W/ GETTERS

```
1.  class Cmpx {
2.  private:          // Can only be accessed/invoked by methods.
3.    double re;
4.    double im;
5.    Cmpx conjugate(void);
6.    double modulus2(void);
7.    Cmpx reciprocal(void);
8.  public:           // Can be invoked by clients.
9.    double getReal();
10.   double getImag();  // "Getters" for access to fields.
11.   Cmpx(void);
12.   Cmpx(std::string);
13.   Cmpx(double rp, double ip);
14.   Cmpx(const Cmpx& that);
15.   Cmpx plus(Cmpx that);
16.   Cmpx times(Cmpx that);
17.   Cmpx over(Cmpx that); // Uses reciprocal.
18.   std::string to_string();
19. };
```

GIVING FIELD/METHOD TO FRIENDS

```
1.  class Cmpx {
2.  private:          // Can only be accessed/invoked by methods.
3.    double re;
4.    double im;
5.    Cmpx conjugate(void);
6.    double modulus2(void);
7.    Cmpx reciprocal(void);
8.  public:           // Can be invoked by clients.
9.    Cmpx(void);
10.   Cmpx(std::string);
11.   Cmpx(double rp, double ip);
12.   Cmpx(const Cmpx& that);
13.   Cmpx plus(Cmpx that);
14.   Cmpx times(Cmpx that);
15.   Cmpx over(Cmpx that);
16.   std::string to_string();
17.   friend Cmpx sum(Cmpx z1, Cmpx z2);
18.   friend Cmpx quotient(Cmpx z1, Cmpx z2);
19. };
```

FRIEND FUNCTIONS

- Friends of a class can access private fields and invoke private methods.

```
1.  class Cmpx {
2.  private:
3.    double re;
4.    double im;
5.    ...
6.    Cmpx reciprocal(void);
7.  public:
8.    ...
9.    friend Cmpx sum(Cmpx z1, Cmpx z2);
10.   friend Cmpx quotient(Cmpx z1, Cmpx z2);
11. };
```

- Function definitions (usually defined within "Cmpx.cc"):

```
Cmpx sum(Cmpx z1, Cmpx z2) {
  return Cmpx {z1.re + z2.re, z1.im + z2.im};
}
Cmpx quotient(Cmpx z1, Cmpx z2) {
  return z1.times(z2.reciprocal());
}
```

CLASS MEMBERS

We can associate values and functions with the class, rather than instances:

```
1.  class Cmpx {
2.  private:
3.    double re;
4.    double im;
5.    static const double kEpsilon;
6.    static void parse(std::string s, double &rp, double &ip);
7.  public:
8.    Cmpx(void);
9.    Cmpx(std::string);
10.   Cmpx(double rp, double ip);
11.   Cmpx(const Cmpx& that);
12.   Cmpx plus(Cmpx that);
13.   std::string to_string();
14.   static const Cmpx I;
15.   static Cmpx product(Cmpx z1, Cmpx z2);
16. };
```

STATIC MEMBERS

We associate values and functions with the class, rather than instances:

```
1.  class Cmpx {
2.  private:
3.     double re;
4.     double im;
5.     static const double kEpsilon;
6.     static void parse(std::string s, double &rp, double &ip);
7.  public:
8.     Cmpx(void);
9.     Cmpx(std::string);
10.    Cmpx(double rp, double ip);
11.    Cmpx(const Cmpx& that);
12.    Cmpx plus(Cmpx that);
13.    std::string to_string();
14.    static const Cmpx I;
15.    static Cmpx product(Cmpx z1, Cmpx z2);
16. };
```

STATIC MEMBERS

We associate values and functions with the class, rather than instances:

```
1.  class Cmpx {
2.  ...
3.    static const double kEpsilon;
4.    static void parse(std::string s, double &rp, double &ip);
5.  ...
6.    static const Cmpx I;
7.    static Cmpx product(Cmpx z1, Cmpx z2);
8.  };
```

- The keyword static distinguishes them from *instance* variables/methods.
- They are called "class variables" (const in this case) and"class methods."
- In the case of variables, there is only one defined, shared by all instances.
- NOTE: "static" is carried from C meaning "can lay out at compile time."

STATIC MEMBERS

We associate values and functions with the class, rather than instances:

```
1.  class Cmpx {
2.  ...
3.    static const double kEpsilon;
4.    static void parse(std::string s, double &rp, double &ip);
5.  ...
6.    static const Cmpx I;
7.    static Cmpx product(Cmpx z1, Cmpx z2);
8.  };
```

- The keyword static distinguishes these variables/methods.
- They are called "static variables" and "static methods."
- In the case of variables, they are shared by all instances.
- NOTE: "static" is carried from C meaning "can lay out at compile time."

i.e. not dynamic, or "at run time"

IMPLEMENTING CLASS MEMBERS

```
1.  const double Cmpx::kEpsilon = 0.000001;
2.
3.  const Cmpx Cmpx::I {0.0,1.0};
4.
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.

IMPLEMENTING CLASS MEMBERS

```
1.  const double Cmpx::kEpsilon = 0.000001;
2.
3.  const Cmpx Cmpx::I {0.0,1.0};
4.
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.

USING CLASS MEMBERS

- Within the class implementation code:

```
Cmpx::Cmpx(std::string) {
  parse(s,this->re,this->im);
}
std::string Cmpx::to_string() {
  if (std::abs(im) < kEpsilon) {
    return std::to_string(re);
  } else if (std::abs(re) < kEpsilon) {
    return std::to_string(im) + "i";
  } else {
   if (im < 0.0) {
      return std::to_string(re)+std::to_string(im) + "i";
    } else {
      return std::to_string(re)+"+"+std::to_string(im) + "i";
    }
  }
}
```

- Within the client's code (if part of the class's public interface):

```
Cmpx rotate(Cmpx z) {
  return Cmpx::product(z, Cmpx::I)
}
```

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.     Rational(void);
9.     Rational(std::string s);
10.    Rational(int n, int d);
11.    Rational(const Rational& q);
12.
13.    Rational plus(Rational that);
14.    Rational times(Rational that);
15.    std::string to_string(void);
16. };
```

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 what I'll share
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 INSTANCE 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... 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. }
```

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);
```

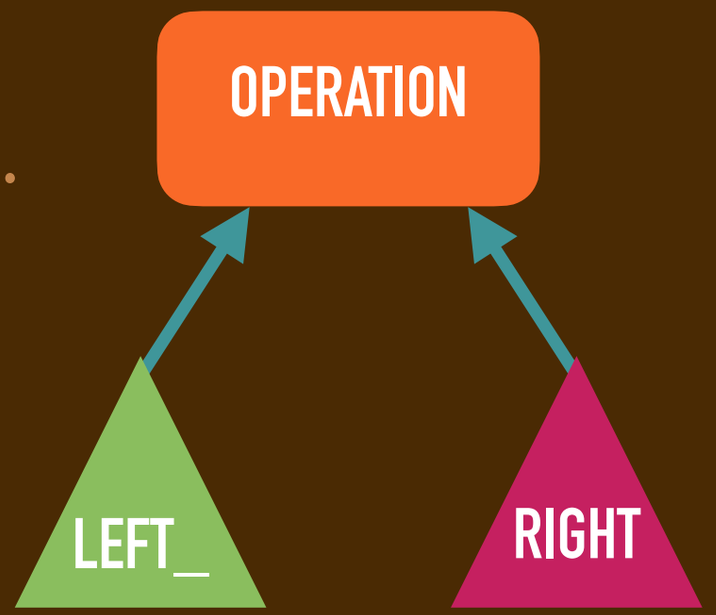
ADDITIONS TO RATIONAL.HH FOR OVERLOADED OPERATORS

```
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;
```



- 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)
```

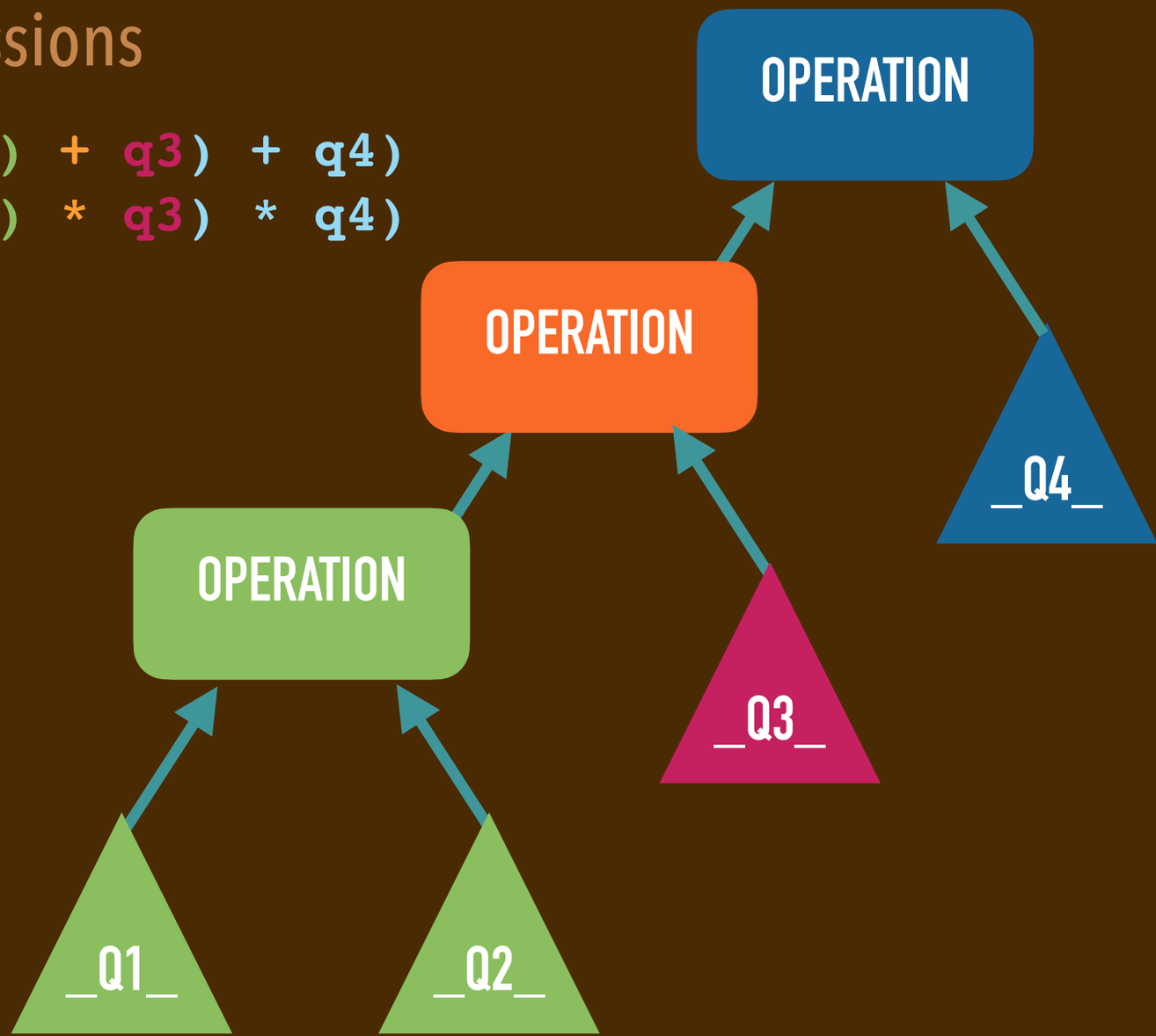
LEFT ASSOCIATIVITY OF PLUS AND TIMES

- 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 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.
  - 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::os
  os <<
  return os;
}
std::istream& operator>>(std::istream& is, Rational& q) {
  std::string s;
  is >> s;
  q = Rational {s};
  return is;
}
```

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

- 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.
  - 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...

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";
```

IMPLEMENTING BINARY OPERATORS AS METHODS

- This declaration for + and * also works, making them methods instead of functions

```
class Rational {
private:
  int num;
  int den;
public:
  ...
  Rational operator+(Rational q);
  Rational operator*(Rational q);
  ...
};
```

- Here is their code:

```
Rational Rational::operator+(Rational q) {
  return plus(q2);
}
Rational Rational::operator*(Rational q) {
  return times(q);
}
```

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. Some can be either operators or 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***.
- Even if they live on the stack, they might have components in the heap.

- Upon construction, we set up the object's data and these relations.
  - ➡ In particular, may need allocate their 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.   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.   elements[num_elements] = value;
15.   num_elements++;
16. }
17.
18. int Stck::pop() {
19.   num_elements--;
20.   return elements[num_elements];
21. }
22.
23. int Stck::top() {
24.   return elements[num_elements-1];
25. }
```

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
```

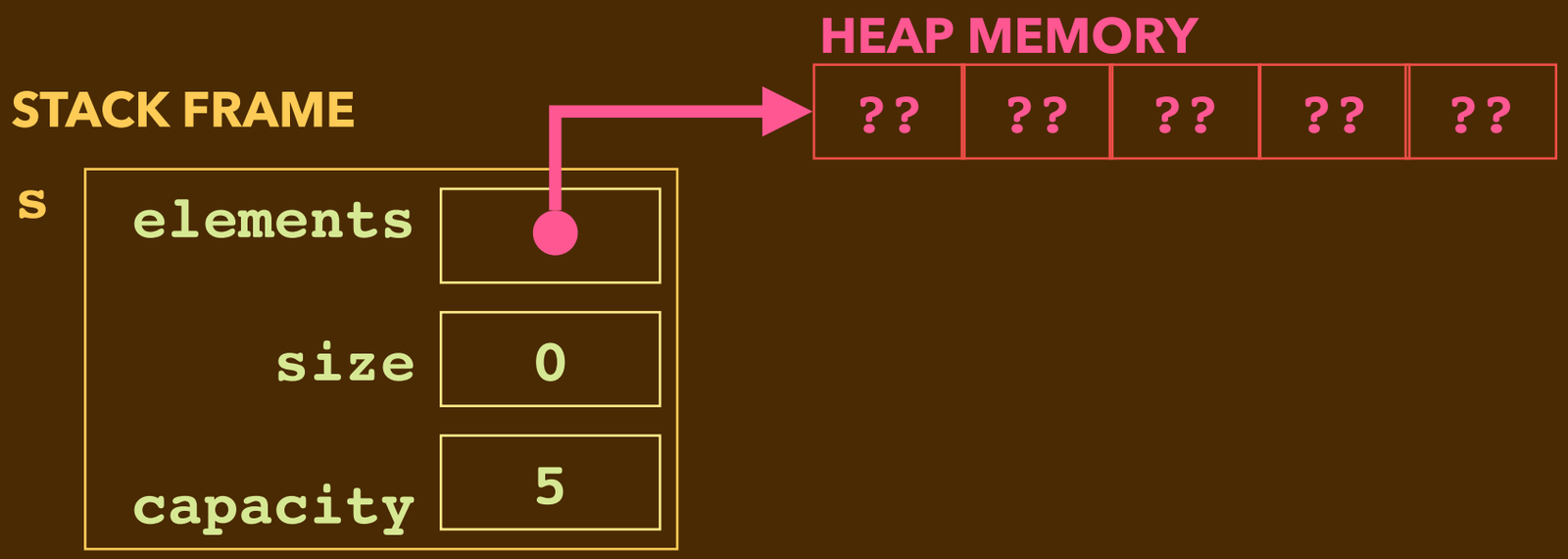


ILLUSTRATION WITH A SIMPLE CLIENT

```
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
```

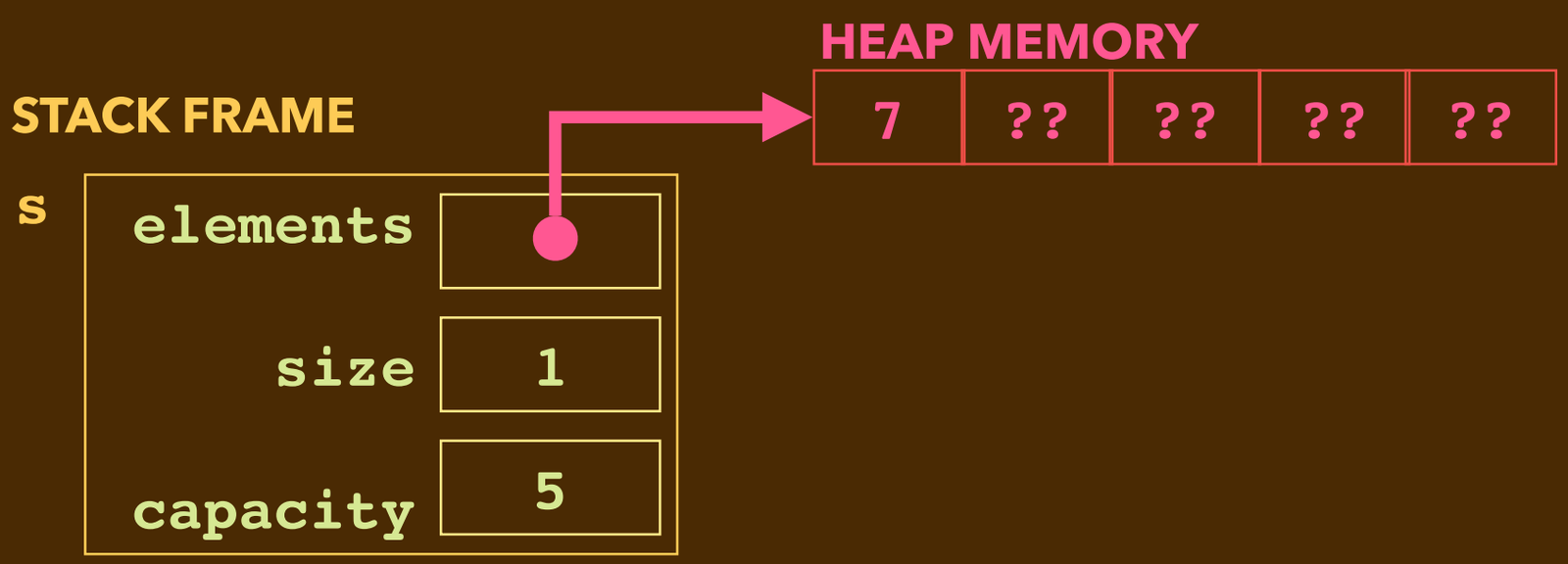


ILLUSTRATION WITH A SIMPLE CLIENT

```
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
```

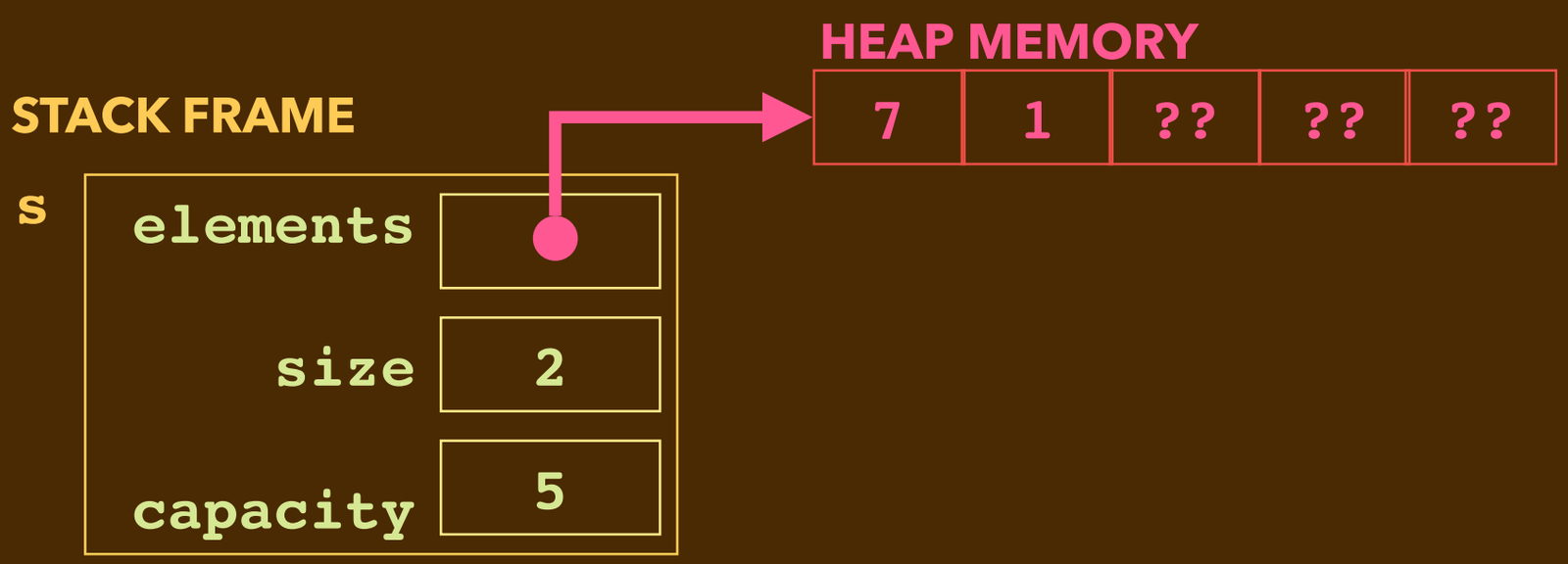


ILLUSTRATION WITH A SIMPLE CLIENT

```
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
```

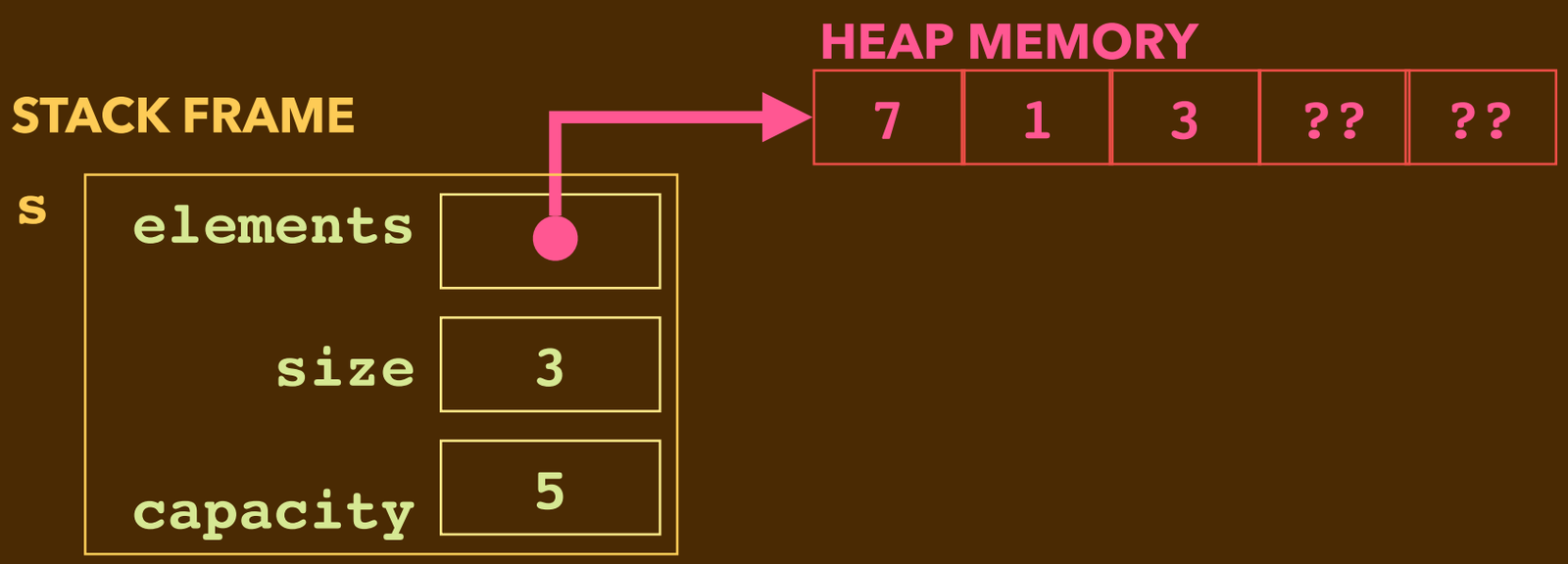


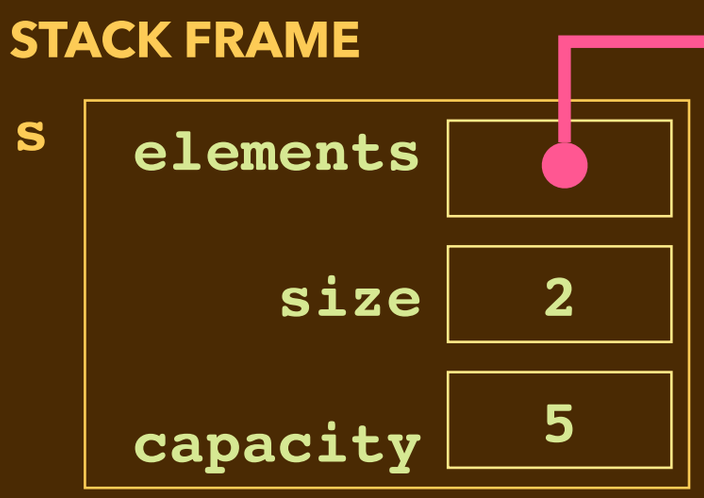
ILLUSTRATION WITH A SIMPLE CLIENT

```
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

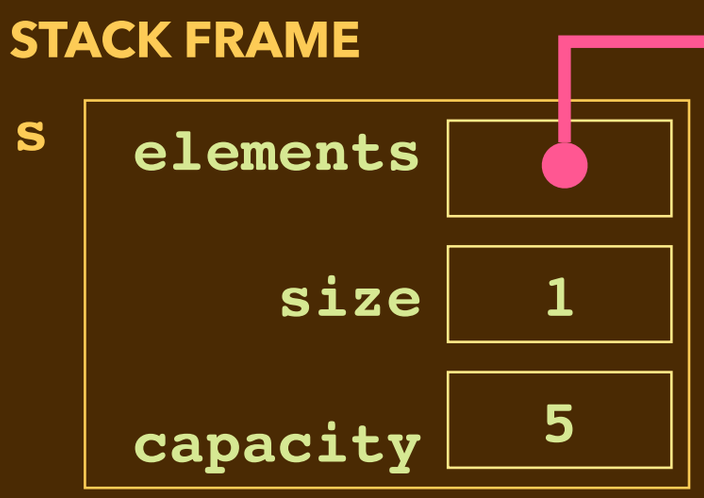
ILLUSTRATION WITH A SIMPLE CLIENT

```
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

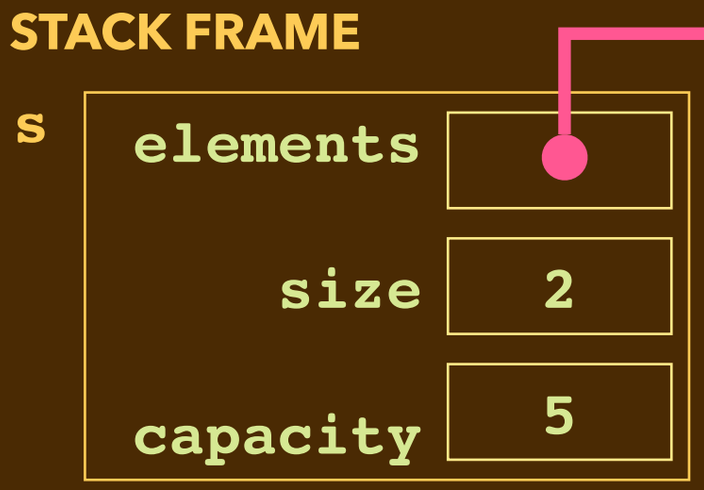
ILLUSTRATION WITH A SIMPLE CLIENT

```
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

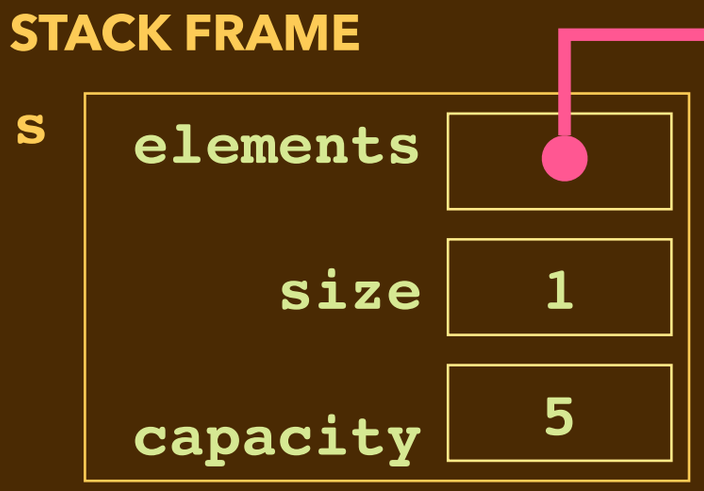
ILLUSTRATION WITH A SIMPLE CLIENT

```
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

ILLUSTRATION WITH A SIMPLE CLIENT

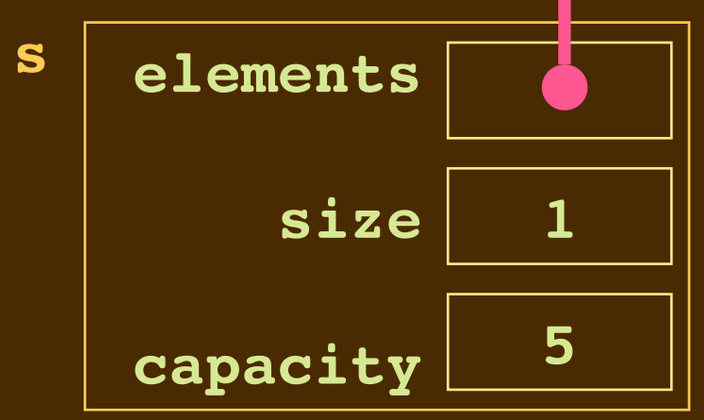
```
9.  #include "Stck.hh"
10. #include <iostream>
11. int main(void) {
12.   Stck s {5};
13.   s.push(7);
14.   s.push(1);
15.   s.push(3);
16.   std::cout << s.pop() << std::endl;
17.   std::cout << s.pop() << std::endl;
18.   s.push(11);
19.   std::cout << s.pop() << std::endl;
20. }
```

Calls the default destructor.

CONSOLE

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

STACK FRAME



HEAP MEMORY

ILLUSTRATION WITH A SIMPLE CLIENT

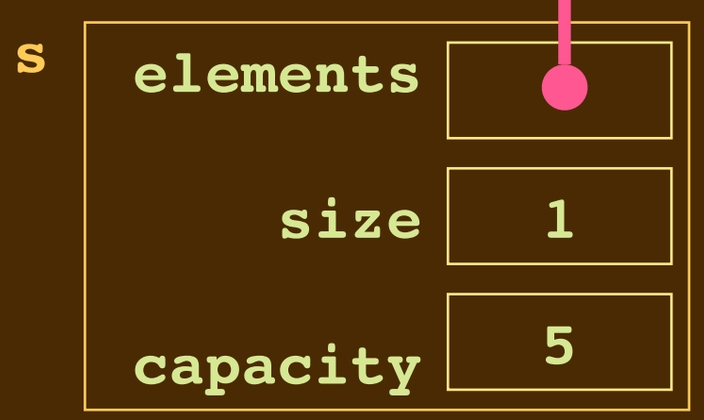
```
9.  #include "Stck.hh"
10. #include <iostream>
11. int main(void) {
12.   Stck s {5};
13.   s.push(7);
14.   s.push(1);
15.   s.push(3);
16.   std::cout << s.pop() << std::endl;
17.   std::cout << s.pop() << std::endl;
18.   s.push(11);
19.   std::cout << s.pop() << std::endl;
20. }
```

Calls the default destructor.

CONSOLE

```
1 3
2 1
3 11
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. }
```

Calls the default

And the frame gets taken down.

CONSOLE

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

STACK FRAME

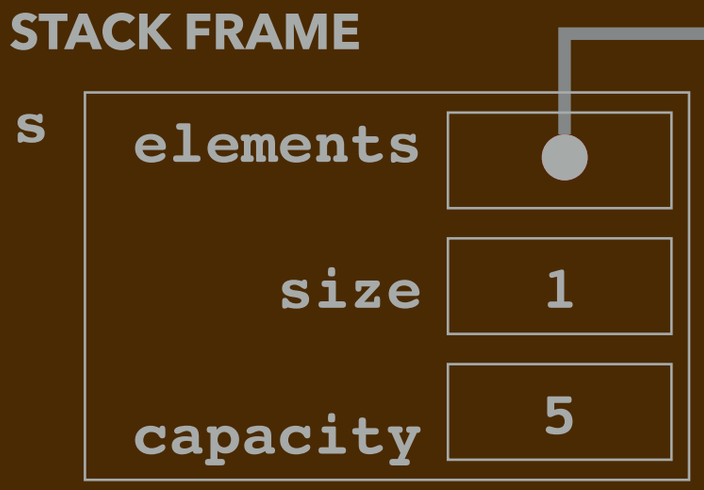


ILLUSTRATION WITH A SIMPLE CLIENT

```
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. }
```

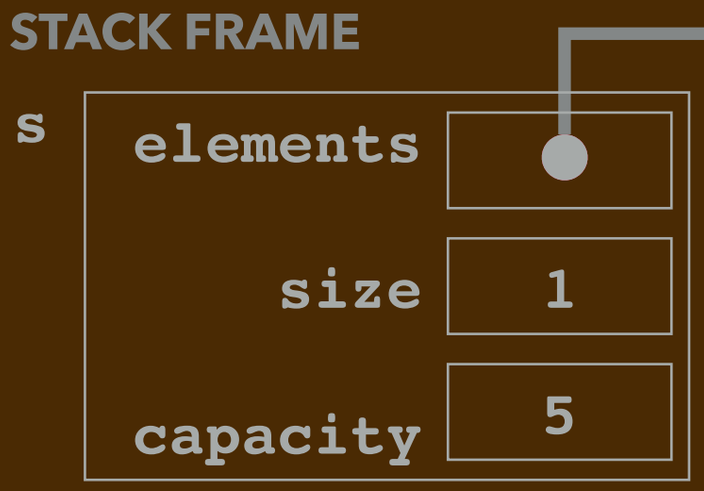
Calls the default

And the frame gets taken down.

CONSOLE

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

STACK FRAME



HEAP MEMORY

But we have a memory leak.

DESTRUCTOR CODE

- Destructor code is executed when a stack-allocated object goes out of scope.
- Here is code we need 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.

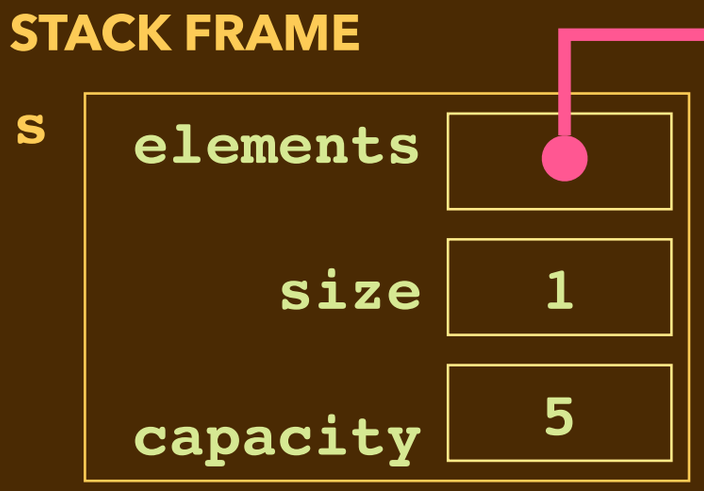
ILLUSTRATION WITH A SIMPLE CLIENT

```
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

IMPLICIT CALL OF THE DESTRUCTOR

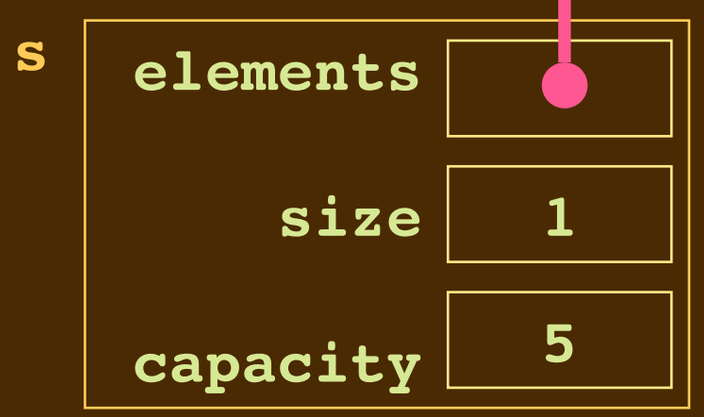
```
9.  #include "Stck.hh"
10. #include <iostream>
11. int main(void) {
12.   Stck s {5};
13.   s.push(7);
14.   s.push(1);
15.   s.push(3);
16.   std::cout << s.pop() << std::endl;
17.   std::cout << s.pop() << std::endl;
18.   s.push(11);
19.   std::cout << s.pop() << std::endl;
20. }
```

Calls the destructor, which `delete`*s.*

CONSOLE

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

STACK FRAME



HEAP MEMORY

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. }
```

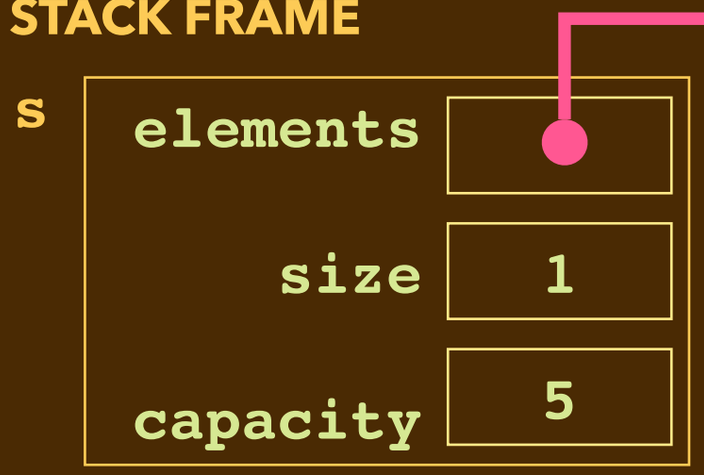
Calls the

And the frame gets taken down.

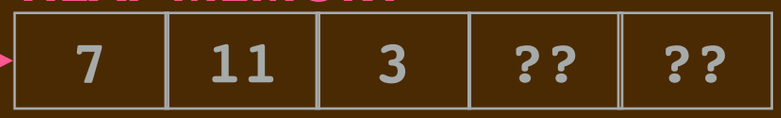
CONSOLE

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

STACK FRAME



HEAP MEMORY



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. }
```

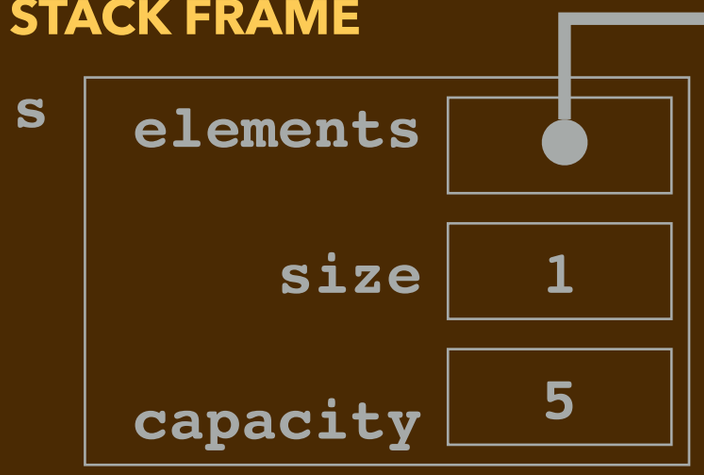
Calls the

And the frame gets taken down.

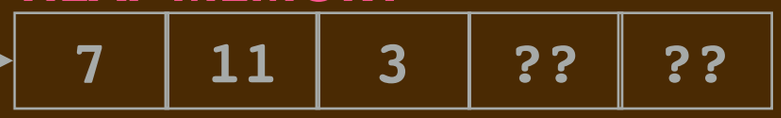
CONSOLE

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

STACK FRAME



HEAP MEMORY



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. }
```

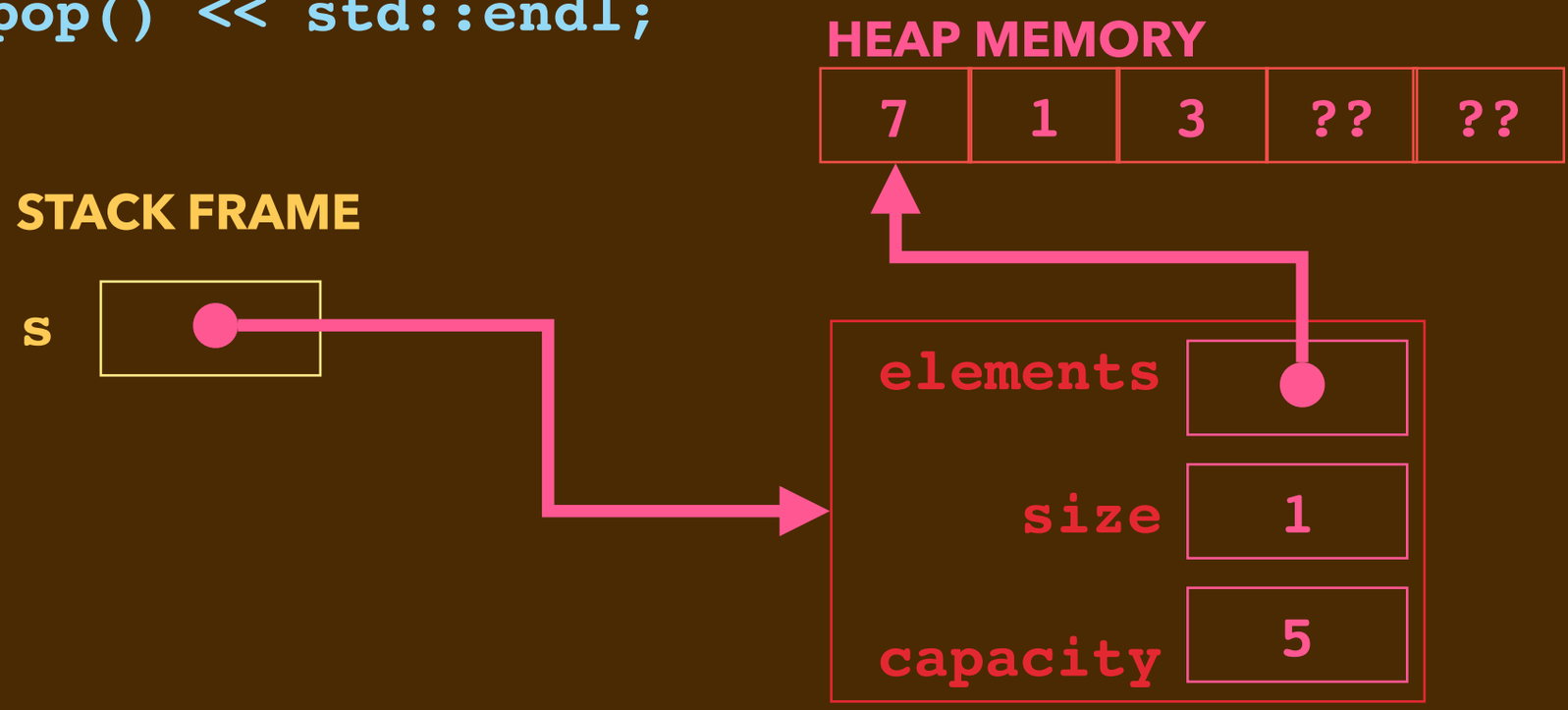
- Now **s** can point to a Stck instance. Its type is **Stck***
- We can construct a new instance that lives on the heap.
- And we must explicitly delete that pointer.

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
```



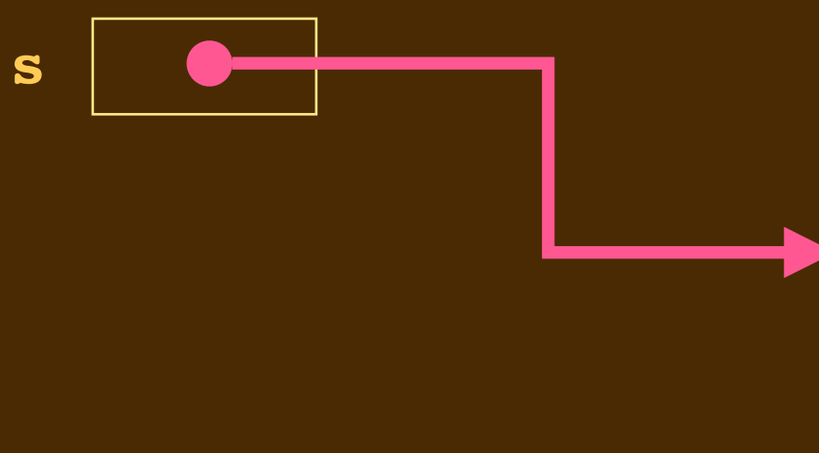
HEAP-ALLOCATED STACK ILLUSTRATED

```
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. }
```

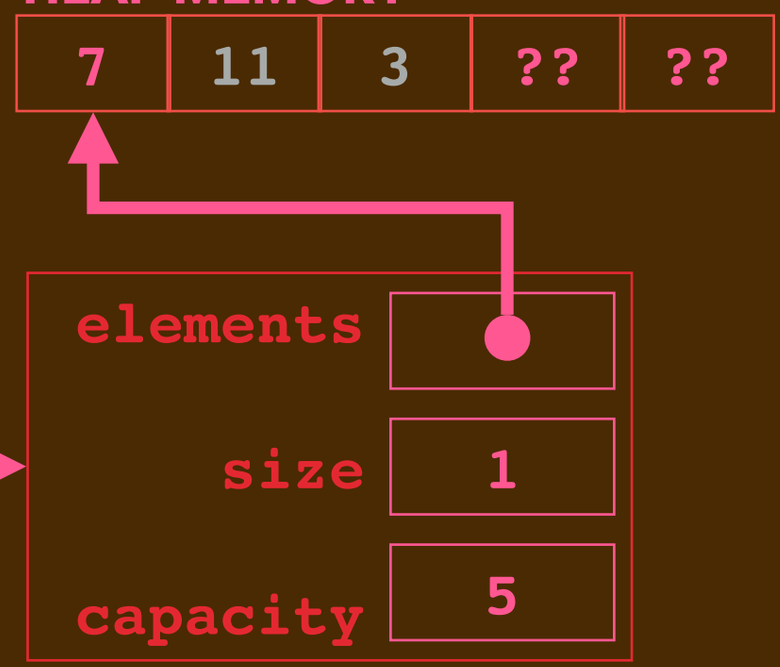
CONSOLE

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

STACK FRAME



HEAP MEMORY



HEAP-ALLOCATED STACK ILLUSTRATED

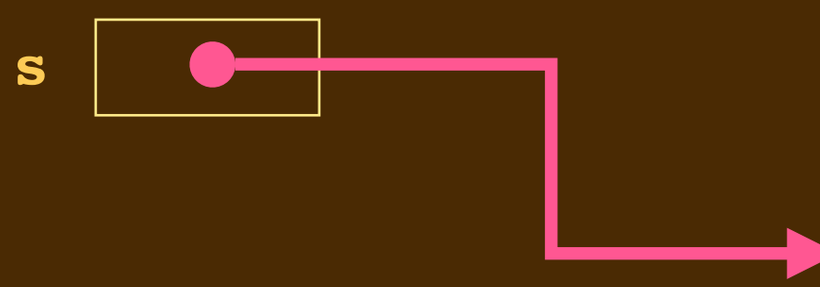
```
9.  #include "Stck.hh"
10. #include <iostream>
11. int main(void) {
12.   Stck* s = new Stck {5};
13.   s->push(7);
14.   s->push(1);
15.   s->push(3);
16.   std::cout << s->pop() << std::endl;
17.   std::cout << s->pop() << std::endl;
18.   s->push(11);
19.   std::cout << s->pop() << std::endl;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

The destructor code gets called with delete

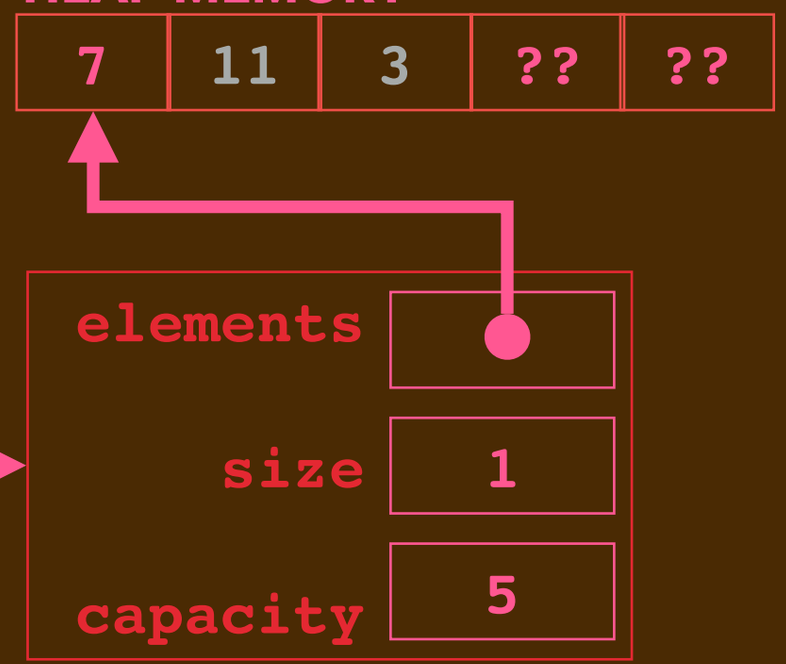
CONSOLE

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

STACK FRAME



HEAP MEMORY



HEAP-ALLOCATED STACK ILLUSTRATED

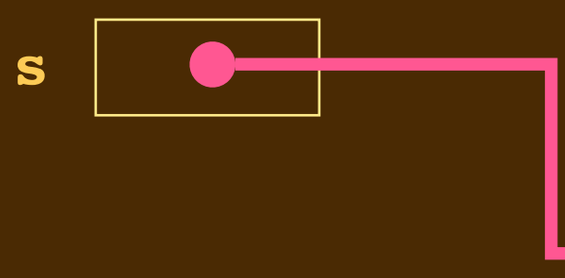
```
9.  #include "Stck.hh"
10. #include <iostream>
11. int main(void) {
12.   Stck* s = new Stck {5};
13.   s->push(7);
14.   s->push(1);
15.   s->push(3);
16.   std::cout << s->pop() << std::endl;
17.   std::cout << s->pop() << std::endl;
18.   s->push(11);
19.   std::cout << s->pop() << std::endl;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

The destructor code gets called with ...which deletes `s->elements`.

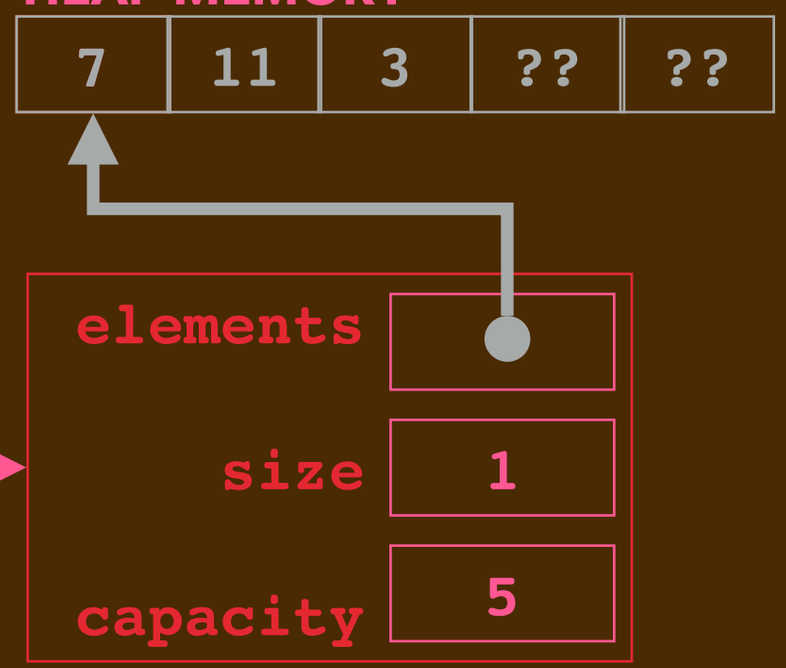
CONSOLE

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

STACK FRAME



HEAP MEMORY



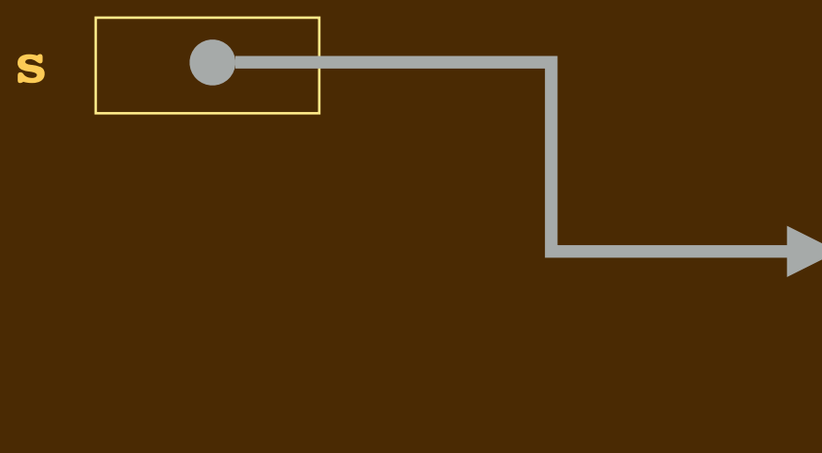
HEAP-ALLOCATED STACK ILLUSTRATED

```
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. }
```

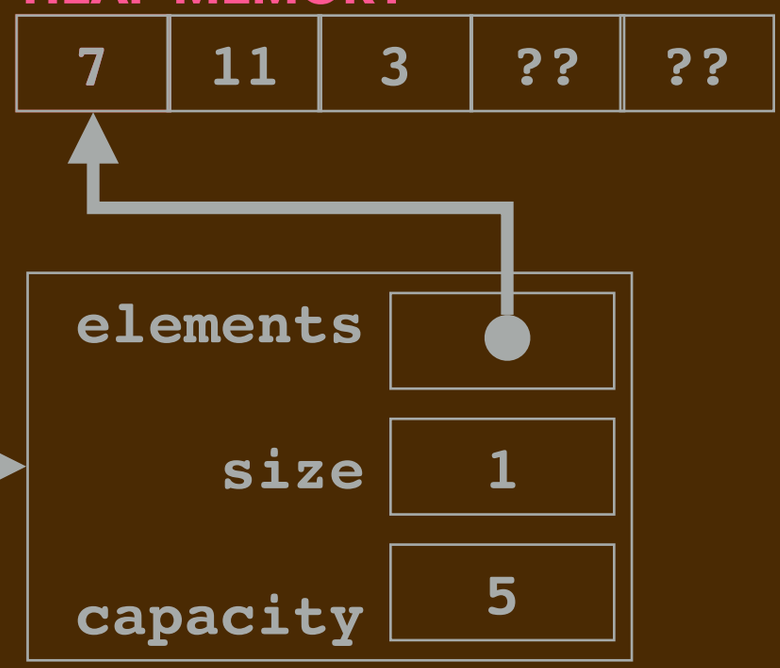
CONSOLE

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

STACK FRAME



HEAP MEMORY



HEAP-ALLOCATED STACK ILLUSTRATED

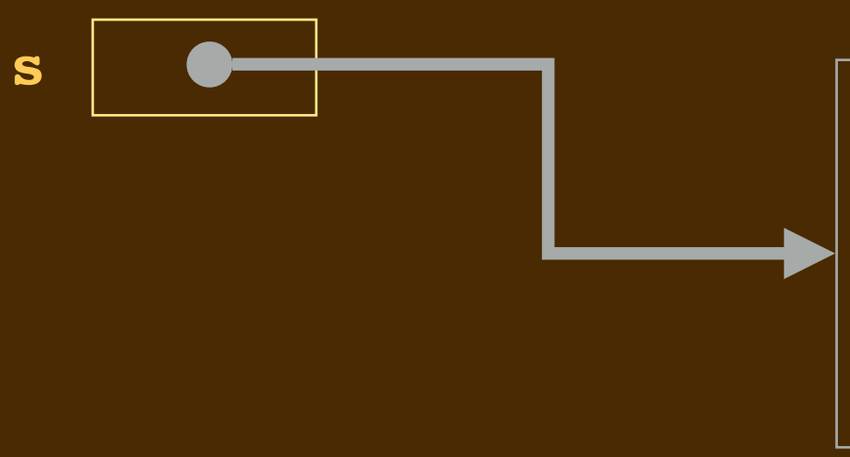
```
9.  #include "Stck.hh"
10. #include <iostream>
11. int main(void) {
12.   Stck* s = new Stck {5};
13.   s->push(7);
14.   s->push(1);
15.   s->push(3);
16.   std::cout << s->pop() << std::endl;
17.   std::cout << s->pop() << std::endl;
18.   s->push(11);
19.   std::cout << s->pop() << std::endl;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

Then the frame gets taken down.

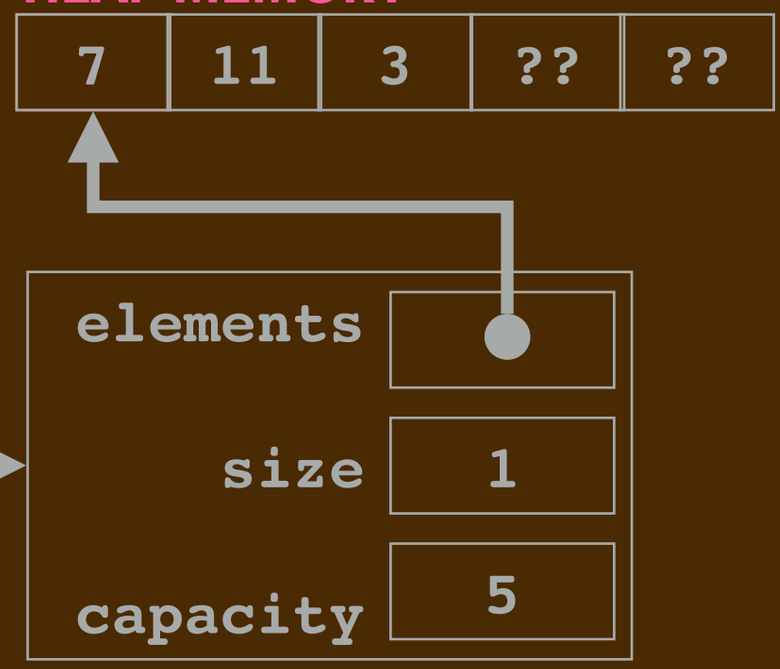
CONSOLE

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

STACK FRAME



HEAP MEMORY



HEAP-ALLOCATED STACK ILLUSTRATED

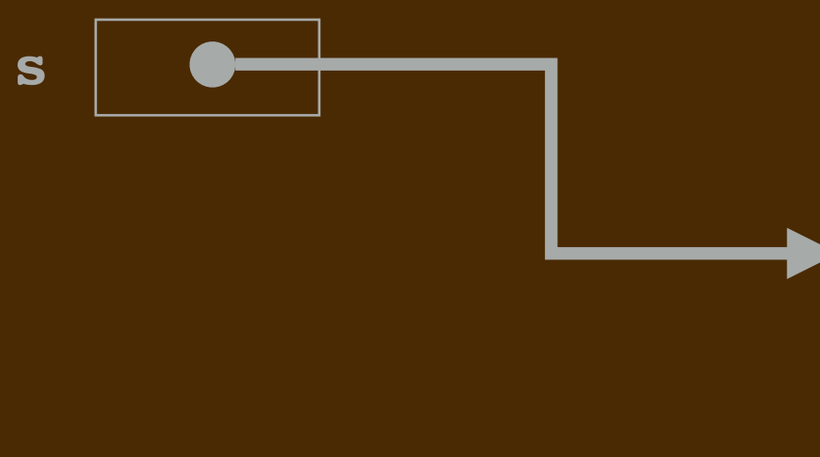
```
9.  #include "Stck.hh"
10. #include <iostream>
11. int main(void) {
12.   Stck* s = new Stck {5};
13.   s->push(7);
14.   s->push(1);
15.   s->push(3);
16.   std::cout << s->pop() << std::endl;
17.   std::cout << s->pop() << std::endl;
18.   s->push(11);
19.   std::cout << s->pop() << std::endl;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

Then the frame gets taken down.

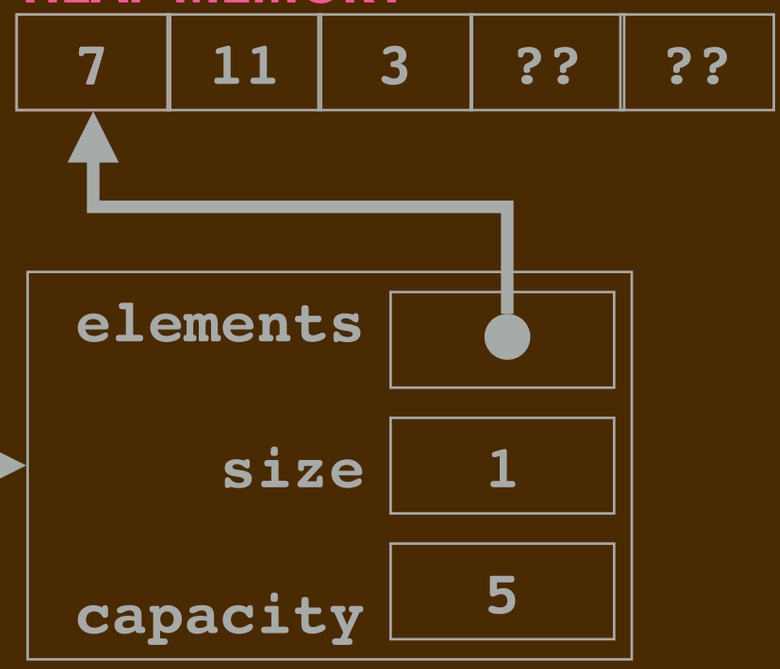
CONSOLE

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

STACK FRAME



HEAP MEMORY



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 Monday***
- TEMPLATES ***next Monday & Wednesday***
- SOME NITTY-GRITTY STUFF
  - OPERATOR OVERLOADING ✓
  - REFERENCES `&` ; `const` ; COPY/MOVE CONSTRUCTORS/ASSIGNMENT
- THE C++ STANDARD TEMPLATE LIBRARY ***next Wednesday***
  - `vector`, `map`, `unordered_map`, ... ***next Wednesday***
- `lambda` ***after Txgvg***
- SMART POINTERS, "RAII": `shared_ptr` AND `weak_ptr` ***last day***

RECALL: IN C++ ARGUMENTS ARE PASSED BY VALUE

- Consider these function definitions

```
void increment(int i) {
  i = i+1;
}
void swap(int x, int y) {
  int tmp = x;
  x = y;
  y = tmp;
}
```

- They don't do much. The code below does this:

```
int count = 10;
int a = 17;
int b = 42;
std::cout << count << " " << a << " " << b << "\n";
increment(count);
swap(a,b);
std::cout << count << " " << a << " " << b << "\n";
```

RECALL: IN C++ ARGUMENTS ARE PASSED BY VALUE

- Consider these function definitions

```
void increment(int i) {
  i = i+1;
}
void swap(int x, int y) {
  int tmp = x;
  x = y;
  y = tmp;
}
```

- They don't do much. The code below does this:

```
int count = 10;
int a = 17;
int b = 42;
std::cout << count << " " << a << " " << b << "\n";
increment(count);
swap(a,b);
std::cout << count << " " << a << " " << b << "\n";
```

CONSOLE

```
1 10 17 42
2 10 17 42
```

PASSING POINTERS

- If we use pointers instead

```
void increment(int* ip) {
  (*ip) = (*ip)+1;
}
void swap(int* xp, int* yp) {
  int tmp = (*xp);
  (*xp) = (*yp);
  (*yp) = tmp;
}
```

- ...then we achieve what we want:

```
int count = 10;
int a = 17;
int b = 42;
std::cout << count << " " << a << " " << b << "\n";
increment(&count);
swap(&a,&b);
std::cout << count << " " << a << " " << b << "\n";
```

PASSING POINTERS

- If we use pointers instead

```
void increment(int* ip) {
  (*ip) = (*ip)+1;
}
void swap(int* xp, int* yp) {
  int tmp = (*xp);
  (*xp) = (*yp);
  (*yp) = tmp;
}
```

- ...then we achieve what we want:

```
int count = 10;
int a = 17;
int b = 42;
std::cout << count << " " << a << " " << b << "\n";
increment(&count);
swap(&a,&b);
std::cout << count << " " << a << " " << b << "\n";
```

CONSOLE

```
1 10 17 42
2 11 42 17
```

PASSING POINTERS

- If we use pointers instead

```
void increment(int* ip) {
  (*ip) = (*ip)+1;
}
void swap(int* xp, int* yp) {
  int tmp = (*xp);
  (*xp) = (*yp);
  (*yp) = tmp;
}
```

- ...then we achieve what we want:

```
int count = 10;
int a = 17;
int b = 42;
std::cout << count << " " << a << " " << b << "\n";
increment(&count);
swap(&a,&b);
std::cout << count << " " << a << " " << b << "\n";
```

We pass pointers that refer to the storage of the variables.

CONSOLE

```
1 10 17 42
2 11 42 17
```

PASSING POINTERS

- If we use pointers instead

```
void increment(int* ip) {
  (*ip) = (*ip)+1;
}
void swap(int* xp, int* yp) {
  int tmp = (*xp);
  (*xp) = (*yp);
  (*yp) = tmp;
}
```

*This makes *ip, *xp, *yp "aliases" of count, a, b.*

We pass pointers that refer to the storage of the variables.

- ...then we achieve what we want:

```
int count = 10;
int a = 17;
int b = 42;
std::cout << count << " " << a << " " << b << "\n";
increment(&count);
swap(&a,&b);
std::cout << count << " " << a << " " << b << "\n";
```

CONSOLE

```
1 10 17 42
2 11 42 17
```

PASSING AND RETURNING STRUCTS

- When a structure is passed as an argument with a function call, each of its components is copied into the local storage of the callee.

```
struct point100d {
  double x1;
  double x2;
  ...
  double x100;
};

void print(point100d p) {
  std::cout << "(" << p.x1 << ",";
  std::cout << p.x2 << ",";
  ...
}
...
  point100d big_point = ...;
  print(big_point);
...
```

Copies 100 doubles, 640 bytes.

PASSING AND RETURNING STRUCTS

- When a structure is passed as an argument with a function call, each of its components is copied into the local storage of the callee.

```
struct point100d {
  double x1;
  double x2;
  ...
  double x100;
};

void print(point100d* p) {
  std::cout << "(" << p->x1 << ",";
  std::cout << p->x2 << ",";
  ...
}
...
  point100d big_point = ...;
  print(&big_point);
...
```

In C, people passed pointers to prevent all this copying... a pointer is only 8 bytes.

~~*Copies 100 doubles, 640 bytes.*~~

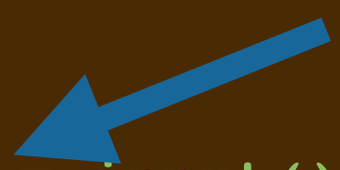
PASSING AND RETURNING STRUCTS

- Copying of components happens when a function returns a struct.

```
struct point100d {
  double x1;
  double x2;
  ...
  double x100;
};

point100d input(void) {
  point100d p;
  std::cin >> p.x1;
  std::cin >> p.x2;
  ...
  return p;
}
...
  point100d big_point = input();
...
```

Copies 100 doubles, 640 bytes.



PASSING ~~AND RETURNING~~ STRUCTS

- Copying of components happens when a function returns a struct.

```
struct point100d {
  double x1;
  double x2;
  ...
  double x100;
};

void get(point100d *p) {
  std::cin >> p->x1;
  std::cin >> p->x2;
  ...
  std::cin >> p->x100;
}
...
  point100d big_point;
  get(&big_point);
...
```

No easy way around this unless you heap allocate the struct's storage...

...or write the code differently.

PASSING "BY REFERENCE"

- C++ allows you to pass parameters ***by reference***.

```
void increment(int& i) {
  i = i+1;
}
void swap(int& x, int& y) {
  int tmp = x;
  x = y;
  y = tmp;
}
```

The use of & makes the parameters i, x, and y aliases of count, a, b.

CONSOLE

```
1 10 17 42
2 11 42 17
```

- The client code looks none the wiser:

```
int count = 10;
int a = 17;
int b = 42;
std::cout << count << " " << a << " " << b << "\n";
increment(count);
swap(a,b);
std::cout << count << " " << a << " " << b << "\n";
```

- But, under the covers, C++ does all the logistical work of passing pointers instead of copying values.

PASSING STRUCTS "BY REFERENCE"

- We can do the same to avoid copying when we pass structs:

```
void print(point100d& p) {
  std::cout << "(" << p.x1 << ",";
  std::cout << p.x2 << ",";
  ...
  std::cout << p.x100 << ")" << std::endl;
}
```

- And we can modify structs' components this way, of course, too:

```
void get(point100d& p) {
  std::cin >> p.x1;
  std::cin >> p.x2;
  ...
  std::cin >> p.x100;
}
```

PASSING OBJECTS BY REFERENCE

- We can do the same to avoid copying when we pass objects as parameters:

```
class Point100d {
  double x1;
  double x2;
  ...
  double x100;
  void operator+=(Point100d& that) {
    this->x1 += that.x1;
    this->x2 += that.x2;
    ...
    this->x100 += that.x100;
  }
};
```

PASSING OBJECTS BY REFERENCE

- We can do the same to avoid copying when we pass objects as parameters:

```
class Point100d {
  double x1;
  double x2;
  ...
  double x100;
  void operator+=(Point100d& that) {
    this->x1 += that.x1;
    this->x2 += that.x2;
    ...
    this->x100 += that.x100;
  }
};
```

- But, this kind of reference passing *might be concerning* to the client.
- It *might not want the method to change* the contents of what it passes.

CONST PARAMETERS

- The keyword const advertises and enforces this restriction:

```
class Point100d {
  double x1;
  double x2;
  ...
  double x100;
  void operator+=(const Point100d& that) {
    this->x1 += that.x1;
    this->x2 += that.x2;
    ...
    this->x100 += that.x100;
  }
};
```

- The **`const`** keyword indicates that the contents of **`that`** aren't modified.
- The compiler enforces this. Raises an error if the method's body violates it.

CONST METHODS

- Consider the print method below:

```
class Point100d {
  double x1;
  double x2;
  ...
  double x100;
  void print(void) const {
    std::cout << "(" << this->x1 << ",";
    std::cout << this->x2 << ",";
    ...
    std::cout << this->x100 << ")";
  }
};
```

- The **`const`** keyword indicates that the contents of **`this`** aren't modified.
- The compiler enforces this, too, makes sure the method body behaves.

EXAMPLE CLASS INTERFACES WITH CONST AND REFERENCE

```
class Rational {
private:
  int num;
  int den;

public:
  // constructors
  Rational(void);
  Rational(std::string s);
  Rational(int n);
  Rational(int n, int d);

  // methods
  Rational plus(const Rational& that) const;
  Rational times(const Rational& that) const;
  std::string to_string(void) const;
};

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

EXAMPLE CLASS INTERFACES WITH CONST AND REFERENCE

```
class Stck {

private:
  int *elements;
  int num_elements;
  int capacity;

public:
  Stck(int capacity);
  bool is_empty() const;
  void push(int value);
  int pop();
  int top() const;
  std::string to_string() const;
  ~Stck();
  friend ostream& operator<<(ostream& os, const Stck& s);
  friend istream& operator<<(istream& is, Stck& s);
};
```

HMMM...

```
class Stck {

private:
  int *elements;
  int num_elements;
  int capacity;

public:
  Stck(int capacity);
  bool is_empty() const;
  void push(int value);
  int pop();
  int top() const;
  std::string to_string() const;
  ~Stck();
  friend ostream& operator<<(ostream& os, const Stck& s);
  friend istream& operator<<(istream& is, Stck& s);
};
```