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.

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

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

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

static bool Cmpx::parse(string s, double& rp, double& ip) {
   ...
}
static Cmpx Cmpx::product(Cmpx z1, Cmpx z2) {
   ...
}
```

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

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

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

static bool Cmpx::parse(string s, double& rp, double& ip) {
    ...
}
static Cmpx Cmpx::product(Cmpx z1, Cmpx z2) {
    ...
}
```

~~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

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

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

bool Cmpx::parse(string s, double& rp, double& ip) {
  ...
}
Cmpx Cmpx::product(Cmpx z1, Cmpx z2) {
  ...
}
```

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

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

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

bool Cmpx::parse(string s, double& rp, double& ip) {
    ...
}
Cmpx Cmpx::product(Cmpx z1, Cmpx z2) {
    ...
}
```

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

```
#include <iostream>
#include "Rational.hh"

int main() {
  std::string s1,s2;
  std::cout << "Enter a rational number: ";
  std::cin >> s1;
  std::cout << "Enter another rational number: ";
  std::cin >> s2;

  Rational q1 {s1};
  Rational q2 {s2};
  Rational sum = q1.plus(q2);
  Rational product = q1.times(q2);

  std::cout << q1.to_string() << std::endl;
  std::cout << q2.to_string() << std::endl;
  std::cout << sum.to_string() << std::endl;
  std::cout << product.to_string() << std::endl;
}
```

MY RATIONAL CLASS SPEC

```
class Rational {

private:
  int num;
  int den;

public:

  // constructors
  Rational(void);
  Rational(std::string s);
  Rational(int n, int d);
  Rational(const Rational& q);

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

MY RATIONAL CLASS CONSTRUCTORS

```
Rational::Rational(int n, int d) {
  if (d == 0) {
    n = 0;
    d = 1;
  }
  if (d < 0) {
    n *= -1;
    d *= -1;
  }
  num = n;
  den = d;
}

Rational::Rational(void) : num {0}, den {1}
{ }

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

MY RATIONAL CLASS CONSTRUCTORS INITIALIZING FIELDS

```
Rational::Rational(int n, int d) {
  if (d == 0) {
    n = 0;
    d = 1;
  }
  if (d < 0) {
    n *= -1;
    d *= -1;
  }
  num = n;
  den = d;
}

Rational::Rational(void) : num {0}, den {1}
{ }

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

MY RATIONAL CLASS CONSTRUCTORS CALLING CONSTRUCTORS

```
Rational::Rational(int n, int d) {
  if (d == 0) {
    n = 0;
    d = 1;
  }
  if (d < 0) {
    n *= -1;
    d *= -1;
  }
  num = n;
  den = d;
}

Rational::Rational(void) : Rational {0,1}
{ }

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

MY RATIONAL CLASS CONSTRUCTORS (CONT'D)

```
Rational::Rational(std::string s) {  // NOTE: simplified from samples
  std::string s_num = "";
  std::string s_den = "1";
  bool saw_slash = false'

  for (int i=0; i<s.length(); i++) {
    char c = s[i];
    if (c >= '0' && c <= '9') {
     if (saw_slash) {
        s_den += c;
      } else {
        s_num += c;
      }
    } else if (c == '-' && i == 0) {
      s_num += c;
    } else if (c == '/') {
      s_den = "";
      saw_slash = true;
    }
  }

  num = std::stoi(s_num);
  den = std::stoi(s_den);
}
```

MY RATIONAL CLASS METHODS

```
Rational Rational::plus(Rational q) {
  return Rational {num*q.den + den*q.num, den*q.den};
}

Rational Rational::times(Rational q) {
  return Rational {num*q.num, den*q.den};
}

std::string Rational::to_string(void) {
  if (den == 1) {
    return std::to_string(num);
  } else {
    return std::to_string(num) + "/" + std::to_string(den);
  }
}
```

NOTE AGAIN HOW ARITHMETIC IS PERFORMED

```
#include <iostream>
#include "Rational.hh"

int main() {
  std::string s1,s2;
  std::cout << "Enter a rational number: ";
  std::cin >> s1;
  std::cout << "Enter another rational number: ";
  std::cin >> s2;

  Rational q1 {s1};
  Rational q2 {s2};
  Rational sum = q1.plus(q2);
  Rational product = q1.times(q2);

  std::cout << q1.to_string() << std::endl;
  std::cout << q2.to_string() << std::endl;
  std::cout << sum.to_string() << std::endl;
  std::cout << product.to_string() << std::endl;
}
```

NOTE AGAIN HOW INPUT AND OUTPUT IS PERFORMED

```
#include <iostream>
#include "Rational.hh"

int main() {
  std::string s1,s2;
  std::cout << "Enter a rational number: ";
  std::cin >> s1;
  std::cout << "Enter another rational number: ";
  std::cin >> s2;

  Rational q1 {s1};
  Rational q2 {s2};
  Rational sum = q1.plus(q2);
  Rational product = q1.times(q2);

  std::cout << q1.to_string() << std::endl;
  std::cout << q2.to_string() << std::endl;
  std::cout << sum.to_string() << std::endl;
  std::cout << product.to_string() << std::endl;
}
```

BUT WHAT IF WE COULD JUST...

```
#include <iostream>
#include "Rational.hh"

int main() {
  std::string s1,s2;
  std::cout << "Enter a rational number: ";
  std::cin >> s1;
  std::cout << "Enter another rational number: ";
  std::cin >> s2;

  Rational q1 {s1};
  Rational q2 {s2};
  Rational sum = q1.plus(q2);
  Rational product = q1.times(q2);

  std::cout << q1.to_string() << std::endl;
  std::cout << q2.to_string() << std::endl;
  std::cout << sum.to_string() << std::endl;
  std::cout << product.to_string() << std::endl;
}
```

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

```
#include <iostream>
#include "Rational.hh"

int main() {
  Rational q1,q2;
  std::cout << "Enter a rational number: ";
  std::cin >> q1;
  std::cout << "Enter another rational number: ";
  std::cin >> q2;



  Rational sum = q1+q2;
  Rational product = q1*q2;

  std::cout << q1 << std::endl;
  std::cout << q2 << std::endl;
  std::cout << sum << std::endl;
  std::cout << product << std::endl;
}
```

OVERLOADING OPERATORS TO TAKE RATIONAL VALUES

```
#include <iostream>
#include "Rational.hh"

int main() {
  Rational q1;
  Rational q2;

  std::cout << "Enter a rational number: ";
  std::cin >> q1;
  std::cout << "Enter another rational number: ";
  std::cin >> q2;

  std::cout << "The first was " << q1 << "." << std::endl;
  std::cout << "The second was " << q2 << "." << std::endl;
  std::cout << "Their sum is " << (q1 + q2) << "." << std::endl;
  std::cout << "Product is " << (q1 * q2) << "." << std::endl;
}
```

MY RATIONAL CLIENT USING OVERLOADED OPERATORS

```
#include <iostream>
#include "Rational.hh"

int main() {
  Rational q1;
  Rational q2;

  std::cout << "Enter a rational number: ";
  std::cin >> q1;
  std::cout << "Enter another rational number: ";
  std::cin >> q2;

  std::cout << "The first was " << q1 << "." << std::endl;
  std::cout << "The second was " << q2 << "." << std::endl;
  std::cout << "Their sum is " << (q1 + q2) << "." << std::endl;
  std::cout << "Product is " << (q1 * q2) << "." << std::endl;
}
```

ADDITIONS TO RATIONAL.HH FOR OVERLOADED OPERATORS

```
class Rational {
private:
  int num;
  int den;
public:
  ...
  // methods
  Rational plus(Rational that);
  Rational times(Rational that);
  std::string to_string(void) const;
};

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);
```

ADDITIONS TO RATIONAL.HH FOR OVERLOADED OPERATORS

```
class Rational {
private:
  int num;
  int den;
public:
  ...
  // methods
  Rational plus(Rational that);
  Rational times(Rational that);
  std::string to_string(void) const;
};

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

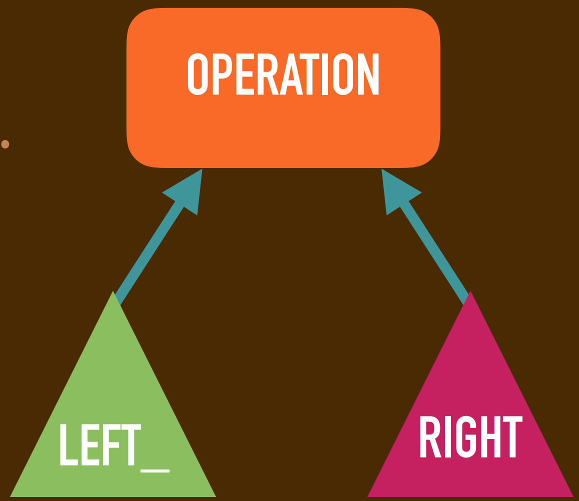
- In C++ code, +, *, >>, << are binary operations.

```
Rational q1;
    Rational q2;
    std::cin >> q1;
    std::cin >> q2;
    Rational sum = (q1 + q2);
    std::cout << sum;
```

USE OF BINARY OPERATIONS

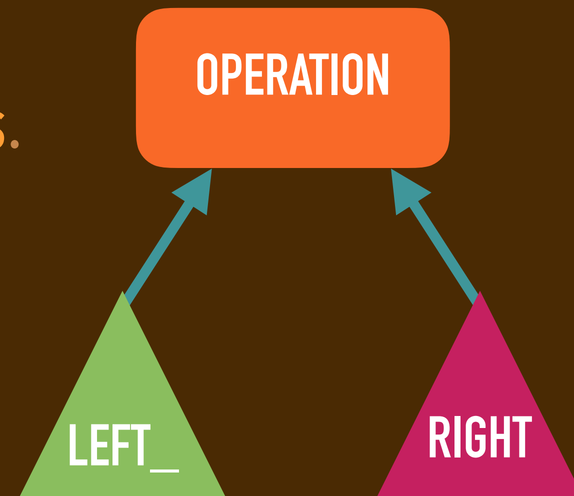
- In C++ code, +, *, >>, << are binary operations.

```
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.

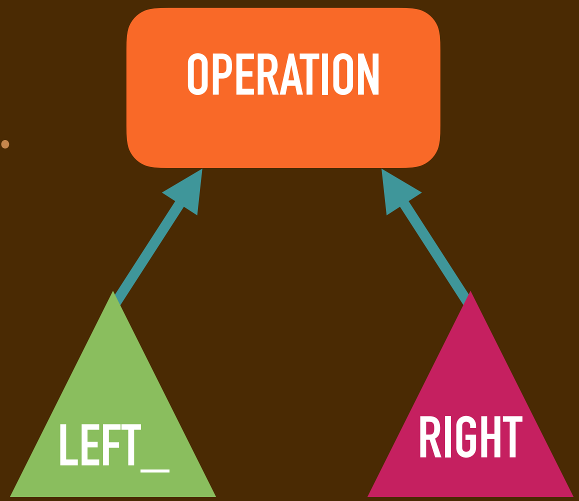
```
Rational q1;
  Rational q2;
  std::cin >> q1;
  std::cin >> q2;
  Rational sum = (q1 >> q2);
  std::cout << sum;
```

- 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.

```
Rational q1;
   Rational q2;
   std::cin >> q1;
   std::cin >> q2;
   Rational sum = (q1 >> q2);
   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)
```

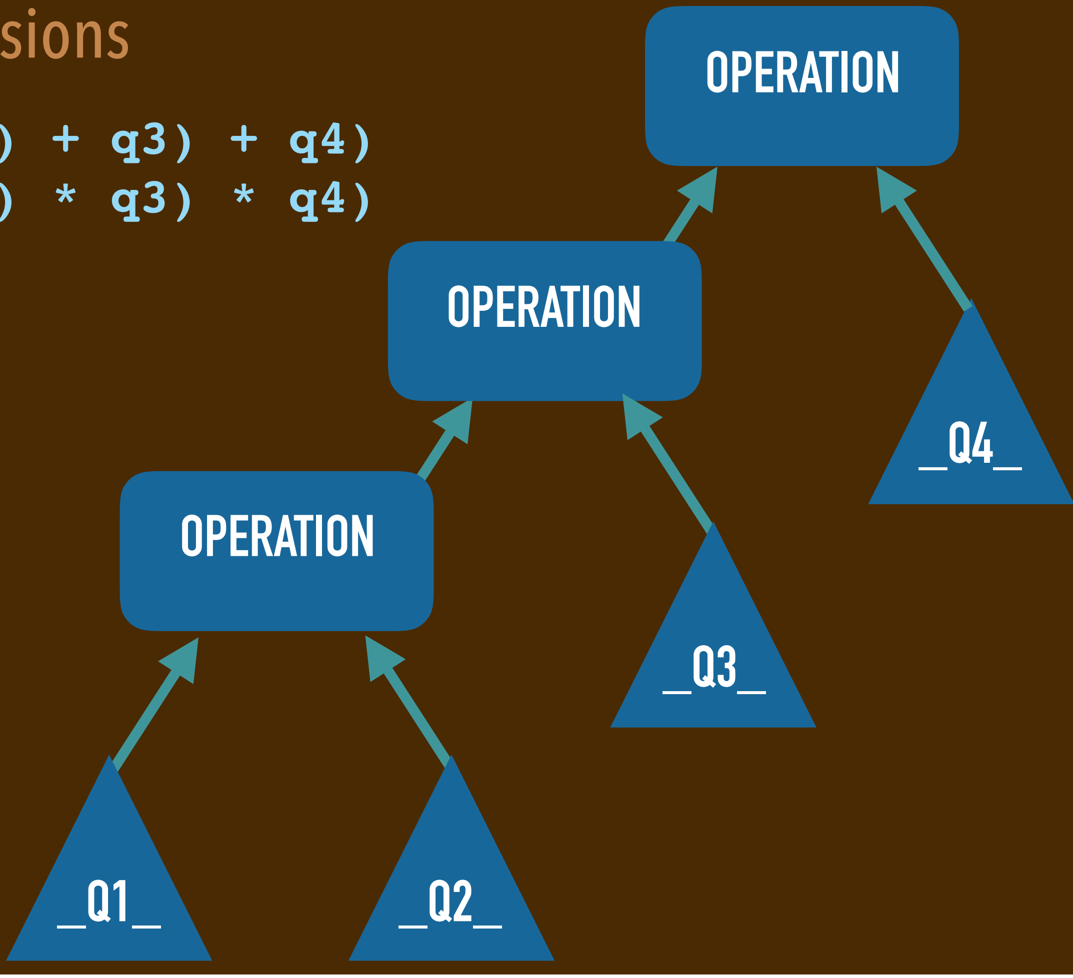
LEFT ASSOCIATIVITY OF PLUS AND TIMES

- 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)
```



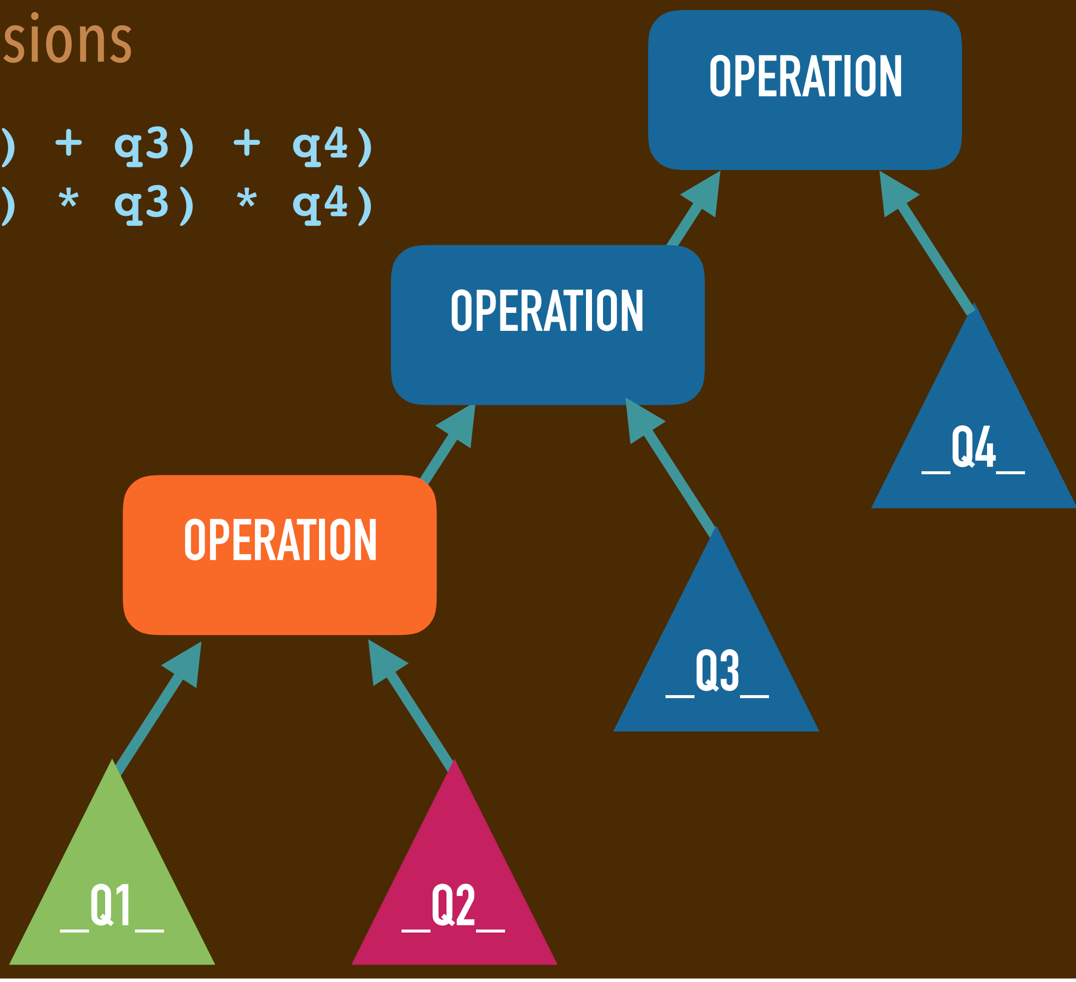
LEFT ASSOCIATIVITY OF PLUS AND TIMES

- 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)
```



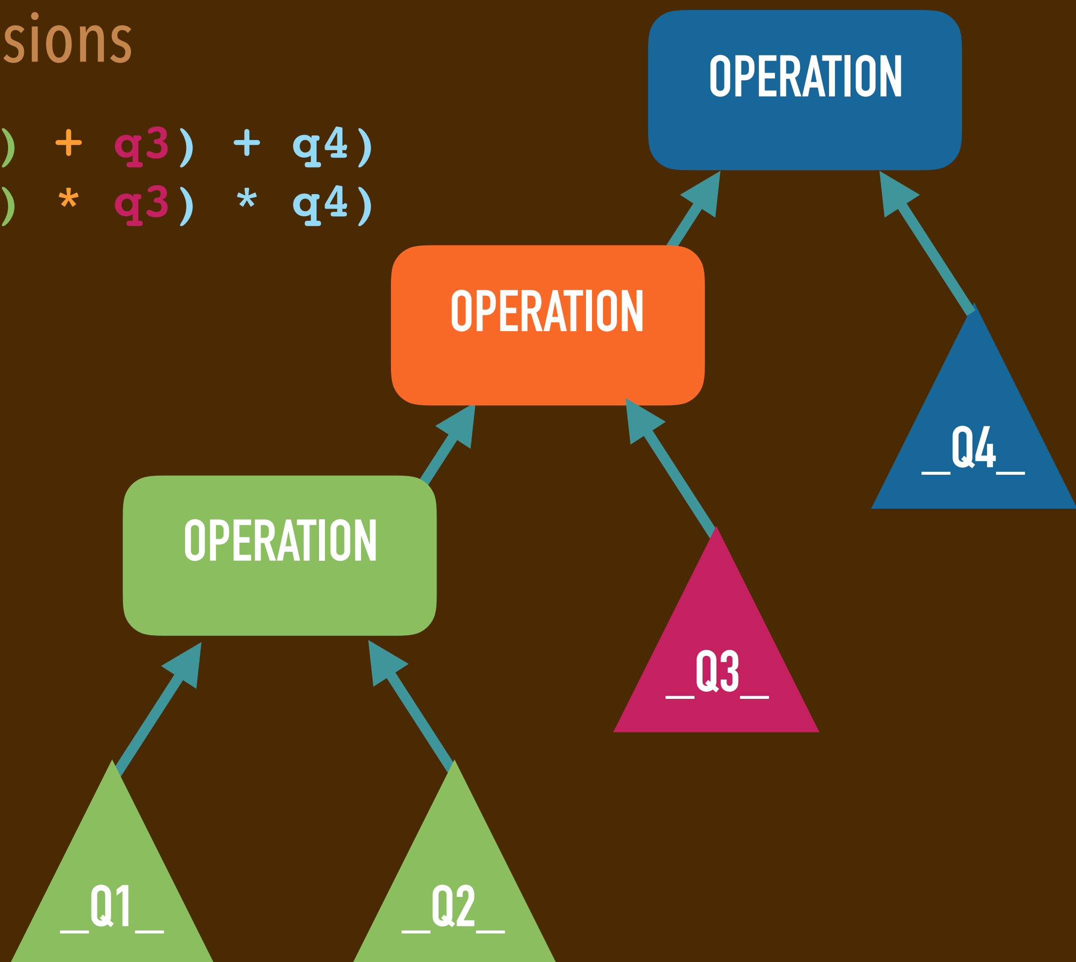
LEFT ASSOCIATIVITY OF PLUS AND TIMES

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

```
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;
```

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`**.

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::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.

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

```
class Rational {
private:
   int num;
   int den;
public:
   ...
   Rational operator-(void);
   int operator[](std::string s);
   ...
};
```

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

OPERATOR METHODS

```
class Rational {
private:
  int num;
  int den;
public:
  ...
  Rational operator-(void);
  int operator[](std::string s);
  ...
};
```

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

```
class Rational {
private:
  int num;
  int den;
public:
  ...
  Rational operator-(void);
  int operator[](std::string s);
  ...
};
```

In the above, I'm overloading **unary minus** and **indexing**. A client can:

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

CONTAINER EXAMPLE: A STACK OBJECT CLASS

```
class Stck {

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

public:
    Stck(int capacity);
    bool is_empty();
    void push(int value);
    int pop();
    int top();
  ~Stck();
};
```

CONTAINER EXAMPLE: A STACK OBJECT CLASS

```
class Stck {

private:
    int *elements; // this will be an array of size capacity
    int num_elements;
    int capacity;

public:
    Stck(int capacity);
    bool is_empty();
    void push(int value);
    int pop();
    int top();
  ~Stck();
};
```

CONTAINER EXAMPLE: A STACK OBJECT CLASS

```
class Stck {

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

public:
  Stck(int capacity); # This will heap-allocate the array.
  bool is_empty();
  void push(int value);
  int pop();
  int top();
 ~Stck();
};
```

CONTAINER EXAMPLE: A STACK OBJECT CLASS

```
class Stck {

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

public:
  Stck(int capacity); # This will heap-allocate the array.
  bool is_empty();
  void push(int value);
  int pop();
  int top();
 ~Stck(); # Destructor. This will "delete" the array.
};
```

IMPLEMENTATION OF THE CONSTRUCTOR

```
#include "Stck.hh"

Stck::Stck(int capacity) {
  this->elements = new int[capacity];
  this->num_elements = 0,
  this->capacity = capacity;
}
```

IMPLEMENTATION USING A CONSTRUCTOR INITIALIZER LIST

```
#include "Stck.hh"

Stck::Stck(int capacity) :
  elements {new int[capacity]},
  num_elements {0},
  capacity {capacity}
{ }
```

IMPLEMENTATION OF STACK METHODS

```
bool Stck::is_empty() {
  return (num_elements == 0);
}

void Stck::push(int value) {
  assert(num_elements < capacity);
  elements[num_elements] = value;
  num_elements++;
}

int Stck::pop() {
  assert(!is_empty());
  num_elements--;
  return elements[num_elements];
}

int Stck::top() {
  assert(!is_empty());
  return elements[num_elements-1];
}
```

ILLUSTRATION WITH A SIMPLE CLIENT

```
#include "Stck.hh"
#include <iostream>
int main(void) {
  Stck s {5};
  s.push(7);
  s.push(1);
  s.push(3);
  std::cout << s.pop() << std::endl;
  std::cout << s.pop() << std::endl;
  s.push(11);
  std::cout << s.pop() << std::endl;
}
```

ILLUSTRATION WITH A SIMPLE CLIENT

```
#include "Stck.hh"
#include <iostream>
int main(void) {
  Stck s {5};
  s.push(7);
  s.push(1);
  s.push(3);
  std::cout << s.pop() << std::endl;
  std::cout << s.pop() << std::endl;
  s.push(11);
  std::cout << s.pop() << std::endl;
}
```

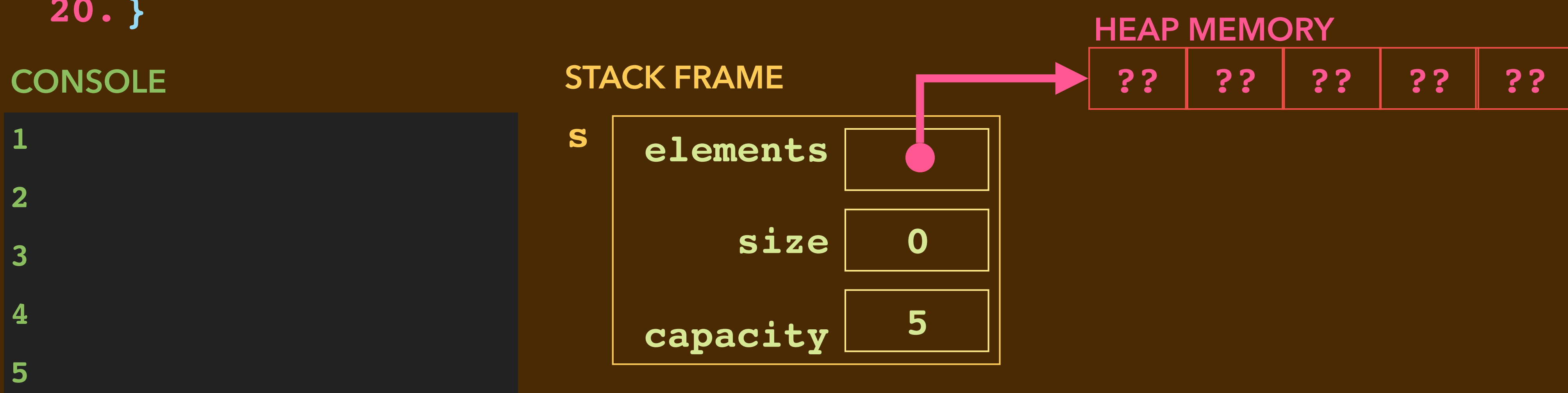


ILLUSTRATION WITH A SIMPLE CLIENT

```cpp
#include "Stck.hh"
#include <iostream>
int main(void) {
  Stck s {5};
  s.push(7);
  s.push(1);
  s.push(3);
  std::cout << s.pop() << std::endl;
  std::cout << s.pop() << std::endl;
  s.push(11);
  std::cout << s.pop() << std::endl;
}
```

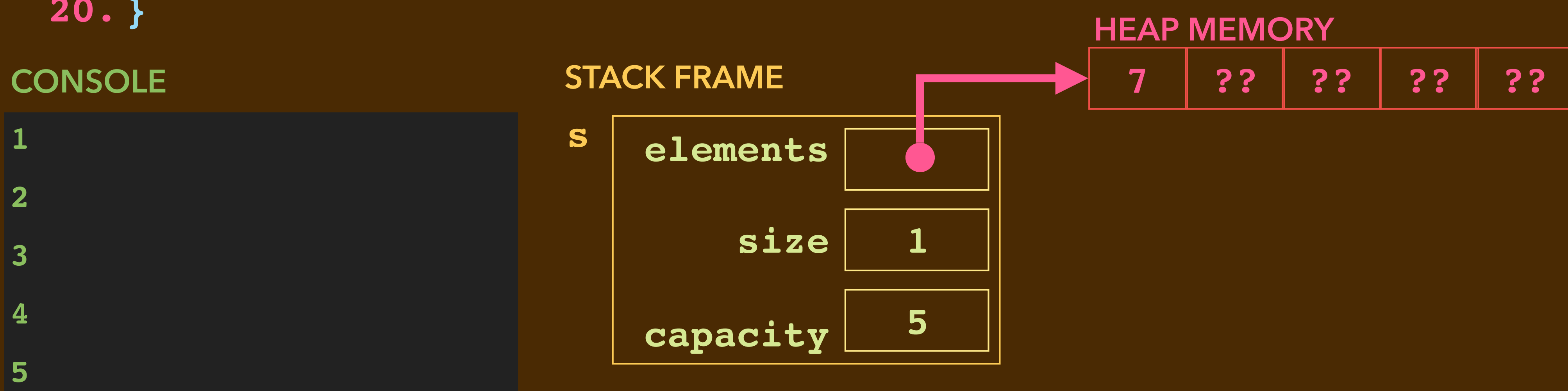


ILLUSTRATION WITH A SIMPLE CLIENT

```
#include "Stck.hh"
#include <iostream>
int main(void) {
  Stck s {5};
  s.push(7);
  s.push(1);
  s.push(3);
  std::cout << s.pop() << std::endl;
  std::cout << s.pop() << std::endl;
  s.push(11);
  std::cout << s.pop() << std::endl;
}
```

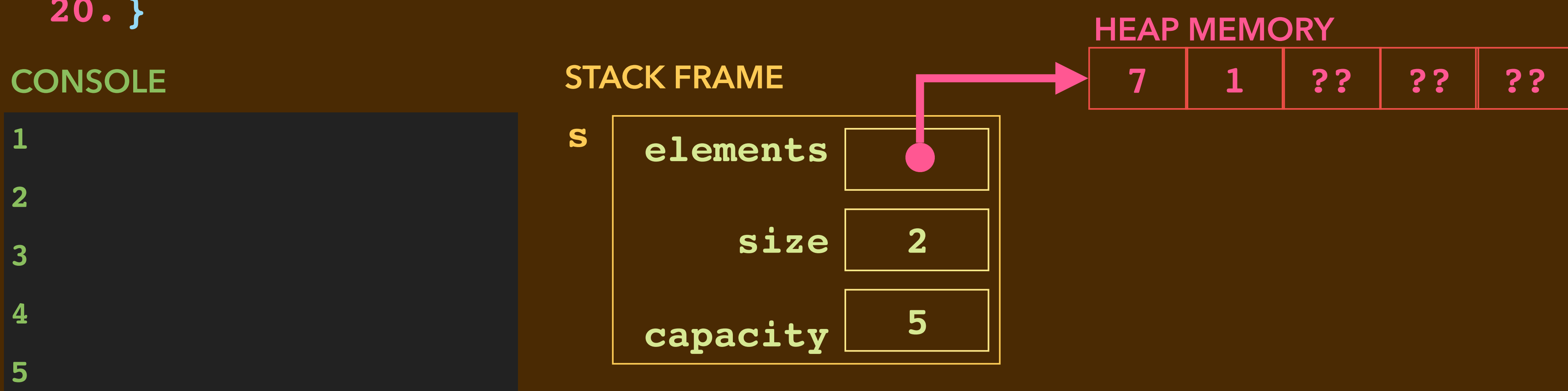


ILLUSTRATION WITH A SIMPLE CLIENT

```
#include "Stck.hh"
#include <iostream>
int main(void) {
  Stck s {5};
  s.push(7);
  s.push(1);
  s.push(3);
  std::cout << s.pop() << std::endl;
  std::cout << s.pop() << std::endl;
  s.push(11);
  std::cout << s.pop() << std::endl;
}
```

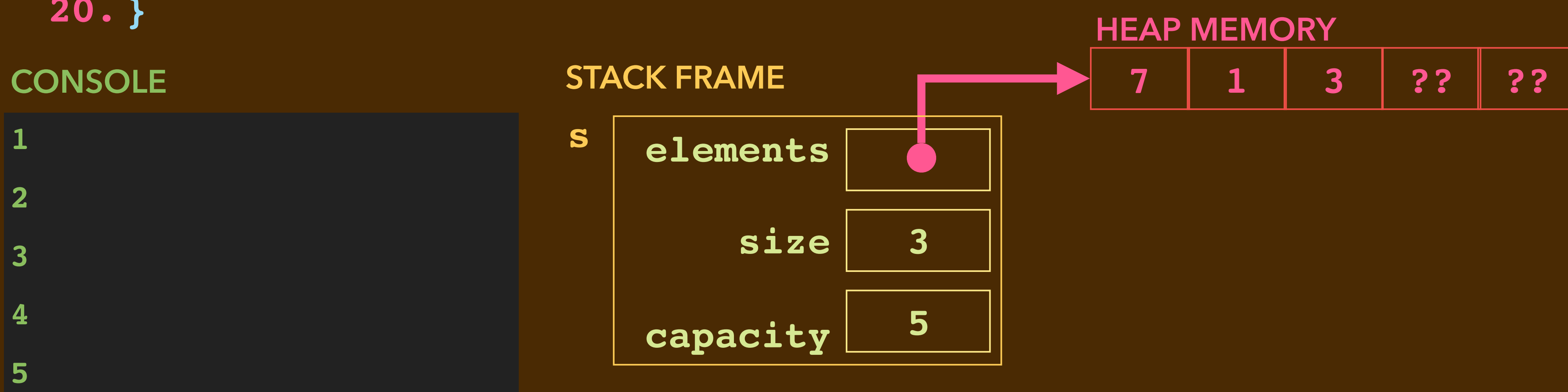


ILLUSTRATION WITH A SIMPLE CLIENT

```
#include "Stck.hh"
#include <iostream>
int main(void) {
  Stck s {5};
  s.push(7);
  s.push(1);
  s.push(3);
  std::cout << s.pop() << std::endl;
  std::cout << s.pop() << std::endl;
  s.push(11);
  std::cout << s.pop() << std::endl;
}
```

CONSOLE

```
3
```

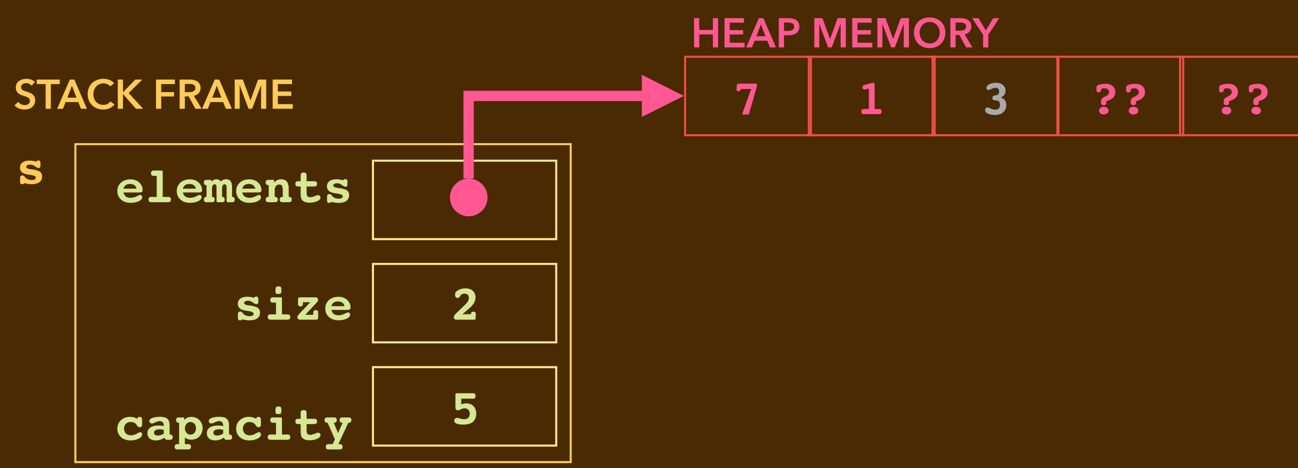


ILLUSTRATION WITH A SIMPLE CLIENT

```
#include "Stck.hh"
#include <iostream>
int main(void) {
  Stck s {5};
  s.push(7);
  s.push(1);
  s.push(3);
  std::cout << s.pop() << std::endl;
  std::cout << s.pop() << std::endl;
  s.push(11);
  std::cout << s.pop() << std::endl;
}
```

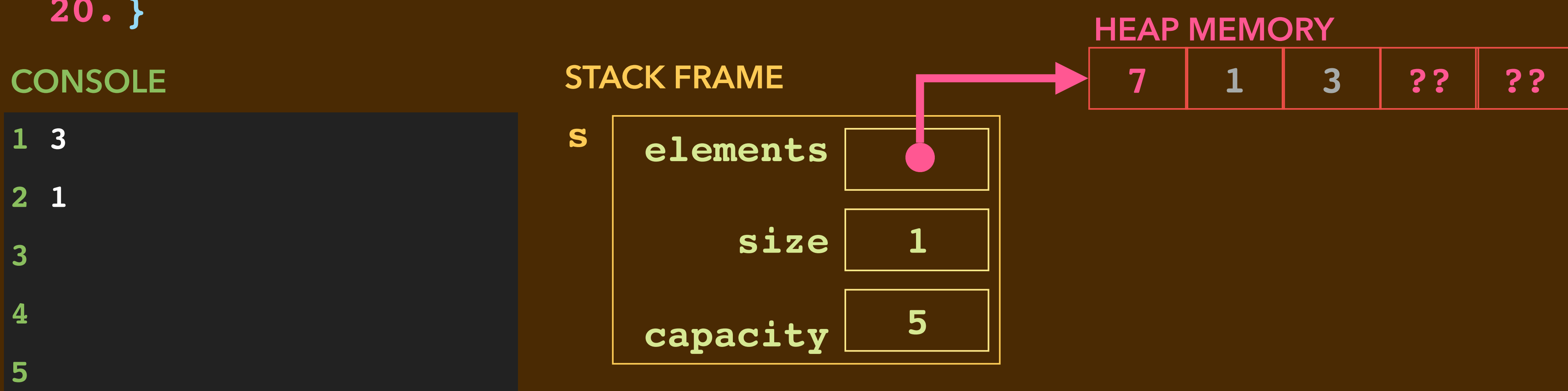


ILLUSTRATION WITH A SIMPLE CLIENT

```
#include "Stck.hh"
#include <iostream>
int main(void) {
  Stck s {5};
  s.push(7);
  s.push(1);
  s.push(3);
  std::cout << s.pop() << std::endl;
  std::cout << s.pop() << std::endl;
  s.push(11);
  std::cout << s.pop() << std::endl;
}
```

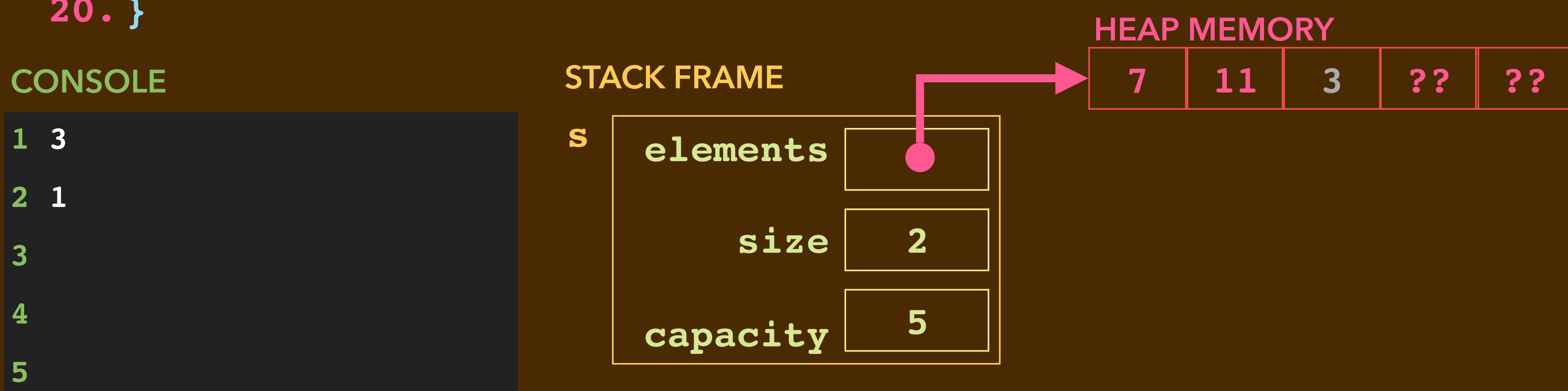
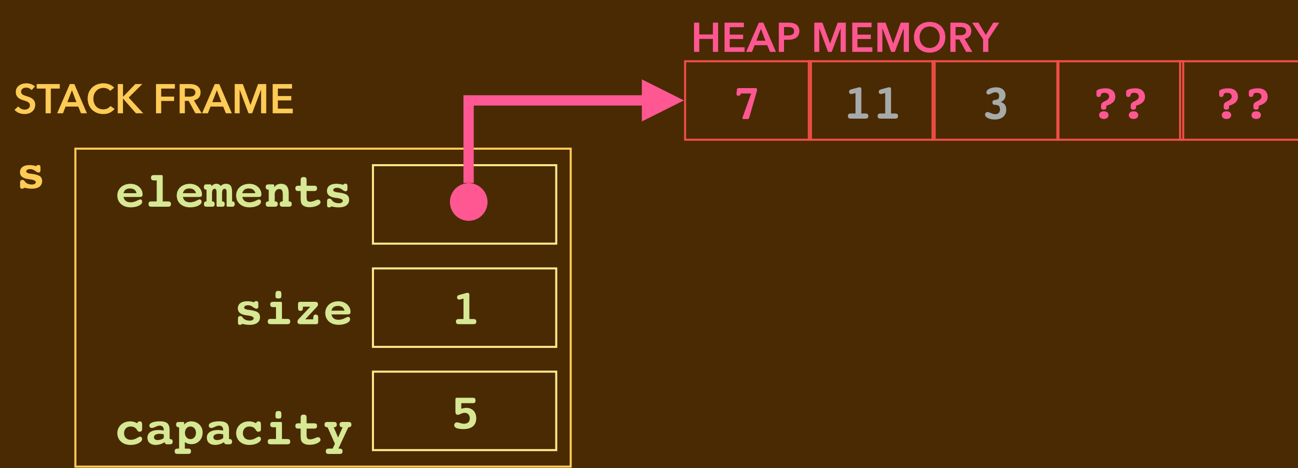


ILLUSTRATION WITH A SIMPLE CLIENT

```
#include "Stck.hh"
#include <iostream>
int main(void) {
  Stck s {5};
  s.push(7);
  s.push(1);
  s.push(3);
  std::cout << s.pop() << std::endl;
  std::cout << s.pop() << std::endl;
  s.push(11);
  std::cout << s.pop() << std::endl;
}
```

CONSOLE

```
3
1
11
```



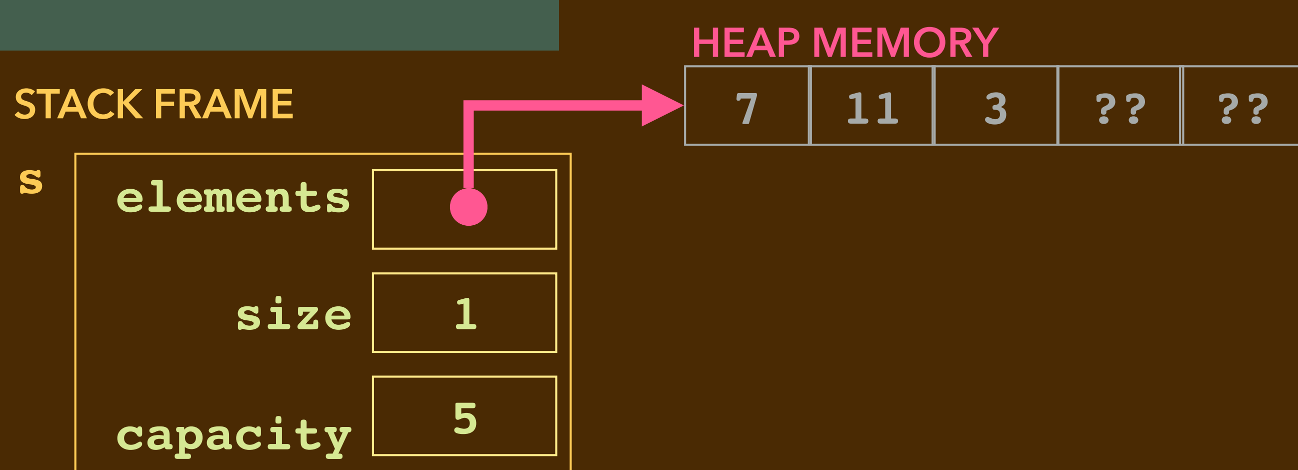
DESTRUCTOR CODE EXECUTION

```
#include "Stck.hh"
#include <iostream>
int main(void) {
  Stck s {5};
  s.push(7);
  s.push(1);
  s.push(3);
  std::cout << s.pop() << std::endl;
  std::cout << s.pop() << std::endl;
  s.push(11);
  std::cout << s.pop() << std::endl;
}
```

Calls the destructor.

CONSOLE

```
3
1
11
```



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:

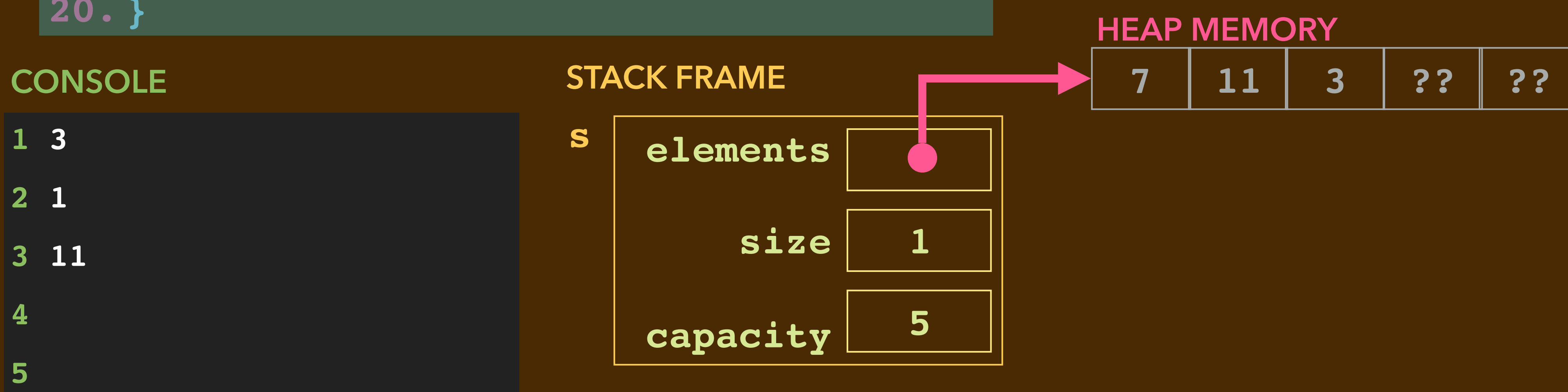
```
Stck::~Stck() {
    delete [] elements;
}
```

- 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

```
#include "Stck.hh"
#include <iostream>
int main(void) {
  Stck s {5};
  s.push(7);
  s.push(1);
  s.push(3);
  std::cout << s.pop() << std::endl;
  std::cout << s.pop() << std::endl;
  s.push(11);
  std::cout << s.pop() << std::endl;
}
```

Calls the destructor.



IMPLICIT CALL OF THE DESTRUCTOR

```
#include "Stck.hh"
#include <iostream>
int main(void) {
  Stck s {5};
  s.push(7);
  s.push(1);
  s.push(3);
  std::cout << s.pop() << std::endl;
  std::cout << s.pop() << std::endl;
  s.push(11);
  std::cout << s.pop() << std::endl;
}
```

Calls the

And the frame gets taken down.

CONSOLE

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

STACK FRAME

s

elements	
size	1
capacity	5

HEAP MEMORY

7	11	3	??	??

IMPLICIT CALL OF THE DESTRUCTOR

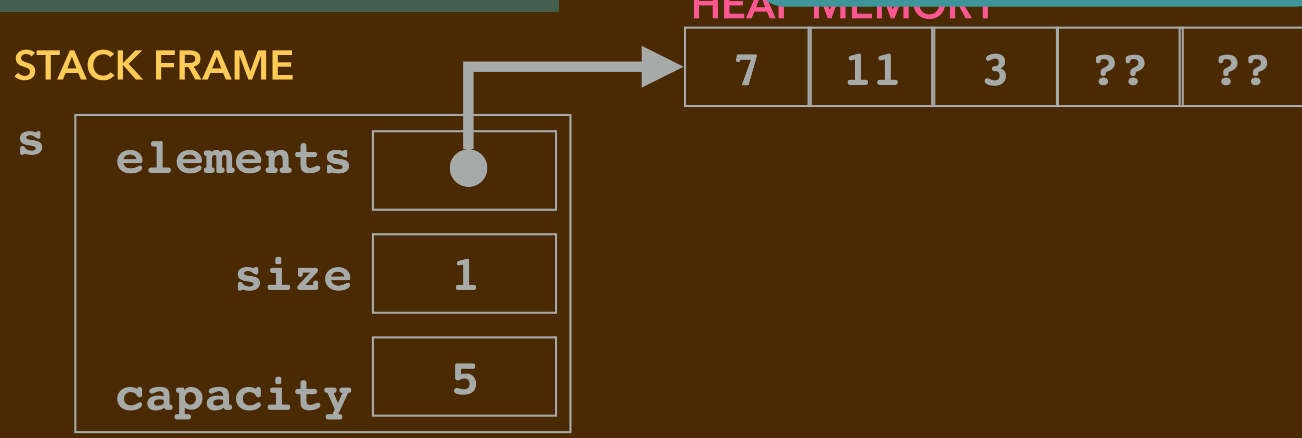
```
#include "Stck.hh"
#include <iostream>
int main(void) {
  Stck s {5};
  s.push(7);
  s.push(1);
  s.push(3);
  std::cout << s.pop() << std::endl;
  std::cout << s.pop() << std::endl;
  s.push(11);
  std::cout << s.pop() << std::endl;
}
```

Calls the

And the frame gets taken down.

CONSOLE

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



HEAP-ALLOCATED STACK

```
#include "Stck.hh"
#include <iostream>
int main(void) {
  Stck* s = new Stck {5};
  s->push(7);
  s->push(1);
  s->push(3);
  std::cout << s->pop() << std::endl;
  std::cout << s->pop() << std::endl;
  s->push(11);
  std::cout << s->pop() << std::endl;
  delete s;
}
```

HEAP-ALLOCATED STACK

```
#include "Stck.hh"
#include <iostream>
int main(void) {
  Stck* s = new Stck {5};
  s->push(7);
  s->push(1);
  s->push(3);
  std::cout << s->pop() << std::endl;
  std::cout << s->pop() << std::endl;
  s->push(11);
  std::cout << s->pop() << std::endl;
  delete s;
}
```

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

HEAP-ALLOCATED STACK

```
#include "Stck.hh"
#include <iostream>
int main(void) {
  Stck* s = new Stck {5};
  s->push(7);
  s->push(1);
  s->push(3);
  std::cout << s->pop() << std::endl;
  std::cout << s->pop() << std::endl;
  s->push(11);
  std::cout << s->pop() << std::endl;
  delete s;
}
```

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

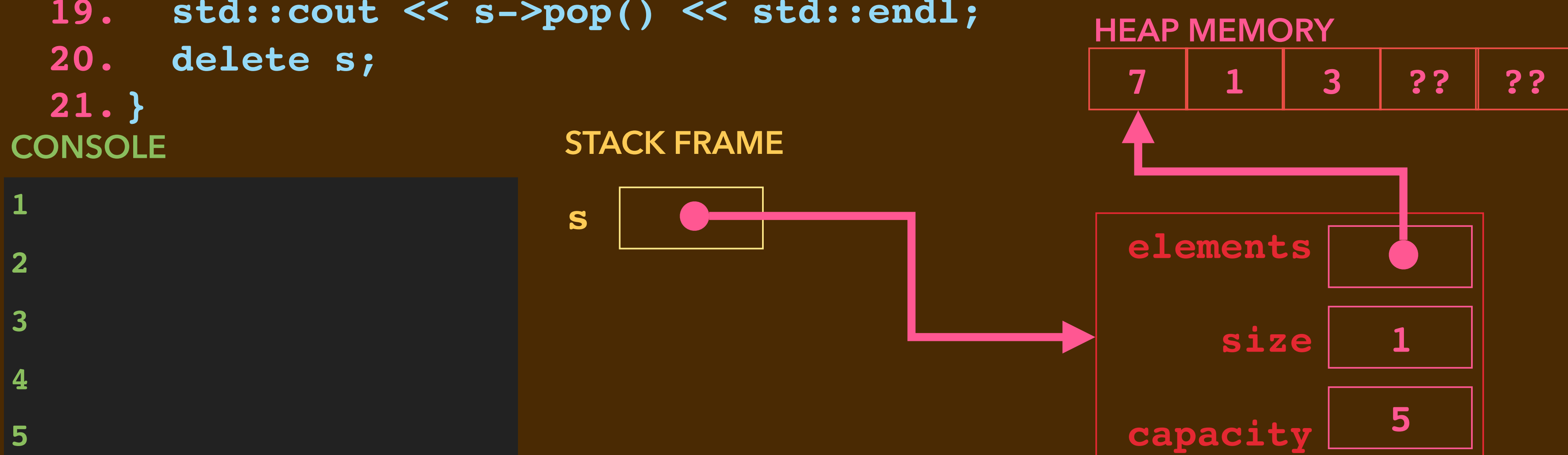
HEAP-ALLOCATED STACK

```
#include "Stck.hh"
#include <iostream>
int main(void) {
  Stck* s = new Stck {5};
  s->push(7);
  s->push(1);
  s->push(3);
  std::cout << s->pop() << std::endl;
  std::cout << s->pop() << std::endl;
  s->push(11);
  std::cout << s->pop() << std::endl;
  delete s;
}
```

- 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

```
#include "Stck.hh"
#include <iostream>
int main(void) {
  Stck* s = new Stck {5};
  s->push(7);
  s->push(1);
  s->push(3);
  std::cout << s->pop() << std::endl;
  std::cout << s->pop() << std::endl;
  s->push(11);
  std::cout << s->pop() << std::endl;
  delete s;
}
```

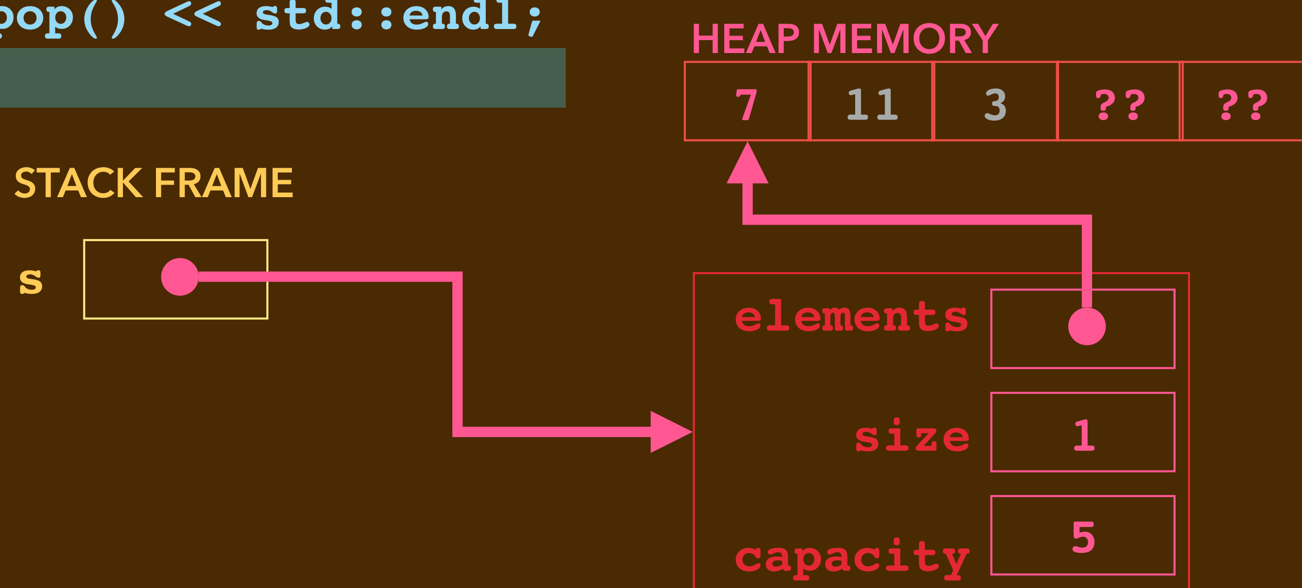


HEAP-ALLOCATED STACK ILLUSTRATED

```
#include "Stck.hh"
#include <iostream>
int main(void) {
  Stck* s = new Stck {5};
  s->push(7);
  s->push(1);
  s->push(3);
  std::cout << s->pop() << std::endl;
  std::cout << s->pop() << std::endl;
  s->push(11);
  std::cout << s->pop() << std::endl;
  delete s;
}
```

CONSOLE

```
3
1
11
```



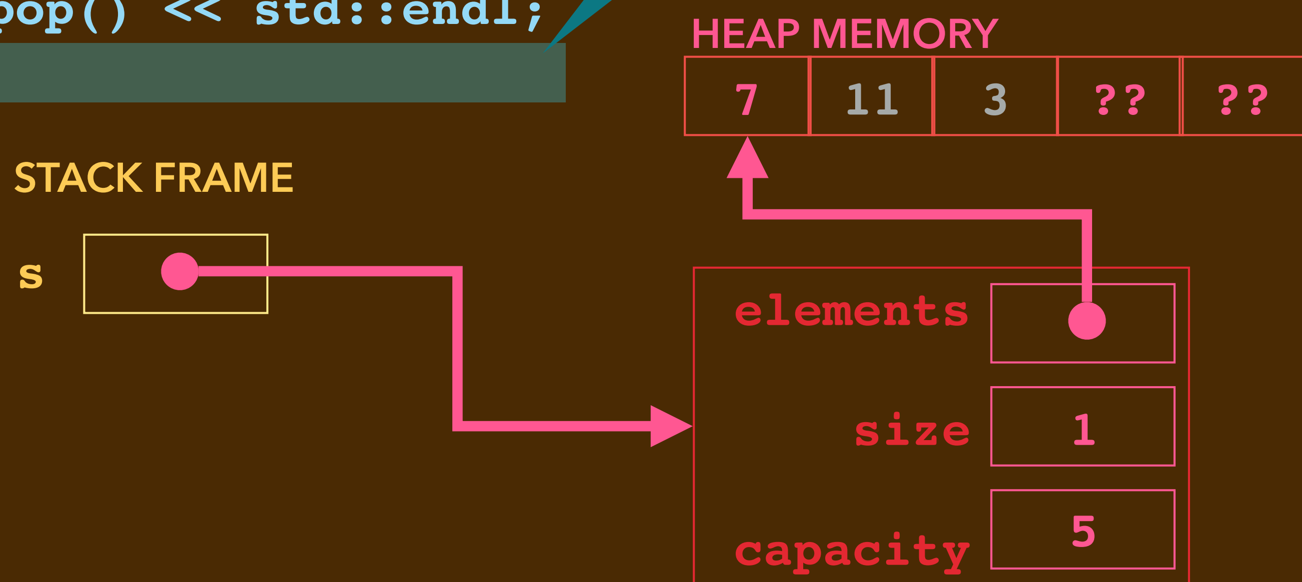
HEAP-ALLOCATED STACK ILLUSTRATED

```
#include "Stck.hh"
#include <iostream>
int main(void) {
  Stck* s = new Stck {5};
  s->push(7);
  s->push(1);
  s->push(3);
  std::cout << s->pop() << std::endl;
  std::cout << s->pop() << std::endl;
  s->push(11);
  std::cout << s->pop() << std::endl;
  delete s;
}
```

The destructor code gets called with delete

CONSOLE

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



HEAP-ALLOCATED STACK ILLUSTRATED

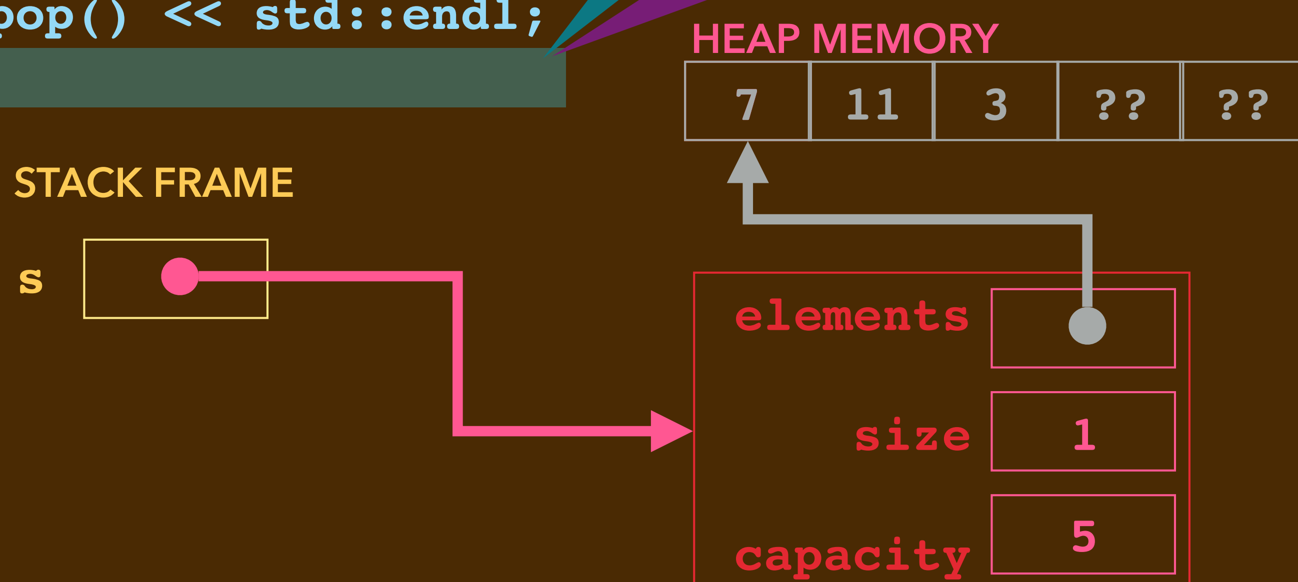
```
#include "Stck.hh"
#include <iostream>
int main(void) {
  Stck* s = new Stck {5};
  s->push(7);
  s->push(1);
  s->push(3);
  std::cout << s->pop() << std::endl;
  std::cout << s->pop() << std::endl;
  s->push(11);
  std::cout << s->pop() << std::endl;
  delete s;
}
```

The destructor code gets called with

...which deletes `s->elements`.

CONSOLE

```
3
1
11
```

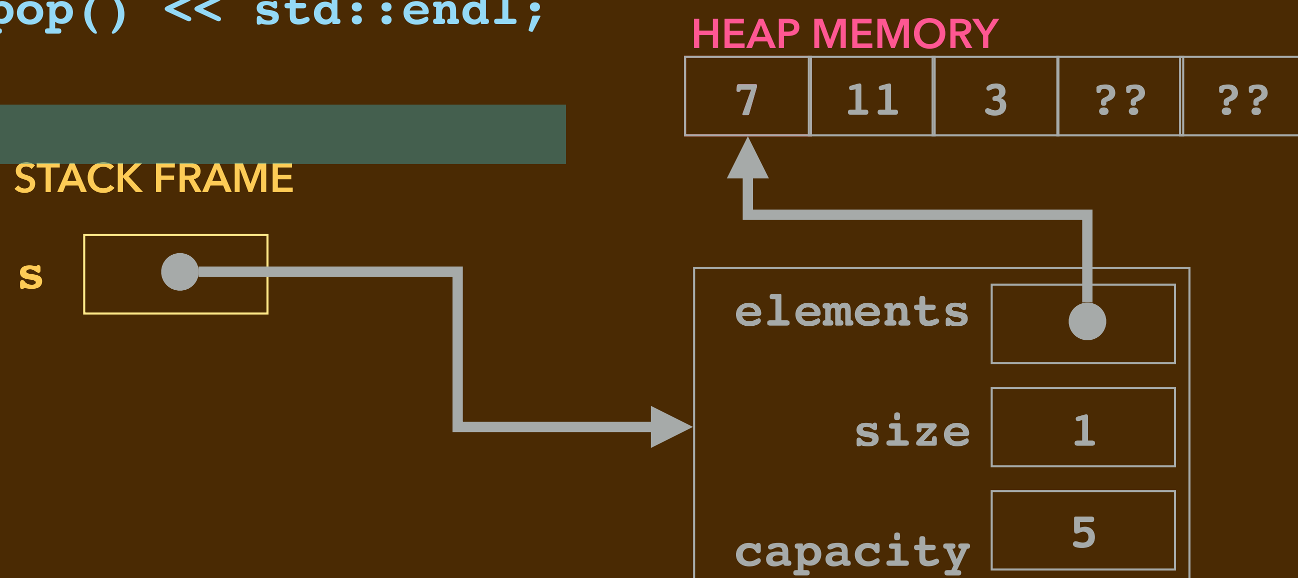


HEAP-ALLOCATED STACK ILLUSTRATED

```cpp
#include "Stck.hh"
#include <iostream>
int main(void) {
  Stck* s = new Stck {5};
  s->push(7);
  s->push(1);
  s->push(3);
  std::cout << s->pop() << std::endl;
  std::cout << s->pop() << std::endl;
  s->push(11);
  std::cout << s->pop() << std::endl;
  delete s;
}
```

CONSOLE

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



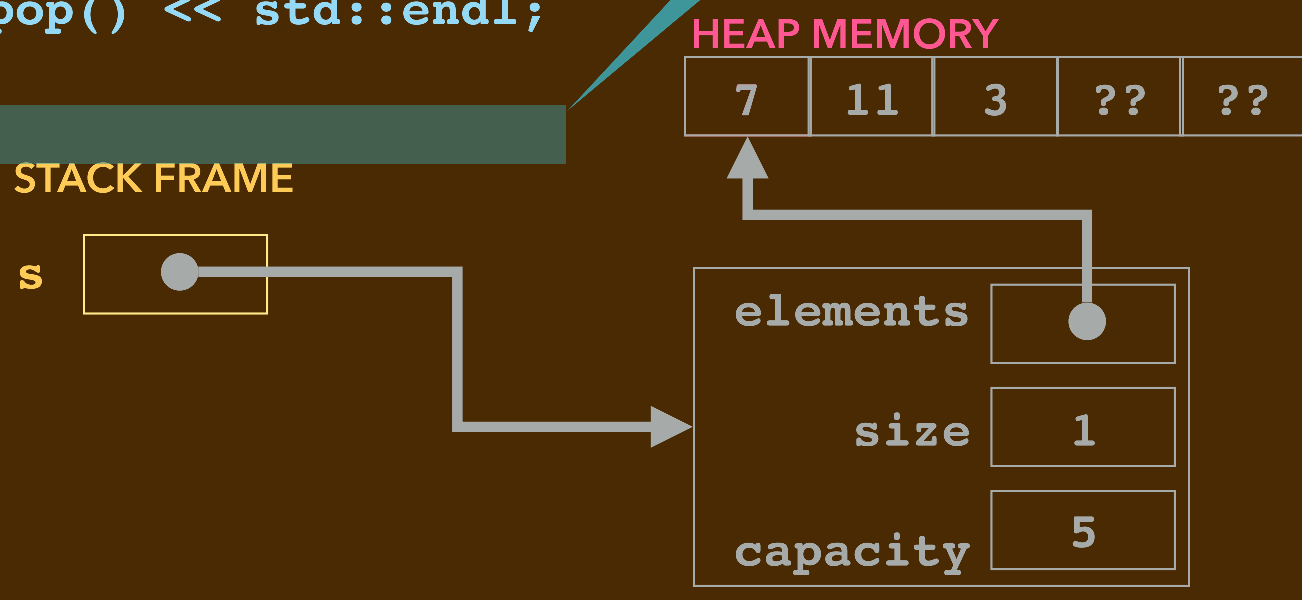
HEAP-ALLOCATED STACK ILLUSTRATED

```
#include "Stck.hh"
#include <iostream>
int main(void) {
  Stck* s = new Stck {5};
  s->push(7);
  s->push(1);
  s->push(3);
  std::cout << s->pop() << std::endl;
  std::cout << s->pop() << std::endl;
  s->push(11);
  std::cout << s->pop() << std::endl;
  delete s;
}
```

Then the frame gets taken down.

CONSOLE

```
3
1
11
```



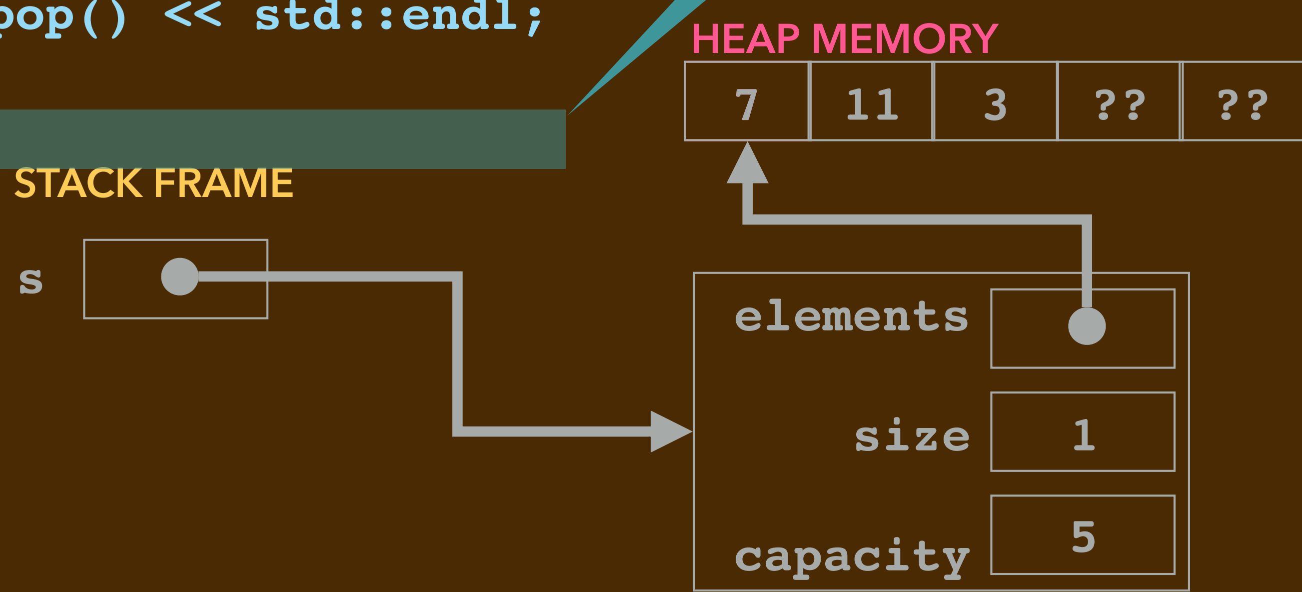
HEAP-ALLOCATED STACK ILLUSTRATED

```
#include "Stck.hh"
#include <iostream>
int main(void) {
  Stck* s = new Stck {5};
  s->push(7);
  s->push(1);
  s->push(3);
  std::cout << s->pop() << std::endl;
  std::cout << s->pop() << std::endl;
  s->push(11);
  std::cout << s->pop() << std::endl;
  delete s;
}
```

Then the frame gets taken down.

CONSOLE

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



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!