IMMUTABILITY, REFERENCE, AND INHERITANCE

LECTURE 12-1

JIM FIX, REED COLLEGE CS2-S20

TODAY'S PLAN

- FINISH DESTRUCTORS
- PASSING PARAMETERS BY REFERENCE
- IMMUTABILITY WITH `const`
- INHERITANCE
  - ACCOUNT EXAMPLES
  - DYNAMIC DISPATCH WITH `virtual`

THIS WEEK'S PLAN

- **There is no lab tomorrow. Work on Homework 11.**
- **Wednesday:**
  - TEMPLATES
  - STANDARD TEMPLATE LIBRARY
  - INTRODUCE PROJECT 2

CS FACULTY CANDIDATES THIS/NEXT WEEK...

- **Tuesday/Tomorrow @4:30pm over Zoom:**

  Archita Agarwal, Brown University

  "Encrypted Distributed Storage Systems"

- **Wednesday @4:30pm over Zoom:**

  Tanya Amert, University of North Carolina

  "Enabling Real-Time Certification of Autonomous-Driving Applications"

- **Next Monday @4:30pm over Zoom:**

  Sonia Roberts, University of Pennsylvania

  (Title forthcoming; will be on her robotics research.)

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.
};
```

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
```

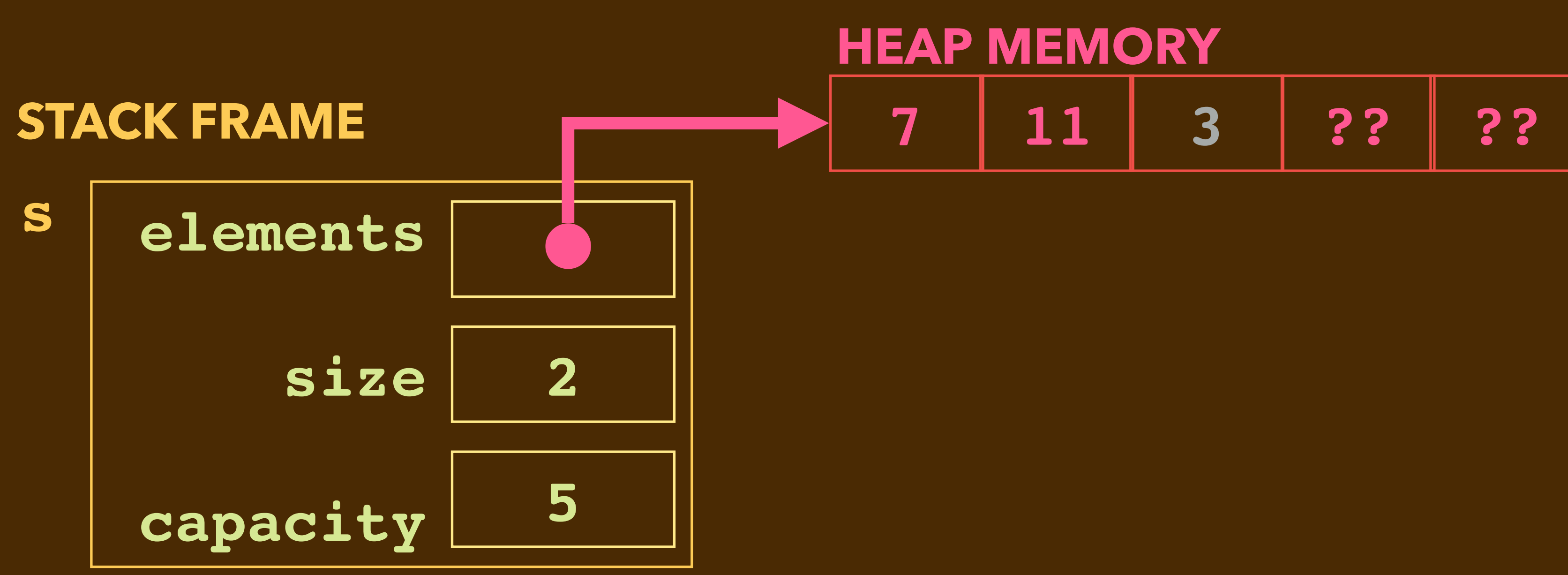


ILLUSTRATION WITH A SIMPLE CLIENT

```
#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
```

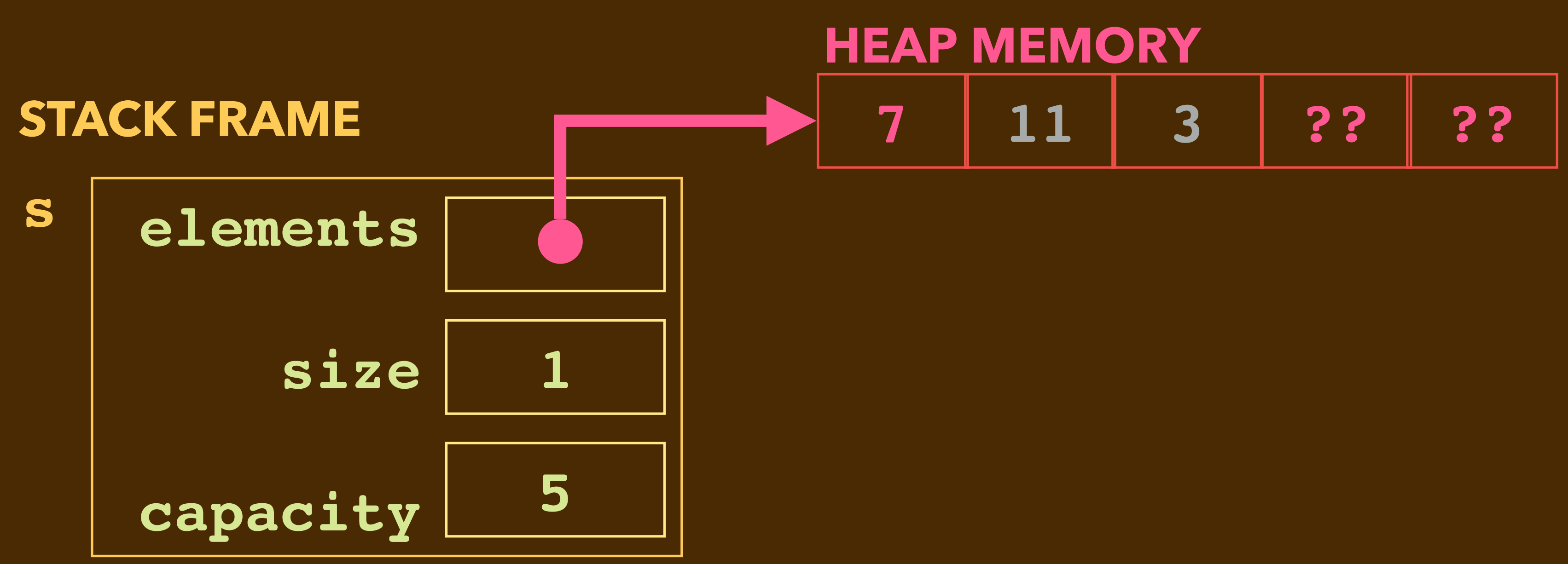


ILLUSTRATION WITH A SIMPLE CLIENT

```
#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;
}
```

Calls the default destructor.

CONSOLE

```
3
1
11
```

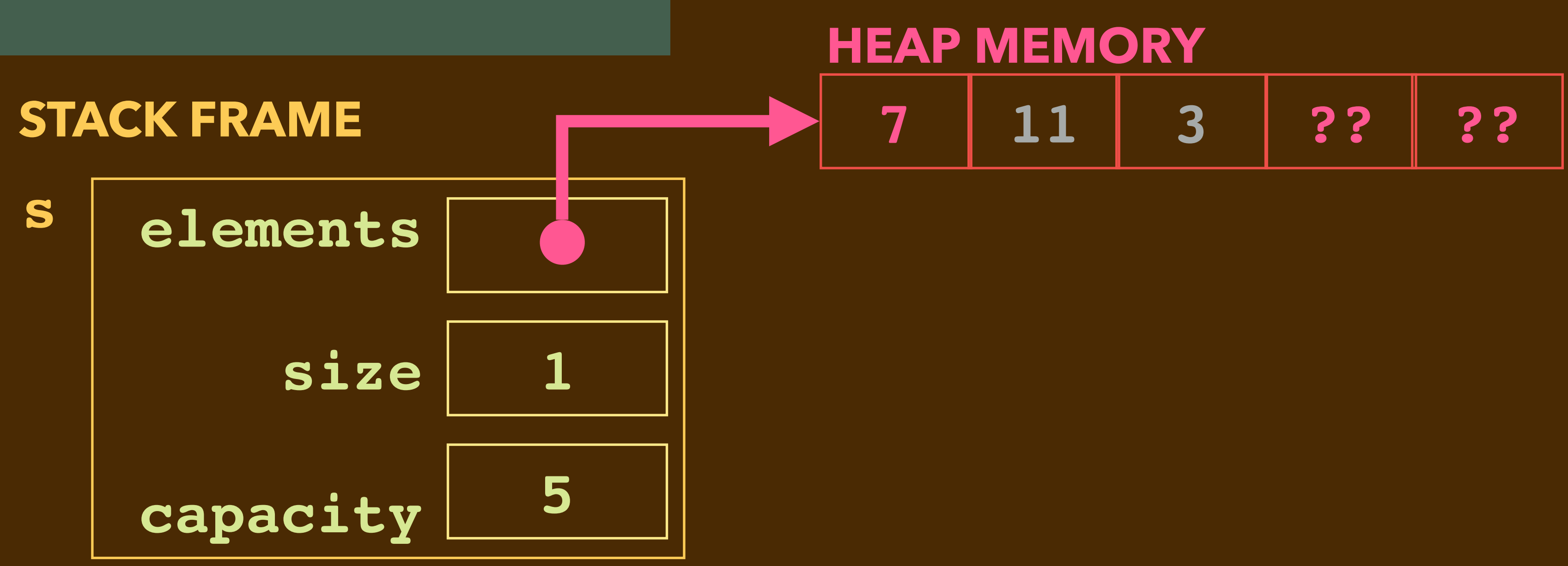


ILLUSTRATION WITH A SIMPLE CLIENT

```
#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;
}
```

Calls the default destructor.

CONSOLE

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

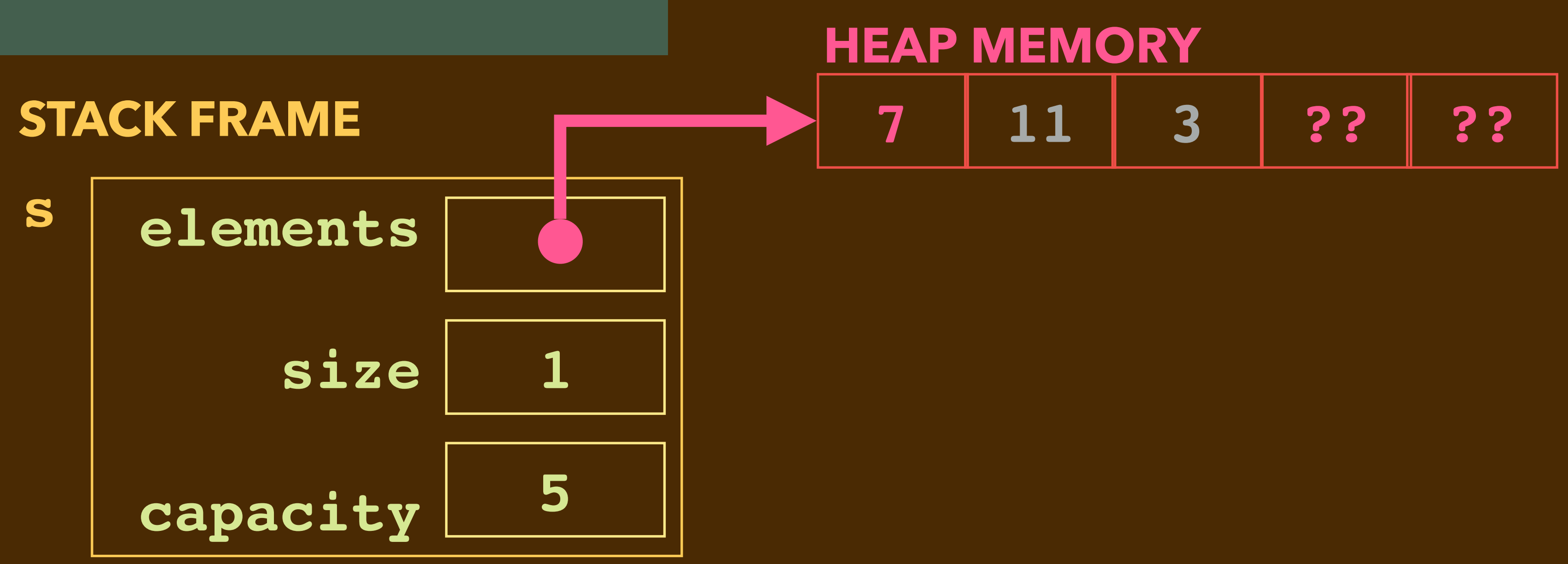


ILLUSTRATION WITH A SIMPLE CLIENT

```
#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;
}
```

Calls the default

And the frame gets taken down.

CONSOLE

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

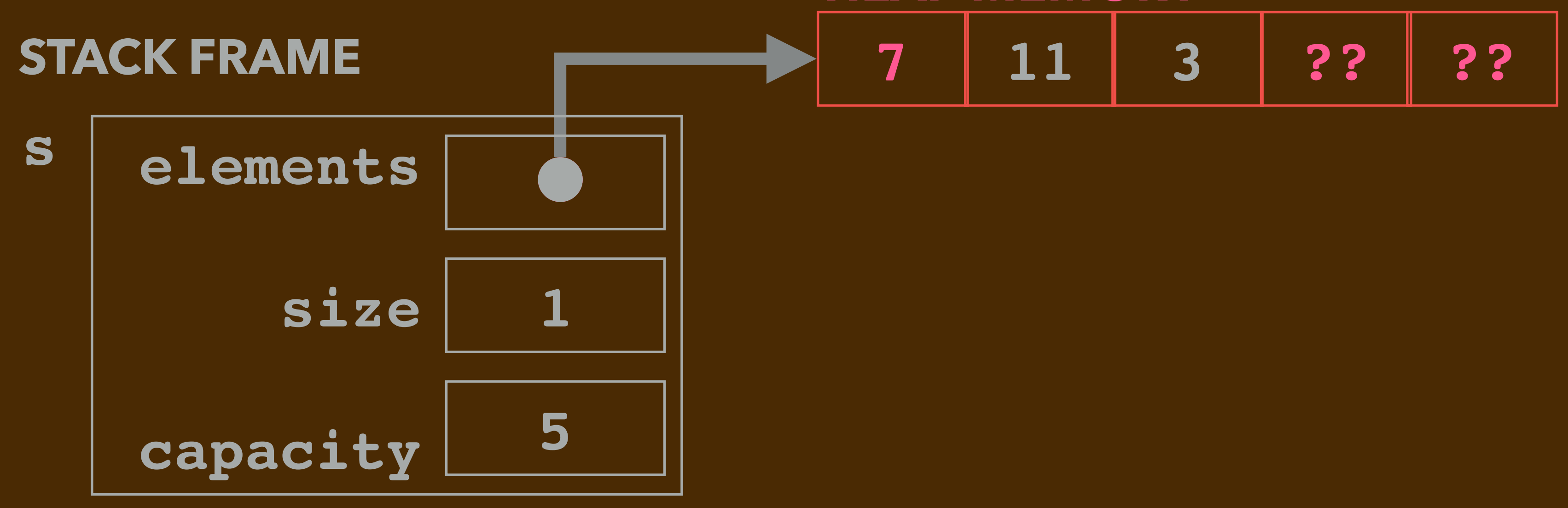


ILLUSTRATION WITH A SIMPLE CLIENT

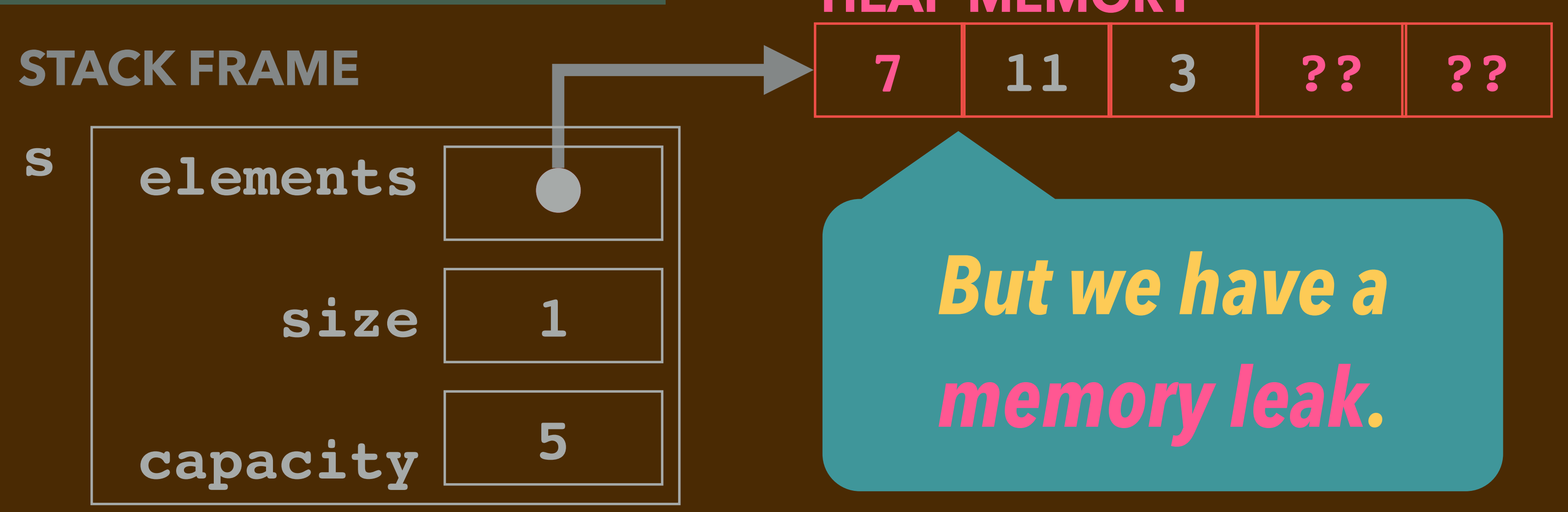
```
#include "Stck.hh"
#include <iostream>
int main(void) {
  Stck s {5};
  s.push(7);
  s.push(1);
  s.push(3);
  std::cout << s.pop() << std::endl;
  std::cout << s.pop() << std::endl;
  s.push(11);
  std::cout << s.pop() << std::endl;
}
```

Calls the default

And the frame gets taken down.

CONSOLE

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



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;
}
```

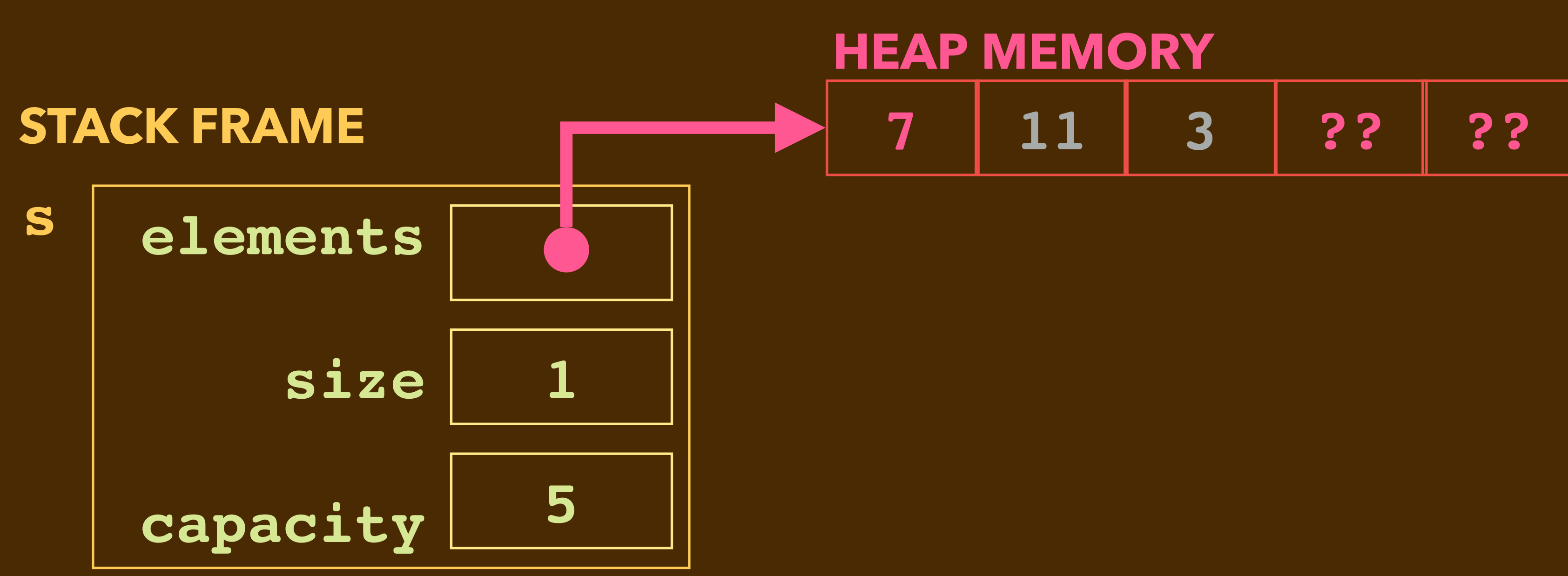
- In this case, we simply delete the pointer to the elements array.
- If we didn't, we'd have a *memory leak*.
  - The 5 words would be reserved, but the program has no access to them.
- This just undoes the work of the constructor; gives back the heap storage.

ILLUSTRATION WITH A SIMPLE CLIENT

```
#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

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



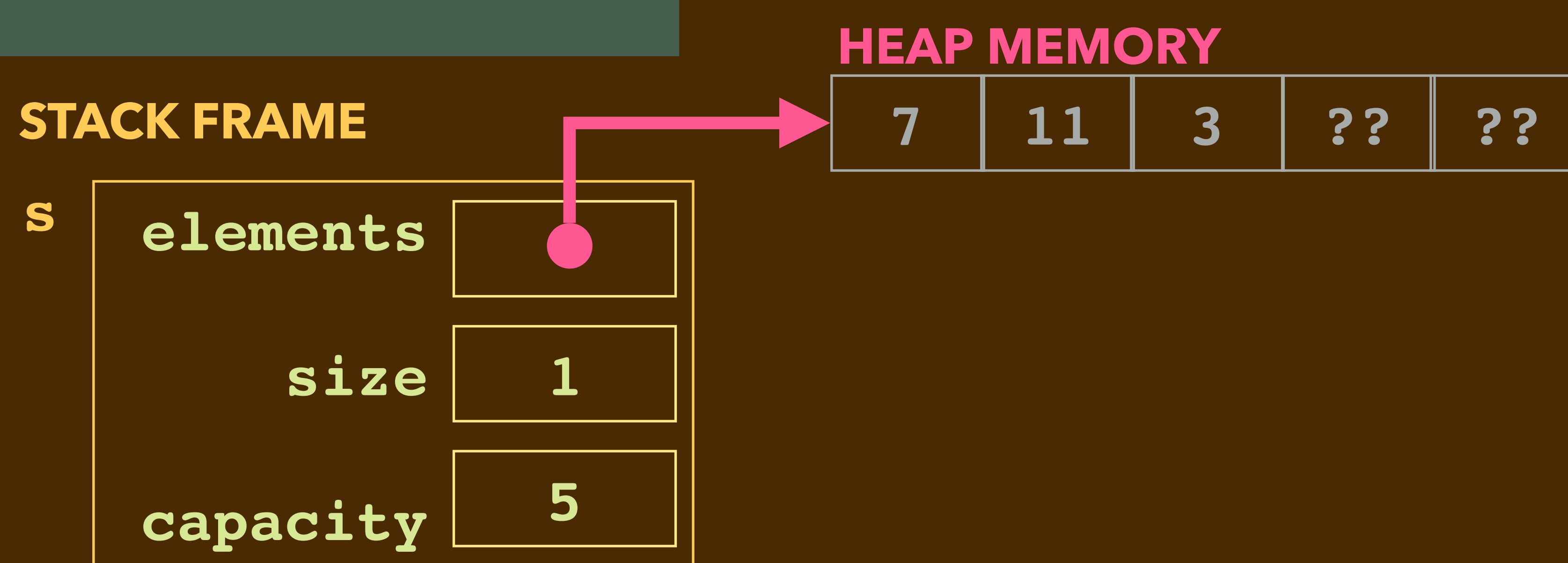
IMPLICIT CALL OF THE DESTRUCTOR

```
#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, which `delete`*s.*

CONSOLE

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



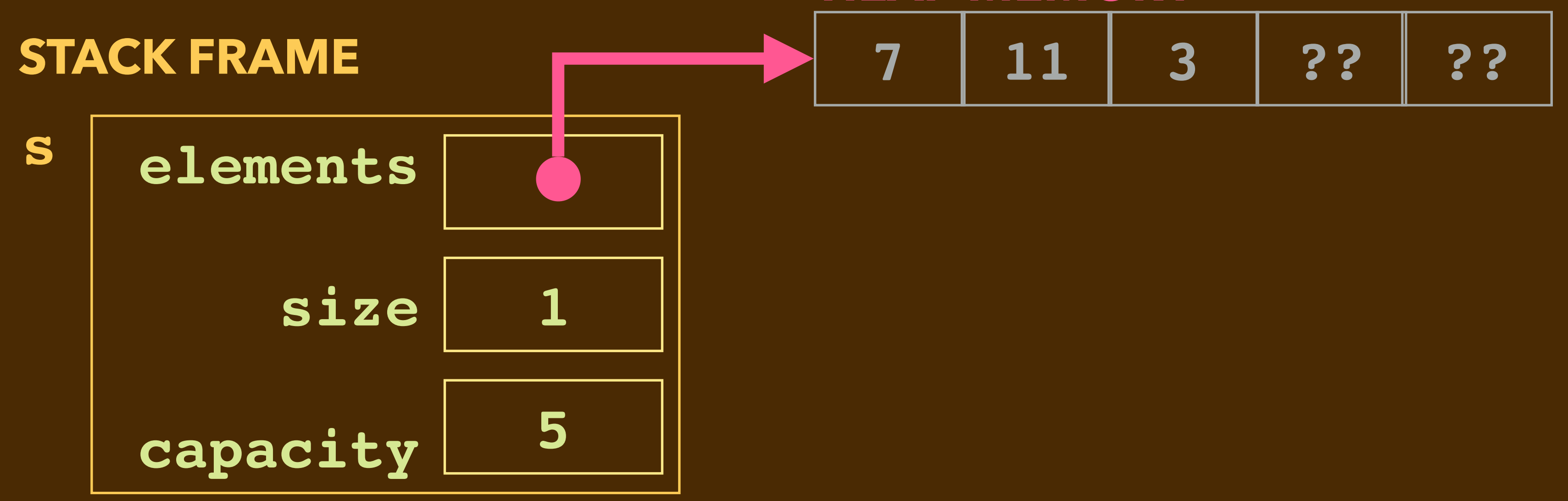
IMPLICIT CALL OF THE DESTRUCTOR

```
#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;
}
```

And the frame gets taken down.

CONSOLE

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



IMPLICIT CALL OF THE DESTRUCTOR

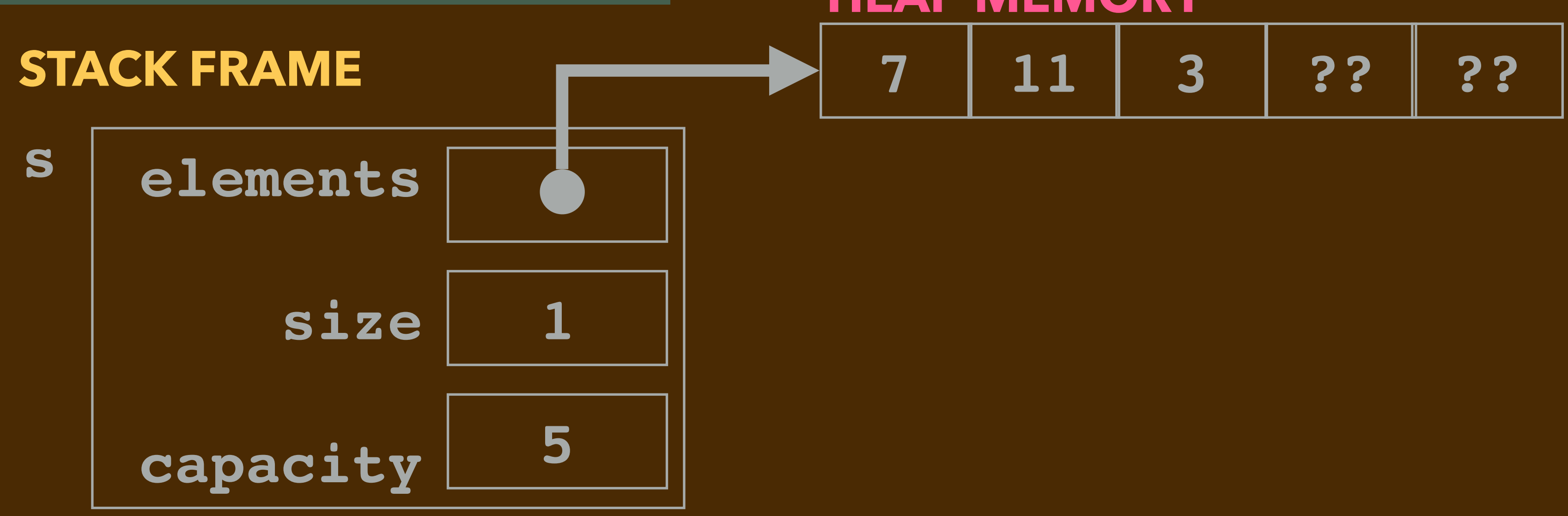
```
#include "Stck.hh"
#include <iostream>
int main(void) {
  Stck s {5};
  s.push(7);
  s.push(1);
  s.push(3);
  std::cout << s.pop() << std::endl;
  std::cout << s.pop() << std::endl;
  s.push(11);
  std::cout << s.pop() << std::endl;
}
```

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.

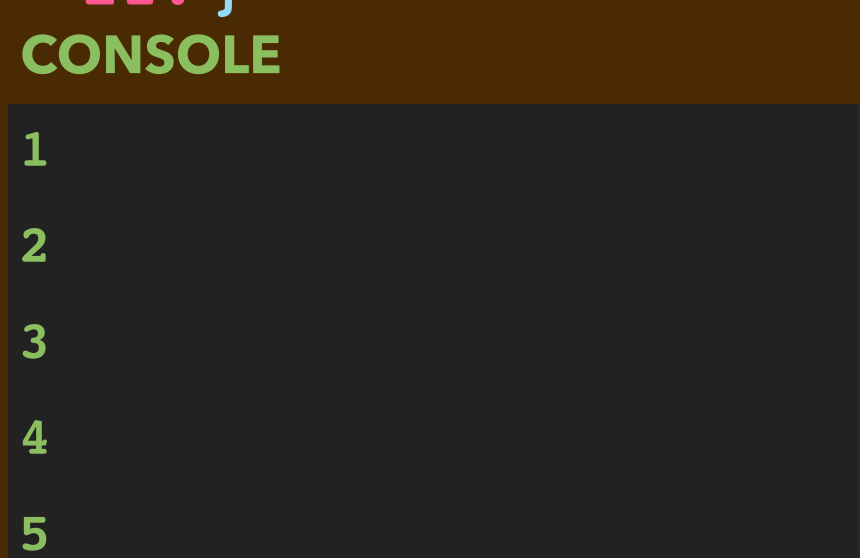
HEAP-ALLOCATED STACK

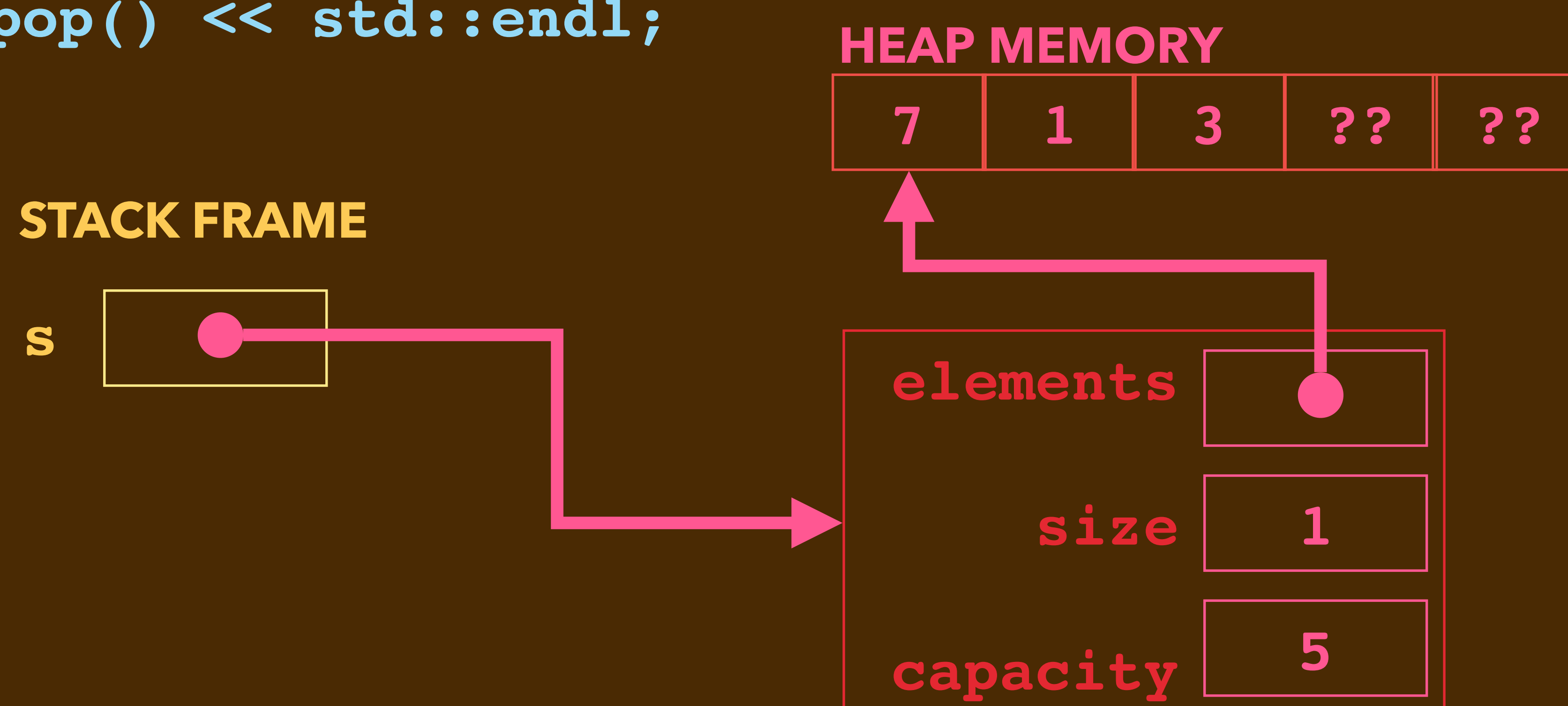
```
#include "Stck.hh"
#include <iostream>
int main(void) {
  Stck* s = new Stck {5};
  s->push(7);
  s->push(1);
  s->push(3);
  std::cout << s->pop() << std::endl;
  std::cout << s->pop() << std::endl;
  s->push(11);
  std::cout << s->pop() << std::endl;
  delete s;
}
```

- 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;
}
```



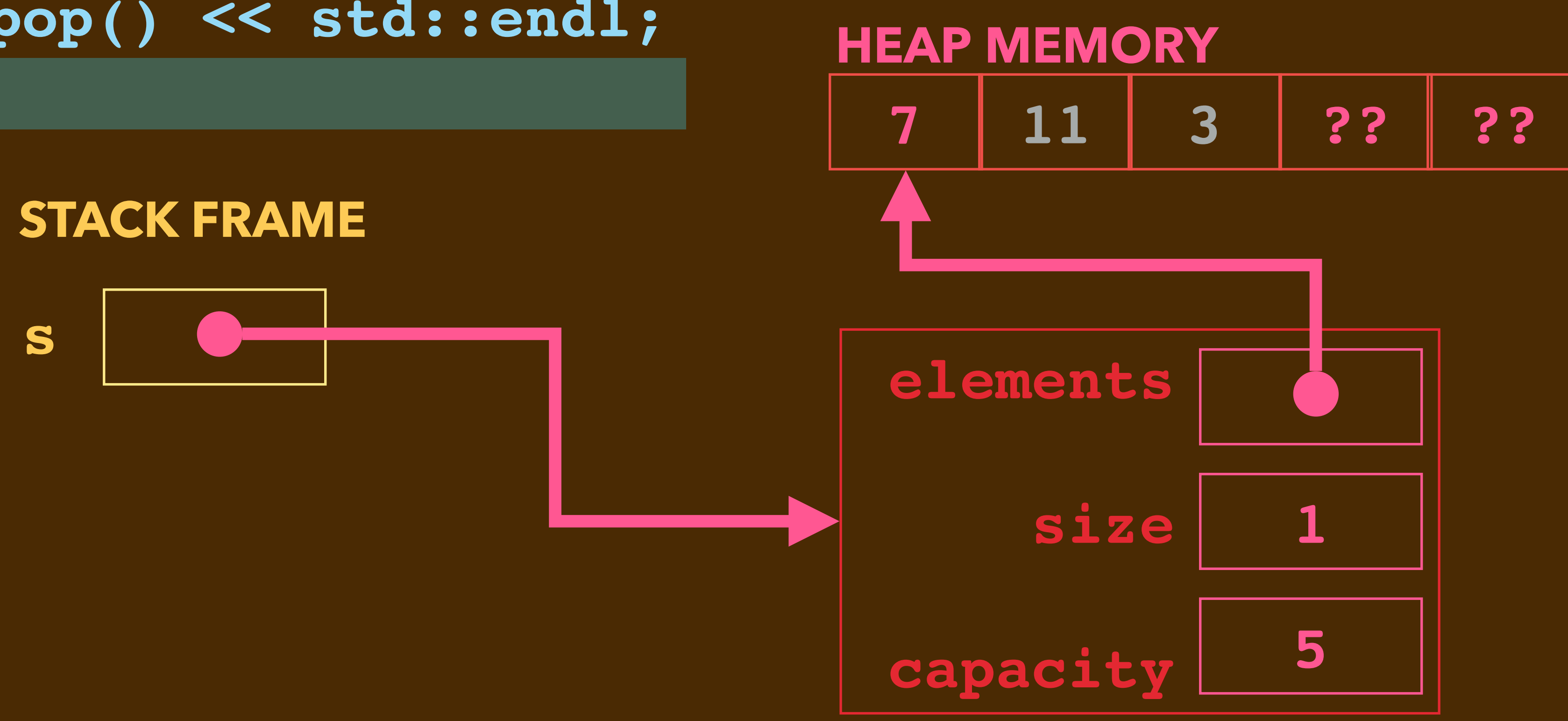


HEAP-ALLOCATED STACK ILLUSTRATED

```
#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

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



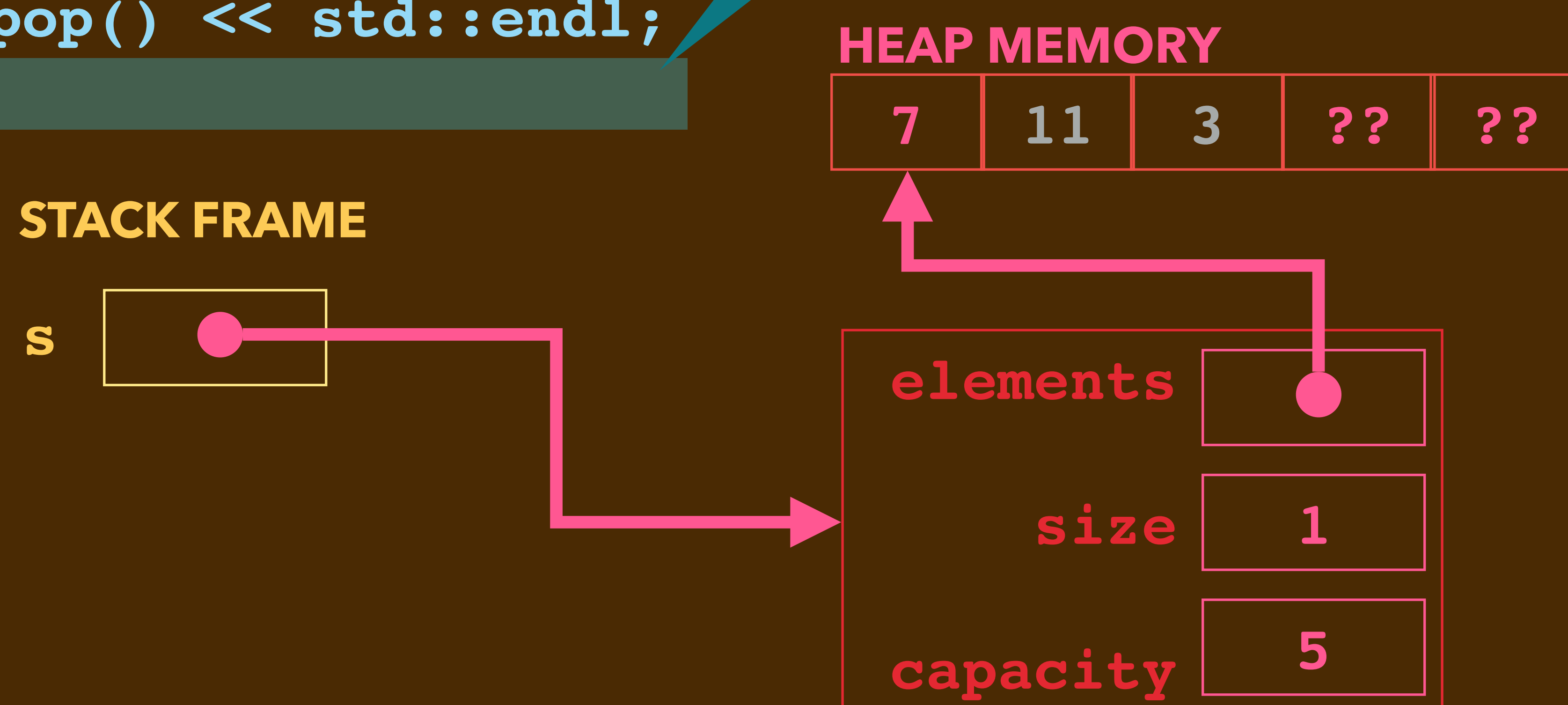
HEAP-ALLOCATED STACK ILLUSTRATED

```
#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 delete

CONSOLE

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

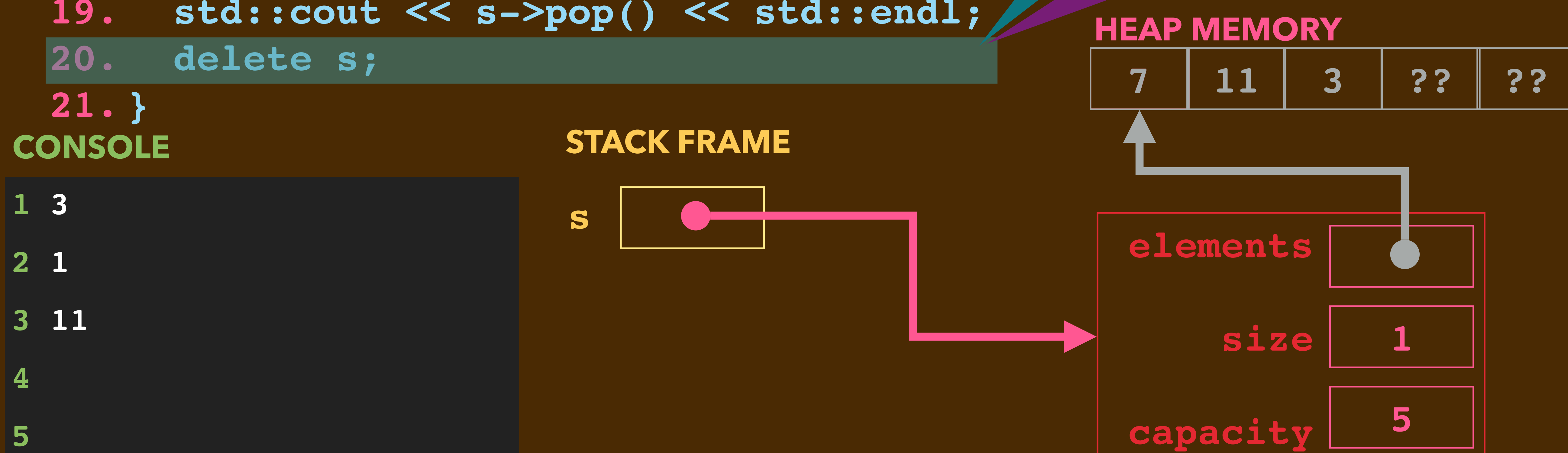


HEAP-ALLOCATED STACK ILLUSTRATED

```
#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.`

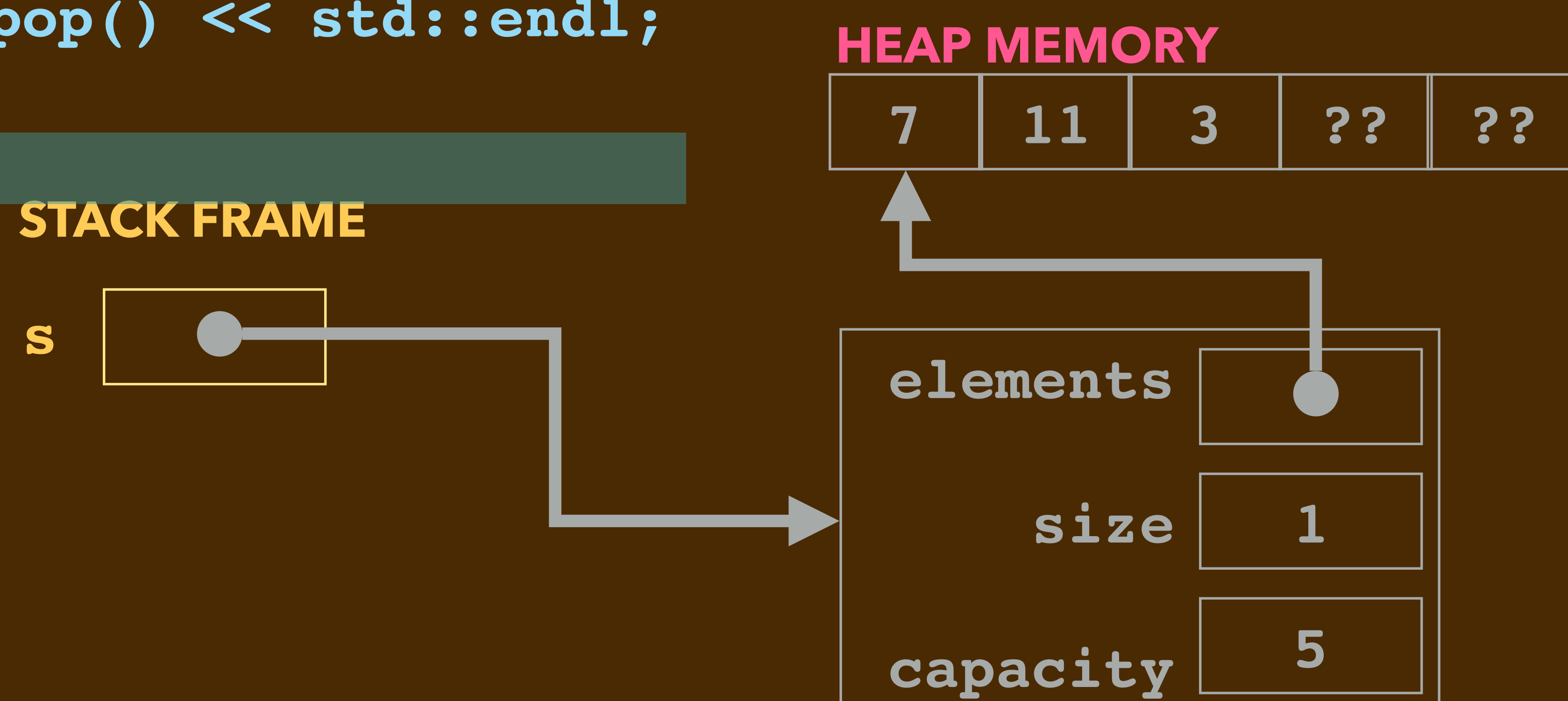


HEAP-ALLOCATED STACK ILLUSTRATED

```
#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

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



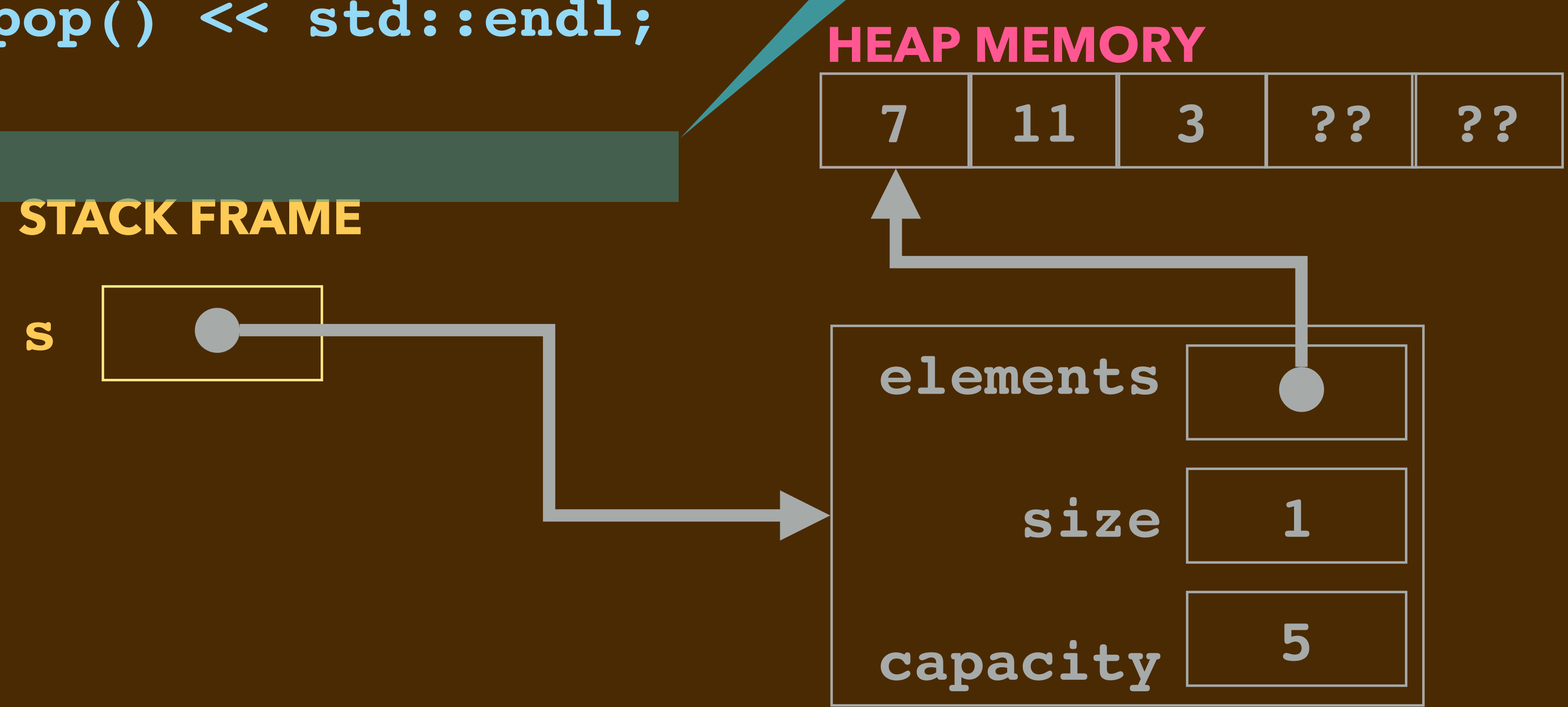
HEAP-ALLOCATED STACK ILLUSTRATED

```
#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
```



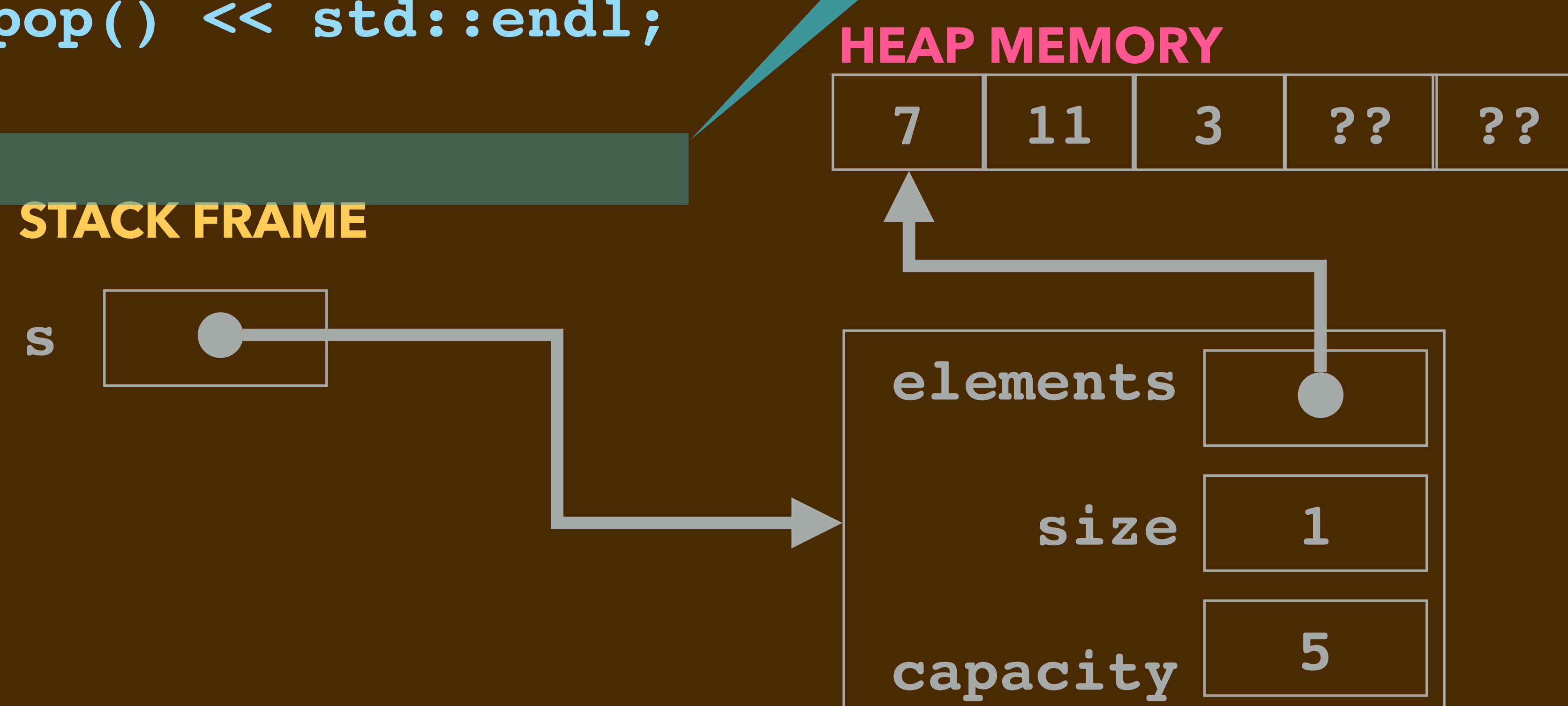
HEAP-ALLOCATED STACK ILLUSTRATED

```
#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

```
3
1
11
```



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

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

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

- 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;
}
```

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);
...
```

One way to prevent all this copying is to pass the address of the struct and have `get` take a pointer.

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 parameters i, x, *and* y *aliases of* count, a, *and* b.

- 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";
```

CONSOLE

```
10 17 42
11 42 17
```

- 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

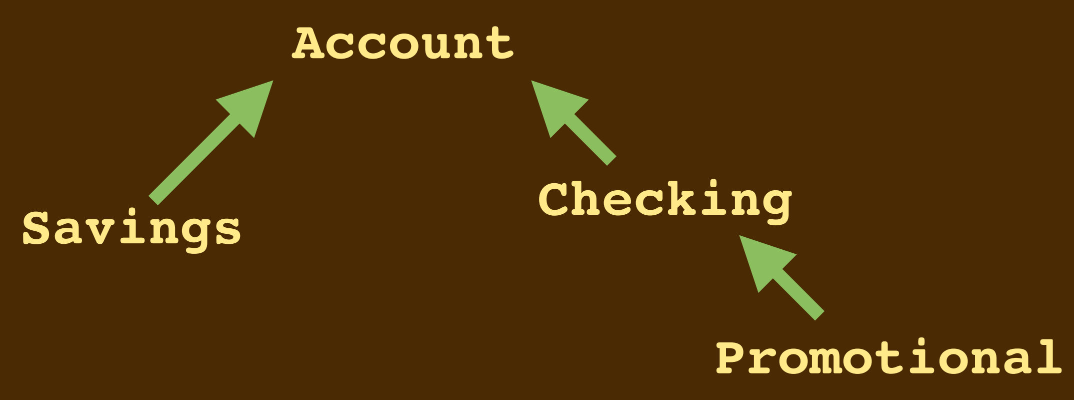
```
class Stck {

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);
};
```

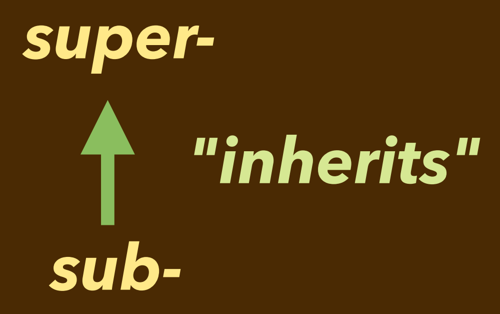
HMMM... LET'S WAIT TO DISCUSS THIS ANOTHER DAY

```
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);
};
```

INHERITANCE

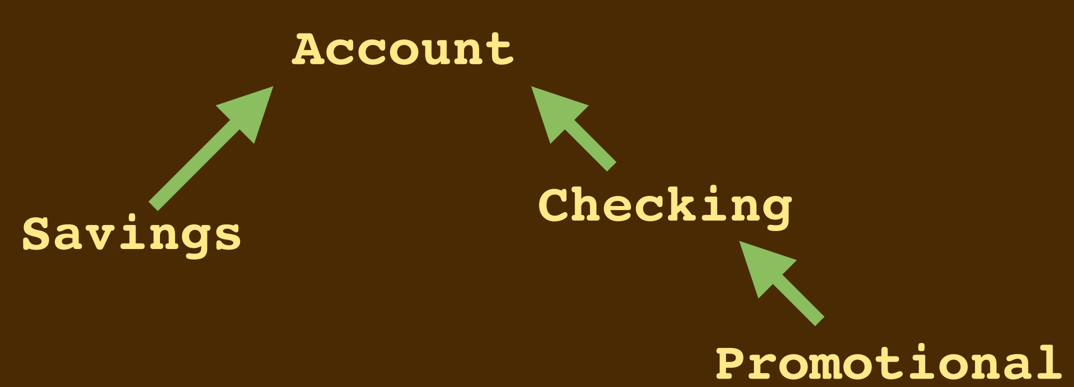
- **RECALL:** OO languages allow us to extend object classes:
  - ➡adding instance variables enhances what they can represent.
  - ➡adding methods enhances their behavior.
- The standard mechanism for this is ***subclassing***.
  - ➡ A subclass ***inherits*** the fields and behavior of its *superclass*.
  - ➡ The extensions make it more ***specialized***.
  - ➡ We can develop a *class hierarchy*.
- Example:

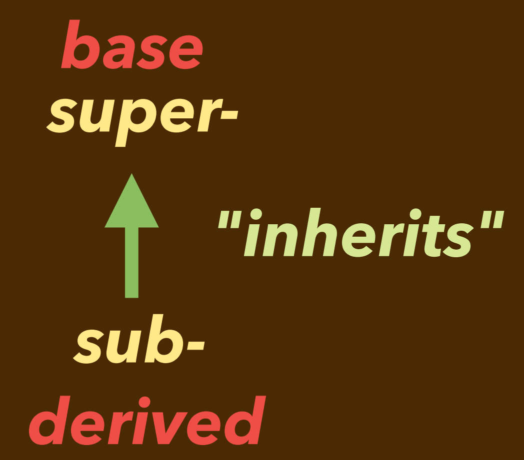




INHERITANCE

- **RECALL:** OO languages allow us to extend object classes:
  - ➡ adding instance variables enhances what they can represent.
  - ➡ adding methods enhances their behavior.
- The standard mechanism for this is ***subclassing***.
  - ➡ A subclass ***inherits*** the fields and behavior of its *superclass*.
  - ➡ The extensions make it more ***specialized***.
  - ➡ We can develop a *class hierarchy*.
- Example:





ACCOUNT CLASS

```
class Account {
private:
  static long gNextNumber; // used to generate account nos.
  // instance variables
  std::string name;          // description of the account
  long number;               // account no.
  double balance;            // money held
  double rate;               // monthly interest
public:
  ...
};
```

ACCOUNT CLASS

```
class Account {
private:
  static long gNextNumber;
  // instance variables
  ...
public:
  Account(std::string name, double amount, double interest);
  // getters
  double getBalance() const;
  std::string getName() const;
  long getNumber() const;
  double getRate() const;
  // methods
  void deposit(double amount);     // add money
  void gainInterest();             // each month
  double withdraw(double amount); // remove money
};
```

ACCOUNT CLASS IMPLEMENTATION (MISSING GETTERS)

```
Account::Account(std::string name, double amount, double
interest) : name {name},
            balance {amount},
            rate {interest},
            number {Account::gNextNumber++}
{ }

void Account::deposit(double amount) {
  balance += amount;
}
void Account::gainInterest() {
  deposit(rate * balance);
}
double Account::withdraw(double amount) {
  if (amount > balance) {
    amount = balance;
    balance = 0.0;
  } else {
    balance -= amount;
  }
  return amount;
}
```

SUBCLASSES OF ACCOUNT

- Savings accounts accrue 2% interest. They charge a penalty for withdrawal.

```
class Savings : public Account { ... }
```

- Checking accounts accrue 1% interest, but only if balance is above $1000.

```
class Checking : public Account { ... }
```

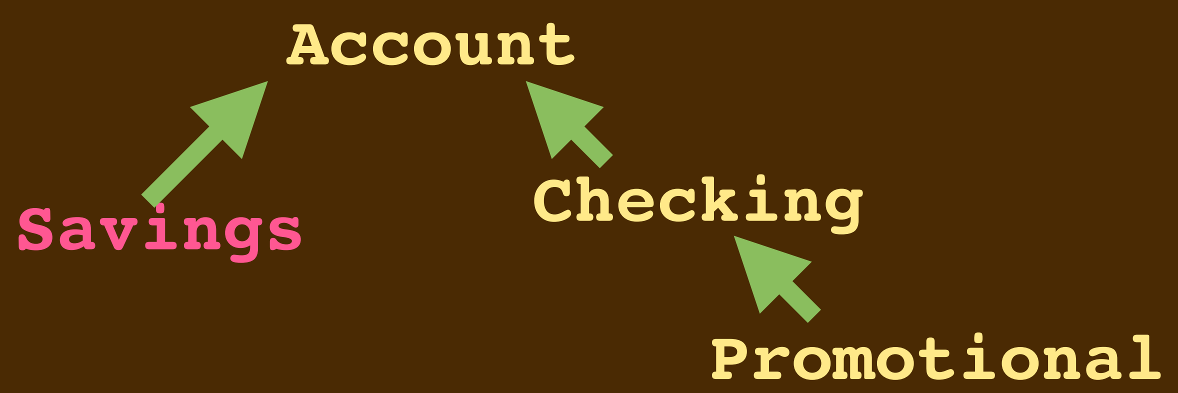
- Promotional checking accounts accrue 0.7% interest, but give you $100 to open the account. You must stay above $100 to earn that interest.

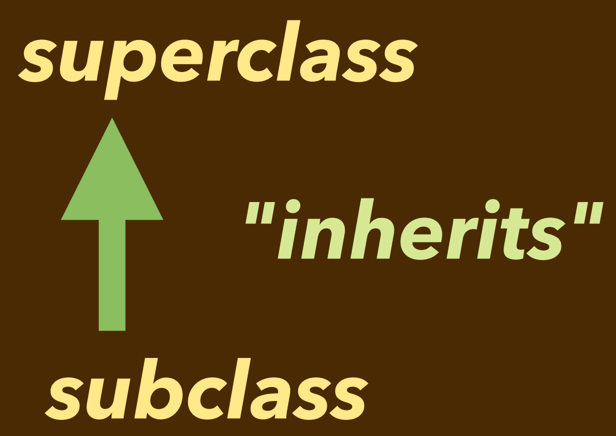
```
class Promotional : public Checking { ... }
```

- The keyword **`public`** means that
  - all public members are accessible as public members in the derived class,
  - all protected members are accessible as public members in the derived class,
  - private members are only accessible if a friend.

CLASS ACCOUNT AND ITS DERIVED CLASSES

- The full class hierarchy we'll flesh out...





- **Savings** accounts accrue 2% interest. They charge a penalty for withdrawal.

```
class Savings : public Account { ... }
```

- .

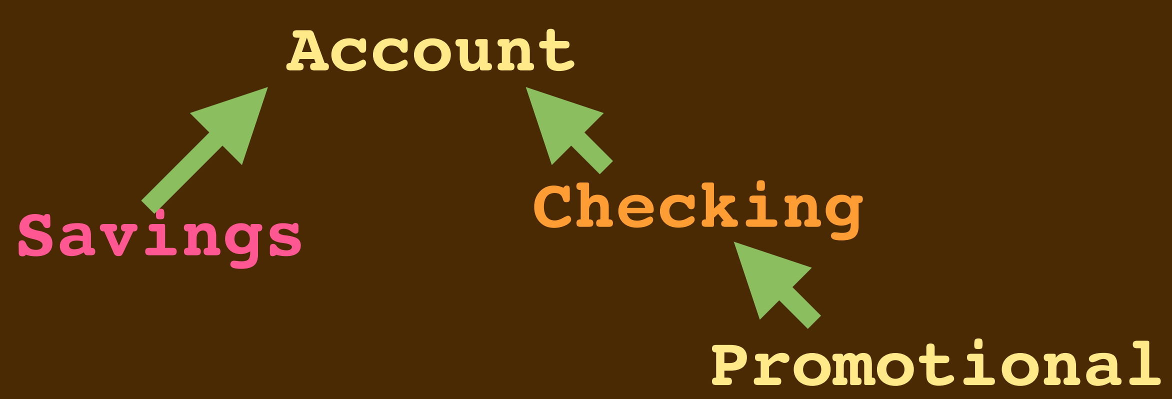
```
class Checking : public Account { ... }
```

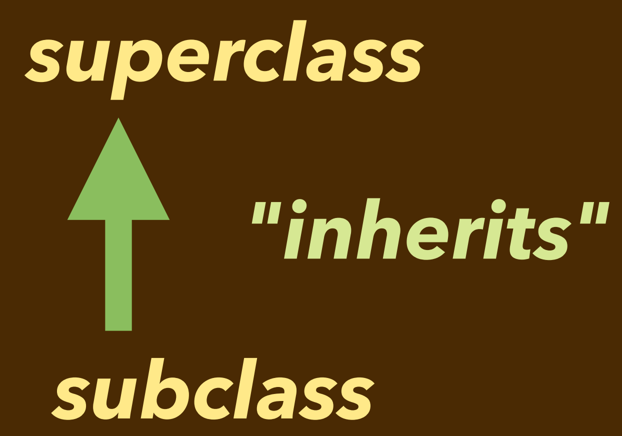
- .

```
class Promotional : public Checking { ... }
```

CLASS ACCOUNT AND ITS DERIVED CLASSES

- The full class hierarchy we'll flesh out...





- **Savings** accounts accrue 2% interest. They charge a penalty for withdrawal.

```
class Savings : public Account { ... }
```

- **Checking** accounts accrue 1% interest, but only if balance is above $1000.

```
class Checking : public Account { ... }
```

- .

```
class Promotional : public Checking { ... }
```

CLASS ACCOUNT AND ITS DERIVED CLASSES

- The full class hierarchy we'll flesh out...

- **Savings** accounts accrue 2% interest. They charge a penalty for withdrawal.

```
class Savings : public Account { ... }
```

- **Checking** accounts accrue 1% interest, but only if balance is above $1000.

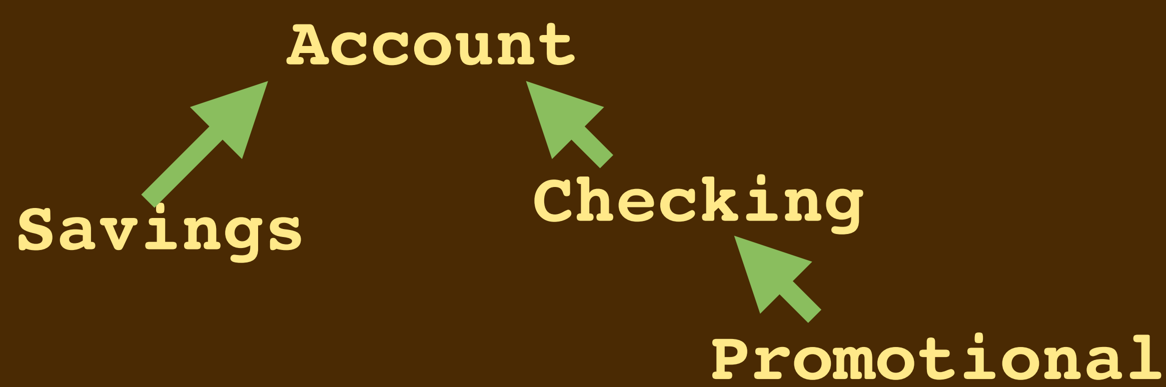
```
class Checking : public Account { ... }
```

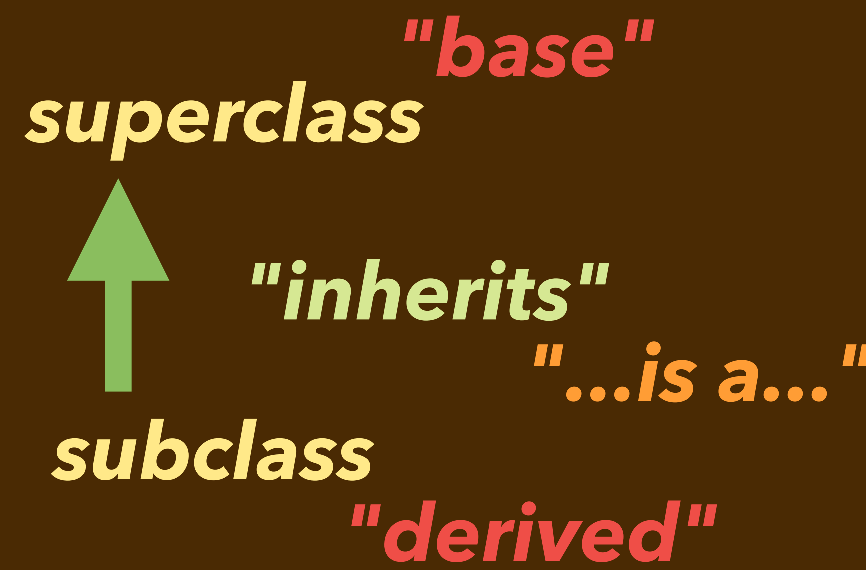
- **Promotional** checking accounts accrue 0.7% interest, but give you $100 to open the account. You must stay above $100 to earn that interest.

```
class Promotional : public Checking { ... }
```

CLASS ACCOUNT AND ITS DERIVED CLASSES

- The full class hierarchy we'll flesh out...





- **Savings** accounts accrue 2% interest. They charge a penalty for withdrawal.

```
class Savings : public Account { ... }
```

- **Checking** accounts accrue 1% interest, but only if balance is above $1000.

```
class Checking : public Account { ... }
```

- **Promotional** checking accounts accrue 0.7% interest, but give you $100 to open the account. You must stay above $100 to earn that interest.

```
class Promotional : public Checking { ... }
```

ACCOUNT CLASS, READIED FOR DERIVING

```
class Account {
private:
  static long gNextNumber;
protected:
  // instance variables
  ...
public:
  // methods
  Account(std::string name, double amount, double interest);
  virtual double getBalance() const;
  virtual std::string getName() const;
  virtual long getNumber() const;
  virtual double getRate() const;
  virtual void deposit(double amount);
  virtual void gainInterest();
  virtual double withdraw(double amount);
};
```

Virtual keyword indicates that the code of overriding methods in subclass will get called.

ACCOUNT CLASS, READIED FOR DERIVING

```
class Account {
private:
  static long gNextNumber;
protected:
  // instance variables
  std::string name;
  long number;
  double balance;
  double rate;
public:
  // methods
  ...
};
```

Not publicly accessible, but accessible to any derived class.

SUBCLASSES OF ACCOUNT

- Example of a subclass **`Savings`** deriving from a base **`Account`**:

```
class Savings : public Account { ... }
```

- The keyword **`public`** means that...

SUBCLASSES OF ACCOUNT

- Example of a subclass **`Savings`** deriving from a base **`Account`**:

```
class Savings : public Account { ... }
```

- The keyword **`public`** means that
  - all public members are accessible as public in the derived class,
  - all protected members are accessible as protected in the derived class,
  - private members are only accessible if that subclass is a **`friend`**.

EXTENSIONS AND OVERRIDES

```
class Savings : public Account {
protected:
  double penalty;  // Savings accounts have a withdrawal penalty.
public:
  Savings(std::string name, double amount);
  double withdraw(double amount);  // Charges a penalty.
};

class Checking : public Account {
protected:
  double level;
public:
  Checking(std::string name, double amount);
  void gainInterest();
};

class Promotional : public Checking {
public:
  Promotional(std::string name, double amount);
};
```

EXTENSIONS AND OVERRIDES

```
class Savings : public Account {
protected:
  double penalty;  // Savings accounts have a withdrawal penalty.
public:
  Savings(std::string name, double amount);
  double withdraw(double amount);  // Charges a penalty.
};

class Checking : public Account {
protected:
  double level;
public:
  Checking(std::string name, double amount);
  void gainInterest();
};

class Promotional : public Checking {
public:
  Promotional(std::string name, double amount);
};
```

EXTENSIONS AND OVERRIDES

```
class Savings : public Account {
protected:
  double penalty;
public:
  Savings(std::string name, double amount);
  double withdraw(double amount);
};

class Checking : public Account {
protected:
  double level; // Checking accounts gain interest above a level
public:
  Checking(std::string name, double amount);
  void gainInterest(); // Checks that level
};

class Promotional : public Checking {
public:
  Promotional(std::string name, double amount);
};
```

EXTENSIONS AND OVERRIDES

```
class Savings : public Account {
protected:
  double penalty;
public:
  Savings(std::string name, double amount);
  double withdraw(double amount);
};

class Checking : public Account {
protected:
  double level;
public:
  Checking(std::string name, double amount);
  void gainInterest();
};

class Promotional : public Checking {
public: // Promotional accounts are a special kind of checking
  Promotional(std::string name, double amount);     // account
};
```

SAVINGS ACCOUNT

Savings accounts accrue 2% interest. They charge a penalty for withdrawal.

- We add a **`penalty`** instance variable.

```
class Savings : public Account {
protected:
  double penalty;
public:
  Savings(std::string name, double amount);
  double withdraw(double amount);
};

Savings::Savings(std::string name, double amount) :
  Account {name,amount,0.02}, penalty {50.0}
{ }

double Savings::withdraw(double amount) {
  double howmuch = Account::withdraw(amount);
  Account::withdraw(penalty);
  return howmuch;
}
```

SAVINGS ACCOUNT

Savings accounts accrue 2% interest. They charge a penalty for withdrawal.

- We *override* the **withdraw** method to charge that penalty.

```
class Savings : public Account {
protected:
  double penalty;
public:
  Savings(std::string name, double amount);
  double withdraw(double amount);
};

Savings::Savings(std::string name, double amount) :
  Account {name,amount,0.02}, penalty {50.0}
{ }

double Savings::withdraw(double amount) {
  double howmuch = Account::withdraw(amount);
  Account::withdraw(penalty);
  return howmuch;
}
```

SAVINGS ACCOUNT

Savings accounts accrue 2% interest. They charge a penalty for withdrawal.

- We rely on **`Account`**'s implementation in several places.

```
class Savings : public Account {
protected:
  double penalty;
public:
  Savings(std::string name, double amount);
  double withdraw(double amount);
};

Savings::Savings(std::string name, double amount) :
  Account {name,amount,0.02}, penalty {50.0}
{ }

double Savings::withdraw(double amount) {
  double howmuch = Account::withdraw(amount);
  Account::withdraw(penalty);
  return howmuch;
}
```

CHECKING ACCOUNT

Checking accounts accrue 1% interest, but only if balance is above $1000.

- We add a **`level`** instance variable.

```
class Checking : public Account {
protected:
  double level;
public:
  Checking(std::string name, double amount);
  void gainInterest();
};

Checking::Checking(std::string name, double amount) :
  Account {name, amount, 0.01}, level {1000.0}
{ }

void Checking::gainInterest() {
  if (balance >= level) {
    Account::gainInterest();
  }
}
```

CHECKING ACCOUNT

Checking accounts accrue 1% interest, but only if balance is above $1000.

- We *override* the **`gainInterest`** method to check that level.

```
class Checking : public Account {
protected:
  double level;
public:
  Checking(std::string name, double amount);
  void gainInterest();
};

Checking::Checking(std::string name, double amount) :
  Account {name, amount, 0.01}, level {1000.0}
{ }

void Checking::gainInterest() {
  if (balance >= level) {
    Account::gainInterest();
  }
}
```

CHECKING ACCOUNT

Checking accounts accrue 1% interest, but only if balance is above $1000.

- We rely on **Account**'s implementation in several places.

```
class Checking : public Account {
protected:
  double level;
public:
  Checking(std::string name, double amount);
  void gainInterest();
};

Checking::Checking(std::string name, double amount) :
  Account {name, amount, 0.01}, level {1000.0}
{ }

void Checking::gainInterest() {
  if (balance >= level) {
    Account::gainInterest();
  }
}
```

PROMOTIONAL (CHECKING) ACCOUNT

Promotional accrues less interest, has an opening gift, has lower threshold.

- It derives from **`Checking`**. There are no extensions or overrides.

```
class Promotional : public Checking {
public:
  Promotional(std::string name, double amount);
};


Promotional::Promotional(std::string name, double amount) :
  Checking {name, amount + 100.0}
{
  rate = 0.07;
  level = 100.0;
}
```

VIRTUAL METHODS: DISPATCH ACCORDING TO CONTENTS

- Consider these two class definitions

```
class A {
    ...
    virtual void m(...); // yes virtual
    ...
}
class B : public A {
    ...
    void m(...);
    ...
}
```

- Consider this client code

```
A *b = new B();
b->m(x);
```

- Since **`m`** is marked virtual, the code for **`B::m`** runs like we'd normally expect.
  -

VIRTUAL METHODS: DISPATCH ACCORDING TO CONTENTS

- Consider these two class definitions

```
class A {
    ...
    virtual void m(...); // yes virtual
    ...
}
class B : public A {
    ...
    void m(...);
    ...
}
```

- Consider this client code

```
A *b = new B();
b->m(x);
```

- Since **m** is marked virtual, the code for **B::m** runs like we'd normally expect.
    - This is sometimes called "***dynamic dispatch***" of the "message" **m**.

VIRTUAL METHODS: DISPATCH ACCORDING TO CONTENTS

- Consider these two class definitions

```
class A {
    ...
    virtual void m(...); // yes virtual
    ...
}
class B : public A {
    ...
    void m(...);
    ...
}
```

- Consider this client code

```
A *b = new B();
b->m(x);
```

- Since **m** is marked virtual, the code for **B::m** runs like we'd normally expect.
  - Code run for **m** is determined by the **contents at b**, i.e. *at run time*.

VIRTUAL METHODS: DISPATCH ACCORDING TO CONTENTS

- Consider these two class definitions

```
class A {
    ...
    virtual void m(...); // yes virtual
    ...
}
class B : public A {
    ...
    void m(...);
    ...
}
```

- Consider this client code

```
A *b = new B();
b->m(x);
```

- Since **m** is marked virtual, the code for **B::m** runs like we'd normally expect. *dynamic!!*
    - Code run for **m** is determined by the **contents at b**, i.e. *at run time*.

NON-VIRTUAL METHODS: DISPATCH ACCORDING TO TYPE

- Consider these two class definitions

```
class A {
    ...
    void m(...); // NOTE: not virtual!!!
    ...
}
class B : public A {
    ...
    void m(...);
    ...
}
```

- Consider this client code

```
A *b = new B();
b->m(x);
```

- Since **m** is not marked virtual, the code for **A::m** runs instead.
  -

NON-VIRTUAL METHODS: DISPATCH ACCORDING TO TYPE

- Consider these two class definitions

```
class A {
    ...
    void m(...); // NOTE: not virtual!!!
    ...
}
class B : public A {
    ...
    void m(...);
    ...
}
```

- Consider this client code

```
A *b = new B();
b->m(x); //
```

- Since **m** is not marked virtual, the code for **A::m** runs instead***!!!!!!!***
    - This is sometimes called "***static dispatch***" of the "message" **m**.

NON-VIRTUAL METHODS: DISPATCH ACCORDING TO TYPE

- Consider these two class definitions

```
class A {
    ...
    void m(...); // NOTE: not virtual!!!
    ...
}
class B : public A {
    ...
    void m(...);
    ...
}
```

- Consider this client code

```
A *a = new B();
a->m(x);
```

- Since **m** is not marked virtual, the code for **A::m** runs instead!!!!!!!
  - Code run for **m** is determined by the **type of b**, i.e. *at compile time*.

NON-VIRTUAL METHODS: DISPATCH ACCORDING TO TYPE

- Consider these two class definitions

```
class A {
    ...
    void m(...); // NOTE: not virtual!!!
    ...
}
class B : public A {
    ...
    void m(...);
    ...
}
```

- Consider this client code

```
A *a = new B();
a->m(x);
```

- Since **m** is not marked virtual, the code for **A::m** runs instead!!!!!!! **static.**
  - Code run for **m** is determined by the **type of b**, i.e. *at compile time*.

WHY YOU WANT DYNAMIC DISPATCH

- Imagine We have the following hierarchy:

```
class Shape { virtual void draw(); ... };
class Oval : public Shape { void draw(); ... };
class Rectangle : public Shape { void draw(); ... };
```

- Consider this client code that has a linked list **`shapes`**:

```
ShapeNode* current = shapes->first;
while (current != nullptr) {
  current->shape->draw();
}
```

-

WHY YOU WANT DYNAMIC DISPATCH

- Imagine We have the following hierarchy:

```
class Shape { virtual void draw(); ... };
class Oval : public Shape { void draw(); ... };
class Rectangle : public Shape { void draw(); ... };
```

- Consider this client code that has a linked list of **`shapes`**:

```
ShapeNode* current = shapes->first;
while (current != nullptr) {
  current->shape->draw();
}
```

- In the above code, **`current->shape`** is of type **`Shape*`**.
-

WHY YOU WANT DYNAMIC DISPATCH

- Imagine We have the following hierarchy:

```
class Shape { virtual void draw(); ... };
class Oval : public Shape { void draw(); ... };
class Rectangle : public Shape { void draw(); ... };
```

- Consider this client code that has a linked list of **`shapes`**:

```
ShapeNode* current = shapes->first;
while (current != nullptr) {
  current->shape->draw();
}
```

- In the above code, **`current->shape`** is of type **`Shape*`**.
- Because the **`draw`** method is **`virtual`**, dynamic dispatch is used.

WHY YOU WANT DYNAMIC DISPATCH

- Imagine We have the following hierarchy:

```
class Shape { virtual void draw(); ... };
class Oval : public Shape { void draw(); ... };
class Rectangle : public Shape { void draw(); ... };
```

- Consider this client code that has a linked list of **`shapes`**:

```
ShapeNode* current = shapes->first;
while (current != nullptr) {
  current->shape->draw();
}
```

- In the above code, **`current->shape`** is of type **`Shape*`**.
- Because the **`draw`** method is **`virtual`**, dynamic dispatch is used.
  - When the list node points to an **`Oval`** instance, **`Oval::draw`** is called.

WHY YOU WANT DYNAMIC DISPATCH

- Imagine We have the following hierarchy:

```
class Shape { virtual void draw(); ... };
class Oval : public Shape { void draw(); ... };
class Rectangle : public Shape { void draw(); ... };
```

- Consider this client code that has a linked list of **`shapes`**:

```
ShapeNode* current = shapes->first;
while (current != nullptr) {
  current->shape->draw();
}
```

- In the above code, **`current->shape`** is of type **`Shape*`**.
- Because the **`draw`** method is **`virtual`**, dynamic dispatch is used.
    - When the list node points to an **`Oval`** instance, **`Oval::draw`** is called.
    - When it points to a **`Rectangle`**, **`Rectangle::draw`** is called.

ABSTRACT CLASSES

- Note that the **`Account`** class probably shouldn't have an instance.
  - Nonetheless, it does define a few methods useful to subclass instances:
    - ➡The **`deposit`** and **`withdraw`** methods as defined in **`Account`** provide a default behavior that subclasses may use, or override.

- Classes not meant to be instantiated are called ***abstract***.

"PURELY VIRTUAL" METHODS IN AN ABSTRACT BASE

- Can't always provide a "default" method behavior in an abstract base...
- In C++ we can designate methods as "purely virtual" with a value of 0:

```
class A {
   ...
   virtual T m(T1 v1, T2 v2, ...) = 0;
   ...
};

class B : public A {
   ...
   T m(T1 v1, T2 v2, ...) { ... /* actual behavior on B */ }
   ...
};
```

➡ Method **`m`** must be defined by classes that derive from abstract **`A`**.

"PURELY VIRTUAL" METHODS IN AN ABSTRACT BASE

- We can't always provide a "default" behavior in the base abstract class.
- In C++ we can designate methods as "*purely virtual*" with a value of 0:

```
class A {
   ...
   virtual T m(T1 v1, T2 v2, ...) = 0;
   ...
};

class B : public A {
   ...
   T m(T1 v1, T2 v2, ...) { ... /* actual behavior on B */ }
   ...
};
```

  - ➡ Method **`m`** must be defined by classes that derive from abstract **`A`**.

"PURELY VIRTUAL" METHODS IN AN ABSTRACT BASE

- We can't always provide a "default" behavior in the base abstract class.
- In C++ we can designate methods as "purely virtual" with a value of 0:

```
class A {
   ...
   virtual T m(T1 v1, T2 v2, ...) = 0;
   ...
};

class B : public A {
   ...
   T m(T1 v1, T2 v2, ...) { ... /* actual behavior on B */ }
   ...
};
```

➡Method **m** must be defined by classes that derive from abstract **A**.

EXAMPLE: SHAPE HIERARCHY

```
class Shape {
public:
  virtual double perimeter(void) const = 0;
  virtual double area(void) const = 0;
  virtual void print(void) const = 0;
  virtual double getHeight(void) const = 0;
  virtual double getWidth(void) const = 0;
  Rectangle bounds(void);
};
```

CIRCLE SUBCLASS DERIVED FROM SHAPE

```
class Circle : public Shape {
private:
  double radius;
public:
  Circle(double r) : radius(r) { }
  double perimeter(void) { return 2.0 * M_PI * radius; }
  double area(void) { return M_PI * radius * radius; }
  void print(void); // This one's many lines long.
  double getHeight(void) { return 2.0 * radius; }
  double getWidth(void) { return 2.0 * radius; }
};

void Circle::print(void) const {
  cout << "A circle with radius " << radius << ":\n" << endl;
  int w = static_cast<int>(ceil(getWidth()));
  if (w == 1) {
    std::cout << "+" << std::endl;
    return;
  }
  ...
```

CIRCLE SUBCLASS DERIVED FROM SHAPE

```
class Circle : public Shape {
private:
  double radius;
public:
  Circle(double r) : radius(r) { }
  double perimeter(void) { return 2.0 * M_PI * radius; }
  double area(void) { return M_PI * radius * radius; }
  void print(void); // This one's many lines long.
  double getHeight(void) { return 2.0 * radius; }
  double getWidth(void) { return 2.0 * radius; }
};

void Circle::print(void) const {
  cout << "A circle with radius " << radius << ":\n" << endl;
  int w = static_cast<int>(ceil(getWidth()));
  if (w == 1) {
    std::cout << "+" << std::endl;
    return;
  }
  ...
```

CIRCLE SUBCLASS DERIVED FROM SHAPE

```
class Circle : public Shape {
private:
  double radius;
public:
  Circle(double r) : radius(r) { }
  double perimeter(void) { return 2.0 * M_PI * radius; }
  double area(void) { return M_PI * radius * radius; }
  void print(void); // This one's many lines long.
  double getHeight(void) { return 2.0 * radius; }
  double getWidth(void) { return 2.0 * radius; }
};

void Circle::print(void) const {
  cout << "A circle with radius " << radius << ":\n" << endl;
  int w = static_cast<int>(ceil(getWidth()));
  if (w == 1) {
    std::cout << "+" << std::endl;
    return;
  }
  ...
```

CIRCLE SUBCLASS DERIVED FROM SHAPE

```
class Circle : public Shape {
private:
  double radius;
public:
  Circle(double r) : radius(r) { }
  double perimeter(void) { return 2.0 * M_PI * radius; }
  double area(void) { return M_PI * radius * radius; }
  void print(void); // This one's many lines long.
  double getHeight(void) { return 2.0 * radius; }
  double getWidth(void) { return 2.0 * radius; }
};

void Circle::print(void) const {
  cout << "A circle with radius " << radius << ":\n" << endl;
  int w = static_cast<int>(ceil(getWidth()));
  if (w == 1) {
    std::cout << "+" << std::endl;
    return;
  }
  ...
```

RECTANGLE SUBCLASS DERIVED FROM SHAPE

```
class Rectangle : public Shape {
private:
  double width;
  double height;
  void depict(void);
public:
  Rectangle(double w,double h) : width(w), height(h) { }
  double perimeter(void) { return 2.0 * (width + height); }
  double area(void) { return width * height; }
  void print(void);
  double getHeight(void) { return height; }
  double getWidth(void) { return width; }
  friend class Square;
};
```

RECTANGLE SUBCLASS DERIVED FROM SHAPE

```
class Rectangle : public Shape {
private:
  double width;
  double height;
  void depict(void);
public:
  Rectangle(double w,double h) : width(w), height(h) { }
  double perimeter(void) { return 2.0 * (width + height); }
  double area(void) { return width * height; }
  void print(void);
  double getHeight(void) { return height; }
  double getWidth(void) { return width; }
  friend class Square;
};
```

RECTANGLE SUBCLASS DERIVED FROM SHAPE

```
class Rectangle : public Shape {
private:
  double width;
  double height;
  void depict(void);
public:
  Rectangle(double w,double h) : width(w), height(h) { }
  double perimeter(void) { return 2.0 * (width + height); }
  double area(void) { return width * height; }
  void print(void);
  double getHeight(void) { return height; }
  double getWidth(void) { return width; }
  friend class Square;
};
```

RECTANGLE SUBCLASS DERIVED FROM SHAPE

```
class Rectangle : public Shape {
private:
  double width;
  double height;
  void depict(void) const;
public:
  Rectangle(double w,double h) : width(w), height(h) { }
  double perimeter(void) { return 2.0 * (width + height); }
  double area(void) { return width * height; }
  void print(void);
  double getHeight(void) { return height; }
  double getWidth(void) { return width; }
  friend class Square;
};
void Rectangle::print(void) const {
  std::cout << "Here is a " << width << "x" << height;
  std::cout << " rectangle:\n" << std::endl;
  depict();
}
```

RECTANGLE SUBCLASS DERIVED FROM SHAPE

```
class Rectangle : public Shape {
private:
  double width;
  double height;
  void depict(void) const;
public:
  Rectangle(double w,double h) : width(w), height(h) { }
  double perimeter(void) { return 2.0 * (width + height); }
  double area(void) { return width * height; }
  void print(void);
  double getHeight(void) { return height; }
  double getWidth(void) { return width; }
  friend class Square;
};
void Rectangle::print(void) const {
  std::cout << "Here is a " << width << "x" << height;
  std::cout << " rectangle:\n" << std::endl;
  depict();
}
```

SQUARE SUBCLASS DERIVED FROM RECTANGLE

```
class Rectangle : public Shape {
private:
  void depict(void);
public:
  ...
  friend Square;
}

class Square : public Rectangle {
public:
  Square(double s) : Rectangle {s, s} { }
  void print(void);
};

void Square::print(void) const {
  std::cout << "Here is a " << getWidth() << "x" << getHeight();
  std::cout << " square:\n" << std::endl;
  Rectangle::depict();
}
```

SQUARE SUBCLASS DERIVED FROM RECTANGLE

```
class Rectangle : public Shape {
private:
  void depict(void);
public:
  ...
  friend Square;
}

class Square : public Rectangle {
public:
  Square(double s) : Rectangle {s, s} { }
  void print(void);
};

void Square::print(void) const {
  std::cout << "Here is a " << getWidth() << "x" << getHeight();
  std::cout << " square:\n" << std::endl;
  Rectangle::depict();
}
```

SQUARE SUBCLASS DERIVED FROM RECTANGLE

```
class Rectangle : public Shape {
private:
  void depict(void);
public:
  ...
  friend Square;
}

class Square : public Rectangle {
public:
  Square(double s) : Rectangle {s, s} { }
  void print(void);
};

void Square::print(void) const {
  std::cout << "Here is a " << getWidth() << "x" << getHeight();
  std::cout << " square:\n" << std::endl;
  Rectangle::depict();
}
```

SHAPE PROGRAM OUTPUT

```
Here is a circle with radius 5:

  ++++++
 ++++++++
++++++++++
++++++++++
++++++++++
++++++++++
++++++++++
++++++++++
 ++++++++
  ++++++

Here is a 7x3 rectangle:

+++++++
+++++++
+++++++

Here is a 1x1 square:

+
```

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