ECE 312 (SP26):

C++ Introduction

Neil Zhao

neil.zhao@utexas.edu

Recap: The Lamp Class/Struct

```
typedef struct {
    const char *color;
    unsigned brightness;  // ranges from 0 to 10
    bool is_on;

    // Invariant:
    // - when is_on == false, brightness == 0
    // - when is_on == true,  1 <= brightness <= 10
} LampInC;

void LampTurnOn(LampInC *lamp);
void LampTurnOff(LampInC *lamp);
void LampPrint(const LampInC *lamp);
void LampSetBrightness(LampInC *lamp, unsigned brightness);
```

Recap: The Lamp Class/Struct

```
struct Lamp {
    // Data members
    const char *color;
    unsigned brightness;  // ranges from 0 to 10
    bool is_on;

    void turnOn() { … }
    void turnOff() { … }
    void print() { … }
    void setBrightness(unsigned brightness) {
      …
      this->brightness = brightness;
      …
    }
};
```

Recap: The Lamp Class/Struct

```
struct Lamp {
private:
    // Data members
    const char *color;
    unsigned brightness;  // ranges from 0 to 10
    bool is_on;

public:
    Lamp(const char *c) : color(c), brightness(0), is_on(false) {}
    void turnOn() { … }
    void turnOff() { … }
    void print() { … }
    void setBrightness(unsigned brightness) { … }
};
```

Recap: References

```
struct Lamp { … };

void setBrightness(Lamp &lamp, unsigned brightness) {
  lamp.setBrightness(brightness);
}

int main() {
  Lamp lamp("blue"); // lamp is off
  Lamp &ref = lamp;

  ref.turnOn(); // lamp now is on

  setBrightness(lamp, 5); // the lamp’s brightness may have been changed
}
```

- Cannot be null
- Must be initialized when created
- Cannot be modified to refer to a different object

New Feature: Function Overloading

```
void print_it(int x) {
  printf("%d\n", x);
}

void print_it(const char *s) {
  if (s) {
    printf("%s\n", s);
  }
}
```

```
print_it("Hello");

print_it(528491);
```

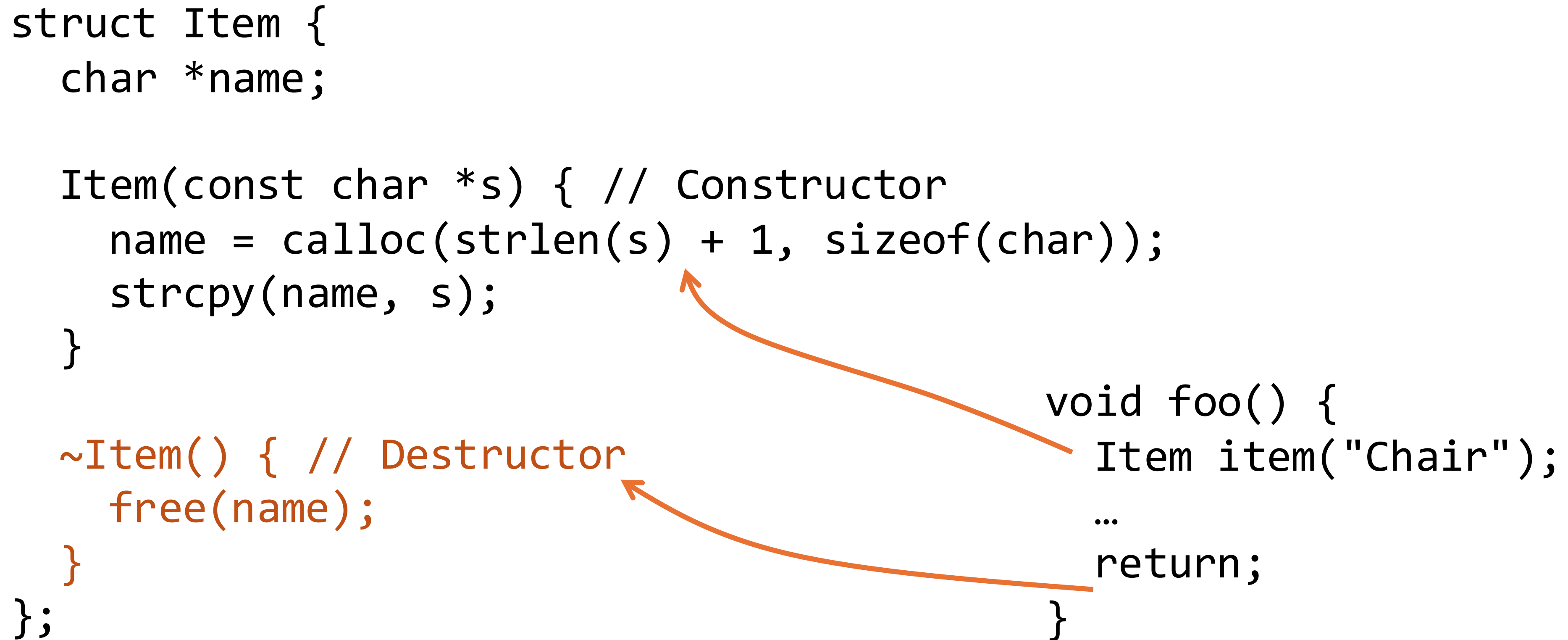
Functions with the same name but different argument lists
=> Different functions
=> Compiler picks the right one based on the argument type

Operator Overloading

```
struct Complex {
  double real, im;

  Complex(double real, double imaginary) : real(real), im(im) {}

  void print() { printf("%f+%fi\n", real, im); }
  Complex operator+(const Complex &other) {
    return Complex(real + other.real, im + other.im);
  }
};

int main() {
  Complex c1{1.0, 0.0}, c2{0.0, 1.0};
  // Complex c3 = c1 + c2;
}
```

Operator overloading allows us to define the meaning of an operator for a class

Constructor and Destructor



Constructor: invoked when the object is created

Destructor: invoked when the object's lifetime ends

new and delete

```
struct Item {
  char *name;

  Item(const char *s) { // Constructor
    name = calloc(strlen(s) + 1, sizeof(char));
    strcpy(name, s);
  }

  ~Item() { // Destructor
    free(name);
  }
};
```

Allocates memory and calls the constructor

```
void foo() {
  Item *item = new Item("Chair");
  …
  delete item;
}
```

Calls the destructor and frees the memory that **item** occupies

new and delete

```
void foo() {
  Item *item = new Item("Chair");
  delete item;

  // Allocating a double array with 100 elements
  double *arr = new double[100];
  delete[] arr;
}
```

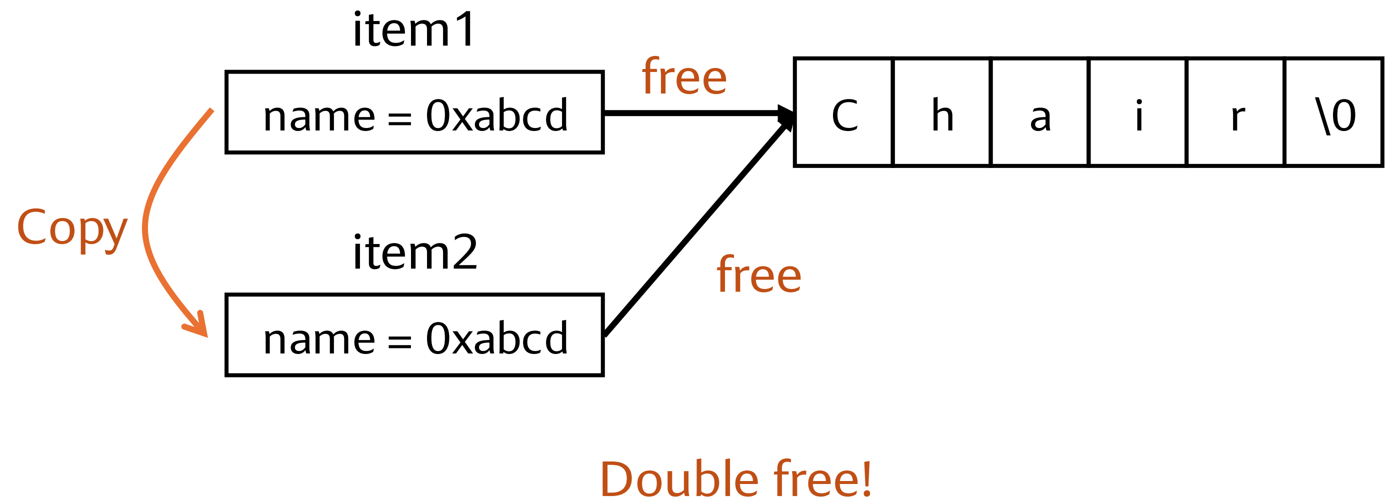
new/delete: single object
new[]/delete[]: object arrays

Caveats:

- new -> delete, new[] -> delete[], don't mix them
- Don't mix new/delete with malloc/calloc/free
    - malloc/calloc only allocates the memory and doesn't call the constructor

Shallow Copy

```
struct Item {
  char *name;
  …
};

void foo() {
  Item item1("Chair"), item2;
  item2 = item1;
}
```



Deep Copy (Desired Behavior in This Example)

```
struct Item {
  char *name;
  …
};

void foo() {
  Item item1("Chair"), item2;
  item2 = item1;
}
```