



# CS 11 C++ track: lecture 2

---

- Today
  - basic class design
    - constructors
    - `new`, `delete`, `delete[]`
    - copy constructors
    - destructors
  - using `assert()`
  - `bool` type



# Class design

---

- Each class should have:
  - constructor(s)
  - copy constructor
  - assignment operator (not this week)
  - destructor
- If don't define, compiler creates for you
  - don't rely on this!



# Example class (bogus.hh)

---

```
class bogus {  
    private:  
        int data;  
    public:  
        bogus() { data = 0; }  
        bogus(int i) { data = i; }  
        void set_data(int i) { data = i; }  
        int get_data() { return data; }  
}; // note semicolon at end
```



# Using the example class

---

```
bogus b1;           // calls no-arg constructor
bogus b1a();       // WRONG!
bogus b2(10);      // calls 1-arg constructor
cout << "b1: " << b1.get_data() << endl;
cout << "b2: " << b2.get_data() << endl;
```



# Dynamic memory allocation (1)

---

- Class may need to allocate memory on the fly
- in C, use `malloc()` to alloc, `free()` to free
- in C++, use `new` to alloc, `delete` to free

```
bogus *b1 = new bogus;  
cout << "b1: " << b1->get_data() << endl;  
delete b1;
```

- `new` and `delete` are keywords, not functions
- newly alloc'ed objects live until `deleted`



## Dynamic memory allocation (2)

---

- Can allocate arrays of objects with **new**
- Only uses default constructor in this case!

```
bogus *b2 = new bogus[10];  
cout << "b2[0]: " << b2[0].get_data()  
      << endl;  
delete [] b2; // NOT just delete!
```



# Dynamic memory allocation (3)

---

- Try using non-default constructor:

```
bogus *b3 = new bogus[10] (22); // WRONG!
```

- Can call init function after creation:

```
bogus *b3 = new bogus[10];  
for (int i = 0; i < 10; i++) {  
    b3[i].set_data(22);  
}
```



# DMA and classes (1)

---

- If class needs DMA, allocate memory in constructor
  - and deallocate in destructor!

```
class bogus2 {  
    private:  
        int *data;  
        int size;  
    public:  
        bogus2(int n);  
        ~bogus(); // need this now  
};
```



## DMA and classes (2)

---

```
bogus2::bogus2(int n) {  
    size = n;  
    data = new int[n]; // NOT initialized  
    for (int i = 0; i < n; i++) {  
        data[i] = 0;  
    }  
}
```



## DMA and classes (3)

---

- Deallocating memory

```
bogus2::~~bogus2() {  
    delete [] data;  
}
```

- Just `delete` will not work!
- `new []` and `delete []` must balance
- Compiler will not check this!



# Destructors (1)

---

- Destructor is automatically invoked when leaving the block that the object was created in:

```
void foo() {  
    bogus2 b; // constructor called  
    // yadda yadda...  
} // bogus2 destructor called on b
```



## Destructors (2)

---

- Can explicitly allocate object with **new**
- Then destructor must be called explicitly

```
void foo2() {  
    bogus2 *b = new bogus2;  
    // yadda yadda...  
    delete b;  
}
```



# Copy constructor (1)

---

- Want to do this:

```
bogus b1(42);
```

```
bogus b2(b1); // b2 is "the same" as b1
```

- Define copy constructor in **bogus.cc**

```
bogus::bogus(const bogus& other) {  
    data = other.get_data(); // OK  
    data = other.data;      // also OK  
}
```



## Copy constructor (2)

---

```
bogus::bogus(const bogus& other) {  
    data = other.data;  
}
```

- Arg to copy constructor must be a reference
  - why?
- Use **const** ref when you don't want to change the object being copied
  - which is nearly always
- When data being copied is an array, usually do NOT want to copy pointer to array contents!
  - why?



## Copy constructor (3)

---

```
class bogus2 {  
    private:  
        int *data;  
        int size;  
    public:  
        bogus2(int n);  
        bogus2(const bogus2& other);  
        ~bogus(); // need this now  
};
```



# Copy constructor (4)

---

```
bogus2::bogus2(const bogus2& other) {  
    size = other.size;  
    // WRONG (usually):  
    // data = other.data;  
    // RIGHT:  
    data = new int[size];  
    for (int i = 0; i < size; i++) {  
        data[i] = other.data[i];  
    }  
}
```



## Copy constructor (5)

---

- Function with non-ref arg of type with copy ctr invokes copy constructor when called:

```
void foo(bogus2 b) // copies b
{
    // whatever...
}
```

- here, `bogus2` copy ctr invoked when `foo` is called



# assert (1)

---

- Sometimes things go wrong
  - e.g. accessing element -1 of array (bad inputs)
  - e.g. logical errors in code
- How to deal with it?
  - bad inputs: use exceptions (later)
  - logical errors (things that "can't happen"):
    - use `assert()`
  - for now, use `assert()` for both cases



## assert (2)

---

- Using `assert()`:

```
#include <cassert>
```

```
// ...
```

```
assert(i >= 0);
```

```
return data[i];
```

- If assertion fails, program terminates!



# `bool` type

---

- Unlike C, C++ has a boolean type (`bool`)
- `true` and `false` are keywords
  - equivalent to 1 and 0
- Can convert freely from numerical types to `bool` and back
  - Yes, it's lame!
- Use `bool` instead of `int` for boolean variables