Dynamic memory management and OOP

CS 115

Dr. Joseph Eremondi, adapted from Dr. Shakil Khan, Dr. Philip Fong, and Dr. Howard Hamilton

Last updated: March 21, 2025

Interaction between dynamic memory management and OOD features, such as composition, inheritance, and dynamic binding

Composition and inheritance

- Recall invocation seq. of const.

Composition and inheritance

- Recall invocation seq. of const.

Composition and inheritance

- Recall invocation seq. of const.

```
class C : ... {
  ...
    };
```

- Invocation sequence does not depend on the order in which constructors are called. Instead:

Composition and inheritance

- Recall invocation seq. of const.

```
class C : ... {
  ...
    };
```

- Invocation sequence does not depend on the order in which constructors are called. Instead:
  - const. of base class

Composition and inheritance

- Recall invocation seq. of const.

```
class C : ... {
  ...
    };
```

- Invocation sequence does not depend on the order in which constructors are called. Instead:
  - const. of base class
  - const. of member vars, in the order they are defined

Composition and inheritance

- Recall invocation seq. of const.

```
class C : ... {
  ...
    };
```

- Invocation sequence does not depend on the order in which constructors are called. Instead:
  - const. of base class
  - const. of member vars, in the order they are defined
  - body of the constructor

Composition and inheritance

- Recall invocation seq. of const.

```
class C : ... {
  ...
    };
```

- Invocation sequence does not depend on the order in which constructors are called. Instead:
  - const. of base class
  - const. of member vars, in the order they are defined
  - body of the constructor

Composition and inheritance

- Recall invocation seq. of const.

```
class C : ... {
  ...
    };
```

- Invocation sequence does not depend on the order in which constructors are called. Instead:
  - const. of base class
  - const. of member vars, in the order they are defined
  - body of the constructor

```
class D : public C {
public:
  D(...);
  ...
  private:
  D1 f1;
  D2 f2;
  ...
};
D::D(...) : C(...), f2(...), f1(...), ... {
  ...}
```

Constructor invocation sequence example

Constructor invocation sequence example

Constructor invocation sequence example

Constructor invocation sequence example

```
class C {
public:
  C();
  C(const char *s);
};
C::C() {
  cout << "C()" << endl;
}
C::C(const char *s) {
  cout <<
    "C(const char *)" << endl;
}
class D : public C {
public:
  D();
};
```

Constructor invocation sequence example

```
class C {
public:
  C();
  C(const char *s);
};
C::C() {
  cout << "C()" << endl;
}
C::C(const char *s) {
  cout <<
    "C(const char *)" << endl;
}
class D : public C {
public:
  D();
};
```

Constructor invocation sequence example

```
class C {
public:
  C();
  C(const char *s);
};
C::C() {
  cout << "C()" << endl;
}
C::C(const char *s) {
  cout <<
    "C(const char *)" << endl;
}
class D : public C {
public:
  D();
};
```

```
D::D() : C() {
  cout << "D()" << endl;
}
class E : public D {
public:
  E();
private:
  C x;
  C z;
};
E::E() : D(),
         x("Hello"),
         z("Goodbye"){
  cout << "E()" << endl;
}
int main() {
  E y;
  return 0;
}
```

Destructor invocation sequence

- Destructor invocation sequence is exactly the opposite of that of the constructor!

Destructor invocation sequence

- Destructor invocation sequence is exactly the opposite of that of the constructor!

Destructor invocation sequence

- Destructor invocation sequence is exactly the opposite of that of the constructor!

```
class C : ... {
  ...
    };
```

- When object of type D goes out of scope, ~D() called, executes body, then calls ~C2(), ~C1(), ~C()

Destructor invocation sequence

- Destructor invocation sequence is exactly the opposite of that of the constructor!

```
class C : ... {
  ...
    };
```

- When object of type D goes out of scope, ~D() called, executes body, then calls ~C2(), ~C1(), ~C()

Destructor invocation sequence

- Destructor invocation sequence is exactly the opposite of that of the constructor!

```
class C : ... {
  ...
    };
```

- When object of type D goes out of scope, ~D() called, executes body, then calls ~C2(), ~C1(), ~C()

```
class D : public C {
public:
  ...
~D();
private:
  C1 f1;
  C2 f2;
  ...
};
D::~D() {
  ... // body of ~D()
    }
```

Destructor invocation sequence (cont'd)

Destructor invocation sequence (cont'd)

Destructor invocation sequence (cont'd)

Destructor invocation sequence (cont'd)

```
class C{
public:
  ~C() { cout << "~C()" << endl; }
};

class C1 : public C {
public:
  ~C1() { cout << "~C1()" << endl; }
};

class C2 : public C {
public:
  ~C2() { cout << "~C2()" << endl; }
};
```

Destructor invocation sequence (cont'd)

```
class C{
public:
  ~C() { cout << "~C()" << endl; }
};

class C1 : public C {
public:
  ~C1() { cout << "~C1()" << endl; }
};

class C2 : public C {
public:
  ~C2() { cout << "~C2()" << endl; }
};
```

Destructor invocation sequence (cont'd)

```
class C{
public:
  ~C() { cout << "~C()" << endl; }
};

class C1 : public C {
public:
  ~C1() { cout << "~C1()" << endl; }
};

class C2 : public C {
public:
  ~C2() { cout << "~C2()" << endl; }
};
```

```
class D : public C {
public:
  ~D() { cout << "~D()"
               << endl; }
private:
  C1 x;
  C2 y;
};

int main() {
  D z;
  return 0;
}
```

Constructors and destructors

Constructors and destructors

```
// calls constructor as usual
String *ps = new String;

// can also specify which constructor to use
String *ps = new String("Hello");

// (explicitly) calls the destructor
delete ps;
```

Recall: static vs. dynamic binding

Recall: static vs. dynamic binding

Recall: static vs. dynamic binding

Recall: static vs. dynamic binding

```
class C {
public:
  virtual void f() {
    /* implementation 1 */ }
  ...
};

class D : public C {
public:
  // implictly virtual
  void f() {
    /* implementation 2 */ }
  ...
};
```

Recall: static vs. dynamic binding

```
class C {
public:
  virtual void f() {
    /* implementation 1 */ }
  ...
};

class D : public C {
public:
  // implictly virtual
  void f() {
    /* implementation 2 */ }
  ...
};
```

Recall: static vs. dynamic binding

```
class C {
public:
  virtual void f() {
    /* implementation 1 */ }
  ...
};

class D : public C {
public:
  // implictly virtual
  void f() {
    /* implementation 2 */ }
  ...
};
```

```
void g(C &c) {
  c.f( );
}

int main() {
  D d;
  d.f();  // static binding: impl.2
  g(d);  // dynamic binding: impl.2
  // invoked
  return 0;
}
```

Virtual destructor

Virtual destructor

Virtual destructor

Virtual destructor

```
class C {
public:
  // Say this is an abstract class
  ...
// WRONG, use virtual ~C() instead
~C();
};

class D : public C {
public:
  ...
~D();
private:
  ...
};
```

Virtual destructor

```
class C {
public:
  // Say this is an abstract class
  ...
// WRONG, use virtual ~C() instead
~C();
};

class D : public C {
public:
  ...
~D();
private:
  ...
};
```

Virtual destructor

```
class C {
public:
  // Say this is an abstract class
  ...
// WRONG, use virtual ~C() instead
~C();
};

class D : public C {
public:
  ...
~D();
private:
  ...
};
```

```
void destroy(C *ptr) {
  ...
    // wanted to call ~D()
    delete ptr;
  ...
    }

int main() {
  C *p = new D;
  destroy(p);
  return 0;
}
```

Using classes that involve dynamically allocated memory

- If you craft your class properly by equipping it with:

Using classes that involve dynamically allocated memory

- If you craft your class properly by equipping it with:
  - a default constructor

Using classes that involve dynamically allocated memory

- If you craft your class properly by equipping it with:
  - a default constructor
  - a copy constructor

Using classes that involve dynamically allocated memory

- If you craft your class properly by equipping it with:
  - a default constructor
  - a copy constructor
  - an assignment operator and

Using classes that involve dynamically allocated memory

- If you craft your class properly by equipping it with:
  - a default constructor
  - a copy constructor
  - an assignment operator and
  - a destructor

Using classes that involve dynamically allocated memory

- If you craft your class properly by equipping it with:
  - a default constructor
  - a copy constructor
  - an assignment operator and
  - a destructor
- then you may simply treat the class as a built-in type

Example (using String to define Book)

- Class definition

Example (using String to define Book)

- Class definition

Example (using String to define Book)

- Class definition

```
class Book {
public:
  Book(const String &a, const String &t);
  Book(const Book &b);
  ~Book();
  Book &operator=(const Book &b);
  ...
  private:
  String author;
  String title;
};
```

Implementation (using String to define Book)

Implementation (using String to define Book)

```
Book::Book(const String &a, const String &t)
  : author(a), title(t) {}
Book::Book(const Book &b)
  : author(b.author), title(b.title) {}

Book &operator=(const Book &b) {
  if (&b != this) {
    author = b.author;
    title = b.title;
  }
  return *this;
}

Book::~Book() {}
```

Polymorphism

- Base class pointer = child class instance : works as expected

Polymorphism

- Base class pointer = child class instance : works as expected
- Child class pointer = base class instance : WRONG

Polymorphism

- Base class pointer = child class instance : works as expected
- Child class pointer = base class instance : WRONG

Polymorphism

- Base class pointer = child class instance : works as expected
- Child class pointer = base class instance : WRONG

```
class Critter {
  ...
};

class Spider : public Critter {
  ...
};

Critter *cp = new Spider(...); // works
Spider *sp1 = new Critter(...); // WRONG!
Spider *sp2;
sp2 = cp; // WRONG!
```

Polymorphism with pure virtual functions

Polymorphism with pure virtual functions

Polymorphism with pure virtual functions

Polymorphism with pure virtual functions

```
class Critter {
private:
  int legCount;
public:
  Critter(int n);
  virtual void print() = 0;
};

Critter::Critter(int n){
  legCount = n;
}
```

Polymorphism with pure virtual functions

```
class Critter {
private:
  int legCount;
public:
  Critter(int n);
  virtual void print() = 0;
};

Critter::Critter(int n){
  legCount = n;
}
```

Polymorphism with pure virtual functions

```
class Critter {
private:
  int legCount;
public:
  Critter(int n);
  virtual void print() = 0;
};

Critter::Critter(int n){
  legCount = n;
}
```

```
// Note: there is no implementation
// pure virtual function named print
class Spider:  public Critter{
private:
  bool poisonous;
public:
  Spider(bool poisonous1);
  virtual void print();
};

void Spider::print(){
  // body implements virtual func.
}

Spider *sp = new Spider(true);
sp->print();
Critter *cp = sp;
cp->print(); // dynamic binding
```

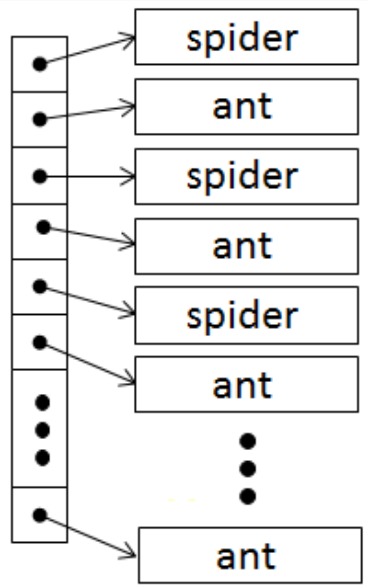
Polymorphism and dynamic arrays

Polymorphism and dynamic arrays

Polymorphism and dynamic arrays

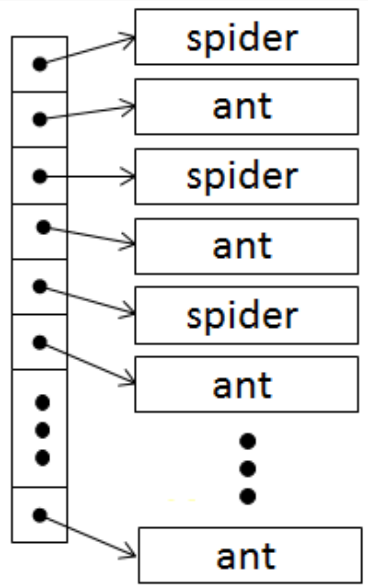
Polymorphism and dynamic arrays

```
Critter *critter_array[100];
for (int i = 0; i < 100; i++) {
  if (i % 2 == 0)
    critter_array[i] =
      new Spider(false);
  else
    critter_array[i] =
      new Ant(6, 50);
 }
```



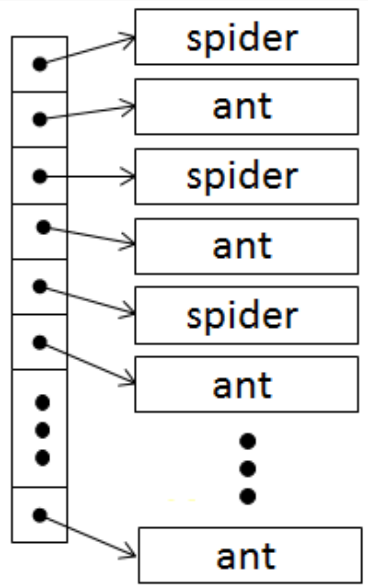
Polymorphism and dynamic arrays

```
Critter *critter_array[100];
for (int i = 0; i < 100; i++) {
  if (i % 2 == 0)
    critter_array[i] =
      new Spider(false);
  else
    critter_array[i] =
      new Ant(6, 50);
 }
```



Polymorphism and dynamic arrays

```
Critter *critter_array[100];
for (int i = 0; i < 100; i++) {
  if (i % 2 == 0)
    critter_array[i] =
      new Spider(false);
  else
    critter_array[i] =
      new Ant(6, 50);
 }
```




```
Critter **critter_array
  = new Critter*[100];
for (int i = 0; i < 100; i++) {
  if (i % 2 == 0)
    critter_array[i] =
      new Spider(false);
  else
    critter_array[i] =
      new Ant(6, 50);
 }
```