# Object-oriented design

CS 115

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

Last updated: January 28, 2025

# Composition, inheritance, polymorphism, dynamic binding, hidden functions & operators

# Terminology

- Top-down design: process of repeatedly decomposing a complicated problem into smaller, more manageable subproblems that can be solved by functions that can be implemented independently of the rest of the project

# Terminology

- Top-down design: process of repeatedly decomposing a complicated problem into smaller, more manageable subproblems that can be solved by functions that can be implemented independently of the rest of the project
- Object-oriented design (OOD): software design technique where the problem domain is decomposed into a set of objects that together solve a software problem

## Terminology

- Top-down design: process of repeatedly decomposing a complicated problem into smaller, more manageable subproblems that can be solved by functions that can be implemented independently of the rest of the project
- Object-oriented design (OOD): software design technique where the problem domain is decomposed into a set of objects that together solve a software problem
- Classes (allows us to define ADT)

# Terminology

- Top-down design: process of repeatedly decomposing a complicated problem into smaller, more manageable subproblems that can be solved by functions that can be implemented independently of the rest of the project
- Object-oriented design (OOD): software design technique where the problem domain is decomposed into a set of objects that together solve a software problem
- Classes (allows us to define ADT)
- Objects (=class instances)

## Terminology

- Top-down design: process of repeatedly decomposing a complicated problem into smaller, more manageable subproblems that can be solved by functions that can be implemented independently of the rest of the project
- Object-oriented design (OOD): software design technique where the problem domain is decomposed into a set of objects that together solve a software problem
- Classes (allows us to define ADT)
- Objects (=class instances)
- Fields (=class member fields/variables)

# Terminology

- Top-down design: process of repeatedly decomposing a complicated problem into smaller, more manageable subproblems that can be solved by functions that can be implemented independently of the rest of the project
- Object-oriented design (OOD): software design technique where the problem domain is decomposed into a set of objects that together solve a software problem
- Classes (allows us to define ADT)
- Objects (=class instances)
- Fields (=class member fields/variables)
- Methods (=class member functions)

# Terminology

- Top-down design: process of repeatedly decomposing a complicated problem into smaller, more manageable subproblems that can be solved by functions that can be implemented independently of the rest of the project
- Object-oriented design (OOD): software design technique where the problem domain is decomposed into a set of objects that together solve a software problem
- Classes (allows us to define ADT)
- Objects (=class instances)
- Fields (=class member fields/variables)
- Methods (=class member functions)
- Message Passing (=invocation of member functions through an object)

# Composition

- Idea:

# Composition

- Idea:
- say we want to define a class P1

# Composition

- Idea:
- say we want to define a class P1
  - conceptually divide P1 into constituent parts

# Composition

- Idea:
- say we want to define a class P1
    - conceptually divide P1 into constituent parts
- in the definition of the P1 class, declare instances of its constituents (which are other classes, say C1, C2, and C3)

# Composition

- Idea:
- say we want to define a class P1
    - conceptually divide P1 into constituent parts
- in the definition of the P1 class, declare instances of its constituents (which are other classes, say C1, C2, and C3)
- C++ compiler will call all of the constituent classes C1, C2, and C3's default constructors before it calls P1's constructor

# Composition

- Idea:
- say we want to define a class P1
    - conceptually divide P1 into constituent parts
- in the definition of the P1 class, declare instances of its constituents (which are other classes, say C1, C2, and C3)
- C++ compiler will call all of the constituent classes C1, C2, and C3's default constructors before it calls P1's constructor
- C++ syntax allows you to call other constructors of C1, C2, and C3 if needed, and pass the appropriate arguments in their parameters

# Composition

- Idea:
- say we want to define a class P1
  - conceptually divide P1 into constituent parts
- in the definition of the P1 class, declare instances of its constituents (which are other classes, say C1, C2, and C3)
- C++ compiler will call all of the constituent classes C1, C2, and C3's default constructors before it calls P1's constructor
- C++ syntax allows you to call other constructors of C1, C2, and C3 if needed, and pass the appropriate arguments in their parameters
- use the methods of C1, C2, and C3 using the declared objects while implementing the methods of P1

# Composition (example)

# Composition (example)

```
class Bicycle {
private:
  Wheel front_wheel;
  Wheel back_wheel;
  Seat seat;
public:
  Bicycle ();
```

- Bicycle (string wheel_manufacturer1,

# Composition (example)

```
class Bicycle {
private:
  Wheel front_wheel;
  Wheel back_wheel;
  Seat seat;
public:
  Bicycle ();
```

- Bicycle (string wheel_manufacturer1,
- string wheel_product1,

# Composition (example)

```
class Bicycle {
private:
  Wheel front_wheel;
  Wheel back_wheel;
  Seat seat;
public:
  Bicycle ();
```

- Bicycle (string wheel_manufacturer1,
- string wheel_product1,
- int diameter_in_inches1,

# Composition (example)

```
class Bicycle {
private:
  Wheel front_wheel;
  Wheel back_wheel;
  Seat seat;
public:
  Bicycle ();
```

- Bicycle (string wheel_manufacturer1,
- string wheel_product1,
- int diameter_in_inches1,
- int weight_in_grams1,

# Composition (example)

```
class Bicycle {
private:
  Wheel front_wheel;
  Wheel back_wheel;
  Seat seat;
public:
  Bicycle ();
```

- Bicycle (string wheel_manufacturer1,
- string wheel_product1,
- int diameter_in_inches1,
- int weight_in_grams1,
- int spokeCount1,

# Composition (example)

```
class Bicycle {
private:
  Wheel front_wheel;
  Wheel back_wheel;
  Seat seat;
public:
  Bicycle ();
```

- Bicycle (string wheel_manufacturer1,
- string wheel_product1,
- int diameter_in_inches1,
- int weight_in_grams1,
- int spokeCount1,
- string wheel_manufacturer2,

# Composition (example)

```
class Bicycle {
private:
  Wheel front_wheel;
  Wheel back_wheel;
  Seat seat;
public:
  Bicycle ();
```

- Bicycle (string wheel_manufacturer1,
- string wheel_product1,
- int diameter_in_inches1,
- int weight_in_grams1,
- int spokeCount1,
- string wheel_manufacturer2,
- string wheel_product2,

# Composition (example)

```cpp
class Bicycle {
private:
  Wheel front_wheel;
  Wheel back_wheel;
  Seat seat;
public:
  Bicycle ();
```

- Bicycle (string wheel_manufacturer1,
- string wheel_product1,
- int diameter_in_inches1,
- int weight_in_grams1,
- int spokeCount1,
- string wheel_manufacturer2,
- string wheel_product2,
- int diameter_in_inches2,

# Composition (example)

```
class Bicycle {
private:
  Wheel front_wheel;
  Wheel back_wheel;
  Seat seat;
public:
  Bicycle ();
```

- Bicycle (string wheel_manufacturer1,
- string wheel_product1,
- int diameter_in_inches1,
- int weight_in_grams1,
- int spokeCount1,
- string wheel_manufacturer2,
- string wheel_product2,
- int diameter_in_inches2,
- int weight_in_grams2,

# Composition (example)

```
class Bicycle {
private:
  Wheel front_wheel;
  Wheel back_wheel;
  Seat seat;
public:
  Bicycle ();
```

- Bicycle (string wheel_manufacturer1,
- string wheel_product1,
- int diameter_in_inches1,
- int weight_in_grams1,
- int spokeCount1,
- string wheel_manufacturer2,
- string wheel_product2,
- int diameter_in_inches2,
- int weight_in_grams2,

# Composition (example)

```
class Bicycle {
private:
  Wheel front_wheel;
  Wheel back_wheel;
  Seat seat;
public:
  Bicycle ();
```

- Bicycle (string wheel_manufacturer1,
- string wheel_product1,
- int diameter_in_inches1,
- int weight_in_grams1,
- int spokeCount1,
- string wheel_manufacturer2,
- string wheel_product2,
- int diameter_in_inches2,
- int weight_in_grams2,

# Composition (example)

```
class Bicycle {
private:
  Wheel front_wheel;
  Wheel back_wheel;
  Seat seat;
public:
  Bicycle ();
```

- Bicycle (string wheel_manufacturer1,
- string wheel_product1,
- int diameter_in_inches1,
- int weight_in_grams1,
- int spokeCount1,
- string wheel_manufacturer2,
- string wheel_product2,
- int diameter_in_inches2,
- int weight_in_grams2,

# Composition (example)

```
class Bicycle {
private:
  Wheel front_wheel;
  Wheel back_wheel;
  Seat seat;
public:
  Bicycle ();
```

- Bicycle (string wheel_manufacturer1,
- string wheel_product1,
- int diameter_in_inches1,
- int weight_in_grams1,
- int spokeCount1,
- string wheel_manufacturer2,
- string wheel_product2,
- int diameter_in_inches2,
- int weight_in_grams2,

# Composition (example)

```
class Bicycle {
private:
  Wheel front_wheel;
  Wheel back_wheel;
  Seat seat;
public:
  Bicycle ();
```

- Bicycle (string wheel_manufacturer1,
- string wheel_product1,
- int diameter_in_inches1,
- int weight_in_grams1,
- int spokeCount1,
- string wheel_manufacturer2,
- string wheel_product2,
- int diameter_in_inches2,
- int weight_in_grams2,

# Composition (example)

- Bicycle::Bicycle()

# Composition (example)

- Bicycle::Bicycle()
- : front_wheel(), back_wheel(),seat()

# Composition (example)

- Bicycle::Bicycle()
- : front_wheel(), back_wheel(),seat()

# Composition (example)

- Bicycle::Bicycle()
- : front_wheel(), back_wheel(),seat()

```
{
  // body of default constructor
}
```

- What happens when you declare a Bicycle object?

# Composition (example)

- Bicycle::Bicycle()
- : front_wheel(), back_wheel(),seat()

```
{
  // body of default constructor
}
```

- What happens when you declare a Bicycle object?

# Composition (example)

- Bicycle::Bicycle()
- : front_wheel(), back_wheel(),seat()

```
{
  // body of default constructor
}
```

- What happens when you declare a Bicycle object?

```
Bicycle b;
```

## Composition (example)

- Bicycle::Bicycle (string wheel_manufacturer1, string wheel_product1, int diameter_in_inches1,

# Composition (example)

- Bicycle::Bicycle (string wheel_manufacturer1, string wheel_product1, int diameter_in_inches1,
- int weight_in_grams1, int spokeCount1, string wheel_manufacturer2,

# Composition (example)

- Bicycle::Bicycle (string wheel_manufacturer1, string wheel_product1, int diameter_in_inches1,
- int weight_in_grams1, int spokeCount1, string wheel_manufacturer2,
- string wheel_product2, int diameter_in_inches2, int weight_in_grams2, int spokeCount2,

# Composition (example)

- Bicycle::Bicycle (string wheel_manufacturer1, string wheel_product1, int diameter_in_inches1,
- int weight_in_grams1, int spokeCount1, string wheel_manufacturer2,
- string wheel_product2, int diameter_in_inches2, int weight_in_grams2, int spokeCount2,
- string seat_manufacturer1, string seat_product1, string seat_colour1)

# Composition (example)

- Bicycle::Bicycle (string wheel_manufacturer1, string wheel_product1, int diameter_in_inches1,
- int weight_in_grams1, int spokeCount1, string wheel_manufacturer2,
- string wheel_product2, int diameter_in_inches2, int weight_in_grams2, int spokeCount2,
- string seat_manufacturer1, string seat_product1, string seat_colour1)
- : front_wheel (wheel_manufacturer1, wheel_product1,

## Composition (example)

- Bicycle::Bicycle (string wheel_manufacturer1, string wheel_product1, int diameter_in_inches1,
- int weight_in_grams1, int spokeCount1, string wheel_manufacturer2,
- string wheel_product2, int diameter_in_inches2, int weight_in_grams2, int spokeCount2,
- string seat_manufacturer1, string seat_product1, string seat_colour1)
- : front_wheel (wheel_manufacturer1, wheel_product1,
- diameter_in_inches1, weight_in_grams1, spokeCount1),

# Composition (example)

- Bicycle::Bicycle (string wheel_manufacturer1, string wheel_product1, int diameter_in_inches1,
- int weight_in_grams1, int spokeCount1, string wheel_manufacturer2,
- string wheel_product2, int diameter_in_inches2, int weight_in_grams2, int spokeCount2,
- string seat_manufacturer1, string seat_product1, string seat_colour1)
- : front_wheel (wheel_manufacturer1, wheel_product1,
- diameter_in_inches1, weight_in_grams1, spokeCount1),
- back_wheel (wheel_manufacturer2, wheel_product2,

# Composition (example)

- Bicycle::Bicycle (string wheel_manufacturer1, string wheel_product1, int diameter_in_inches1,
- int weight_in_grams1, int spokeCount1, string wheel_manufacturer2,
- string wheel_product2, int diameter_in_inches2, int weight_in_grams2, int spokeCount2,
- string seat_manufacturer1, string seat_product1, string seat_colour1)
- : front_wheel (wheel_manufacturer1, wheel_product1,
- diameter_in_inches1, weight_in_grams1, spokeCount1),
- back_wheel (wheel_manufacturer2, wheel_product2,
- diameter_in_inches2, weight_in_grams2, spokeCount2),

# Composition (example)

- Bicycle::Bicycle (string wheel_manufacturer1, string wheel_product1, int diameter_in_inches1,
- int weight_in_grams1, int spokeCount1, string wheel_manufacturer2,
- string wheel_product2, int diameter_in_inches2, int weight_in_grams2, int spokeCount2,
- string seat_manufacturer1, string seat_product1, string seat_colour1)
- : front_wheel (wheel_manufacturer1, wheel_product1,
- diameter_in_inches1, weight_in_grams1, spokeCount1),
- back_wheel (wheel_manufacturer2, wheel_product2,
- diameter_in_inches2, weight_in_grams2, spokeCount2),
- seat (seat_manufacturer1, seat_product1, seat_colour1)

# Composition (example)

- Bicycle::Bicycle (string wheel_manufacturer1, string wheel_product1, int diameter_in_inches1,
- int weight_in_grams1, int spokeCount1, string wheel_manufacturer2,
- string wheel_product2, int diameter_in_inches2, int weight_in_grams2, int spokeCount2,
- string seat_manufacturer1, string seat_product1, string seat_colour1)
- : front_wheel (wheel_manufacturer1, wheel_product1,
- diameter_in_inches1, weight_in_grams1, spokeCount1),
- back_wheel (wheel_manufacturer2, wheel_product2,
- diameter_in_inches2, weight_in_grams2, spokeCount2),
- seat (seat_manufacturer1, seat_product1, seat_colour1)

## Composition (example)

- Bicycle::Bicycle (string wheel_manufacturer1, string wheel_product1, int diameter_in_inches1,
- int weight_in_grams1, int spokeCount1, string wheel_manufacturer2,
- string wheel_product2, int diameter_in_inches2, int weight_in_grams2, int spokeCount2,
- string seat_manufacturer1, string seat_product1, string seat_colour1)
- : front_wheel (wheel_manufacturer1, wheel_product1,
- diameter_in_inches1, weight_in_grams1, spokeCount1),
- back_wheel (wheel_manufacturer2, wheel_product2,
- diameter_in_inches2, weight_in_grams2, spokeCount2),
- seat (seat_manufacturer1, seat_product1, seat_colour1)

```
{
  // body of initializing constructor
}
```

# Composition (example)

# Composition (example)

```
class Seat {
private:
  string manufacturer;
  string product;
  string colour;
public:
  Seat ();
  Seat (string manufacturer1, string product1, string colour1);
  Seat (const Seat &original);
  ~Seat ();
  Seat &operator= (const Seat &original);
  void read  (istream &in);
  void print (ostream &out);
};
```

- Seat::Seat (string manufacturer1,

# Composition (example)

```
class Seat {
private:
  string manufacturer;
  string product;
  string colour;
public:
  Seat ();
  Seat (string manufacturer1, string product1, string colour1);
  Seat (const Seat &original);
  ~Seat ();
  Seat &operator= (const Seat &original);
  void read  (istream &in);
  void print (ostream &out);
};
```

- Seat::Seat (string manufacturer1,
- string product1, string colour1)

# Composition (example)

```
class Seat {
private:
  string manufacturer;
  string product;
  string colour;
public:
  Seat ();
  Seat (string manufacturer1, string product1, string colour1);
  Seat (const Seat &original);
  ~Seat ();
  Seat &operator= (const Seat &original);
  void read  (istream &in);
  void print (ostream &out);
};
```

- Seat::Seat (string manufacturer1,
- string product1, string colour1)

# Composition (example)

```
class Seat {
private:
  string manufacturer;
  string product;
  string colour;
public:
  Seat ();
  Seat (string manufacturer1, string product1, string colour1);
  Seat (const Seat &original);
  ~Seat ();
  Seat &operator= (const Seat &original);
  void read  (istream &in);
  void print (ostream &out);
};
```

- Seat::Seat (string manufacturer1,
- string product1, string colour1)

```
: manufacturer(manufacturer1), // copy cons
```

# Composition (example)

```
class Seat {
private:
  string manufacturer;
  string product;
  string colour;
public:
  Seat ();
  Seat (string manufacturer1, string product1, string colour1);
  Seat (const Seat &original);
  ~Seat ();
  Seat &operator= (const Seat &original);
  void read  (istream &in);
  void print (ostream &out);
};
```

- Seat::Seat (string manufacturer1,
- string product1, string colour1)

```
: manufacturer(manufacturer1), // copy cons
```

# Composition (example)

```
class Seat {
private:
  string manufacturer;
  string product;
  string colour;
public:
  Seat ();
  Seat (string manufacturer1, string product1, string colour1);
  Seat (const Seat &original);
  ~Seat ();
  Seat &operator= (const Seat &original);
  void read  (istream &in);
  void print (ostream &out);
};
```

- Seat::Seat (string manufacturer1,
- string product1, string colour1)

```
: manufacturer(manufacturer1), // copy cons
```

# Composition (example)

```
class Seat {
private:
  string manufacturer;
  string product;
  string colour;
public:
  Seat ();
  Seat (string manufacturer1, string product1, string colour1);
  Seat (const Seat &original);
  ~Seat ();
  Seat &operator= (const Seat &original);
  void read  (istream &in);
  void print (ostream &out);
};
```

- Seat::Seat (string manufacturer1,
- string product1, string colour1)

```
: manufacturer(manufacturer1), // copy cons
```

# Composition (example)

```
class Seat {
private:
  string manufacturer;
  string product;
  string colour;
public:
  Seat ();
  Seat (string manufacturer1, string product1, string colour1);
  Seat (const Seat &original);
  ~Seat ();
  Seat &operator= (const Seat &original);
  void read  (istream &in);
  void print (ostream &out);
};
```

- Seat::Seat (string manufacturer1,
- string product1, string colour1)

```
: manufacturer(manufacturer1), // copy cons
```

## Composition (example)

- Bicycle::Bicycle (const Bicycle &original)

# Composition (example)

- Bicycle::Bicycle (const Bicycle &original)
- : front_wheel (original.front_wheel),

# Composition (example)

- Bicycle::Bicycle (const Bicycle &original)
- : front_wheel (original.front_wheel),
- back_wheel (original.back_wheel),

# Composition (example)

- Bicycle::Bicycle (const Bicycle &original)
- : front_wheel (original.front_wheel),
- back_wheel (original.back_wheel),
- seat (original.seat)

# Composition (example)

- Bicycle::Bicycle (const Bicycle &original)
- : front_wheel (original.front_wheel),
- back_wheel (original.back_wheel),
- seat (original.seat)

# Composition (example)

- Bicycle::Bicycle (const Bicycle &original)
- : front_wheel (original.front_wheel),
- back_wheel (original.back_wheel),
- seat (original.seat)

```
{
  // body of copy constructor
}
```

# Composition (another example)

# Composition (another example)

```
typedef int ItemType;

class GuardedArray {
public:
  static const unsigned int LENGTH = 500;
  GuardedArray();
  GuardedArray(ItemType x);
  ItemType retrieve(unsigned int i) const;
  void store(unsigned int i, ItemType x);
private:
  ItemType data_array[LENGTH];
};
```

# Composition (another example)

# Composition (another example)

```
GuardedArray::GuardedArray() {
  for (unsigned int i = 0; i < LENGTH; i++)
    data_array[i] = 0;
}

GuardedArray::GuardedArray(ItemType x) {
  for (unsigned int i = 0; i < LENGTH; i++)
    data_array[i] = x;
}

ItemType GuardedArray::retrieve(unsigned int i) const {
  assert(i < LENGTH);
  return data_array[i];
}

void GuardedArray::store(unsigned int i, ItemType x) {
  assert(i < LENGTH);
  data_array[i] = x;
}
```

# Composition (another example)

# Composition (another example)

```
class ManagedArray {

public:
  static const unsigned int MAX_LENGTH = GuardedArray::LENGTH;

  ManagedArray();
  ManagedArray(unsigned int n);
  ManagedArray(unsigned int n, ItemType x);

  unsigned int length() const;
  ItemType retrieve(unsigned int i) const;
  void store(unsigned int i, ItemType x);
  void insert(unsigned int i, ItemType x);
  void remove(unsigned int i);

private:
  unsigned int count;
  GuardedArray guaurded_array;

};
```

# Composition (another example)

# Composition (another example)

```
ManagedArray::ManagedArray(unsigned int n, ItemType x) : guaurded_arra
  assert(n <= MAX_LENGTH);
  count = n;
}

ItemType ManagedArray::retrieve(unsigned int i) const {
  assert(i < length());
  return guaurded_array.retrieve(i);
}
```

# Composition (another example)

# Composition (another example)

```
void ManagedArray::insert(unsigned int i, ItemType x) {
  assert(i <= length());
  assert(count < MAX_LENGTH);

  for (unsigned int j = count; j > i; j--)
    guaurded_array.store(j, guaurded_array.retrieve(j-1));
  guaurded_array.store(i, x);
  count++;
}
```

## Composition (yet another example)

- Can in turn define Multiset using ManagedArray (see notes for full details)

## Composition (yet another example)

- Can in turn define Multiset using ManagedArray (see notes for full details)
- Another approach (inheritance):

## Composition (yet another example)

- Can in turn define Multiset using ManagedArray (see notes for full details)
- Another approach (inheritance):
- start with base class (parent/super-class) that gives a vague idea of the objects that we are after

## Composition (yet another example)

- Can in turn define Multiset using ManagedArray (see notes for full details)
- Another approach (inheritance):
- start with base class (parent/super-class) that gives a vague idea of the objects that we are after
- define other more specialized derived classes (child/sub-classes) that “inherits” everything in the parent class

# Composition (yet another example)

- Can in turn define Multiset using ManagedArray (see notes for full details)
- Another approach (inheritance):
- start with base class (parent/super-class) that gives a vague idea of the objects that we are after
- define other more specialized derived classes (child/sub-classes) that “inherits” everything in the parent class
- can create a hierarchy of classes linked by the ancestor-descendant relation

# Inheritance

- Child class inherits everything in the parent class

# Inheritance

- Child class inherits everything in the parent class
- when an object of the child class is instantiated,

# Inheritance

- Child class inherits everything in the parent class
- when an object of the child class is instantiated,
- all fields of the parent class will be allocated

# Inheritance

- Child class inherits everything in the parent class
- when an object of the child class is instantiated,
- all fields of the parent class will be allocated
- But can only directly access some fields and methods

# Inheritance

- Child class inherits everything in the parent class
- when an object of the child class is instantiated,
- all fields of the parent class will be allocated
- But can only directly access some fields and methods
- those that are public (and protected)

# Inheritance

- Child class inherits everything in the parent class
- when an object of the child class is instantiated,
- all fields of the parent class will be allocated
- But can only directly access some fields and methods
- those that are public (and protected)
- Child class can re-implement some functions of the parent!

# Inheritance

- Child class inherits everything in the parent class
- when an object of the child class is instantiated,
- all fields of the parent class will be allocated
- But can only directly access some fields and methods
- those that are public (and protected)
- Child class can re-implement some functions of the parent!
- this is called function overriding

# Inheritance

- Child class inherits everything in the parent class
- when an object of the child class is instantiated,
- all fields of the parent class will be allocated
- But can only directly access some fields and methods
- those that are public (and protected)
- Child class can re-implement some functions of the parent!
- this is called function overriding
- Add to this mix the hierarchy of classes

# Inheritance

- Child class inherits everything in the parent class
- when an object of the child class is instantiated,
- all fields of the parent class will be allocated
- But can only directly access some fields and methods
- those that are public (and protected)
- Child class can re-implement some functions of the parent!
- this is called function overriding
- Add to this mix the hierarchy of classes
- e.g. C extends P, GC extends C

# Inheritance

- Child class inherits everything in the parent class
- when an object of the child class is instantiated,
- all fields of the parent class will be allocated
- But can only directly access some fields and methods
- those that are public (and protected)
- Child class can re-implement some functions of the parent!
- this is called function overriding
- Add to this mix the hierarchy of classes
- e.g. C extends P, GC extends C
- then all publicly inherited public fields of C will be members of GC

# Inheritance (public vs. private)

# Inheritance (public vs. private)

```
class P {
public:
  void f1();
  int f2() const;
  int f3() const;
private:
```

- int v1

# Inheritance (public vs. private)

```
class P {
public:
  void f1();
  int f2() const;
  int f3() const;
private:
```

- int v1

# Inheritance (public vs. private)

```
class P {
public:
  void f1();
  int f2() const;
  int f3() const;
private:
```

- int v1

```
int v2;
};




class C : public P {
public:
  void f4();
  double f5() const;
```

# Inheritance (public vs. private)

```
class P {
public:
  void f1();
  int f2() const;
  int f3() const;
private:
```

- int v1

```
int v2;
};



class C : public P {
public:
  void f4();
  double f5() const;
```

# Inheritance (public vs. private)

```cpp
class P {
public:
  void f1();
  int f2() const;
  int f3() const;
private:
```

- int v1

```cpp
int v2;
};




class C : public P {
public:
  void f4();
  double f5() const;
```

# Inheritance (public vs. private)

```
class P {
public:
  void f1();
  int f2() const;
  int f3() const;
private:
```

- int v1

```
int v2;
};



class C : public P {
public:
  void f4();
  double f5() const;
```

# Inheritance (public vs. private)

```
class P {
public:
  void f1();
  int f2() const;
  int f3() const;
private:
```

- int v1

```
int v2;
};



class C : public P {
public:
  void f4();
  double f5() const;
```

## Inheritance (hierarchy, overriding)

- Can specify a hierarchy:

# Inheritance (hierarchy, overriding)

- Can specify a hierarchy:

# Inheritance (hierarchy, overriding)

- Can specify a hierarchy:

```
class C : public P { ... };
class GC : public C { ... };
```

- Can override an inherited function:

# Inheritance (hierarchy, overriding)

- Can specify a hierarchy:

```
class C : public P { ... };
class GC : public C { ... };
```

- Can override an inherited function:

# Inheritance (hierarchy, overriding)

- Can specify a hierarchy:

```
class C : public P { ... };
class GC : public C { ... };
```

- Can override an inherited function:

```
class P {
public:
  void f1();
};

void P::f1(){
  // definition 1
}
class C : public P {
public:
  void f1();
  void f2();
};
```

# Inheritance (constructors)

- Constructor of the base class is implicitly invoked

# Inheritance (constructors)

- Constructor of the base class is implicitly invoked
- Can specify constructors as well

# Inheritance (constructors)

- Constructor of the base class is implicitly invoked
- Can specify constructors as well

# Inheritance (constructors)

- Constructor of the base class is implicitly invoked
- Can specify constructors as well

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

- To invoke a constructor of D:

# Inheritance (constructors)

- Constructor of the base class is implicitly invoked
- Can specify constructors as well

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

- To invoke a constructor of D:

# Inheritance (constructors)

- Constructor of the base class is implicitly invoked
- Can specify constructors as well

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

- To invoke a constructor of D:

# Inheritance (constructors)

- Constructor of the base class is implicitly invoked
- Can specify constructors as well

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

- To invoke a constructor of D:

# Inheritance (protected)

- Supports more flexibility

# Inheritance (protected)

- Supports more flexibility

# Inheritance (protected)

- Supports more flexibility

```
class P {
public:
  void f1();
protected:
  void f2();
private:
  int x;
};



class C : public P {
public:
  void f3();
private:
  int y;
};

class GC : public C {
```

## Inheritance type

- All permutations possible

## Inheritance type

- All permutations possible

# Inheritance type

- All permutations possible

```
class P {
public:
  void f1();
protected:
  void f2();
private:
  int x;
};



class C1 : public P {
  ...
};

class C2 : protected P {
  ...
};
```

# Inheritance type (cont'd)

# Inheritance type (cont'd)

# Inheritance type (cont'd)

```
class P {
public:
  void f1();
private:
  int x;
};

class C : protected P {
public:
  void f3();
};


void C::f3(){
  f1();         // all good
  x = 7;      // error, not accessible!
}

int main(){
  P p1;
```

# Example: the Building (base) class

# Example: the Building (base) class

```
class Building {

protected:
  // default constructor
  Building();

  // assignment constructor
```

- Building(const string& address1,

# Example: the Building (base) class

```
class Building {

protected:
  // default constructor
  Building();

  // assignment constructor
```

- Building(const string& address1,
- const string& owner1,

# Example: the Building (base) class

```
class Building {

protected:
  // default constructor
  Building();

  // assignment constructor
```

- Building(const string& address1,
- const string& owner1,
- unsigned int cost1,

# Example: the Building (base) class

```
class Building {

protected:
  // default constructor
  Building();

  // assignment constructor
```

- Building(const string& address1,
- const string& owner1,
- unsigned int cost1,

# Example: the Building (base) class

```
class Building {

protected:
  // default constructor
  Building();

  // assignment constructor
```

- Building(const string& address1,
- const string& owner1,
- unsigned int cost1,

```
unsigned int area1);


protected:
// member variables
string address;
string owner;
```

# Example: the House (child) class

# Example: the House (child) class

```
class House : public Building {

public:
  // constructors
  House();
```

- House(const string& address1,

# Example: the House (child) class

```
class House : public Building {

public:
  // constructors
  House();
```

- House(const string& address1,
- const string& owner1,

# Example: the House (child) class

```
class House : public Building {

public:
  // constructors
  House();
```

- House(const string& address1,
- const string& owner1,
- unsigned int cost1,

# Example: the House (child) class

```
class House : public Building {

public:
  // constructors
  House();
```

- House(const string& address1,
- const string& owner1,
- unsigned int cost1,
- unsigned int area1,

# Example: the House (child) class

```
class House : public Building {

public:
  // constructors
  House();
```

- House(const string& address1,
- const string& owner1,
- unsigned int cost1,
- unsigned int area1,
- unsigned int roomCount1,

# Example: the House (child) class

```
class House : public Building {

public:
  // constructors
  House();
```

- House(const string& address1,
- const string& owner1,
- unsigned int cost1,
- unsigned int area1,
- unsigned int roomCount1,
- bool fireplace1,

# Example: the House (child) class

```
class House : public Building {

public:
  // constructors
  House();
```

- House(const string& address1,
- const string& owner1,
- unsigned int cost1,
- unsigned int area1,
- unsigned int roomCount1,
- bool fireplace1,

# Example: the House (child) class

```
class House : public Building {

public:
  // constructors
  House();
```

- House(const string& address1,
- const string& owner1,
- unsigned int cost1,
- unsigned int area1,
- unsigned int roomCount1,
- bool fireplace1,

```
unsigned int applianceCount1);

// print data
void print() const;
```

# Example: implementation of House

- House::House(const string& address1,

# Example: implementation of House

- House::House(const string& address1,
- const string& owner1,

# Example: implementation of House

- House::House(const string& address1,
- const string& owner1,
- unsigned int cost1,

# Example: implementation of House

- House::House(const string& address1,
- const string& owner1,
- unsigned int cost1,
- unsigned int area1,

# Example: implementation of House

- House::House(const string& address1,
- const string& owner1,
- unsigned int cost1,
- unsigned int area1,
- unsigned int roomCount1,

## Example: implementation of House

- House::House(const string& address1,
- const string& owner1,
- unsigned int cost1,
- unsigned int area1,
- unsigned int roomCount1,
- bool fireplace1,

# Example: implementation of House

- House::House(const string& address1,
- const string& owner1,
- unsigned int cost1,
- unsigned int area1,
- unsigned int roomCount1,
- bool fireplace1,
- unsigned int applianceCount1)

## Example: implementation of House

- House::House(const string& address1,
- const string& owner1,
- unsigned int cost1,
- unsigned int area1,
- unsigned int roomCount1,
- bool fireplace1,
- unsigned int applianceCount1)
- : Building(address1, owner1,

# Example: implementation of House

- House::House(const string& address1,
- const string& owner1,
- unsigned int cost1,
- unsigned int area1,
- unsigned int roomCount1,
- bool fireplace1,
- unsigned int applianceCount1)
- : Building(address1, owner1,

# Example: implementation of House

- House::House(const string& address1,
- const string& owner1,
- unsigned int cost1,
- unsigned int area1,
- unsigned int roomCount1,
- bool fireplace1,
- unsigned int applianceCount1)
- : Building(address1, owner1,

```
cost1, area1) {
  roomCount = roomCount1;
  fireplace = fireplace1;
  applianceCount = applianceCount1;
              }
void House:: print() const {
  cout << "HOUSE"<< endl;
  cout << "Location: "<< address;
  cout << endl;
  ...
```

# Example: the Barn (base) class

# Example: the Barn (base) class

```cpp
class Barn : public Building {
public:
  // constructors
  Barn();
```

- Barn(const string& address1,

# Example: the Barn (base) class

```
class Barn : public Building {
public:
  // constructors
  Barn();
```

- Barn(const string& address1,
- const string& owner1,

# Example: the Barn (base) class

```
class Barn : public Building {
public:
  // constructors
  Barn();
```

- Barn(const string& address1,
- const string& owner1,
- unsigned int cost1,

# Example: the Barn (base) class

```
class Barn : public Building {
public:
  // constructors
  Barn();
```

- Barn(const string& address1,
- const string& owner1,
- unsigned int cost1,
- unsigned int area1,

# Example: the Barn (base) class

```
class Barn : public Building {
public:
  // constructors
  Barn();
```

- Barn(const string& address1,
- const string& owner1,
- unsigned int cost1,
- unsigned int area1,

# Example: the Barn (base) class

```
class Barn : public Building {
public:
  // constructors
  Barn();
```

- Barn(const string& address1,
- const string& owner1,
- unsigned int cost1,
- unsigned int area1,

```
float hayCapacity1);

// print
void print() const;
private:
// variables
float hayCapacity;

}
```

# Example: client code

## Example: client code

```
Barn b1("123 Farmyard Lane", "Jed", 135000, 1000, 24.3);
b1.print();

House h1("321 Walnut Ave", "Clem", 182000, 2400, 3, true, 6);
h1.print();
```

# Issues with inheritance

- Implementation inheritance = examples that we have seen earlier

# Issues with inheritance

- Implementation inheritance = examples that we have seen earlier
- allows code reuse

# Issues with inheritance

- Implementation inheritance = examples that we have seen earlier
- allows code reuse
- Reuse can be done better using composition

# Issues with inheritance

- Implementation inheritance = examples that we have seen earlier
- allows code reuse
- Reuse can be done better using composition
- easier to understand code

# Issues with inheritance

- Implementation inheritance = examples that we have seen earlier
- allows code reuse
- Reuse can be done better using composition
- easier to understand code
- encapsulation boundary are better protected

# Issues with inheritance

- Implementation inheritance = examples that we have seen earlier
- allows code reuse
- Reuse can be done better using composition
- easier to understand code
- encapsulation boundary are better protected
- less interdependencies

# Issues with inheritance

- Implementation inheritance = examples that we have seen earlier
- allows code reuse
- Reuse can be done better using composition
- easier to understand code
- encapsulation boundary are better protected
- less interdependencies
- For code reuse, we will almost always use composition rather than implementation inheritance

# Issues with inheritance

- Implementation inheritance = examples that we have seen earlier
- allows code reuse
- Reuse can be done better using composition
- easier to understand code
- encapsulation boundary are better protected
- less interdependencies
- For code reuse, we will almost always use composition rather than implementation inheritance
- Another more powerful use of inheritance = interface inheritance

# Interface inheritance

- Rather than reusing implementation, reuse interface!

# Interface inheritance

- Rather than reusing implementation, reuse interface!
- program to an interface, not an implementation

# Interface inheritance

- Rather than reusing implementation, reuse interface!
- program to an interface, not an implementation

# Interface inheritance

- Rather than reusing implementation, reuse interface!
- program to an interface, not an implementation

```
Say we want to develop 3 similar functions; how to rather implement or
```

- via a common interface

# Interface inheritance

- Rather than reusing implementation, reuse interface!
- program to an interface, not an implementation

```
Say we want to develop 3 similar functions; how to rather implement or
```

- via a common interface
- Key idea:

# Interface inheritance

- Rather than reusing implementation, reuse interface!
- program to an interface, not an implementation

```
Say we want to develop 3 similar functions; how to rather implement or
```

- via a common interface
- Key idea:
- introduce abstract interface (the base class)

# Interface inheritance

- Rather than reusing implementation, reuse interface!
- program to an interface, not an implementation

```
Say we want to develop 3 similar functions; how to rather implement or
```

- via a common interface
- Key idea:
- introduce abstract interface (the base class)
- write the function in terms of this interface

# Interface inheritance

- Rather than reusing implementation, reuse interface!
- program to an interface, not an implementation

```
Say we want to develop 3 similar functions; how to rather implement or
```

- via a common interface
- Key idea:
- introduce abstract interface (the base class)
- write the function in terms of this interface
- develop 3 derived classes that extend this base class and implements (virtual) functions of the base class

# Interface inheritance

- Rather than reusing implementation, reuse interface!
- program to an interface, not an implementation

```
Say we want to develop 3 similar functions; how to rather implement or
```

- via a common interface
- Key idea:
- introduce abstract interface (the base class)
- write the function in terms of this interface
- develop 3 derived classes that extend this base class and implements (virtual) functions of the base class
- C++ compiler will do the rest via dynamic binding

# Example: data sources

# Example: data sources

```
int sumArray(const int A[], unsigned int n) {
  int sum = 0;
  unsigned int i = 0;
  while (i < n) {
    sum += A[i];
    i++;
  }
  return sum;
}
```

# Example: data sources (cont'd)

# Example: data sources (cont'd)

```
int sumManagedArray(const ManagedArray &A) {
  int sum = 0;
  unsigned i = 0;
  while (i < A.length()) {
    sum += A.retrieve(i);
    i++;
  }
  return sum;
}
```

# Example: data sources (cont'd)

# Example: data sources (cont'd)

```cpp
int sumStandardInputStream() {
  int sum = 0;
  int next;
  cin >> next;
  while (cin) {
    sum += next;
    cin >> next;
  }
  return sum;
}
```

# Example: data sources (cont'd)

# Example: data sources (cont'd)

```
int sumDataSource(a data source) {
  int sum = 0;
  while (data source has not been exhausted) {
    sum += next entry in the data source;
    exclude the retrieved entry from future consideration;
  }
  return sum;
}
```

# Example: data sources (cont'd)

# Example: data sources (cont'd)

```
class DataSource {

public:

  // exhausted
  virtual bool exhausted() const = 0; // pure virtual function

  // next
  virtual int next() = 0; // pure virtual function

};
```

- Abstract class can't be instantiated (but can be referenced)

# Example: data sources (cont'd)

# Example: data sources (cont'd)

```
int sumDataSource(DataSource &ds) {
  int sum = 0;
  while (! ds.exhausted()) {
    sum += ds.next();
  }
  return sum;
}
```

- What's new: can be applied to instances of any derived class of DataSource

# Example: data sources (cont'd)

```
int sumDataSource(DataSource &ds) {
  int sum = 0;
  while (! ds.exhausted()) {
    sum += ds.next();
  }
  return sum;
}
```

- What's new: can be applied to instances of any derived class of DataSource
- Called a polymorphic function

# Example: data sources (cont'd)

# Example: data sources (cont'd)

```
const unsigned ARRAY_DATA_SOURCE_CAPACITY = 1000;

class ArrayDataSource : public DataSource {
public:
  ArrayDataSource(const int A[], unsigned int n);
  virtual bool exhausted() const;
  virtual int next();
private:
  int data[ARRAY_DATA_SOURCE_CAPACITY];
  unsigned length;
  unsigned i;
};
```

# Example: data sources (cont'd)

# Example: data sources (cont'd)

```
ArrayDataSource::ArrayDataSource(const int A[], unsigned int n) {
  assert(n < ARRAY_DATA_SOURCE_CAPACITY);
  for (unsigned int k = 0; k < n; k++)
    data[k] = A[k];
  length = n;
  i = 0;
}
bool ArrayDataSource::exhausted() const {
  return i == length;
}
int ArrayDataSource::next() {
  assert(! exhausted());
  i++;
  return data[i - 1];
}
```

# Example: data sources (cont'd)

# Example: data sources (cont'd)

```
// set up and initialize managed array data source
int A[ ] = { 1, 3, 9, -2 };
ArrayDataSource ads(A, 4);

// call sumDataSouce to add up entries
int sum = sumDataSource(ads);
```

- Which version of exhausted() and next() to use in sumDataSource(ads)?

# Example: data sources (cont'd)

```
// set up and initialize managed array data source
int A[ ] = { 1, 3, 9, -2 };
ArrayDataSource ads(A, 4);

// call sumDataSouce to add up entries
int sum = sumDataSource(ads);
```

- Which version of exhausted() and next() to use in sumDataSource(ads)?
- determined at runtime

# Example: data sources (cont'd)

```
// set up and initialize managed array data source
int A[ ] = { 1, 3, 9, -2 };
ArrayDataSource ads(A, 4);

// call sumDataSouce to add up entries
int sum = sumDataSource(ads);
```

- Which version of exhausted() and next() to use in sumDataSource(ads)?
- determined at runtime
- depends on the exact type of object ads is bound to

# Example: data sources (cont'd)

# Example: data sources (cont'd)

```cpp
class ManagedArrayDataSource : public DataSource {
public:
  ManagedArrayDataSource(const ManagedArray &A);
  virtual bool exhausted() const;
  virtual int next();
private:
  ManagedArray array;
  unsigned int i;
};
```

# Example: data sources (cont'd)

- ManagedArrayDataSource::ManagedArrayDataSource(const ManagedArray& A)

# Example: data sources (cont'd)

- ManagedArrayDataSource::ManagedArrayDataSource(const ManagedArray& A)

# Example: data sources (cont'd)

- ManagedArrayDataSource::ManagedArrayDataSource(const ManagedArray& A)

```
: array(A.length()) {
  for (unsigned int k = 0; k < A.length(); k++)
    array.store(k, A.retrieve(k));
  i = 0;
  }
bool ManagedArrayDataSource::exhausted() const {
  return i == array.length();
}
int ManagedArrayDataSource::next() {
  assert(! exhausted());
  i++;
  return array.retrieve(i - 1);
}
```

# Example: data sources (cont'd)

# Example: data sources (cont'd)

```
// set up and initialize managed array data source
int A[] = { 1, 3, 9, -2 };
ManagedArray ma;
for (unsigned int i = 0; i < 4; i++)
  ma.store(i, A[i]);
ManagedArrayDataSource mads(ma);

// call sumDataSouce to add up entries
int sum = sumDataSource(mads);
```

# Static vs. dynamic binding

# Static vs. dynamic binding

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

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




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

int main() {
```

# Static vs. dynamic binding (cont'd)

# Static vs. dynamic binding (cont'd)

```
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() {
```

# Static vs. dynamic binding (cont'd)

# Static vs. dynamic binding (cont'd)

```
class E : public C {
public:
  // This does not override f() in class C
  // so it is not implicitly virtual
  void f(int i) { /* implementation 3 */ }
  ...
};









int main() {
  E e;
  e.f();  // static binding: impl.1 invoked
  e.f(4); // static binding: impl.3 invoked
  return 0;
```

## Hidden functions and operators

- A function or operator in the base class with the same name and parameters as a function in the derived class

# Hidden functions and operators

- A function or operator in the base class with the same name and parameters as a function in the derived class
- can still access a hidden function using the base-class type qualifier

# Hidden functions and operators

- A function or operator in the base class with the same name and parameters as a function in the derived class
- can still access a hidden function using the base-class type qualifier

# Hidden functions and operators

- A function or operator in the base class with the same name and parameters as a function in the derived class
- can still access a hidden function using the base-class type qualifier

```
void Derived1::func() {
  Base1::func(); // func() is defined in both the base and the child c
  // ...
}
```

- And similarly for operators

# Hidden functions and operators

- A function or operator in the base class with the same name and parameters as a function in the derived class
- can still access a hidden function using the base-class type qualifier

```
void Derived1::func() {
  Base1::func(); // func() is defined in both the base and the child c
  // ...
}
```

- And similarly for operators

# Hidden functions and operators

- A function or operator in the base class with the same name and parameters as a function in the derived class
- can still access a hidden function using the base-class type qualifier

```
void Derived1::func() {
  Base1::func(); // func() is defined in both the base and the child
  // ...
}
```

- And similarly for operators

```
Derived1 &Derived1::operator=(const Derived1 &original) {
  if (this != &original) {
    Base1::operator=(original); // = is defined in both the base and
    field1 = original.field1;
  }
  return *this;
}
```