Object-oriented design: Composition and Inheritance

CS 115

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

Last updated: February 6, 2025

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

Terminology

- 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

- 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

- 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

- 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

- 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

- 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 (=calling of member functions through an object)

Composition

- Build new classes from existing ones

Composition

- Build new classes from existing ones
- e.g. define a class P1

Composition

- Build new classes from existing ones
- e.g. define a class P1
    - conceptually divide P1 into parts

Composition

- Build new classes from existing ones
- e.g. define a class P1
  - conceptually divide P1 into parts
- in the defn of P1 class, declare instances of parts

Composition

- Build new classes from existing ones
- e.g. define a class P1
  - conceptually divide P1 into parts
- in the defn of P1 class, declare instances of parts
  - (objects of classes, say C1, C2, and C3)

Composition

- Build new classes from existing ones
- e.g. define a class P1
  - conceptually divide P1 into parts
- in the defn of P1 class, declare instances of parts
  - (objects of classes, say C1, C2, and C3)
- C++ compiler will call C1, C2, and C3's default constructors before calls P1's constructor

Composition

- Build new classes from existing ones
- e.g. define a class P1
  - conceptually divide P1 into parts
- in the defn of P1 class, declare instances of parts
  - (objects of classes, say C1, C2, and C3)
- C++ compiler will call C1, C2, and C3's default constructors before calls P1's constructor
  - Can call other constructors of C1, C2, and C3 if needed, and pass the appropriate arguments in their parameters

Composition

- Build new classes from existing ones
- e.g. define a class P1
  - conceptually divide P1 into parts
- in the defn of P1 class, declare instances of parts
  - (objects of classes, say C1, C2, and C3)
- C++ compiler will call C1, C2, and C3's default constructors before calls P1's constructor
  - Can 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 from fields to implement P1 methods

Composition (example)

Composition (example)

Composition (example)

Composition (example)

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

public:
  Bicycle();
```

Composition (example)

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

public:
  Bicycle();
```

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,
          int spokeCount2,
          string seat_manufacturer1,
          string seat_product1,
          string seat_colour1);
  Bicycle(const Bicycle &original);
  Bicycle &operator=(const Bicycle &original)
  void read(istream &in);
  void print(ostream &out);
};
```

Explicit Initializers

- Special syntax to say how to initialize fields for default constructor

Explicit Initializers

- Special syntax to say how to initialize fields for default constructor

Explicit Initializers

- Special syntax to say how to initialize fields for default constructor

```
Bicycle::Bicycle()
    : front_wheel(), back_wheel(), seat()
{
  // body of default constructor
}
```

  - Could also give arguments depending how `Wheel` and `Seat` are defined

Explicit Initializers

- Special syntax to say how to initialize fields for default constructor

```
Bicycle::Bicycle()
    : front_wheel(), back_wheel(), seat()
{
  // body of default constructor
}
```

  - Could also give arguments depending how `Wheel` and `Seat` are defined
  - What happens when you declare a Bicycle object?

Explicit Initializers

- Special syntax to say how to initialize fields for default constructor

```
Bicycle::Bicycle()
    : front_wheel(), back_wheel(), seat()
{
  // body of default constructor
}
```

  - Could also give arguments depending how `Wheel` and `Seat` are defined
  - What happens when you declare a Bicycle object?

Explicit Initializers

- Special syntax to say how to initialize fields for default constructor

```
Bicycle::Bicycle()
    : front_wheel(), back_wheel(), seat()
{
  // body of default constructor
}
```

  - Could also give arguments depending how `Wheel` and `Seat` are defined
  - What happens when you declare a Bicycle object?

```
Bicycle b;
```

Initializers in non-default Constructors

Initializers in non-default Constructors

```
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) {...}
```

The Constituent Classes

The Constituent Classes

The Constituent Classes

The Constituent Classes

```
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 &o);
  void read(istream &in);
  void print(ostream &out);
};
```

The Constituent Classes

```
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 &o);
  void read(istream &in);
  void print(ostream &out);
};
```

The Constituent Classes

```
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 &o);
  void read(istream &in);
  void print(ostream &out);
};
```

```
Seat::Seat(string manufacturer1,
           string product1,
           string colour1)
    // copy constructors
    : manufacturer(manufacturer1),
      product(product1),
      colour(colour1)
{
}

Seat::Seat(const Seat &orig)
  : manufacturer(orig.manufacturer)
    product(orig.product),
    colour(orig.colour) {
}
```

Calling the Copy Constructor

Calling the Copy Constructor

```
Bicycle::Bicycle (const Bicycle &original)
 : front_wheel (original.front_wheel),
   back_wheel (original.back_wheel),
   seat (original.seat)
  {
    // body of copy constructor
  }
```

Another Example: Safe Arrays

Another Example: Safe Arrays

```
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];
};
```

Implementation

Implementation

```
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;
}
```

Managed Array with Insert/Remove

Managed Array with Insert/Remove

```
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;

};
```

Implementation (1)

Implementation (1)

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

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

Implementation (2)

Implementation (2)

```
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 vs. Inheritance

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

Composition vs. Inheritance

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

Composition vs. Inheritance

- 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 vs. Inheritance

- 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 vs. Inheritance

- 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

- Inheritance lets you *extend* a class into a new class

Inheritance

- Inheritance lets you *extend* a class into a new class
  - Child class inherits everything in the parent class

Inheritance

- Inheritance lets you *extend* a class into a new class
  - Child class inherits everything in the parent class
  - when an object of the child class is instantiated,

Inheritance

- Inheritance lets you *extend* a class into a new class
  - 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

- Inheritance lets you *extend* a class into a new class
  - 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
- Can can only directly access some fields and methods

Inheritance

- Inheritance lets you *extend* a class into a new class
  - 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
- Can can only directly access some fields and methods
  - those that are public (and protected)

Inheritance

- Inheritance lets you *extend* a class into a new class
  - 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
- Can can only directly access some fields and methods
  - those that are public (and protected)
  - `protected`: like `private` except child classes can access

Inheritance

- Inheritance lets you *extend* a class into a new class
  - 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
- Can can only directly access some fields and methods
  - those that are public (and protected)
  - `protected`: like `private` except child classes can access
- Child class can re-implement some functions of the parent

Inheritance

- Inheritance lets you *extend* a class into a new class
  - 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
- Can can only directly access some fields and methods
  - those that are public (and protected)
  - `protected`: like `private` except child classes can access
- Child class can re-implement some functions of the parent
  - this is called *function overriding*

Inheritance

- Inheritance lets you *extend* a class into a new class
  - 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
- Can can only directly access some fields and methods
  - those that are public (and protected)
  - `protected`: like `private` except child classes can access
- Child class can re-implement some functions of the parent
  - this is called *function overriding*
- Add to this mix the hierarchy of classes

Inheritance

- Inheritance lets you *extend* a class into a new class
  - 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
- Can can only directly access some fields and methods
  - those that are public (and protected)
  - `protected`: like `private` except child classes can access
- 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

- Inheritance lets you *extend* a class into a new class
  - 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
- Can can only directly access some fields and methods
  - those that are public (and protected)
  - `protected`: like `private` except child classes can access
- 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)

Inheritance (public vs. private)

Inheritance (public vs. private)

```
class P {
public:
  void f1();
  int f2() const;
  int f3() const;
private:
int v1
int v2;
};
```

Inheritance (public vs. private)

```
class P {
public:
  void f1();
  int f2() const;
  int f3() const;
private:
int v1
int v2;
};
```

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;
private:
  double v3;
};
```

- what happens when C x is declared?

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;
private:
  double v3;
};
```

- what happens when C x is declared?
- can we access f1 from inside C or its clients? 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;
private:
  double v3;
};
```

- what happens when C x is declared?
- can we access f1 from inside C or its clients? v1?
- how can we access v1 from inside C or its clients?

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;
private:
  double v3;
};
```

- what happens when C x is declared?
- can we access f1 from inside C or its clients? v1?
- how can we access v1 from inside C or its clients?
- what if we wrote `: private P`?

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;
private:
  double v3;
};
```

- what happens when C x is declared?
- can we access f1 from inside C or its clients? v1?
- how can we access v1 from inside C or its clients?
- what if we wrote `: private P`?

Inheritance (hierarchy, overriding)

Inheritance (hierarchy, overriding)

Inheritance (hierarchy, overriding)

- Can specify a hierarchy: GC
  `<:  C  <:  P`

Inheritance (hierarchy, overriding)

- Can specify a hierarchy: GC
  `<:  C  <:  P`

Inheritance (hierarchy, overriding)

- Can specify a hierarchy: GC <: C <: P

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

- Can override an inherited function:

Inheritance (hierarchy, overriding)

- Can specify a hierarchy: GC <: C <: P

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

- Can override an inherited function:

Inheritance (hierarchy, overriding)

- Can specify a hierarchy: GC <: C <: P

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

- Can override an inherited function:

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

void P::f1(){
  // definition 1
}
```

Inheritance (hierarchy, overriding)

- Can specify a hierarchy: GC <: C <: P

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

- Can override an inherited function:

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

void P::f1(){
  // definition 1
}
```

Inheritance (hierarchy, overriding)

- Can specify a hierarchy: GC <: C <: P

```
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();
};
void C::f1(){
  // definition 2
}
void C::f2(){
  f1(); // which f1?
}
// how to call P's f1() in C?
```

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

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(...), ...
{ ...   }
```

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:
  - a constructor C is invoked (which may initiate the invocation of other constructors)

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:
  - a constructor C is invoked (which may initiate the invocation of other constructors)
  - a constructor of each member field `fi` is invoked (which may initiate the invocation of other constructors)

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:
  - a constructor C is invoked (which may initiate the invocation of other constructors)
  - a constructor of each member field `fi` is invoked (which may initiate the invocation of other constructors)
  - the body of the constructor of D is invoked

Inheritance (protected)

Inheritance (protected)

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;
};
```

Inheritance (protected)

- Supports more flexibility

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

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 {
public:
  void f4();
private:
  int z;
};
```

Inheritance type

Inheritance type

Inheritance type

- All permutations possible

Inheritance type

- All permutations possible
  - `: public P` makes all public P members public in C

Inheritance type

- All permutations possible
  - `: public P` makes all public P members public in C
  - `: protected P` makes all public P members protected in C

Inheritance type

- All permutations possible
  - `: public P` makes all public P members public in C
  - `: protected P` makes all public P members protected in C
  - `: private P` makes all public P members protected in C

Inheritance type

- All permutations possible
  - `: public P` makes all public P members public in C
  - `: protected P` makes all public P members protected in C
  - `: private P` makes all public P members protected in C

Inheritance type

- All permutations possible
  - `: public P` makes all public P members public in C
  - `: protected P` makes all public P members protected in C
  - `: private P` makes all public P members protected in C

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

Inheritance type

- All permutations possible
  - `: public P` makes all public P members public in C
  - `: protected P` makes all public P members protected in C
  - `: private P` makes all public P members protected in C

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

Inheritance type

- All permutations possible
  - `: public P` makes all public P members public in C
  - `: protected P` makes all public P members protected in C
  - `: private P` makes all public P members protected in C

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

```
class C1 : public P {
  ...
};

class C2 : protected P {
  ...
};

class C3 : private P {
  ...
};

// stronger qualifier ``wins'' !
```

Inheritance type (cont’d)

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();
};
```

Inheritance type (cont'd)

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

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

Inheritance type (cont'd)

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

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

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

int main(){
  P p1;
  C c1;
  // works
  p1.f1();
  // error, not accessible!
  c1.f1();
  ...
    }
```

Example: the Building (base) class

- Make a type that it’s impossible to create a value of

Example: the Building (base) class

- Make a type that it’s impossible to create a value of
  - BUT which is the common parent for other types

Example: the Building (base) class

- Make a type that it’s impossible to create a value of
  - BUT which is the common parent for other types

Example: the Building (base) class

- Make a type that it’s impossible to create a value of
  - BUT which is the common parent for other types

Example: the Building (base) class

- Make a type that it’s impossible to create a value of
  - BUT which is the common parent for other types

Example: the Building (base) class

- Make a type that it's impossible to create a value of
  - BUT which is the common parent for other types

```
class Building {

protected:
  // default constructor
  Building();

  // assignment constructor
  Building(const string &address1,
           const string &owner1,
           unsigned int cost1,
           unsigned int area1);
```

Example: the Building (base) class

- Make a type that it's impossible to create a value of
  - BUT which is the common parent for other types

```
class Building {

protected:
  // default constructor
  Building();

  // assignment constructor
  Building(const string &address1,
           const string &owner1,
           unsigned int cost1,
           unsigned int area1);
```

Example: the Building (base) class

- Make a type that it's impossible to create a value of
  - BUT which is the common parent for other types

```
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;
  unsigned int cost;
  unsigned int area;
};

// Assumes: won't ever create a
// Building object!
```

Example: the House (child) class

Example: the House (child) class

Example: the House (child) class

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);
```

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);
```

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;

private:
  // additional member variables
  unsigned int roomCount;
  bool fireplace;
  unsigned int applianceCount;
};
```

Example: implementation of House

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;
}

cout << "HOUSE" << endl;
cout << "Location: " << address;
cout << endl;
... cout << "Bedrooms: " << roomCount;
cout << endl;
...
}
```

Example: the Barn (base) class

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 vs. Composition

- Implementation inheritance = examples that we have seen earlier

Issues with Inheritance vs. Composition

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

Issues with Inheritance vs. Composition

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

Issues with Inheritance vs. Composition

- 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 vs. Composition

- 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 vs. Composition

- 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
  - fewer interdependencies

Issues with Inheritance vs. Composition

- 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
  - fewer interdependencies
- For code reuse, we will almost always use composition rather than implementation inheritance

Issues with Inheritance vs. Composition

- 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
  - fewer interdependencies
- For code reuse, we will almost always use composition rather than implementation inheritance
- More powerful: interface inheritance