Object-oriented design: Interfaces, Subtying and Polymorphism

CS 115

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

Last updated: February 6, 2025

Polymorphism, dynamic binding, hidden functions & operators

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
- Say we want to develop 3 similar functions; how to rather implement one?

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 one?
  - 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 one?
  - 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 one?
  - 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 one?
    - 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 one?
    - 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 one?
  - 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: Summing All Elements in a List-like structure

- Array version:

Example: Summing All Elements in a List-like structure

- Array version:

Example: Summing All Elements in a List-like structure

- Array version:

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

ManagedArray Version

ManagedArray Version

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

Standard Input Stream Version

Standard Input Stream Version

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

Reducing Redundancy

- All the examples were doing essentially the same thing

Reducing Redundancy

- All the examples were doing essentially the same thing
- In Pseudocode:

Reducing Redundancy

- All the examples were doing essentially the same thing
- In Pseudocode:

Reducing Redundancy

- All the examples were doing essentially the same thing
- In Pseudocode:

```
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;
}
```

- How can we make this concrete code?

Abstract Classes and the Virtual keyword

- When we declare a method as virtual, it *must be overridden* in some child class

Abstract Classes and the Virtual keyword

- When we declare a method as virtual, it *must be overridden* in some child class
  - Give a dummy value of `0`

Abstract Classes and the Virtual keyword

- When we declare a method as virtual, it *must be overridden* in some child class
  - Give a dummy value of `0`
- An *abstract class* has at least one virtual method

Abstract Classes and the Virtual keyword

- When we declare a method as virtual, it *must be overridden* in some child class
  - Give a dummy value of `0`
- An *abstract class* has at least one virtual method
  - Gives *interface* without an implementation

Abstract Classes and the Virtual keyword

- When we declare a method as virtual, it *must be overridden* in some child class
  - Give a dummy value of `0`
- An *abstract class* has at least one virtual method
  - Gives *interface* without an implementation

Abstract Classes and the Virtual keyword

- When we declare a method as virtual, it *must be overridden* in some child class
  - Give a dummy value of 0
- An *abstract class* has at least one virtual method
  - Gives *interface* without an implementation

```
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 or extended)

Generic Code via Abstract Classes

Generic Code via Abstract Classes

```
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

Generic Code via Abstract Classes

```
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**

Extending the Interface

- Array Version

Extending the Interface

- Array Version
  - Has the same methods as `DataSource`

Extending the Interface

- Array Version
  - Has the same methods as `DataSource`
  - BUT we specify the concrete representation

Extending the Interface

- Array Version
  - Has the same methods as `DataSource`
  - BUT we specify the concrete representation
    - `private` fields

Extending the Interface

- Array Version
    - Has the same methods as `DataSource`
    - BUT we specify the concrete representation
        - `private` fields

Extending the Interface

- Array Version
  - Has the same methods as `DataSource`
  - BUT we specify the concrete representation
    - `private` fields

```
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;
};
```

Implementing the Interface

- Must give implementation of each unspecified method from `DataSource`

Implementing the Interface

- Must give implementation of each unspecified method from `DataSource`

Implementing the Interface

- Must give implementation of each unspecified method from DataSource

```
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];
}
```

Dynamic Binding

Dynamic Binding

```
// 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)`?

Dynamic Binding

```
// 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**

Dynamic Binding

```
// 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

Dynamic Binding

```
// 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
    - In this case, `ArrayDataSource`

A Different Instance

- For `ManagedArray`

A Different Instance

- For `ManagedArray`

A Different Instance

- For ManagedArray

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

Implementation

Implementation

```
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);
}
```

Dynamic Binding for the Managed Array Version

Dynamic Binding for the Managed Array Version

```
// 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);
```

- The code of `sumDataSource` *has not changed!*

Dynamic Binding for the Managed Array Version

```
// 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);
```

- The code of `sumDataSource` *has not changed!*
  - But `next` and `exhausted` functions it was provided have

Static Binding Example

- Static means “at compile time”

Static Binding Example

- Static means “at compile time”
- C++ uses static by default

Static Binding Example

- Static means "at compile time"
- C++ uses static by default
    - Not true in other languages (Java, Python)

Static Binding Example

- Static means “at compile time”
- C++ uses static by default
  - Not true in other languages (Java, Python)

Static Binding Example

- Static means “at compile time”
- C++ uses static by default
    - Not true in other languages (Java, Python)

Static Binding Example

- Static means “at compile time”
- C++ uses static by default
  - Not true in other languages (Java, Python)

Static Binding Example

- Static means “at compile time”
- C++ uses static by default
  - Not true in other languages (Java, Python)

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

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

Static Binding Example

- Static means “at compile time”
- C++ uses static by default
  - Not true in other languages (Java, Python)

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

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

Static Binding Example

- Static means “at compile time”
- C++ uses static by default
  - Not true in other languages (Java, Python)

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

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

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

int main() {
  D d;
  // static binding: impl.2 invoked
  d.f();
  // static binding: impl.1 invoked
  g(d);
  return 0;
}
```

Dynamic Binding Example

- Dynamic means “at run time”

Dynamic Binding Example

- Dynamic means “at run time”
  - `virtual` keyword tells C++ to use dynamic binding

Dynamic Binding Example

- Dynamic means “at run time”
    - `virtual` keyword tells C++ to use dynamic binding

Dynamic Binding Example

- Dynamic means "at run time"
  - `virtual` keyword tells C++ to use dynamic binding

Dynamic Binding Example

- Dynamic means "at run time"
  - `virtual` keyword tells C++ to use dynamic binding

Dynamic Binding Example

- Dynamic means “at run time”
  - `virtual` keyword tells C++ to use dynamic binding

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

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

Dynamic Binding Example

- Dynamic means "at run time"
  - `virtual` keyword tells C++ to use dynamic binding

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

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

Dynamic Binding Example

- Dynamic means “at run time”
  - `virtual` keyword tells C++ to use dynamic binding

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

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

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

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

Overloading is Static

- Overloading, *not overriding*

Overloading is Static

- Overloading, *not overriding*

Overloading is Static

- Overloading, *not overriding*

```
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;
}
```

Polymorphism

- From the Greek

Polymorphism

- From the Greek
  - *Poly* : many

Polymorphism

- From the Greek
  - *Poly* : many
  - *Morphos* : forms

Polymorphism

- From the Greek
  - *Poly* : many
  - *Morphos* : forms
- OOP lets us write one function that works on many types

Polymorphism

- From the Greek
  - *Poly* : many
  - *Morphos* : forms
- OOP lets us write one function that works on many types
- This is *subtyping polymorphism*

Polymorphism

- From the Greek
  - *Poly* : many
  - *Morphos* : forms
- OOP lets us write one function that works on many types
- This is *subtyping polymorphism*
  - We'll see another kind later

Polymorphism

- From the Greek
  - *Poly* : many
  - *Morphos* : forms
- OOP lets us write one function that works on many types
- This is *subtyping polymorphism*
  - We'll see another kind later
  - If A extends B, you can use a B object anywhere an A is expected

Polymorphism

- From the Greek
  - *Poly* : many
  - *Morphos* : forms
- OOP lets us write one function that works on many types
- This is *subtyping polymorphism*
  - We'll see another kind later
  - If A extends B, you can use a B object anywhere an A is expected
  - If A extends B, we say A is a *subtype* of B

What Actually Are Objects?

- We can finally say what an object *is*:

What Actually Are Objects?

- We can finally say what an object *is*:
    - An instance of a class

What Actually Are Objects?

- We can finally say what an object *is*:
  - An instance of a class
  - Containing a value for each field of that class

What Actually Are Objects?

- We can finally say what an object *is*:
  - An instance of a class
  - Containing a value for each field of that class
  - *and* containing a pointer to the implementation of each virtual method of that class

What Actually Are Objects?

- We can finally say what an object *is*:
  - An instance of a class
  - Containing a value for each field of that class
  - *and* containing a pointer to the implementation of each virtual method of that class
- Objects package data and operations together

Subyping as a Safer Alternative to Unions

- Recall our library code

Subyping as a Safer Alternative to Unions

- Recall our library code

Subyping as a Safer Alternative to Unions

- Recall our library code

```
struct CatalogEntry {
  string title;
  string author;
  string publisher;
  unsigned int publishingYear;
  string callNumber;
  CatalogEntryType tag;
  union {
    struct { unsigned int pages; } book;
    struct { unsigned int discs, minutes; } dvd;
  } variant;
};
```

  - Can do better with OOP

CatalogEntry Abstract Class

- Base class with all of the shared fields

CatalogEntry Abstract Class

- Base class with all of the shared fields

CatalogEntry Abstract Class

- Base class with all of the shared fields

```
class CatalogEntry {
public:
  // e.g. an operation we want to do for any entry
  virtual void printInfo() = 0;
protected:
  string title;
  string author;
  string publisher;
  unsigned int publishingYear;
  string callNumber;
  // No tag or variant-specific info
};
```

A Book Subtype

A Book Subtype

```
//Book.h
class BookEntry : public CatalogEntry {
public:
 void printInfo();
private:
  int pages;
}
// Book.cpp
void BookEntry::printInfo(){
  // Have all CatalogEntry fields
  // plus pages
  cout << title << author ...
     << pages;
}
```

A DVD Subtype

A DVD Subtype

```cpp
//Book.h
class DVDEntry : public CatalogEntry {
public:
 void printInfo();
private:
  int discs;
  int minutes;
}
// Book.cpp
void DVDEntry::printInfo(){
  // Have all CatalogEntry fields
  // plus pages
  cout << title << author ...
     << discs << minutes;
}
```

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 access 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 access 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 access using the base-class type qualifier

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

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

Hidden Function Example

Hidden Function Example

Hidden Function Example

- Leave `CatalogEntry::printInfo()` as virtual, but give it an implementation

Hidden Function Example

- Leave `CatalogEntry::printInfo()` as virtual, but give it an implementation

Hidden Function Example

- Leave `CatalogEntry::printInfo()` as virtual, but give it an implementation

```
void CatalogEntry::printInfo(){
cout << title << author <<
    ... << callNumber;
  }
```

- `BookEntry` and `DVDEntry` then only need to deal with their specific fields

Hidden Function Example

- Leave `CatalogEntry::printInfo()` as virtual, but give it an implementation

```
void CatalogEntry::printInfo(){
cout << title << author <<
    ... << callNumber;
  }
```

- `BookEntry` and `DVDEntry` then only need to deal with their specific fields
  - Can call the base type version for the rest;

Hidden Function Example

- Leave `CatalogEntry::printInfo()` as virtual, but give it an implementation

```
void CatalogEntry::printInfo(){
cout << title << author <<
    ... << callNumber;
  }
```

- `BookEntry` and `DVDEntry` then only need to deal with their specific fields
  - Can call the base type version for the rest;

Hidden Function Example

- Leave `CatalogEntry::printInfo()` as virtual, but give it an implementation

```
void CatalogEntry::printInfo(){
cout << title << author <<
    ... << callNumber;
  }
```

- `BookEntry` and `DVDEntry` then only need to deal with their specific fields
    - Can call the base type version for the rest;

Hidden Function Example

- Leave `CatalogEntry::printInfo()` as virtual, but give it an implementation

```
void CatalogEntry::printInfo(){
cout << title << author <<
    ... << callNumber;
  }
```

- `BookEntry` and `DVDEntry` then only need to deal with their specific fields
  - Can call the base type version for the rest;

```
void BookEntry::printInfo() {
  // Call the base class version
  // to print the shared fields
  CatalogEntry::printInfo();
  // Print our specifc fields
  cout << pages;
}

void DVDEntry::printInfo() {
  CatalogEntry::printInfo();
  cout << discs << minutes;
}
```