Programmer-Defined Conversions

- Supply way for compiler to convert automatically instances of one data type into instances of another type
- Conversion defined by special member functions called conversion operators
  - Consequence: Cannot convert from a built-in type to another built-in type
  - Possible to convert from built-in type to class instance and vice versa, or between class types

Two Conversion Kinds

- **Inward** conversion: member function defined in conversion destination class
- **Outward** conversion: member function defined in conversion source class
- Inward conversion done with single-argument constructor in target class
- Example: The String constructor with char* parameter converts from char* to String (i.e., from C-style to C++-style strings)

Example of Inward Conversion

- Conversion operator from source type char* to destination class String through String constructor

```
String::String(const char* s)   {
  size_ = strlen(s);
  string_ = new char[size_ + 1];
  strcpy(string_, s) ;
}
```


Example of Inward Conversion

- Conversion applied automatically by compiler whenever possible (including initializations), except for message receivers

```
String s1 = “hello”;          // use inward conversion operator

“hi” < s1;                    // problem if operator<() defined as String
                              // member function; recall that the type of
                              // literal “hi” is const char*
```


Outward Conversions

- Special operator in source class of conversion
- Syntax:
  - function name: operator type_name()
  - empty parameter list
  - no return type, but must have return statement
  - Example: Conversion from String to char*

```
String::operator char*() {
  char* s;
  s = new char[size_ + 1];      // Beware of asymmetric deallocation!
  strcpy(s, string_);
  return s; }
```


Potential Conversion Error

- **Don't define outward and inward conversions meant for same source and destination classes**
- Resulting code is ambiguous because multiple conversions are applicable (a compile-time error)
- Example: Person class has subclass Employee
  - If Person has outward conversion to Employee, and Employee has constructor from Person, all conversions from Person to Employee are errors

```
Person::operator Employee() {  ...  }
Employee::Employee (const Person&)  { ... }
Employee e1 = person1;                    // Ambiguous
```

Casts and Overloading Resolution

Recall that programmer-defined conversions are part of standard overloading resolution (static conversions):

1. Exact match or trivial conversion
2. Numeric promotion
3. Standard conversion
4. **Programmer-defined conversion**
5. Ellipses

Note: Compiler can apply programmer-defined conversions before or after promotion or standard conversion!

Potential Resolution Error

- Suppose that:
  - Classes C1 and C2 have mutual conversions (with constructor and/or conversion operator)
  - Binary operator op is overloaded for arguments of type C1 or C2
  - op called with a C1 and a C2 argument
  - Result: Error (multiple conversions possible)

Example of Resolution Error

- Example: Suppose String and char* have mutual conversions

```
// file scope overloaded definitions
bool operator<(String&, String&);
bool operator<(char*, char*);

String s1 = "hello";
char*  ptr1 = "hi";

 // Error: 2 possible conversions
if (s1 < ptr1) …
```


Inheritance in C++

- Multiple inheritance model with support for both static and dynamic binding of messages and methods
- Basic terminology
  - Superclass = base class
  - Subclass = derived class
- Derived class inherits all data members from each of its base classes
- Derived class inherits all member functions from each of its base classes, except for constructors (until C++11), and the assignment operator

Definition of Derived Classes

- Base classes declared in so-called derivation list of derived class
- Derivation list
  - located between header and body of derived class
  - preceded by colon
  - comma-separated list of base classes
  - each base class can be preceded by an optional access specification (private, protected, or public)

Examples of Derived Class Definitions

- Point3D is a subclass of Point (is-a inclusion)

```
class point3D : public Point  {
   // member declarations
} ;
```

- A class of its own:

```
class UICprofessor : public Animal,
    protected FerociousCreature,
    private ViciousBeing { ... } ;
```


Example of Subclass Definition

- Definition of class Point3D

```
class Point3D : public Point {
  public:
    int z() { return z_; }
    void set_z(int new_z) {z_ = new_z; }
    double distance(Point3D& pt);
  protected:
    int z_;
};
```

- Now Point3D has data members x_, y_, and z_; and member functions x(), y(), z() and so on
- Notice: Do not redeclare x_, y_, x(), y() in subclass

Example of Subclass Definition

- Example of member function in class Point3D

```
double Point3D::distance(Point3D& pt)   {
  int dist_x, dist_y, dist_z;
  dist_x = pt.x() - x_;
  dist_y = pt.y() - y_;
  dist_z = pt.z() - z_;
  return sqrt(dist_x * dist_x + dist_y * dist_y + dist_z * dist_z); }
```

- Inherited data members x_ and y_ are accessible directly only if they were defined protected in superclass Point

Access Levels to Base Classes

- Determine how inherited members are treated by derived class (i.e. how they can be accessed by clients and subclasses of the derived class)
  - *Public* base class: Members inherited from that class are given same access levels in derived class as in base class

    Members public/protected/private in base class are still public/protected/private in derived class

Access Levels to Base Classes

- Base class levels (continued):
  - *Protected* base class: Public members in base class become protected in derived class

    Members protected/private in base class remain protected/private in derived class
  - *Private* base class: All inherited members are private in derived class, regardless of how they were defined in base class

Notes on Derivation Access Levels

- Protected and public members of base class are accessible in derived class even if derivation was private
- Private derivation prevents access to all inherited members by subclasses and clients of derived class
- Sensible choice: public derivation
  - If base class has public (protected) member, derived class is likely to keep member public (protected)
- Default case (no access specified): private

Guidelines on Access Level Definition

How should one define access levels?

- Generally, member functions are public and data members are protected
- Protected constructors useful for abstract classes

  (e.g., to initialize inherited portion of concrete subclasses)
- Use protected functions to let subclasses access and modify private information

Guidelines on Access Level Definition

How should one define access levels? (continued)

- Private members useful when writer of base class is different from writer of derived class
  - Hide representation details of base class from programmers of derived classes
- Access levels in class derivation: Public derivation (access level to inherited identifiers not changed) probably your choice most frequently

Examples of Access Levels

```
class Base {
 private:
  int x;
 protected:
  int y;
 public:
  int z; }
} ;
```

```
class Derived : protected Base {
public:
  int w;
  void mem1() {
    x++; // Error, no access
    y++; // OK
  } ;
```

```
int client() {
  Base a;
  Derived b;

  a.z++;        // OK, z public in Base
  b.z++;        // Error, z protected in Derived
  b.w++;  … } // OK, w public in Derived
```


More Examples

not covered

- Derived class does not get public access to inherited members

```
class Base {
 protected:
   int x_, y_;      ... } ;

class Derived : public Base {
 public:
   void mem1(Base bs, Derived der) {
     ...
   der.x_++; // OK: der is in Derived
   bs.x_++;  // Error: No public access to Base from Derived
   x_++;     // OK: receiver is in Derived
     ... }
 ... } ;
```


Base vs. Derived Class Access

not covered

- An inconsistency between base class access and derived class access?
  - Recall that class gets public access to members of instances (suppose x_ and y_ private in Point)

```
double Point::distance(Point& p1) {
double x_diff = x_ - p1.x_ ;
double y_diff = y_ - p1.y_ ;
return sqrt(x_diff*x_diff + y_diff*y_diff) ;
}
```

- Not true for instances of base classes
  - See example on previous slide

Derived Class Scope

- Problem: Derived class requires direct visibility (without scope operator) over identifiers defined in base class
- Solution: Convention on derived class scope:
  - Derived class definition assumes that derived class is nested within each of its base classes; however,
  - Derived class identifier is still assumed to be defined at file-scope (unless derived class happens to be nested in another class)

Consequences of Nesting Rule

- Clients of derived class can refer to derived class without using the scope operator, e.g.,

  `Derived derInstance;`

- Clients of derived class can send base-class messages to instances of derived class directly (identifier scope resolution starts from derived class context), e.g.,

  `derInstance.baseMethod();`

More Consequences

- Derived class member functions do not need scope operator to refer to nonprivate inherited members
- Multiple inheritance can create conflicts

Examples of Derived Class Scope

```
class Base {
  protected:
    int x_, y_;
    ... } ;
```

```
class Derived : public Base {
  public:
    void mem1() {
      ...
      x_++ ;          // OK(no scope operator)
      Base::x_++;  // same as above
      ... }
    } ;
```

```
void client () {
  Base bs;                         // OK, Base defined at file scope
  Derived der;                     // OK, Derived defined at file scope also
  ... } ;
```


More on Derived Class Scope

- Derived class can redefine members inherited from base class
- No conflict because identifiers are defined in different scopes (base class vs. derived class)
- Data members: Two data members with the same identifier will be in each instance of derived class
  - Unqualified references in derived class will get noninherited member (e.g., x or D::x)
  - Use scope operator with base class name to get inherited member (e.g., B::x)

More on Derived Class Scope

- Member functions: Derived class can redefine inherited member function with or without same argument signature as inherited member
  - Either way, function definition in derived class **overrides (hides), does not overload**, inherited function definition
  - Function defined in base class accessible in derived class through scope operator (just like data members)

More Examples of Derived Scope

```
class Base {
  protected:
    int x_;        // Base::x_
  public:
    void mem1(char*);
    ... }
```

```
class Derived : public Base {
  protected:
    int x_;        // Derived::x_
  public:
    void mem1(double)
    ... }
```

```
void client () {
 Base bs, *pbs;
 Derived der;
 bs.mem1(''hello'');   // OK, Base::mem1() invoked
 der.mem1(3.14);        // OK, Derived::mem1()
 der.mem1(''hi'');      // Error
 pbs = &der;
 pbs->mem1(3.14);  }   // Try guessing
```


Binding of Messages and Methods

- Static identifier resolution process in context of receiver's class leads to static binding of messages and methods
  - Not very OO, but efficient because done by compiler (Compiler will generate binary code that calls directly function obtained by identifier resolution process)
- Dynamic binding (message polymorphism) obtainable through virtual member functions (we'll see later)

Method Refinement

- Method refinement: Ability for a derived class to extend an inherited method
  - Use scope operator to invoke inherited member function from body of function in derived class
- If derived class wishes to overload inherited member function, it must define both overloadings (including a dummy redefinition of inherited member)

Examples of Method Refinements

```
class Base {
  public:
    void foo(int i);
    void mem1(char*);
    ... }
```

```
class Derived : public Base {
  public:

    void foo(int i) {
      Base::foo(i);
        ...    // refinements
      }

      // overloaded definitions
      void mem1(int);
      void mem1(char* s)
        { Base::mem1(s); }
      …
}
```


Friends and Inheritance

- Friends of a derived class have same access privileges as class' s own member functions
  - In particular, no access to base class's private members
- Friendship is not inherited
  - If class Base has been declared a friend by C, Derived is not necessarily a friend of C (must be declared as friend by C also)
  - If Base declares a friend (function or class) F, F not automatically a friend of Derived

Static Data Members and Inheritance

- Static data members defined by base class still accessible to derived class (if not private)
  - In general, one copy shared by base and derived classes
  - If derived needs its own copy, it redeclares static data member (identifier resolution by scope rules and, if necessary, scope operator)

Constructors and Inheritance

- Rule 1: Constructors are not inherited (up to C++03)
  - Avoid possibility of errors caused by failure to properly initialize noninherited portion of derived class instance
- Rule 2: Inherited portions of a derived class instance should be initialized by base class constructors (remember information hiding?)
  - Use initialization list of derived class constructor to invoke base class constructors with appropriate argument signature

Constructors and Inheritance

- Rule 2: (continued)
  - Initialization of inherited portion of derived class is similar to initialization of embedded instances for composite classes
  - Syntax: Initializer uses base class name followed by appropriate argument list, rather than member name (as with initializers of data members)

```
Point3D::Point3D (int i, int j, int k)
    :  Point(i, j), z_(k)  {  }
```

Order of Execution of Constructors

- Rule 3 — Portions of a constructor executed in this order to initialize derived instance:
  1. Constructors of base classes, in the order in which they appear in the derivation list (not the order in which they appear in initialization list)
     - If base class has a base class too, invocation done recursively
  2. Constructors of embedded classes, in the order in which embedded members are declared in definition of derived class (not the initialization list)
  3. Constructor body

Example of Execution Order

```
class Person  {
  protected:
    String name_;
    Date dob_;
    long  SSN_;
    …
} ;
```

```
class Student : public Person  {
  protected:
    CourseList grades_;
    Person advisor_;
    double GPA_;
    …
} ;
```

```
Student::Student (char* x, CourseList& y, Date& z,
              Person& u, long v, double w)
   :  grades_(y),          //  Call CourseList(const CourseList&)
      Person(x, v, z),     //  Call Person(char*, long, Date&)
      advisor_(u)          //  Call Person(const Person&)
   {GPA_ = w;}
```


Missing Initializers

- Rule 4: If initializer missing, default constructor for corresponding base class executed instead
  - Similar to case of embedded class constructor
  - Like before, this is probably wasteful and possibly erroneous (i.e., if base or embedded class does not have a default constructor)
- Don't forget to include const data members and reference data members in initialization list of derived class (as well as initializers for base classes and embedded classes)

Example of Constructor Definitions

Class definition containing plausible constructor declarations for class Point3D

```
// definition of class Point3D
class Point3D : public Point {
  public:
    Point3D() ;                          // default constructor
    Point3D(int, int, int) ;
    Point3D(const Point&) ;              // conversion from Point
    int z() { return z_; }
    void set_z(int new_z) { z_ = new_z; }
    double distance(Point3D& pt);
  protected:
    int z_; }
```


Example of Constructor Definitions

Plausible constructor definitions for class Point3D

```
// default constructor
Point3D::Point3D() {
   // implicit invocation of Point::Point() sets x_ and y_ to 0
   z_ = 0; }

// same effects, but  invocation of Point::Point() is explicit
Point3D::Point3D() : Point()   {
   z_ = 0; }

Point3D::Point3D(int i, int j, int k)  : Point(i, j), z_(k)
  { }         // did it all in initialization list
```


More Examples

- Conversion constructor from class Point

```
//  Note Point::Point(int, int) not inherited in Point3D
Point3D::Point3D(const Point& pt) :
  Point(pt)                    // copy constructor of Point invoked
  { z_ = 0; }
```

- Note that copy constructor is not needed

```
//  Default copy constructor for Point3D
//  Note implicit conversion of pt3D from Point3D to Point
Point3D::Point3D(const Point3D& pt3D) :
  Point(pt3D)                  // copy constructor of Point invoked
  { z_ = pt3D.z_; }
```

Destructors in Derived Classes

- Destructor of a derived class is responsible for releasing resources allocated by derived class constructor
- If a derived class has no destructor, compiler generates automatically a destructor that:
  1. Invokes destructors of embedded (class) members, and
  2. Invokes destructors of base classes

Order of Execution of Destructors

- Order of execution of destructors is order of constructor execution reversed, i.e.,
  1. Body is executed first
  2. Destructors of embedded classes are next (implicit invocation)
  3. Destructors of base classes are last (implicit invocation)

Compiler-Generated Constructors

- If derived class has no constructor, compiler generates a default default constructor, which invokes default constructors of base classes and embedded classes
- Compiler-generated copy constructor:
  1. Invoke copy constructor of base classes
  2. Invoke copy constructor of embedded members
  3. Do memberwise initialization of remaining members:

```
Derived::Derived(const Derived& d)
  : Base1(d), Base2(d), ...,  mem1_(d.mem1_), mem2_(d.mem2_), ...
  { }
```


Summary

Summary of constructors and destructors in derived classes

- Constructors work from the "inside out"
- Destructors work from the "outside in"
- Initialization lists useful for initializing inherited portion of derived class instance, embedded class instances, const data members, reference data members, and (if wanted) other members

Overloading Assignment Operator

- Probably appropriate whenever derived class defines copy constructor (e.g., because there are pointer data members and referents must be copied)
- Caveat (assignment operator refinement):

  Beware of infinite recursion when invoking operator=() defined in base class (i.e., to handle assignment of inherited portion)

```
Derived& operator=(const Derived& rhs) {
  …
  *this = rhs ;                  // infinite recursion!
}
```


Overloading Assignment Operator

- Needs explicit cast on receiver or use of scope operator to invoke operator=() in base class
- These two solutions will work:

```
// You can cast receiver explicitly to Base class reference.
// rhs converted implicitly (Derived& → Base& rhs conversion is automatic).
(Base&) *this = rhs;

// Alternatively, use long syntax invocation of Base class's assignment
// operator.
// Again rhs argument in call is converted implicitly to Base&.
this->Base::operator=(rhs);
```

Overloading Assignment Operator

- Beware: These two solutions will **not** work:

```
// casting rhs argument still tries to invoke Derived::operator=()
*this = (Base&) rhs;

// value cast creates temp object, (receiver unchanged)
(Base) *this = rhs;
```

- Example: operator=() for class Student (a Person subclass)

```
Student& Student::operator=(const Student& rhs) {
  if (this == &rhs) return *this;
  else {
    Person::operator=(rhs);        // initialize inherited portion
    GPA_ = rhs.GPA_;
    …                              // other noninherited members
    return *this; }
}
```


In C++ a Value Cast Causes a Conversion

Identifier and Object Polymorphism

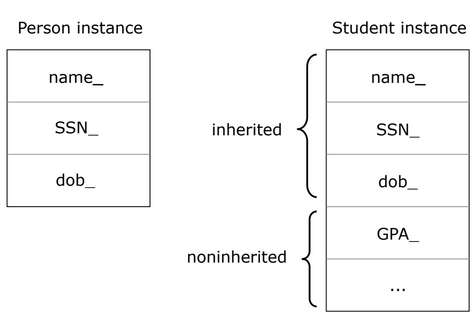
- Identifier polymorphism: Identifier of class C (value, reference or pointer) can be bound to instances of C and of any subclass of C
  - Identifier of class Person can be bound to instances of Person, Employee, Manager, etc.
- Object polymorphism: Instance of class D can be viewed as an instance of any superclass of D
  - Manager instance can also be viewed as an Employee or a Person instance

Identifier and Object Polymorphism

- Binding subclass instance to superclass identifier can cause storage problems (for value identifiers)
  - C++ allocates only memory needed by superclass instance

```
Person p;       // A person instance
Student s;      // A student instance
p = s;          // This is legal, however…
```


Diagram of Subclass Instance



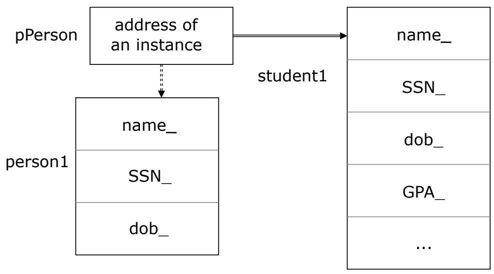
Storage of Derived Instances

- Pointer identifiers OK, because pointers have fixed size (i.e., regardless of size of their referents)

```
Person person1;
Person* pPerson = &person1;
Student student1;

// OK, now pPerson points to a Student instance
pPerson = &student1;
```

Diagram of Previous Assignment

pPerson = &student1

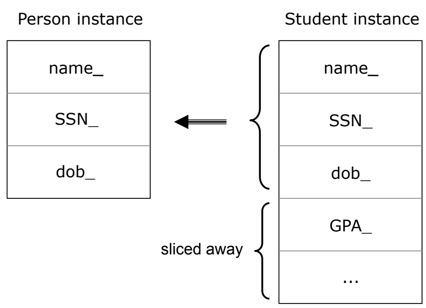


Storage of Derived Instances

- Value identifiers are a problem because subclass instance is usually larger than superclass instance
- **Slicing:** The loss of information caused by assigning derived instance to base class identifier
  - Noninherited data members of derived instance ignored by assignment

```
Person p;          // A person instance
Student s;         // A student instance
p = s;             // GPA_ and other noninherited members
                   // do not take part in assignment
```


Diagram of Previous Assignment



Static vs. Dynamic Identifier Class

- Static class of an identifier: The class used when the identifier is declared or defined
  - Determined at compile-time
- Dynamic class of identifier: Class of identifier's referent
  - Can change at run-time

```
Student s;          // Assume Student is Person subclass
Professor p;        // Professor is also Person subclass
Person* pPerson;    // Static class of pPerson is Person
if (…)
  pPerson = &s;     // Dynamic class of pPerson is Student
else pPerson = &p; // Dynamic class of pPerson is Professor
```


Semantics of Assignment

- Assignment semantics different for value (or reference in this case) identifiers vs. pointer identifiers
  - Value and reference identifiers: Copying semantics
  - Pointers: Pointer semantics

```
Person person1, *pPerson;
Student student1, *pStudent;
pStudent = new Student;
pPerson = pStudent;    // pointer semantics (no copying)
person1 = student1;    // copying semantics with slicing
```


More on Assignment

- Identifier cannot be bound to superclass instances
  - Absence of noninherited members would cause inconsistencies in assignment

```
Person person1;
Student student1;
student1 = person1;   // Error!  person1 does not have GPA_
                      // and other noninherited members
```

- C++ jargon: Conversion from derived class to base class is automatic; opposite is not

Casting to Base Classes

- Generally, can use a derived class object whenever base class instance is needed
  - Instance initialization, argument passing, rhs in assignment, etc.
- Again, different behavior for value and reference (or pointer) identifiers
  - Value identifiers: Create new instance with copy constructor of base class
  - Pointer identifiers: No copying, original object is used

Example of Implicit Conversion

- Value identifier:

```
// a file-scope function computing distance between Points
double pointDistance(Point p1, Point p2) ;
…
// definition of arguments in call below
Point3D instPoint3D;
Point instPoint;
…
// second argument copied using copy constructor of Point class
pointDistance(instPoint, instPoint3D);    // example of call
…
```


Example of Implicit Cast

- Reference identifier (pointers are similar):

```
// Suppose that now pointDistance use reference parameters
// of class Point
double pointDistance(Point& p1, Point& p2) ;
…
// invocation example–Same arguments as earlier!!!!
Point3D instPoint3D;
Point instPoint;
…
// But now work with original instances -- no copying here
pointDistance(instPoint, instPoint3D);
…
```


More on Casts and Conversions

- Implicit cast can be used to resolve calls to overloaded functions
  - Treated as an automatic conversion, not an exact match
  - Therefore, applied before user-defined conversions
- On explicit conversions: Outward conversion operators are inherited, but inward conversions are not (because they use constructors)

Downward Cast

- Downward cast: Conversion of base class identifier to derived class
  - Safe only when identifier is reference/pointer holding address of derived class instance
  - Useful to make protocol of derived class accessible through base class identifier (e.g., for non-virtual or noninherited methods)

```
Derived der;
Base *pbs=&der;
// suppose memfun1() defined in Derived, not in Base
((Derived*)pbs)->memfun1();
```


Downward Cast

- Problem with down cast: Unsafe
  - What if referent is not an instance of subclass given?
- (Slightly) safer way: dynamic_cast<> operator
  - Run-time check for appropriate target class
  - Conversion done only when appropriate
  - Return null pointer if incorrect
  - Argument is pointer or reference identifier of polymorphic class (see later definition)
  - Target type must be **polymorphic** (see next)

Dynamic Cast Operator

- Syntax of dynamic cast operator:

  dynamic_cast<Derived*>(baseClassPtr)

- Example of use:

```
Student student1, *pStudent;
Person *pPerson = &student1 ;
…
pStudent = dynamic_cast<Student*>(pPerson) ;
if (pStudent != NULL) {
  pStudent->student_method();
  …
}
```


More on Dynamic Cast

- If conversion argument is a reference, exception bad_cast is raised
  - There is nothing equivalent to **null** pointer for reference identifiers

    dynamic_cast<Derived&>(instBaseRef)

Run-Time Type Information

- Basic approach of C++ is not to have type information at run-time (similar to ANSI C)
- Sometimes run-time information is necessary, e.g.,
  - to implement message polymorphism
  - to find out what kind of instance is bound to a base class identifier, e.g., to do something like

```
DisplayObject* pDisplayList;
if (pDisplayList->widget()->type() == TextBox)
  (pDisplayList->widget()->content()).italicizeText() ;
  else …
```


Run-Time Type Information System

- Idea: Make information kept by run-time system to implement message polymorphism available to programmers
- Run-Type Type Information (RTTI) system:
  1. type_info class — Each instance holds information about classes
  2. typeid operator — A message to find out class of an instance (an instance of type_info returned)

RTTI: Class Definition

Class type_info:

- Only publicly available member functions are equality test, a before() function, and a name()
  - before() provides ordering (probably irrelevant)
  - name() allows class name to be printed as a string
- Constructors are private to prevent programmers from corrupting RTTI system

RTTI: Class Definition

Definition of class type_info:

```
class type_info {
public:
  virtual ~type_info() ;                  // virtual destructors explained later
  bool operator==(const type_info&) const ;
  bool operator!=(const type_info&) const;
  bool before(const type_info&) const;         // ignore it
  const char* name() const ;                   // print name of class
private:
   type_info(const type_info&) ;               // private copy constructor
   type_info& operator=(const type_info&) ;
                                               // private assignment op
   …
} ;
```

RTTI: Operator Definition

- Operator typeid can be applied to expressions and type identifiers, including class names
- When used with identifier, identifier type must belong to polymorphic class
- Logical declaration (an operator, not a function, though):

  `const type_info& typeid(expression) ;`
- Usable also with identifiers of built-in types

Examples of Use of RTTI System

- Use both type_info class and typeid operator:

```
DisplayObject* pDisplayList;
if (typeid(*pDisplayList) == typeid(TextBox);
   (pDisplayList->content).italicizeText;
 …
typeid(pDisplayList) == typeid(TextBox*)          // false
typeid(pDisplayList) == typeid(DisplayObject*)    // true
typeid(*pDisplayList) == typeid(TextBox)          // possibly true
typeid(*pDisplayList) == typeid(DisplayObject)    // probably false
```


Virtual Member Functions

- Support message polymorphism
  - When member function called, actual function executed is determined by dynamic class of receiver
  - Override traditional identifier resolution process when searching for a method
  - Default is non-virtual because of C++'s obsession with performance
- More object-oriented, but also more expensive than default (non-virtual) case
  - Indiscriminate use of virtual member functions can slow down program execution many times over

Syntax of Virtual Member Functions

- Declare a member function virtual in root of class sub-hierarchy where dynamic binding is desired
  - Virtual declaration should be repeated in subclasses, although this is not necessary
- **Polymorphic class**: One that has at least one virtual member function
  - implementation of polymorphic classes is different from other classes
  - every instance has **vptr**; every class has **vtbl**

Example of Virtual Function

- Suppose Person class has subclasses Employee and Student; Employee has subclasses Manager and Individual

```
class Person {
virtual void print(ostream& s);
…  } ;

class Employee : public Person {
virtual void print(ostream& s);
…  } ;

class Student : public Person {
virtual void print(ostream& s);
…  } ;
```


More on Virtual Functions

- Virtual method must be defined in ancestor class with same argument signature as derived class method (except return type Base& or Base* is matched by Derived& and Derived*)
  - method overriding rule still in effect
  - parameter lists must match exactly

Invocation of Virtual Functions

- Send message to derived class instance through reference or pointer identifier of base class, e.g.,

```
Manager m1;
Person *pPerson=&m1;
...
// dynamically bound call invokes Manager::print()
pPerson->print(cout);

// Beware: pPerson's referent will automatically go out of scope when
// function containing this code returns (m1's referent is allocated
// automatically unless definitions of m1 and pPerson are at file scope)
```


Static Binding of Virtual Functions

- Virtual member functions still statically bound in 4 cases
  1. Receiver is value identifier (not reference or pointer identifier)
     - slicing happens:

```
Manager m1;
Person p1;
...
p1 = m1;              // slicing!! p1 is still a Person, not a Manager
...
//statically bound call invokes Person::print()
p1.print(cout);
```


Static Binding of Virtual Functions

2. Caller uses scope operator indicating explicitly class where search should be started
   - Suppose class Person has subclass Student, and Student has subclass Undergraduate

```
Undergraduate u1;
Person *pPerson=&u1;
...
//statically bound call invokes Person::mem1(), if Student does not
// redefine mem1()
pPerson->Student::mem1();
```


Static Binding of Virtual Functions

3. Derived class's function has argument signature different from base class's virtual function
   - In this case base class method is overridden
   - Example: suppose Student redefines mem1() with an integer argument

```
Student s1;
Person *pPerson=&s1;
...
// next call is statically bound
pPerson->mem1();       //  Fine, Person::mem1() called
pPerson->mem1(10);     //  Error, argument mismatch
```


Static Binding of Virtual Functions

- Argument signatures of inherited and derived class functions must match in order for dynamic binding to occur
  - Only exception: If inherited (base class) function returns either a Base& or a Base* object, then derived class is allowed to change that to a Derived& or a Derived* type

```
//  Function emergency_contact() is dynamically bound
virtual Person& Person::emergency_contact() ;
virtual Student& Student::emergency_contact();
```


Static Binding of Virtual Functions

4. Virtual function invoked by base class constructor or destructor
   - Noninherited members of derived class may not be available yet (constructor) or no longer be available (destructor)

Static Binding of Virtual Functions

- Example: Suppose constructor for Student class invokes Person constructor, which then invokes virtual member function print()

```
Student::Student(...)  : Person(...)
    { }

Person::Person(...) : ...
    { ...
    // statically bound call invokes Person::print(), although
    // receiver is Student instance
     print(cout);
     ... }
```


Implementation of Polymorphic Classes

- Support for method search at run-time
  1. vtbl — Every polymorphic class has a table that associates virtual function names with address of corresponding code
  2. vptr — Every instance of a polymorphic class has a pointer to the corresponding vtbl
- vtbl of class C has address of all virtual functions that instances of C can execute (inherited and noninherited)

Implementation of Polymorphic Classes

- Search for virtual member function:
  1. Follow vptr to correct vtbl
  2. Compute vtbl entry with correct function identifier (as an offset into the table, usually a single h/w instruction)
  3. Jump to corresponding address (offset)

Advantages and disadvantages of C++

- Big advantage of method search in C++ wrt Smalltalk: **Efficiency!**
  — Never follow a superclass pointer
- Disadvantage of C++: vtbl "hard-wires" Smalltalk's method search and hierarchy search
  — After changing a class, must recompile class hierarchy below that class to recompute their vtbls

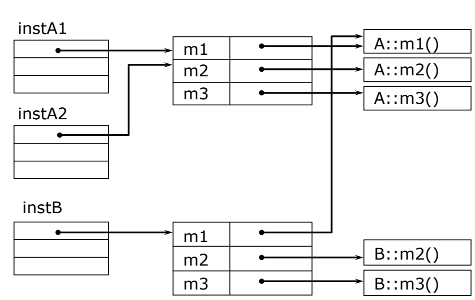
Example of Implementation

- Consider the following two classes:

```
class A {
 virtual void m1();
 virtual void m2();
 virtual void m3();
 …
 }

class B : public A {
 virtual void m2();
 virtual void m3();
 …
 }
```


Diagram of Class Example

- vptr and vtbl for classes A and B



Generating Virtual Tables

- Compiler generates vtbl for polymorphic class using following data available at compile-time:
  - All the methods that instances of class can execute (inherited and locally defined)
  - Whether each of above methods is virtual or not
- Non-polymorphic classes do not have vtbl (and instances do not have vptr)

More on Virtual Functions

- First class that declares a function virtual in class hierarchy must either define function or defer function definition (pure virtual function)
  - Cannot use inherited definition
- Static member functions and file-scope functions cannot be dynamically bound
  - Invocations contained in static and file-scope functions can be dynamically bound under usual circumstances

More on Virtual Functions

- Virtual member functions sometimes called methods (e.g., by Bjarne Stroustrup)
- Subclass need not redefine inherited virtual functions
  - Suppose class A has subclass B, B has subclass C
  - Also, suppose A and B define (virtual) mem1(), but C does not

```
C  instC;
A  *ptrA = &instC;
ptrA->mem1();        // invokes B::mem1()
```


Access Specifications

- Access specifications for virtual functions always defined by static class of receiver (even for virtual functions)
  - Access check is done by compiler
  - Good practice: Keep same access level in derived classes for virtual functions as in ancestors
  - Don't let access depend on dynamic class of receiver
  - If virtual function public in base class but protected in derived class, derived class instance responds to message only if sent through base class identifier

Example of Inconsistent Access

- Suppose print() public in Person, protected in Student

```
void client() {
Student s1, *pStudent;
Person *pPerson = &s1;
 …
pStudent = &s1;
 …
// Now *pStudent, *pPerson and s1 are the same instance
pPerson->print(cout);      // OK, print() public in Person
s1.print(cout);            // Error!  print() protected in Student
pStudent->print(cout);     // Same error
  …
 }
```


Compiling Calls to Virtual Functions

- Compiler does the following:
  1. Check that function identifier is visible and accessible at point of call
  2. If yes, generate code that will:
     A. Dereference vptr of receiver
     B. Compute offset of vtbl entry for function involved
     C. Dereference pointer to function code
     D. Transfer control to function code (along with other actions typically involved in a call)

Problem with Destructors

- Suppose derived class instance bound to base class identifier must be deleted
  - Normally, this would invoke base class destructor

```
Base *pbs = new Derived(…) ;
…
delete pbs;
```

- Problem: Base destructor does not know about noninherited members of Derived
  - Problem if one such member is pointer whose referent must be deallocated

Virtual Destructors

- Virtual destructor: Base class declares destructor to be virtual
- Derived class's virtual destructor is invoked when Derived instance is deleted through base class pointer
  - If Person::~Person declared virtual, destructor for Student executed here:

```
Person *pPerson= new Student(…) ;
 ...
// virtual Student::~Student() executed because it is virtual
delete pPerson;
```


Constructors and Virtual Copying

- Constructors are non-virtual
- This is usually OK
  - Derived class constructor will invoke base class constructor to initialize inherited portion of derived class instance
  - Exception: Copy constructor (called by base class ptr)

```
Student s1;
Person *ptr1 = &s1;
Person *ptr2 = new Person(*ptr1);
 // Person::Person(const Person&) invoked
```

  - No easy way around this

Simulating Virtual Copying

Two basic kinds of approaches

1. Explicit type checks (using RTTI extension)

   Could be implemented by overloading **new** operator in superclass

2. Define virtual member function copy()

   Similar approach to Smalltalk's, except class is hard-coded when creating new instance in C++

   Compare C++ with Smalltalk's "soft" approach:

   Smalltalk: **newInstance := oldInstance class new.**

   C++: **newInstance := new HardCodedClassNameHere;**

Virtual copy() method

- Common superclass defines virtual function copy() with no arguments
  - Method returns a copy of receiver, allocated dynamically (by exclusion)
  - Copy constructor can be used to initialize new instance from receiver
- Each subclass refines copy(), by creating instance of that subclass
  - Again, subclass's copy constructor can initialize new instance

Coding copy() method

- Base class's code:

```
class Base {
  // copy constructor, if needed for deep copying
  Base(const Base& bs)  {
    …
   }
  // virtual copy method
  virtual Base* copy() {
    // create new instance, invoke copy constructor implicitly, and return it
    // But don't do this if Base class is abstract; in this case do nothing
    // or define copy() as pure virtual function.
    Base* aBase = new Base(*this);
    return aBase;
  } …
}
```


Coding copy() method

- Derived class's code:

```
class Derived1 : public Base {
  // copy constructor, if needed for deep copying
  Derived1(const Derived1& der) : Base(der) {
   …
  }
  // virtual copy method
 virtual Derived1* copy() {
    // create new instance, invoke copy constructor implicitly, and return it
    Derived1* pDer = new Derived1(*this);
    return pDer;
  }
…
}
```

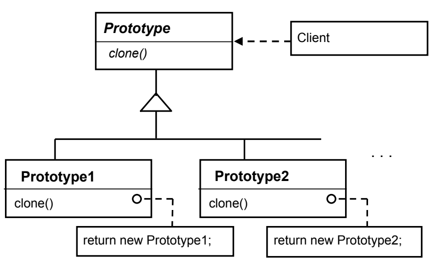

Caveats on virtual copying

- Client responsible for deletions
  - Beware of asymmetry with allocations: Instance allocation and deletion now done by different units!
  - Clients must deallocate objects **whose allocation they cannot see** (because done in virtual copy() method)
  - Example of use:

```
class client(const Collection& aCollection[]) {
  Base* pbs1 = new Derived1(…);
  Base* pbs2 = pbs1->copy();    // this is ok, beware of invisible allocation
   …
  // Don't forget to delete pbs2!
  }
```


The Prototype pattern

- A design pattern in pattern system of the Gang of 4
- Design pattern = A reusable piece of design



Using Virtual Functions

not covered

- When should a method be virtual?
  - Generally, whenever a class may have subclasses
  - Performance penalty (vptrs and vtbls)
  - However, inclusion polymorphism, program extensibility lost with static binding
  - Nature of application also plays important role (performance critical?)
  - Problem: Try to predict whether a class will have subclasses or not (will the program be extended?)

Using Virtual Functions

not covered

- The risk in refining a non-virtual method:
  - If derived instance accessed via base class pointer or reference identifier, base class method executed
  - Suppose now print() is non-virtual in Person

```
Student s1;
Student* pStudent = &s1;
Person* pPerson = &s1;
 …
// pPerson and pStudent denote same instance
pPerson->print(cout);        // invokes Person::print()
pStudent->print(cout);       // invokes Student::print()
```


Abstract Classes in C++

- Classes with no instances
- Classes with at least one deferred method
  - Deferred method defines argument signature, but defers method definition to subclasses
- Purpose: to define a common protocol for a set of concrete subclasses
- Pure virtual function: Deferred method in C++

Syntax of Abstract Classes

- Pure virtual member function is declared, but not defined in abstract class
- Function prototype in definition of abstract class is followed by **=0** syntax

```
class Collection {
//  An iterator for collections of integers
//  Note that iterator is a pointer to a function on integers
virtual Collection& repeat(void (*iterator) (int))=0;
…
} ;
```


More on Abstract Classes

- Any class containing a pure virtual function is abstract, as well as all subclasses that do not give a definition (executable body) for this function
  – First class that overrides method is concrete (unless it has other pure virtual functions)
- Attempts to create instances will always result in errors, whether by static, automatic or dynamic allocation
  – **Value parameters and data members also disallowed**

More on Abstract Classes

- Constructors should be protected
  – Useful to initialize inherited portions of concrete subclasses (assuming abstract class has data members requiring initialization)
- Use abstract class only to create subclasses
  – Very useful in statically typed language to define common protocol for set of concrete subclasses
  – Dynamically bound message must be defined in common ancestor of subclasses receiving message

Example of an Abstract Class

```
class DisplayObject  {
  public:
   virtual void rotate(int degrees)=0;
   virtual void resize(double percent)=0;
   virtual void remove(...)=0;
    …
  protected:
   Window* pDisplayWindow;
    …
} ;
```


Mixin class

- An entirely abstract class
  – All methods are deferred (pure virtual member functions)
  – No concrete member functions (whether inherited or locally-defined)
  – No data members (inherited or local)
- Similar to Java interfaces
- Recommended way of using multiple inheritance
  – Avoid the three disadvantages of multiple inheritance

Multiple Inheritance in C++

- C++ supports full multiple inheritance scheme
- Syntax: Specify multiple base classes in derivation list of subclass (the order of base classes matters)
  – Each base preceded by access level specification, defaulting to private

```
class DisplaySquare:  public DisplayObject, public Square {
  …
  // now DisplaySquare has all the members of
  // DisplayObject and Square
  …
}
```

More on Multiple Inheritance

- Order of base classes in derivation list determines structure of derived instances and order of execution of constructors; otherwise, irrelevant
  - Example of DisplaySquare instance:

DisplaySquare instance:	
	DisplayObject members
	Square members
	Noninherited members

Details on multiple inheritance

- Conversion from derived instance to any of its base classes is well defined
  - DisplaySquare instance can be automatically converted to either DisplayObject or Square
  - Conversion to other base classes than first one involves adding offset to instance's starting address (Done automatically by compiler)

Automatic Conversions

- Example of automatic conversion to Square:

```
void foobar(Square sq) ;
…
DisplaySquare aDispSq ;
foobar(aDispSq);            // aDispSq converted to Square
```


Conversions Involving Pointers

- Problem: Start of embedded instance target of conversion may not coincide with start of derived instance

```
void foobar(Square* pSquare) ;
…
DisplaySquare aDispSq ;
foobar(&aDispSq);    // &aDispSq converted to Square*; however,
                     // Square instance does not start where aDispSq starts
                     // (preceded by DisplayObject instance)
```


Conversions Involving Pointers

- Problem is solved by compiler adding appropriate offset to address of derived class instance
- This is transparent to programmer (thankfully!)

```
void foobar(Square* pSquare) ;
…
DisplaySquare aDispSq
foobar(&aDispSq);    // compiler adds size of DisplayObject to address
                     // of aDispSq upon call resulting pointer has
                     // address of Square instance embedded in aDispSq
```


Inheriting Virtual Functions

- Inherited functions must work with base class instance embedded in receiver
  - Pass appropriate portion of receiver to function
  - Similar problem to previous conversion; this time solution is complicated by presence of vptr and vtbl
- If multiple base classes are polymorphic, derived class has multiple vtbls, one for each base class
  - Each embedded base class instance has a vptr to corresponding vtbl
- Store appropriate offset in each vtbl

Issues in Multiple Inheritance

- C++ must address three usual problems of multiple inheritance and some additional problems (e.g., conversions to base classes, polymorphic base classes):
  - Name ambiguity
  - Method search
  - Common ancestor problem (aka repeated inheritance or diamond problem)

Problem 1: Name Conflicts

- Suppose that multiple base classes define data members or member functions with the same name
  - A classical problem in multiple inheritance
  - Solution is different depending on whether data member or member function is involved
- Data members: Must use scope operator with member identifier to resolve ambiguity
- Member functions: Member functions defined indifferent base classes overload each other
  - Calls may be disambiguated by argument lists

Examples of Name Conflicts

```
class A {
  int x;
  …
  void foo(int) ;
  int bar(int);
} ;
```

```
class B {
  char* x;
  …
  void foo() ;
  void bar(int);
} ;
```

```
class C : public A,  public  B   {
  void mem1() {
    x++;
    A::x++;
    foo(3);
    bar(3);
    B::bar(3);
   ...
```


Examples of Name Conflicts

```
class A {
  int x;
  …
  void foo(int) ;
  int bar(int) ;
} ;
```

```
class B {
  char* x;
  …
  void foo() ;
  void bar(int) ;
} ;
```

```
class C : public A,  public  B   {
  void mem1() {
    x++ ;          // Error: x reference ambiguous
    A::x++ ;       //  OK
    foo(3) ;       //  OK, A::foo() is invoked
    bar(3) ;       //  Overloaded call is ambiguous
    B::bar(3);     //  OK, B::bar() is invoked
   ...
```


Name Conflicts and Clients

- Clients must also be aware of name conflicts when using instances of derived classes with multiple base classes

```
C* pInstC = new C(…);
pInstC->bar(10);               // ambiguous call
pInstC->B::bar(10);            // OK, although ugly
```

- Additional issue: Dynamic binding of functions inherited from multiple base classes
  - Dynamic binding happens only if function declared virtual by all base classes
  - Example: bar() virtual in C if virtual in both A and B

Name Conflicts and Clients

- Derived class can avoid violation of information hiding by redefining and renaming functions inherited from multiple base classes

```
class C : public A, public B {
  void barFromA(int x) {  A::bar(x);  }
  void barFromB(int x) {  B::bar(x);  }
  …  } ;
```

- Now clients need not be aware of C's derivation list
  - Same is true for subclasses of C
  - Explicit use of scope operator precludes dynamic binding, though

Common Ancestor Problem in C++

- Recall common ancestor problem occurs when derived class has two (or more) base classes which in turn inherit from common base class

```
class A {
  … } ;

class B :  public A {
 … } ;

class C  :  public A {
  … } ;

class D : public B, public C {
… } ;
```


Common Ancestor Problem in C++

- C++ approach to common ancestor problem:
  - Default case: multiple copies of common ancestor instance in subclass instances
  - Programmer can choose to have single copy

Instance of D:

Instance of A
Noninherited B members
Instance of A
Noninherited C members
Noninherited D members

Default Case

- References to data members inherited from common ancestor must be disambiguated by scope operator
  - Two copies of common ancestor's instance means two copies of each data member

Instance of D:

Instance of A
Noninherited B members
Instance of A
Noninherited C members
Noninherited D members

Code Example

```
class  A   {
 protected:
    int x;
 …
} ;

class B : public A {
 …
} ;

class C : public A {
 …
} ;
```

```
class  D : public B, public C  {
   void mem1  {
   x++ ;
   B::x++ ;
   C::x++ ;
   A::x++ ;
     …
   }
  …
} ;
```


Code Example

```
class  A   {
 protected:
    int x;
 …
} ;

class B : public A {
 …
} ;

class C : public A {
 …
} ;
```

```
class  D : public B, public C  {
   void mem1  {
   x++ ;      // error, which x?
   B::x++ ; // OK
   C::x++ ; // OK, other x
   A::x++ ; // Error again
     …
   }
  …
} ;
```


More on Default Case

- Conversions from derived class to common ancestor class become ambiguous
  - Which copy of repeated ancestor should be used?
  - Code examples:

```
void foobar(A instA) ;
…
D instD;
D* ptrD = &instD;
foobar(instD);          // ambiguous conversion
(A*)ptrD;               // again ambiguous
```

More on Default Case

- Ambiguity applies only to nonstatic members
  - There is only one copy of static data members defined in common ancestor
- Ambiguity applies also to functions inherited from common ancestor, not overridden in intermediate base classes or derived class

Code Example

```
class  A   {
 public:
    int mem1();
  …
} ;

class B : public A {
  …
} ;

class C : public A {
  …
} ;
```

```
class  D : public B, public C  {
 …
} ;

void client () {
  D* ptrD = new D(…) ;
  …
  ptrD->mem1();
  ptrD->B::mem1();
  ptrD->C::mem1();
  …
} ;
```


Code Example

```
class  A   {
 public:
    int mem1();
  …
} ;

class B : public A {
  …
} ;

class C : public A {
  …
} ;
```

```
class  D : public B, public C  {
 …
} ;

void client () {
  D* ptrD = new D(…) ;
  …
  ptrD->mem1();      // error
  ptrD->B::mem1(); // OK
  ptrD->C::mem1(); // OK
  …
} ;
```

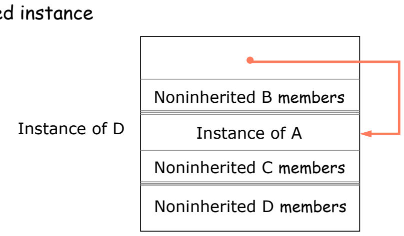

Single Copy in Multiple Inheritance

- Virtual Base Class: Declaring intermediate base classes virtual causes one copy only
  - Syntax: Keyword virtual precedes access level specification (if any) and ancestor class name in header of base class definitions
  - Our code example:

```
class B:  virtual public A { …
};
class C:  virtual public A { …
};
```

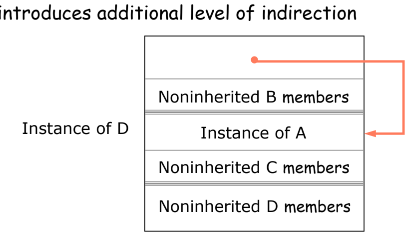

Storage Structure

- One copy of common ancestor stored in derived instances
  - Other copies are replaced by a pointer to the only embedded instance



Notes on Virtual Base Classes

- Additional pointer dereferencing handled automatically by compiler
  - Pointer introduces additional level of indirection



More on Virtual Base Classes

- Efficiency is adversely affected
- If some base classes are virtual, but others are not, all virtual base classes share one ancestor copy, other (nonvirtual) base classes each have their own copy
- Now, unqualified references to nonstatic data members inherited from common ancestor are fine
  - One copy of each data member exists; ambiguity is eliminated

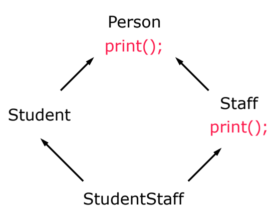
Code Example

```
class  A  {
  int mem1();
  int x;

 …
} ;

class B : virtual public A
{ …
} ;

class C : virtual public A
{ …
} ;
```

```
class  D : public B, public C  {
  …
} ;

void client () {
  D* ptrD = new D(…) ;
  …
  ptrD->mem1();          // OK
  ptrD->B::mem1();       // OK
  ptrD->C::mem1();       // OK
  x++;                   // OK
  A::x++;                // OK
  …
} ;
```


An Additional Issue

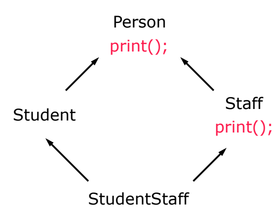
- **Asymmetric refinement** of ancestor method
  - Suppose base class refines virtual method defined in ancestor, but another base class does not



Question:
What print() would
StudentStaff use?

Dominance

- **Dominance:** Ancestor definition is overridden by definition in one of base classes
  - If both base classes define print(), ambiguous



Answer:
Staff::Print(), because
of dominance!

Code Examples Involving Dominance

- Assume Person, Student, Staff, StudentStaff classes

```
StudentStaff  john ;
Person* ptrPerson = &john;
Student* ptrStudent = &john;
Staff* ptrStaff = &john;
StudentStaff* ptrJohn = &john;
…
ptrPerson->print();          // invokes Staff::print()
ptrStudent->print();         // invokes Staff::print(), ehm!
ptrStaff->print();           // invokes Staff::print()
ptrJohn->print();            // invokes Staff::print()
ptrJohn->Person::print();    // invokes Person::print()
ptrJohn->Student::print();   // invokes Person::print()
```


Constructors and Virtual Derivation

- Single instance of ancestor class embedded in derived class instance must be initialized once
- Suppose classes B and C are derived from class A; class D has virtual base classes B and C
- Constructor for D must invoke constructors for B and C in its initialization list

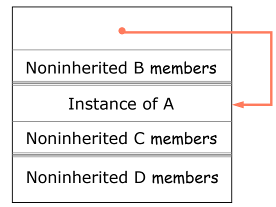
Constructors and Virtual Derivation

- Problem: Each of the constructors for B and C will attempt to invoke constructor for A; however, there is only one instance of A embedded in a D instance

```
D::D(…) : B(…), C(…)   {…}   ;
B::B(…) : A(…) {…}   ;
C::C(…) : A(…) {…}   ;
```


Constructors and Virtual Derivation

- Recall structure of a D instance



Constructor Definition

- Special rules for building instance of derived class with multiple virtual base classes
  1. Invoke constructor of common ancestor explicitly in initialization list of derived class
  2. Compiler skips execution of ancestor constructor while executing constructors of virtual base classes

```
// Example of a constructor for derived class
D::D(…) :  A(…),   // invoke constructor of common ancestor
           B(…),   // this skips constructor of A class
           C(…)    // this skips constructor of A class
{…}        //  body of D constructor
```


Read Chapter 12 in Drake's book

http://gcc.gnu.org/projects/cxx0x.html

Thank you very much !

Enjoy your break !