The C++ Programming Language

Single and Multiple Inheritance in C++

Douglas C. Schmidt

www.cs.wustl.edu/~schmidt/
schmidt@cs.wustl.edu

Washington University, St. Louis
Background

- Object-oriented programming is often defined as the combination of Abstract Data Types (ADTs) with Inheritance and Dynamic Binding

- Each concept addresses a different aspect of system decomposition:
  1. ADTs decompose systems into two-dimensional grids of modules
     - Each module has public and private interfaces
  2. Inheritance decomposes systems into three-dimensional hierarchies of modules
     - Inheritance relationships form a “lattice”
  3. Dynamic binding enhances inheritance
     - e.g., defer implementation decisions until late in the design phase or even until run-time!
Data Abstraction vs. Inheritance

DATA ABSTRACTION
(2 DIMENSIONAL)

INHERITANCE
(3 DIMENSIONAL)
Motivation for Inheritance

• Inheritance allows you to write code to handle certain cases and allows other developers to write code that handles more specialized cases, while your code continues to work.

• Inheritance partitions a system architecture into semi-disjoint components that are related hierarchically.

• Therefore, we may be able to modify and/or reuse sections of the inheritance hierarchy without disturbing existing code, e.g.,
  – Change sibling subtree interfaces
    * i.e., a consequence of inheritance
  – Change implementation of ancestors
    * i.e., a consequence of data abstraction
Inheritance Overview

- A type (called a *subclass* or *derived type*) can inherit the characteristics of another type(s) (called a *superclass* or *base type*)
  - The term *subclass* is equivalent to *derived type*

- A derived type acts just like the base type, except for an explicit list of:
  1. *Specializations*
     - Change implementations *without* changing the base class interface
       * Most useful when combined with dynamic binding
  2. *Generalizations/Extensions*
     - Add new operations or data to derived classes
Visualizing Inheritance
Types of Inheritance

- Inheritance comes in two forms, depending on number of \textit{parents} a subclass has

1. \textit{Single Inheritance} (SI)
   - Only one parent per derived class
   - Form an inheritance “tree”
   - SI requires a small amount of run-time overhead when used with dynamic binding
   - \textit{e.g.}, Smalltalk, Simula, Object Pascal

2. \textit{Multiple Inheritance} (MI)
   - More than one parent per derived class
   - Forms an inheritance “Directed Acyclic Graph” (DAG)
   - Compared with SI, MI adds additional run-time overhead (also involving dynamic binding)
   - \textit{e.g.}, C++, Eiffel, Flavors (a LISP dialect)
Inheritance Trees vs. Inheritance DAGs

**Inheritance Tree**

1. Derived 1
2. Base
3. Derived 2
4. Derived 4

**Inheritance DAG**

1. Base 1
2. Derived 1
3. Derived 3
4. Derived 4
Inheritance Benefits

1. Increase reuse and software quality
   - Programmers reuse the base classes instead of writing new classes
     - Integrates *black-box* and *white-box* reuse by allowing extensibility and modification without changing existing code
   - Using well-tested base classes helps reduce bugs in applications that use them
   - Reduce object code size

2. Enhance extensibility and comprehensibility
   - Helps support more flexible and extensible architectures (along with dynamic binding)
     - *i.e.*, supports the open/closed principle
   - Often useful for modeling and classifying hierarchically-related domains
Inheritance Liabilities

1. May create deep and/or wide hierarchies that are hard to understand and navigate without class browser tools

2. May decrease performance slightly
   - *i.e.*, when combined with *multiple inheritance* and *dynamic binding*

3. Without dynamic binding, inheritance has only limited utility
   - Likewise, dynamic binding is almost totally useless without inheritance

4. Brittle hierarchies, which may impose dependencies upon ancestor names
Inheritance in C++

- Deriving a class involves an extension to the C++ class declaration syntax

- The class head is modified to allow a "derivation list" consisting of base classes

- e.g.,

```cpp
class Foo { /* ... */};
class Bar : public Foo { /* ... */};
class Foo : public Foo, public Bar { /* ... */};
```
Key Properties of C++

Inheritance

• The base/derived class relationship is explicitly recognized in C++ by predefined standard conversions

  – *i.e.*, a pointer to a derived class may always be assigned to a pointer to a base class that was inherited *publically*

  * But not vice versa…

• When combined with dynamic binding, this special relationship between inherited class types promotes a type-secure, *polymorphic* style of programming

  – *i.e.*, the programmer need not know the actual type of a class at compile-time

  – Note, C++ is not truly polymorphic

    * *i.e.*, operations are not applicable to objects that don’t contain definitions of these operations at some point in their inheritance hierarchy
Simple Screen Class

• The following code is used as the base class:

```cpp
class Screen {
public:
    Screen (int = 8, int = 40, char = ’ ’);
    ~Screen (void);
    short height (void) const { return this->height_; }
    short width (void) const { return this->width_; }
    void height (short h) { this->height_ = h; }
    void width (short w) { this->width_ = w; }
    Screen &forward (void);
    Screen &up (void);
    Screen &down (void);
    Screen &home (void);
    Screen &bottom (void);
    Screen &display (void);
    Screen &copy (const Screen &);
    // ... 
private:
    short height_, width_;
    char *screen_, *cur_pos_; 
};
```
Subclassing from Screen

- class Screen can be a public base class of class Window

- e.g.,

```cpp
class Window : public Screen {
public:
    Window (const Point &, int rows = 24,
            int columns = 80,
            char default_char = ' ');
    void set_foreground_color (Color &);
    void set_background_color (Color &);
    void resize (int height, int width);
    // ...
private:
    Point center_;
    Color foreground_;  // ...
    Color background_; 
    // ...
};
```
Multiple Levels of Derivation

- A derived class can itself form the basis for further derivation, \textit{e.g.,}

```cpp
class Menu : public Window {
public:
    void set_label (const char *l);
    Menu (const Point & , int rows = 24,
          int columns = 80,
          char default_char = ' ');
    // ...
private:
    char *label_;
    // ...
};
```

- \textit{class Menu} inherits data and methods from both \textit{Window} and \textit{Screen}

  \textit{--- i.e., sizeof (Menu) >= sizeof (Window) >= sizeof (Screen)}
The Screen Inheritance Hierarchy

- Screen/Window/Menu hierarchy
A pointer to a derived class can be assigned to a pointer to any of its *public* base classes without requiring an explicit cast:

```plaintext
Menu m; Window &w = m; Screen *ps1 = &w;
Screen *ps2 = &m;
```
Using the Screen Hierarchy

- e.g.,

```cpp
class Screen { public: virtual void dump (ostream &); = 0 }
class Window : public Screen {
    public: virtual void dump (ostream &);
};
class Menu : public Window {
    public: virtual void dump (ostream &);
};
// stand-alone function
void dump_image (Screen *s, ostream &o) {
    // Some processing omitted
    s->dump (o);
    // (*s->vptr[1]) (s, o));
}

Screen s; Window w; Menu m;
Bit_Vector bv;

// OK: Window is a kind of Screen
dump_image (&w, cout);
// OK: Menu is a kind of Screen
dump_image (&m, cout);
// OK: argument types match exactly
dump_image (&s, cout);
// Error: Bit_Vector is not a kind of Screen!
dump_image (&bv, cout);
```
Using Inheritance for Specialization

- A derived class specializes a base class by adding new, more specific state variables and methods
  - Method use the same interface, even though they are implemented differently
    * i.e., “overridden”
  - Note, there is an important distinction between overriding, hiding, and overloading...

- A variant of this is used in the template method pattern
  - i.e., behavior of the base class relies on functionality supplied by the derived class
  - This is directly supported in C++ via abstract base classes and pure virtual functions
Specialization Example

- Inheritance may be used to obtain the features of one data type in another closely related data type

- For example, class Date represents an arbitrary Date:

  ```cpp
class Date {
  public:
    Date (int m, int d, int y);
    virtual void print (ostream &s) const;
    // ...
  private:
    int month_, day_, year_;
};
```

- Class Birthday derives from Date, adding a name field representing the person’s birthday, e.g.,

  ```cpp
class Birthday : public Date {
  public:
    Birthday (const char *n, int m, int d, int y)
      : Date (m, d, y), person_ (strdup (n)) {}
    ~Birthday (void) { free (person_); }
    virtual void print (ostream &s) const;
    // ...
  private:
    const char *person_;
};
```
Implementation and Use-case

- Birthday::print could print the person’s name as well as the date, e.g.,

```cpp
void Birthday::print (ostream &s) const {
    s << this->person_ << " was born on ";
    Date::print (s);
    s << "\n";
}
```

- e.g.,

```cpp
const Date july_4th (7, 4, 1993);
Birthday my_birthday ("Douglas C. Schmidt", 7, 18, 1962);

july_4th.print (cerr);
// july 4th, 1993
my_birthday.print (cout);
// Douglas C. Schmidt was born on july 18th, 1962

Date *dp = &my_birthday;
dp->print (cerr);
// ??? what gets printed ???
// (*dp->vptr[1])(dp, cerr);
```
Alternatives to Specialization

- Note that we could also use *object composition* instead of *inheritance* for this example, *e.g.*,

```cpp
class Birthday {
public:
    Birthday(char *n, int m, int d, int y):
        date_(m, d, y), person_(n) {}  // same as before
private:
    Date date_;  // same as before
    char *person_;  // same as before
};
```

- However, in this case we would not be able to utilize the dynamic binding facilities for base classes and derived classes

  - *e.g.*,

    ```cpp
    Date *dp = &my_birthday;
    // ERROR, Birthday is not a subclass of date!
    ```
    
    - While this does not necessarily affect reusability, it does affect extensibility...
Using Inheritance for Extension/Generalization

- Derived classes add *state variables* and/or *operations* to the *properties* and *operations* associated with the base class
  - Note, the interface is generally widened!
  - Data member and method access privileges may also be modified

- Extension/generalization is often used to facilitate reuse of *implementations*, rather than *interface*
  - However, it is not always necessary or correct to export interfaces from a base class to derived classes
Extension/Generalization

Example

• Using class Vector as a private base class for derived class Stack

```cpp
class Stack : private Vector { /* ... */ };
```

• In this case, Vector’s `operator[]` may be reused as an implementation for the Stack `push` and `pop` methods

  Note that using private inheritance ensures that `operator[]` does not show up in the interface for class Stack!

• Often, a better approach in this case is to use a composition/Has-A rather than a descendant/Is-A relationship...
Vector Interface

- Using class Vector as a base class for a derived class such as class Checked_Vector or class Ada_Vector
  
  - One can define a Vector class that implements an unchecked, uninitialized array of elements of type T

- e.g., /* File Vector.h (incomplete wrt initialization and assignment) */

  // Bare-bones implementation, fast but not safe
template <class T>
class Vector {
public:
    Vector (size_t s);
    ~Vector (void);
    size_t size (void) const;
    T &operator[] (size_t index);

private:
    T *buf_;
    size_t size_;
};
Vector Implementation

- e.g.,

```cpp
template <class T>
Vector<T>::Vector (size_t s): size_ (s), buf_ (new T[s]) {} 

template <class T>
Vector<T>::~Vector (void) { delete [] this->buf_; } 

template <class T> size_t 
Vector<T>::size (void) const { return this->size_; } 

template <class T> T &
Vector<T>::operator[] (size_t i) { return this->buf_[i]; } 

int main (void) { 
    Vector<int> v (10); 
    int i = v[v.size ()]; // oops, out of range! 
    // destructor automatically called 
} 
```
Benefits of Inheritance

- Inheritance enables modification and/or extension of ADTs without changing the original source code
  
  - e.g., someone may want a variation on the basic Vector abstraction:
    
    1. A vector whose bounds are checked on every reference
    
    2. Allow vectors to have lower bounds other than 0
    
    3. Other vector variants are possible too…
      
      * e.g., automatically-resizing vectors, initialized vectors, etc.

- This is done by defining new derived classes that inherit the characteristics of the Vector base class
  
  - Note that inheritance also allows code to be shared
## Checked_Vector Interface

- The following is a subclass of Vector that allows run-time range checking:

- /* File Checked-Vector.h (incomplete wrt initialization and assignment) */

```cpp
struct RANGE_ERROR {
    "range_error" (size_t index);
    // ...
};

template <class T>
class Checked_Vector : public Vector<T> {
    public:
        Checked_Vector (size_t s);
        T &operator[] (size_t i) throw (RANGE_ERROR);
        // Vector::size () inherited from base class Vector.
    protected:
        bool in_range (size_t i) const;
    private:
        typedef Vector<T> inherited;
};
```
Implementation of
Checked_Vector

- e.g.,

```cpp
template <class T> bool
Checked_Vector<T>::in_range (size_t i) const {
    return i < this->size ();
}

template <class T>
Checked_Vector<T>::Checked_Vector (size_t s)
    : inherited (s) {}

template <class T> T &
Checked_Vector<T>::operator[] (size_t i)
    throw (RANGE_ERROR)
{
    if (this->in_range (i))
        return (*(inherited *) this)[i];
        // return BASE::operator[] (i);
    else
        throw RANGE_ERROR (i);
}
```
**Checked Vector Use-case**

- *e.g.*,

```cpp
#include "Checked_Vector.h"

typedef Checked_Vector<int> CV_INT;

int foo (int size)
{
    try
    {
        CV_INT cv (size);
        int i = cv[cv.size ()]; // Error detected!
            // exception raised…
        // Call base class destructor
    }
    catch (RANGE_ERROR)
    {
        /* …*/
    }
}
```
Design Tip

- Note, dealing with parent and base classes
  - It is often useful to write derived classes that do not encode the names of their direct parent class or base class in any of the method bodies
  - Here’s one way to do this systematically:

```cpp
class Base {
public:
    int foo (void);
};
class Derived_1 : public Base {
    typedef Base inherited;
public:
    int foo (void) { inherited::foo (); } // first
};
class Derived_2 : public Derived_1 {
    typedef Derived_1 inherited;
public:
    int foo (void) { // second
        inherited::foo ();
    }
};
```

- This scheme obviously doesn’t work as transparently for multiple inheritance...
Ada_Vector Interface

- The following is an Ada Vector example, where we can have array bounds start at something other than zero

- /* File ada_vector.h (still incomplete wrt initialization and assignment....) */

```c
#include "vector.h"

// Ada Vectors are also range checked!

template <class T>
class Ada_Vector : private Checked_Vector<T> {
public:
    Ada_Vector (size_t l, size_t h);
    T &operator ()(size_t i) throw (RANGE_ERROR)
        inherited::size; // explicitly extend visibility

private:
    typedef Checked_Vector<T> inherited;
    size_t lo_bnd_;
};
```
Ada_Vector Implementation

• e.g., class Ada_Vector (cont’d)

```cpp
template <class T>
Ada_Vector<T>::Ada_Vector (size_t lo, size_t hi)
   : inherited (hi - lo + 1), lo_bnd_ (lo) {}

template <class T> T &
Ada_Vector<T>::operator ()(size_t i)
    throw (RANGE_ERROR) {
    if (this->in_range (i - this->lo_bnd_))
        return Vector<T>::operator[] (i - this->lo_bnd_);
        // or Vector<T> &self = *(Vector<T> *) this;
        // self[i - this->lo_bnd_];
    else
        throw RANGE_ERROR (i);
    }
```
Ada_Vector Use-case

- Example Ada Vector Usage (File main.C)

```c
#include <iostream.h>
#include <stdlib.h>
#include "ada_vector.h"

int main (int argc, char *argv[]) {
  try {
    size_t lower = ::atoi (argv[1]);
    size_t upper = ::atoi (argv[2]);
    Ada_Vector<int> ada_vec (lower, upper);

    ada_vec (lower) = 0;
    for (size_t i = lower + 1; i <= ada_vec.size (); i++)
      ada_vec (i) = ada_vec (i - 1) + 1;

    // Run-time error, index out of range
    ada_vec (upper + 1) = 100;

    // Vector destructor called when
    // ada_vec goes out of scope
  }
  catch (RANGE_ERROR) { /* ... */ }
}
```

34
Memory Layout

• Memory layouts in derived classes are created by concatenating memory from the base class(es)
  
  – e.g., // from the cfront-generated .c file

    ```
    struct Vector {
        T *buf__6Vector;
        size_t size__6Vector;
    };
    struct Checked_Vector {
        T *buf__6Vector;
        size_t size__6Vector;
    };
    struct Ada_Vector {
        T *buf__6Vector; // Vector
        size_t size__6Vector; // part
        size_t lo_bnd__10Ada_Vector; // Ada_Vector
    };
    ```

• The derived class constructor calls the base constructor in the “base initialization section,” i.e.,

  ```
  Ada_Vector<T>::Ada_Vector (size_t lo, size_t hi) :
      inherited (hi – lo + 1), lo_bnd_ (lo) {}
  ```
Base Class Constructor

- Constructors are called from the “bottom up”

- Destructors are called from the “top down”

- e.g.,

```c
/* Vector constructor */
struct Vector * __ct__6VectorFi (struct Vector *__0this, size_t __0s) {
    if (__0this || (__0this = __nw__FUi (sizeof (struct Vector))))
        ((__0this->size__6Vector = __0s),
            (__0this->buf__6Vector = __nw__FUi ((sizeof (int)) * __0s)));
    return __0this;
}
```
Derived Class Constructors

- e.g.,

```c
/* Checked_Vector constructor */
struct Checked_Vector *__ct__14Checked_VectorFi (  
    struct Checked_Vector *__0this, size_t __0s) {
    if (__0this || (__0this =  
        __nw__FUi (sizeof (struct Checked_Vector))))
        __0this = __ct__6VectorFi (__0this, __0s);
    return __0this;
}
/* Ada_Vector constructor */
struct Ada_Vector *__ct__10Ada_VectorFiT1 (  
    struct Ada_Vector *__0this, size_t __0lo, size_t __0hi) {
    if (__0this || (__0this =  
        __nw__FUi (sizeof (struct Ada_Vector))))
        if ((__0this = __ct__14Checked_VectorFi (__0this,  
            __0hi - __0lo + 1)))
            __0this->lo_bnd__10Ada_Vector = __0lo;
    return __0this;
}
```
Destructor

- Note, destructors, constructors, and assignment operators are *not* inherited

- However, they may be called automatically were necessary, *e.g.*, 

  ```c
  char __dt__6VectorFv (  
    struct Vector *__0this, int __0__free) {  
      if (__0this) {  
        __dl__FPv ((__char *) __0this->buf__6Vector);  
        if (__0this)  
          if (__0__free & 1)  
            __dl__FPv ((__char *) __0this);  
      }  
    }
  ```
Describing Relationships Between Classes

• **Consumer/Composition/Aggregation**

  – A class is a consumer of another class when it makes use of the other class’s services, as defined in its interface

  * For example, a Stack implementation could rely on an array for its implementation and thus be a consumer of the Array class

  – Consumers are used to describe a **Has-A relationship**

• **Descendant/Inheritance/Specialization**

  – A class is a descendant of one or more other classes when it is designed as an extension or specialization of these classes. This is the notion of inheritance

  – Descendants are used to describe an **Is-A relationship**
Has-A vs. Is-A Relationships

CONSUMER RELATIONSHIP

Stack → Vector

DESCENDANT RELATIONSHIP

Vector → Checked Vector → Ada Vector
**Interface vs. Implementation**

**Inheritance**

- Class inheritance can be used in two primary ways:

  1. *Interface inheritance*: a method of creating a subtype of an existing class for purposes of setting up dynamic binding, *e.g.*,

     - Circle is a subclass of Shape (*i.e.*, Is-A relation)

     - A Birthday is a subclass of Date

  2. *Implementation inheritance*: a method of reusing an implementation to create a new class type

     - *e.g.*, a **class** Stack that inherits from **class** Vector. A Stack is not really a subtype or specialization of Vector

     - In this case, inheritance makes implementation easier, since there is no need to rewrite and debug existing code.

     * This is called “using inheritance for reuse”

     * *i.e.*, a pseudo-Has-A relation
The Dangers of Implementation

Inheritance

- Using inheritance for reuse may sometimes be a dangerous misuse of the technique

  - Operations that are valid for the base type may not apply to the derived type at all

    * e.g., performing an subscript operation on a stack is a meaningless and potentially harmful operation

      ```
      class Stack : public Vector {
          // ...
      }
      Stack s;
      s[10] = 20; // could be big trouble!
      ```

  - In C++, the use of a private base class minimizes the dangers

    * i.e., if a class is derived “private,” it is illegal to assign the address of a derived object to a pointer to a base object

- On the other hand, a consumer/Has-A relation might be more appropriate...
Private vs Public vs Protected
Derivation

- Access control specifiers (i.e., public, private, protected) are also meaningful in the context of inheritance.

- In the following examples:
  - <...> represents actual (omitted) code
  - [... ] is implicit.

- Note, all the examples work for both data members and methods.
Public Derivation

• *e.g. ,*

```cpp
class A {
public:
    <public A>
protected:
    <protected A>
private:
    <private A>
};

class B : public A {
public:
    [public A]
    <public B>
protected:
    [protected A]
    <protected B>
private:
    <private B>
};
```
Private Derivation

• *e.g.*,

```cpp
class A {
    public:
        <public A>
    private:
        <private A>
    protected:
        <protected A>
};

class B : private A { // also class B : A
    public:
        <public B>
    protected:
        <protected B>
    private:
        [public A]
        [protected A]
        <private B>
};
```
Protected Derivation

• e.g.,

```cpp
class A {
public:
    <public A>
protected:
    <protected A>
private:
    <private A>
};

class B : protected A {
public:
    <public B>
protected:
    [protected A]
    [public A]
    <protected B>
private:
    <private B>
};
```
Summary of Access Rights

- The following table describes the access rights of inherited methods

  - The vertical axis represents the access rights of the methods of base class

  - The horizontal access represents the mode of inheritance

```
<table>
<thead>
<tr>
<th>INHERITANCE</th>
<th>ACCESS</th>
</tr>
</thead>
<tbody>
<tr>
<td>M A</td>
<td>public</td>
</tr>
<tr>
<td>E C</td>
<td>----------</td>
</tr>
<tr>
<td>M C</td>
<td>protected</td>
</tr>
<tr>
<td>B E</td>
<td>----------</td>
</tr>
<tr>
<td>E S</td>
<td>private</td>
</tr>
<tr>
<td>R S</td>
<td>----------</td>
</tr>
</tbody>
</table>

  p  p  p
  u  r  r
  b  o  i
  l  t  v
```

- Note that the resulting access is always the most restrictive of the two
Other Uses of Access Control
Specifiers

- Selectively redefine visibility of individual methods from base classes that are derived *privately*

```cpp
class A {
public:
    int f();
    int g;
    ...
private:
    int p;
};

class B : private A {
public:
    A::f;  // Make public
protected:
    A::g;  // Make protected
};
```
Common Errors with Access Control Specifiers

• It is an error to “increase” the access of an inherited method in a derived class
  
  – e.g., you may not say:

  ```
  class B : private A {
     // nor protected nor public!
     public:
     A::p_; // ERROR!
  };
  ```

• It is also an error to derive *publically* and then try to selectively decrease the visibility of base class methods in the derived class
  
  – e.g., you may not say:

  ```
  class B : public A {
     private:
     A::f; // ERROR!
  };
  ```
General Rules for Access Control

Specifiers

- Private methods of the base class are not accessible to a derived class (unless the derived class is a friend of the base class)

- If the subclass is derived *publically* then:
  1. Public methods of the base class are accessible to the derived class
  2. Protected methods of the base class are accessible to derived classes and friends only
Caveats

• Using protected methods weakens the data hiding mechanism since changes to the base class implementation might affect all derived classes. *e.g.*,

```cpp
class Vector {
    public:
        //...
    protected:
        // allow derived classes direct access
        T *buf_;
        size_t size_;
};
class Ada_Vector : public Vector {
    public:
        T &operator[](size_t i) {
            return this->buf_[i];
        }
        // Note the strong dependency on the name buf_
};
```

• However, performance and design reasons may dictate use of the protected access control specifier

  — Note, inline functions often reduces the need for these efficiency hacks...
Overview of Multiple Inheritance in C++

- C++ allows *multiple inheritance*
  
  - *i.e.*, a class can be simultaneously derived from two or more base classes

  - *e.g.,*

    ```
    class X { /* .... */ };
    class Y : public X { /* .... */ };
    class Z : public X { /* .... */ };
    class YZ : public Y, public Z { /* .... */ };
    ```

  - Derived classes Y, Z, and YZ inherit the data members and methods from their respective base classes
Multiple Inheritance Illustrated

NON-VIRTUAL INHERITANCE

VIRTUAL INHERITANCE
Liabilities of Multiple Inheritance

- A base class may legally appear only once in a derivation list, e.g.,
  - `class Two_Vector : public Vector, public Vector // ERROR!`

- However, a base class may appear multiple times within a derivation hierarchy
  - e.g., `class YZ contains two instances of class X`

- This leads to two problems with multiple inheritance:
  1. It gives rise to a form of method and data member ambiguity
     - Explicitly qualified names and additional methods are used to resolve this
  2. It also may cause unnecessary duplication of storage
     - “Virtual base classes” are used to resolve this
Motivation for Virtual Base Classes

- Consider a user who wants an `Init_Checked_Vector`:

```cpp
class Checked_Vector : public virtual Vector
{
    /* .... */
};
class Init_Vector : public virtual Vector
{
    /* .... */
};
class Init_Checked_Vector :
    public Checked_Vector, public Init_Vector
{
    /* .... */
};
```

- In this example, the `virtual` keyword, when applied to a base class, causes `Init_Checked_Vector` to get one `Vector` base class instead of two
Overview of Virtual Base Classes

- Virtual base classes allow class designers to specify that a base class will be shared among derived classes
  - No matter how often a virtual base class may occur in a derivation hierarchy, only “one” shared instance is generated when an object is instantiated
    * Under the hood, pointers are used in derived classes that contain virtual base classes

- Understanding and using virtual base classes correctly is a non-trivial task since you must plan in advance
  - Also, you must be aware when initializing sub-classes objects...

- However, virtual base classes are used to implement the client and server side of many implementations of CORBA distributed objects
Virtual Base Classes Illustrated

Non-Virtual Inheritance:

- Vector
  - Checked Vector
    - Init Checked Vector
      - Checked Vector

Virtual Inheritance:

- Vector
  - Checked Vector
  - Init Checked Vector
  - Checked Vector
Initializing Virtual Base Classes

- With C++, you must choose one of two methods to make constructors work correctly for virtual base classes:

1. You need to either supply a constructor in a virtual base class that takes no arguments (or has default arguments), e.g.,

   \[
   \text{Vector}::\text{Vector} \left( \text{size}_t \text{ size} = 100 \right); \quad // \text{has problems} \ldots
   \]

2. Or, you must make sure the most derived class calls the constructor for the virtual base class in its \textit{base initialization section}, e.g.,

   \[
   \text{Init\_Checked\_Vector} \left( \text{size}_t \text{ size}, \textbf{const} \ T \ &\text{init} \right):
   \begin{align*}
   &\text{Vector} \left( \text{size} \right), \text{Check\_Vector} \left( \text{size} \right), \\
   &\text{Init\_Vector} \left( \text{size}, \text{init} \right)
   \end{align*}
   \]
Vector Interface Revised

- The following example illustrates templates, multiple inheritance, and virtual base classes in C++

```cpp
#include <iostream.h>
#include <assert.h>

// A simple-minded Vector base class, no range checking, no initialization.

template <class T>
class Vector
{
    public:
        Vector (size_t s): size_ (s), buf_ (new T[s]) {}
        T &operator[] (size_t i) { return this->buf_[i]; }
        size_t size (void) const { return this->size_; }

    private:
        size_t size_;  
        T *buf_;  
};
```
Init_Vector Interface

- A simple extension to the Vector base class, that enables automagical vector initialization

\[
\text{template <class } T\text{>}
\]
\[
\text{class Init\_Vector : public virtual Vector}\langle T\rangle
\]
\[
\{
\text{public:}
\]
\[
\text{ Init\_Vector (size\_t size, const } T \&\text{init)}}
\]
\[
\text{ : Vector}\langle T\rangle \text{ (size) }
\]
\[
\{ 
\text{for (size\_t i = 0; i < this->size (); i++)}
\]
\[
\text{ (*this)[i] = init; }
\]
\[
\} // Inherits subscripting operator and size().
\];
**Checked Vector Interface**

- A simple extension to the Vector base class that provides range checked subscripting

```cpp
template <class T>
class Checked_Vector : public virtual Vector<T> {
public:
    Checked_Vector (size_t size): Vector<T> (size) {} 
    T &operator[] (size_t i) throw (RANGE_ERROR) {
        if (this->in_range (i))
            return *(inherited *) this)[i];
        else throw RANGE_ERROR (i);
    }
    // Inherits inherited::size.
private:
    typedef Vector<T> inherited;

    bool in_range (size_t i) const {
        return i < this->size ();
    }
};
```
Init_Checked_Vector Interface and Driver

• A simple multiple inheritance example that provides for both an initialized and range checked Vector

```cpp
template <class T>
class Init_Checked_Vector :
    public CheckedVectorizer<T>, public InitVectorizer<T> {
    public:
        Init_Checked_Vector(size_t size, const T &init):
            Vector<T>(size),
            InitVectorizer<T>(size, init),
            CheckedVectorizer<T>(size) {}
    // Inherits CheckedVectorizer::operator[]
};

• Driver program

```int main (int argc, char *argv[]) {
    try {
        size_t size = ::atoi (argv[1]);
        size_t init = ::atoi (argv[2]);
        Init_Checked_Vector<int> v (size, init);
        cout << "vector size = " << v.size ()
            << " , vector contents = ";
        for (size_t i = 0; i < v.size (); i++)
            cout << v[i];
        cout << "\n" << ++v[v.size () - 1] << "\n";
    }
    catch (RANGE_ERROR) { /* ... */ }
}
Multiple Inheritance Ambiguity

Consider the following:

```c
struct Base_1 { int foo (void); /* .... */ };  
struct Base_2 { int foo (void); /* .... */ };  
struct Derived : Base_1, Base_2 { /* .... */ };  
int main (void) {
    Derived d;
    d.foo (); // Error, ambiguous call to foo ()
}
```

There are two ways to fix this problem:

1. Explicitly qualify the call, by prefixing it with the name of the intended base class using the scope resolution operator, e.g.,

   ```
   d.Base_1::foo (); // or d.Base_2::foo ()
   ```

2. Add a new method foo to class Derived (similar to Eiffel’s renaming concept) e.g.,

```c
struct Derived : Base_1, Base_2 {
    int foo (void) {
        Base_1::foo (); // either, both
        Base_2::foo (); // or neither
    }
};
```
Summary

• Inheritance supports evolutionary, incremental development of reusable components by specializing and/or extending a general interface/implementation.

• Inheritance adds a new dimension to data abstraction, e.g.,

  – Classes (ADTs) support the expression of commonality where the general aspects of an application are encapsulated in a few base classes.

  – Inheritance supports the development of the application by extension and specialization without affecting existing code.

• Without browser support, navigating through complex inheritance hierarchies is difficult...