Single & Multiple Inheritance in C++

Douglas C. Schmidt

Professor
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt/

Department of EECS
Vanderbilt University
(615) 343-8197
Background

- Object-oriented programming is often defined as the combination of Abstract Data Types (ADTs) with Inheritance & 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 & 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 DIMENTIONAL)

INHERITANCE
(3 DIMENTIONAL)
Motivation for Inheritance

- Inheritance allows you to write code to handle certain cases & 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

Base

Derived

Derived

Derived

Derived

Derived

Derived
Types of Inheritance

- Inheritance comes in two forms, depending on number of parents a subclass has

1. 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
   - e.g., Smalltalk, Simula, Object Pascal

2. 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)
   - e.g., C++, Eiffel, Flavors (a LISP dialect)
Inheritance Trees vs. Inheritance DAGs
Inheritance Benefits

1. *Increase reuse & software quality*
   
   - Programmers reuse the base classes instead of writing new classes
   - Integrates *black-box* & *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 & comprehensibility*
   
   - Helps support more flexible & extensible architectures (along with dynamic binding)
   - *i.e.*, supports the open/closed principle
   - Often useful for modeling & classifying hierarchically-related domains
Inheritance Liabilities

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

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

3. Without dynamic binding, inheritance has limited utility, *i.e.*, can only be used for implementation inheritance
   - & dynamic binding is essentially pointless 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 Baz : 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 *publicly*
    - *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 *arbitrarily* polymorphic
    - *i.e.*, operations are not applicable to objects that don’t contain definitions of these operations at some point in their inheritance hierarchy
class Screen {
    /* Base class. */

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, e.g.,
  
  ```
  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_;
  };
  ```

- `class Menu` inherits data & methods from both `Window` & `Screen`, i.e.,
  
  ```
  sizeof (Menu) >= sizeof (Window) >= sizeof (Screen)
  ```
The Screen Inheritance 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:

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

class Screen {
    public: virtual void dump (ostream &); }

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);
    // translates to: (*s->vptr[1]) (s, o));
}
Using the Screen Hierarchy, (cont’d)

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 & methods
  - Method use the same interface, even though they are implemented differently
    - *i.e.*, “overridden”
  - Note, there is an important distinction between overriding, hiding, & 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 & 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, we can create a class `Date` that represents an arbitrary date:

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

- Class Birthday derives from Date, adding a name field, e.g.,

```cpp
#include <string>

class Birthday : public Date {
public:
    Birthday (const std::string &n, int m, int d, int y)
      : Date (m, d, y),
      person_ (n) { }
    virtual void print (ostream &s) const; // . . .
private:
    std::string person_; 
};
```
Implementation & 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 << std::endl;
}
```

```cpp
const Date july_4th (7, 4, 1993);
july_4th.print (cout); // july 4, 1993
Birthday igors_birthday ("Igor Stravinsky", 6, 17, 1882);
igors_birthday.print (cout);
// Igor Stravinsky was born on june 17, 1882
```

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

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

```cpp
class Birthday {
public:
    Birthday (const std::string &n, int m, int d, int y):
        date_ (m, d, y), person_ (n) {}
    // same as before

private:
    Date date_;  
    std::string person_;  
};
```
Alternatives to Specialization, (cont’d)

• However, in this case we would not be able to utilize the dynamic binding facilities for base classes & derived classes, e.g.,

```c
Date *dp = &igors_birthday;
// ERROR, Birthday is not a subclass of date!
```

• While this does not necessarily affect reusability, it does affect extensibility . . .
Another View of Inheritance

- Inheritance can also be viewed as a way to construct a hierarchy of types that are “incomplete” except for the leaves of the hierarchy
  - *e.g.*, you may wish to represent animals with an inheritance hierarchy. Let’s call the root class of this hierarchy “Animal”
  - Two classes derive from Animal: Vertebrate and Invertebrate
  - Vertebrate can be derived to Mammal, Reptile, Bird, Fish, *etc.*
  - Mammals can be derived into Rodents, Primates, Pachyderms, *etc.*
  - Primates can be derived into Apes, Sloths, Humans, *etc.*
  - Humans can be derived into Males & Females
  * We can then declare objects to represent specific males & females, *e.g.*, Bob, Ted, Carol, & Alice
Another View of Inheritance

- Advantages
  - Share code & set-up dynamic binding
  - Model & classify external objects with design & implementation
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 & method access privileges may also be modified

- Extension/generalization is often used to faciliate 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**:
  
  – **class Stack : private Vector { /* . . . */ };**

• In this case, Vector’s **operator[]** may be reused as an implementation for the Stack **push & pop** methods
  
  – Note that using private inheritance ensures that **operator[]** does not appear in **class Stack**’s interface!
Extension/Generalization Example, (cont’d)

- 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`

/* Bare-bones Vector implementation, fast but not safe:
   the array of elements is uninitialized, & ranges are
   not checked. Also, assignment is not supported. */

```cpp
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

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];
}
Vector Use-case

```c++
int main (int, char **) {
    Vector<int> v (10);


    int i = v[v.size ()]; // oops, out of range!

    // destructor automatically called

    return 0;
}
```
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 allows run-time range checking:

/* File Checked-Vector.h (incomplete wrt initialization & assignment) */
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:
    int in_range (size_t i) const;
private:
    typedef Vector<T> inherited;
};
Implementation of Checked_Vector

template <class T> int
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];
    // equivalent to: return inherited::operator[](i);
    else throw Range_Error (i); }
### Checked_Vector Use-case

```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)
  {
    /* . . . */
  }
}
```
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, our Bounded_Stack implementation relies on Array for its implementation, & thus is 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
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, because there is no need to rewrite & 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 & 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 & methods
Public Derivation

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>  
};
Protected Derivation

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

class B : private A {
    // same as class B : A
    public:
        <public B>
    protected:
        <protected B>
    private:
        [public A]
        [protected A]
        <private B>
};
Derived Class Access to Base Class Members

<table>
<thead>
<tr>
<th>Base Class Access Control</th>
<th>Inheritance mode</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>public</td>
</tr>
<tr>
<td>public</td>
<td>public</td>
</tr>
<tr>
<td>protected</td>
<td>protected</td>
</tr>
<tr>
<td>private</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td>protected</td>
</tr>
<tr>
<td></td>
<td>private</td>
</tr>
<tr>
<td></td>
<td>none</td>
</tr>
<tr>
<td></td>
<td>none</td>
</tr>
<tr>
<td></td>
<td>none</td>
</tr>
</tbody>
</table>

- The vertical axis represents the access rights specified in the base class.
- The horizontal access represents the mode of inheritance used by the derived class.
- 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 inherited from base classes. NOTE: the redifinition can only be to the visibility of the base class. Selective redefinition can only override the additional control imposed by inheritance.

```cpp
class A {
public:
   int f (void);
   int g_;...
private:
   int p_;
};
class B : private A {
public:
   A::f; // Make public
protected:
   A::g_; // Make protected
};
```
Common Issues with Access Control Specifiers

- It is an error to *increase* the access of an inherited method above the level given in the base class.

- Deriving *publicly* & then selectively decreasing the visibility of base class methods in the derived class should be used with caution: *removes* methods from the public interface at lower scopes in the inheritance hierarchy.

```cpp
// Error if p_ is // protected in A!
class B : private A {
    private:
        A::f; // hides A::f
    public:
        A::p_;  
};

class B : public A {
    private:
        A::f; // hides A::f
    public:
        A::p_;  
};
```
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 *publicly* 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 & friends only
Caveats

- Using protected methods weakens the data hiding mechanism because changes to the base class implementation might affect all derived classes.

- However, performance & design reasons may dictate use of the protected access control specifier
  - Note, inlining functions often reduces the need for these efficiency hacks.
Caveats, example

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 buf_
};
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, & YZ inherit the data members & methods from their respective base classes
Liabilities of Multiple Inheritance

- A base class may legally appear only once in a derivation list, *e.g.*, 

  ```cpp
class Two_Vect : public Vect, public Vect // 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 & data member ambiguity
     - Explicitly qualified names & 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 & using virtual base classes correctly is a non-trivial task because you must plan in advance
  - Also, you must be aware when initializing subclasses objects . . .

- However, virtual base classes are used to implement the client & server side of many implementations of CORBA distributed objects
Virtual Base Classes Illustrated
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.,
   \[
   \text{Vector::Vector (size_t size = 100); // not clean!}
   \]

2. Or, you must make sure the \textit{most derived class} calls the constructor for the virtual base class in its \textit{base initialization section}, e.g.,
   \[
   \text{Init_Checked_Vector (size_t size, const T &init):}
   \]
   \[
   \text{Vector (size), Check_Vector (size),}
   \]
   \[
   \text{Init_Vector (size, init)}
   \]
Virtual Base Class Initialization Example

```cpp
#include <iostream.h>
class Base {
 public:
   Base (int i) { cout << "Base::Base (" << i << ")" << endl; }
};

class Derived1 : public virtual Base {
 public:
   Derived1 (void) : Base (1) { cout << "Derived1 (void)" << endl; }
};

class Derived2 : public virtual Base {
 public:
   Derived2 (void) : Base (2) { cout << "Derived2 (void)" << endl; }
};
```
Virtual Base Class Initialization Example, (cont’d)

class Derived : public Derived1, public Derived2 {
public:

    // The Derived constructor _must_ call the Base
    // constructor explicitly, because Base doesn’t
    // have a default constructor.
    Derived (void) : Base (3) {
        cout << "Derived (void)" << endl;
    }
};
Virtual Base Class Initialization Example, (cont’d)

```c
int main (int, char **[])
{
    Base b (0);    // Direct instantiation of Base:
                    //     Base::Base (0)
    Derived1 d1;   // Instantiates Base via Derived1 ctor:
                    //     Base::Base (1)
    Derived2 d2;   // Instantiates Base via Derived2 ctor:
                    //     Base::Base (2)
    Derived d;     // Instantiates Base via Derived ctor:
                    //     Base::Base (3)
    return 0;
}
```
The following example illustrates templates, multiple inheritance, and virtual base classes in C++:

```cpp
#include <iostream.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

```cpp
template <class T>
class Init_Vector : public virtual Vector<T>
{
public:
    Init_Vector (size_t size, const T &init)
    : Vector<T> (size)
    {
        for (size_t i = 0; i < this->size (); i++)
            (*this)[i] = init;
    }
    // Inherits subscripting operator & size().
};
```
Checked_Vector Interface

- Extend Vector to provide 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)
        else throw Range_Error (i);
    }
    // Inherits inherited::size.
private:
    typedef Vector<T> inherited;
    int in_range (size_t i) const
    {
        return i < this->size ();
    }
};
```
Init_Checked_Vector Interface

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

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

```c
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 (int, char *[]) {
    Derived d;
    d.foo (); // Error, ambiguous call to foo ()
}
```
Multiple Inheritance Ambiguity, (cont’d)

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, \textit{e.g.},
   
   \begin{verbatim}
   d.Base_1::foo (); // or d.Base_2::foo ()
   \end{verbatim}

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

   \begin{verbatim}
   struct Derived : Base_1, Base_2 {
       int foo (void) {
           Base_1::foo (); // either, both
           Base_2::foo (); // or neither
       }
   };
   \end{verbatim}
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. *Tools can help.*