Lecture Material

- Multiple inheritance
- Class hierarchies

Multiple Inheritance

- A class can have more than one direct base class
- Consider a simulation in which concurrent activities are represented by a class *Task* and data gathering and display is achieved through a class *Displayed*. We can then define a class of simulated entities, class *Satellite*:

```
class Satellite : public Task, public Displayed {
 // ...
};
```

- The use of more than one immediate base class is usually called *multiple inheritance*. In contrast, having just one direct base class is called *single inheritance*.

Multiple Inheritance

- In addition to whatever operations are defined specifically for a *Satellite* , the union of operations on *Tasks* and *Displayed*s can be applied.

```
void f(Satellite& s)
{
 s.draw() ; // Displayed::draw()
 s.delay(10) ; // Task::delay()
 s.transmit() ; // Satellite::transmit()
}
```

- A *Satellite* can be passed to functions that expect a *Task* or a *Displayed*.

```
void highlight(Displayed*) ;
void suspend(Task*) ;
void g(Satellite* p)
{
 highlight(p) ; // pass a pointer to the Displayed part of the Satellite
 suspend(p) ; // pass a pointer to the Task part of the Satellite
}
```

Multiple Inheritance

- With multiple inheritance virtual functions work as usual.

```
class Task {
 // ...
 virtual void pending() = 0;
};
class Displayed {
 // ...
 virtual void draw() = 0;
};
class Satellite : public Task, public Displayed
{
 // ...
 void pending() ; // override Task::pending()
 void draw() ; // override Displayed::draw()
};
```

- This ensures that *Satellite::draw()* and *Satellite::pending()* will be called for a *Satellite* treated as a *Displayed* and a *Task*, respectively.

Ambiguity Resolution

- Two base classes may have member functions with the same name.

```
class Task {
 // ...
 virtual debug_info* get_debug() ;
};
class Displayed {
 // ...
 virtual debug_info* get_debug() ;
};
```

- When a *Satellite* is used, these functions must be disambiguated:

```
void f(Satellite* sp)
{
 debug_info* dip = sp->get_debug() ; // error: ambiguous
 dip = sp->Task::get_debug() ; // ok
 dip = sp->Displayed::get_debug() ; // ok
}
```

Ambiguity Resolution

- Explicit disambiguation is messy, so it is usually best to resolve such problems by defining a new function in the derived class:

```
class Satellite : public Task, public Displayed {
 // ...
 debug_info* get_debug() // override Task::get_debug()
                         // and Displayed::get_debug()
 {
   debug_info* dip1 = Task::get_debug() ;
   debug_info* dip2 = Displayed::get_debug() ;
   return dip1->merge(dip2) ;
 }
};
```

- This localizes the information about *Satellite*'s base classes. Because *Satellite::get_debug()* overrides the *get_debug()* functions from both of its base classes, *Satellite::get_debug()* is called wherever *get_debug()* is called for a *Satellite* object.

Ambiguity Resolution

- A qualified name *Telstar::draw()* can refer to a *draw()* declared either in *Telstar* or in one of its base classes.

```
class Telstar : public Satellite {
 // ...
 void draw()
 {
   draw() ; // oops!: recursive call
   Satellite::draw() ; // finds Displayed::draw
   Displayed::draw() ;
   Satellite::Displayed::draw() ; // redundant double qualification
 }
};
```

- If a *Satellite::draw()* doesn't resolve to a *draw()* declared in *Satellite*, the compiler recursively looks in its base classes; that is, it looks for *Task::draw()* and *Displayed::draw()*. If exactly one match is found, that name will be used. Otherwise, *Satellite::draw()* is either not found or is ambiguous.

Inheritance and *Using* Declarations

- Overload resolution is not applied across different class scopes. In particular, ambiguities between functions from different base classes are not resolved based on argument types.
- When combining essentially unrelated classes, such as *Task* and *Displayed* in the *Satellite* example, similarity in naming typically does not indicate a common purpose. When such name clashes occur, they often come as quite a surprise to the programmer.

```
class Task {
 // ...
 void debug(double p) ; // print info only if priority is lower than p
};
class Displayed {
 // ...
 void debug(int v) ; // the higher the 'v,'
                     // the more debug information is printed
};
class Satellite : public Task, public Displayed {
 // ...
};
void g(Satellite* p){
 p->debug(1) ; // error: ambiguous.
               // Displayed::debug(int) or Task::debug(double) ?
 p->Task::debug(1) ; // ok
 p->Displayed::debug(1) ; // ok
}
```

Inheritance and *Using* Declarations

- What if the use of the same name in different base classes was the result of a deliberate design decision and the user wanted selection based on the argument types? In that case, a *using* declaration can bring the functions into a common scope.

```
class A {
 public:
   int f(int) ;
   char f(char) ;
   // ...
};
class B {
 public:
   double f(double) ;
   // ...
};
class AB: public A, public B {
 public:
   using A::f;
   using B::f;
   char f(char) ; // hides A::f(char)
   AB f(AB) ;
};
void g(AB& ab)
{
 ab.f(1) ;        // A::f(int)
 ab.f(´a´) ;      // AB::f(char)
 ab.f(2.0) ;      // B::f(double)
 ab.f(ab) ;       // AB::f(AB)
}
```

Inheritance and *Using* Declarations

- *Using* declarations allow a programmer to compose a set of overloaded functions from base classes and the derived class. Functions declared in the derived class hide functions that would otherwise be available from a base. Virtual functions from bases can be overridden as ever.
- A *using* declaration in a class definition must refer to members of a base class. A *using* declaration may not be used for a member of a class from outside that class, its derived classes, and their member functions. A *using* directive may not appear in a class definition and may not be used for a class.

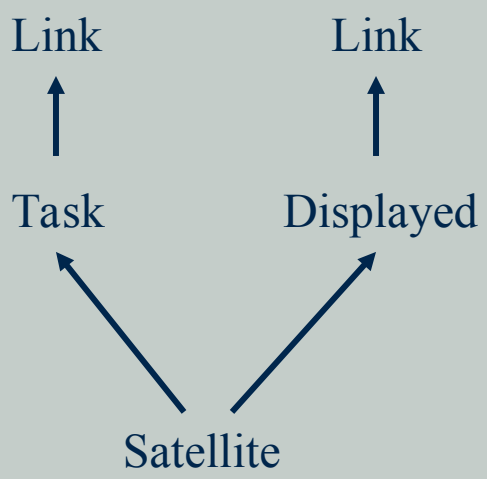
Replicated Base Classes

- With the ability of specifying more than one base class comes the possibility of having a class as a base twice. For example, had *Task* and *Displayed* each been derived from a *Link* class, a *Satellite* would have two *Link*s:

```
struct Link {
 Link* next;
};
class Task : public Link {
 // the Link is used to maintain a list of all Tasks
 // (the scheduler list)
 // ...
};
class Displayed : public Link {
 // the Link is used to maintain a list
 // of all Displayed objects (the display list)
 // ...
};
```

Replicated Base Classes

- This causes no problems. Two separate *Link* objects are used to represent the links, and the two lists do not interfere with each other.
- One cannot refer to members of the *Link* class without risking an ambiguity. A *Satellite* object could be drawn like this:



Replicated Base Classes

- Examples of where the common base class shouldn't be represented by two separate objects can be handled using a virtual base class.
- Usually, a base class that is replicated the way *Link* is here is an implementation detail that shouldn't be used from outside its immediate derived class. If such a base must be referred to from a point where more than one copy of the base is visible, the reference must be explicitly qualified to resolve the ambiguity.

```
void mess_with_links(Satellite* p)
{
 p->next = 0; // error: ambiguous (which Link?)
 p->Link::next = 0; // error: ambiguous (which Link?)
 p->Task::next = 0; // ok
 p->Displayed::next =0; // ok
 // ...
}
```

Overriding of Virtual Functions

- A virtual function of a replicated base class can be overridden by a (single) function in a derived class. For example, one might represent the ability of an object to read itself from a file and write itself back to a file.

```
class Storable {
 public:
   virtual const char* get_file() = 0;
   virtual void read() = 0;
   virtual void write() = 0;
   virtual ~Storable() {write() ; } // to be called
                                    // from overriding destructors
};
```

- It can be used to to develop classes that can be used independently or in combination to build more elaborate classes.

Overriding of Virtual Functions

- One way of stopping and restarting a simulation is to store components of a simulation and then restore them later. That idea might be implemented like this:

```
class Transmitter : public Storable {
 public:
   void write() ;
   // ...
};
class Receiver : public Storable {
 public:
   void write() ;
   // ...
};
class Radio : public Transmitter, public Receiver {
 public:
   const char* get_file() ;
   void read() ;
   void write() ;
   // ...
};
```

- Typically, an overriding function calls its base class versions and then does the work specific to the derived class:

```
void Radio::write()
{
 Transmitter::write() ;
 Receiver::write() ;
 // write radio-specific information
}
```

Virtual Base Classes

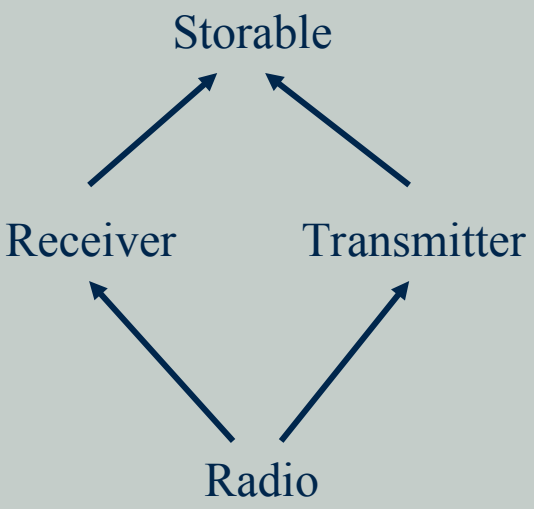
- The *Radio* example works because class *Storable* can be safely, conveniently, and efficiently replicated. Often, that is not the case for the kind of class that makes a good building block for other classes. For example, we might define *Storable* to hold the name of the file to be used for storing the object:

```
class Storable {
 public:
   Storable(const char* s) ;
   virtual void read() = 0;
   virtual void write() = 0;
   virtual ~Storable() ;
 private:
   const char* store;
   Storable(const Storable&) ;
   Storable& operator=(const Storable&) ;
};
```

Virtual Base Classes

- Given this apparently minor change to *Storable*, we must must change the design of *Radio*. All parts of an object must share a single copy of *Storable*; otherwise, it becomes unnecessarily hard to avoid storing multiple copies of the object. One mechanism for specifying such sharing is a virtual base class. Every virtual base of a derived class is represented by the same (shared) object.

```
class Transmitter : public virtual Storable {
 public:
   void write() ;
   // ...
};
class Receiver : public virtual Storable {
 public:
   void write() ;
   // ...
};
class Radio : public Transmitter, public Receiver
{
 public:
   void write() ;
   // ...
};
```



Programming Virtual Bases

- When defining the functions for a class with a virtual base, the programmer in general cannot know whether the base will be shared with other derived classes. This can be a problem when implementing a service that requires a base class function to be acalled exactly once.
- The language ensures that a constructor of a virtual base is called exactly once. The constructor of a virtual base is invoked (implicitly or explicitly) from the constructor for the complete object (the constructor for the most derived class).

```
class A { // no constructor
 // ...
};
class B {
 public:
   B(); // default constructor
   // ...
};
class C {
 public:
   C(int); // no default constructor
};
class D : virtual public A, virtual public B, virtual public C
{
 public:
  D() { /* ... */ }; // error: no default constructor for C
  D(int i) :C(i) { /* ... */ }; // ok
  // ...
};
class E: public D
{
 public:
  E() { /* ... */ }; // error: no default constructor for C
  E(int i) :C(i) { /* ... */ }; // ok
  // ...
};
```

- The constructor for a virtual base is called before the constructors for its derived classes.

Programming Virtual Bases

- Where needed, the programmer can simulate this scheme by calling a virtual base class function only from the most derived class. For example, assume we have a basic *Window* class that knows how to draw its contents:

```
class Window {
 // basic stuff
 virtual void draw() ;
};
```

- In addition, we have various ways of decorating a window and adding facilities:

```
class Window_with_border : public virtual Window {
 // border stuff
 void own_draw() ; // display the border
 void draw() ;
};
class Window_with_menu : public virtual Window {
 // menu stuff
 void own_draw() ; // display the menu
 void draw() ;
};
```

- The *own_draw()* functions need not be virtual because they are meant to be called from within a virtual *draw()* function that "knows" the type of the object for which it was called.

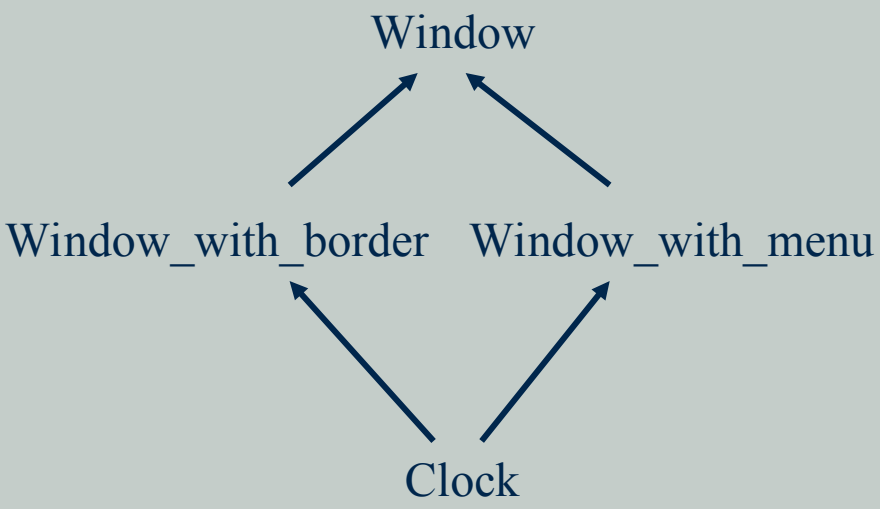
Programming Virtual Bases

- From this, we can compose a plausible *Clock* class:

```
class Clock : public Window_with_border, public Window_with_menu {
 // clock stuff
 void own_draw(); // display the clock face and hands
 void draw();
};
```

- The *draw()* functions can now be written using the *own_draw()* functions so that a caller of any *draw()* gets *Window::draw()* invoked exactly once. This is done independently of the kind of *Window* for which *draw()* is invoked:

```
void Window_with_border::draw()
{
 Window::draw() ;
 own_draw(); // display the border
}
void Window_with_menu::draw()
{
 Window::draw() ;
 own_draw(); // display the menu
}
void Clock::draw()
{
 Window::draw() ;
 Window_with_border::own_draw() ;
 Window_with_menu::own_draw() ;
 own_draw() ; // display the clock
              // face and hands
}
```

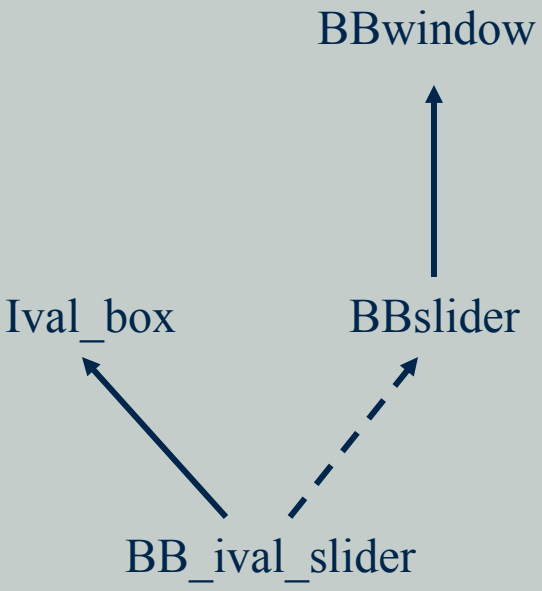


Using Multiple Inheritance

- The simplest and most obvious use of multiple inheritance is to "glue" two otherwise unrelated classes together as part of the implementation of a third class. The *Satellite* class built out of the *Task* and *Displayed* classes is an example of this.
- This use of multiple inheritance is crude, effective, and important, but not very interesting. Basically, it saves the programmer from writing a lot of forwarding functions. This technique does not affect the overall design of a program significantly and can occasionally clash with the wish to keep implementation details hidden.
- Using multiple inheritance to provide implementations for abstract classes is more fundamental in that it affects the way a program is designed.

Using Multiple Inheritance

```
class Ival_box {
 public:
   virtual int get_value() = 0;
   virtual void set_value(int i) = 0;
   virtual void reset_value(int i) = 0;
   virtual void prompt() = 0;
   virtual bool was_changed() const = 0;
   virtual ~Ival_box() { }
};
class BB_ival_slider
 : public Ival_box // interface
 ,protected BBslider // implementation
{
 // implementation of functions required
 // by 'Ival_slider' and 'BBslider'
 // using the facilities provided by 'BBslider'
};
```



- In this example, the two base classes play logically distinct roles. One base is a public abstract class providing the interface and the other is a protected concrete class providing implementation "details".
- The use of multiple inheritance is close to essential here because the derived class needs to override virtual functions from both the interface and the implementation.

Overriding Virtual Base Functions

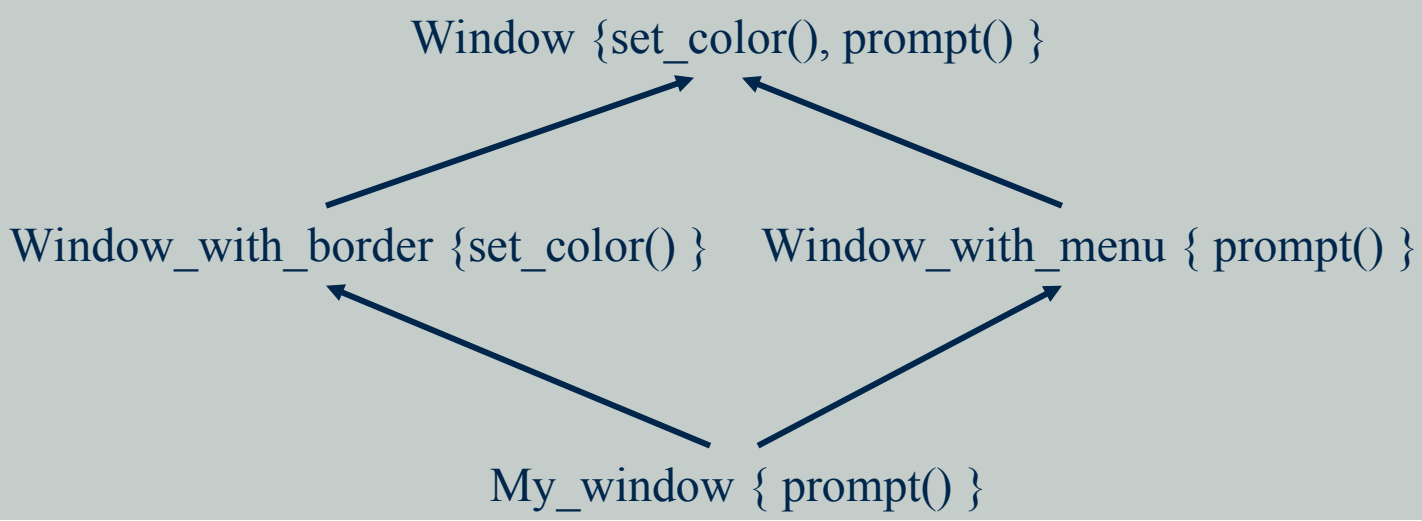
- A derived class can override a virtual function of its direct or indirect virtual base class.
- In particular, two different classes might override different virtual functions from the virtual base.
- In that way, several derived classes can contribute implementations to the interface presented by a virtual base class.

```
class Window {
 // ...
 virtual set_color(Color) = 0; // set background color
 virtual void prompt() = 0;
};
class Window_with_border : public virtual Window {
 // ...
 set_color(Color) ; // control background color
};
class Window_with_menu : public virtual Window {
 // ...
 void prompt() ; // control user interactions
};
class My_window : public Window_with_menu, public Window_with_border {
 // ...
};
```

Overriding Virtual Base Functions

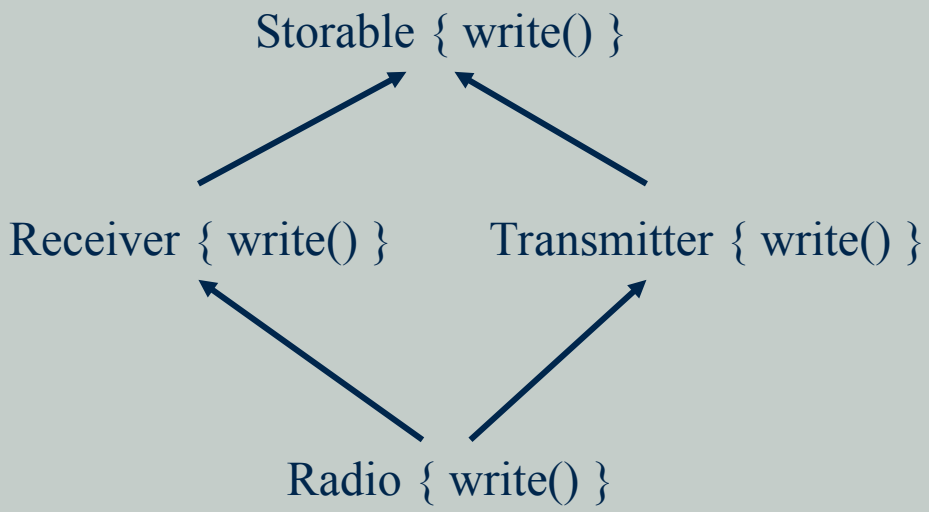
- If different derived classes override the same function, it is allowed if and only if some overriding class is derived from every other class that overrides the function. That is, one function must override all others.
- *My_window* could override *prompt()* to improve on what *Window_with_menu* provides:

```
class My_window : public Window_with_menu, public Window_with_border {
 // ...
 void prompt() ; // don't leave user interactions to base
};
```



Overriding Virtual Base Functions

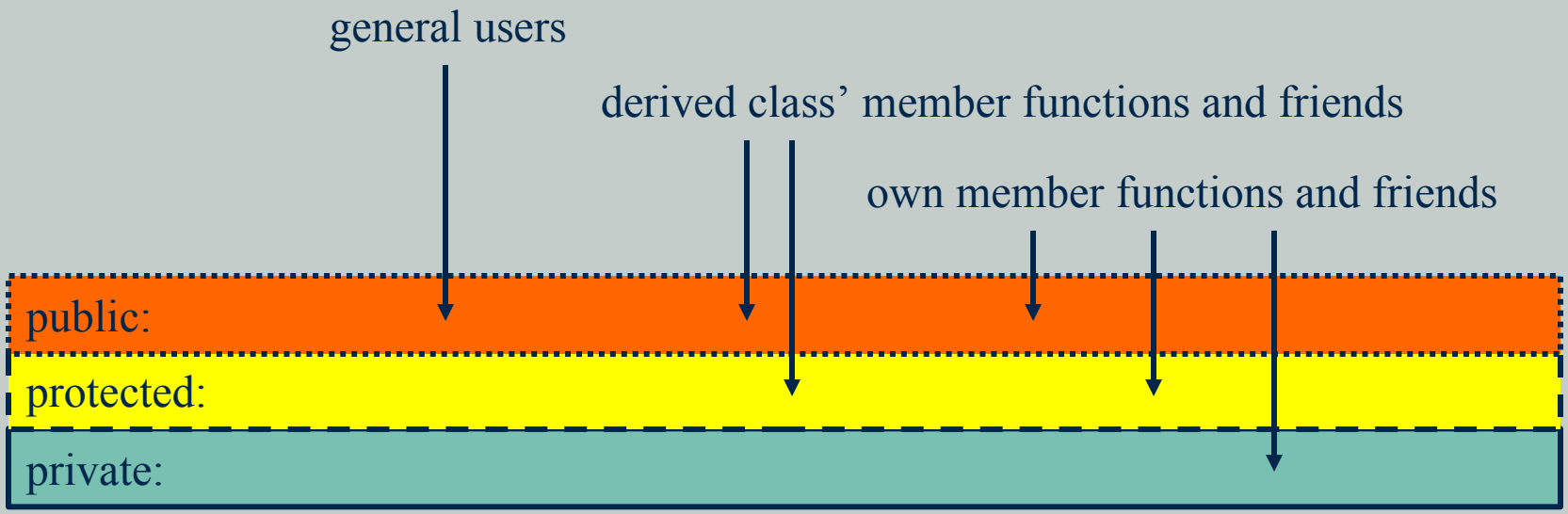
- If two classes override a base class function, but neither overrides the other, the class hierarchy is an error.
- No virtual function table can be constructed because a call to that function on the complete object would have been ambiguous.
- Had *Radio* not declared *write()*, the declarations of *write()* in *Receiver* and *Transmitter* would have caused an error when defining *Radio*.
- Such a conflict is resolved by adding an overriding function to the most derived class.



- A class that provides some – but not all – of the implementation for a virtual base class is often called a "mixin".

Access Control

- A member of a class can be private, protected, or public:
  - If it is private, its name can be used only by member functions and friends of the class in which it is declared.
  - If it is protected, its name can be used only by member functions and friends of the class in which it is declared and by member functions and friends of classes derived from this class.
  - If it is public, its name can be used by any function.



Protected Members

- As an example of how to use protected members, consider the *Window* example
- The *own_draw()* functions were (deliberately) incomplete in the service they provided.
- They were designed as building blocks for use by derived classes (only) and are not safe or convenient for general use.
- The *draw()* operations, on the other hand, were designed for general use.
- It can be stated explicitly:

```
class Window_with_border {
 public:
   virtual void draw() ;
   // ...
 protected:
   void own_draw() ;
   // other toolbuilding stuff
 private:
   // representation, etc.
};
```

Protected Members

- A derived class can access a base class' protected members only for objects of its own type:

```
class Buffer {
 protected:
   char a[128] ;
   // ...
};
class Linked_buffer : public Buffer{ /* ... */ };
class Cyclic_buffer : public Buffer {
// ...
 void f(Linked_buffer* p) {
   a[0] = 0; // ok: access to cyclic_buffer's own protected member
   p->a[0] = 0; // error: access to protected member of different type
 }
};
```

- This prevents subtle errors that would otherwise occur when one derived class corrupts data belonging to other derived classes.

Use of Protected Members

- The simple private/public model of data hiding serves the notion of concrete types well.
- However, when derived classes are used, there are two kinds of users of a class: derived classes and "the general public".
- The members and friends that implement the operations on the class operate on the class objects on behalf of these users.
- The private/public model allows the programmer to distinguish clearly between the implementers and the general public, but it does not provide a way of catering specifically to derived classes.
- Declaring data members protected is usually a design error.
  - Placing significant amounts of data in a common class for all derived classes to use leaves that data open to corruption.
  - Protected data, like public data, cannot easily be restructured because there is no good way of finding every use.

Access to Base Classes

- Like a member, a base class can be declared private, protected, or public.

```
class X : public B{ /* ... */ };
class Y : protected B{ /* ... */ };
class Z : private B{ /* ... */ };
```

- Public derivation makes the derived class a subtype of its base; this is the most common form of derivation.
- Protected and private derivation are used to represent implementation details.
  - Protected bases are useful in class hierarchies in which further derivation is the norm.
  - Private bases are most useful when defining a class by restricting the interface to a base so that stronger guarantees can be provided.

Access to Base Classes

- The access specifier for a base class can be left out. In that case, the base defaults to a private base for a class and a public base for a struct.

```
class XX :B{ /* ... */ }; // B is a private base
struct YY :B{ /* ... */ }; // B is a public base
```

- The access specifier for a base class controls the access to members of the base class and the conversion of pointers and references from the derived class type to the base class type. Consider a class *D* derived from a base class *B*:
  - If *B* is a private base, its public and protected members can be used only by member functions and friends of *D*. Only friends and members of *D* can convert a *D** to a *B**.
  - If *B* is a protected base, its public and protected members can be used only by member functions and friends of *D* and by member functions and friends of classes derived from *D*. Only friends and members of *D* and friends and members of classes derived from *D* can convert a *D** to a *B**.
  - If *B* is a public base, its public members can be used by any function. In addition, its protected members can be used by members and friends of *D* and members and friends of classes derived from *D*. Any function can convert a *D** to a *B**.

Multiple Inheritance and Access Control

- If a name or a base class can be reached through multiple paths in a multiple inheritance lattice, it is accessible if it is accessible through any path.

```
struct B {
 int m;
 static int sm;
 // ...
};
class D1 : public virtual B{ /* ... */ } ;
class D2 : public virtual B{ /* ... */ } ;
class DD : public D1, private D2{ /* ... */ };
DD* pd = new DD;
B* pb = pd;  // ok: accessible through D1
int i1 = pd->m; // ok: accessible through D1
```

- If a single entity is reachable through several paths, we can still refer to it without ambiguity.

```
class X1 : public B{ /* ... */ } ;
class X2 : public B{ /* ... */ } ;
class XX : public X1, public X2{ /* ... */ };
XX* pxx = new XX;
int i1 = pxx->m; // error, ambiguous: XX::X1::B::m or XX::X2::B::m
int i2 = pxx->sm; // ok: there is only one B::sm in an XX
```

Using Declarations and Access Control

- A *using* declaration cannot be used to gain access to additional information. It is simply a mechanism for making accessible information more convenient to use. On the other hand, once access is available, it can be granted to other users.

```
class B {
 private:
   int a;
 protected:
   int b;
 public:
   int c;
};
class D : public B {
 public:
   using B::a; // error: B::a is private
   using B::b; // make B::b publically available through D
};
```

- When a *using* declaration is combined with private or protected derivation, it can be used to specify interfaces to some, but not all, of the facilities usually offered by a class.

```
class BB : private B{ // give access to B::b and B::c, but not B::a
 using B::b;
 using B::c;
};
```