15

Class Hierarchies

Abstraction is selective ignorance.
– Andrew Koenig

Multiple inheritance — ambiguity resolution — inheritance and *using-declaration*s — replicated base classes — virtual base classes — uses of multiple inheritance — access control — protected — access to base classes — run-time type information — ***dynamic_cast*** — static and dynamic casts — casting from virtual bases — ***typeid*** — extended type information — uses and misuses of run-time type information — pointers to members — free store — virtual constructors — advice — exercises.

15.1 Introduction and Overview [hier.intro]

This chapter discusses how derived classes and virtual functions interact with other language facilities such as access control, name lookup, free store management, constructors, pointers, and type conversions. It has five main parts:

§15.2 Multiple Inheritance
§15.3 Access Control
§15.4 Run-time Type Identification
§15.5 Pointers to Members
§15.6 Free Store Use

In general, a class is constructed from a lattice of base classes. Because most such lattices historically have been trees, a *class lattice* is often called a *class hierarchy*. We try to design classes so that users need not be unduly concerned about the way a class is composed out of other classes. In particular, the virtual call mechanism ensures that when we call a function `f()` on an object, the same function is called whichever class in the hierarchy provided the declaration of `f()` used for the call. This chapter focuses on ways to compose class lattices and to control access to parts of classes and on facilities for navigating class lattices at compile time and run time.

15.2 Multiple Inheritance [hier.mi]

As shown in §2.5.4 and §12.3, a class can have more than one direct base class, that is, more than one class specified after the : in the class declaration. 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*.

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

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

Similarly, a ***Satellite*** can be passed to functions that expect a ***Task*** or a ***Displayed***. For example:

```
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
}
```

The implementation of this clearly involves some (simple) compiler technique to ensure that functions expecting a ***Task*** see a different part of a ***Satellite*** than do functions expecting a ***Displayed***. Virtual functions work as usual. For example:

```
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.

Note that with single inheritance (only), the programmer's choices for implementing the classes ***Displayed***, ***Task***, and ***Satellite*** would be limited. A ***Satellite*** could be a ***Task*** or a ***Displayed***, but not both (unless ***Task*** was derived from ***Displayed*** or vice versa). Either alternative involves a loss of flexibility.

Why would anyone want a class ***Satellite***? Contrary to some people's conjectures, the ***Satellite*** example is real. There really was – and maybe there still is – a program constructed along the lines used to describe multiple inheritance here. It was used to study the design of communication systems involving satellites, ground stations, etc. Given such a simulation, we can answer questions about traffic flow, determine proper responses to a ground station that is being blocked by a rainstorm, consider tradeoffs between satellite connections and Earth-bound connections, etc. Such simulations do involve a variety of display and debugging operations. Also, we do need to store the state of objects such as ***Satellite***s and their subcomponents for analysis, debugging, and error recovery.

15.2.1 Ambiguity Resolution [hier.ambig]

Two base classes may have member functions with the same name. For example:

```
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
}
```

However, 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.

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

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

In other words, 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.

15.2.2 Inheritance and Using-Declarations [hier.using]

Overload resolution is not applied across different class scopes (§7.4). 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. For example:

```
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
}
```

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* (§8.2.2) can bring the functions into a common scope. For example:

```
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)
}
```

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 (§15.2.3.1).

A *using-declaration* (§8.2.2) 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* (§8.2.3) may not appear in a class definition and may not be used for a class.

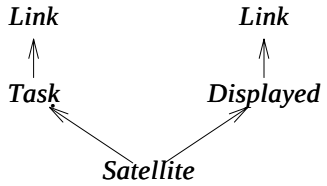
A *using-declaration* cannot be used to gain access to additional information. It is simply a mechanism for making accessible information more convenient to use (§15.3.2.2).

15.2.3 Replicated Base Classes [hier.replicated]

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)
    // ...
};
```

This causes no problems. Two separate ***Link*** objects are used to represent the links, and the two lists do not interfere with each other. Naturally, one cannot refer to members of the ***Link*** class without risking an ambiguity (§15.2.3.1). A ***Satellite*** object could be drawn like this:



Examples of where the common base class shouldn't be represented by two separate objects can be handled using a virtual base class (§15.2.4).

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. For example:

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

```
        p->Task::Link::next = 0;        // ok
        p->Displayed::Link::next = 0;   // ok
        // ...
}
```

This is exactly the mechanism used to resolve ambiguous references to members (§15.2.1).

15.2.3.1 Overriding [hier.override]

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 like this:

```
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 (see §15.2.4.1)
};
```

Naturally, several programmers might rely on this to develop classes that can be used independently or in combination to build more elaborate classes. For example, 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
}
```

Casting from a replicated base class to a derived class is discussed in §15.4.2. For a technique for overriding each of the ***write()*** functions with separate functions from derived classes, see §25.6.

15.2.4 Virtual Base Classes [hier.vbase]

The ***Radio*** example in the previous subsection 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:

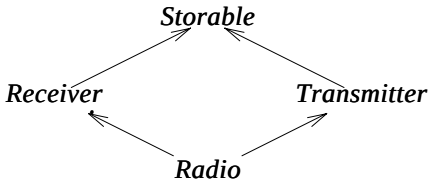
```
class Storable {
public:
    Storable(const char* s);
    virtual void read() = 0;
    virtual void write() = 0;
    virtual ~Storable();
private:
    const char* store;

    Storable(const Storable&);
    Storable& operator=(const Storable&);
};
```

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. For example:

```
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();
    // ...
};
```

Or graphically:



Compare this diagram with the drawing of the ***Satellite*** object in §15.2.3 to see the difference between ordinary inheritance and virtual inheritance. In an inheritance graph, every base class of a given name that is specified to be virtual will be represented by a single object of that class. On the other hand, each base class not specified ***virtual*** will have its own sub-object representing it.

15.2.4.1 Programming Virtual Bases [hier.vbase.prog]

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 called exactly once. For example, 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). For example:

```
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
{
    D() { /* ... */ }                 // error: no default constructor for C
    D(int i) : C(i) { /* ... */ };    // ok
    // ...
};
```

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

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:

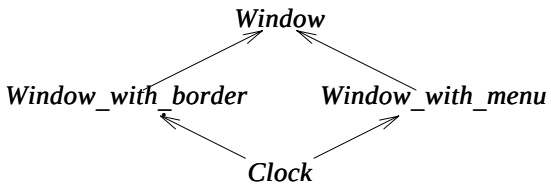
```
class Window_with_border : public virtual Window {
    // border stuff
    void own_draw();    // display the border
    void draw();
};

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.

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();
};
```

Or graphically:



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
}
```

Casting from a ***virtual*** base class to a derived class is discussed in §15.4.2.

15.2.5 Using Multiple Inheritance [hier.using.mi]

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 in §15.2 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. However, a technique doesn't have to be clever to be useful.

Using multiple inheritance to provide implementations for abstract classes is more fundamental in that it affects the way a program is designed. Class ***BB_ival_slider*** (§12.3) is an example:

```
class BB_ival_slider
    : public Ival_slider   // 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.'' These roles are reflected in both the style of the classes and in the access control provided. 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.

Multiple inheritance allows sibling classes to share information without introducing a dependence on a unique common base class in a program. This is the case in which the so-called

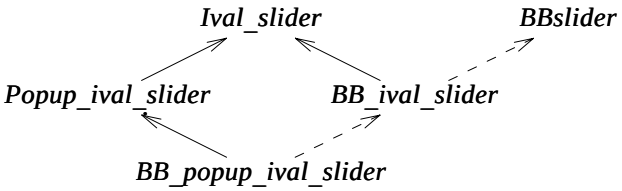
diamond-shaped inheritance occurs (for example, the ***Radio*** (§15.2.4) and ***Clock*** (§15.2.4.1)). A virtual base class, as opposed to an ordinary base class, is needed if the base class cannot be replicated.

I find that a diamond-shaped inheritance lattice is most manageable if either the virtual base class or the classes directly derived from it are abstract classes. For example, consider again the ***Ival_box*** classes from §12.4. In the end, I made all the ***Ival_box*** classes abstract to reflect their role as pure interfaces. Doing that allowed me to place all implementation details in specific implementation classes. Also, all sharing of implementation details was done in the classical hierarchy of the windows system used for the implementation.

It would make sense for the class implementing a ***Popup_ival_slider*** to share most of the implementation of the class implementing a plain ***Ival_slider***. After all, these implementation classes would share everything except the handling of prompts. However, it would then seem natural to avoid replication of ***Ival_slider*** objects within the resulting slider implementation objects. Therefore, we could make ***Ival_slider*** a virtual base:

```
class BB_ival_slider : public virtual Ival_slider, protected BBslider { /* ... */ };
class Popup_ival_slider : public virtual Ival_slider { /* ... */ };
class BB_popup_ival_slider
      : public virtual Popup_ival_slider, protected BB_ival_slider { /* ... */ };
```

or graphically:

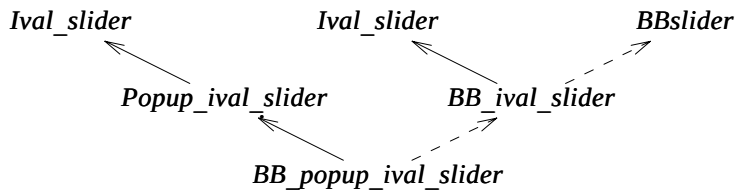


It is easy to imagine further interfaces derived from ***Popup_ival_slider*** and further implementation classes derived from such classes and ***BB_popup_slider***.

If we take this idea to its logical conclusion, all of the derivations from the abstract classes that constitute our application's interfaces would become virtual. This does indeed seem to be the most logical, general, and flexible approach. The reason I didn't do that was partly historical and partly because the most obvious and common techniques for implementing virtual bases impose time and space overhead that make their extensive use within a class unattractive. Should this overhead become an issue for an otherwise attractive design, note that an object representing an ***Ival_slider*** usually holds only a virtual table pointer. As noted in §15.2.4, such an abstract class holding no variable data can be replicated without ill effects. Thus, we can eliminate the virtual base in favor of ordinary ones:

```
class BB_ival_slider : public Ival_slider, protected BBslider { /* ... */ };
class Popup_ival_slider : public Ival_slider { /* ... */ };
class BB_popup_ival_slider
      : public Popup_ival_slider, protected BB_ival_slider { /* ... */ };
```

or graphically:



This is most likely a viable optimization to the admittedly cleaner alternative presented previously.

15.2.5.1 Overriding Virtual Base Functions [hier.dominance]

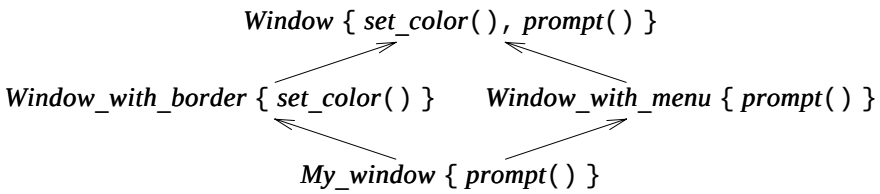
A derived class can override a virtual function of its direct or indirect virtual base class. In particular, two different classes might override different virtual functions from the virtual base. In that way, several derived classes can contribute implementations to the interface presented by a virtual base class. For example, the ***Window*** class might have functions ***set_color()*** and ***prompt()***. In that case, ***Window_with_border*** might override ***set_color()*** as part of controlling the color scheme and ***Window_with_menu*** might override ***prompt()*** as part of its control of user interactions:

```
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 {
    // ...
};
```

What if different derived classes override the same function? This 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. For example, ***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
};
```

or graphically:



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. For example, had ***Radio*** in §15.2.4 not declared ***write*( )**, the declarations of ***write*( )** in ***Receiver*** and ***Transmitter*** would have caused an error when defining ***Radio***. As with ***Radio***, such a conflict is resolved by adding an overriding function to the most derived class.

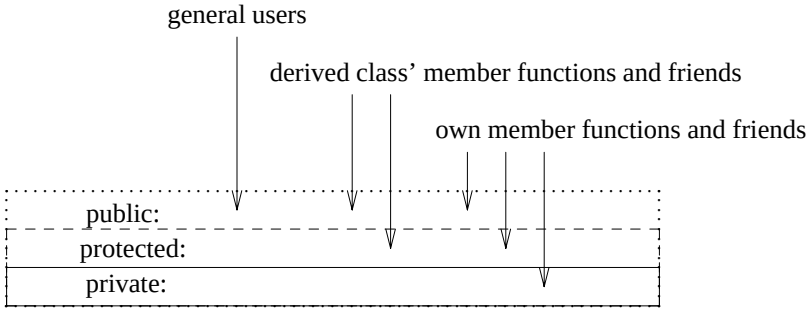
A class that provides some – but not all – of the implementation for a virtual base class is often called a ‘‘mixin.’’

15.3 Access Control [hier.access]

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 (see §11.5).
- If it is ***public***, its name can be used by any function.

This reflects the view that there are three kinds of functions accessing a class: functions implementing the class (its friends and members), functions implementing a derived class (the derived class' friends and members), and other functions. This can be presented graphically:



The access control is applied uniformly to names. What a name refers to does not affect the control of its use. This means that we can have private member functions, types, constants, etc., as well as private data members. For example, an efficient non-intrusive (§16.2.1) list class often requires data structures to keep track of elements. Such information is best kept private:

```
template<class T> class List {
private:
        struct Link { T val; Link* next; };
        struct Chunk {
                enum { chunk_size = 15 };
                Link v[chunk_size];
                Chunk* next;
        };
        class Underflow { };

        Chunk* allocated;
        Link* free;
        Link* get_free();
        Link* head;
public:
        void insert(T);
        T get();
        // ...
};

template<class T> void List<T>::insert(T val)
{
        Link* lnk = get_free();
        lnk->val = val;
        lnk->next = head;
        head = lnk;
}

template<class T> List<T>::Link* List<T>::get_free()
{
        if (free == 0) {
                // allocate a new chunk and place its Links on the free list
        }
        Link* p = free;
        free = free->next;
        return p;
}

template<class T> T List<T>::get()
{
        if (head == 0) throw Underflow();

        Link* p= head;
        head = p->next;
        p->next = free;
        free = p;
        return p->val;
}
```

The ***List<T>*** scope is entered by saying ***List<T>::*** in a member function definition. Because the return type of ***get_free()*** is mentioned before the name ***List<T>::get_free()*** is mentioned, the full name ***List<T>::Link*** must be used instead of the abbreviation ***Link<T>***.

Nonmember functions (except friends) do not have such access:

```
void would_be_meddler(List<T>* p)
{
    List<T>::Link* q = 0;                    // error: List<T>::Link is private
    // ...
    q = p->free;                             // error: List<T>::free is private
    // ...
    if (List<T>::Chunk::chunk_size > 31) {   // error: List<T>::Chunk::chunk_size is private
        // ...
    }
}
```

In a ***class***, a member is by default private; in a ***struct***, a member is by default public (§10.2.8).

15.3.1 Protected Members [hier.protected]

As an example of how to use ***protected*** members, consider the ***Window*** example from §15.2.4.1. 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. This distinction can be expressed by separating the interface of the ***Window*** classes in two, the ***protected*** interface and the ***public*** interface:

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

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.

15.3.1.1 Use of Protected Members [hier.protected.use]

The simple private/public model of data hiding serves the notion of concrete types (§10.3) 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.

Members declared ***protected*** are far more open to abuse than members declared ***private***. In particular, 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. Worse, protected data, like public data, cannot easily be restructured because there is no good way of finding every use. Thus, protected data becomes a software maintenance problem.

Fortunately, you don't have to use protected data; ***private*** is the default in classes and is usually the better choice. In my experience, there have always been alternatives to placing significant amounts of information in a common base class for derived classes to use directly.

Note that none of these objections are significant for protected member *functions*; ***protected*** is a fine way of specifying operations for use in derived classes. The ***Ival_slider*** in §12.4.2 is an example of this. Had the implementation class been ***private*** in this example, further derivation would have been infeasible.

Technical examples illustrating access to members can be found in §C.11.1.

15.3.2 Access to Base Classes [hier.base.access]

Like a member, a base class can be declared ***private***, ***protected***, or ***public***. For example:

```
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; the ***Ival_slider*** from §12.4.2 is a good example of that. Private bases are most useful when defining a class by restricting the interface to a base so that stronger guarantees can be provided. For example, ***Vec*** adds range checking to its private base ***vector*** (§3.7.1) and the ***list*** of pointers template adds type checking to its ***list<void*>*** base (§13.5).

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***. For example:

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

For readability, it is best always to use an explicit access specifier.

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****.

This basically restates the rules for member access (§15.3). We choose access for bases in the same way as for members. For example, I chose to make ***BBwindow*** a ***protected*** base of ***Ival_slider*** (§12.4.2) because ***BBwindow*** was part of the implementation of ***Ival_slider*** rather than part of its interface. However, I couldn't completely hide ***BBwindow*** by making it a private base because I wanted to be able to derive further classes from ***Ival_slider***, and those derived classes would need access to the implementation.

Technical examples illustrating access to bases can be found in §C.11.2.

15.3.2.1 Multiple Inheritance and Access Control [hier.mi.access]

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. For example:

```
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. For example:

```
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
```

15.3.2.2 Using-Declarations and Access Control [hier.access.using]

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. For example:

```
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. For example:

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

See also §15.2.2.

15.4 Run-Time Type Information [hier.rtti]

A plausible use of the ***Ival_box***es defined in §12.4 would be to hand them to a system that controlled a screen and have that system hand objects back to the application program whenever some activity had occurred. This is how many user-interfaces work. However, a user-interface system will not know about our ***Ival_box***es. The system's interfaces will be specified in terms of the system's own classes and objects rather than our application's classes. This is necessary and proper. However, it does have the unpleasant effect that we lose information about the type of objects passed to the system and later returned to us.

Recovering the ''lost'' type of an object requires us to somehow ask the object to reveal its type. Any operation on an object requires us to have a pointer or reference of a suitable type for the object. Consequently, the most obvious and useful operation for inspecting the type of an object at run time is a type conversion operation that returns a valid pointer if the object is of the expected type and a null pointer if it isn't. The ***dynamic_cast*** operator does exactly that. For example, assume that ''the system'' invokes ***my_event_handler***() with a pointer to a ***BBwindow***, where an activity has occurred. I then might invoke my application code using ***Ival_box***'s ***do_something***():

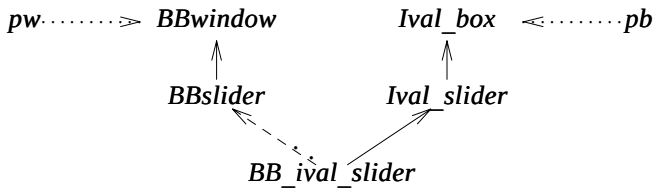
```
void my_event_handler(BBwindow* pw)
{
    if (Ival_box* pb = dynamic_cast<Ival_box*>(pw))    // does pw point to an Ival_box?
        pb->do_something();
    else {
        // Oops! unexpected event
    }
}
```

One way of explaining what is going on is that ***dynamic_cast*** translates from the implementation-oriented language of the user-interface system to the language of the application. It is important to note what is *not* mentioned in this example: the actual type of the object. The object will be a particular kind of ***Ival_box***, say an ***Ival_slider***, implemented by a particular kind of ***BBwindow***, say a ***BBslider***. It is neither necessary nor desirable to make the actual type of the object explicit in this interaction between ''the system'' and the application. An interface exists to represent the essentials of an interaction. In particular, a well-designed interface hides inessential details.

Graphically, the action of

```
pb = dynamic_cast<Ival_box*>(pw)
```

can be represented like this:



The arrows from ***pw*** and ***pb*** represent the pointers into the object passed, whereas the rest of the arrows represent the inheritance relationships between the different parts of the object passed.

The use of type information at run time is conventionally referred to as ''run-time type information'' and often abbreviated to RTTI.

Casting from a base class to a derived class is often called a *downcast* because of the convention of drawing inheritance trees growing from the root down. Similarly, a cast from a derived class to a base is called an *upcast*. A cast that goes from a base to a sibling class, like the cast from ***BBwindow*** to ***Ival_box***, is called a *crosscast*.

15.4.1 Dynamic_cast [hier.dynamic.cast]

The ***dynamic_cast*** operator takes two operands, a type bracketed by < and >, and a pointer or reference bracketed by (and).

Consider first the pointer case:

```
dynamic_cast<T*>(p)
```

If ***p*** is of type ***T**** or an accessible base class of ***T***, the result is exactly as if we had simply assigned ***p*** to a ***T****. For example:

```
class BB_ival_slider : public Ival_slider, protected BBslider {
    // ...
};

void f(BB_ival_slider* p)
{
    Ival_slider* pi1 = p;        // ok
    Ival_slider* pi2 = dynamic_cast<Ival_slider*>(p);        // ok

    BBslider* pbb1 = p;        // error: BBslider is a protected base
    BBslider* pbb2 = dynamic_cast<BBslider*>(p);        // ok: pbb2 becomes 0
}
```

That is the uninteresting case. However, it is reassuring to know that ***dynamic_cast*** doesn't allow accidental violation of the protection of private and protected base classes.

The purpose of ***dynamic_cast*** is to deal with the case in which the correctness of the conversion cannot be determined by the compiler. In that case,

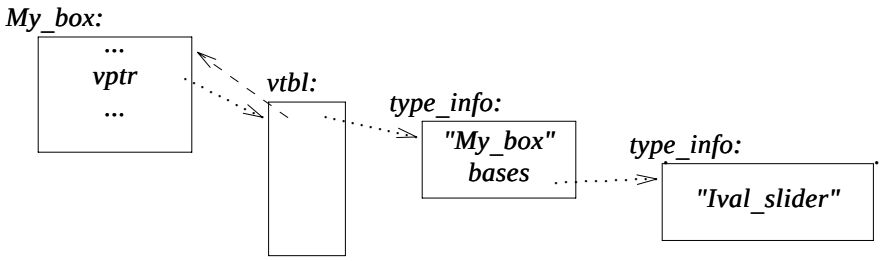
```
dynamic_cast<T*>(p)
```

looks at the object pointed to by ***p*** (if any). If that object is of class ***T*** or has a unique base class of type ***T***, then ***dynamic_cast*** returns a pointer of type ***T**** to that object; otherwise, ***0*** is returned. If the value of ***p*** is ***0***, ***dynamic_cast<T*>(p)*** returns ***0***. Note the requirement that the conversion must be to a uniquely identified object. It is possible to construct examples where the conversion fails and ***0*** is returned because the object pointed to by ***p*** has more than one sub-object representing bases of type ***T*** (see §15.4.2).

A ***dynamic_cast*** requires a pointer or a reference to a polymorphic type to do a downcast or a crosscast. For example:

```
class My_slider: public Ival_slider { // polymorphic base (Ival_slider has virtual functions)
    // ...
};

class My_date : public Date {  // base not polymorphic (Date has no virtual functions)
    // ...
};

void g(Ival_box* pb, Date* pd)
{
    My_slider* pd1 = dynamic_cast<My_slider*>(pb);    // ok
    My_date* pd2 = dynamic_cast<My_date*>(pd);        // error: Date not polymorphic
}
```

Requiring the pointer's type to be polymorphic simplifies the implementation of ***dynamic_cast*** because it makes it easy to find a place to hold the necessary information about the object's type. A typical implementation will attach a ''type information object'' to an object by placing a pointer to the type information in the object's virtual function table (§2.5.5). For example:



The dashed arrow represents an offset that allows the start of the complete object to be found given only a pointer to a polymorphic sub-object. It is clear that ***dynamic_cast*** can be efficiently implemented. All that is involved are a few comparisons of ***type_info*** objects representing base classes; no expensive lookups or string comparisons are needed.

Restricting ***dynamic_cast*** to polymorphic types also makes sense from a logical point of view. This is, if an object has no virtual functions, it cannot safely be manipulated without knowledge of its exact type. Consequently, care should be taken not to get such an object into a context in which its type isn't known. If its type *is* known, we don't need to use ***dynamic_cast***.

The target type of ***dynamic_cast*** need not be polymorphic. This allows us to wrap a concrete type in a polymorphic type, say for transmission through an object I/O system (see §25.4.1), and then ''unwrap'' the concrete type later. For example:

```
class Io_obj {                  // base class for object I/O system
    virtual Io_obj* clone() = 0;
};

class Io_date : public Date, public Io_obj { };

void f(Io_obj* pio)
{
    Date* pd = dynamic_cast<Date*>(pio);
    // ...
}
```

A ***dynamic_cast*** to ***void**** can be used to determine the address of the beginning of an object of polymorphic type. For example:

```
void g(Ival_box* pb, Date* pd)
{
    void* pd1 = dynamic_cast<void*>(pb);    // ok
    void* pd2 = dynamic_cast<void*>(pd);    // error: Date not polymorphic
}
```

This is only useful for interaction with very low-level functions.

15.4.1.1 Dynamic_cast of References [hier.re.cast]

To get polymorphic behavior, an object must be manipulated through a pointer or a reference. When a ***dynamic_cast*** is used for a pointer type, a ***0*** indicates failure. That is neither feasible nor desirable for references.

Given a pointer result, we must consider the possibility that the result is *0*; that is, that the pointer doesn't point to an object. Consequently, the result of a ***dynamic_cast*** of a pointer should always be explicitly tested. For a pointer ***p***, ***dynamic_cast<T*>(p)*** can be seen as the question, ''Is the object pointed to by ***p*** of type ***T***?''

On the other hand, we may legitimately assume that a reference refers to an object. Consequently, ***dynamic_cast<T&>(r)*** of a reference ***r*** is not a question but an assertion: ''The object referred to by ***r*** is of type ***T***.'' The result of a ***dynamic_cast*** for a reference is implicitly tested by the implementation of ***dynamic_cast*** itself. If the operand of a ***dynamic_cast*** to a reference isn't of the expected type, a ***bad_cast*** exception is thrown. For example:

```
void f(Ival_box* p, Ival_box& r)
{
    if (Ival_slider* is = dynamic_cast<Ival_slider*>(p)) {      // does p point to an Ival_slider?
        // use 'is'
    }
    else {
        // *p not a slider
    }

    Ival_slider& is = dynamic_cast<Ival_slider&>(r);        // r references an Ival_slider!
    // use 'is'
}
```

The difference in results of a failed dynamic pointer cast and a failed dynamic reference cast reflects a fundamental difference between references and pointers. If a user wants to protect against bad casts to references, a suitable handler must be provided. For example:

```
void g()
{
    try {
        f(new BB_ival_slider, *new BB_ival_slider);     // arguments passed as Ival_boxs
        f(new BBdial, *new BBdial);                     // arguments passed as Ival_boxs
    }
    catch (bad_cast) {    // §14.10
        // ...
    }
}
```

The first call to ***f()*** will return normally, while the second will cause a ***bad_cast*** exception that will be caught by ***g()***.

Explicit tests against ***0*** can be – and therefore occasionally will be – accidentally omitted. If that worries you, you can write a conversion function that throws an exception instead returning ***0*** (§15.8[1]) in case of failure.

15.4.2 Navigating Class Hierarchies [hier.navigate]

When only single inheritance is used, a class and its base classes constitute a tree rooted in a single base class. This is simple but often constraining. When multiple inheritance is used, there is no single root. This in itself doesn't complicate matters much. However, if a class appears more than

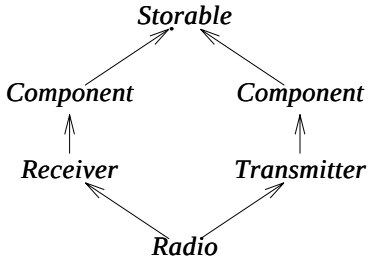
once in a hierarchy, we must be a bit careful when we refer to the object or objects that represent that class.

Naturally, we try to keep hierarchies as simple as our application allows (and no simpler). However, once a nontrivial hierarchy has been made we soon need to navigate it to find an appropriate class to use as an interface. This need occurs in two variants. That is, sometimes, we want to explicitly name an object of a base class or a member of a base class; §15.2.3 and §15.2.4.1 are examples of this. At other times, we want to get a pointer to the object representing a base or derived class of an object given a pointer to a complete object or some sub-object; §15.4 and §15.4.1 are examples of this.

Here, we consider how to navigate a class hierarchy using type conversions (casts) to gain a pointer of the desired type. To illustrate the mechanisms available and the rules that guide them, consider a lattice containing both a replicated base and a virtual base:

```
class Component : public virtual Storable { /* ... */ };
class Receiver : public Component { /* ... */ };
class Transmitter : public Component { /* ... */ };
class Radio : public Receiver, public Transmitter { /* ... */ };
```

Or graphically:



Here, a ***Radio*** object has two sub-objects of class ***Component***. Consequently, a ***dynamic_cast*** from ***Storable*** to ***Component*** within a ***Radio*** will be ambiguous and return a ***0***. There is simply no way of knowing which ***Component*** the programmer wanted:

```
void h1(Radio& r)
{
    Storable* ps = &r;
    // ...
    Component* pc = dynamic_cast<Component*>(ps); // pc = 0
}
```

This ambiguity is not in general detectable at compile time:

```
void h2(Storable* ps)    // ps might or might not point to a Component
{
    Component* pc = dynamic_cast<Component*>(ps);
    // ...
}
```

This kind of run-time ambiguity detection is needed only for virtual bases. For ordinary bases,

there is always a unique sub-object of a given cast (or none) when downcasting (that is, towards a derived class; §15.4). The equivalent ambiguity occurs when upcasting (that is, towards a base; §15.4) and such ambiguities are caught at compile time.

15.4.2.1 Static and Dynamic Casts [hier.static.cast]

A ***dynamic_cast*** can cast from a polymorphic virtual base class to a derived class or a sibling class (§15.4.1). A ***static_cast*** (§6.2.7) does not examine the object it casts from, so it cannot:

```
void g(Radio& r)
{
    Receiver* prec = &r;                        // Receiver is ordinary base of Radio
    Radio* pr = static_cast<Radio*>(prec);      // ok, unchecked
    pr = dynamic_cast<Radio*>(prec);            // ok, run-time checked

    Storable* ps = &r;                          // Storable is virtual base of Radio
    pr = static_cast<Radio*>(ps);               // error: cannot cast from virtual base
    pr = dynamic_cast<Radio*>(ps);              // ok, run-time checked
}
```

The ***dynamic_cast*** requires a polymorphic operand because there is no information stored in a non-polymorphic object that can be used to find the objects for which it represents a base. In particular, an object of a type with layout constraints determined by some other language – such as Fortran or C – may be used as a virtual base class. For objects of such types, only static type information will be available. However, the information needed to provide run-time type identification includes the information needed to implement the ***dynamic_cast***.

Why would anyone want to use a ***static_cast*** for class hierarchy navigation? There is a small run-time cost associated with the use of a ***dynamic_cast*** (§15.4.1). More significantly, there are millions of lines of code that were written before ***dynamic_cast*** became available. This code relies on alternative ways of making sure that a cast is valid, so the checking done by ***dynamic_cast*** is seen as redundant. However, such code is typically written using the C-style cast (§6.2.7); often obscure errors remain. Where possible, use the safer ***dynamic_cast***.

The compiler cannot assume anything about the memory pointed to by a ***void****. This implies that ***dynamic_cast*** – which must look into an object to determine its type – cannot cast from a ***void****. For that, a ***static_cast*** is needed. For example:

```
Radio* f(void* p)
{
    Storable* ps = static_cast<Storable*>(p);  // trust the programmer
    return dynamic_cast<Radio*>(ps);
}
```

Both ***dynamic_cast*** and ***static_cast*** respect ***const*** and access controls. For example:

```
class Users : private set<Person> { /* ... */ };
```

```
void f(Users* pu, const Receiver* pcr)
{
    static_cast<set<Person>*>(pu);        // error: access violation
    dynamic_cast<set<Person>*>(pu);       // error: access violation

    static_cast<Receiver*>(pcr);          // error: can't cast away const
    dynamic_cast<Receiver*>(pcr);         // error: can't cast away const

    Receiver* pr = const_cast<Receiver*>(pcr);      // ok
    // ...
}
```

It is not possible to cast to a private base class, and ''casting away ***const***'' requires a ***const_cast*** (§6.2.7). Even then, using the result is safe only provided the object wasn't originally declared ***const*** (§10.2.7.1) .

15.4.3 Class Object Construction and Destruction [hier.class.obj]

A class object is more than simply a region of memory (§4.9.6). A class object is built from ''raw memory'' by its constructors and it reverts to ''raw memory'' as its destructors are executed. Construction is bottom up, destruction is top down, and a class object is an object to the extent that it has been constructed or destroyed. This is reflected in the rules for RTTI, exception handling (§14.4.7), and virtual functions.

It is extremely unwise to rely on details of the order of construction and destruction, but that order can be observed by calling virtual functions, ***dynamic_cast***, or ***typeid*** (§15.4.4) at a point where the object isn't complete. For example, if the constructor for ***Component*** in the hierarchy from §15.4.2 calls a virtual function, it will invoke a version defined for ***Storable*** or ***Component***, but not one from ***Receiver***, ***Transmitter***, or ***Radio***. At that point of construction, the object isn't yet a ***Radio***; it is merely a partially constructed object. It is best to avoid calling virtual functions during construction and destruction.

15.4.4 Typeid and Extended Type Information [hier.typeid]

The ***dynamic_cast*** operator serves most needs for information about the type of an object at run time. Importantly, it ensures that code written using it works correctly with classes derived from those explicitly mentioned by the programmer. Thus, ***dynamic_cast*** preserves flexibility and extensibility in a manner similar to virtual functions.

However, it is occasionally essential to know the exact type of an object. For example, we might like to know the name of the object's class or its layout. The ***typeid*** operator serves this purpose by yielding an object representing the type of its operand. Had ***typeid***( ) been a function, its declaration would have looked something like this:

```
class type_info;
const type_info& typeid(type_name) throw(bad_typeid);    // pseudo declaration
const type_info& typeid(expression);                     // pseudo declaration
```

That is, ***typeid***( ) returns a reference to a standard library type called ***type_info*** defined in ***<type-info>***. Given a *type-name* as its operand, ***typeid***( ) returns a reference to a ***type_info*** that represents the *type-name*. Given an *expression* as its operand, ***typeid***( ) returns a reference to a

type_info that represents the type of the object denoted by the *expression*. A ***typeid*()** is most commonly used to find the type of an object referred to by a reference or a pointer:

```
void f(Shape& r, Shape* p)
{
    typeid(r);      // type of object referred to by r
    typeid(*p);     // type of object pointed to by p
    typeid(p);      // type of pointer, that is, Shape* (uncommon, except as a mistake)
}
```

If the value of a pointer or a reference operand is ***0***, ***typeid*()** throws a ***bad_typeid*** exception.

The implementation-independent part of ***type_info*** looks like this:

```
class type_info {
public:
    virtual ~type_info();                          // is polymorphic

    bool operator==(const type_info&) const;       // can be compared
    bool operator!=(const type_info&) const;
    bool before(const type_info&) const;           // ordering

    const char* name() const;                      // name of type
private:
    type_info(const type_info&);                   // prevent copying
    type_info& operator=(const type_info&);        // prevent copying
    // ...
};
```

The ***before*()** function allows ***type_info***s to be sorted. There is no relation between the relationships defined by ***before*** and inheritance relationships.

It is *not* guaranteed that there is only one ***type_info*** object for each type in the system. In fact, where dynamically linked libraries are used it can be hard for an implementation to avoid duplicate ***type_info*** objects. Consequently, we should use == on ***type_info*** objects to test equality, rather than == on pointers to such objects.

We sometimes want to know the exact type of an object so as to perform some standard service on the whole object (and not just on some base of the object). Ideally, such services are presented as virtual functions so that the exact type needn't be known. In some cases, no common interface can be assumed for every object manipulated, so the detour through the exact type becomes necessary (§15.4.4.1). Another, much simpler, use has been to obtain the name of a class for diagnostic output:

```
#include<typeinfo>

void g(Component* p)
{
    cout << typeid(*p).name();
}
```

The character representation of a class' name is implementation-defined. This C-style string resides in memory owned by the system, so the programmer should not attempt to ***delete*[]** it.

15.4.4.1 Extended Type Information [hier.extended]

Typically, finding the exact type of an object is simply the first step to acquiring and using more-detailed information about that type.

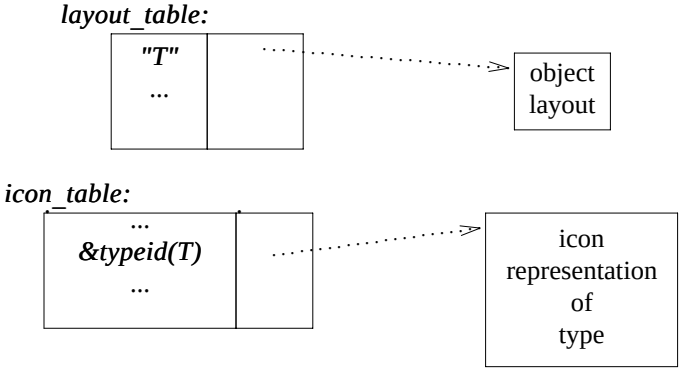
Consider how an implementation or a tool could make information about types available to users at run time. Suppose I have a tool that generates descriptions of object layouts for each class used. I can put these descriptors into a ***map*** to allow user code to find the layout information:

```
map<const char*, Layout> layout_table;

void f(B* p)
{
    Layout& x = layout_table[typeid(*p).name()];
    // use x
}
```

Someone else might provide a completely different kind of information:

```
struct TI_eq {
    bool operator()(const type_info* p, const type_info* q) { return *p==*q; }
};

struct TI_hash {
    int operator()(const type_info* p); // compute hash value (§17.6.2.2)
};

hash_map<type_info*, Icon, hash_fct, TI_hash, TI_eq> icon_table;      // §17.6

void g(B* p)
{
    Icon& i = icon_table[&typeid(*p)];
    // use i
}
```

This way of associating ***typeid***s with information allows several people or tools to associate different information with types totally independently of each other:



This is most important because the likelihood is zero that someone can come up with a single set of information that satisfies every user.

15.4.5 Uses and Misuses of RTTI [hier.misuse]

One should use explicit run-time type information only when necessary. Static (compile-time) checking is safer, implies less overhead, and – where applicable – leads to better-structured programs. For example, RTTI can be used to write thinly disguised *switch-statement*s:

```
// misuse of run-time type information:

void rotate(const Shape& r)
{
    if (typeid(r) == typeid(Circle)) {
        // do nothing
    }
    else if (typeid(r) == typeid(Triangle)) {
        // rotate triangle
    }
    else if (typeid(r) == typeid(Square)) {
        // rotate square
    }
    // ...
}
```

Using ***dynamic_cast*** rather than ***typeid*** would improve this code only marginally.

Unfortunately, this is not a strawman example; such code really does get written. For many people trained in languages such as C, Pascal, Modula-2, and Ada, there is an almost irresistible urge to organize software as a set of *switch-statement*s. This urge should usually be resisted. Use virtual functions (§2.5.5, §12.2.6) rather than RTTI to handle most cases when run-time discrimination based on type is needed.

Many examples of proper use of RTTI arise when some service code is expressed in terms of one class and a user wants to add functionality through derivation. The use of ***Ival_box*** in §15.4 is an example of this. If the user is willing and able to modify the definitions of the library classes, say ***BBwindow***, then the use of RTTI can be avoided; otherwise, it is needed. Even if the user is willing to modify the base classes, such modification may cause its own problems. For example, it may be necessary to introduce dummy implementations of virtual functions in classes for which those functions are not needed or not meaningful. This problem is discussed in some detail in §24.4.3. A use of RTTI to implement a simple object I/O system can be found in §25.4.1.

For people with a background in languages that rely heavily on dynamic type checking, such as Smalltalk or Lisp, it is tempting to use RTTI in conjunction with overly general types. Consider:

```
// misuse of run-time type information:

class Object { /* ... */ }; // polymorphic

class Container : public Object {
public:
    void put(Object*);
    Object* get();
    // ...
};
```

```
class Ship : public Object { /* ... */ };

Ship* f(Ship* ps, Container* c)
{
        c->put(ps);
        // ...
        Object* p = c->get();
        if (Ship* q = dynamic_cast<Ship*>(p)) { // run-time check
                return q;
        }
        else {
                // do something else (typically, error handling)
        }
}
```

Here, class ***Object*** is an unnecessary implementation artifact. It is overly general because it does not correspond to an abstraction in the application domain and forces the application programmer to use an implementation-level abstraction. Problems of this kind are often better solved by using container templates that hold only a single kind of pointer:

```
Ship* f(Ship* ps, list<Ship*>& c)
{
        c.push_front(ps);
        // ...
        return c.pop_front();
}
```

Combined with the use of virtual functions, this technique handles most cases.

15.5 Pointers to Members [hier.ptom]

Many classes provide simple, very general interfaces intended to be invoked in several different ways. For example, many ''object-oriented'' user-interfaces define a set of requests to which every object represented on the screen should be prepared to respond. In addition, such requests can be presented directly or indirectly from programs. Consider a simple variant of this idea:

```
class Std_interface {
public:
        virtual void start() = 0;
        virtual void suspend() = 0;
        virtual void resume() = 0;
        virtual void quit() = 0;
        virtual void full_size() = 0;
        virtual void small() = 0;

        virtual ~Std_interface() {}
};
```

The exact meaning of each operation is defined by the object on which it is invoked. Often, there is a layer of software between the person or program issuing the request and the object receiving it.

Ideally, such intermediate layers of software should not have to know anything about the individual operations such as ***resume***() and ***full_size***(). If they did, the intermediate layers would have to be updated each time the set of operations changed. Consequently, such intermediate layers simply transmit some data representing the operation to be invoked from the source of the request to its recipient.

One simple way of doing that is to send a ***string*** representing the operation to be invoked. For example, to invoke ***suspend***() we could send the string ***"suspend"***. However, someone has to create that string and someone has to decode it to determine to which operation it corresponds – if any. Often, that seems indirect and tedious. Instead, we might simply send an integer representing the operation. For example, *2* might be used to mean ***suspend***(). However, while an integer may be convenient for machines to deal with, it can get pretty obscure for people. We still have to write code to determine that *2* means ***suspend***() and to invoke ***suspend***().

C++ offers a facility for indirectly referring to a member of a class. A pointer to a member is a value that identifies a member of a class. You can think of it as the position of the member in an object of the class, but of course an implementation takes into account the differences between data members, virtual functions, non-virtual functions, etc.

Consider ***Std_interface***. If I want to invoke ***suspend***() for some object without mentioning ***suspend***() directly, I need a pointer to member referring to ***Std_interface***::***suspend***(). I also need a pointer or reference to the object I want to suspend. Consider a trivial example:

```
typedef void (Std_interface::* Pstd_mem)();    // pointer to member type

void f(Std_interface* p)
{
    Pstd_mem s = &Std_interface::suspend;

    p->suspend();        // direct call

    (p->*s)();           // call through pointer to member
}
```

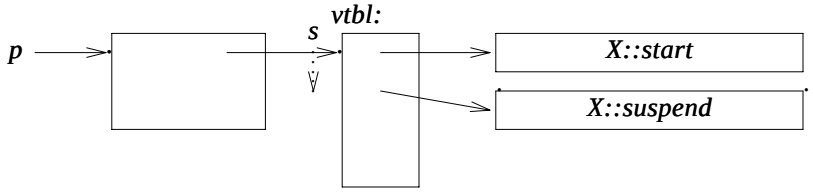
A *pointer to member* can be obtained by applying the address-of operator & to a fully qualified class member name, for example, &***Std_interface***::***suspend***. A variable of type ''pointer to member of class ***X***'' is declared using a declarator of the form ***X***::*.

The use of ***typedef*** to compensate for the lack of readability of the C declarator syntax is typical. However, please note how the ***X***::* declarator matches the traditional * declarator exactly.

A pointer to member ***m*** can be used in combination with an object. The operators ->* and .* allow the programmer to express such combinations. For example, ***p***->***m*** binds ***m*** to the object pointed to by ***p***, and ***obj***.***m*** binds ***m*** to the object ***obj***. The result can be used in accordance with ***m***'s type. It is not possible to store the result of a ->* or a .* operation for later use.

Naturally, if we knew which member we wanted to call we would invoke it directly rather than mess with pointers to members. Just like ordinary pointers to functions, pointers to member functions are used when we need to refer to a function without having to know its name. However, a pointer to member isn't a pointer to a piece of memory the way a pointer to a variable or a pointer to a function is. It is more like an offset into a structure or an index into an array. When a pointer to member is combined with a pointer to an object of the right type, it yields something that identifies a particular member of a particular object.

This can be represented graphically like this:



Because a pointer to a virtual member (*s* in this example) is a kind of offset, it does not depend on an object's location in memory. A pointer to a virtual member can therefore safely be passed between different address spaces as long as the same object layout is used in both. Like pointers to ordinary functions, pointers to non-virtual member functions cannot be exchanged between address spaces.

Note that the function invoked through the pointer to function can be ***virtual***. For example, when we call ***suspend***() through a pointer to function, we get the right ***suspend***() for the object to which the pointer to function is applied. This is an essential aspect of pointers to functions.

An interpreter might use pointers to members to invoke functions presented as strings:

```
map<string,Std_interface*> variable;
map<string,Pstd_mem> operation;

void call_member(string var, string oper)
{
    (variable[var]->*operation[oper])();    // var.oper()
}
```

A critical use of pointers to member functions is found in ***mem_fun***() (§3.8.5, §18.4).

A static member isn't associated with a particular object, so a pointer to a static member is simply an ordinary pointer. For example:

```
class Task {
    // ...
    static void schedule();
};

void (*p)() = &Task::schedule;          // ok
void (Task::* pm)() = &Task::schedule;  // error: ordinary pointer assigned
                                        // to pointer to member
```

Pointers to data members are described in §C.12.

15.5.1 Base and Derived Classes [hier.contravariance]

A derived class has at least the members that it inherits from its base classes. Often it has more. This implies that we can safely assign a pointer to a member of a base class to a pointer to a member of a derived class, but not the other way around. This property is often called *contravariance*. For example:

```
class text : public Std_interface {
public:
        void start();
        void suspend();
        // ...
        virtual void print();
private:
        vector s;
};

void (Std_interface::* pmi)() = &text::print;    // error
void (text::*pmt)() = &Std_interface::start;     // ok
```

This contravariance rule appears to be the opposite of the rule that says we can assign a pointer to a derived class to a pointer to its base class. In fact, both rules exist to preserve the fundamental guarantee that a pointer may never point to an object that doesn't at least have the properties that the pointer promises. In this case, ***Std_interface*::*** can be applied to any ***Std_interface***, and most such objects presumably are not of type ***text***. Consequently, they do not have the member ***text*::*print*** with which we tried to initialize ***pmi***. By refusing the initialization, the compiler saves us from a run-time error.

15.6 Free Store [hier.free]

It is possible to take over memory management for a class by defining ***operator new*()** and ***operator delete*()** (§6.2.6.2). However, replacing the global ***operator new*()** and ***operator delete*()** is not for the fainthearted. After all, someone else might rely on some aspect of the default behavior or might even have supplied other versions of these functions.

A more selective, and often better, approach is to supply these operations for a specific class. This class might be the base for many derived classes. For example, we might like to have the ***Employee*** class from §12.2.6 provide a specialized allocator and deallocator for itself and all of its derived classes:

```
class Employee {
        // ...
public:
        // ...
        void* operator new(size_t);
        void operator delete(void*, size_t);
};
```

Member ***operator new*()**s and ***operator delete*()**s are implicitly ***static*** members. Consequently, they don't have a ***this*** pointer and do not modify an object. They provide storage that a constructor can initialize and a destructor can clean up.

```
void* Employee::operator new(size_t s)
{
        // allocate 's' bytes of memory and return a pointer to it
}
```

```
void Employee::operator delete(void* p, size_t s)
{
    // assume 'p' points to 's' bytes of memory allocated by Employee::operator new()
    // and free that memory for reuse
}
```

The use of the hitherto mysterious ***size_t*** argument now becomes obvious. It is the size of the object actually deleted. Deleting a ''plain'' ***Employee*** gives an argument value of ***sizeof(Employee)***; deleting a ***Manager*** gives an argument value of ***sizeof(Manager)***. This allows a class-specific allocator to avoid storing size information with each allocation. Naturally, a class-specific allocator can store such information (like a general-purpose allocator must) and ignore the ***size_t*** argument to ***operator delete()***. However, that makes it harder to improve significantly on the speed and memory consumption of a general-purpose allocator.

How does a compiler know how to supply the right size to ***operator delete()***? As long as the type specified in the ***delete*** operation matches the actual type of the object, this is easy. However, that is not always the case:

```
class Manager : public Employee {
    int level;
    // ...
};

void f()
{
    Employee* p = new Manager;    // trouble (the exact type is lost)
    delete p;
}
```

In this case, the compiler will not get the size right. As when an array is deleted, the user must help. This is done by adding a virtual destructor to the base class, ***Employee***:

```
class Employee {
public:
    void* operator new(size_t);
    void operator delete(void*, size_t);
    virtual ~Employee();
    // ...
};
```

Even an empty destructor will do:

```
Employee::~Employee() { }
```

In principle, deallocation is then done from within the destructor (which knows the size). Furthermore, the presence of a destructor in ***Employee*** ensures that every class derived from it will be supplied with a destructor (thus getting the size right), even if the derived class doesn't have a user-defined destructor. For example:

```
void f()
{
    Employee* p = new Manager;
    delete p;        // now fine (Employee is polymorphic)
}
```

Allocation is done by a (compiler-generated) call:

```
Employee::operator new(sizeof(Manager))
```

and deallocation by a (compiler-generated) call:

```
Employee::operator delete(p,sizeof(Manager))
```

In other words, if you want to supply an allocator/deallocator pair that works correctly for derived classes, you must either supply a virtual destructor in the base class or refrain from using the ***size_t*** argument in the deallocator. Naturally, the language could have been designed to save you from such concerns. However, that can be done only by also ''saving'' you from the benefits of the optimizations possible in the less safe system.

15.6.1 Array Allocation [hier.array]

The ***operator new()*** and ***operator delete()*** functions allow a user to take over allocation and deallocation of individual objects; ***operator new[]()*** and ***operator delete[]()*** serve exactly the same role for the allocation and deallocation of arrays. For example:

```
class Employee {
public:
    void* operator new[](size_t);
    void operator delete[](void*, size_t);
    // ...
};

void f(int s)
{
    Employee* p = new Employee[s];
    // ...
    delete[] p;
}
```

Here, the memory needed will be obtained by a call,

```
Employee::operator new[](sizeof(Employee)*s+delta)
```

where ***delta*** is some minimal implementation-defined overhead, and released by a call:

```
Employee::operator delete[](p,s*sizeof(Employee)+delta)
```

The number of elements (***s***) is ''remembered'' by the system.

15.6.2 "Virtual Constructors" [hier.vctor]

After hearing about virtual destructors, the obvious question is, ''Can constructors be virtual?'' The short answer is no; a slightly longer one is, no, but you can easily get the effect you are looking for.

To construct an object, a constructor needs the exact type of the object it is to create. Consequently, a constructor cannot be virtual. Furthermore, a constructor is not quite an ordinary function. In particular, it interacts with memory management routines in ways ordinary member functions don't. Consequently, you cannot have a pointer to a constructor.

Both of these restrictions can be circumvented by defining a function that calls a constructor and returns a constructed object. This is fortunate because creating a new object without knowing its exact type is often useful. The *Ival_box_maker* (§12.4.4) is an example of a class designed specifically to do that. Here, I present a different variant of that idea, where objects of a class can provide users with a clone (copy) of themselves or a new object of their type. Consider:

```
class Expr {
public:
        Expr();                         // default constructor
        Expr(const Expr&);              // copy constructor

        virtual Expr* new_expr() { return new Expr(); }
        virtual Expr* clone() { return new Expr(*this); }
        // ...
};
```

Because functions such as *new_expr()* and *clone()* are virtual and they (indirectly) construct objects, they are often called ''virtual constructors'' – by a strange misuse of the English language. Each simply uses a constructor to create a suitable object.

A derived class can override *new_expr()* and/or *clone()* to return an object of its own type:

```
class Cond : public Expr {
public:
        Cond();
        Cond(const Cond&);

        Cond* new_expr() { return new Cond(); }
        Cond* clone() { return new Cond(*this); }
        // ...
};
```

This means that given an object of class *Expr*, a user can create a new object of ''just the same type.'' For example:

```
void user(Expr* p)
{
        Expr* p2 = p->new_expr();
        // ...
}
```

The pointer assigned to *p2* is of an appropriate, but unknown, type.

The return type of *Cond::new_expr()* and *Cond::clone()* was *Cond** rather than *Expr**.

This allows a *Cond* to be cloned without loss of type information. For example:

```
void user2(Cond* pc, Expr* pe)
{
    Cond* p2 = pc->clone();
    Cond* p3 = pe->clone();  // error
    // ...
}
```

The type of an overriding function must be the same as the type of the virtual function it overrides, except that the return type may be relaxed. That is, if the original return type was *B**, then the return type of the overriding function may be *D**, provided *B* is a public base of *D*. Similarly, a return type of *B*& may be relaxed to *D*&.

Note that a similar relaxation of the rules for argument types would lead to type violations (see §15.8 [12]).

15.7 Advice [hier.advice]

[1] Use ordinary multiple inheritance to express a union of features; §15.2, §15.2.5.
[2] Use multiple inheritance to separate implementation details from an interface; §15.2.5.
[3] Use a *virtual* base to represent something common to some, but not all, classes in a hierarchy; §15.2.5.
[4] Avoid explicit type conversion (casts); §15.4.5.
[5] Use *dynamic_cast* where class hierarchy navigation is unavoidable; §15.4.1.
[6] Prefer *dynamic_cast* over *typeid*; §15.4.4.
[7] Prefer *private* to *protected*; §15.3.1.1.
[8] Don't declare data members *protected*; §15.3.1.1.
[9] If a class defines *operator delete*(), it should have a virtual destructor; §15.6.
[10] Don't call virtual functions during construction or destruction; §15.4.3.
[11] Use explicit qualification for resolution of member names sparingly and preferably use it in overriding functions; §15.2.1

15.8 Exercises [hier.exercises]

1. (∗1) Write a template *ptr_cast* that works like *dynamic_cast*, except that it throws *bad_cast* rather than returning *0*.
2. (∗2) Write a program that illustrates the sequence of constructor calls at the state of an object relative to RTTI during construction. Similarly illustrate destruction.
3. (∗3.5) Implement a version of a Reversi/Othello board game. Each player can be either a human or the computer. Focus on getting the program correct and (then) getting the computer player ''smart'' enough to be worth playing against.
4. (∗3) Improve the user interface of the game from §15.8[3].
5. (∗3) Define a graphical object class with a plausible set of operations to serve as a common base class for a library of graphical objects; look at a graphics library to see what operations were

supplied there. Define a database object class with a plausible set of operations to serve as a common base class for objects stored as sequences of fields in a database; look at a database library to see what operations were supplied there. Define a graphical database object with and without the use of multiple inheritance and discuss the relative merits of the two solutions.

6. (∗2) Write a version of the ***clone()*** operation from §15.6.2 that can place its cloned object in an ***Arena*** (see §10.4.11) passed as an argument. Implement a simple ***Arena*** as a class derived from ***Arena***.
7. (∗2) Without looking in the book, write down as many C++ keywords you can.
8. (∗2) Write a standards-conforming C++ program containing a sequence of at least ten consecutive keywords not separated by identifiers, operators, punctuation characters, etc.
9. (∗2.5) Draw a plausible memory layout for a ***Radio*** as defined in §15.2.3.1. Explain how a virtual function call could be implemented.
10. (∗2) Draw a plausible memory layout for a ***Radio*** as defined in §15.2.4. Explain how a virtual function call could be implemented.
11. (∗3) Consider how ***dynamic_cast*** might be implemented. Define and implement a ***dcast*** template that behaves like ***dynamic_cast*** but relies on functions and data you define only. Make sure that you can add new classes to the system without having to change the definitions of ***dcast*** or previously-written classes.
12. (∗2) Assume that the type-checking rules for arguments were relaxed in a way similar to the relaxation for return types so that a function taking a ***Derived**** could overwrite a ***Base****. Then write a program that would corrupt an object of class ***Derived*** without using a cast. Describe a safe relaxation of the overriding rules for argument types.