```
                                    int starting_height)
                    : Shape(starting_location), _height(starting_height)
{
}

int Vertical_line::height() const
{
     return _height;
}

void Vertical_line::change_height(int new_height)
{
     _height = new_height;
}

void Vertical_line::draw()
{
     draw_with_char('|');
}

void Vertical_line::draw_with_char(Display_char ch)
{
     Display.add_line(location(), location() + Point(0, height()), ch);
}

Rectangle::Rectangle(const Point &starting_location,
                     const Point &starting_size)
                    : Shape(starting_location), _size(starting_size)
{
}

Point Rectangle::size() const
{
     return _size;
}

void Rectangle::change_size(const Point &new_size)
{
     _size = new_size;
}

void Rectangle::draw()
{
     Display.add_line(location(), location() + Point(size().x(), 0), '-');
     Display.add_line(location(), location() + Point(0, size().y()), '|');
     Display.add_line(location() + size(), location() + Point(size().x(), 0), '|');
     Display.add_line(location() + size(), location() + Point(0, size().y()), '-');
}

void Rectangle::draw_with_char(Display_char ch)
{
     Display.add_line(location(), location() + Point(size().x(), 0), ch);
     Display.add_line(location(), location() + Point(0, size().y()), ch);
     Display.add_line(location() + size(), location() + Point(size().x(), 0), ch);
     Display.add_line(location() + size(), location() + Point(0, size().y()), ch);
}
```

FILE: **shapes.c**

```
#include "Shapes.h"
#include "Screen.h"
```

UNIT 17

Lab Exercises (Answers)

1. Since **Window** is publicly derived from **Display_medium** and **Display_object**, and **Bottom_Window** is publicly derived from **Window**, we can use a **Bottom_Window** as an argument to either **number_lines** (which has a **Window *** parameter), **say_hello** (which has a **Display_medium** parameter), or **bounce** (which has a **Display_object** parameter), without any further changes to our code.

2. The first change we must make to our shape classes is to derive them from class **Display_object**. This will allow the use of a shape whenever a **Display_object** is needed. If we list **Display_object** as a base of **Shape**, all shapes will be derived from **Display_object** indirectly. If we made only this change, we would not see a **Rectangle** bouncing up and down on the screen, because bounce calls move, which adjusts the **_location** member without displaying the **Rectangle** on the screen.

   To make the **Rectangle** bounce up and down on the screen, we must modify the **move** function so that it erases the **Rectangle**, adjusts the **_location**, and then draws the **Rectangle** again. This can be done most easily by adding an **erase** member function, and calling it from the new **move**:

```
void Shape::move(const Point &new_location)
{
    erase();
    _location = new_location;
    draw();
}
```

To make shapes consistent with class **Display_object**, we must modify the **move** function so that it erases the shape, changes its location, and draws the shape at the new position. A shape can be erased by drawing over it with blank spaces. If we change the way shapes behave, existing code that uses shapes (such as *copies.c*) will no longer work. This is an inescapable result of making an incompatible change.

The *Shapes.h* header file changes necessary to implement the above are:

1. Add a **#include** of "Display_obj.h" to the *Shapes.h* header file.

```
main(int, char *[])
{
     Window w(Point(2, 1),
              Point(30, 20),
              "A Window");

     Bottom_Window bw(Point(35, 1),
                      Point(30, 20),
                      "A Bottom_Window");

     number_lines(&w);
     number_lines(&bw);

     sleep(2);
     say_hello(w);
     say_hello(bw);

     sleep(2);
     bounce(w);
     bounce(bw);

     sleep(2);
     return 0;
}
```

---

2. This question does not use multiple inheritance, but it does illustrate some of the subtle issues involved in using inheritance, whether single or multiple.

   Change to the *unit17/shapes* directory. The files *Display_obj.h*, *bounce.h*, *bounce.c*, and *bounce_rect.c* have been added to your *unit17/shapes* directory. Change your shape classes (**Shape**, **Rectangle**, etc.) so that the program *bounce_rect.c* (shown following the SUMMARY) will show a rectangle bouncing up and down on the screen. Getting this to work is harder than it may seem at first.

   After making the changes, you can compile and execute the test program by entering '**make prob2**' or you can compile and execute it directly using the commands:

```
$ CC -I../../pre_windows shapes.c \
     bounce_rect.c bounce.c -L../../pre_windows \
     -lpre_wind -lcurses -o bounce_rect
$ bounce_rect
```

   HINT: If class Shape is to be publicly derived from **Display_object**, class **Shape** must work in a way that is consistent with a **Display_object** (see page 17-10 in the Student Guide). The shape classes should already obey the rule "**location** returns the point the object was last moved to," but the shapes are inconsistent with class **Display_object** in a more subtle way. When we call the **move** function for a **Display_object**, the **Display_object** will disappear from its original location on the screen, and re-appear at the new location. The move function for a Shape simply updates the **_location** member without re-drawing the shape.

UNIT 17

Lab Exercises

1. Change to the *unit17/window* directory. In unit 17, class **Window** is derived from both class **Display_medium** and class **Display_object**, so **Bottom_Window** is indirectly derived from these classes. The files *Bot_Wind.h* and *bot_wind.c* are copies of the solutions from the *unit13/windows* directory. Compile and execute the test program *use_bot_w.c*. Do you need to change either of your files to make the test program work?

   You can compile and execute this program by entering '**make prob1**' or you can compile and execute it directly using the commands:

```
$ CC -I../../pre_wind use_bot_w.c bot_wind.c \
     say_hello.c bounce.c window.c -L../../pre_wind \
     -lpre_wind -lcurses -o use_bot_w
$ use_bot_w
```

SUMMARY	
DIRECTORY	unit17/window
DECLARATION	pre_windows/Point.h, pre_windows/Screen.h, pre_windows/max.h, pre_windows/Display_med.h, Window.h, Display_obj.h, Bot_Wind.h, bounce.h, say_hello.h
IMPLEMENTATION	pre_windows/libpre_wind.a, window.c, bot_wind.c, bounce.c, say_hello.c
TEST PROGRAM	use_bot_w.c

FILE: **use_bot_w.c**

```
#include "say_hello.h"
#include "bounce.h"
#include "Bot_Wind.h"
#include <libc.h>
#include <stdio.h>

void number_lines(Window *w)
{
     int i;
     char buf[4];

     for (i=0; i<w->size().y(); i++) {
          w->move_cursor(Point(0, i));
          sprintf(buf, "%3d", i);
          w->add(buf);
     }
}
```

Lab Exercises

Lab Exercises

Summary

A class may have more than one base.

- class should do what is expected of each public base
- ambiguities may arise
  - — resolved in each call with ::
  - — prevented with overridding function

Summary

Preventing Ambiguities

```
class Display_medium {
public:
    virtual void clear();
};

class Display_object {
public:
    virtual void clear();
};


class Window : public Display_medium, public Display_object {
public:
    void clear();
};

void Window::clear()
{
    Display_medium::clear();
    Display_object::clear();
}


main(int, char *[])
{
    Window w(Point(1, 1), Point(10, 10), "test");

    w.clear();

    return 0;
}
```

Preventing Ambiguities

Alternatively, the ambiguity can be avoided if the function is overridden in the derived class. There are some situations in which the compiler forces the author of a class to avoid potential ambiguities in this way[1]. If the derived class overrides the base class functions, then the derived class function will be used for derived class objects in functions like `blank` (unless the keyword **virtual** was not used in the base class member function declarations).

We recommend overriding functions that are inherited from more than one base class unless there is some reason why the user should be forced to choose one function or the other. If we override potentially ambiguous functions while creating the derived class, we not only make coding easier on the users, but also avoid a potential maintenance problem. If a derived class does not override the function, and later we need to add an overriding function, the users will have to go back and remove the explicit references to base class functions from their code.

1. We will learn about these situations in the next unit.

Resolving Ambiguities

```
class Display_medium {
public:
    virtual void clear();
};

class Display_object {
public:
    virtual void clear();
};


class Window : public Display_medium, public Display_object {
public:
    // No clear function
};


main(int, char *[])
{
    Window w(Point(1, 1), Point(10, 10), "test");

    w.Display_medium::clear();
    w.Display_object::clear();

    return 0;
}
```

Resolving Ambiguities

Such ambiguities can be resolved with the scope resolution operator, when the function is called. If the function is called with a base class pointer or reference, the scope resolution operator is not needed:

```
example(Display_object &d_o)
{
     d_o.clear();   // Display_object::clear
}
```

The `example` function will call the `Display_medium::clear` function if called with a `Window` argument.

Ambiguities

```
class Display_medium {
public:
    virtual void clear();
};

class Display_object {
public:
    virtual void clear();
};


class Window : public Display_medium, public Display_object {
public:
    // No clear function
};


main(int, char *[])
{
    Window w(Point(1, 1), Point(10, 10), "test");

    w.clear(); // error -- which clear?

    return 0;
}
```

Ambiguities

If a class inherits several functions with the same name and parameter types from different bases, calls to those functions may be ambiguous. For example, if both `Display_medium` and `Display_object` had a `clear` function, the call to `clear` shown on the facing page would be ambiguous, and therefore illegal.

Using Windows

```
#include "say_hello.h"
#include "bounce.h"
#include "Window.h"
#include <stdlib.h>

main(int, char *[])
{
    Window w(Point(2, 2), Point(60, 10), "test");

    say_hello(w);
    sleep(2);

    bounce(w);
    sleep(2);

    return 0;
}
```

Using Windows

Since `Window` is derived publicly from both `Display_medium` and `Display_object`, a `Window` can be used when either a `Display_medium` or a `Display_object` is expected. Since class `Window` obeys the abstraction invariants of both of its base classes, functions written for those bases will work with `Window` arguments.

```
void say_hello(Display_medium &m)
{
     m.add("hello, world\n");
}

void bounce(Display_object &d)
{
    Point at = d.location();
    Point up = at - Point(0, 2);
    Point down = at + Point(0, 2);

    d.move(up);
    d.move(down);
    d.move(up);
    d.move(down);
    d.move(at);
}
```

Window::location

```
#include "Window.h"

Point Window::location() const
{
    return upper_left();
}
```

Window::location

When we override a member function, we must be consistent with both the declaration and the meaning of the base class function. Even if the base class function is a pure virtual function (with no implementation), the base class's documentation may still describe, on an abstract level, what the functions will do. Such a rule is often called an *abstraction invariant,* because it is defined in terms of the abstraction, not any particular representation. Abstraction invariants may be stated explicitly in comments or documentation, or they may be implicit in the programmer's understanding of the class. If the abstraction invariants are not stated explicitly, we run the risk that not all programmers will have the same understanding of what the class is supposed to do.

The `location` function must return the last legal value given as an argument to `move`, no matter how the derived classes implement `location` and `move`. We must, therefore, define a window's `location` as its upper left corner.

If it is not possible to define our class function in a way that is consistent with the abstraction invariant of the base class, then our class should not be derived from that base. If we did derive it from the base, it could be passed to functions requiring base type arguments, and such functions often rely on the base's abstraction invariant.

```
class Display_object {
public:
    virtual void move(const Point &new_location) = 0;
    virtual Point location() const = 0;
  // location returns the point the object was last moved to
};
```

Class Window Revisited

```
#include "Display_med.h"
#include "Display_obj.h"

class Window : public Display_medium, public Display_object {
public:
    Window(const Point &upper_left,
        const Point &size,
        const String &title);
    ~Window();

    void move(const Point &new_upper_left);
  Point location() const;
    Point upper_left() const;
    Point lower_right() const;

    Point size() const;
    virtual void change_size(const Point &new_size);

    int move_cursor(const Point &where);
    Point cursor() const;

    Display_char character() const;  // char under cursor
    String line() const;            // line cursor is on
    void add(Display_char c);        // put c in window
    void add(const String &str);     // put str in window
    void clear();

    void scroll_up();
    void scroll_down();
private:
};
```

Class Window Revisited

To derive class `Window` from two bases, we simply list both bases, separated by a comma, in the declaration of class `Window`. Each base class may be either public or private. Class `Window` must override any pure virtual functions of its base. Class `Window` already provides all the functions required by `Display_medium`, and the `move` function required by `Display_object`. We need only add a `location` function to our class `Window` to make it compatible with its abstract base `Display_object`.

```
#include "Point.h"
#include "String.h"
#include "Display_ch.h"

class Display_medium {
public:
    virtual Point size() const = 0;

    virtual Point cursor() const = 0;
    virtual int move_cursor(const Point &p) = 0;

    virtual Display_char character() const = 0 ;
    virtual String line() const = 0 ;

    virtual void add(Display_char ch) = 0;
    virtual void add(const String &s) = 0;

    virtual void add_line(const Point &start,
                    const Point &end,
                    Display_char ch);
    virtual void clear();

private:

};

class Display_object {
public:
    virtual void move(const Point &new_location) = 0;
    virtual Point location() const = 0;
   // location returns the point the object was last moved to
};
```

Class Display_object

```
class Display_object {
public:
     virtual void move(const Point &new_location) = 0;
     virtual Point location() const = 0;
   // location returns the point the object was last moved to
};

void bounce(Display_object &d)
{
     Point at = d.location();
     Point up = at - Point(0, 2);
     Point down = at + Point(0, 2);

     d.move(up);
     d.move(down);
     d.move(up);
     d.move(down);
     d.move(at);
}
```

Class Display_object

All objects that can be displayed must have a location, and they can be moved around. The abstract class `Display_object` lets us write polymorphic functions to move any kind of object around on the screen.

Multiple Inheritance

A class may have more than one immediate base

`Window` can be derived from:

- class `Display_medium`
- class `Display_object`

Multiple Inheritance

Once we develop a rich set of base classes, we may find that a class that we are writing can be derived from more than one base. For example, our class library might contain both the class `Display_medium`, that we created earlier, and also a class `Display_object`, to group the common features of all kinds of objects that can be displayed (such as text and graphics).

We will see that both `Display_medium` and `Display_object` could be used as base classes for `Window`. A window is a kind of `Display_medium` because we display things on it, and it is a kind of `Display_object` because the windows are themselves displayed on the terminal screen.

Objectives

At the end of this unit we will be able to:

- Derive a class from more than one base
- Resolve ambiguities if two base's members have the same name

CONTENTS

Unit 17 - Multiple Inheritance

Unit 17

Object-Oriented Programming in C++

Multiple Inheritance