```
                : 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()), '-');
}
```

```
{
     return _location;
}

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

void Shape::draw()
{
}


Horizontal_line::Horizontal_line(const Point &starting_location,
                                 int starting_width)
                : Shape(starting_location), _width(starting_width)
{
}

int Horizontal_line::width() const
{
     return _width;
}

void Horizontal_line::change_width(int new_width)
{
     _width = new_width;
}

void Horizontal_line::draw()
{
     Display.add_line(location(), location() + Point(width(), 0), '-');
}


Vertical_line::Vertical_line(const Point &starting_location,
                             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()
{
     Display.add_line(location(), location() + Point(0, height()), '|');
}


Rectangle::Rectangle(const Point &starting_location,
                     const Point &starting_size)
```

```
};

class Horizontal_line : public Shape {
public:
     Horizontal_line(const Point &starting_location,
                     int starting_length);

     int width() const;
     void change_width(int new_width);

     void draw();

private:
     int _width;
};


class Vertical_line : public Shape {
public:
     Vertical_line(const Point &starting_location,
                   int starting_height);

     int height() const;
     void change_height(int new_height);

     void draw();

private:
     int _height;
};

class Rectangle : public Shape {
public:
     Rectangle(const Point &starting_location,
               const Point &starting_size);

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

     void draw();

private:
     Point _size;
};
```

FILE: **shapes.c**

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


Shape::Shape(const Point &starting_location)
            : _location(starting_location)
{
}

Point Shape::location() const
```

FILE: **bot_wind.c**

```
#include "Bot_Wind.h"

void Bottom_Window::change_size(const Point &new_size)
{
     int shrinkage = size().y() - new_size.y();
     int i;
     for (i=0; i<shrinkage; i++) {
         scroll_up();
     }

     Window::change_size(new_size);
}

Bottom_Window::Bottom_Window(const Point &upper_left,
                             const Point &size,
                             const String &title)
          : Window(upper_left, size, title)
{
}
```

4. This exercise creates a base class and several derived classes.

   a. The program *use_screen.c* is shown below:

FILE: **use_screen.c**

```
#include "Screen.h"

main(int, char *[])
{
     Display.add_line(Point(5, 5), Point(50, 12), '*');
     return 0;
}
```

   b-f. The completed classes are declared in *Shapes.h*, and the member functions are written in *shapes.c*, both of which are shown below:

FILE: **Shapes.h**

```
#include "Point.h"

class Shape {
public:
     Shape(const Point &starting_location);

     Point location() const;
     void move(const Point &new_location);

     void draw();

private:
     Point _location;
```

UNIT 12

Lab Exercises (Answers)

1. The types **int** and **float** have many common operations, but the only way to take advantage of this fact is with a macro that depends on the common operations:

   #define average(a, b) ((a) + (b) / 2)

   There is no way to write a single function that works with either **int**s or **float**s. This is analogous to our first version of the window system, which had three independent classes (and there was no way to write a single function that worked with any kind of window).

2. UNIX file descriptors provide a single type that represents three concepts: file I/O, inter-process communication, and terminal I/O. This is analogous to our attempt to represent all kinds of windows with a single class, and shares one of the main drawbacks of that scheme: it is not possible to have the compiler check to see that the right kind of argument is used:

```
// The following function moves backward in a file, to allow
// later re-reading of input. It will work with files, but
// not pipes or terminals. Unfortunately, there is no way for
// the compiler to ensure that it is called only with file
// descriptors that correspond to files.

back_up_n_bytes(int fd, long n_bytes)
{
    lseek(fd, -n_bytes, 1);
}
```

3. Class **Bottom_Window** can be written by deriving it from **Window** and overriding the **change_size** member function, so that the contents of a **Bottom_Window** will scroll up before it shrinks. Class **Bottom_Window** will also need a constructor, and that constructor must provide arguments for the base class constructor (as is noted on the last paragraph of the page "Derived Class Constructors" (page 12-36 in the Student Guide). It passes its arguments to the base class constructor without modifying them.

FILE: **Bot_Wind.h**

```
#include "Window.h"

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

    void change_size(const Point &extent);
};
```

```
    h.draw();
    v.draw();

    if (r.location() != Point(25, 2) ||
        h.location() != Point(23, 20) ||
        v.location() != Point(45, 12))
    {
         fprintf(stderr, "INCONSISTENT LOCATION.\n");
         return 2;
    }

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

classes with the program *use_shapes.c* (in the *unit12/shapes* directory).

You can compile and execute this program by entering '**make prob4f**' or you can compile and execute it directly using the commands:

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

The **-I**, **-L**, and **-l** options will cause C++ to search the *pre_windows* directory for header files and libraries.

SUMMARY	
DIRECTORY	unit12/shapes
DECLARATION	pre_windows/*.h, Shapes.h (modify)
IMPLEMENTATION	pre_windows/libpre_wind.a, shapes.c (modify)
TEST PROGRAM	use_shapes.c

FILE: **use_shapes.c**

```
#include "Shapes.h"
#include "Screen.h"
#include <stdlib.h>


main(int, char *[])
{
     Rectangle r(Point(12, 8), Point(8, 4));
     Horizontal_line h(Point(40, 8), 20);
     Vertical_line v(Point(8, 10), 8);

     r.draw();
     h.draw();
     v.draw();

     if (r.location() != Point(12, 8) ||
         h.location() != Point(40, 8) ||
         v.location() != Point(8, 10))
     {
          fprintf(stderr, "INCONSISTENT LOCATION.\n");
          return 1;
     }

     sleep(2);
     r.move(Point(25, 2));
     h.move(Point(23, 20));
     v.move(Point(45, 12));

     Display.clear();
     r.draw();
```

can compile and execute it directly using the commands:

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

The **-I**, **-L**, and **-l** options will cause C++ to search the *pre_windows* directory for header files and libraries.

SUMMARY	
DIRECTORY	unit12/shapes
DECLARATION	pre_windows/*.h, Shapes.h (modify)
IMPLEMENTATION	pre_windows/libpre_wind.a, shapes.c (modify)
TEST PROGRAM	use_hline.c

FILE: **use_hline.c**

```
#include "Shapes.h"
#include "Screen.h"
#include <stdlib.h>

main(int, char *[])
{
     Horizontal_line h(Point(40, 8), 20);

     h.draw();

     if (h.location() != Point(40, 8))
     {
          fprintf(stderr, "INCONSISTENT LOCATION.\n");
          return 1;
     }

     h.move(Point(23, 20));
     Display.clear();
     h.draw();

     if (h.location() != Point(23, 20))
     {
          fprintf(stderr, "INCONSISTENT LOCATION.\n");
          return 2;
     }

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

f. Write the classes **Vertical_line** and **Rectangle**. Put the class declarations in *Shapes.h* and the code for the member functions in *shapes.c*. Test your new

b. Write the public section of the base class in the file *Shapes.h*, in the *unit12/shapes* directory. The base class should be named **Shape**, and should contain all the member functions that are common to the three classes in *Shapes3.h*. Do not worry about the private section, or about writing the member functions yet. If you like, you may use the file *Shapes3.h* (in your *unit12/shapes* directory) as a starting point. Although there is no test program for this exercise, you can check the syntax of your class by creating a file *shapes.c* that contains one line:

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

and compiling that file. You can compile the program by entering '**make prob4b**' or '**make shapes.o**' or you can compile it directly using the command:

```
$ CC -c shapes.c
```

c. What data can we put in the private section of the base class? Write the private section of the base class, and the base class member functions. If you have a **draw** function in the base class, and are unsure what to put there, simply create a function that does nothing. Put the source code for the member functions in the file *shapes.c*. This class is not yet complete enough to run, but you can check the syntax by compiling the *shapes.c* file. You can compile the program by entering '**make prob4c**' or '**make shapes.o**' or you can compile it directly using the command:

```
$ CC -c shapes.c
```

d. Write the public section for class **Horizontal_line** in the file *Shapes.h*. Check the syntax by compiling the *shapes.c* file. You can compile the program by entering '**make prob4d**' or '**make shapes.o**' or you can compile it directly using the command:

```
$ CC -c shapes.c
```

e. Write the private section and the member functions for class **Horizontal_line**. The member functions should be added to the file *shapes.c*. Remember that all the members of the base class will be inherited by a derived class, so you should not duplicate the data defined in the base class. Use the defining operations of class **Screen** to draw the line on the global variable **Display**. Test the class **Horizontal_line** with the program *use_hline.c* in the *unit12/shapes* directory.

You can compile and execute this program by entering '**make prob4e**' or you

f of this question guide you through this process one step at a time.

FILE: **Shapes3.h**

```
#include "Point.h"

class Rectangle {
public:
     Rectangle(const Point &starting_location,
               const Point &starting_size);
     Point location();
     Point size();

     void move(const Point &new_location);
     void change_size(const Point &new_size);

     void draw();

private:
     Point _location;
     Point _size;
};

class Horizontal_line {
public:
     Horizontal_line(const Point &starting_location,
                     int starting_width);
     Point location();
     int width();

     void move(const Point &new_location);
     void change_width(int new_width);

     void draw();

private:
     Point _location;
     int _width;
};

class Vertical_line {
public:
     Vertical_line(const Point &starting_location,
                   int starting_height);
     Point location();
     int height();

     void move(const Point &new_location);
     void change_height(int new_height);

     void draw();

private:
     Point _location;
     int _height;
};
```

a. This exercise will focus on a graphics program that draws different kinds of shapes on the screen. You should be in the *unit12/shapes* directory and do all the work for all parts of exercise 4 in that directory.

Before starting to work on the class **shapes**, you should familiarize yourself with the class **Screen**, which provides access to the terminal screen. Class **Screen** is declared in the file *Screen.h*, in the *pre_windows* directory. Write a test program, named *use_screen.c*, in the *unit12/shapes* directory. Your program should use the **add_line** function to draw a line on the global variable **Display**, which represents the terminal used to run the program. **Display** is a variable of type **Screen** that is declared in *Screen.h*. Therefore, any file that includes *Screen.h*, can use any of class **Screen**'s member and friend functions to work with **Display**.

You can compile and execute your test program by entering '**make prob4a**' or you can compile and execute it directly using the commands:

```
$ CC -I../../pre_windows use_screen.c \
        -L../../pre_windows -lpre_wind \
        -lcurses -o use_screen
$ use_screen
```

The **-I**, **-L**, and **-l** options will cause C++ to search the *pre_windows* directory for header files and libraries.

SUMMARY	
DIRECTORY	unit12/shapes
DECLARATION	pre_windows/*.h
IMPLEMENTATION	pre_windows/libpre_wind.a
TEST PROGRAM	use_screen.c (new)

Now that you have had a chance to learn about class **Screen**, you can start to work on classes representing the shapes that will be drawn on the screen by our test program. A real graphics program would probably be able to draw circles, squares, ellipses, rectangles, and many other shapes. It would probably also allow the rotation of shapes to any angle. In this exercise, our test program will draw only rectangles, horizontal lines, and vertical lines, and it will always draw rectangles with two vertical and two horizontal sides.

In this exercise, you will create classes representing horizontal lines, vertical lines, and rectangles. If you did not know about inheritance, you might create three separate classes, as in the file *Shapes3.h* (which is shown below). Now that you know about inheritance, you can define the features that are common to these classes in a base class, and derive the other classes from the base. Parts *b* through

```
    char buf[4];

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

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);

    w.change_size(Point(20, 12));
    bw.change_size(Point(20, 12));
    sleep(2);

    return 0;
}
```

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

```
$ CC -I../../pre_windows use_bot_w.c bot_wind.c \
        window.c -L../../pre_windows -lpre_wind \
        -lcurses -ltermcap -o use_bot_w
$ use_bot_w
```

SUMMARY	
DIRECTORY	unit12/window
DECLARATION	pre_windows/*.h, Window.h, Bot_Wind.h (new)
IMPLEMENTATION	pre_windows/libpre_wind.a, window.c, bot_wind.c (new)
TEST PROGRAM	use_bot_w.c

4. Change to the *unit12/shapes* directory. In this lab exercise, you will create a base class and several derived classes. The question is divided into several parts, to help you approach the problem one step at a time. You may want to read all parts before starting part *a*.

UNIT 12

Lab Exercises

1. The C types **int** and **float** have many operations in common. Is there any way for a programmer to take advantage of these similarities? How does this relate to our discussion of related types?

2. In the UNIX operating system, files, devices, and inter-process communication channels (pipes) are all represented by a single type of value (known as a file descriptor). This provides one of the major advantages of the UNIX system over operating systems that provide three different mechanisms. It is easy to create functions or even entire programs that can read and write from files, devices (such as the user terminal), or pipes. How does this fit into our discussion of related types in this section? What is one drawback of this setup?

3. Change to the *unit12/window* directory. In this exercise, you will derive a new class, **Bottom_Window**, from an existing base, **Window**. All the files for class **Window** are in the *unit12/window* directory: the declaration is in the *Window.h* header file and the implementation is in file *window.c*. Put the new class declaration in the file *Bot_Wind.h*, and the member functions in *bot_wind.c*. When you have finished creating your new class, test it with the program *use_bot_w.c*.

   A **Bottom_Window** is different from other kinds of windows in the following way: if a **Bottom_Window** shrinks in size, it will preserve the text on the bottom part of the window (rather than the top). For example, if we shrink a regular **Window** that was 20 lines high, so that it becomes only 10 lines high, we will only be able to see the text that appeared on the first 10 lines of the window. If we shrink a **Bottom_Window** from 20 lines to 10, we should be able to see only the text that appeared on the last 10 lines of the **Bottom_Window**. You may implement any reasonable response when a **Bottom_Window** gets bigger.

   HINT: find out how much shorter the **Bottom_Window** will be, by using the **y()** member function (from class **Point**) on the current size of the **Bottom_Window** and on the new size. Then scroll the contents of the **Bottom_Window** up that many times by calling the **scroll_up** function repeatedly. The class declarations for the classes **Point** and **String** used by this exercise are in the *pre_windows* directory. You may want to look at some of the class declarations, but you should not need to read the definitions of the member functions for any of the classes you use (i.e., do **not** look at *window.c* or *pre_windows/string.c*).

FILE: **use_bot_w.c**

```
#include "Window.h"
#include "Bot_Wind.h"
#include <stdio.h>
#include <stdlib.h>

void number_lines(Window *w)
{
     int i;
```

Exercises 12 Ex

Object-Oriented Programming in C++

Lab Exercises

Summary

- common operations grouped in base class
- a derived class inherits the members of the base class.
- new functions may be added in the derived class.
- base class functions may be overridden in the derived class.
- the derived class may declare a constructor

Summary

Using class Shell_Window

```
#include "Shell_Wind.h"
#include <stdlib.h>

main(int, char *[])
{
    Window w(Point(10, 16), Point(40, 6), "A Window");

   Shell_Window sw(Point(1, 1), Point(10, 10),
            "A Shell_Window");

    w.add("This is a window");
    sleep(2);

    sw.add("Files:\n");
    sw.execute("ls");
    sleep(2);

    sw.change_size(Point(2,2));
    w.change_size(Point(2,2));
    sleep(2);

    return 0;
}
```

Using class Shell_Window

When a derived class object is created, both the base class constructor and the derived class constructor will be called.

Derived Class Constructors

```
#include "Shell_Wind.h"

Shell_Window::Shell_Window(const Point &upper_left,
                  const Point &size,
                  const String &title)
            : Window(upper_left,
                max(size, MIN_SW_SIZE),
                "! " + title + " !")
{
    // this constructor will be called
    // after the base class constructor
}
```

Derived Class Constructors

Both the derived class constructor and the base class constructor will be called when a derived class object is created. The base class constructor will be called before the body of the derived class constructor, but the derived class constructor can control the arguments given to the base class constructor. The syntax for providing this list of arguments is similar to the syntax used to provide arguments to the constructors of a class's data members, but starts with the base class name rather than the member name. The base class name is optional if there is only one base class[2].

The `Shell_Window` constructor compares the size argument against the minimum size, to ensure that a valid size is passed to the `Window` constructor. It also adds !'s to the title.

If the base class has one or more constructors, but no default constructor, then the derived class *must* provide a constructor, which *must* provide base class constructor arguments. If the base class has a default constructor and constructors with parameters, then the derived class may provide constructors, and derived class constructors may provide base class constructor arguments (if they do not, the default constructor will be used). If the base class has only a default constructor, or no constructor at all, then the derived class may provide constructors, but those constructors can not provide base class constructor arguments.

2. older versions of C++ (before 2.0) that provide only single inheritance require that the base class name be omitted.

Using an Overridden Function

```
main(int, char *[])
{
    Window w(Point(10, 16), Point(40, 6), "A Window");

    Shell_Window sw(Point(1, 1), Point(50, 10),
            "A Shell_Window");

    sleep(2);
  sw.change_size(Point(2,2));
  w.change_size(Point(2,2));

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

Using an Overridden Function

If `change_size` is called for a `Shell_Window`, the derived class member function will be used. If `change_size` is called for a `Window`, the base class member function will be used.

Overriding a Base Class Function

```
#include "Shell_Wind.h"


const Point MIN_SW_SIZE(40, 3);

void Shell_Window::change_size(const Point &new_size)
{
    Window::change_size(max(new_size, MIN_SW_SIZE));
}
```

Overriding a Base Class Function

If a member function in a derived class has the same name and argument types as a base class function, the new function overrides the original function in the derived class. If `change_size` is called with a `Window` variable, `Window::change_size` will be used. If it is called with a `Shell_Window` variable, `Shell_Window::change_size` will be used.

The easiest way to write the new change_size function is to call the old one after adjusting the argument. The `max` function for `Point` (defined in Point.h) returns a `Point` that has x and y values equal to the maximum x and y values of the two arguments. Therefore, the position that gets passed along to class `Window's change_size` function will be at least as big as `MIN_SW_SIZE` in both dimensions.

Using Derived Class Member Functions

```
main(int, char *[])
{
    Window w(Point(10, 16), Point(40, 6), "A Window");

    Shell_Window sw(Point(1, 1), Point(50, 10),
            "A Shell_Window");

    w.add("This is a window");
    sleep(2);

    sw.add("Files:\n");
    sw.execute("ls");
    sleep(2);

    return 0;
}
```

Using Derived Class Member Functions

All of the `Window` member functions will be inherited by class `Shell_Window`, so they can be called with `Shell_Window` variables. The derived class member functions can also be called with `Shell_Window` variables.

Defining Derived Class Functions

```
#include "Shell_Wind.h"
#include <stdio.h>

int Shell_Window::execute(const String &command)
{
    FILE *pipe;
    pipe = popen( command.as_char_pointer(), "r" );
    if (pipe == NULL) return -1;

    char buffer[256];
    while ( fgets(buffer, 256, pipe) != 0 ) {
      add(buffer);
      // put text in invoking Shell_Window
    }

    return pclose(pipe);
}
```

Defining Derived Class Functions

The derived class functions, like `Shell_Window::execute`, can not access the private members of the base class. Therefore, `Shell_Window::execute` can only control the data it is displaying on the screen by calling the member functions declared in the public section of class `Window`. It will use `Window::add` function to add text into the invoking `Shell_Window`, but it can NOT access private data declared in class `Window`.

The C language popen function is used to run the shell command. `fgets` is used to read the output of the command into the variable `buffer`. The base class `add` function is then used to add the contents of the buffer into the invoking window.

Declaring a Derived Class

```
#include "Window.h"

extern const Point MIN_SW_SIZE;  // 40 by 3

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

     int execute(const String &command);
      // run command, put output in Window
      // returns exit status or -1 for failure

     void change_size(const Point &new_size);
};
```

Declaring a Derived Class

The declaration of a derived class gives the name of the new class (in this case, `Shell_Window`), a colon followed by either "public" or "private", and the name of the base class (in this case, `Window`). This shows the C++ compiler that `Shell_Window` is derived from `Window`. Class `Shell_Window` will inherit all the functions and data of class `Window`. The keyword **public** in the first line of the declaration means that all the public members of the base class will still be public in the derived class. If **private** had been used instead, the public members of the base would become private members of the derived class. Private derivations will be covered in a later unit.

The derived class may add new members, or override base class member functions. Class `Shell_Window` adds the member function `execute`, which we will use to run shell commands. No additional data are needed in class `Shell_Window`, so none are added.

To ensure that `Shell_Windows` can not become any smaller than 40 by 3, class `Shell_Window` overrides the `change_size` function. If `change_size` is used on a `Shell_Window`, the function from class `Shell_Window` will be called (it enforced the size limit).

Class `Shell_Window` also declares a constructor function. This constructor will ensure that `Shell_Windows` will be at least 40 by 3 when they are created.

Inheritance Example: class Shell_window

a `Shell_Window` is just like a `Window` except:

- also has `execute` member function
- must always be at least 40 x 3 characters in size
- titles will be marked with '!'

Inheritance Example: class Shell_window

When we derive a class from class window, it will, by definition, contain all the members of class window. We therefore concentrate, in our description of the derived class, on how the new class differs from class window.

Using Windows

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

main(int, char *[])
{
    Window w(Point(10,2), Point(40,6), "test window");

    sleep(2);
    w.move_cursor(Point(0,1));
    w.add("This is a test.");

    sleep(2);
    w.clear();
    w.add("It vanishes at the\n");
    w.add("end of main().");

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

Using Windows

This example shows the declaration, initialization, and use of a window variable.

class Window

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

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

    Point size() const;
    void change_size(const Point &new_size);
    // if new size is less than a minimum,
    // shrink only to that minimum 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:
    // data for a Window
};
```

Creating a Base Class

To create a base class for windows, we simply find all the member functions common to all the window classes we wish to create. We can implement our class `Window` the way we would implement any other class: by picking data for the private section, and writing the member functions. In this section, we will be not discuss the implementation of class `Window`.

Some of the member functions have comments describing their actions. The author of a class should provide extensive comments, or a complete manual, to describe how to use the functions. Note, in particular, the comment about `change_size`. Some kinds of windows may have limits on their sizes, so we can not count on being able to make a window arbitrarily small. For example, shell windows, which we will see in a few pages, must always be at least 40 x 3, so if we tried to change a shell window's size to 50 x 1 it would only shrink to 50 x 3.

Note that class `Window` makes use of other classes (i.e., String and Point). Class Point is used to represent a location on the screen or in a window, such as the upper left hand corner of the window or the position of the cursor within the window. Since class `Window` uses classes `Point` and `String`, we include String.h and Point.h before the declaration of class `Window` in Window.h. The #include directives are omitted from the facing page due to lack of space.

Inheritance

a base class

- contains the common members
- any class may be used as a base

derived classes

- *inherit* the members from the base
- may add additional members
- may override inherited member functions

Inheritance

We can avoid these problems by using *inheritance*. To do this, we put all the members that are common to all the window classes into a class known as the *base class,* and *derive* the other classes from the base class.

Base classes are written just like the classes we have seen. In fact, any class may be used as a base. A derived class contains (inherits) all the members (both data and functions) of its base. The derived class may have additional members, or it may re-define member functions inherited from its base class.

Using the Single Class

```
// "require_yes_or_no"
// requires a yes or no answer to a question
// returns 1 for yes, 0 for no
// works only with input windows.

int require_yes_or_no(Input_Window &w, const String &question)
{
    String answer;
    do {
      w.clear();
      w.move_cursor(Point(0, 0));
      w.add(question);
      answer = w.read_input();
    } while ( answer != "yes" && answer != "no" );

    return answer == "yes";
}
```

Using the Single Class

The users of our `Window` class will not be able to write functions that work with only one kind of window. The `require_yes_or_no` function needs to call the `read_input` function, which is only appropriate when working with an input window, but there is no way to restrict the arguments to `require_yes_or_no` to only input windows.

Writing the Single Class

```
void Window::clear()
{
    // clear any kind of window
}

void Window::change_size(const Point &new_size)
{
    switch (k) {
    case text:    // change the size of a text window
      break;
    case input:    // change the size of an input window
      break;
    case shell:    // change the size of a shell window
      break;
    }
}

String Window::read_input()
{
    switch (k) {
    case input:    // read & return in the input
      break;
    default:
      fprintf(stderr,
          "Called read_input for wrong kind of window\n");
      exit(1);
    }
}
```

Writing the Single Class

Our single class `Window`, like our group of three classes, causes problems for the author. If we need to use different code to change the size of different kinds of windows, then the `change_size` function must **switch** on the value of the member `k`. This situation might arise if we place special limitations on the size of a shell_window that do not apply to regular windows; perhaps shell_windows are required to be at least 40 columns wide (anyone who has ever tried to use a terminal with less that 40 columns will probably agree that this is a reasonable restriction).

This use of a **switch** statement causes trouble if we try to add new types of windows. To do so, we must add another case into the existing **switch.** This is only possible if we have control of the window class, and even if we do, there is the chance that we will introduce bugs into what was working code.

A Single Class

```
enum kind_of_window { text, input, shell };

class Window {
public:
    Window(kind_of_window which_kind);

    kind_of_window kind();

    void clear();
    void move_cursor(const Point &);
    void add(char);
    void add(const String &);
    void move(const Point &);
    void change_size(const Point &);

    void execute(const String &command);
    String read_input();

private:
    kind_of_window k;

    // data common to all windows
    union {
      // data for a Text_window
      // data for a Shell_window
      // data for an Input_window
    } special_data;
};
```

A Single Class

Since we want to allow functions that will work with any type of window, we could try to combine the three types into one class, that would represent all kinds of windows.

The single class `Window` would have all the member functions for all kinds of windows. When a `Window` is created, the user must specify which kind of window it is. The `execute` function is only appropriate for `Windows` that were created with the kind `shell`, and the `read_input` function is only appropriate for `Windows` that were created with the kind `input`.

We will see that there are also problems with this technique.

Using Independent Classes

```
#include "Window.h"

// "blank" function
// uses only functions common to
// all kinds of windows, so we should
// be able to use it on any window.

void blank(Window &w)
{
    w.clear();
    w.move_cursor(Point(0, 0));
    w.add("This window intentionally left blank");
}
```

Using Independent Classes

An even more severe problem[1] occurs when we try to write code to use the three window classes. There is no way to create functions that will work with any kind of window. A user of our window classes would have to write three copies of each function: one to work with `Text_windows`, one for `Shell_windows`, and one for `Input_windows`.

1. This problem is worse than the one on the previous page because it causes problems when we try to use the class. If a class has many users, the number of functions that use the class may be many times the number of functions in the class itself. Therefore, a problem for the users can be many times worse than a problem for the author of a class.

Writing Independent Classes

```
void Text_window::clear()
{
    // clear a text window
}

void Shell_window::clear()
{
    // clear a shell window
}

void Input_window::clear()
{
    // clear an input window
}


void Text_window::change_size(const Point &new_size)
{
    // change the size of a text window
}

void Shell_window::change_size(const Point &new_size)
{
    // change the size of a shell window
}

void Input_window::change_size(const Point &new_size)
{
    // change the size of an input window
}
```

Writing Independent Classes

One problem will become apparent when we try to write the member functions of the three classes. We will need to write each of the common member functions three times (once for each class).

There is a way of reducing the size of this problem, but we won't discuss it here because there is a second, even more serious, problem.

Independent Classes

```
class Text_window {
public:
     void clear();
     void move_cursor(const Point &);
     void add(char);
     void add(const String &);
     void move(const Point &);
     void change_size(const Point &);
private:      // data for a Text_window
};
class Shell_window {
public:
     void clear();
     void move_cursor(const Point &);
     void add(char);
     void add(const String &);
     void move(const Point &);
     void change_size(const Point &);
   void execute(const String &command);
private:      // data for a Shell_window
};
class Input_window {
public:
     void clear();
     void move_cursor(const Point &);
     void add(char);
     void add(const String &);
     void move(const Point &);
     void change_size(const Point &);
   String read_input();
private:      // data for an Input_window
};
```

Independent Classes

Since we need three kinds of windows, we could define three classes, one for each kind of window. We will see that there are several problems with this approach.

Related Types

A program may contain groups of related, but not identical types.

For an object-oriented window system:

Text_window	**Shell_window**	**Input_window**
clear	clear	clear
move_cursor	move_cursor	move_cursor
add	add	add
move	move	move
change_size	change_size	change_size
	execute	read_input

Related Types

A program may contain groups of related, but not identical classes. For example, a program that displays information in many different windows on the screen might need to have several types of windows:

- Some windows would just display some text.
- Some would display the output of a shell command.
- Some would prompt the user for input.

All windows can be cleared, have the cursor moved around on them, have text added to them, be moved around on the screen, or have their sizes changed.

NOTE: The details of these window classes will be presented after the introduction to the concept of inheritance. The summaries on the facing page show just enough functions to make this introduction clear.

Groups of classes that have much in common, like this group of window classes, can be created in a simple, flexible, and efficient manner with inheritance. Before we show how to use inheritance, lets see what happens if we try to create these classes using just the C++ features we already know...

Objectives

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

- Use inheritance to create groups of classes that share common features
- Describe why inheritance is needed to create these groups

CONTENTS

Unit 12 - Inheritance

Exercises 12 Ex - Lab Exercises

Answers 12 Ans - Exercise Answers

Unit 12

Object-Oriented Programming in C++

Inheritance