```
        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;
};
```

---

3. If a programmer forgot to override **draw** in a class derived from **Shape**, the compiler would generate an error message (since pure virtual functions must be overridden). If a programmer tried to create and **draw** a **Shape**, the compiler would generate an error for the creation of a **Shape** object, since it is not legal to create an object of an abstract class. With the following two statements added to file *copies.c*, the compiler should generate an error:

```
Shape s;  // Create a 'Shape' variable
s.draw(); // Attempt to draw it
```

UNIT 15

Lab Exercises (Answers)

1. The lines marked "illegal" in the file *use_priv.c* will not compile, since the **insert** function is considered a private member function of classes **Stack** and **Set**.

2. To make the **draw** function in class Shape into a pure virtual function, the initializer "=0" must be added to the declaration of the member function in *Shapes.h* (shown below). The code for the member function **Shape::draw** can now be deleted from *shapes.c*.

   The Unit 13 test program (*copies.c*) will still work with **Shape::draw** declared as a pure virtual function.

   The revised *Shapes.h* file is:

FILE: **Shapes.h**

```
#include "Point.h"

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

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

     virtual void draw() = 0;

private:
     Point _location;
};

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);
```

```
Shape s;
s.draw();
```

Add the above two statements to the *copies.c* test program and try to compile it. NOTE: the statements are already in the file as comments.

You can attempt to compile the test program by entering '**make prob3**' or '**make copies**' or you can attempt to compile it directly using the command:

```
$ CC -c -I../../pre_windows copies.c
```

SUMMARY	
DIRECTORY	unit15/shapes
DECLARATION	pre_windows/*.h, Shapes.h
IMPLEMENTATION	Not Applicable
TEST PROGRAM	copies.c (modify)

UNIT 15

Lab Exercises

1. Change to the *unit15/lecture* directory. Change the file *use_priv.c* (shown on page 15-17 of your Student Guide) so that the lines marked as "illegal" are no longer commented out. Try to compile the file and observe the results.

   You can attempt to compile this file by entering '**make prob1**' or '**make use_priv.o**' or you can attempt to compile it directly using the command:

```
$ CC -c use_priv.c
```

SUMMARY	
DIRECTORY	unit15/lecture
DECLARATION	priv_der.h
IMPLEMENTATION	Not Applicable
TEST PROGRAM	use_priv.c (modify)

2. Change to the *unit15/shapes* directory. The base class **Shape** contains the virtual member function **draw**. Since this function should be declared in the base class, but we can not put any code in the base class function, change it into a pure virtual member function (since class **Shape** will have a pure virtual function, it will be an abstract class.) Are any changes needed to re-compile the test program *copies.c* (the same test program from Unit 13)?

   You can compile the test program by entering '**make prob2**' or '**make copies**' or you can compile it directly using the command:

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

SUMMARY	
DIRECTORY	unit15/shapes
DECLARATION	pre_windows/*.h, Shapes.h (modify)
IMPLEMENTATION	pre_windows/lib_pre_wind.a, shapes.c (modify)
TEST PROGRAM	copies.c

3. Remain in the *unit15/shapes* directory. What would happen if a programmer derived a class from **Shape** and forgot to override draw? What would happen if a programmer tried to create a **Shape** variable and draw it? For example:

Exercises 15 Ex

Object-Oriented Programming in C++

Lab Exercises

Summary

Classes may share:

- both behavior and implementation
  - — derived classes inherit base's functions and data
  - — polymorphic functions may depend on base's functions
  - — inherited implementation simplifies derived classes
- only implementation
  - — private data or private derivation
  - — simplify creation of classes
  - — do not affect users of classes
- only behavior
  - — abstract classes define only functions (behavior)
  - — polymorphic functions may depend on base's functions
  - — no implementation is inherited

Summary

Using an Abstract Class

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

extern Screen Display;
extern Printer line_printer;

main(int, char *[])
{
    say_hello(Display);
    say_hello(line_printer);
    line_printer.print();

    return 0;
}
```

Using an Abstract Class

Even though abstract classes do not simplify the creation of derived classes, they are important because they let users of the classes create polymorphic functions. Since `Screen` and `Printer` are publicly derived from `Display_medium`, they must contain all the member functions declared in `Display_medium`. We can therefore use objects of these classes (or any other classes publicly derived from `Display_medium`) as arguments to the function `say_hello`.

Deriving From an Abstract Class

```
#include "Display_med.h"

class Screen : public Display_medium {
public:
    Screen();
    ~Screen();

    Point size() const;

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

    Display_char character() const ;
    String line() const ;

    void add(Display_char ch);
    void add(const Point &, Display_char ch);
    void add(const String &s);
    void add(const Point &, const String &s);

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

private:
    // implementation
};
```

Deriving From an Abstract Class

We can now derive classes `Screen` and `Printer` from `Display_medium`. A class derived from an abstract base must override all the pure virtual functions it inherits.

We will have to declare the private data for classes `Screen` and `Printer`, and write their member functions, before we can create programs that use them. The use of an abstract base class has not made this process of implementing the classes significantly easier.

```
#include "Display_med.h"

class Printer : public Display_medium {
public:
    Point size() const;

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

    Display_char character() const;
    String line() const ;

    void add(Display_char ch);
    void add(const String &s);

    void clear();
    virtual void print();

private:
    // implementation
};
```

An Abstract Base Class

```
#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:

};
```

An Abstract Base Class

The abstract base class for our classes `Screen` and `Printer` is shown on the facing page. It contains all the member functions that will be needed by both classes.

Since we can not write most of the member functions for class `Display_medium`, we have declared them as pure virtual member functions. Pure virtual member functions are identified by the initializer "= 0". C++ will not let us create an object of an abstract class, since its pure virtual functions will be declared but not defined. If we could create an object, we might end up calling a function that does not exist.

An abstract base class may have some functions that are not pure virtual functions. For example, we can write the functions `Display_medium::clear` and `Display_medium::add_line`, even though class `Display_medium` has no data.

```
void Display_medium::clear()
{
    int x, y;
    for (x = 0; x < size().x(); x++) {
       for (y = 0; y < size().y(); y++) {
           move_cursor(Point(x, y));
           add(' ');
       }
    }
}
```

Sharing Behavior

Classes `Screen` and `Printer`

- share many operations
  - — add text
  - — clear page
- have different implementations

To share only behavior, we can

- list common operations in an *abstract base class*
  - — little or no data
  - — *pure virtual* member functions

Sharing Behavior

Classes may also share a common interface even thought they are implemented in very different ways. For example, consider a class representing terminal screens, and a class representing a sophisticated interface to a printer. Even though printers do not have "cursors" like terminals, our printer driver simulates a cursor that moves around the page and lets us add text anywhere. Once we have added all the text we need on a page, we use the `print` member function to send the page to the printer.

```
class Screen {
public:
    void clear();

    void add(Display_char ch);
    void add(const String &s);

    Point size() const;
    Point cursor() const;
    int move_cursor(const Point &p);

private:  // implementation of Screen
};

class Printer {
public:
    void clear();

    void add(Display_char ch);
    void add(const String &s);

    Point size() const;
    Point cursor() const;
    int move_cursor(const Point &p);

    void print();

private: // implementation of Printer
};
```

These classes have very similar sets of member functions, but they may have completely different implementations. If we create a base class for these two classes, any data in the base class will be inherited by both classes. If there are no common data members, then we should not list any data in the base. If the base class has no data, we will we will not be able to write many of the member functions. A virtual member function that is declared but not defined is called a *pure virtual* member function. A base class containing one or more pure virtual functions (and used for classes that share only behavior) is called an *abstract class*.

Using classes Stack and Set

```
main(int, char *[])
{
    Stack st;
    Set s;
    List l;

    l.insert(0, 256);
    l.insert(1, 128);
    l.insert(0, 128);

    st.push(1);
    st.push(2);
    st.push(3);
  // st.insert(1, 4);  // illegal

    s.add(2);
    s.add(3);
    s.add(5);
  // s.insert(1, 5);  // illegal

    return 0;
}
```

Using classes Stack and Set

Private derivation and the inclusion of a private member function are quite different from public derivation. Since neither will allow the use of `List` operations on `Stack` and `Set` objects, the users of these classes will not be able to write code that depends on our implementation. `Stacks` and `Sets` do not share the behavior of class `List`. C++ will not let us pass a `Stack` or `Set` as an argument to a function expecting a `List`.

Member Functions using Private Data

```
void Stack::push(const element &e)
{
    l.insert(l.size(), e);
// or, equivalently,
//   this->l.insert(this->l.size(), e);
}

void Set::add(const element &e)
{
    if (l.find(e) == -1) {
      l.insert(0, e);
    }
}
```

Member Functions using Private Data

The use of a private data member lets the author of the classes take advantage of the common implementation. The `Stack` and `Set` member functions can be written quite easily in terms of operations on the `List` data member. Since the data member is private, the users of these classes can not use the `List` operations on a `Stack` or a `Set`.

Private Data

```
class List {
public:
    int find(const element &);
    element lookup(int index);
    void set(int index, const element &);
    void insert(int index, const element &);
    void remove(int index);
    int size();

private:
    //...
};

class Stack {
public:
    void push(const element &);
    element pop();
    int size();
private:
    List l;
};

class Set {
public:
    void add(const element &);
    int contains(const element &);
    int size();
private:
    List l;
};
```

Private Data

If `Stack` and `Set` each have a private data member of type `List`, the member functions of `Stack` and `Set` can access that `List`; other functions can not.

Member Functions using Private Derivation

```
void Stack::push(const element &e)
{
    insert(size(), e);
// or, equivalently,
//   this->insert(this->size(), e);
}

void Set::add(const element &e)
{
    if (find(e) == -1) {
      insert(0, e);
    }
}
```

Member Functions using Private Derivation

The use of private derivation provides the advantages of inheritance to the author of the classes, without affecting the users' code. The `Stack` and `Set` member functions can be written quite easily in terms of the `List` operations, but the users of these classes can not use the `List` operations to create illegal `Stacks` or `Sets`.

Private Derivation

```
class List {
public:
    int find(const element &); // returns -1 if not found
    element lookup(int index);
    void set(int index, const element &);
    void insert(int index, const element &);
    void remove(int index);
    int size();

private:
    //...
};

class Stack : private List {
public:
    void push(const element &);
    element pop();
    int size();

private:
};

class Set : private List {
public:
    void add(const element &);
    int contains(const element &);
    int size();

private:
};
```

Private Derivation

If we derive `Stack` and `Set` from class `List` with a private derivation, then the public members of class `List` will be inherited into the private sections of those classes.

The member functions of `Stack` and `Set` can use the `List` member functions, but other functions can not.

Sharing Implementation

Classes `Stack` and `Set`

- can be implemented with a linked list
- should not allow the use of list operations

To share only implementation, we can:

- use a private derivation, or
- use a private data member

Sharing Implementation

A stack and a set can both be implemented easily in terms of a linked list. But they have very different public interfaces: A stack keeps track of the order in which elements were added, and only allows access to the most recently added element. A set does not order its elements, and a single element may occur only once in a set.

Deriving classes `Stack` and `Set` publicly from `List` would allow the use of `List` operations on `Stacks` and `Sets`. A user could `insert` an element in the middle of a `Stack`, or `insert` one element twice in one `Set`. We can allow classes `Stack` and `Set` to share the linked-list implementation in two ways: we can use private derivations, or we can declare `List` data members in the private sections of classes `Stack` and `Set`.

The code below shows examples of the uses of `Lists`, `Stacks`, and `Sets`:

```
main(int, char *[])
{
    Stack st;
    Set s;
    List l;

    l.insert(0, 256);
    l.insert(1, 128);
    l.insert(0, 128);

    st.push(1);
    st.push(2);
    st.push(3);
  // st.insert(1, 4);  // illegal

    s.add(2);
    s.add(3);
    s.add(5);
  // s.insert(1, 5);  // illegal

    return 0;
}
```

Inheritance

Inheritance is used to group common features:

- common behavior
  - — derived classes share base's public operations
  - — user can create polymorphic code
- common implementation
  - — derived classes share base's data
  - — derived class operations can call base's operations

Classes may share:

- behavior and implementation
- only behavior
- only implementation

Inheritance

A derived class contains all the members from its base class; both the public members and the private members. With a publicly derived class, the base class's public members are still public, and the private data is still hidden from the outside world (it is even hidden from the derived class's member functions).

Since a derived class object can "act like" a base class object (code that was written for a base class object will work for a derived class), we say that the derived classes share the *behavior* defined by the base. This common behavior lets users of the classes create polymorphic functions. Functions with base class arguments can only use the operations that are defined in the base (and therefore shared), so these functions can work with arguments of any derived class.

Since the derived class also contains the private section of the base class, it shares the base class's implementation. This simplifies the creation of the derived classes, as the common data and functions are defined only once, in the base.

There may be situations where we have several classes that share only common behavior, but not a common implementation, or situations in which we have several classes with a common implementation but very different interfaces. In this unit, we will see how to handle such situations.

Objectives

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

- Distinguish inheritance of behavior from inheritance of implementation
- Create groups of classes that share only behavior
- Create groups of classes that share only implementation

CONTENTS

Unit 15

Object-Oriented Programming in C++

Kinds of Hierarchies