```
     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;
};
```

```
        collapse(bw);

        sleep(5);
        return 0;
}
```

3. When the keyword **virtual** is missing from the declaration of the **change_size** member function in the **Window** base class, C++ will use static binding to determine which **change_size** function will be called in **collapse**. Since static binding relies on the type information in the variable declaration, and the parameter of **collapse** is declared with type **Window**, C++ will use the **Window::change_size** function. The sizes of **Shell_Windows** and **Bottom_Windows** will be changed incorrectly when they are passed to **collapse**.

4. If the **draw** function was not declared in class **Shape**, C++ would not allow the use of **s.draw** inside the **copies** function (in file *copies.c*), as s (the parameter of **copies**) is declared as **Shape &s**. Therefore, to compile *copies.c*, we must declare **draw** in the base class.

   If the **draw** function were not declared as a virtual function in class **Shape**, C++ would select the **draw** functions used for shapes according to the type shown in the declaration. Since s is declared as **Shape &s**, C++ would always select **Shape::draw** for s in the **copies** function, even if s actually refers to a **Rectangle**, **Vertical_line** or **Horizontal_line**. To make C++ select the **draw** function according to the type of object that is actually passed to **copies**, we must declare **draw** as a virtual function in the base class. If we do not do this, none of the shapes will be drawn by **copies**. The code for the functions, in *shapes.c*, does not need to be modified. After the keyword **virtual** is added to the declaration of **draw**, *Shapes.h* will look like this:

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();

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);
```

UNIT 13

Lab Exercises (Answers)

1. The **Shell_Window** only shrinks to 5 lines high (as requested) by 40 characters wide (the minimum), since its size is changed by **Shell_Window::change_size**.

2. When a **Bottom_Window** is shrunk by the **collapse** function, the text from the bottom part is shown, since the size is changed with the **Bottom_Window::change_size** function. The files *collapse.c* and *u_collapse.c* have been modified as shown:

FILE: **collapse.c**

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

// collapse function
// clear a window and make it small

void collapse(Window &w)
{
// The following statement is removed to see effect of change_size
// w.clear();
     w.change_size(Point(5, 5));
     // which change_size function will be called
     // (a)        Window::change_size(const Point &)
     // (b)  Shell_Window::change_size(const Point &)
}
```

FILE: **u_collapse.c**

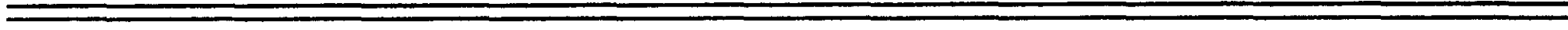
```
#include "Shell_Wind.h"
#include "Bot_Wind.h"
#include <stdlib.h>

void collapse(Window &);

main(int, char *[])
{
     Window w(Point(10, 16), Point(40, 6), "W");
     Shell_Window sw(Point(1, 1), Point(10, 10), "SW");
     Bottom_Window bw(Point(50, 1), Point(10, 10), "BW");

     w.add("Text in the Window");
     sw.add("User       Terminal     Login time\n");
     sw.execute("who | sort");
     bw.add("Placing text in the Window\n");
     bw.add("More text in the Window\n");
     bw.add("Even More text in the Window\n");

     sleep(2);
     collapse(w);
     collapse(sw);
```



can compile and execute it directly using the commands:

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

SUMMARY	
DIRECTORY	unit13/shapes
DECLARATION	pre_windows/*.h, Shapes.h (modify)
IMPLEMENTATION	pre_windows/lib_pre_wind.a, shapes.c
TEST PROGRAM	copies.c

FILE: **copies.c**

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

// produce many copies of a shape on the screen

void copies(Shape &s)
{
        Point p = s.location();
        int i;

        for (i=5 ; i > 0 ; i--) {
                s.move(p + Point(i * 5, i * 2));
                s.draw();
        }
}

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();

     sleep(2);
         copies(r);
         copies(h);
         copies(v);

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

SUMMARY	
DIRECTORY	unit13/window
DECLARATION	pre_windows/*.h, Shell_Wind.h, Bot_Wind.h
IMPLEMENTATION	pre_windows/libpre_wind.a, window.c, shell_wind.c, bot_wind.c
TEST PROGRAM	collapse.c (modify), u_collapse.c (modify)

3. Remain in the *unit13/window* directory. Remove the keyword **virtual** from the declaration of the **change_size** member function in the *Window.h* header file. Recompile the test program from the last exercise, and observe the results when it is run. To what size does the **Shell_Window** shrink? Which data are left in the **Bottom_Window**?

   You can compile and execute the test program by entering '**make prob3**' or you can compile and execute it directly using the commands:

```
$ CC -I../../pre_windows u_collapse.c collapse.c \
        window.c shell_wind.c input_wind.c \
        bot_wind.c -L../../pre_windows -lpre_wind \
        -lcurses -o collapse
$ collapse
```

   After you have compiled and executed the test program, put the keyword **virtual** back in the declaration of the **change_size** member function in the *Window.h* header file.

SUMMARY	
DIRECTORY	unit13/window
DECLARATION	pre_windows/*.h, Window.h (modify),
IMPLEMENTATION	pre_windows/libpre_wind.a, window.c, shell_wind.c, bot_wind.c
TEST PROGRAM	collapse.c, u_collapse.c

4. This exercise uses the classes created for exercise 4 of the Unit 12 Lab. The *unit13/shapes* directory contains the final results from the Unit 12 Lab and is the starting point for this exercise.

   Change to the *unit13/shapes* directory. Add the keyword **virtual** to the appropriate member function(s) of your base class in the file *Shapes.h*. Compile and execute the program *copies.c* (shown below for reference.) What would happen if none of the base class member functions were virtual?

   You can compile and execute the test program by entering '**make prob4**' or you

UNIT 13

Lab Exercises

1. Change to the *unit13/window* directory. The files *collapse.c* and *u_collapse.c* contain the **collapse** and **main** functions shown on page 13-15 of the Student Guide. Compile these functions and run the resulting program (*collapse*). How large is the Shell_Window after it collapses?

   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_windows u_collapse.c collapse.c \
          window.c shell_wind.c -L../../pre_windows \
          -lpre_wind -lcurses -o collapse
   $ collapse
   ```

SUMMARY	
DIRECTORY	unit13/window
DECLARATION	pre_windows/*.h, Shell_Wind.h
IMPLEMENTATION	pre_windows/libpre_wind.a, window.c, shell_wind.c
TEST PROGRAM	collapse.c, u_collapse.c

2. Remain in the *unit13/window* directory. The files *Bot_Wind.h* and *bot_wind.c* contain the solutions from the Unit 12 lab exercises. Change the test program in the file *u_collapse.c* so that it declares a **Bottom_Window** variable, in addition to the **Window** and **Shell_Window** variables. It should also call the **collapse** function with the new **Bottom_Window** variable. Which part of the **Bottom_Window** is visible after it has been collapsed? Note: you will need to add some text to the **Bottom_window**, and remove the call to the **clear** function from the **collapse** function (in the file *collapse.c*), so that you can tell which data are shown in the **Bottom_Window** after the call to **collapse**.

   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 u_collapse.c collapse.c \
          window.c shell_wind.c input_wind.c \
          bot_wind.c -L../../pre_windows -lpre_wind \
          -lcurses -o collapse
   $ collapse
   ```

Version 3.0.2
Copyright © 1990 AT&T
All Rights Reserved

Lab Exercises

Version 3.0.2
Copyright © 1990 AT&T
All Rights Reserved

Summary

With inheritance and dynamic binding, we can

- add new derived classes without modifying code
- write polymorphic functions
- write functions for only one of the classes

Dynamic binding

- more execution time & memory than static binding
- works even if declared type != object type
- in C++, must be selected by programmer

Version 3.0.2
Copyright © 1990 AT&T
All Rights Reserved

Summary

Now that we know how and when to use dynamic binding, we can use inheritance to write a group of classes that represent different kinds of windows, members without any of the problems that arose before we used inheritance:

- We can add additional kinds of windows without having to modify the existing classes
- We can write functions that use only the member functions common to all windows (e.g., blank and collapse), and call them with any kind of window.
- If we write functions that require the member functions of one particular kind of window (e.g., require_yes_or_no), we can call it only with the appropriate kind of window.

Version 3.0.2
Copyright © 1990 AT&T
All Rights Reserved

Copying a Derived Class Object

```
void collapse2(Window w)
{
    w.clear();
    w.change_size(Point(5, 5));
}

main(int, char *[])
{
    Window w(Point(10, 16), Point(40, 6), "W");
    Shell_Window sw(Point(1, 1), Point(10, 10), "SW");

    sleep(2);

    w = sw;

    sleep(2);
    collapse2(sw);

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

Version 3.0.2
Copyright © 1990 AT&T
All Rights Reserved

Copying a Derived Class Object

Since a derived class object can be used wherever a base class object is required, we can assign a derived class value to a base class object, or initialize a base class object with a derived class value. When we do this, the base class members of the derived class are assigned or copied into the base class object. The type of the base class object does not change, so if a virtual function is called, the base class function will be used.

If assignment of base class objects is illegal (if the base class's assignment operator is in its private section), then neither base nor derived class objects can be assigned to base class variables. If copying of base class objects is illegal (if the base class's copy constructor is in its private section), then neither base nor derived class objects can be used to initialize base class variables.

Version 3.0.2
Copyright © 1990 AT&T
All Rights Reserved

Polymorphic Arrays

```
void collapse(Window &);

main(int, char *[])
{
    Window w(Point(10, 16), Point(40, 6), "W");
    Shell_Window sw(Point(1, 1), Point(10, 10), "SW");
    Input_Window iw(Point(45, 1), Point(30, 4), "IW");

    Window *all_windows[4];
    all_windows[0] = &w;
    all_windows[1] = &sw;
    all_windows[2] = &iw;
    all_windows[3] = 0;

    sleep(2);
    int i;
    for (i = 0; all_windows[i]; i++)
      collapse(*all_windows[i]);

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

Version 3.0.2
Copyright © 1990 AT&T
All Rights Reserved

Polymorphic Arrays

Since a derived class object can be used anywhere a base class object is expected, we can point a base class pointer to a derived class object. We can therefore create an array of base class pointers, and point those pointers to several different types of objects.

We can create an array with one type of pointer, and those pointers can point to different types, but we can not create an array that contains different types of elements, since there is no way to declare such an array.

Version 3.0.2
Copyright © 1990 AT&T
All Rights Reserved

Choosing virtual Functions

Which functions should be **virtual** ?

- anticipate which member functions will be overridden
- make all member functions **virtual**
- add **virtual** to base class while developing derived classes

Approach should depend on use of class

Version 3.0.2
Copyright © 1990 AT&T
All Rights Reserved

Choosing virtual Functions

Base classes are often created because a programmer sees that several classes share common member functions (as is the case with the window example). When this happens, the common functions are put in the base class, and the classes that originally inspired the base class become the derived classes. In this case, the descriptions of the derived classes are known when the base class is written, so the author of the base class can easily tell which functions will be overridden, and make them **virtual.**

In some cases, however, you may need to add a new derived class to an existing base, or even derive a class from some class that was not designed as a base class. If you need to override a base class member function that was not declared **virtual,** you will have to change the base class. Earlier in this unit, we saw that we could add new derived classes without having to modify or re-compile existing code. That is only the case if we do not have to add the keyword **virtual** to any base class functions.

To address this potential maintenance problem, we may want to consider three styles when we create any class in C++:

(1) Try to anticipate which member functions might be overridden in derived classes, and make them virtual.

This works well if the author of a class knows about the classes that will be derived from it.

(2) Make all member functions virtual in case a derived class needs to override one of them.

This style simulates what happens in languages where dynamic binding is used for all function calls. We accept the storage and time overhead, but have the benefit that we never need to change existing classes when we derive from them. Some programmers use this style when developing classes, and then remove the keyword **virtual** (where possible) as a form of optimization, before a completed program is shipped.

(3) Don't make any functions virtual until you write the derived classes, and then go back and change (and re-compile) the base.

This style is appropriate when performance is critical, or if there will probably not be any need to derive a class from the one being written. Note that even if you use this style, and have to add **virtual** to the base class at a later time, the changes you must make in the base class are far less significant than the changes we had to make to our single-class window system that we described before we learned about inheritance.

If your product is a class library rather than a program, you may not be able to tell which member functions will be overridden by the users of your library. In that case, you may want to provide two versions of the class: one with the virtual functions only where you expect functions to be overridden, and the other with all member functions virtual (just in case you were wrong).

Version 3.0.2
Copyright © 1990 AT&T
All Rights Reserved

Virtual Functions

```
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;
  virtual 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
};
```

Version 3.0.2
Copyright © 1990 AT&T
All Rights Reserved

Virtual Functions

If we simply add the keyword **virtual** in front of the `change_size` function in class `Window`, the C++ compiler will use dynamic binding to call `change_size`. Therefore, if we call `collapse` with a `Shell_Window` argument, the `Shell_Window::change_size` function will be used to change the `Shell_Window's` size.

Version 3.0.2
Copyright © 1990 AT&T
All Rights Reserved

Type Information and Function Selection

Static (compile-time) Binding

- uses type information from variable declaration
- assumes declared type == object type
- default in C++

Dynamic (run-time) Binding

- uses type information in the object itself
- used only for member function's invoking object
- programmer selects with the keyword **virtual**

Version 3.0.2
Copyright © 1990 AT&T
All Rights Reserved

Type Information and Function Selection

Dynamic binding allows a compiler to delay the selection of the function until run time, when there is more information about the object type. If we use static binding, calls to `change_size` will always use information from the declaration of the invoking object. The `collapse` function would always call `Window::change_size`, because the variable `w` is declared as a `Window`. If we use dynamic binding, calls to `change_size` will use information found in the invoking object itself, at run time. The `collapse` function would call `Window::change_size` if `w` is a `Window`, and `Shell_Window::change_size` if `w` is a `Shell_Window`. If `w` is an `Input_Window`, `collapse` will call `Input_Window::change_size` if it exists, or `Window::change_size` if class `Input_Window` does not override `change_size`.

To perform dynamic binding on the `change_size` function, the compiler must add information to every `Window`, `Shell_Window`, or `Input_Window` object that is created. It must then check this information when the `change_size` function is called. The overhead of a member function call using dynamic binding is about equal to that of a call to a member function for our single "window" class that we wrote before we learned about inheritance (the one that had a data member that can be used to identify the kind of window, and had member functions that switched on that member to select the code to be used to, for example change the size). Therefore, if we use dynamic binding only when it is necessary, it will not change the size or execution time of our program significantly.

Unnecessary use of dynamic binding, however, can result in a waste of storage and execution time. For this reason, C++ lets the programmer choose either static or dynamic binding (many object-oriented languages provide only dynamic binding). A member function will be selected with dynamic binding in C++ if it is declared as a *virtual* function in the base class. Note that, in C++, dynamic binding only applies to the invoking object, not to function arguments.

Version 3.0.2
Copyright © 1990 AT&T
All Rights Reserved

Another Polymorphic Function

```
// collapse function
// clear a window and make it small

void collapse(Window &w)
{
    w.clear();
    w.change_size(Point(5, 5));
    // which change_size function will be called
    // (a)       Window::change_size(const Point &)
    // (b)  Shell_Window::change_size(const Point &)
}

main(int, char *[])
{
    Window w(Point(10, 16), Point(40, 6), "W");
    Shell_Window sw(Point(1, 1), Point(10, 10), "SW");

    w.add("Text in the Window");
    sw.add("User      Terminal    Login time\n");
    sw.execute("who | sort");

    sleep(2);
    collapse(w);
    collapse(sw);

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

Version 3.0.2
Copyright © 1990 AT&T
All Rights Reserved

Another Polymorphic Function

The new type checking rule has an impact on the C++ compiler's ability to select a function. The C++ compiler selects a function based on the function's name, the number of arguments, and the types of the invoking object and the arguments.

The new type checking rule allows us to write functions (such as `blank` and `collapse`) in which a pointer or reference declared with the base type (`Window`) may be used with an object of a derived type (`Shell_Window` or `Input_Window`). What happens when such a function uses an overridden member function (e.g., `change_size`)? Will it call `Window::change_size` or `Shell_Window::change_size`?

Version 3.0.2
Copyright © 1990 AT&T
All Rights Reserved

Type Checking in Object-Oriented Languages

When an object of type T is expected, we can use

- an object of type T, or
- an object of a class publicly derived from T

This rule

- allows polymorphic code
- introduces new problems...

Version 3.0.2
Copyright © 1990 AT&T
All Rights Reserved

Type Checking in Object-Oriented Languages

In C, when one type of object is required, only an object of that type can be used. The object may be the result of an implicit or explicit type cast, but it must be the right type (e.g., if we compile `sqrt(2)` (or `sqrt( (double) 2)`, on older C compilers), `sqrt` will still be called with a *double* (the result of the cast). In C++, however, when one type of object is expected, we can use an object of that type, or any class publicly derived from that type. If we compile `blank(sw)`, the `blank` function will be called with a *Shell_Window* argument, even though its parameter is of type `Window`. No conversion occurs.

Note that only objects of *publicly* derived classes can be used. We will study private derivations in a later unit.

This type checking rule works (i.e., it still stops us from compiling bad code) because the derived classes must, by definition, inherit all the functions and data from their base class. So, a function that is written to work with the base class can only rely on base class functions and data, which must be present in the public section of the derived classes.

This type checking rule lets us create polymorphic functions, but it also introduces a new problem, which we will see on the next page.

Version 3.0.2
Copyright © 1990 AT&T
All Rights Reserved

Functions for the Base Class

To create a polymorphic function:

- with independent window classes
  - — impossible (can only create overloaded functions)
- with inheritance:
  - — use a base class parameter

```
void blank(Window &);

main(int, char *[])
{
    Window w(Point(10, 16), Point(40, 6), "W");
    Shell_Window sw(Point(1, 1), Point(10, 10), "SW");
    Input_Window iw(Point(45, 1), Point(30, 4), "IW");
    sleep(2);
    blank(w);
    blank(sw);
    blank(iw);

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

Version 3.0.2
Copyright © 1990 AT&T
All Rights Reserved

Functions for the Base Class

It may come as a surprise that if a function has a Window parameter, we can call it with a Window argument, an Input_Window argument, or a Shell_Window argument.

Such function calls are legal because the type checking rules of C++ are more complicated than those of C.

```
#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");
}
```

Version 3.0.2
Copyright © 1990 AT&T
All Rights Reserved

Functions for Derived Classes

To write a function for one kind of window:

- with a single class:
  - — check `kind()` of window in function
  - — not type safe
- with inheritance:
  - — use derived class parameter

```
int require_yes_or_no(Input_Window &, const String &);

main(int, char *[])
{
    Window w(Point(10, 16), Point(40, 6), "W");
    Shell_Window sw(Point(1, 1), Point(10, 10), "SW");
    Input_Window iw(Point(45, 1), Point(30, 4), "IW");

    require_yes_or_no(iw, "End program?");
//    require_yes_or_no(w, "End program?");  illegal
//    require_yes_or_no(sw, "End program?");  illegal

    return 0;
}
```

Version 3.0.2
Copyright © 1990 AT&T
All Rights Reserved

Functions for Derived Classes

It should come as no surprise that if a function has an `Input_Window` parameter, we can only call it with an `Input_Window` argument. We can easily create functions that can only be called with one particular kind of window (which we could not do when we had a single window 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";
}
```

Version 3.0.2
Copyright © 1990 AT&T
All Rights Reserved

Adding New Derived Classes

To add a new kind of window

- with a single class:
  - — add to enumeration
  - — add to union
  - — add cases to switch statements in member functions
  - — need access to the source to change it
  - — need to understand existing code
  - — need to re-compile the member functions & user's code
  - — need to re-test the member functions we changed
- with inheritance:
  - — simply add a new derived class

Version 3.0.2
Copyright © 1990 AT&T
All Rights Reserved

Adding New Derived Classes

When we tried to use one class to represent the three kinds of windows, some of the member functions contained **switch** statements:

```
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);
    }
}
```

The code above is hard to extend, because the addition of new kinds of windows requires the modification of existing code.

When we saw how to create a derived class, we saw that we could add new functions (or data) for the derived class, override functions in the derived class, and create a specialized constructor for the derived class. None of these activities required any change to the base class, or any existing derived window classes.

Version 3.0.2
Copyright © 1990 AT&T
All Rights Reserved

Inheritance

Creating related types:

- independent classes
  - — couldn't write `blank`
- single class
  - — hard to add new types
  - — couldn't write `require_yes_or_no`
- base & derived classes
  - — none of the above problems, if used with *dynamic binding*

Version 3.0.2
Copyright © 1990 AT&T
All Rights Reserved

Inheritance

Before we learned about derived classes, we saw the problems associated with other ways of creating a group of similar classes (e.g., our window classes):

With several independent classes, we had no way to write a function that worked with all kinds of windows (such as the function `blank`). This would force us to write three `blank` functions, one for each kind of window. This problem would become worse and worse as our program grows. If our final product had five kinds of windows, and the users wanted to write 30 different functions to manipulate those windows, they would end up writing 120 unnecessary functions.

With a single class for all kinds of windows, we had to modify existing code to add a new kind of window, and we had no way to write a function that could be called with only one kind of window (such as the function `require_yes_or_no`).

In the next few pages, we will see that the use of inheritance avoids the problems we found with both of the original solutions, but only if we use *dynamic binding* (dynamic binding will be defined after we see what happens when we try to use inheritance without it).

Version 3.0.2
Copyright © 1990 AT&T
All Rights Reserved

Objectives

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

- Distinguish dynamic binding from static binding
- Describe why dynamic binding is needed
- Use dynamic binding in C++

Version 3.0.2
Copyright © 1990 AT&T
All Rights Reserved

CONTENTS

Unit 13 - Dynamic Binding

Inheritance 13-5
Adding New Derived Classes 13-7
Functions for Derived Classes 13-9
Functions for the Base Class 13-11
Type Checking in Object-Oriented Languages 13-13
Type Information and Function Selection 13-17
Polymorphic Arrays 13-25
Summary 13-29

Exercises 13 Ex - Lab Exercises

Answers 13 Ans - Exercise Answers

Unit 13

Object-Oriented Programming in C++

Dynamic Binding