

Chapter 4

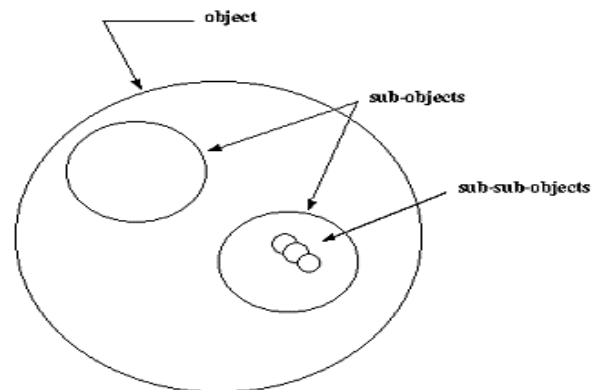
Implementing a New Class

Implementing a new class involves...

- design ideas
(what's a good class)
- design representation
(e.g., diagrams)
- aggregation
(compose & hide sub-objects)
- language features
(C++ syntax/rules)
- tools
(to code/debug)

Section 4.2:

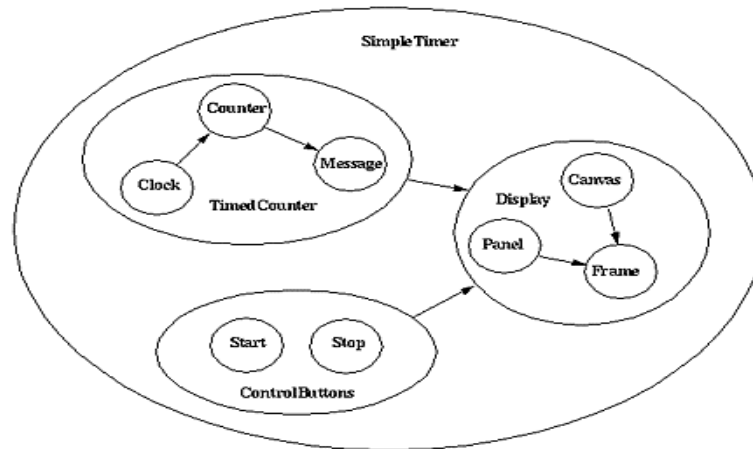
Structure of an Aggregation



Advantages of Aggregation

- **simplicity**
(one object, not many)
- **safety**
(via encapsulation of sub-objects)
- **specialization**
(of interface)
- **structure**
(isomorphic to real-world)
- **substitution**
(replacing sub-objects)

Composition via Aggregation



Types of Aggregation

- **Static aggregation**
number & organization of sub-objects are predefined and unvarying
- **Dynamic aggregation**
the opposite: sub-object allocated dynamically
- **Example:** Is max stack size fixed or varying?

Examples

■ Static

- Car - four tires
- Book - fixed number of pages
- Page in a book - one "next" page
- Stack - fixed size
 - specified on construction, but fixed thereafter

■ Dynamic

- Tire store - varying number of tires
- Web site - number of "pages" can change
- Web page - many possible "next" pages

Try it!

- Stack -varying size
 - Stack::Push(int) can grow stack size

Review - Class Stack

```
class Stack {  
    public:  
        Stack(int sz);  
        void Push(int value){ASSERT(numItems<10); stack[numItems++]=value;}  
    private:  
        int size;           // Max capacity of stack.  
        int numItems;      // Index of the lowest unused position.  
        int* stack;        // Pointer to array that holds contents.  
};  
  
Stack::Stack(int sz) {  
    size = sz;  
    numItems = 0;  
    stack = new int[size]; // Allocate an array of integers.  
}
```

Redefine Stack::Push(int) so it automatically increases size of array stack

Stack Without Size Limit: Stack::Push(int) can grow stack size

```
void Stack::Push(int value) {  
  
    if (numItems >= size) // if numItems is greater than size  
    {  
        // allocate a new int array, twice current size  
        int* tempS = new int[2*size]; size = size*2;  
  
        // copy current contents into new array  
        for (unsigned i=0; i<numItems; i++) { tempS[i] = stack[i]; }  
  
        // swap old & new stacks; delete old stack  
        stack = tempS; delete stack;  
    }  
  
    // Finally! We can push new value onto stack  
    stack[numItems++] = value;  
}
```

<http://ei.cs.vt.edu/~cs2704/fall98/Begole/StackTest1.zip>

Properties of a Good Class

- abstraction
- correctness
- safety
- efficiency

Section 4.3:

Implementing a class means defining:

■ **data:**

encapsulated (hidden, private) variables
recording current "state" of object

■ **code:**

member functions (operations, methods)
performing actions on data

Example Interface

```
class Location {
    private:
        int currentX, currentY;
    public:
        Location(int x, int y);
        Location();           // Default location
        int Xcoord();        // Return x-axis coordinate
        int Ycoord();        // Return y-axis coordinate
        ~Location();
};
```

Example Implementation

```
Location::Location(int x, int y)
    { currentX = x; currentY = y; }

Location::Location () { currentX = currentY = -1; }

int Location::Xcoord() { return currentX; }

int Location::Ycoord() { return currentY; }

Location::~~Location() {}
```

Try this now...

- Add Location::Equal(Location&)
 - Not using accessor methods
 - Using accessor methods

```
class Location {
private:
    int currentX, currentY;
public:
    Location(int x, int y);
    Location();           // Default location
    int Xcoord();        // Return x-axis coordinate
    int Ycoord();        // Return y-axis coordinate
    ~Location();
};
```

Solution...

Not using accessor:

```
int Location::Equals(Location& other)
{
    return (currentX == other.currentX
            && currentY == other.currentY);
}
```

Using accessor:

```
int Location::Equals(Location& other)
{
    return (currentX == other.Xcoord()
            && currentY == other.Ycoord());
}
```

Try this now...

- Implement a **safe** Stack::Pop() function:
If there's nothing on the stack, Pop "somehow" signals the caller that there's an error.

Stack::Pop() Interface

```
enum ErrorType { NO_ERROR, STACK_UNDERFLOW, STACK_OVERFLOW};

class Stack {
public:
    Stack(unsigned sz=5);
    void Push(int value);
    int Pop();
    ErrorType GetError();
    ~Stack();

private:
    unsigned size;
    unsigned numItems;
    int* stack;
    ErrorType errorType;
};
```

Stack.h

Implementing Stack::Pop()

```
int Stack::Pop() {
    if (numItems > 0) {
        numItems --;
        errorType = NO_ERROR;
        return stack[numItems ];
    } else {
        errorType = STACK_UNDERFLOW;
        return -1; // -1 could be a legal value
    }
}

int Stack::GetError() { return errorType; }
```

Stack.cpp

Using Stack::Pop()

```
#include "Stack.h"
#include <iostream.h>

main() {
    Stack s1;

    s1.Push(99);    s1.Push(345);    s1.Push(235);

    for (int i=0; i<5; i++ ) { // causes 2 underflow errors
        s1.Pop();

        // check for error after each Pop()
        if (s1.GetError() != NO_ERROR)
            cout << "Error!" << endl;
    }
}
```

Member Function Syntax

- General syntax of member function implementation:

***ReturnType* *ClassName::memberFctName* (*ArgList*) {*Stmts*}**

int Location::SetX(int newX) {x=newX; return x;}

- where

- ReturnType*** - type of value returned (e.g, int),
- ClassName*** - class to which member function belongs
- memberFctName*** - function name
- ArgList*** - list of arguments for this member function
- Stmts*** - code defining what member function does when invoked

Try Defining this Shape Class

```
class Shape {  
  
    private:  
        int height;  
        int width;  
  
    public:  
        Shape(int width, int height); // specific shape  
        Shape();                     // default shape  
        int Height();                 // return height  
        int Width();                  // return width  
        Shape Resize(float factor);  // return adjusted shape  
        Boolean Equal(Shape s);      // are shapes same?  
        Boolean LessThan(Shape s);   // is one shape smaller?  
};
```

Section 4.4:

Organizing the Code

- Separate interface & implementation:
 - .h: Interface
 - .C, .cc, .cpp: Implementation

Organization Choices

■ .h files:

- 1 .h file = 1 class

Ex: Rectangle.h, Circle.h, ...

- 1 .h file = N related classes

*Ex: Graphics.h contains classes
Rectangle, Circle, ...*

■ .cpp files:

- 1 .cpp file = 1 class implementation (why?)

Ordering Your Code for the C++ Compiler

■ C++ is *statically typed*

So compiler must see class defn before class use

■ 3 cases

- 1. Shape.cpp #includes Shape.h
- 2. One .h file refers to second .h file
- 3. One .cpp file refers to another .h file

1. Shape.cpp #includes Shape.h

Shape.h:

```
class Shape {  
    ...  
    Shape(int x, int y);  
    ...  
}
```

Shape.cpp:

```
#include "Shape.h"  
Shape::Shape(int x,  
             float y)  
{  
    ...  
}
```

C++ compiler checks method "signatures": (int, int) vs. (int, float)

Compiler detects mismatch

2. One .h file refers to second .h file

Message.h:

```
#include "Location.h"  
class Message {  
    ...  
    Message(Location where);  
}
```

Compiler wants to check if argument class (Location) actually exists.

Message.cpp:

```
#include "Message.h"  
Message::Message(Location  
    where)  
{...}
```

Can compiler check if class Location exists even though Location.h isn't included?

3. One .cpp file refers to another .h file

FileChooser.h:

```
class FileChooser {  
private:  
    char* thePath;  
    char* theFilter;  
public:  
    FileChooser(  
        char* path,  
        char* filter);  
    File AskUser();  
}
```

FileChooser.cpp:

```
#include "Directory.h"  
File FileChooser::AskUser()  
{  
    Directory dir(thePath,  
                 theFilter);  
    ...  
}
```

.h needs no #includes; only .cpp refers to a second class

C++ Preprocessor

```
// A.h  
A1  
#include B  
A2
```

```
// B.h  
#include C  
B
```

```
// C.h  
#include D  
C
```

=

```
A1  
D  
C  
B  
A2
```

Preprocessor uses stack while scanning #include files.

Section 4.5:

A Simple Static Aggregation

An object with a fixed number
of simple internal parts.

Class Rectangle

How do we draw a rectangle?

Rectangle Class Implementation

```
void Rectangle::MoveUp(int deltaY)
{
    upperLeft    = Location(upperLeft.Xcoord(),
                           upperLeft.Ycoord() + deltaY);
    Reset3Corners();
}

// ... MoveDown, MoveLeft, MoveRight similar to MoveUp
```

Rectangle Class Implementation (Continued)

```
void Rectangle::Draw(Canvas& canvas) {
    canvas.DrawLine(upperLeft, upperRight);
    canvas.DrawLine(upperRight, lowerRight);
    canvas.DrawLine(lowerRight, lowerLeft);
    canvas.DrawLine(lowerLeft, upperLeft);
}

void Rectangle::Clear(Canvas& canvas) {
    canvas.Clear(upperLeft, area)
}

Rectangle::~~Rectangle() {}
```

A Suprising Fact

- These 4 variables are initialized twice!

Location upperLeft, upperRight, lowerLeft, lowerRight;

- Why?

Why Location Vars are Initialized Twice

```
class Location {  
    private: int currentX, currentY;  
    public: Location() {currentX=currentY= -1;}  
    ...};
```

```
class Rectangle {  
    private: Location upperLeft, ...; ...  
    Rectangle(Location corner, Shape shape)  
    { upperLeft = corner; ...;}
```

Once!

Twice!

Which runs first? Why?

- Subobjects constructors? (Rectangle)
- Object constructors? (Location)

```
class Rectangle {  
    private:    Location upperLeft, ...;  
}
```

(Subobject)

(Object)

Which order do destructors run in? Why?

Construction/Destruction Sequence

```
void main()  
{  
    cout<<"Begin Test"<<endl;  
    Rectangle rect(location);  
    cout<<"End Test"<<endl;  
}
```

Begin Test

Location default constructor
Location default constructor
Location default constructor
Location default constructor
Rectangle constructor
End Test

Rectangle destructor
Location (10,10) destructor
Location (10,10) destructor
Location (10,10) destructor
Location (10,10) destructor

main()
function
lifetime

Subobject
lifetime

Object
lifetime

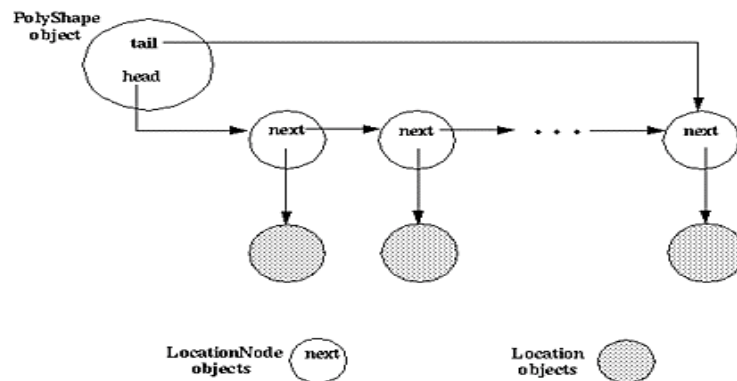
Section 4.7:

A Dynamic Aggregation

An object with a variable number of internal parts.

Example: a class representing polygon with an arbitrary number of sides.

The PolyShape Structure



PolyShape Class Interface

```
class PolyShape
{
    private:
        LocationNode *head;
        LocationNode *tail;
        int currentX, currentY;
        int length;
```

```
public:
    PolyShape(int x, int y);
    void Up (int n);
    void Down (int n);
    void Left (int n);
    void Right(int n);
    void Mark();
    void Draw(Canvas& canvas);
    ~PolyShape();
};
```

Basic Methods of the PolyShape Class

```
PolyShape::PolyShape (int x, int y)
{ currentX = x;
  currentY = y;
  Location *start = new Location(x,y);
  head=tail=new LocationNode(start);
  length = 1;
}
```

```
void PolyShape::Up(int n)
{ currentY = currentY - n; }

void PolyShape::Down(int n)
{ currentY = currentY + n; }

void PolyShape::Left(int n)
{ currentX = currentX - n; }

void PolyShape::Right(int n)
{ currentX = currentX + n; }
```

Adding to the PolyShape

```
void PolyShape::Mark()
{
    Location *newPoint = new Location(currentX, currentY);
    LocationNode *newNode = new LocationNode(newPoint);
    tail->Next(newNode);
    tail = newNode;
    length = length + 1;
}
```

Drawing a PolyShape

```
void PolyShape::Draw (Canvas& canvas)
{
    if (length == 1) return;
    LocationNode *node, *next;
    node = head;
    for(int i=0; i<length-1; i++)
    {
        next = node->Next();
        canvas.DrawLine(node->Contents(),
                        next->Contents());
        node = next;
    }
    canvas.DrawLine(head->Contents(), tail->Contents());
}
```

LocationNode Class

```
class LocationNode {
private:
    LocationNode *next;
    Location      *location;
public:
    LocationNode(Location *loc);
    LocationNode* Next();
    void Next(LocationNode* nxt);
    Location& Contents();
    ~LocationNode();
};
```

LocationNode Class (continued)

```
LocationNode::LocationNode(Location *loc)
{ location = loc;
  next = (LocationNode*)0;
}

LocationNode* LocationNode::Next()
{ return next; }

void LocationNode::Next(LocationNode* nxt)
{ next = nxt; }

Location& LocationNode::Contents()
{ return *location; }

LocationNode::~~LocationNode()
{ delete location; }
```

PolyShape Class Destructor

```
PolyShape::~~PolyShape()
{
    LocationNode *next = head;
    while (next)
    {
        LocationNode *node = next->Next();
        delete next;
        next = node;
    }
}
```

Opinion: Weaknesses of Text's PolyShape

- Lack of symmetry in LocationNode:
 - Class user (e.g., Mark()) allocates Location
 - But class LocationNode deallocates Location
- Complex interface:
 - Splitting atomic function of adding vertex into Up/Down/Left/Right + Mark can cause erroneous use.

Try this

- Implement the following:

void PolyShape::Add(Location &offset)

```
// Add new vertex using a LocationNode;  
// offset is relative to location specified in  
// PolyShape's constructor
```

A Solution

What could happen, if we don't make a copy of offset?

```
void PolyShape::Add(Location &offset) {  
    Location initial = *head;  
    Location *newPoint = new Location(  
        initial.Xcoord() + offset.Xcoord(),  
        initial.Ycoord() + offset.Ycoord());  
    LocationNode *newNode = new LocationNode(newPoint);  
    newNode->Next(head);  
    tail->Next(newNode);  
    tail = newNode;  
    currentX = newPoint->Xcoord();  
    currentY = newPoint->Ycoord();  
    length = length + 1;  
}
```

offset is relative to initial position.

Why make a copy of offset?

- Safety
- *offset* was created in another part of the program and could be deleted
 - Dangling reference in LocationNode
- When LocationNode is destructed, it deletes Location *location
 - Dangling reference in other part of program

A Simpler Solution

No need to repeat the code already used in Mark():

```
void PolyShape::Add(Location &offset) {
    Location initial = *head;
    currentX = initial.Xcoord() + offset.Xcoord();
    currentY = initial.Ycoord() + offset.Ycoord();
    Mark();
}
```

Or, we could rewrite Mark() to call the complex version of add()

Section 4.8:

Controlling Change

Goal:

Protect integrity of object/variable by limiting ways in which it may be modified.

Keyword: **const**

(confusion: location of **const** keyword varies!)

Constant Variables and Objects

```
const double Pi = 3.141593;           // mathematical constant
const int MAX_ARRAY_LENGTH = 100;     // system limit
const int YesAnswer = 0;              // program convention
const int NoAnswer = 1;              // program convention
const int VersionNumber = 1;         // program information
const int ReleaseNumber = 5;         // program information
const Location dialogLocation (200,200); // application information
```

■ Primitive types and Objects can be const

Constant Variables or Objects

Compiler flags assignments to constant variables/objects as errors.

Illegal Examples:

```
Pi = 2.5;  
NoAnswer = 2;  
dialogLocation = Location(100,100);
```

MS VC++ 5.0 compiler error message:

FileName.cpp(35) : error C2166: l-value specifies const object

Constant *Methods* in Class Location

```
class Location { // Extension 1  
private:  
    int currentX, currentY;  
  
public:  
    Location(int x, int y); // specific location  
    Location(); // default location  
    int Xcoord() const; // return x-axis coordinate  
    int Ycoord() const; // return y-axis coordinate  
    void setX(int newX); // change x coordinate  
    void setY(int newY); // change y coordinate  
};
```

const methods won't modify currentX/Y.

Constant Methods in the Location Class

```
// Implementation follows
Location::Location( int x, int y )
    { currentX = x; currentY = y; }
Location::Location () { currentX = -1; currentY = -1; }
int Location::Xcoord() const { return currentX; }
int Location::Ycoord() const { return currentY; }
void Location::setX(int newX) { currentX = newX; }
void Location::setY(int newY) { currentY = newY; }
```

const must appear *both* in interface *and* implementation

Explain why statements are OK or Error...

```
const Location dialogLocation (200,200);
//...
int dialogX = dialogLocation.Xcoord(); // OK
int dialogY = dialogLocation.Ycoord(); // OK
dialogLocation.setX(300); // ERROR

Location loc(20, 50);
// ...
int locX = loc.Xcoord(); // OK
int locY = loc.Ycoord(); // OK
loc.setX(300); // OK
```

const Parameters

```
class Location { // Extension 2
    // private data
public:
    // ... other public methods
    int isSameAs(const Location& other) const;
    // is other same as this Location?
};
```

other will not
be changed

this will not
be changed

const Parameters (cont'd)

Implementation must match interface declaration:

```
int Location::isSameAs(const Location& other) const {
    if ((currentX == other.currentX) &&
        (currentY == other.currentY))
        return 1;
    else return 0;
}
```

Using const Parameters

```
const Location dialogLocation(200, 200);
const Location somewhere;

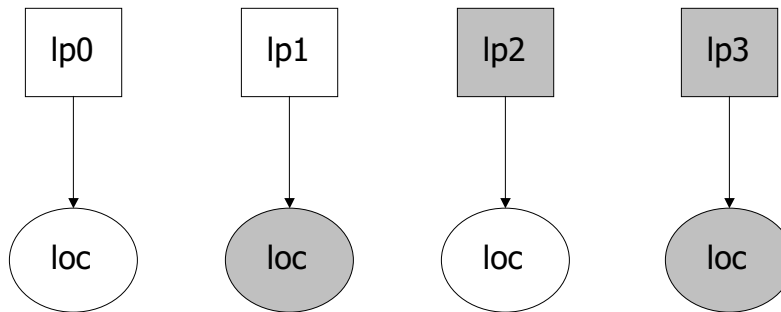
// ...
if (dialogLocation.isSameAs(somewhere) {
    // ...
}
```

const Pointers

```
Location loc(100, 100);
Location other(50, 50);
Location *lp0           = &loc; // unconstrained
const Location *lp1     = &loc; // object constant
// Location const *lp1  = &loc; // same
Location * const lp2    = &loc; // pointer constant
const Location * const lp3 = &loc; // both constant
// Location const * const lp3 = &loc // same
```

*Note: book is wrong - Location const * lp2 is object constant*

const Pointers Diagram



Shaded parts are const

Using const Pointers

```
lp0->Xcoord(); // Okay
lp0->setX(10); // Okay
lp0 = &other; // Okay
```

```
lp1->Xcoord(); // Okay
lp1->setX(10); // Error
lp1 = &other; // Okay
```

```
lp2->Xcoord(); // Okay
lp2->setX(10); // Okay
lp2 = &other; // Error
```

```
lp3->Xcoord(); // Okay
lp3->setX(10); // Error
lp3 = &other; // Error
```

Compiler error messages:

```
C:\Sample1.cpp(38) : error C2662: 'setX' : cannot convert 'this' pointer from 'const
class Location' to 'class Location &' Conversion loses qualifiers
```

```
C:\Sample1.cpp(47) : error C2166: l-value specifies const object
```

const Pointer Parameters

```
int Location::isInList(const Location * const list,
                      int length) const {
    for (int i = 0; i < length; i++)
        if ((currentX == list[i].currentX) &&
            (currentY == list[i].currentY))
            return 1;
    return 0;
}
```

array elements will
not be changed

pointer (list) will not
be changed

this will not
be changed

Section 4.9

Managing Dynamically Allocated Storage

Goals:

1. Avoid unintended early deletion
2. Avoid memory leaks

Simple Example:

```
class Message {
private:
    char* message;
    Location position;
public:
    Message (char* text, Location p);
    Message (char* text);
    ~Message() { delete message; }
};
```

pointer & dynamic allocation

Remember this!

Unintended Early Deletion Caused by Copying

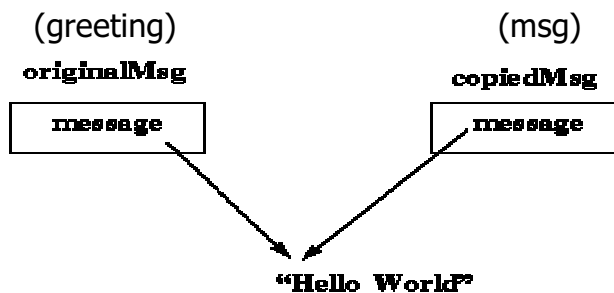
```
class Displayer {  
    public:  
    ...  
    void Show(Message msg);  
};  
  
Message greeting ("Hello World");  
Displayer display;  
  
display.Show(greeting);
```

// pass by copy

// pass by copy

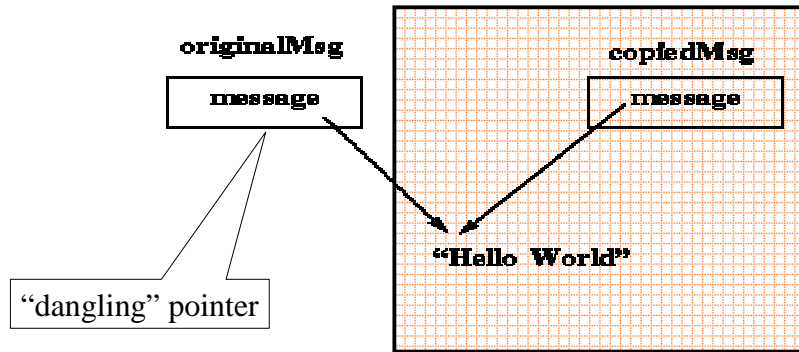
Shared Data: What happens in Show(greeting)

Default copy constructor *copies original object byte-by-byte*.
It doesn't traverse pointers!



What happens when "msg" goes out of scope at end of display.Show(greeting) call?

Shared Data



Answer: All of the memory used by variables and objects in the shaded box is freed, including the actual text.

Solution: Define a Copy Constructor

Default copy constructor *copies original object byte-by-byte*.
It doesn't traverse pointers!

```
class Message {
private:
...
public:
    Message(const Message& other); // copy constructor
};
Message::Message(const Message& other) {
    message = copystring(other.message); }
```

This copy constructor initializes object's private data using private data of another object in same class.

Managing Dynamically Allocated Storage

```
Message::Message(char* text, Location p) {  
    position = p;  
    message = copystring(text);  
}
```

dynamic allocation

```
Message::~Message(){ delete message; }
```

::copystring (from wxWindows documentation)

```
char * copystring(char *s)
```

Makes a copy of string s using C++ new operator, so it can be deleted with **delete** operator.

Defining a Copy Constructor

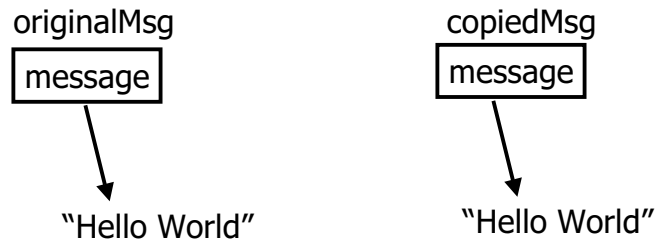
```
class Message {  
    private:  
    ...  
    public:  
        Message(const Message& other); // copy constructor  
};  
Message::Message(const Message& other) {  
    message = copystring(other.message); }
```

Both must be the class name.
Also, only 1 parameter.

Reference

const keyword

Copied Data



Message copied when there is a copy constructor that copies data that is pointed to by private data.

Section 4.10

Memory Leaks Due to Assignment

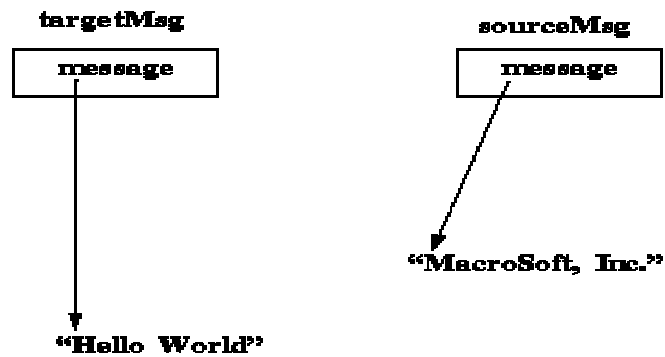
Given the following statements:

```
Message targetMsg("Hello Word");  
Message sourceMsg("MacroSoft, Inc.");
```

What is the effect of an assignment statement like:

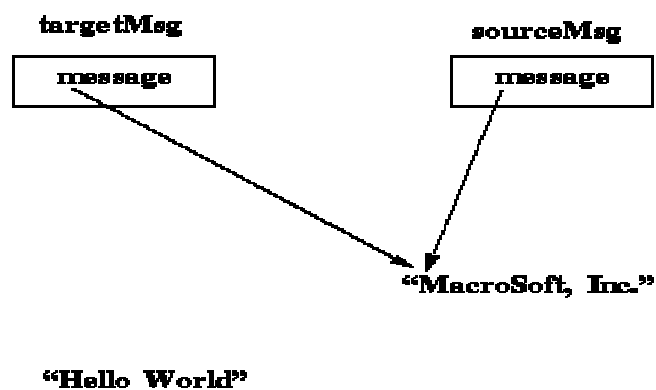
```
targetMsg = sourceMsg;    // assignment
```

Before Assignment Statement

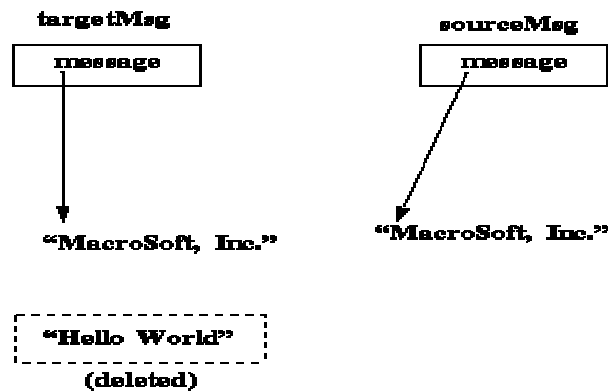


After Assignment Statement

The default means for assigning an object is a bit-wise assignment.



Desired Result of Assignment



Defining a Class-Specific Assignment Operator

```
class Message {
private:
    char* message;
public:
    Message& operator=(const Message& source);
};

Message& Message::operator = (const Message& source) {
    delete message; // deallocate previous string
    message = copystring(source.message); // copy new string
    return *this;
}
```

Notice the signature!

Overloading the assignment operator

lhsObject = rhsObject;

Allows chaining
e.g., a=(b=c);

Right hand side
object (rhsObject)

```
ClassName& ClassName::operator=(const ClassName& source) {  
    //... perform the assignment  
    return *this;  
}
```

this is the lhsObject

Solid C++ Classes

A solid C++ class should always include the following:

- ! **destructor** - Deallocate resources owned by object
- ! **null constructor** - Constructor with no arguments. Needed to create arrays of objects
- ! **copy constructor** - Constructor with specific signature. Implicitly used when copying an object.
- ! **assignment operator** - Overload the assignment operator (=). Implicitly used when an object is assigned to another.

Section 4.11

Other Class Features

- inline methods
- this pointer
- private methods
- static variables
- friend classes

Normal Method Invocation

- Method invocations have processing overhead:

poly.Up(10); ==>

- save registers on the stack
- create new call frame
- push function arguments on the stack
- go to poly.Up
- remove call frame
- restore saved registers

Normal versus Inline Method Invocation

```
PolyShape quad(20, 20);  
int deltaY = 10;  
quad.Up(deltaY);
```

Normal Invocation

```
STORE  deltaY, 10  
PUSH   AX  
PUSH   BX  
PUSH   deltaY  
JMP    UP_CODE  
POP    BX  
POP    AX
```

Inline Expansion

```
STORE  deltaY, 10  
MOVE   AX, quad.currentY  
ADD    AX, deltaY  
STORE  AX, quad.currentY
```

Inline Methods

- Generally more efficient for *small* methods
- Expanded in-place of invocation
 - Eliminates method invocation overhead
 - But can increase executable size
- Two ways to specify an inline method
 - Provide implementation during class definition (default inline)
 - Use 'inline' keyword (explicit inline)

Inline Method Examples

```
class PolyShape {  
    private: // ...  
    public: //...  
        void Up(int n) {currentY = currentY - n;}  
        void Down(int n) {currentY = currentY + n;}  
};
```

default inline

```
inline void PolyShape::Up(int n) {  
    currentY = currentY - n;  
}
```

explicit inline

Tradeoffs of inline methods

- Default inline exposes implementation
- All code that invokes an inline must be recompiled when:
 - method is changed
 - switching from inline to regular or vice-versa
- Inline is request, not command to compiler
- Executable size may increase

'this' pointer

- A predefined variable which is a pointer to the object itself.
- Examples:
 - within class Message:
 - | Message * this;
 - within class Location:
 - | Location * this;

Using "this" (Observer-Observable Pattern)

```
class Listener { // Observer of Location objects
public:
    void LocationMoved(Location * loc); // called when a Location moves
};
```

```
class Location { // Observable
    int currentX, currentY;
    Listener *listener;
public:
    void AddListener(Listener& listener) {this->listener = &listener;}
    void Move(int x, int y);
};
void Location::Move (int x, int y) {
    currentX = x; currentY = y;
    if (listener) listener->LocationMoved(this);
}
```

this distinguishes between listener variables

Notify listener that this Location has moved

Using “this” (cont’d)

- Return a reference to the object itself

```
class PolyShape {
private:
    // same before
public:
    PolyShape & Up(int n);
    PolyShape & Down(int n);
    PolyShape & Left(int n);
    PolyShape & Right(int n);
    PolyShape & Mark();
};
```

What if these functions
returned by copy instead?

```
PolyShape & PolyShape::Up(int n) {
    currentY -= n;
    return*this;
}
```

```
// usage
PolyShape quad(20, 20);
quad.Right(100).Down(50).Mark();
quad.Left(20).Mark();
```

Pretty cool, huh?

Private Method

- A method that can only be called within the class
- Avoids duplication of code
- Useful for:
 - Sub-Algorithms
 - Error Checking
 - Repeated Code
 - Intermediate values

Private Method Example

```
Class Rectangle {
private:
    Location upperLeft;
    // Other Location ...
    Shape area;
    void AdjustCorners(); // private method
public:
    Rectangle( Location corner, Shape shape);
    void MoveUp( int deltaY);
    // ... Other public methods
};
```

Example(cont'd)

```
void Rectangle::AdjustCorners() {
    upperRight = Location(upperLeft.Xcoord() + area.Width(), upperLeft.Ycoord());
    lowerLeft = Location(upperLeft.Xcoord(), upperLeft.Ycoord() + area.Height());
    lowerRight = Location(upperLeft.Xcoord() + area.Width(),
                          upperLeft.Ycoord() + area.Height());
}
Rectangle::Rectangle(Location corner, Shape shape) {
    area = shape;
    upperLeft = corner;
    AdjustCorners(); // call private method
}
void Rectangle::MoveUp (int deltaY) {
    upperLeft = Location(upperLeft.Xcoord(), upperLeft.Ycoord() + deltaY);
    AdjustCorners(); // call private method
}
```

Static Variable

- Class-wide data (aka "class variable")
- Shares data among all instances
- Avoids need for global variables
- Restrictions:
 - Must be initialized, but
 - Should not be initialize in constructor. Why?

Static Variable Example

```
class Rectangle {  
    private:  
        // ... as before  
        static Color rectangleColor; // class variable  
    public:  
        Rectangle (Location corner, Shape shape);  
        void setColor(Color newColor);  
        //.. other methods as before  
};
```

```
Color Rectangle::rectangleColor = Color(200,0,0); //a shade of red  
  
void Rectangle::setColor(Color color) {  
    rectangleColor = color; // change the color for all Rectangle objects  
}
```

Friend Classes

- Allow access to private members
 - Not Symmetric
 - | Just because you trust me, doesn't mean that I trust you
 - Not Transitive
 - | Your friends are not necessarily my friends
- Sometimes useful
 - For Efficiency
 - For Security

Rectangle3.h

```
class Rectangle3 {  
private:  
    Location upperLeft;  
    Location upperRight;  
    Location lowerLeft;  
    Location lowerRight;  
    Shape area;  
    static Color rectangleColor; // class variable  
    Rectangle3 (Location corner, Shape shape);  
    friend class RectangleManager;  
  
public:  
    // no public constructor  
    ~Rectangle3();  
};
```

Allow RectangleManager
to access private
data and methods

RectangleManager.h

```
class RectangleManager {
private:
    Rectangle3 ** rects;    // array of Rectangle pointers
    Shape commonShape;    // common size for managed rectangles
    int numRects;        // number of managed rectangles
public:
    RectangleManager(Shape shapeForAll=Shape(100,100));
    void CreateRectangleAt(Location loc);    // create Rectangle
    Rectangle3 * GetRectangleAt(int x, int y); // returns Rectangle
        // that contains these coordinates, or null
    void Draw(Canvas& canvas);    // draw managed rects
    ~RectangleManager();
};
```

RectangleManager.cpp

```
RectangleManager::RectangleManager(Shape shapeForAll):
    // subobject construction list
    numRects(0) // construct an int with variableName(intValue)
{
    commonShape = shapeForAll;
    rects = new Rectangle3*[MaxRects];
}
void RectangleManager::CreateRectangleAt(Location loc){
    if (numRects < MaxRects) {
        rects[numRects++] = new Rectangle3(loc, commonShape);
    }
}
RectangleManager::~~RectangleManager(){
    for (int i = 0; i < numRects; i++)
        delete rects[i];
    delete rects;
}
```

Safety: Because the constructor is private, only RectangleManager can create Rectangles, ensuring they all have common size.

RectangleManager.cpp (cont'd)

```
Rectangle3 * RectangleManager::GetRectangleAt(int x, int y) {
    for (int i = 0; i < numRects; i++) {
        if ((x >= rects[i]->upperLeft.Xcoord() &&
            (x <= rects[i]->upperRight.Xcoord() &&
            (y >= rects[i]->upperLeft.Ycoord() &&
            (y <= rects[i]->lowerLeft.Ycoord() ) ) {
            return rects[i];
        }
    }
    return (Rectangle3*)0; // null
}
```

Accessing private data directly
can be more **efficient** than
via method invocation

```
void RectangleManager::Draw(Canvas& canvas){
    canvas.Clear();
    for (int i = 0; i < numRects; i++)
        rects[i]->Draw(canvas);
}
```

Using RectangleManager

```
Frame window("Rectangle Test", Location(100,100), Shape(300,300));
Canvas canvas(window, "Rectangle Area", Location(10,10), Shape(200,200));
RectangleManager squareManager(Shape(50,50));

void OnStart() {
    squareManager.CreateRectangleAt(Location(10,10)); // create rectangles
    squareManager.CreateRectangleAt(Location(70,10));
    squareManager.CreateRectangleAt(Location(20,100));
    squareManager.CreateRectangleAt(Location(20,150));
    OnPaint();
}

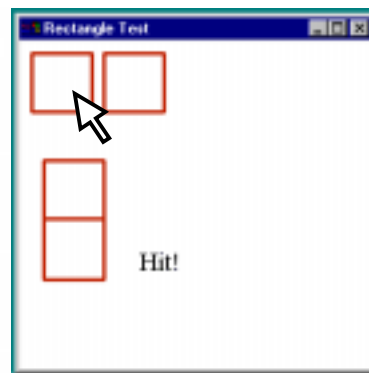
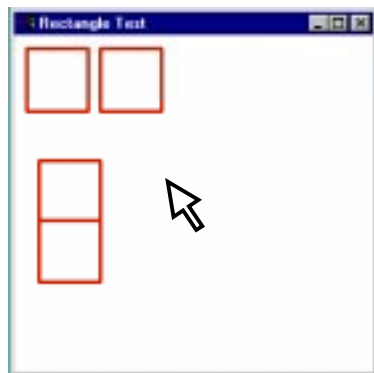
void OnPush(char* buttonName) {
}

void OnTimerEvent(char* timerName){
}
```

Using RectangleManager (cont'd)

```
int drawMessage = FALSE;
void OnPaint() {
    squareManager.Draw(canvas);
    if (drawMessage) {
        canvas.DrawText("Hit!", Location(100,175));
    }
}
void OnMouseEvent(char* canvasName, int x, int y, int buttonState) {
    if ((buttonState & leftButtonDown)) {
        if (squareManager.GetRectangleAt(x,y) != (Rectangle3*)0)
            drawMessage = TRUE;
        else
            drawMessage = FALSE;
        OnPaint();
    }
}
```

RectangleManager



Color Class

```
#ifndef _COLOR_H
#define _COLOR_H

class Color
{
private:
    int redValue;
    int greenValue;
    int blueValue;
public:
    Color(int red, int green, int
blue);
    int Red();
    int Green();
    int Blue();
    ~Color();
};
#endif
```

```
#include "Color.h"

Color::Color(int red, int green, int blue){
    redValue = red;
    greenValue = green;
    blueValue = blue;
}

int Color::Red(){ return redValue; }
int Color::Green() { return greenValue; }
int Color::Blue() { return blueValue; }

Color::~~Color() {}
```

Problem: Design a ShapeManager (Class?)

One solution consists of the following classes:

- Circle: Abstracts a circle object.
- MovingCircle: Abstracts a circle object that can be moved.
- CircleList: Abstracts a linked list of circles.
- CircleNode: Abstracts each node of the list of circles.

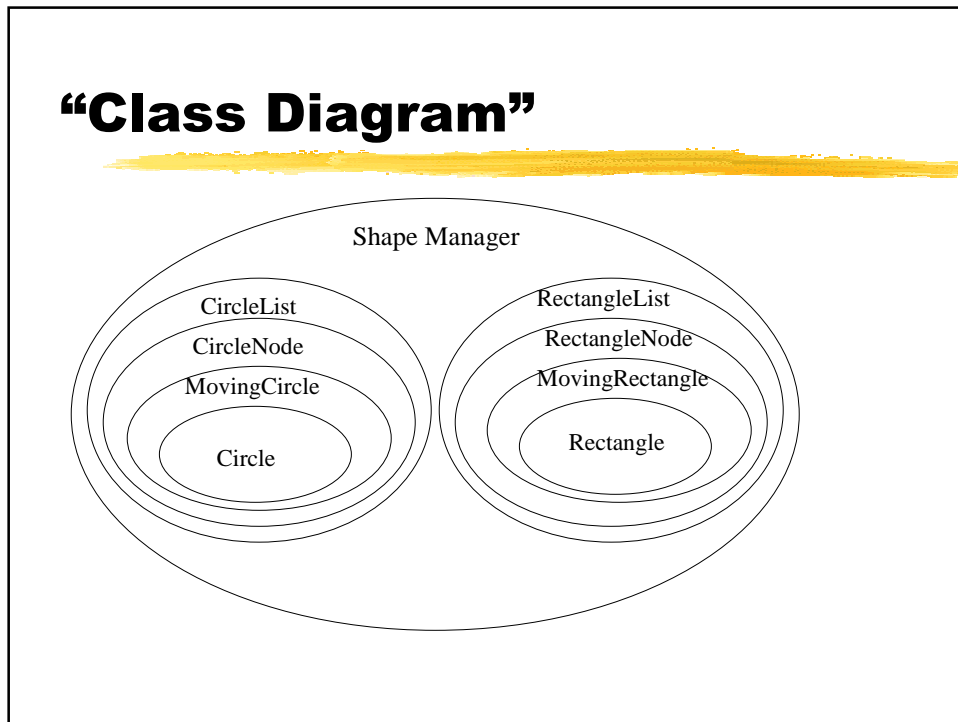
... More Classes ...

- Rectangle: Abstracts a rectangle object.
- MovingRectangle: Abstracts a rectangle object that can be moved.
- RectangleList: Abstracts a linked list of rectangles.
- RectangleNode: Abstracts each node of the list of rectangles.
- ShapeManager: Manages all the rectangles and circles.

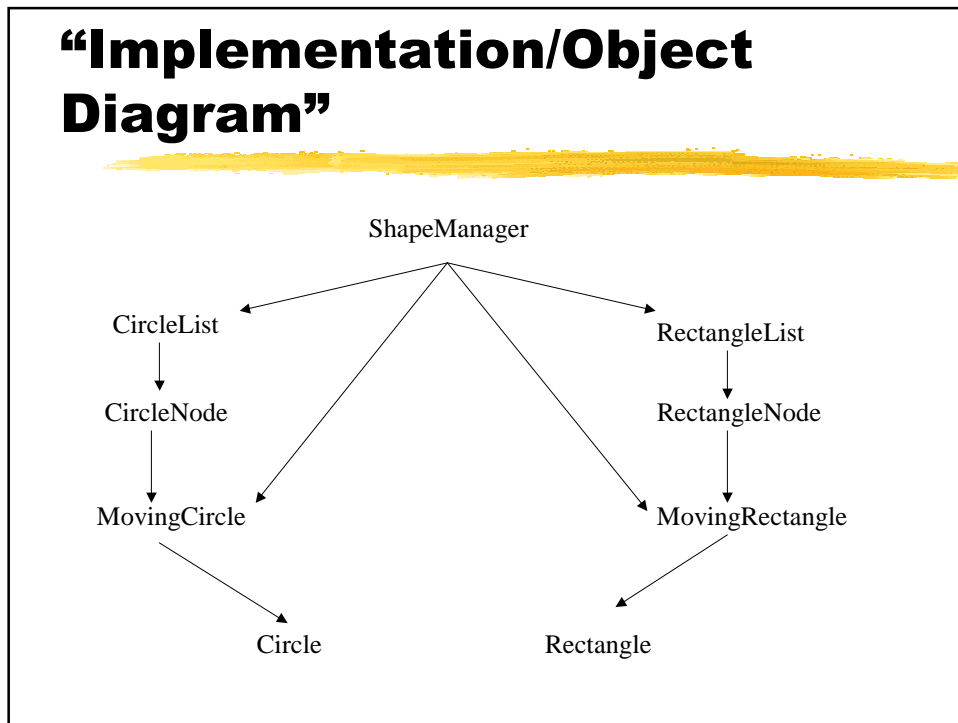
Useful Diagrams

- "Class Diagram": Shows which class "contains" another class.
- "Implementation/Object Diagram": Shows which class objects can access (or are aware of) which other class objects.

“Class Diagram”



“Implementation/Object Diagram”



Interface

```
class Circle {  
    private:  
        float radius;  
        Location center;  
        :  
    public:  
        Circle();  
        Circle(float radiusToBeSet, location centerToBeSet);  
        void Resize(float factor);  
        void Draw();  
        void ChangeCenter(Location newCenter);  
        ~Circle();  
        :  
};
```

No dynamic allocation

Copy constructor and assignment operator not needed

Destructor might not do anything

class MovingCircle

```
{ private:  
    Circle circle;  
    float direction;  
    float speed;  
    : // and others  
public:  
    MovingCircle();  
    MovingCircle(float center, float radius);  
    void MoveCircle();  
    :  
    ~MovingCircle();  
};
```

No dynamic allocation

Copy constructor and assignment operator not needed

Destructor might not do anything

class CircleNode

```
{ private:
    MovingCircle *mCircle;
    CircleNode *next;
    :
public:
    CircleNode();
    CircleNode(MovingCircle movCircle);
    CircleNode (const CircleNode& other);
    void operator= (const CircleNode& source);
    CircleNode* GetNext();
    :
    ~CircleNode();
};
```

Dynamic allocation

Copy constructor and assignment operator needed to copy node

Destructor must free dynamic memory by deleting mCircle

class CircleList

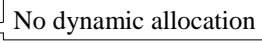
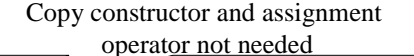
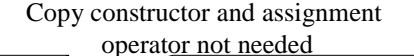
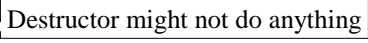
```
{ private:
    CircleNode *head;
    :
public:
    CircleList();
    CircleList(CircleList* initialHead);
    CircleList(const CircleList& other);
    void operator= (const CircleList& source);
    void insert(CircleList* node);
    Boolean remove(CircleList* node);
    :
    ~CircleList();
};
```

Dynamic allocation

Copy constructor and = operator are needed to copy whole list

Destructor must free dynamic memory by deleting each node in the list

Class ShapeManager

```
{  
  private:  
    CircleList circleList;   
    RectangleList rectangleList;  
    :   
  public:  
    ShapeManager ();   
    void MoveAllCircles ();  
    :  
    ~ShapeManager ();   
};
```