Chapter 4

Implementing a New Class

Progression of Roles

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
Using Association to Build a StopWatch

Using Aggregation to Build a StopWatch

SimpleTimer Aggregation

SimpleTimer = StopWatch without buttons
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
  - Stack - varying size
    - Stack::Push(int) can grow stack size

Try it!
Review - Class Stack

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

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* temp = new int[2*size]; size = size*2;
    // copy current contents into new array
    for (unsigned i=0; i<numItems; i++) { temp[i] = stack[i]; }
    // swap old & new stacks; delete old stack
    stack = temp; delete stack;
    // Finally! We can push new value onto stack
    stack[numItems] = value;
}

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

```cpp
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

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

```cpp
class Location {
    private:
        int currentX, currentY;
    public:
        Location(int x, int y);
        Location() : Location(0, 0) {} // Default location
        int Xcoord(); // Return x-axis coordinate
        int Ycoord(); // Return y-axis coordinate
        ~Location();
};
```

Solution...

Not using accessor:
```cpp
int Location::Equal(Location& other)
{
    return (currentX == other.currentX
            && currentX == other.currentY);
}
```

Using accessor:
```cpp
int Location::Equal(Location& other)
{
    return (currentX == other.Xcoord()
            && currentX == 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.*
en um ErrorType { NO_ERROR, STACK_UNDERFLOW, STACK_OVERFLOW };

class Stack {
public:
    Stack(unsigned sz=5);
    void Push(int value);
    int Pop();
    ErrorType GetError();
private:
    unsigned size;
    unsigned numItems;
    int* stack;
    ErrorType errorType;
};

void 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; }

#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::memberFcnName (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.
  - `memberFcnName` - function name.
  - `ArgList` - list of arguments for this member function.
  - `Stmts` - code defining what member function does when invoked.

---

### Recall class Counter...

```cpp
class Counter {
private:
    int initial;
    int value;
    Message* message;

public:
    Counter(int start = 0);
    void ConnectTo(Message& msg);
    void Next();  // Inc value & tell message
    void Reset();  // Reset to initial
    ~Counter();
};
```

Try writing the implementation in small groups!

---

### Counter Class Implementation

```cpp
Counter::Counter (int start) {
    initial = start;
    value = start;
    message = (Message*)0;
}

void Counter::ConnectTo(Message& msg) {
    message = &msg;
}
```
Counter Class Implementation

```cpp
void Counter::Next()
{
    value++; // Increase the value of the counter.
    stringstream convert(asString,10);
    convert << value << '0';
    if (message) message->SetText(asString);
}

void Counter::Reset(){ value = initial; }

Counter::Counter() {}  // Constructor
Counter::~Counter() {}   // Destructor
```

---

Add Equal Method to Counter Class

```cpp
class Counter{
private:
    ...
    int value;
    ...

public:
    ...
    int Equal(Counter& other);   // return 1 iff this & other have same value
};
```

Try writing the Equal method now.

---

Add Equal Method to Counter Class

```cpp
int Counter::Equal(Counter& other)
{
    if (value == other.value) // Check if the values are equal.
        return 1; // equal
    else
        return 0;    // not equal
}
```

Does this violate encapsulation?
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, int 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);
    ...
};
```

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

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

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

FileChooser.cpp: .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

// D.h
D
```

= A1
B

Preprocessor uses stack while scanning #include files.

---

Section 9.2:

**Class Design: Perspectives**
3 Perspectives

- **Behavioral:**
  - Emphasizes actions in system
- **Structural:**
  - Emphasizes relationships among components
- **Information:**
  - Emphasizes role of information/data/state and how it's manipulated

Give examples of each perspective for Project 1

Examples of each perspective for Project 1

- **Behavioral (actions):**
  - Shapes move
  - Click mouse to score
- **Structural (relationships):**
  - Ords and Squares are kinds of shapes
  - 2 windows contain shapes; 1 contains controls
- **Information (state):**
  - What’s user’s score?
  - How many shapes?
  - What are colors, sizes, locations of each shape?

Behavioral Perspective

*Consider some action in a program...*

What object...
  - initiates action?

What objects...
  - help perform action?
  - are changed by action?
  - are interrogated during action?

How does this apply to project 1?
Behavioral Perspective

Consider mouse click action
What object...
- initiates action? ⇒ User
What objects...
- help perform action?
  ⇒ Globals (OnMouseEvent)
- are changed by action?
  ⇒ Possibly some shape
- are interrogated during action?
  ⇒ ShapeManager (sees if mouse lies in any shape)

Behavioral Classes

• Actor (does something)
  ⇒ Game object with Start() & Stop()
• Reactor (system events, external & user events)
  ⇒ mouse click reactor, buttons
• Agent (messenger, server, finder, communicator)
  ⇒ Communicator lets user choose file to load old game
• Transformer (data formatter, data filter)

Structural Perspective

• What objects...
  - are involved in relationship?
  - are necessary to sustain (implement, realize, maintain) relationship?
• What objects not in relationship...
  - are aware of and exploit relationship?

How does this apply to project 1?
Structural Perspective

"2 windows contain shapes; 1 contains controls"
- What objects...
  - are involved in relationship?
    ⇒ Frames, Shapes, Controls
  - are necessary to realize relationship?
    ⇒ ShapeManager (maps shapes to frames)
    ControlArea (holds buttons, sliders, ...)
- What objects not in relationship...
  - are aware of and exploit relationship?
    ⇒ Clock associates with ShapeManager,
    which distributes ticks to all Shapes

Structural Classes

- Acquaintance (symmetric, asymmetric)
  - ShapeManager might be asymmetric
    (it knows about shapes and frames, but
    frames/shapes are unaware of ShapeManager)
- Containment (collaborator, controller)
  - ControlArea class contains buttons, textboxes,...
- Collection (peer, iterator, coordinator)
  - ShapeManager might have iterator so external
    user can traverse all shapes

Information Perspective

What objects...
- represent the data or state?
- read data or interrogate state?
- write data or update state?

How does this apply to project 1?
Data Versus State

<table>
<thead>
<tr>
<th>Data</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>• <strong>Definition:</strong> Info processed by system</td>
<td>• <strong>Definition:</strong> Info used by system to perform processing</td>
</tr>
<tr>
<td>• <strong>Example:</strong> Text you type into a word processor</td>
<td>• <strong>Example:</strong> Variables specifying font, size, width</td>
</tr>
</tbody>
</table>

Information Perspective

What objects represent the state?
- **Circle, Rectangle** give color, size, location

read or interrogate state?
- **ShapeManager** might ask **Shape** if it contains x,y mouse coordinate

write data or update state?
- **ShapeManager** might tell Shapes to move

Summary

- Try creating Table 9-5 for Project 1!
Behavioral Perspective

- **Actors:** InterfaceManager
- **Reactors:** DisplayManager, EventManager
- **Agent:**
  - Servers: FileReader
  - Communicator: CheckBox, Slider, TextBox, Button

---

Behavioral Perspective

- **Transformer:**
  - Filter: Filter
  - Formatter: Plotter

---

Structural Perspective

- **Acquaintance:**
  - Symmetric:
    - None
  - Asymmetric:
    - EventManager with FileReader, Timer, DisplayManager
    - InterfaceManager with DisplayManager
    - DisplayManager with FileReader
Structural Perspective

- **Containment:**
  - **Collaborator:**
    - User Manager contains Frame, Panel, Sliders, CheckBox, TextBox, Buttons
    - DisplayManager contains Canvas, Color, Symbol, Plotter, Filter
  - **Controller:** EventManager controls DisplayManager and InterfaceManager
- **No collection objects**

Information Perspective

- **Data:**
  - Source: FileReader
  - Sink: Plotter
  - Result: Location
  - Synchronizers: Timer, EventManager
- **State:**
  - There are is real state information stored
Evaluating a Class Design

- Evaluation needed to accept, revise or reject a class design
- Five aspects to be evaluated:
  - Abstraction: useful?
  - Responsibilities: reasonable?
  - Interface: clean, simple?
  - Usage: "right" set of methods?
  - Implementation: reasonable?

Tests to determine adequacy of Abstraction

- **Identity:**
  Are class & method names simple & suggestive?

- **Clarity:**
  Can purpose of class be given in brief, dictionary-style definition?

- **Uniformity:**
  Do operations have uniform level of abstraction?

Good or Bad Abstractions?

- **class Date:**
  Date represents a specific instant in time, with millisecond precision.

- **class TimeZone:**
  - TimeZone represents a time zone offset, and also figures out daylight savings.
### Tests to determine adequacy of Responsibilities

- **Clear:**
  Does class have specific responsibilities?

- **Limited:**
  Do responsibilities should fit abstraction (no more/less)?

- **Coherent:**
  Do responsibilities make sense as a whole?

- **Complete:**
  Does class completely capture abstraction?

### Tests to determine adequacy of Interface

- **Naming:**
  Do names clearly express intended effect?

- **Symmetry:**
  Are names & effects of pairs of inverse operations clear?

- **Flexibility:**
  Are methods adequately overloaded?

- **Convenience:**
  Are default values used when possible?

### Example of Poor Naming:

class ItemList
{
    // Hard to remember difference!
    public:
        void Delete(Item item);
            // take Item node out of list and delete Item
        void Remove(Item item);
            // take Item's node out of the list but do not delete Item
        void Erase(Item item);
            // keep Item's node in List but with no information
};
Tests to determine adequacy of Usage

- Examine how objects of the class are used in different contexts (see next slide...)
- Incorporate all operations that may be useful in these contexts

Original Location Class:

```java
class Location {
    private:
        int xCoord, yCoord; //coordinates

    public:
        Location(int x, int y):
            xCoord(x), yCoord(y) { //return xCoord value
            yCoord(y); //return yCoord value
        }

        //Usage
        Location point[100, 100];
        ... point = Location[point.xCoord()=5, point.yCoord()+10]; } //shift: point
```

Revised Location Class:

```java
class Location {
    private:
        int xCoord, yCoord; //coordinates

    public:
        Location(int x, int y):
            xCoord(x), yCoord(y) { //return xCoord value
            yCoord(y); //return yCoord value
            void shiftBy(int dx, int dy): // shift by relative coordinates
        }

        //Usage
        Location point[100, 100];
        ... point.shiftBy(5, 10); } //shift point
```
Implementation

- Least important, mostly easily changed aspect to be evaluated

- Complex implementation may mean
  - Class not well conceived
  - Class has been given too much responsibility

A Simple Static Aggregation

An object with a fixed number of simple internal parts.

Class Rectangle

How do we draw a rectangle?
```cpp
class Rectangle {
    private:
        Location upperLeft, upperRight, lowerLeft, lowerRight;
        Shape area;
    public:
        Rectangle (Location corner, Shape shape);
        void MoveUp (int deltaY);
        void MoveDown (int deltaY);
        void MoveLeft (int deltaX);
        void MoveRight (int deltaX);
        void Draw (Canvas& canvas);
        void Clear (Canvas& canvas);
        ~Rectangle () ;
};
```

```
class RectangleImplementation {
    void Rectangle::MoveUp (int deltaY) {
        upperLeft = Location (upperLeft.xCoord() + area.Width(),
                                upperLeft.yCoord());
        lowerLeft = Location (lowerLeft.xCoord(),
                               lowerLeft.yCoord() + area.Height());
        lowerRight = Location (lowerRight.xCoord() + area.Width(),
                                lowerRight.yCoord() + area.Height());
    }
```

```
// ... MoveDown, MoveLeft, MoveRight similar to MoveUp
```
Rectangle Class Implementation (Continued)

```cpp
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

```cpp
class Location {
    private: int currrentX, currrentY;
    public: Location() {currrentX=currrentY= -1;}
};

class Rectangle {
    private: Location upperLeft, ...;
    
    Rectangle(Location corner, Shape shape) {
        upperLeft = corner; ...;
    }
```
Which runs first? Why?

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

```java
class Rectangle {
    private: location upperLeft, ...
} (Subobject)
```

Which order do destructors run in? Why?

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

Section 4.6:

A More Complex Static Aggregation

Consider object with a fixed number of more complicated internal parts.
StopWatch Class Interface

class StopWatch {
  private:
    Button startButton;
    Button stopButton;
    Clock clock;
    Counter clockCount;
    Message clockDisplay;
    Panel buttonPanel;
    Canvas canvas;
}

public:
  StopWatch(Frame& frame,
    Location where,
    int interval = 1000);
  void ButtonPushed(char* buttonName);
  void Tick();
  int ElapsedTime();
  ~StopWatch();
};

StopWatch Class Implementation

void StopWatch::ButtonPushed(char* buttonName)
{ if (startButton.IsNamed(buttonName))
    clock.Start();
  else if (stopButton.IsNamed(buttonName))
    clock.Stop();
}

void StopWatch::Tick(){ clockCount.Next();}

int StopWatch::ElapsedTime() { return clockCount.Value(); }

StopWatch::~StopWatch() {}

Section 4.6, page 165:

Subobject Construction
Layout of the StopWatch User-Interface Subobjects

Placement of User Interface Subobjects

Let \( x = \text{where.xcoord()} \) and \( y = \text{where.ycoord()} \):

Message: Location(60,10) in Canvas
           (shape set automatically)
StartButton: Location(10,10) in Panel
             Shape(50,20)
StopButton: Location(70,10) in Panel
             Shape(50,20)
Canvas: Location(\( x+10, y+20 \)) in Frame
        Shape(130, 30)
Panel: Location(\( x+10, y+60 \)) in Frame
       Shape(130, 40)

Constructor form

global objects;

ClassName::ClassName(args) :
    constructor(args),
    constructor(args),

{ constructor body }
StopWatch Constructor (1 of 3)

//StopWatch Class Constructor

Location StartButtonLocn = Location(10,10);
Shape StartButtonShape = Shape(50,20);
Location StopButtonLocn = Location(70,10);
Shape StopButtonShape = Shape(50,20);
Location ButtonPanelLocn = Location(10,60);
Shape ButtonPanelShape = Shape(130,40);
Location CanvasLocn = Location(10,20);
Shape CanvasShape = Shape(130,30);
Location ClockDisplayLocn = Location(60,10);

StopWatch Constructor (2 of 3)

StopWatch::StopWatch(Frame frame, Location where, int interval):
// sub-object constructor list

counter(0),
clock("StopWatchClock", interval),
buttonPanel(frame, "ButtonPanel",
Location(where.Xcoord()+ButtonPanelLocn.Xcoord(),
where.Ycoord()+ButtonPanelLocn.Ycoord()),
ButtonPanelShape),
startButton("Start", StartButtonLocn, StartButtonShape),
stopButton("Stop", StopButtonLocn, StopButtonShape),
canvas(frame, "StopWatchCanvas",
Location(where.Xcoord()+CanvasLocn.Xcoord(),
where.Ycoord()+CanvasLocn.Ycoord()),
CanvasShape),
clockDisplay("0", ClockDisplayLocn)
{
// constructor body
buttonPanel.Add(stopButton);
buttonPanel.Add(startButton);
canvas.Clear();
clockCount.ConnectTo(clockDisplay);
clockDisplay.DisplayIn(canvas);
clock.Start();
}

StopWatch Constructor (3 of 3)
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

```cpp
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 = head; 
for (int i = 0; i < length - 1; i++) 
{  
next = node->next();  
canvas.drawLine(next->Contents(), node->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* next);
    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.
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();

```cpp
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

```cpp
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:**

```cpp
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

```cpp
class Location {   // Extension 1
private:
    int currentX, currentY;
    
    const methods won't modify currentX/Y.

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
};
```
Constant Methods in the Location Class

```cpp
// Implementation follows
const 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; }
```

---

Explain why statements are "OK" or "Error"

```cpp
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

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

  // other will not be changed
  // this will not be changed
```

```cpp
};
```

---


### const Parameters (cont’d)

Implementation must match interface declaration:

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

---

### Using const Parameters

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

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

---

### const Pointers

```c
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

```c
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 C1166: 1-value specifies const object

### const Pointer Parameters

```c
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
```
Managing Dynamically Allocated Storage

Goals:
1. Avoid unintended early deletion
2. Avoid memory leaks

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

Unintended Early Deletion Caused by Copying

```cpp
class Display { 
    public:
        
            void Show(Message msg);  // Pass by copy
        
    };

Message greeting ("Hello World");
Display display;

display.Show(greeting);  // Pass by copy
```

Shared Data: What happens in Show(greeting)

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

```
(greeting) (msg)

message originalMsg

message copiedMsg

"Hello World"
```

What happens when “msg” goes out of scope at end of display.Show(greeting) call?
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(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);
}

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

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

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

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 World");
Message sourceMsg("MacroSoft, Inc.");
```

What is the effect of an assignment statement like:

```
targetMsg = sourceMsg;  // assignment
```
The default means for assigning an object is a bit-wise 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 = strcpy(source.message); // copy new string
    return *this;
}

Overloading the assignment operator

lhsObject = rhsObject;

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

Right hand side object (rhsObject)

classNames. Classname::operator=(const ClassNames &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

<table>
<thead>
<tr>
<th>Normal Invocation</th>
<th>Inline Expansion</th>
</tr>
</thead>
<tbody>
<tr>
<td>STORE deltaY, 10</td>
<td>STORE deltaY, 10</td>
</tr>
<tr>
<td>PUSH AX</td>
<td>MOVE AX, quad.currentY</td>
</tr>
<tr>
<td>PUSH BX</td>
<td>ADD AX, deltaY</td>
</tr>
<tr>
<td>PUSH deltaY</td>
<td>STORE AX, quad.currentY</td>
</tr>
<tr>
<td>CALL UP_CODE</td>
<td></td>
</tr>
<tr>
<td>POP BX</td>
<td></td>
</tr>
<tr>
<td>POP AX</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 4

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;}
};

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

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"

```java
class listener {
  public:
    void locationMoved(Location * loc); // called when Location moves
};

class Location {
  int currentX, currentY;
  listener "listener; //same before public:
    void addListener(listener & listener) { this->listener = &listener; }
    void Move(int x, int y);
};

class LocationMove { Location * loc; void Move(int x, int y) {
  currentX = x; currentY = y;
  if (listener->locationMoved(this));
}
```

Notify listener that this Location has moved

Using "this" (cont'd)

- Return a reference to the object itself

```java
class PolyShape {
  public:
    // same before public:
    PolyShape & up(int n);
    PolyShape & down(int n);
    PolyShape & left(int n);
    PolyShape & right(int n);
    PolyShape & 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

```java
class Rectangle {
    private:
        Location upperleft;
        // Other Location ...
        Shape area;
        void AdjustCorners(); // private method
    public:
        Location corner, Shape shape;
        void Move(int delta); // ... Other public methods
};
```

Example (cont’d)

```java
void Rectangle::AdjustCorners() {
    upright = 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::Move(int delta) {
    upperleft = Location(upperleft.xCoord() + delta, upperleft.yCoord() + delta);
    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

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

Static Variable Example (2)

// Default color is red
Rectangle rect1(..),rect2(..); Canvas can(..);

rect1.Draw(can); rect2.Draw(can);//red

rect2.setColor(Color(0,0,200)); // blue
rect1.Draw(can); rect2.Draw(can);

Both are blue!
### Color Class

```
C S 2704

```

### 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

```
C S 2704

```
Overview: RectangleManager.h

- Constructs all rectangles of *same shape*
  `RectangleManager(size) CreateRectangleAt(location)`
- Finds rectangle containing certain \((x,y)\)
  `GetRectangleAt(x, y)`
- Redraws all rectangles
  `Draw(canvas)`
- Destructs all rectangles
  `~RectangleManager()`

RectangleManager.h

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

Using RectangleManager (1)
Using RectangleManager (2)

```cpp
Frame window("Rectangle Test", Location(100, 100), Shape(300, 300));
Canvas canvas(window, "Rectangle Area", Location(30, 30), Shape(280, 280));
RectangleManager squareManager(Shape(50, 50));
void OnStart() {
    squareManager.CreateRectangleAt(Location(10, 10)); // create rectangles
    squareManager.CreateRectangleAt(Location(70, 10));
    squareManager.CreateRectangleAt(Location(30, 100));
    squareManager.CreateRectangleAt(Location(20, 150));
    OnPaint();
}
void OnPaint(char* buttonName) {
    }
}
```

Using RectangleManager (3)

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

RectangleManager.cpp

```cpp
RectangleManager::RectangleManager(Shape shapeForAll) {
    commonShape = shapeForAll;
    rects = new Rectangle*[MaxRecs];
}
void RectangleManager::CreateRectangleAtLocation(int x, int y) {
    if (numRecs < MaxRecs) {
        rects[numRecs++] = new RectangleShape(commonShape);
    }
    RectangleManager::~RectangleManager() {
        for (int i = 0; i < numRecs; i++)
            delete rects[i];
    }
```
RectangleManager.cpp (cont’d)

```cpp
Rectangle3 * RectangleManager::GetRectangle(int x, int y) {
    for (int i = 0; i < numRecs; i++) {
        if (x >= rect[i]->upperLeft.xcoord() &&
            y <= rect[i]->upperLeft.ycoord() &&
            y <= rect[i]->lowerRight.ycoord() &&
            y <= rect[i]->lowerRight.ycoord()) {
            return rect[i];
        }
        return (Rectangle3*)NULL; // null
    }
}
```

Accessing private data directly (vs. method invocation) is more efficient

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.
### Interface

```cpp
class Circle {
private:
    float radius;
    Location center;
public:
    Circle();
    Circle(float radiusToSet, Location centerToBeSet);
    void Reset(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

```cpp
class MovingCircle {
private:
    Circle circle;
    float direction;
    float speed;
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

```cpp
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;  // Dynamic allocation
  
  public:
  
  CircleList();
  CircleList(CircleList* initialHead);
  CircleList(const CircleList& other);
  void operator=(const CircleList& source);
  void insert(CircleList* node);
  Boolean remove(CircleList* node);
  
  ~CircleList();  // Destructor must free dynamic memory
  
};

Copy constructor and = operator are needed to copy whole list

Copy constructor and assignment operator not needed

Destructor might not do anything

Class ShapeManager

{ 
  private:
  
  CircleList circleList;  // No dynamic allocation
  RectangleList rectangleList;
  
  public:
  
  ShapeManager();
  void MoveAllCircles();
  
  ~ShapeManager();  // Destructor might not do anything
  
};