Normal Procedure Calls

- Procedure invocation has overhead
  1. Save registers to memory
  2. Allocate activation record on stack
  3. Store parameters
  4. Enter procedure
  5. Remove activation record
  6. Restore registers

Inline Methods

- Inlining replaces procedure call by code of function
- Avoids some overhead of call
- Causes executable to be larger
- More efficient for small methods

Default Inlining

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

Explicit Inlining

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

- Function definition must occur in the header file!
- Inline declaration must appear before first use
Tradeoffs of Inlining

- Default inline exposes implementation
- All code using 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

Inlining and Style

- Want class declaration to present interface
- Place no more information in class declaration than necessary
  - Hides implementation details from reader
  - Makes changing implementation easier

“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” Example

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

Using “this” Example (2)

```cpp
// Observer of Location objects
class Listener {
    public:
        void LocationMoved(Location * loc);
    
    void Location::Move(int x, int y) {
        currentX = x; currentY = y;
        if (listener) listener->LocationMoved(this);
    }
}
```

Using “this” to Return Reference

```cpp
Shape& operator=(const Shape& s) {
    if (this != &s) {
        height = s.height;
        width = s.width;
    }
    return *this;
}
```
Private Methods

- Allows for hidden subfunctions
- Can help implement methods and yet hide details of implementation
- Sometimes useful to hide default or copy constructors

Private Helper Methods

class Table {
public:
    ... 
    // Interface doesn't require knowing how table
    // is implemented
    bool search(const Key k) const;
private:
    // Helper can use "knowledge" of
    // implementation to be more efficient
    TreeNode* search_help(TreeNode* t, const Key k) const;
};

Private Constructors

- You may not want certain classes to have default objects, or allow object copying
- Simply not writing constructor is not sufficient. Why?
- Enforce use restriction by defining the constructor, but making constructor private

Private Copy Constructor

- Each GasSensor object is linked to 1 hardware sensor
  class GasSensor {
    public:
    ... 
    private:
    ... GasSensor(const GasSensor& gs) {} 
    
    // Cannot "copy" the hardware

(Static) Class Variables

- Variables that belong to class not one object
- Share data among all instances
- Helps reduce need for global variables
- Must be initialized outside of constructor(s)

Example Class Variable

class Rectangle {
public:
    Rectangle (Location corner, Shape shape);
    void setColor(Color newColor);
    ... 
private:
    ... static Color rectangleColor; // class variable
};
Example Class Variable (2)

```cpp
// Initialize class variable to a shade of red
Color Rectangle::rectangleColor = Color(200,0,0);

// change the color for all Rectangle objects
void Rectangle::setColor(Color color) {
    rectangleColor = color;
}
```

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
- Can be useful for
  - Efficiency
  - Security

Friend Class Example (1)

```cpp
class Rectangle3 {  
public:
    // no public constructor
    ~Rectangle3();
private:
    ...  
    // private constructor
    Rectangle3 (Location corner, Shape shape);  
    // RectangleManager has access to private
    friend class RectangleManager;
};
```

Friend Class Example (2)

```cpp
class RectangleManager {  
public:
    RectangleManager (Shape shape) {  
        // create rectangles
        Rectangles += GetRectangleAt(0, y);  
    }
private:
    ...  
    // no managed rectangles
    int numRects;
};
```

Friend Class Example (3)

```cpp
RectangleManager::RectangleManager (Shape shape) {  
    numRects = 0;  
    // construct an int with variable name h value
    commonShape = shape;
    rects = new Rectangle3[numRects];  
    // Only RectangleManager objects can create rectangle objects
    void RectangleManager::CreateRectangleAt (Location loc) {  
        if (numRects < numRects++) {  
            rects[numRects] = new Rectangle3(loc, commonShape);  
        }
    }
}
```

Friend Class Example (4)

```cpp
More efficient to directly manipulate objects

```cpp
void Rectangle3 = RectangleManager::GetRectangle(int x, int y) {  
    for (int i = 0; i < numRects; i++) {  
        if ((x >= rects[i].GetX() && y >= rects[i].GetY()) &&  
            (x <= rects[i].GetRight() && y <= rects[i].GetBottom())) {  
            return rects[i];
        }
    }
    return (Rectangle3)*0;  // null
}
```
Problems with Friends

- A class and its friends are “joined at the implementation”
- Change to class that declares friends may require change to friends
- Generally want to avoid lowering protections, but may be good reasons to do so