Abstraction
Type abstraction for type `double`.
Structure abstraction

```c
struct Rectangle {
    double length;
    double width;
};

myRectangle.length = 2.0;
```
Statement abstraction

load length, r1
load width, r2
add r1, r2, r3
mul 2.0, r3, r3
store r3, perim

perim = 2.0 * (length + width);
double perimeter(Rectangle r) {
    return 2.0 * (r.length + r.width);
}

cout << perimeter(myRectangle);

cout << perimeter(yourRectangle);
int gcd(int m, int n) {
    if (0 == n) {
        return m;
    } else {
        return gcd(n, m % n);
    }
}

temp = gcd(num, denom);
**Design Patterns for Data Structures**

**Figure 1.5**

**Class abstraction**

```c
struct Rectangle {
    double length;
    double width;
};

double area(Rectangle r);

double perimeter(Rectangle r);
```

```c
class Rectangle {
private
    double length;
    double width;
public
    double area();
    double perimeter();
};
```

<table>
<thead>
<tr>
<th><strong>Rectangle</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>– length: double</td>
</tr>
<tr>
<td>– width: double</td>
</tr>
<tr>
<td>+ area( ): double</td>
</tr>
<tr>
<td>+ perimeter( ): double</td>
</tr>
</tbody>
</table>
OO terminology

- *class* corresponds to *type*
- *object* corresponds to *variable*
- *method* corresponds to *procedure or function*
<table>
<thead>
<tr>
<th>UML terminology</th>
<th>C++ terminology</th>
</tr>
</thead>
<tbody>
<tr>
<td>class</td>
<td>class</td>
</tr>
<tr>
<td>superclass</td>
<td>base class</td>
</tr>
<tr>
<td>subclass</td>
<td>derived class</td>
</tr>
<tr>
<td>attribute</td>
<td>data member</td>
</tr>
<tr>
<td>operation / method</td>
<td>member function</td>
</tr>
<tr>
<td>visibility</td>
<td>access specifier</td>
</tr>
<tr>
<td>parameterized class</td>
<td>template</td>
</tr>
<tr>
<td>abstract</td>
<td>pure virtual</td>
</tr>
</tbody>
</table>
class Rectangle {
private:
    double length;
    double width;
public:
    double area();
    double perimeter();
};
class Rectangle {
private:
    double length;
    double width;
public:
    double area();
    double perimeter();
};

double Rectangle::perimeter() {
    return 2.0 * (length + width);
}
```cpp
class Rectangle {
private:
    double length;
    double width;
public:
    double area();
    double perimeter();
};

double Rectangle::perimeter() {
    return 2.0 * (length + width);
}

Rectangle myRectangle;
```
```cpp
class Rectangle {
  private:
    double length;
    double width;
  public:
    double area();
    double perimeter();
};

double Rectangle::perimeter() {
  return 2.0 * (length + width);
}

Rectangle myRectangle;

cout << myRectangle.perimeter();
```
Behavior abstraction

- Abstract Shape
- 

(a) Behavior abstraction for shapes rendered geometrically.
(b) Behavior abstraction for shapes rendered with the UML symbol for inheritance.
Design Patterns for Data Structures

Figure 1.7

Abstract Shape

(a) Behavior abstraction for shapes rendered geometrically.

(b) Behavior abstraction for shapes rendered with the UML symbol for inheritance.
The abstract shape class

```cpp
#ifndef AShape_hpp
#define AShape_hpp

#include <iostream> // ostream.
using namespace std;

class AShape {
public:
    virtual ~AShape() = default;
    // Virtual destructor necessary for subclassing.

    virtual double area() = 0;
    // Post: The area of this shape is returned.

    virtual double perimeter() = 0;
    // Post: The perimeter of this shape is returned.

    virtual void scale(double factor) = 0;
    // Pre: factor > 0.0
    // Post: This shape's dimensions are multiplied by factor.

    virtual void display() = 0;
    // Post: This shape's name and dimensions are printed to cout.

    virtual void promptAndSetDimensions() = 0;
    // Post: This shape's dimensions are prompted and set.

};
#endif
```

Figure 1.8

The abstract shape class

<table>
<thead>
<tr>
<th>AShape</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ area( ): double</td>
</tr>
<tr>
<td>+ perimeter( ): double</td>
</tr>
<tr>
<td>+ scale(factor: double)</td>
</tr>
<tr>
<td>+ display( )</td>
</tr>
<tr>
<td>+ promptAndSetDimensions( )</td>
</tr>
</tbody>
</table>
virtual double perimeter() = 0;
// Post: The perimeter of this shape is returned.

virtual void scale(double factor) = 0;
// Pre: factor > 0.0
// Post: This shape’s dimensions are multiplied by factor.

virtual void display(ostream &os) = 0;
// Post: This shape’s name and dimensions are printed to os.

virtual void promptAndSetDimensions() = 0;
// Post: This shape’s dimensions are prompted and set.
// No dimension is negative.

};
class Rectangle : public AShape {
private:
    double _length;
    double _width;

public:
    explicit Rectangle(double length = 0.0, double width = 0.0);
    // Pre: length >= 0.0 and width >= 0.0.
    // Post: This rectangle is initialized with
    // length length and width width.
    double area() override;
    double perimeter() override;
    void scale(double factor) override;
    void display() override;
    void promptAndSetDimensions() override;
};
Figure 1.10

```
<table>
<thead>
<tr>
<th>AShape</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ area( ) : double</td>
</tr>
<tr>
<td>+ perimeter( ) : double</td>
</tr>
<tr>
<td>+ scale(factor : double)</td>
</tr>
<tr>
<td>+ display( )</td>
</tr>
<tr>
<td>+ promptAndSetDimensions( )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rectangle</th>
</tr>
</thead>
<tbody>
<tr>
<td>_length : double</td>
</tr>
<tr>
<td>_width : double</td>
</tr>
<tr>
<td>+ Rectangle(length : double, width : double)</td>
</tr>
<tr>
<td>+ area( ) : double</td>
</tr>
<tr>
<td>+ perimeter( ) : double</td>
</tr>
<tr>
<td>+ scale(factor : double)</td>
</tr>
<tr>
<td>+ display( )</td>
</tr>
<tr>
<td>+ promptAndSetDimensions( )</td>
</tr>
</tbody>
</table>
```
```cpp
#include "Rectangle.hpp"
#include "Utilities.hpp"

Rectangle::Rectangle(double length, double width) {
    if (length < 0.0 || width < 0.0) {
        cerr << "Rectangle precondition violated: "
             << "length and width cannot be negative." << endl;
        throw -1;
    }
    _length = length;
    _width = width;
}

double Rectangle::area() {
    return _length * _width;
}

double Rectangle::perimeter() {
    return 2.0 *(_length + _width);
}
void Rectangle::scale(double factor) {
    cerr << "scale: Exercise for the student."
         << endl;
    throw -1;
}

void Rectangle::display(ostream &os) {
    os << "Rectangle" " << endl;
    os << "Length: " _length " << endl;
    os << "Width: " _width " << endl;
}

void Rectangle::promptAndSetDimensions() {
    _length = promptDoubleGE("Length?", 0.0);
    _width = promptDoubleGE("Width?", 0.0);
}
Component diagram

- AShape.hpp
- Rectangle.hpp
- Rectangle.cpp
The C++ `try/catch/throw` statement

```cpp
Rectangle::Rectangle(double length, double width) {
    try {
        if (length < 0.0 || width < 0.0) {
            throw -1;
        }
        _length = length;
        _width = width;
    }
    catch(int a) {
        cerr << "Rectangle precondition violated: "
            << "length and width cannot be negative." << endl;
        exit(1);
    }
}
```
Class diagram for abstract and concrete shapes

- **AShape**
  - `area()`: double
  - `perimeter()`: double
  - `scale(factor: double)`
  - `display()`
  - `promptAndSetDimensions()`

- **Line**
  - `_length`: double
  - `Line(length: double)`

- **Circle**
  - `_radius`: double
  - `Circle(radius: double)`

- **NullShape**

- **Rectangle**
  - `_length`: double
  - `_width`: double
  - `Rectangle(length: double, width: double)`

- **RightTriangle**
  - `_base`: double
  - `_height`: double
  - `RightTriangle(base: double, height: double)`
Demo dp4ds Shape project
void initialize(shared_ptr<AShape> shapes[], int cap);
// Pre: shapes[0..cap - 1] is allocated.
// Post: All shapes[0..cap - 1] are initialized to the null shape.

void promptLoop(shared_ptr<AShape> shapes[], int cap);
// Loop to prompt the user with the top-level main prompt.
// Post: User has selected the quit option.

void makeShape(shared_ptr<AShape> &sh);
// Prompts user for dimensions.
// Post: Original sh is deleted and new sh is created.

char shapeType();
// Prompts user for shape letter, lowercase or uppercase.
// Post: Uppercase character L, R, C, T, or M is returned.
void clearShape(shared_ptr<AShape> &sh);
// Post: sh is made the null shape.

void printArea(shared_ptr<AShape> sh);
// Post: sh’s area is printed to standard output.

void printPerimeter(shared_ptr<AShape> sh);
// Post: The perimeter of this sh is printed to standard output.

void scaleShape(shared_ptr<AShape> sh);
// Prompts user for scale factor.
// Post: sh’s dimensions are multiplied by the factor.

void displayShape(shared_ptr<AShape> sh);
// Post: sh’s name and dimensions are printed to standard output.
Visual depiction of parameters

(a) Visual depiction of parameter $sh$.

(b) Visual depiction of parameter $shapes$.

$shapes[0]$ 

$shapes[1]$ 

$shapes[2]$ 

$shapes[3]$ 

$shapes[4]$
The ShapeMain program

```cpp
int main() {
    const int NUM_SHAPES = 5;
    shared_ptr<AShape> shapes[NUM_SHAPES];
    initialize(shapes, NUM_SHAPES);
    promptLoop(shapes, NUM_SHAPES);
    return EXIT_SUCCESS;
}
```
void initialize(shared_ptr<AShape> shapes[], int cap) {
    for (int i = 0; i < cap; i++) {
        shapes[i] = make_shared<NullShape>();
    }
}

Figure 1.19

Design Patterns for Data Structures

#include <cstdlib> // EXIT_SUCCESS.
#include <cctype> // toupper.
#include "Utilities.hpp" // promptIntBetween, promptDoubleGE.
#include "AShape.hpp"
#include "Line.hpp"
#include "Rectangle.hpp"
#include "Circle.hpp"
#include "RightTriangle.hpp"
#include "NullShape.hpp"
#include "ShapeMain.hpp"
#include "MysteryShape.hpp"

int main() {
const int NUM_SHAPES = 5;
shared_ptr<AShape> shapes[NUM_SHAPES];
initialize(shapes, NUM_SHAPES);
promptLoop(shapes, NUM_SHAPES);
return EXIT_SUCCESS;

Figure 1.19
ShapeMain.cpp

The main theme in this program is the use of abstract and concrete classes. Function initialize is shown here. The program continues in the next figure.

Derived derived;
where Derived inherits from Base as follows.

class Derived : public Base

The cardinal rule of object-oriented assignment says that you can assign the specific to the general, but you cannot assign the general to the specific. That is, the assignment base = derived;
is legal, but the assignment derived = base;
is not legal.

The class that is general contains items that are common to all the specific classes derived from it. A specific class inherits all those items and may contain additional
When the `make_shared<>()` function executes, it

- allocates storage from the heap for the attributes of the object, and
- returns a pointer to the newly allocated storage.
```cpp
void promptLoop(shared_ptr<AShape> shapes[], int cap) {
    char response = '\0';
    do {
        cout << "\nThere are [0.." << cap - 1 << "] shapes." << endl;
        cout << "(m)ake (c)lear (a)rea (p)erimeter (s)cale (d)isplay " << "(q)uit: ";
        cin >> response;
        switch (toupper(response)) {
            case 'M':
                makeShape(shapes[promptIntBetween("Which shape?", 0, cap-1)]); break;
            case 'C':
                clearShape(shapes[promptIntBetween("Which shape?", 0, cap-1)]); break;
            case 'A':
                printArea(shapes[promptIntBetween("Which shape?", 0, cap-1)]); break;
            default:
                cout << "Illegal command." << endl;
                break;
        }
    } while (toupper(response) != 'Q');

Figure 1.20
ShapeMain.cpp (continued). Function promptLoop from the main program listing. The main program listing continues in the next figure.
```

Because you can assign a specific object to a general object, you can endow a general object dynamically (that is, during execution of the program) with more items than were included in its original specification statically (that is, during compilation of the program). You can give, but you cannot take away. During execution, an object can get more specific than its static declaration, but it cannot get more general.

In function initialize(), formal parameter shapes[] is an array of pointers to AShape, the base class, which is general. An example of a assigning to the specific is when initialize() executes the statement:

case 'P':
    printPerimeter(shapes[promptIntBetween("Which shape?", 0, cap-1)]);
    break;
  case 'S':
    scaleShape(shapes[promptIntBetween("Which shape?", 0, cap-1)]);
    break;
  case 'D':
    displayShape(shapes[promptIntBetween("Which shape?", 0, cap-1)]);
    break;
  case 'Q':
    break;
default:
    cout << "\nIllegal command.\n" << endl;
    break;
  }
} while (toupper(response) != 'Q');
void makeShape(shared_ptr<AShape> &sh) {
    switch (shapeType()) {
    case 'L':
        sh = make_shared<Line>();
        break;
    case 'R':
        sh = make_shared<Rectangle>();
        break;
    case 'C':
        sh = make_shared<Circle>();
        break;
    case 'T':
        sh = make_shared<RightTriangle>();
        break;
    case 'M':
        //Exercise for the student.
        break;
    default:
        break;
    }
    sh->promptAndSetDimensions();
}
```cpp
char shapeType() {
    char ch;
    cout << "(l)ine (r)ectangle (c)ircle right(t)riangle (m)ystery: ";
    cin >> ch;
    ch = toupper(ch);
    while (ch != 'L' && ch != 'R' && ch != 'C' && ch != 'T' && ch != 'M') {
        cout << "Must be l, r, c, t, or m. Which type? ";
        cin >> ch;
        ch = toupper(ch);
    }
    return ch;
}
```
When the `make_shared<>` operator executes on an object, it

- allocates storage from the heap for the attributes of the object,
- calls the constructor based on the number and types of parameters in the parameter list, and
- returns a pointer to the newly allocated storage.
void clearShape(shared_ptr<AShape> &sh) {
    sh = make_shared<NullShape>();
}

void printArea(shared_ptr<AShape> sh) {
    cout << "\nArea: " << sh->area() << endl;
}

void printPerimeter(shared_ptr<AShape> sh) {
    cout << "\nPerimeter: " << sh->perimeter() << endl;
}

void scaleShape(shared_ptr<AShape> sh) {
    sh->scale(promptDoubleGE("Scale factor?", 0.0));
}

void displayShape(shared_ptr<AShape> sh) {
    cout << endl;
    sh->display();
}
The abstraction process

Data
- Type abstraction → Structure abstraction → Rectangles
- Type abstraction

Control
- Statement abstraction → Procedure abstraction → Class abstraction
- Statement abstraction

Data
- Type abstraction → Structure abstraction → Circles
- Type abstraction

Control
- Statement abstraction → Procedure abstraction → Class abstraction
- Statement abstraction

Behavior abstraction