Chapter 6: Part 2

Generalization:
Inheritance & Polymorphism

Steps to Unlocking the Power of Inheritance

Pure Virtual Methods
Virtual Methods
Type Casting
Type Casting

- Allows flexible structures to be built
- **Idea:**
  Treat a derived class object as if it were a base class object
- **Syntax:**
  BaseClassName (derivedClassObject)
  or
  (BaseClassName) derivedClassObject

This syntax cannot be used with pointers.
Type Casting Using Pointers to an Object

TextBox display(Location(100,100), Shape(75, 50));
Number* number = new Number(100);
DisplayableNumber* dnp;
Number->Next();

Type Cast

dnp = (DisplayableNumber*) number;

dnp->Showln(display);
dnp->Show();

We may only call methods from DisplayableNumber on dnp, even though it's really an object of type Number!

Type Casting Using References

TextBox display(Location(100,100), Shape(75, 50));
Number number(100);
number.Next();

DisplayableNumber&
    displayable = (DisplayableNumber&) number;

displayable.Showln(display);
displayable.Show();
Type Casting Errors

DisplayableNumber* numPtr;
Number *count1 = new Number(100);
Number  count2(200);
umPtr = (DisplayableNumber*) count1;
DisplayableNumber& numRef = (DisplayableNumber)count2

numPtr->Next();
numRef.Next();

Wrong! DisplayableNumber doesn’t have a Next() method!

Why Type Casting?

- Allows us to treat a collection of objects uniformly by viewing them as their base class.
- Example 1: Shapes from Project 1
- Example 2: NumberPanel class example
Implicit vs. Explicit Type Casting

**Explicit**

```cpp
Number *number;
DisplayableNumber *displayable = (DisplayableNumber*)number
```

**Implicit**

```cpp
Number *number;
DisplayableNumber *displayable = number;
```

Avoid implicit typecasting.
Say what you mean in your code!

Implicit vs. Explicit Type Casting In Parameter Passing

**Given the following method:**

```
NumberPanel::Add(DisplayableNumber * dn);
```

**Call with Explicit Type Cast**

```cpp
NumberPanel panel;
Number *n = new Number(100);
panel.Add((DisplayableNumber*)n);
```

**Call with Implicit Type Cast**

```cpp
NumberPanel panel;
Number *c = new Number(100);
panel.Add(c);
```

Other developers may not realize that Add takes a DisplayableNumber.
Widening vs. Narrowing

- **Widening**
  - Type casting from a derived class to a base class.
  - *Always Safe!*
  - Can be checked by the compiler.

- **Narrowing**
  - Type casting from a base class to a derived class.
  - Safety depends on the programmer.

Widening/Narrowing Example

```cpp
DisplayableNumber *DNptr;
Number *number = new Number(100);
Cycler *cycler;

DNptr = (DisplayableNumber*)number;  // safe; it widens

Nptr = (Cycler*)DNptr;  // oops!

cycler->Next();  // who knows what this will do!
```
Suppose

GameCircle * circ = new GameCircle(Location(100,100));
Shape * shape = (Shape*)circ;
shape.Draw();                      // which Draw() is called?

class Clock {
    private:
        Number* number;   // Connect(ed)To a Number
        Cycler* cycler;   // Connect(ed)To a Cycler
    ...
    public:
    ...
        ConnectTo(Number& cnt);
        ConnectTo(Cycler& cnt);
    ...
};
Revised Clock Class: (Continued)

```cpp
void Clock::ConnectTo(Number& cnt) { number = &cnt; };
void Clock::ConnectTo(Cycler& cnt) { cycler = &cnt; };
void Clock::Notify()
{ if (number) number->Next();
  else if (cycler) cycler->Next();
}
```

Repetitive Code in the Clock Class

class Clock {
private:
  Number* number; //Connect(ed)To a Counter
  Cycler* cycler;  //Connect(ed)To a Cycler
  BiCycler* biCycler; //Connect(ed)To a BiCycler
  UpCounter* upCounter; //Connect(ed)To a UpCounter
  DownCounter* downCounter; //Connect(ed)To a DownCounter
  BatchCounter* batchCounter; //Connect(ed)To a BatchCounter
  JumpCounter* jumpCounter; //Connect(ed)To a JumpCounter
  SwitchCounter* switchCounter; //Connect(ed)To a SwitchCounter
  ...

YUCK! We need a better way!
Declaring a Virtual Method

class DisplayableNumber {
    private:
    ...
    public:
    ...
        virtual void Next();   // virtual Next method
    ...
};

// in implementation file
void DisplayableNumber::Next() {}   // default definition

Redefining the Clock Class

class Clock {
    private:
        DisplayableNumber *number;   // only refer to base class

    public:
    ...
        void ConnectTo(DisplayableNumber* dn);
        void ConnectTo(DisplayableNumber& dn);
        void Notify();
    };
Redefining the Clock Class (Continued)

// in implementation file
void Clock::ConnectTo (DisplayableNumber* dn)
{ number = dn; }

void Clock::ConnectTo (DisplayableNumber& dn)
{ number = &dn; }

void Clock::Notify()
{ number->Next(); // invokes derived class method
  number->Show(); // invokes base class method
}

Using the Revised Clock Class

Clock oneMinute(60*1000),
    oneSecond(1000);
Number minutes(0);
Cycler seconds(60);

oneMinute.ConnectTo( (DisplayableNumber&) minutes);
oneSecond.ConnectTo( (DisplayableNumber&) seconds);

oneMinute.Start();
oneSecond.Start();
Example of Run-Time Binding

class Clock {
    private:
        DisplayableNumber *number; // only refer to base class
    public:
        void Notify();
    }

    void Clock::Notify() {
        number->Next(); // invokes derived class method
        ...
    }

    Binding of Next() in Notify() is done at compile-time by an invisible pointer since actual type isn’t known until execution!

Binding

- Selection of code for a function invocation
- Static, compile time
  - efficient
- Dynamic, run time
  - flexible
  - default in many o-o languages
    - Java, Smalltalk
  - Must be explicitly programmed in C++
Pure Virtual Methods

Definition of “Abstract” Base Class

```cpp
class AbstractDisplayableNumber {
    ... public:
    virtual void Next() = 0;  // pure virtual method
    ...};
```

Definition of a “Concrete” Derived Class

```cpp
class ConcreteCounter : public AbstractDisplayableNumber {
    ...};
void ConcreteCounter::Next() {...some implementation ... }
```

Examples of Using Abstract Classes

```cpp
AbstractDisplayableNumber *adn;  // ok - pointer to abstract
   // class
ConcreteCounter*cc;  // ok - pointer to concrete class

// NO! - class is abstract
AbstractDisplayableNumber n(100);

ConcreteCounter c(100);  // ok - instance of concrete class

cc = &c;                  // ok - types are the same
adn = &c;                 // ok
and->Next();              // ok
```
Class Structure for Polymorphism

Typical Component Class Hierarchy
Example from Project 1

- No default Draw() for Shape class
- Want ShapeManager to call Draw() on its Shape objects.
- What do we do?

Use of Polymorphism: Component Example

class ComponentManager {
    private:
        Component * member[10];
        int        number;
    ...
    public:
        ComponentManager() : number(0) {}  
        void Add(Component * cmp) { member[number++] = cmp; }
        Component* Inside(int x, int y)  
            {for(int next = 0;  next<number; next++)
                if (member[next]->inside(x,y))
                    return member[next];
            return NULL;
            }
    ...
};
**Problem**

**Problem:** The representation of the octal number in the TextBox should appear in an octal number representation: preceded by a zero and without the digits 8 or 9.

**Solution:** Override the base class method `Show`

**Implication:** The TextBox variable must be moved to the protected section

**Design decisions revealed:**
- The DisplayableObject’s value is displayed in a TextBox,
- The TextBox is accessed via a pointer,
- The name of the pointer is `textBox`, and
- The TextBox has a `SetText` method.
Solution:

Factor the base class to separate:
- Formatting the character string to be displayed, and
- Displaying the string in the TextBox.

Refactoring the DisplayableNumber Class

class DisplayableNumber {
    private:
        TextBox* textBox;
    protected:
        int value;
        virtual char* Format(); // produce string to display
        virtual int Parse(char* input); // convert user
                                        // input to value
    public:
        DisplayableNumber(int initValue = 0);
        void ShowIn(TextBox& p);
        void Show();
        void Reset();
        ~DisplayableNumber();
};
Refactoring the DisplayableNumber Class

char* DisplayableNumber::Format()
{
    char* asString = new char[10];
    ostrstream format(asString);
    format << value; // use decimal formatting
    return asString;
}

void DisplayableNumber::Show()
{
    if (textBox) textBox->SetText(Format());
}

int DisplayableNumber::Parse(char* input)
{
    int decimalValue;
    istrstream format(input);
    format >> decimalValue; // use decimal formatting
    return decimalValue;
}

void DisplayableNumber::Reset()
{
    if (textBox) value = Parse(textBox->GetText());
}
Defining the Octal Number Class

class OctalNumber : public Number {
    protected:
        char* Format(); // how to print an octal number
        int Parse(char* input) // how to read an octal number

    public:
        OctalNumber(int init);
        ~OctalNumber();
};

OctalNumber::OctalNumber (int init) : Number(init) {}
char* OctalNumber::Format()
{ char* asString = new char[10];
    ofstream format(asString);
    format << oct << value; // format value as octal
    return asString;
}
int OctalNumber::Parse(char* input)
{ int octalValue;
    istream is(input);
    is.flags(ios::oct); // set stream to read as octal
    is >> octalValue;
    return octalValue;
}
OctalNumber::~OctalNumber() {}