Introduction to Object Oriented Programming 2E Timothy A. Budd

Chapter 12

Implications of Inheritance

Idealization of *is-a* Relationship

- A TextWindow *is-a* Window
- Because TextWindow subclasses Window, all behavior of Windows present in instances of TextWindow
- Therefore, variable declared as Window should be able to hold value of type TextWindow

Unfortunately, practical programming language implementation issues complicate this idealized picture.

Memory Allocation – Stack and Heap Based

Generally, programming languages use two different techniques for allocation of memory:

Stack-based allocation: space requirement determined at compile time based on static types

- Memory allocation/release tied to procedure entry/exit
- Performed very efficiently

Heap-based allocation: space requirement determined at run-time based upon dynamic considerations

- Memory allocation/release not tied to procedure entry/exit
- Handled by user or by run-time library (garbage collection)
- Considered somewhat less efficient

Specific Example of Problem

```
class Window {
public:
    virtual void oops();
private:
    int height;
    int width;
};

class TextWindow : public Window {
public:
    virtual void oops();
private:
    char * contents;
    int cursorLocation;
};

Window x;
TextWindow y;
```

Memory Strategies

How much memory should be set aside for the variable `x` ?

Minimum Static Space Allocation: allocate space necessary for base class only

Maximum Static Space Allocation: allocate space necessary for largest subclass

Dynamic Space Allocation: allocate only space necessary for pointer

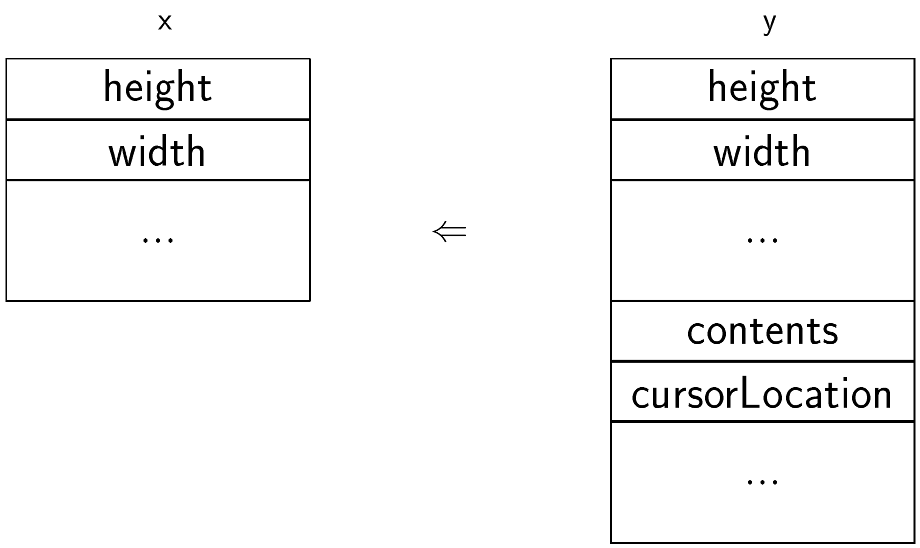
Minimum Static Space Allocation

- C++ uses minimum static space allocation approach
- Very efficient, but leads to some subtle difficulties
- What happens in the following assignment?

```
Window x;
TextWindow y;

x = y;
```

Assigning a Larger Value to a Smaller Box



The Slicing Problem

Problem: trying to take a large box and squeeze it into a small space

Clearly this won't work.

Thus, the extra fields are simply sliced off (i.e., a type conversion)

Question: Does it matter?

Answer: Only if somebody notices

Solution: Design the language to make it difficult to notice

Rules for Member Function Binding

Deciding what member function to execute complicated by slicing problem

variables declared normally: binding of member function name to function body based on static type (regardless whether the function virtual)

variables declared as references or pointers: binding of member function name to function body based on dynamic type if function virtual, on static type otherwise

Illustration

```
void Window::oops()
{    printf("Window oops\n"); }

void TextWindow::oops()
{    printf("TextWindow oops %d\n", cursorLocation); }

TextWindow x;

Window a;
Window * b;
TextWindow * c;

a = x;    a.oops();    // executes Window version
b = &x; b->oops();    // executes TextWindow or Window version
c = &x; c->oops();    // executes TextWindow version
```

Maximum Static Space Allocation

- Allocate maximum amount of space for any subclass instance
- Would nicely solve slicing problem
- Would often allocate unused space
- Maximum amount of space not known until all classes examined
- Hence, not used in practice

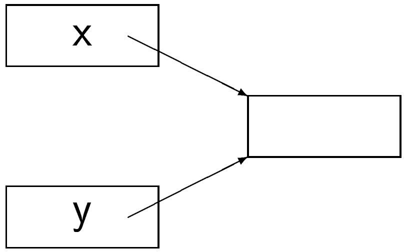
Dynamic Memory Allocation

- All objects are actually pointers
- Only enough space for pointer allocated at compile time
- Actual data storage allocated on heap at run-time
- Used in Java, Smalltalk, Object Pascal, and Objective-C
- Requires user to explicitly allocate new objects and, in some languages, explicitly free no longer used storage
- May also lead to pointer semantics for assignment and equality testing

Copy Semantics versus Pointer Semantics

Suppose a value is indirectly accessed through a pointer

When an assignment is performed (or equality test is made), is the quantity assigned (compared) simply the pointer or is it the actual value?



Problems with Pointer Semantics

- If `x` is assigned to `y` and then changes are made to `x`, are these changes reflected in `y`?
- If `x` is explicitly freed, what happens if the user tries to access memory through `y`?
- In C++, a programmer can make assignment (equality testing) mean anything they want
- Object Pascal uses pointer semantics, no built-in provision for copies
- Smalltalk and Objective-C use pointer semantics, have techniques for making copies
- Java uses pointer semantics for objects, has `clone` method for making copies

Equality Testing

- Equality testing compares values in some languages and compares pointers in others
  - Java uses pointer semantics for objects, value semantics for primitives
- Data fields can be compared bitwise, or (better) recursively using whatever equality means for data fields
- If objects can redefine equality, no guarantee it remains communtative

  e.g., $\mathsf{x} = \mathsf{y}$ might hold, but not $\mathsf{y} = \mathsf{x}$!

Changing Method Arguments

Equality testing is one place where it might useful to be able to change the types of method *arguments*

```
class Shape
{   ...
    public boolean equals (Shape s) { return false; }
}

class  Triangle extends Shape
{   ...
    public boolean equals (Triangle t) { ... }
}

class Square extends Shape
{   ...
    public boolean equals (Square sq) { ... }
}
```


Covariance and Contravariance

- An argument made more general is said to be covariant
- An argument made more restrictive is said to be contravariant
- Both can destroy the is-a relation, have tricky semantics
- Most languages forbid both

Example: Equality Testing in Java

```
class Shape
{   ...
    public boolean equals (Shape s) { return false; }
}

class  Triangle extends Shape
{   ...
    public boolean equals (Shape s)
    {   if (s instanceof Shape)
        {   Triangle t = (Triangle) s;
        // do comparison and return result
        }
    else
            return false;
    }
}
```

```
class Square extends Shape
{   ...
    public boolean equals (Shape s)
    {   Triangle t;
    try { t = (Triangle) s; } catch (ClassCastException e) { return false;}
    // do comparison and return result
    }
}
```