Preaching/Teaching Objects from Day One

Bruno R. Preiss
Electrical and Computer Engineering
University of Waterloo

http://www.pads.uwaterloo.ca/Bruno.Preiss/talks/1997/cccee/slides.ps

The villany you teach me I will execute,
and it shall go hard,
but I will better the instruction.

William Shakespeare:
The Merchant of Venice. Act iii. Sc. 1[**?**]

Of right and wrong he taught
Truths as refined as ever Athens heard;
And (strange to tell!)
he practised what he preached.

John Armstrong (1709-1779):
The Art of Preserving Health. Book iv. Line 301.[**?**]

Outline

1. Background
2. Thesis
3. Discussion
4. Summary

1. Background

E&CE 150
Introduction to Computing

*High level language syntax, procedures and functions, recursion, searching and sorting algorithms for linear arrays, arrays and record data types, pointer variables, linked lists, linked list data structures, binary trees, tree traversals, numerical error propagation, numerical root finding and integration algorithms[**?**].*

E&CE 250
Algorithms and Data Structures

*Algorithms and Data Structures emphasizes the following topics: data structures, abstract data types, recursive algorithms, algorithm analysis, sorting and searching, and problem-solving strategies[**?**].*

First Course in Programming—Evolution

Fortran

- expression and procedural abstraction

$\rightsquigarrow$ Pascal

- structured programming, modularity, info hiding

$\rightsquigarrow$ C

- industrial strength/relevance

First Course in Programming—Evolution

$\rightsquigarrow$ C++

- a "better" C
- OO of secondary importance

$\rightsquigarrow$ Java?

- significant number of CS schools moving to Java in the first course
- Java's "secure" execution environment has its pedagogic virtues

2. Thesis

Thesis

- benefits of object-oriented methodologies are profound

- it does not make sense to save OO methods for advanced courses

$\rightsquigarrow$ Teach/preach objects from day one!

- adopting OO methods requires a *paradigm shift*
  - often this is harder for the teacher than the student!

- significant amount of inertia must be overcome

Pedagogic Benefits of OO Methods

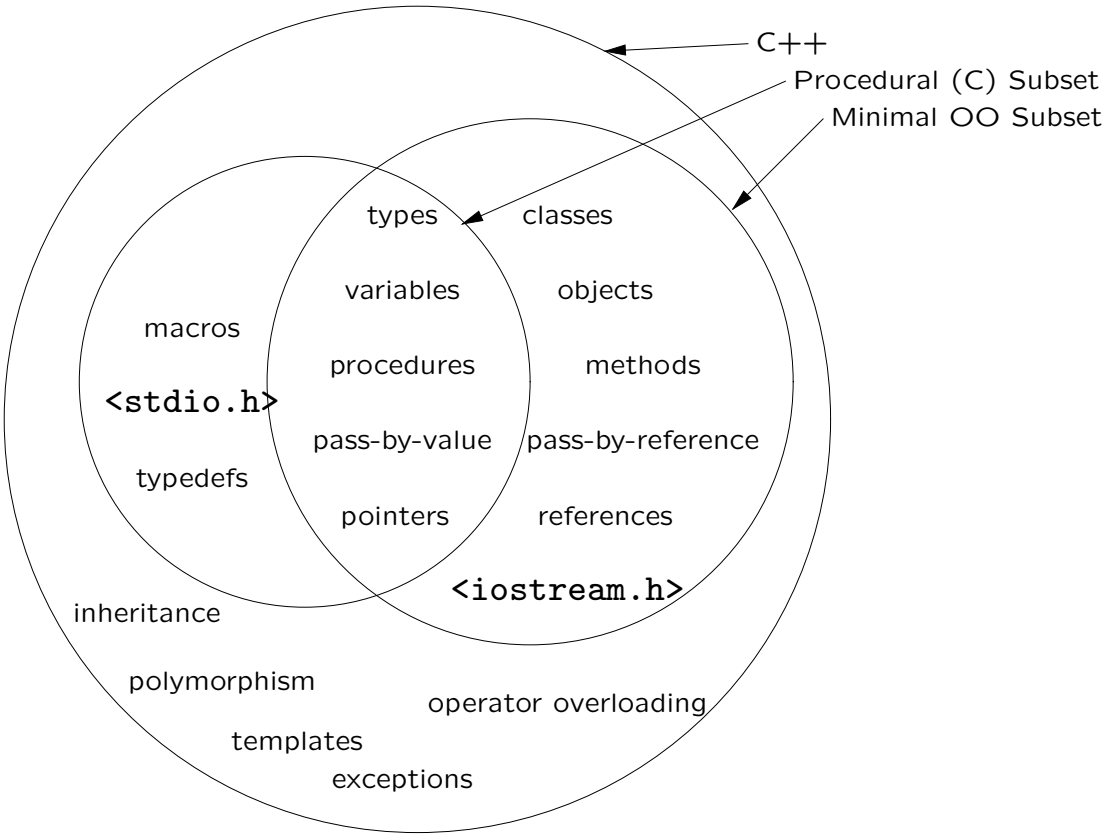
- *abstraction*
  - data abstraction + procedure abstraction $\rightsquigarrow$ ADTs

- *polymorphism*
  - express *commonalities*
    - data
    - code
    - interface

- *late (dynamic) binding*
  - algorithmic abstraction

3. Discussion

Choosing a Subset of C++



Model of the Machine

- students learn to program before they learn how a computer really works

- to do this they create a conceptual model of the machine

  $\rightsquigarrow$ a *virtual machine*

- claim: the basic C++ virtual machine is no more difficult to understand than, say, the Pascal virtual machine

The Second Course: Data Structures and Algorithms

- *Computing Curricula 1991* report of the ACM/IEEE-CS Joint Curriculum Task Force[**?**]

- knowledge units:

  AL1 Basic Data structures,

  AL2 Abstract Data Types,

  AL3 Recursive Algorithms,

  AL4 Complexity Analysis,

  AL6 Sorting and Searching, and

  AL8 Problem-Solving Strategies.

The Second Course: Engineering Design Component

- use *object-oriented design patterns*[**?**] extensively
  - singleton
  - container
  - iterator
  - adapter
  - visitor
- present all the data structures using a *single, unified, polymorphic class hierarchy*
- use *algorithmic abstraction* whenever possible

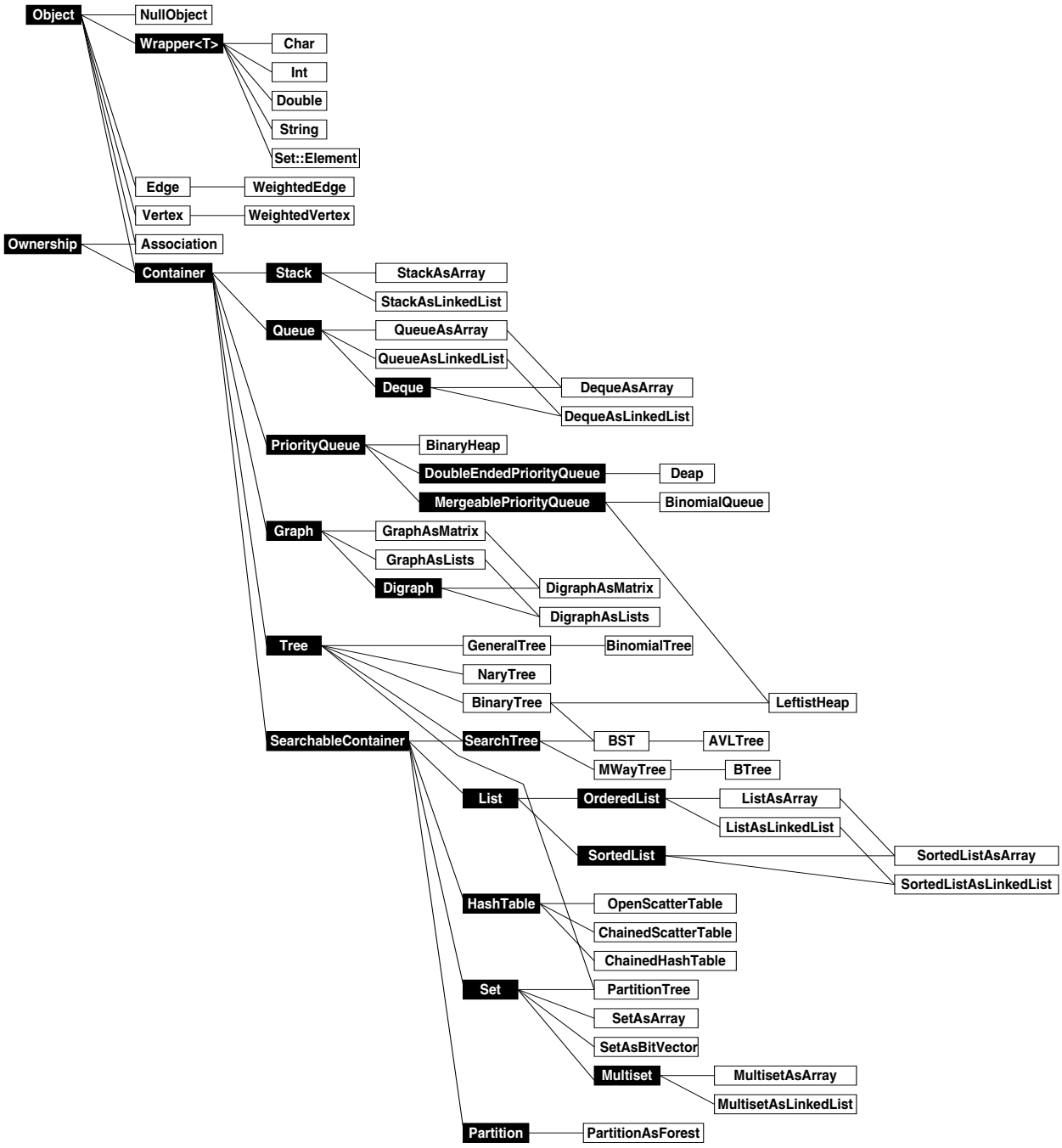
Design Patterns (Part 1)

```
class Object
{
public:
    virtual int Compare (Object const&) const = 0;
    virtual bool IsNull () const;
    virtual HashValue Hash () const = 0;
    virtual ostream& Put (ostream&) const = 0;
    // ...
};

class Container : public virtual Object
{
public:
    virtual unsigned int Count () const;
    virtual bool IsEmpty () const;
    virtual bool IsFull () const;
    virtual Iterator& NewIterator () const;
    virtual void Accept (Visitor&) const = 0;
    // ...
};

class Visitor
{
public:
    virtual void PreVisit (Object&) = 0;
    virtual void Visit (Object&) = 0;
    virtual void PostVisit (Object&) = 0;
    virtual bool IsDone () const = 0;
};
```

Design Patterns (Part 2)

```
class HashingVisitor : public Visitor
{
    HashValue value;
public:
    HashingVisitor ()
        { value = 0; }
    void Visit (Object& object)
        { value += object .Hash (); }
    HashValue Value () const
        { return value; }
};

HashValue Container::Hash () const
{
    HashingVisitor visitor;
    Accept (visitor);
    return visitor .Value ();
}
```

Class Hierarchy from [?]



Benefits of the Class Hierarchy

- data structures and algorithms are more than just a bag of tricks

- there are many commonalities
  - structural
  - algorithmic

- the hierarchy makes the commonalities explicit
  - a taxonomy of data structures
  - inheritance exposes the commonalities

The Second Course: Engineering Analysis Component

- eschew mystery math
- present mathematical tools *just in time*
- present analysis techniques *in context*
- balance intuitive understanding with mathematical rigour

Code vs. Pseudo-code

- students hate pseudo-code
  - what are the rules for writing pseudo-code?
  - there is no "standard" pseudo-code
- pseudo-code sometimes seems like hand-waving
- authors often resort to "pseudo-code" when the code gets too complex
  - why is the code too complex?
  - can OO techniques simplify matters?

Algorithmic Abstraction (Part 1)

```
class Tree : public virtual Container
{
public:
    virtual Object& Key () const = 0;
    virtual Tree& Subtree (unsigned int) const = 0;
    virtual bool IsEmpty () const = 0;
    virtual unsigned int Degree () const = 0;
    virtual void DepthFirstTraversal (Visitor&) const;
    // ...
};

void Tree::DepthFirstTraversal (Visitor& visitor) const
{
    if (visitor .IsDone ())
        return;
    if (!IsEmpty ())
    {
        visitor .PreVisit (Key ());
        for (unsigned int i = 0; i < Degree (); ++i)
            Subtree (i) .DepthFirstTraversal (visitor);
        visitor .PostVisit (Key ());
    }
}
```

Algorithmic Abstraction (Part 2)

```
class Solution : public Object
{
public:
    virtual bool IsFeasible () const = 0;
    virtual bool IsComplete () const = 0;
    virtual int Objective () const = 0;
    virtual int Bound () const = 0;
    virtual Iterator& Successors () const = 0;
};

class Solver
{
protected:
    void UpdateBest (Solution const&);
    int BestObjective () const;
    virtual void DoSolve (Solution const&) = 0;
public:
    virtual Solution& Solve (Solution const&);
};

Solution& problem = *new KnapsackProblem;
Solver& solver = *new DepthFirstSolver;
Solution& solution = solver .Solve (problem);
```

Algorithmic Abstraction (Part 3)

```
void DepthFirstSolver::DoSolve (Solution const& solution)
{
    if (solution .IsComplete ())
        UpdateBest (solution);
    else
    {
        for (Iterator& i = solution .Successors (); i; ++i)
        {
            Solution& successor = dynamic_cast<Solution&> (*i);
            DoSolve (successor);
        }
    }
}

void DepthFirstBranchAndBoundSolver::DoSolve (
    Solution const& solution)
{
    if (solution .IsComplete ())
        UpdateBest (solution);
    else
    {
        for (Iterator& i = solution .Successors (); i; ++i)
        {
            Solution& successor = dynamic_cast<Solution&> (*i);
            if (successor .IsFeasible () &&
                    successor .Bound () < BestObjective ())
                DoSolve (successor);
        }
    }
}
```

4. Summary

Summary

- Teach/preach objects from day one!

- How?
  - Speak the *vocabulary* of objects
  - Develop the OO *virtual machine* model
  - Use appropriate *abstractions*
  - Demonstrate common *design patterns*
  - Explicate *relationships* between classes
  - Exploit *commonalities* between classes
    - data/code/interface
  - Illustrate the salient *differences*