The Iterator Pattern

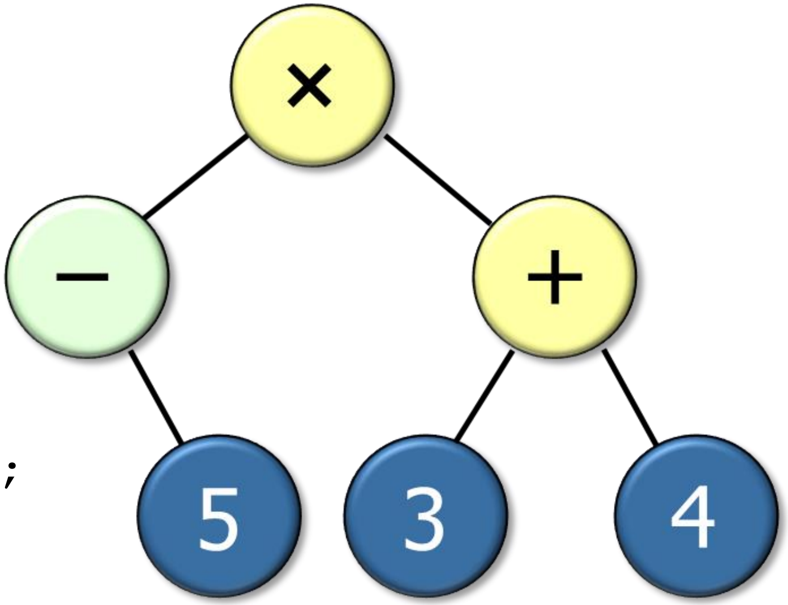
Motivating Example

Douglas C. Schmidt

Learning Objectives in This Lesson

- Recognize how the *Iterator* pattern can be applied to access all nodes in an expression tree flexibly & extensibly.

```
Expression_Tree tree = ...;
Visitor print_visitor = ...;

for (auto iter = tree.begin(order);
     iter != tree.end(order);
     ++iter)
   (*iter).accept(print_visitor);
```

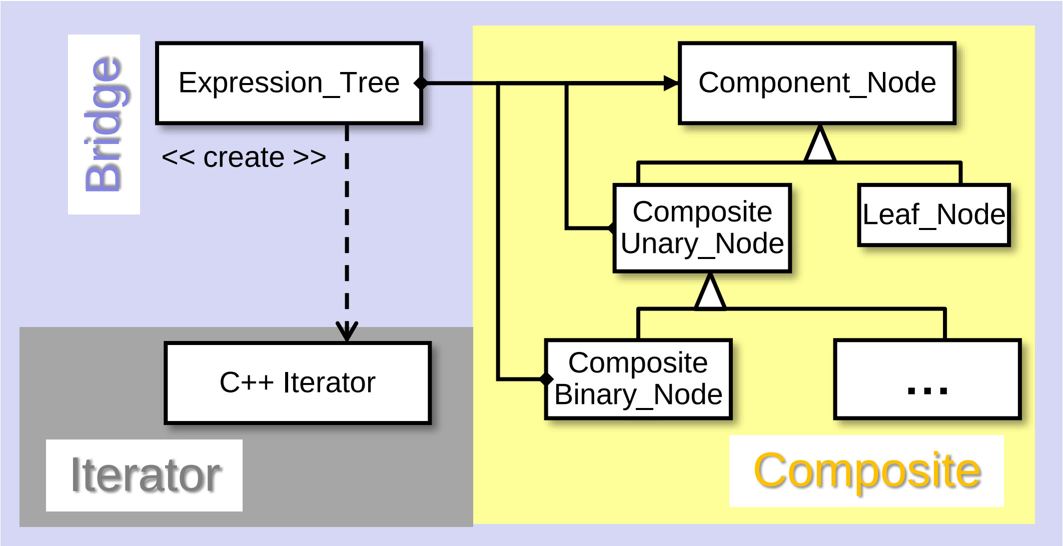


Douglas C. Schmidt

Motivating the Need for the Iterator Pattern in the Expression Tree App

A Pattern for Transparently Traversing Aggregates

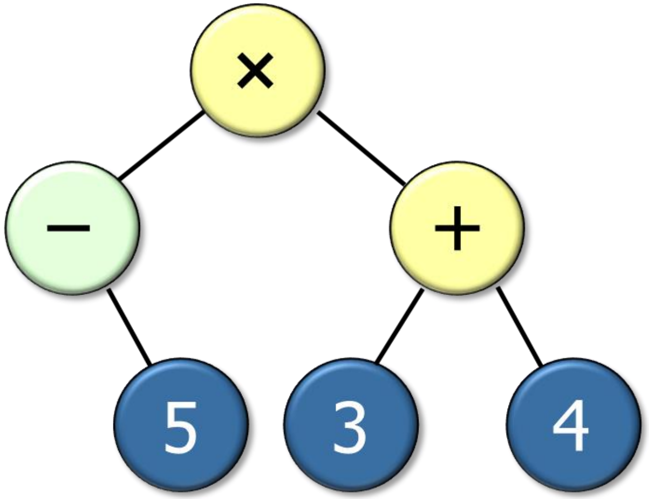
***Purpose**: Create objects that traverse the Composite-based expression tree & access each of its elements one at a time.*



Iterator decouples expression tree traversal from its internal structure.

Context: OO Expression Tree Processing App

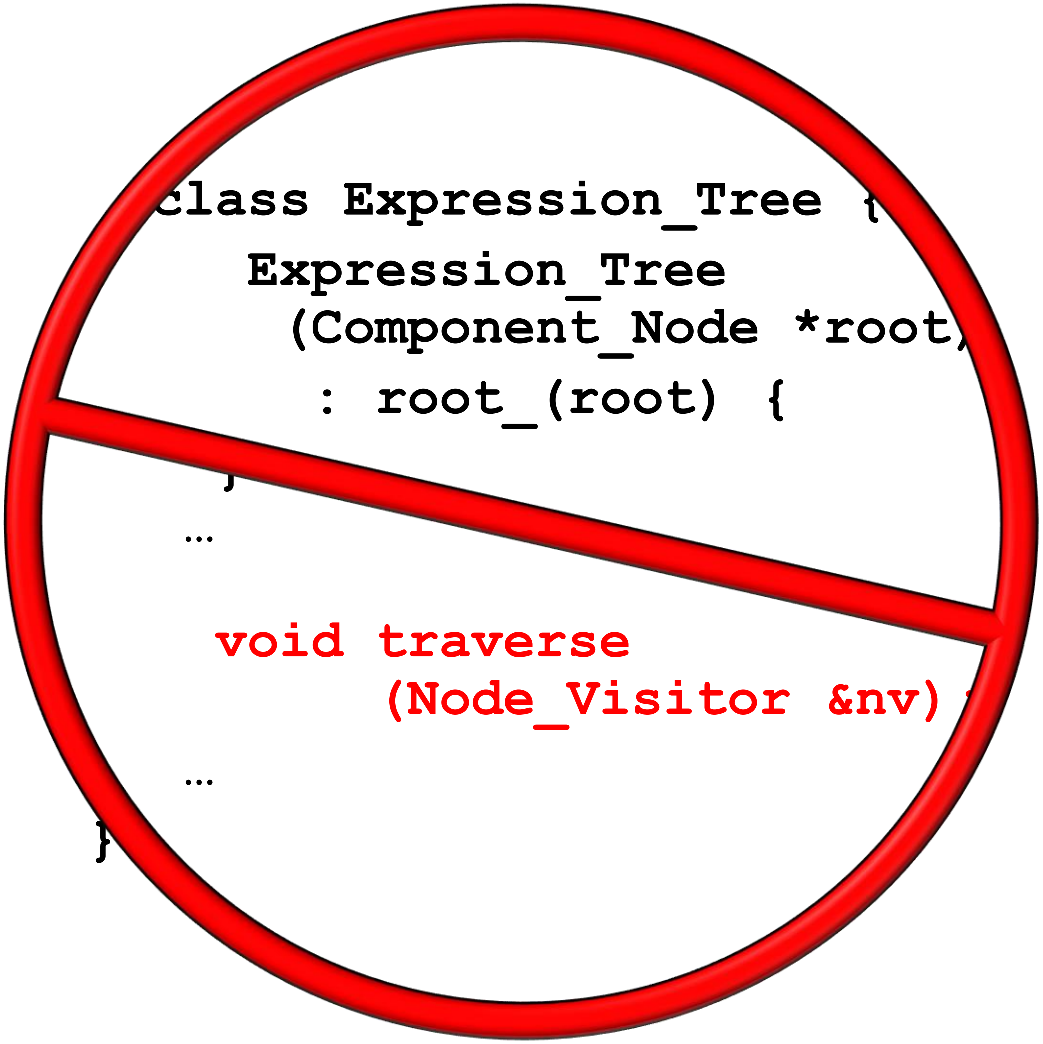
- Several user command requests require accessing all nodes in an expression tree.



Operation	Behavior
format	Allows the user to select the format of the input expression
expr	Allows the user to designate the current input expression
set	Sets a variable that can be used in an expression
print	Print the current input expression using the designated traversal order
eval	Evaluate the value of the current input expression
quit	Exit the program

Problem: Inflexible Expression Tree Traversal

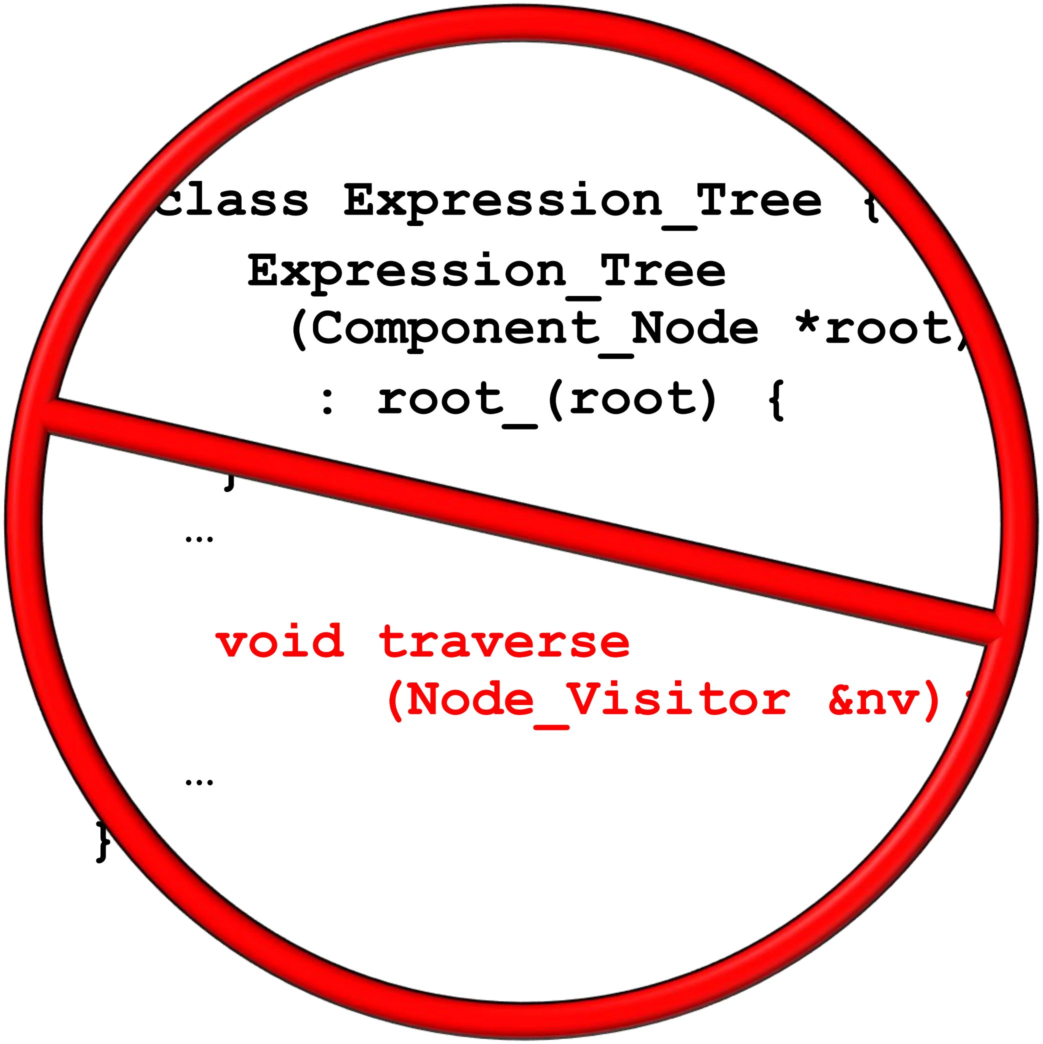
- Hard-coding the traversal logic into the expression tree itself is inflexible



```
class Expression_Tree {
   Expression_Tree
     (Component_Node *root)
      : root_(root) {
   }
   …

   void traverse
        (Node_Visitor &nv)
   …
}
```

Problem: Inflexible Expression Tree Traversal

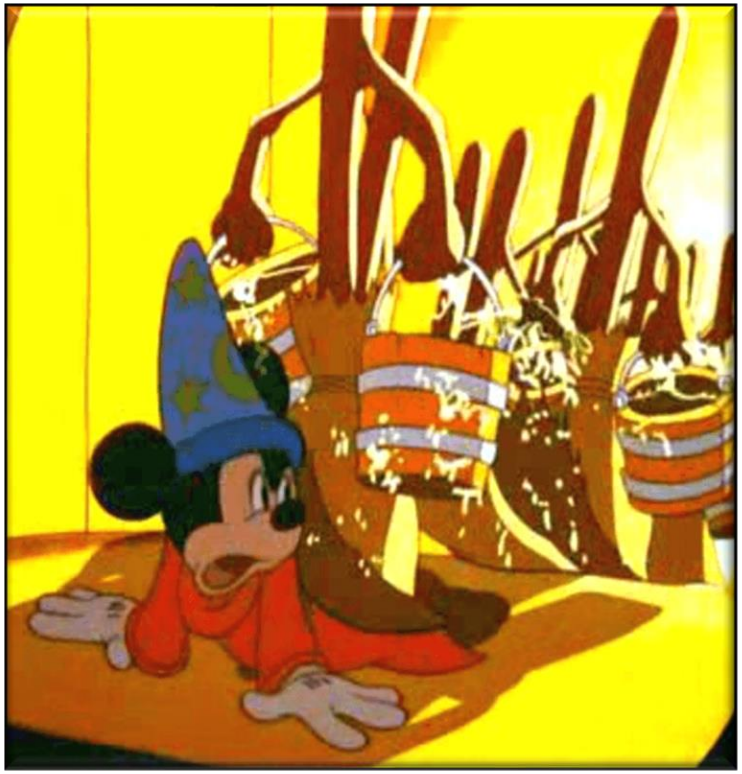
- Hard-coding the traversal logic into the expression tree itself is inflexible, e.g.
    - Only one traversal is allowed at a time



```
class Expression_Tree {
  Expression_Tree
    (Component_Node *root)
      : root_(root) {
  }
  ...

  void traverse
         (Node_Visitor &nv);
  ...
}
```

Problem: Inflexible Expression Tree Traversal

- Hard-coding the traversal logic into the expression tree itself is inflexible, e.g.
  - Only one traversal is allowed at a time
  - Hard to control where/when to stop the traversal



```
class Expression_Tree {
    Expression_Tree
      (Component_Node *root)
       : root_(root) {
    }
    …

    void traverse
           (Node_Visitor &nv) ;
    …
}
```

Problem: Inflexible Expression Tree Traversal

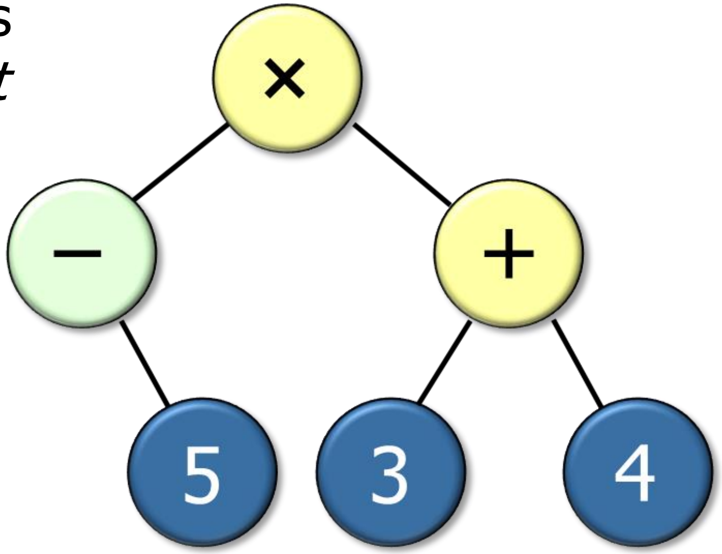
- Having a client explicitly traverse an expression tree via its internal links impedes extensibility.

```
void pre_order_traversal
          (Expression_Tree root) {
  if (!root.is_null())  {
    // Do something with root node
            ...

    // traverse left branch
    pre_order_traversal(root.left());

    // traverse right branch
    pre_order_traversal(root.right())
  }
```

This code breaks if we enhance ***Expression_Tree*** *to support ternary nodes.*

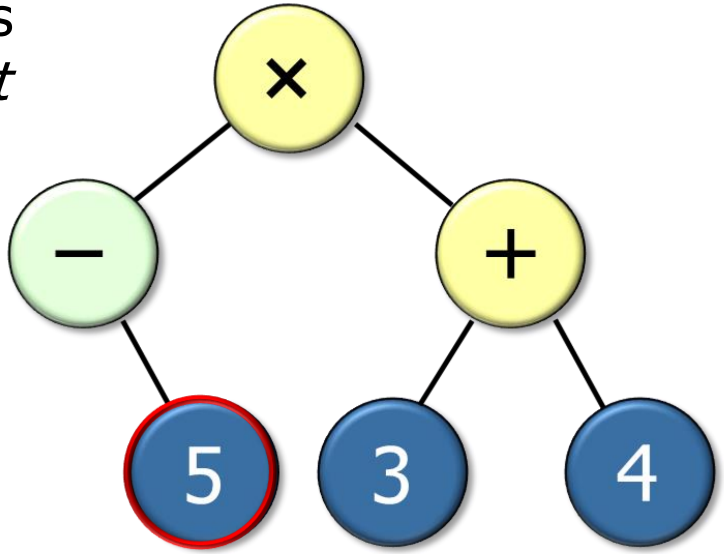
Solution: Encapsulate Traversal as an Object

- Create an iterator object that encapsulates the traversal of an expression tree *without* requiring clients to know how the tree is structured internally.



Solution: Encapsulate Traversal as an Object

- Create an iterator object that encapsulates the traversal of an expression tree *without* requiring clients to know how the tree is structured internally.

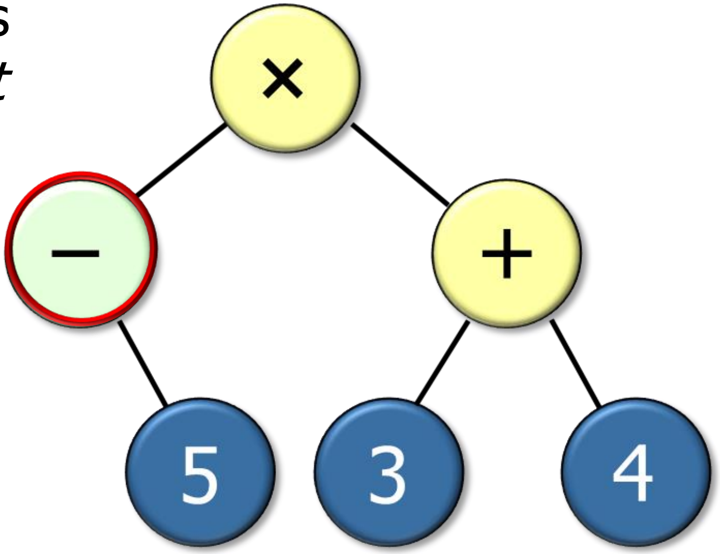


“Post-order” traversal =

5

Solution: Encapsulate Traversal as an Object

- Create an iterator object that encapsulates the traversal of an expression tree *without* requiring clients to know how the tree is structured internally.



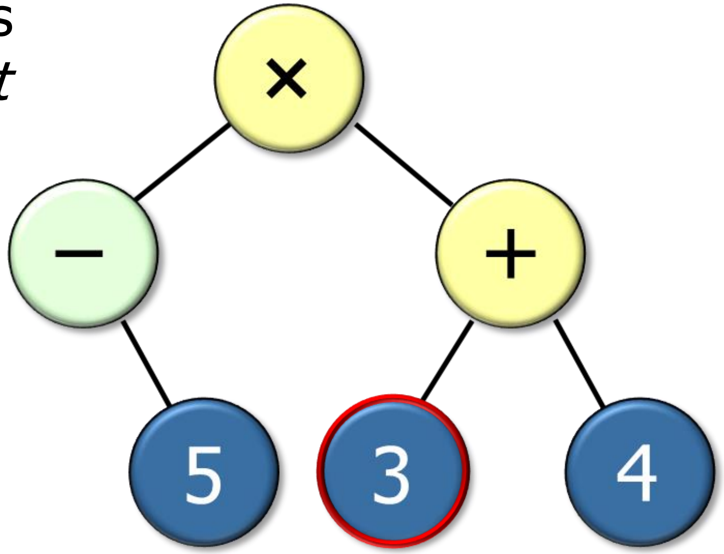
"Post-order" traversal =

5 ~

The '~' is used for post-order negate since '−' is ambiguous!

Solution: Encapsulate Traversal as an Object

- Create an iterator object that encapsulates the traversal of an expression tree *without* requiring clients to know how the tree is structured internally.

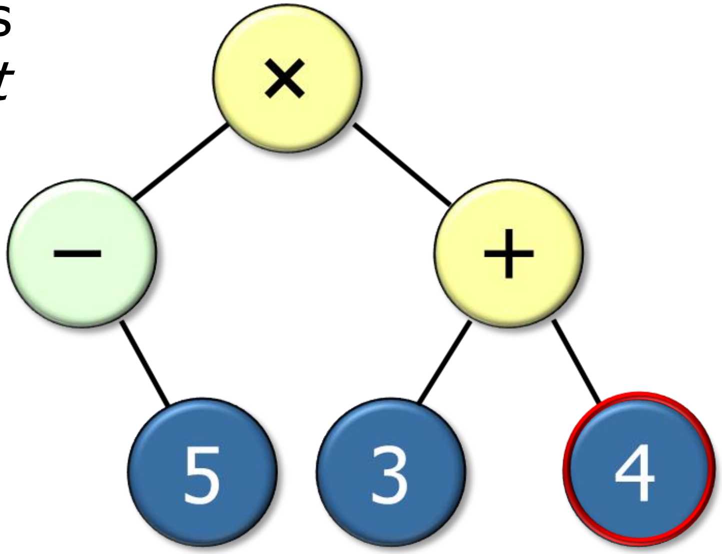


"Post-order" traversal =

5 ~ 3

Solution: Encapsulate Traversal as an Object

- Create an iterator object that encapsulates the traversal of an expression tree *without* requiring clients to know how the tree is structured internally.

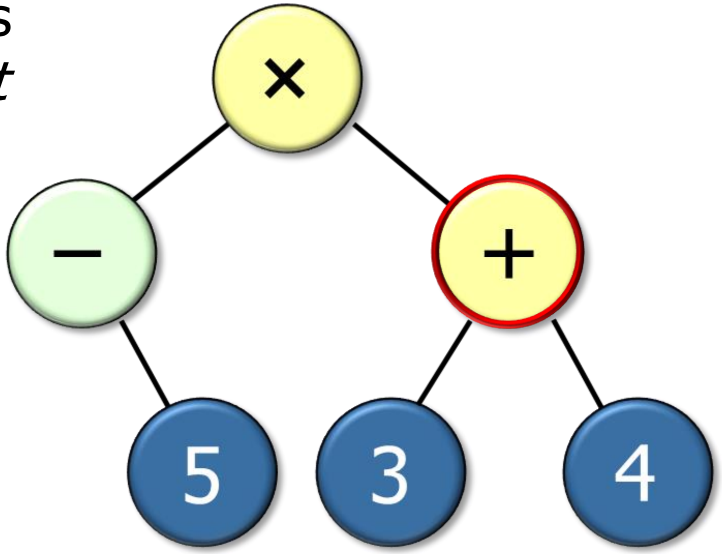


"Post-order" traversal =

5 ~ 3 4

Solution: Encapsulate Traversal as an Object

- Create an iterator object that encapsulates the traversal of an expression tree *without* requiring clients to know how the tree is structured internally.

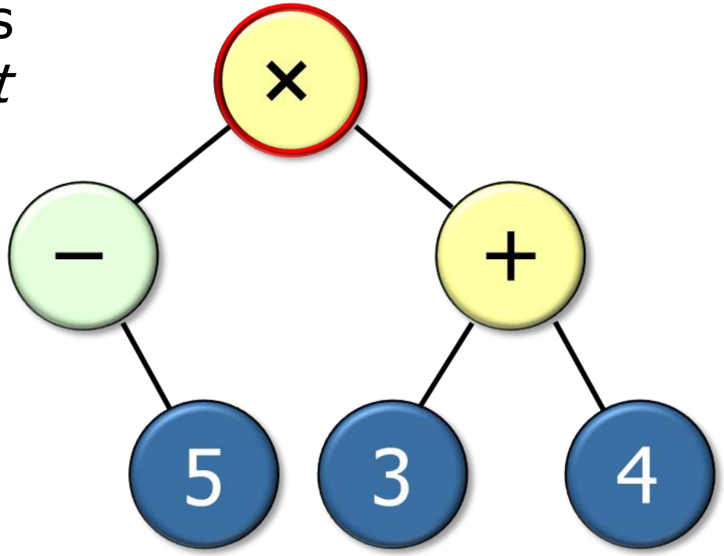


“Post-order” traversal =

5 ~ 3 4 +

Solution: Encapsulate Traversal as an Object

- Create an iterator object that encapsulates the traversal of an expression tree *without* requiring clients to know how the tree is structured internally.

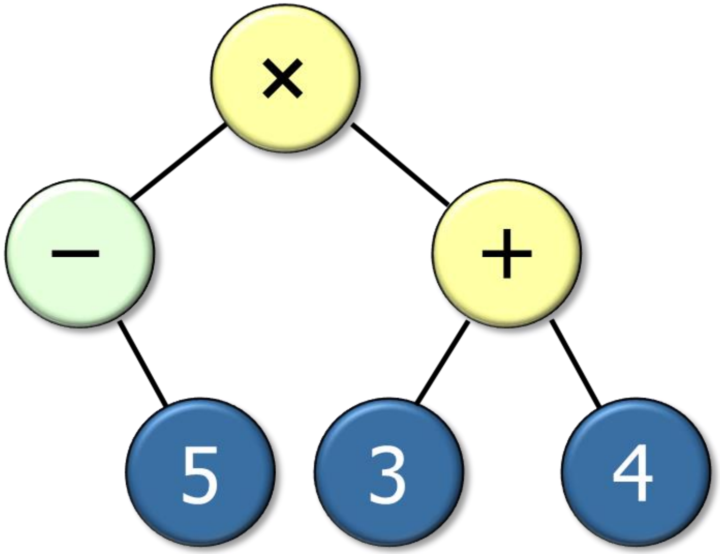


"Post-order" traversal =

5 ~ 3 4 + ×

Solution: Encapsulate Traversal as an Object

- Define methods to:
  1. Create an iterator (via factory method)



```
Expression_Tree tree = ...;
Visitor print_visitor = ...;

for (auto iter = tree.begin(order);
     iter != tree.end(order);
     ++iter)
   (*iter).accept(print_visitor);
```

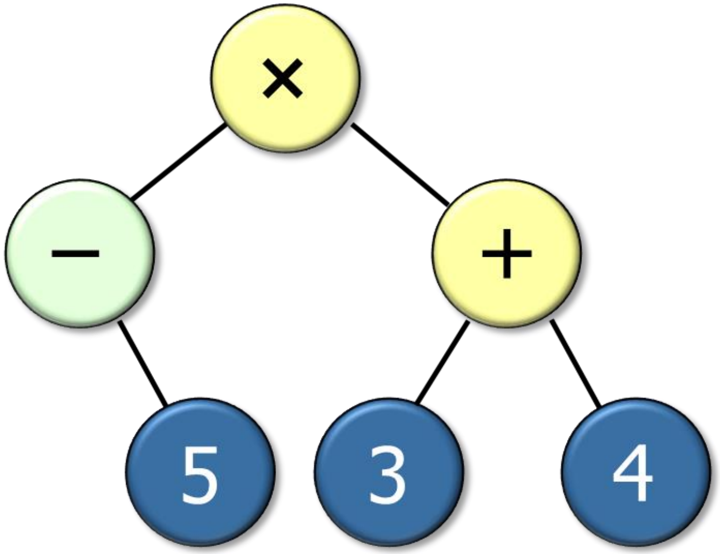
See en.wikipedia.org/wiki/Factory_method_pattern

Solution: Encapsulate Traversal as an Object

- Define methods to:

  1. Create an iterator (via factory method)

  2. Check to see if it's finished

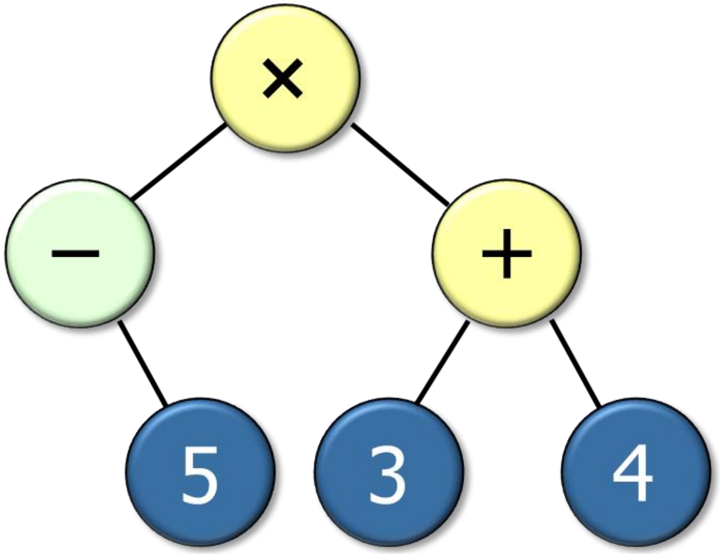


```
Expression_Tree tree = ...;
Visitor print_visitor = ...;

for (auto iter = tree.begin(order);
     iter != tree.end(order);
     ++iter)
   (*iter).accept(print_visitor);
```

Solution: Encapsulate Traversal as an Object

- Define methods to:

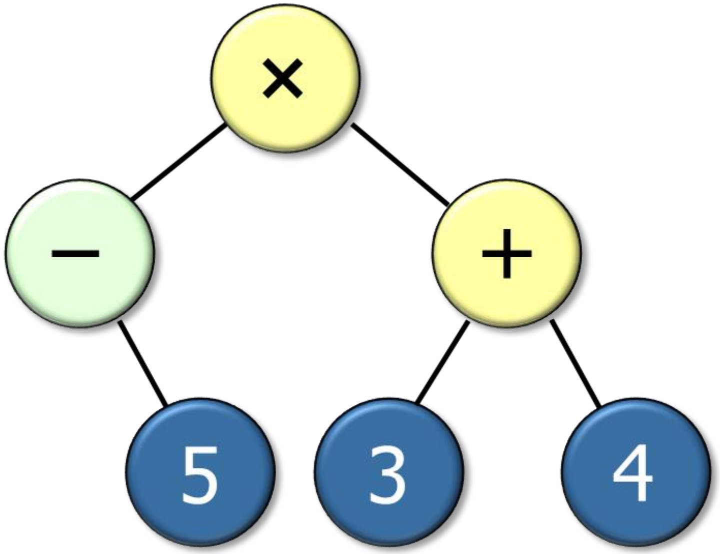
  1. Create an iterator (via factory method)
  2. Check to see if it's finished
  3. Access & process each element if it's not finished



```
Expression_Tree tree = ...;
Visitor print_visitor = ...;

for (auto iter = tree.begin(order);
     iter != tree.end(order);
     ++iter)
   (*iter).accept(print_visitor);
```

Solution: Encapsulate Traversal as an Object

- Define methods to:
  1. Create an iterator (via factory method)
  2. Check to see if it's finished
  3. Access & process each element if it's not finished
  4. Advanced the iterator by one

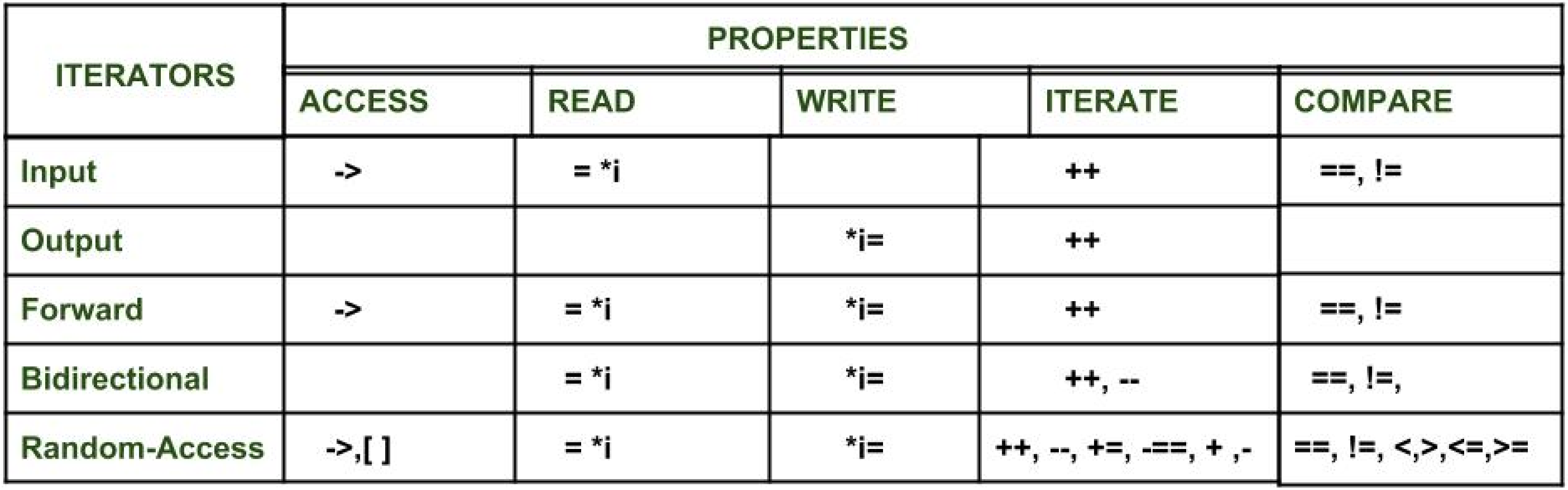


```
Expression_Tree tree = ...;
Visitor print_visitor = ...;

for (auto iter = tree.begin(order);
     iter != tree.end(order);
     ++iter)
   (*iter).accept(print_visitor);
```

C++ Iterator Interface Overview

- C++ STL defines a generic “interface” for traversing aggregate data

Iterator operations



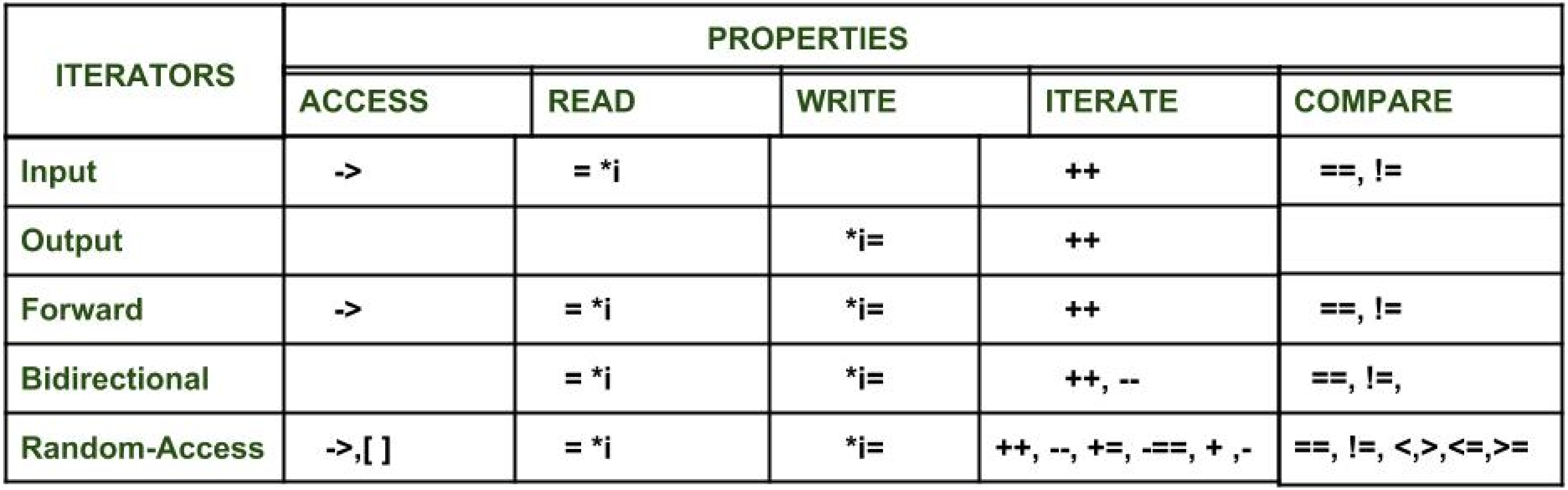
ITERATORS	PROPERTIES				
	ACCESS	READ	WRITE	ITERATE	COMPARE
Input	->	= *i		++	==, !=
Output			*i=	++	
Forward	->	= *i	*i=	++	==, !=
Bidirectional		= *i	*i=	++, --	==, !=,
Random-Access	->,[ ]	= *i	*i=	++, --, +=, -==, + ,-	==, !=, <,>,<=,>=

See www.geeksforgeeks.org/introduction-iterators-c

C++ Iterator Interface Overview

- C++ STL defines a generic “interface” for traversing aggregate data

Iterator operations



ITERATORS	PROPERTIES				
	ACCESS	READ	WRITE	ITERATE	COMPARE
Input	->	= *i		++	==, !=
Output			*i=	++	
Forward	->	= *i	*i=	++	==, !=
Bidirectional		= *i	*i=	++, --	==, !=,
Random-Access	->,[ ]	= *i	*i=	++, --, +=, -==, + ,-	==, !=, <,>,<=,>=

- **Commonality**: provides a common interface for expression tree iterators that conform to the C++ STL iterator interface
- **Variability**: can be configured with specific expression tree iterator implementation strategies via a *Creational* pattern