Design Pattern Case Studies with C++

Douglas C. Schmidt

http://www.cs.wustl.edu/~schmidt/

schmidt@cs.wustl.edu

Washington University, St. Louis

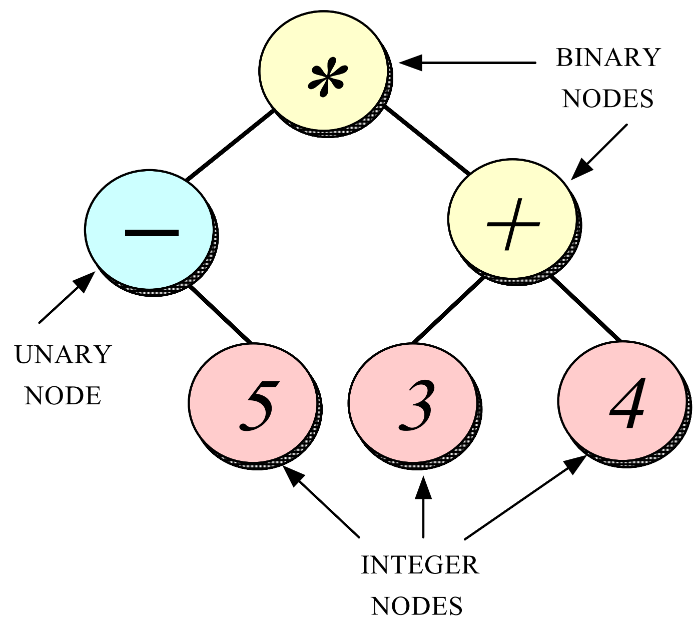
Case Studies Using Patterns

- The following slides describe several case studies using C++ and patterns to build highly extensible software
- The examples include
  1. Expression trees
     - *e.g.*, Bridge, Factory, Adapter
  2. System Sort
     - *e.g.*, Facade, Adapter, Iterator, Singleton, Factory Method, Strategy, Bridge, Double-Checked Locking Optimization
  3. Sort Verifier
     - *e.g.*, Strategy, Factory Method, Facade, Iterator, Singleton

Case Study 1: Expression Tree Evaluator

- The following inheritance and dynamic binding example constructs *expression trees*
  - Expression trees consist of nodes containing operators and operands
    - ∗ Operators have different *precedence levels*, *different associativities*, and different *arities*, *e.g.*,
      - · Multiplication takes precedence over addition
      - · The multiplication operator has two arguments, whereas unary minus operator has only one
    - ∗ Operands are integers, doubles, variables, etc.
      - · We'll just handle integers in this example...

Expression Tree Diagram



Expression Tree Behavior

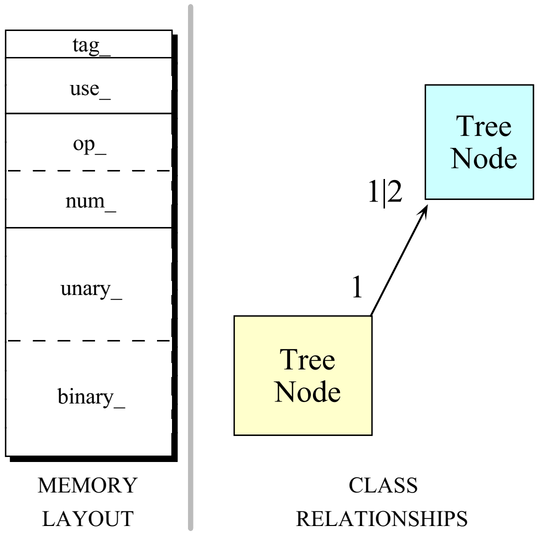
- *Expression trees*
  - Trees may be "evaluated" via different traversals
    - *e.g.*, in-order, post-order, pre-order, level-order
  - The evaluation step may perform various operations, *e.g.*,
    - Traverse and print the expression tree
    - Return the "value" of the expression tree
    - Generate code
    - Perform semantic analysis

C Version

- A typical functional method for implementing expression trees in C involves using a **struct/union** to represent data structure, *e.g.*,

```
typedef struct Tree_Node Tree_Node;
struct Tree_Node {
      enum {
            NUM, UNARY, BINARY
      } tag_;
      short use_; /* reference count */
      union {
            int num_;
            char op_[2];
      } o;
#define num_ o.num_
#define op_ o.op_
      union {
            Tree_Node *unary_;
            struct { Tree_Node *l_, *r_; } binary_;
      } c;
#define unary_ c.unary_
#define binary_ c.binary_
};
```


Memory Layout of C Version



- Here's what the memory layout of a **struct** Tree_Node object looks like

Print_Tree Function

- Typical C implementation (cont'd)
  - Use a **switch** statement and a recursive function to build and evaluate a tree, *e.g.*,

```
void print_tree (Tree_Node *root) {
      switch (root->tag_) {
      case NUM: printf ("%d", root->num_); break;
      case UNARY:
            printf ("(%s", root->op_[0]);
            print_tree (root->unary_);
            printf (")"); break;
      case BINARY:
            printf ("(");
            print_tree (root->binary_.l_);
            printf ("%s", root->op_[0]);
            print_tree (root->binary_.r_);
            printf (")"); break;
      default:
            printf ("error, unknown type\n");
            exit (1);
      }
}
```

Limitations with C Approach

- Problems or limitations with the typical C approach include
  - Language feature limitations in C
    - *e.g.*, no support for inheritance and dynamic binding
  - Incomplete modeling of the application domain, which results in
    1. Tight coupling between nodes and edges in **union** representation
    2. Complexity being in *algorithms* rather than the *data structures*
       - *e.g.*, **switch** statements are used to select between various types of nodes in the expression trees
         - compare with binary search!
       - Data structures are "passive" and functions do most processing work explicitly

More Limitations with C Approach

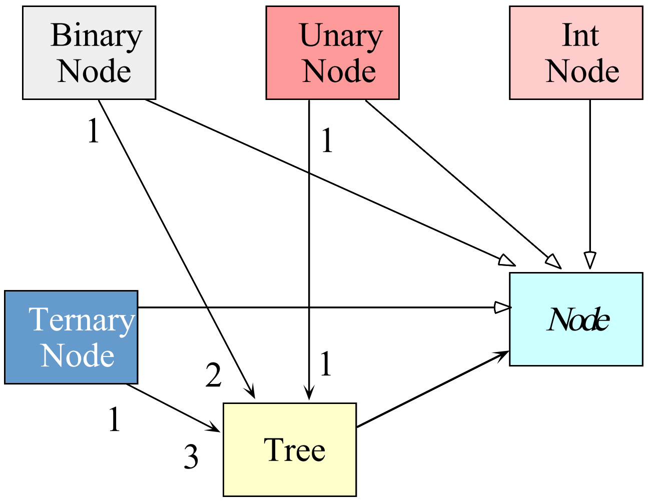
- The program organization makes it difficult to extend, *e.g.*,
  - Any small changes will ripple through the entire design and implementation
    - *e.g.*, see the "ternary" extension below
  - Easy to make mistakes **switch**ing on type tags...
- Solution wastes space by making worst-case assumptions *wrt* **struct**s and **union**s
  - This is not essential, but typically occurs
  - Note that this problem becomes worse the bigger the size of the largest item becomes!

OO Alternative

- Contrast previous functional approach with an object-oriented decomposition for the same problem:
  - Start with OO modeling of the "expression tree" application domain:
    - *e.g.*, go back to original picture
  - There are several classes involved:

    class Node: base class that describes expression tree vertices:
    class Int_Node: used for implicitly converting **int** to Tree node
    class Unary_Node: handles unary operators, *e.g.*, −10, +10, !a, or ~foo, etc.
    class Binary_Node: handles binary operators, *e.g.*, a + b, 10 − 30, etc.
    class Tree: "glue" code that describes expression tree edges
  - Note, these classes model elements in the application domain
    - *i.e.*, nodes and edges (or vertices and arcs)

Relationships Between Trees and Nodes



Design Patterns in the Expression Tree Program

- Adapter
  - "Convert the interface of a class into another interface clients expect"
    - *e.g.*, make `Tree` conform to interfaces expected by C++ iostreams operators
- Factory
  - "Centralize the assembly of resources necessary to create an object"
    - *e.g.*, decouple `Node` subclass initialization from their subsequent use
- Bridge
  - "Decouple an abstraction from its implementation so that the two can vary independently"
    - *e.g.*, printing the contents of a subtree

C++ Node Interface

- // Node.h

```
#if !defined (_NODE_H)
#define _NODE_H
class Tree; // Forward decl

// Describes the Tree vertices
class Node {
friend class Tree;

protected: // Only visible to derived classes
      Node (void)
            : use_ (1) {}

      // pure virtual
      virtual void print (ostream &) const = 0;

      // Important to make destructor virtual!
      virtual ~Node (void);

private:
      int use_; // Reference counter.
};
#endif /* _NODE_H */
```


C++ Tree Interface

- // Tree.h

```
#include "Node.h"

// Describes the Tree edges and acts as a Factory
class Tree {
public:
      // Factory operations
      Tree (int);
      Tree (const char *, Tree &);
      Tree (const char *, Tree &, Tree &);

      Tree (const Tree &t); // Copy constructor
      void operator= (const Tree &t); // Assignment
      ~Tree (void); // Destructor

      void print (ostream &) const;

private:
      Node *node_; // pointer to a rooted subtree.
};
```


C++ Int_Node and Unary_Node Interface

- // Int_Node.h

```
#include "Node.h"

class Int_Node : public Node {
public:
      Int_Node (int k);
      virtual void print (ostream &stream) const;
private:
      int num_; // operand value.
};
```

- // Unary_Node.h

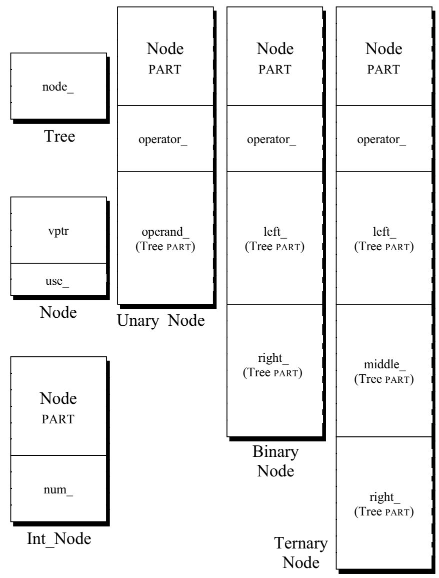
```
#include "Node.h"

class Unary_Node : public Node {
public:
      Unary_Node (const char *op, const Tree &t);
      virtual void print (ostream &stream) const;
private:
      const char *operation_;
      Tree operand_;
};
```

C++ Binary_Node Interface

- // Binary_Node.h

```
#include "Node.h"

class Binary_Node : public Node {
public:
      Binary_Node (const char *op,
                   const Tree &t1,
                   const Tree &t2);
      virtual void print (ostream &s) const;
private:
      const char *operation_;
      Tree left_;
      Tree right_;
};
```


Memory Layout for C++ Version



- Memory layouts for different subclasses of `Node`

C++ Int_Node and Unary_Node Implementations

- // Int_Node.C

```
#include "Int_Node.h"

Int_Node::Int_Node (int k): num_ (k) { }

void Int_Node::print (ostream &stream) const {
      stream << this->num_;
}
```

- // Unary_Node.C

```
#include "Unary_Node.h"

Unary_Node::Unary_Node (const char *op, const Tree &t1)
      : operation_ (op), operand_ (t1) { }

void Unary_Node::print (ostream &stream) const {
      stream << "(" << this->operation_ << " "
            << this->operand_ // recursive call!
            << ")";
}
```


C++ Binary_Node Implementation

- // Binary_Node.C

```
#include "Binary_Node.h"

Binary_Node::Binary_Node (const char *op,
                          const Tree &t1,
                          const Tree &t2):
      operation_ (op), left_ (t1), right_ (t2)
      { }

void Binary_Node::print (ostream &stream) const {
      stream << "(" << this->left_ // recursive call
            << " " << this->operation_
            << " " << this->right_ // recursive call
            << ")";
}
```

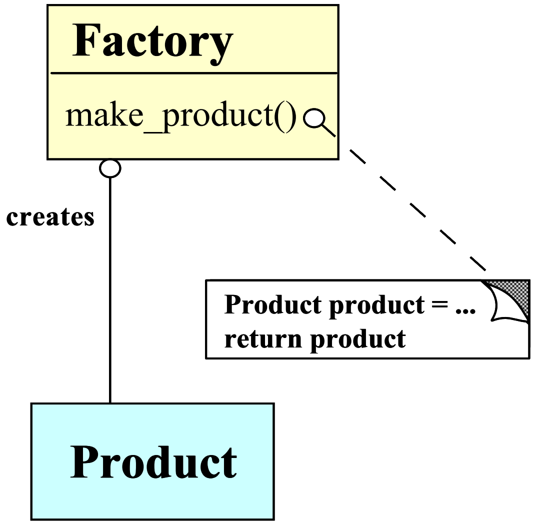
Initializing the Node Subclasses

- *Problem*
  - How to ensure the Node subclasses are initialized properly
- *Forces*
  - There are different types of Node subclasses
    * *e.g.*, take different number and type of arguments
  - We want to centralize initialization in one place because it is likely to change...
- *Solution*
  - Use a *Factory* pattern to initialize the Node subclasses

The Factory Pattern

- *Intent*
  - "Centralize the assembly of resources necessary to create an object"
    * Decouple object creation from object use by localizing creation knowledge
- This pattern resolves the following forces:
  - Decouple initialization of the `Node` subclasses from their subsequent use
  - Makes it easier to change or add new Node subclasses later on
    * *e.g.*, Ternary nodes...
- A variant of the Factory Method

Structure of the Factory Pattern



Using the Factory Pattern

- The Factory pattern is used by the Tree class to initialize Node subclasses:

```
Tree::Tree (int num)
    : node_ (new Int_Node (num)) {}

Tree::Tree (const char *op, const Tree &t)
    : node_ (new Unary_Node (op, t)) {}

Tree::Tree (const char *op,
            const Tree &t1,
            const Tree &t2):
    : node_ (new Binary_Node (op, t1, t2)) {}
```

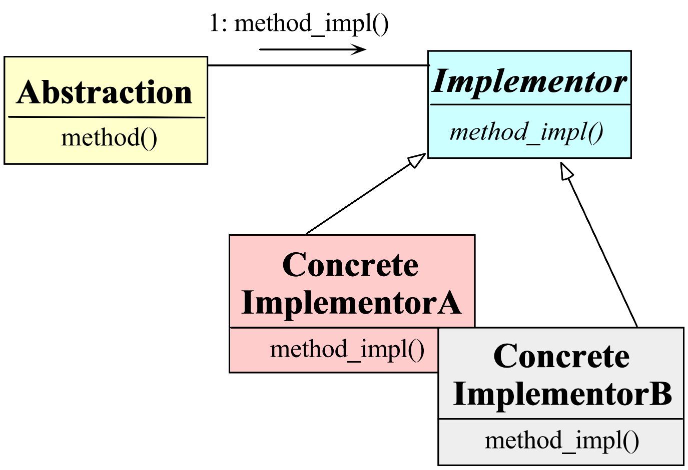
Printing Subtrees

- *Problem*
  - How do we print subtrees without revealing their types?
- *Forces*
  - The `Node` subclass should be hidden within the `Tree` instances
  - We don't want to become dependent on the use of `Nodes`, inheritance, and dynamic binding, etc.
- *Solution*
  - Use the *Bridge* pattern to shield the use of inheritance and dynamic binding

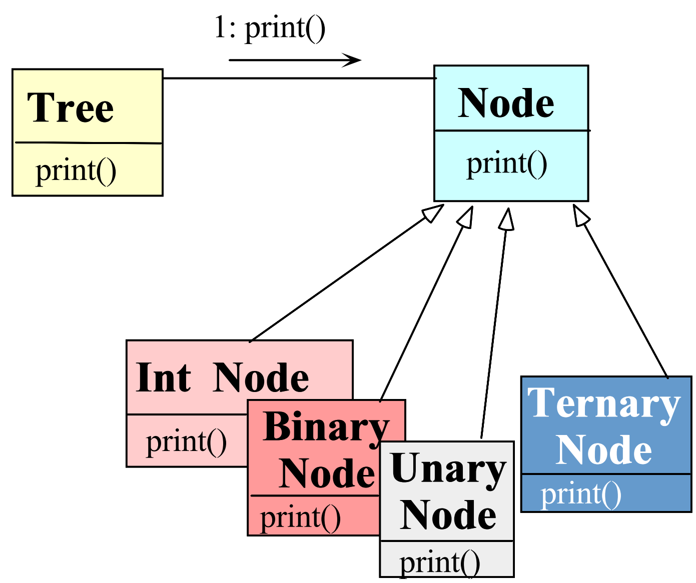
The Bridge Pattern

- *Intent*
  - "Decouple an abstraction from its implementation so that the two can vary independently"
- This pattern resolves the following forces that arise when building extensible software with C++
  1. *How to provide a stable, uniform interface that is both closed and open, i.e.,*
     - Interface is *closed* to prevent direct code changes
     - Implementation is *open* to allow extensibility
  2. *How to simplify the implementation of* `operator<<`

Structure of the Bridge Pattern



Using the Bridge Pattern



Illustrating the Bridge Pattern in C++

- The Bridge pattern is used for printing expression trees:

```
void Tree::print (ostream &os) const
{
    this->node_->print (os);
}
```

- Note how this pattern decouples the `Tree` interface for printing from the `Node` subclass implementation
  - *i.e.*, the `Tree` interface is *fixed*, whereas the `Node` implementation varies
  - However, clients need not be concerned about the variation...

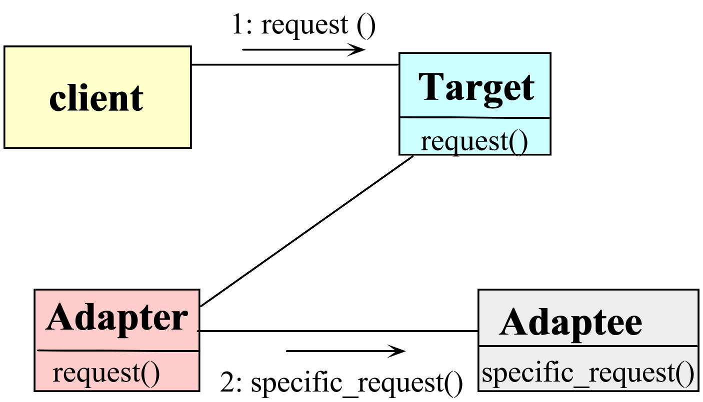
Integrating with C++ I/O Streams

- *Problem*
  - Our `Tree` interface uses a `print` method, but most C++ programmers expect to use I/O Streams
- *Forces*
  - Want to integrate our existing C++ `Tree` class into the I/O Stream paradigm without modifying our class or C++ I/O
- *Solution*
  - Use the *Adapter* pattern to integrate `Tree` with I/O Streams

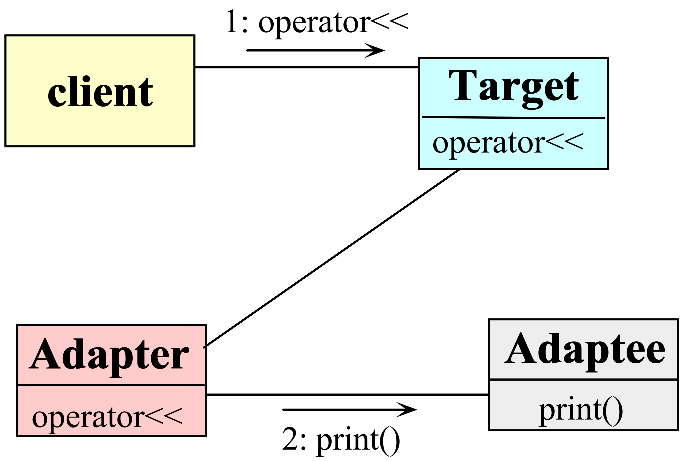
The Adapter Pattern

- *Intent*
  - Convert the interface of a class into another interface client expects
    - ∗ Adapter lets classes work together that couldn't otherwise because of incompatible interfaces
- This pattern resolves the following force:
  1. How to transparently integrate the `Tree` with the C++ iostream operators

Structure of the Adapter Pattern



Using the Adapter Pattern



Using the Adapter Pattern

- The Adapter pattern is used to integrate with C++ I/O Streams

```
ostream &operator<< (ostream &s, const Tree &tree) {
    tree.print (s);
    // This triggers Node * virtual call via
    // tree.node_->print (s), which is
    // implemented as the following:
    // (*tree.node_->vptr[1]) (tree.node_, s);
    return s;
}
```

- Note how the C++ code shown above uses I/O streams to "adapt" the `Tree` interface...

C++ Tree Implementation

- Reference counting via the "counted body" idiom

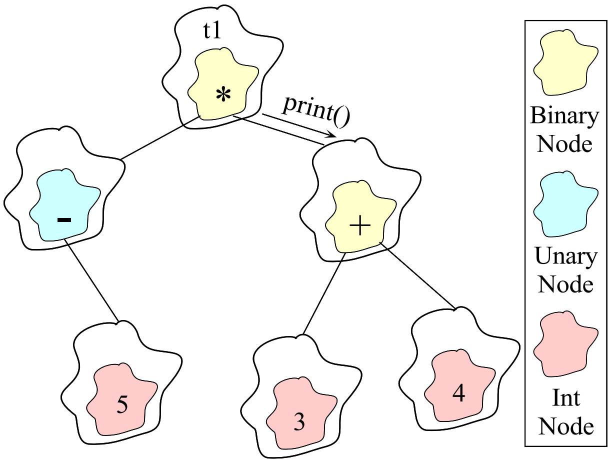
```
Tree::Tree (const Tree &t): node_ (t.node_) {
    // Sharing, ref-counting.
    this->node_->use_++;
}

void Tree::operator= (const Tree &t) {
    // order important here!
    t.node_->use_++;
    this->node_->use_--;
    if (this->node_->use_ == 0)
        delete this->node_;
    this->node_ = t.node_;
}

Tree::~Tree (void) {
    // Ref-counting, garbage collection
    this->node_->use_--;
    if (this->node_->use_<= 0)
        delete this->node_;
}
```


C++ Main Program

- // main.C

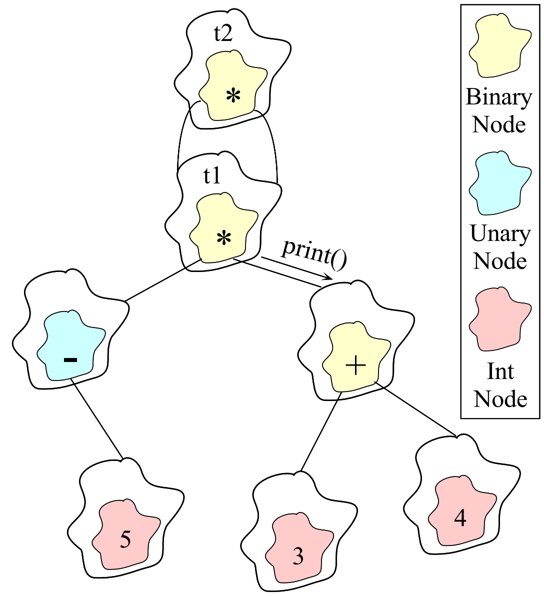
```
#include <iostream.h>
#include "Tree.h"

int main (void)
{
    const Tree t1 = Tree ("*",
                        Tree ("-", 5),
                        Tree ("+", 3, 4));
    // Tree ("*",
    // Tree ("-", Tree (5)),
    // Tree ("+", Tree (3), Tree (4)));

    // prints ((-5) * (3 + 4)).
    cout << t1 << endl;
    const Tree t2 = Tree ("*", t1, t1);

    // prints (((-5) * (3 + 4)) * ((-5) * (3 + 4))).
    cout << t2 << endl;

    // Destructors of t1 and t2 recursively
    // delete entire tree leaving scope.
}
```

Expression Tree Diagram 1



- Expression tree for t1 = ((−5) * (3 + 4))

Expression Tree Diagram 2



- Expression tree for t2 = (t1 * t1)

Adding Ternary_Nodes

- Extending the existing program to support ternary nodes is straightforward
  - *i.e.*, just derived new class Ternary_Node

    class Ternary_Node: handles ternary operators, *e.g.*, a == b ? c : d, etc.

- // Ternary_Node.h

```
#include "Node.h"

class Ternary_Node : public Node {
public:
      Ternary_Node (const char *, const Tree &,
                    const Tree &, const Tree &);
      virtual void print (ostream &) const;
private:
      const char *operation_;
      Tree left_, middle_, right_;
};
```


C++ Ternary_Node Implementation

- // Ternary_Node.C

```
#include "Ternary_Node.h"
Ternary_Node::Ternary_Node (const char *op,
                            const Tree &a,
                            const Tree &b,
                            const Tree &c)
      : operation_ (op), left_ (a), middle_ (b),
           right_ (c) {}

void Ternary_Node::print (ostream &stream) const {
      stream << this->operation_ << "("
           << this->left_ // recursive call
           << "," << this->middle_ // recursive call
           << "," << this->right_ // recursive call
           << ")";
}
```

- // Modified **class** Tree Factory

```
class Tree { // add 1 class constructor
public:
      Tree (const char *, const Tree &,
           const Tree &, const Tree &)
      : node_ (new Ternary_Node (op, l, m, r)) {}
// Same as before...
```

Differences from C Implementation

- On the other hand, modifying the original C approach requires changing:
  - The original data structures, *e.g.*,

```
struct Tree_Node {
    enum {
        NUM, UNARY, BINARY, TERNARY
    } tag_;
    // same as before
    union {
        // same as before
        // add this
        struct {
            Tree_Node *l_, *m_, *r_;
        } ternary_;
    } c;
#define ternary_ c.ternary_
};
```

  - and many parts of the code, *e.g.*,

```
void print_tree (Tree_Node *root) {
    // same as before
    case TERNARY: // must be TERNARY.
        printf ("(");
        print_tree (root->ternary_.l_);
        printf ("%c", root->op_[0]);
        print_tree (root->ternary_.m_);
        printf ("%c", root->op_[1]);
        print_tree (root->ternary_.r_);
        printf (")"); break;
    // same as before
}
```


Summary of Expression Tree Example

- OO version represents a more complete modeling of the application domain
  - *e.g.*, splits data structures into modules that correspond to "objects" and relations in expression trees
- Use of C++ language features simplifies the design and facilitates extensibility
  - *e.g.*, the original source was hardly affected
- Use of patterns helps to motivate and justify design choices

Potential Problems with OO Design

- Solution is very "data structure rich"
  - *e.g.*, requires configuration management to handle many headers and .C files!
- May be somewhat less efficient than original C approach
  - *e.g.*, due to virtual function overhead
- In general, however, virtual functions may be no less inefficient than large **switch** statements or **if/else** chains...
- As a rule, be careful of micro vs. macro optimizations
  - *i.e.*, always profile your code!

Case Study 2: System Sort

- Develop a general-purpose system sort
  - It sorts lines of text from standard input and writes the result to standard output
  - *e.g.*, the UNIX system sort

    % sort < big.file > sorted.file
- In the following, we'll examine the primary forces that shape the design of this application
- For each force, we'll examine patterns that resolve it

External Behavior of System Sort

- A "line" is a sequence of characters terminated by a newline
- Default ordering is lexicographic by bytes in machine collating sequence
- The ordering is affected globally by the following options:
  - Ignore case (`-i`)
  - Sort numerically (`-n`)
  - Sort in reverse (`-r`)
  - Begin sorting at a specified field (`-f`)
  - Begin sorting at a specified column (`-c`)
- Note, our program need not sort files larger than main memory

High-level Forces

- Solution should be both time and space efficient
  - *e.g.*, must use appropriate algorithms and data structures
  - Efficient I/O and memory management are particularly important
  - Our solution uses minimal dynamic binding (to avoid unnecessary overhead)
- Solution should leverage reusable components
  - *e.g.*, iostreams, Array and Stack classes, etc.
- Solution should yield reusable components
  - *e.g.*, efficient input classes, generic sort routines, etc.

General Form of Solution

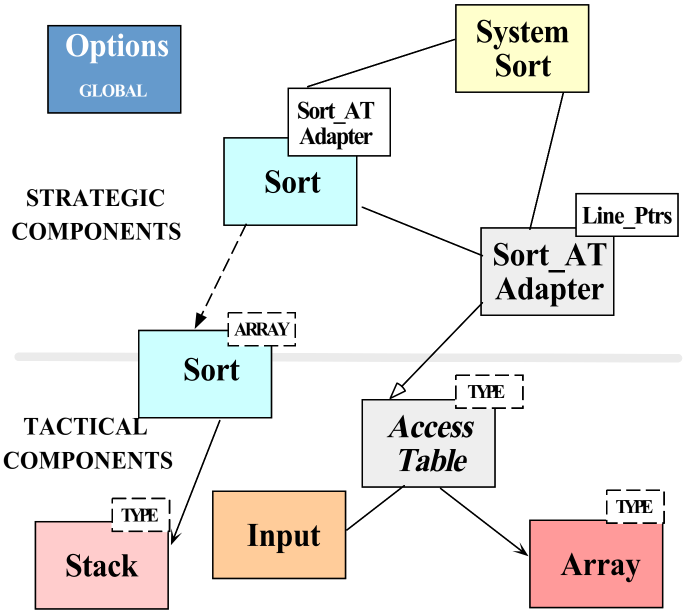
- Note the use of existing C++ mechanisms like I/O streams

```
// Reusable function
template <class ARRAY> void
sort (ARRAY &a);

int main (int argc, char *argv[])
{
  parse_args (argc, argv);

  Input_Array input;

  cin >> input;

  sort (input);

  cout << input;
}
```


General OOD Solution Approach

- Identify the "objects" in the application and solution space
  - *e.g.*, stack, array, input class, options, access table, sorts, etc.
- Recognize and apply common design patterns
  - *e.g.*, Singleton, Factory, Adapter, Iterator
- Implement a framework to coordinate components
  - *e.g.*, use C++ classes and parameterized types

C++ Class Model



C++ Class Components

- *Tactical components*
  - Stack
    - Used by non-recursive quick sort
  - Array
    - Stores pointers to lines and fields
  - Access_Table
    - Used to store and sort input
  - Input
    - Efficiently reads arbitrary sized input using only 1 dynamic allocation and 1 copy

C++ Class Components

- *Strategic components*
  - System_Sort
    - Integrates everything...
  - Sort_AT_Adapter
    - Integrates the `Array` and the `Access_Table`
  - Options
    - Manages globally visible options
  - Sort
    - *e.g.*, both quicksort and insertion sort

Detailed Format for Solution

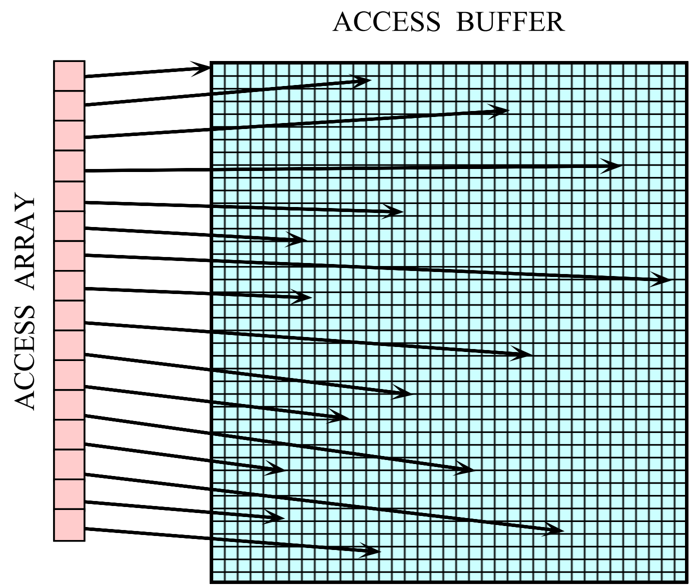
- Note the separation of concerns

```
// Prototypes
template <class ARRAY> void sort (ARRAY &a);
void operator >> (istream &,
                  Access_Table<Line_Ptrs> &);
void operator << (ostream &,
                  const Access_Table<Line_Ptrs> &);

int main (int argc, char *argv[])
{
  Options::instance ()->parse_args (argc, argv);

  cin >> System_Sort::instance ()->access_table ();

  sort (System_Sort::instance ()->access_table ());

  cout << System_Sort::instance ()->access_table ();
}
```

Reading Input Efficiently

- *Problem*
  - The input to the system sort can be arbitrarily large (*e.g.*, up to size of main memory)
- *Forces*
  - To improve performance solution must minimize:
    1. Data copying and data manipulation
    2. Dynamic memory allocation
- *Solution*
  - Create an `Input` class that reads arbitrary input efficiently

Access Table Format



The Input Class

- Efficiently reads arbitrary-sized input using only 1 dynamic allocation

```
class Input
{
public:
  // Reads from <input> up to <terminator>,
  // replacing <search> with <replace>.  Returns
  // pointer to dynamically allocated buffer.
  char *read (istream &input,
      int terminator = EOF,
      int search = '\n',
      int replace = '\0');
  // Number of bytes replaced.
  size_t replaced (void) const;

  // Size of buffer.
  size_t size (void) const;

private:
  // Recursive helper method.
  char *recursive_read (void);

  // ...
};
```


Design Patterns in System Sort

- Facade
  - "Provide a unified interface to a set of interfaces in a subsystem"
    * Facade defines a higher-level interface that makes the subsystem easier to use
  - *e.g.*, `sort` provides a facade for the complex internal details of efficient sorting
- Adapter
  - "Convert the interface of a class into another interface clients expect"
    * Adapter lets classes work together that couldn't otherwise because of incompatible interfaces
  - *e.g.*, make `Access_Table` conform to interfaces expected by `sort` and iostreams

Design Patterns in System Sort (cont'd)

- Factory
  - "Centralize the assembly of resources necessary to create an object"
  - *e.g.*, decouple initialization of `Line_Ptrs` used by `Access_Table` from their subsequent use
- Bridge
  - "Decouple an abstraction from its implementation so that the two can vary independently"
  - *e.g.*, comparing two lines to determine ordering
- Strategy
  - "Define a family of algorithms, encapsulate each one, and make them interchangeable"
  - *e.g.*, allow flexible pivot selection

Design Patterns in System Sort (cont'd)

- Singleton
  - "Ensure a class has only one instance, and provide a global point of access to it"
  - *e.g.*, provides a single point of access for system sort and for program options
- Double-Checked Locking Optimization
  - "Ensures atomic initialization or access to objects and eliminates unnecessary locking overhead"
  - *e.g.*, allows multiple threads to execute `sort`
- Iterator
  - "Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation"
  - *e.g.*, provides a way to print out the sorted lines without exposing representation

Sort Algorithm

- For efficiency, two types of sorting algorithms are used:
  1. *Quicksort*
     - Highly time and space efficient sorting arbitrary data
     - O(n log n) average-case time complexity
     - O(n2) worst-case time complexity
     - O(log n) space complexity
     - Optimizations are used to avoid worst-case behavior
  2. *Insertion sort*
     - Highly time and space efficient for sorting "almost ordered" data
     - O(n2) average- and worst-case time complexity
     - O(1) space complexity

Quicksort Optimizations

1. *Non-recursive*
   - Uses an explicit stack to reduce function call overhead
2. *Median of 3 pivot selection*
   - Reduces probability of worse-case time complexity
3. *Guaranteed* (log n) *space complexity*
   - Always "pushes" larger partition
4. *Insertion sort for small partitions*
   - Insertion sort runs fast on almost sorted data

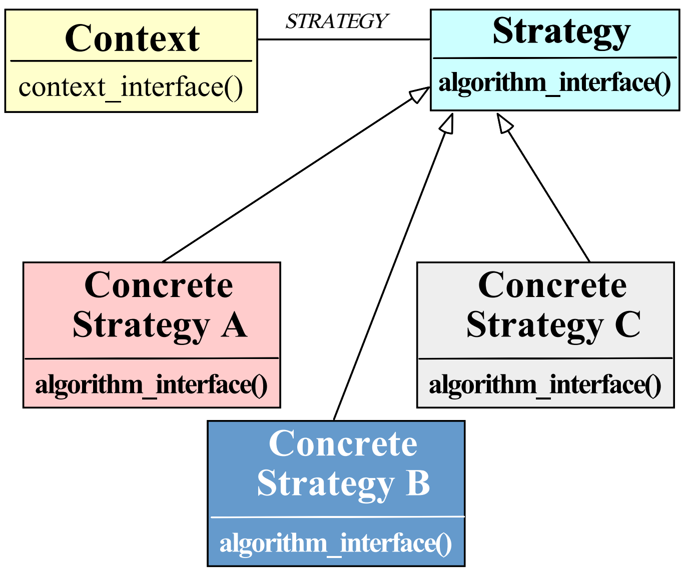
Selecting a Pivot Value

- *Problem*
  - There are various algorithms for selecting a pivot value
    - *e.g.*, randomization, median of three, etc.
- *Forces*
  - Different input may sort more efficiently using different pivot selection algorithms
- *Solution*
  - Use the *Strategy* pattern to select the pivot selection algorithm

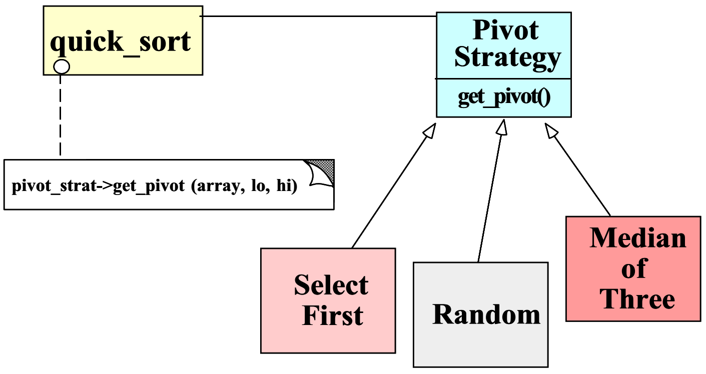
The Strategy Pattern

- *Intent*
  - Define a family of algorithms, encapsulate each one, and make them interchangeable
    - Strategy lets the algorithm vary independently from clients that use it
- This pattern resolves the following forces
  1. *How to extend the policies for selecting a pivot value without modifying the main quicksort algorithm*
  2. Provide a *one size fits all* interface without forcing a *one size fits all* implementation

Structure of the Strategy Pattern



Using the Strategy Pattern



Implementing the Strategy Pattern

- ARRAY is the particular "context"

```
template <class ARRAY>
void sort (ARRAY &array)
{
  Pivot<ARRAY> *pivot_strat = Pivot<ARRAY>::make_pivot
    (Options::instance ()->pivot_strat ());

  quick_sort (array, pivot_strat);
}

template <class ARRAY, class PIVOT_STRAT>
quick_sort (ARRAY &array, PIVOT_STRAT *pivot_strat)
{
  for (;;) {
    ARRAY::TYPE pivot; // typename ARRAY::TYPE pivot...

    pivot = pivot_strat->get_pivot (array, lo, hi);

    // Partition array[lo, hi] relative to pivot...
  }
}
```

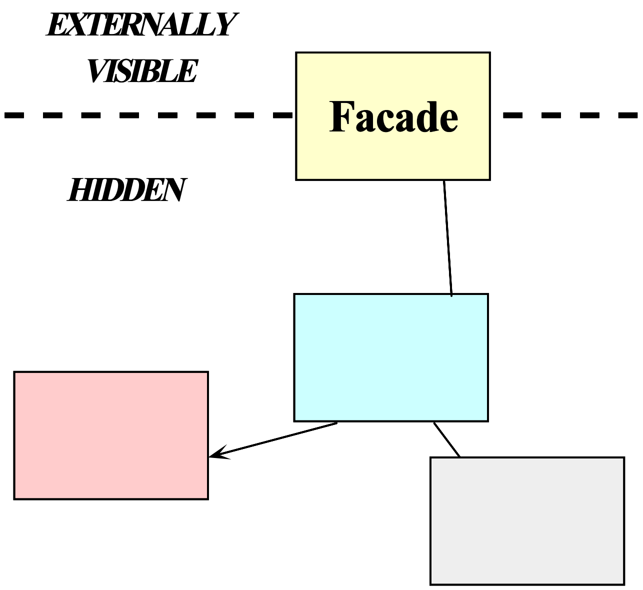

Devising a Simple Sort Interface

- *Problem*
  - Although the implementation of the `sort` function is complex, the interface should be simple to use
- *Key forces*
  - Complex interface are hard to use, error prone, and discourage extensibility and reuse
  - Conceptually, sorting only makes a few assumptions about the "array" it sorts
    * *e.g.*, supports `operator[]` methods, size, and element `TYPE`
  - We don't want to arbitrarily limit types of arrays we can sort
- *Solution*
  - Use the *Facade* and *Adapter* patterns to simplify the sort program

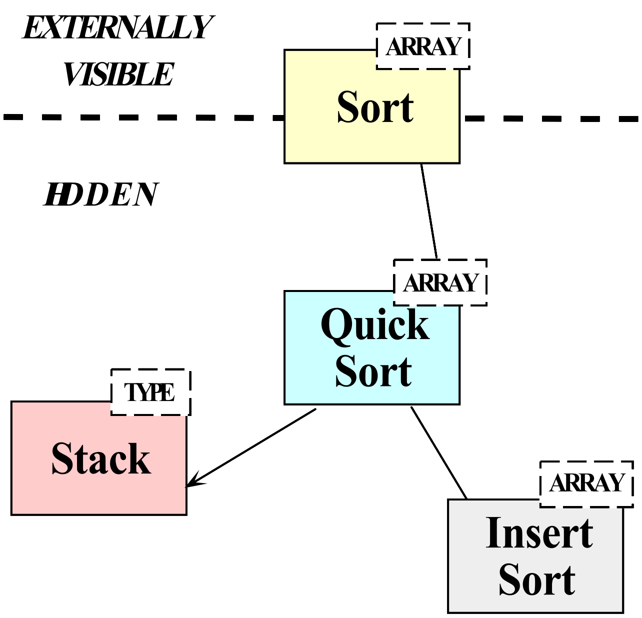
Facade Pattern

- *Intent*
  - Provide a unified interface to a set of interfaces in a subsystem
    * Facade defines a higher-level interface that makes the subsystem easier to use
- This pattern resolves the following forces:
  1. Simplifies the `sort` interface
     - *e.g.*, only need to support `operator[]` and `size` methods, and element `TYPE`
  2. Allows the implementation to be efficient and arbitrarily complex without affecting clients

Structure of the Facade Pattern



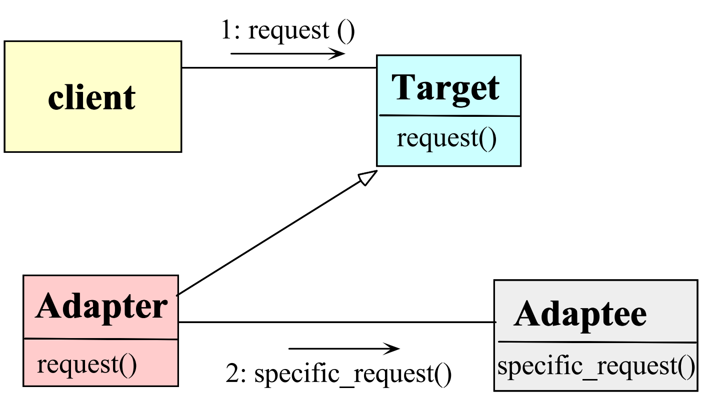
Using the Facade Pattern



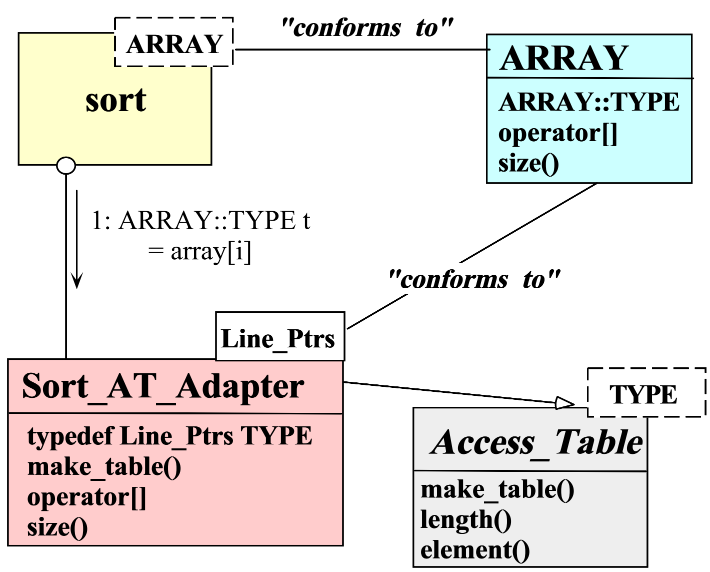
The Adapter Pattern

- *Intent*
  - "Convert the interface of a class into another interface clients expect"
    - Adapter lets classes work together that couldn't otherwise because of incompatible interfaces
- This pattern resolves the following forces:
  1. How to transparently integrate the `Access_Table` with the `sort` routine
  2. How to transparently integrate the `Access_Table` with the C++ iostream operators

Structure of the Adapter Pattern



Using the Adapter Pattern



Dynamic Array

- Defines a variable-sized array for use by the `Access_Table`

```
template <class T>
class Array
{
public:
  typedef T TYPE; // Type "trait"

  Array (size_t size = 0);
  int init (size_t size);
  T &operator[](size_t index);
  size_t size (void) const;
  // ...

private:
  T *array_;
  size_t size_;
};
```


The Access_Table Class

- Efficiently maps indices onto elements in the data buffer

```
template <class T>
class Access_Table
{
public:
  // Factory Method for initializing Access_Table.
  virtual int make_table (size_t num_lines,
                          char *buffer) = 0;
  // Release buffer memory.
  virtual ~Access_Table (void) { delete [] buffer_; }

  // Retrieve reference to <indexth> element.
  T &element (size_t index) {
    return access_array_[index];
  }

  // Length of the access_array.
  size_t length (void) const {
    return access_array_.size ();
  }
protected:
  Array<T> access_array_; // Access table is array of T.
  char *buffer_; // Hold the data buffer.
};
```


The Sort_AT_Adapter Class

- Adapts the Access_Table to conform to the `ARRAY` interface expected by `sort`

```
struct Line_Ptrs {
  // Comparison operator used by sort().
  int operator< (const Line_Ptrs &);

  // Beginning of line and field/column.
  char *bol_, *bof_;
};

class Sort_AT_Adapter :
  // Note the use of the "Class form" of the Adapter
  private Access_Table<Line_Ptrs> {
public:
  virtual int make_table (size_t num_lines, char *buffer);

  typedef Line_Ptrs TYPE; // Type "trait".

  // These methods adapt Access_Table methods...
  T &operator[] (size_t index) {
    return element (index);
  }

  size_t size (void) const { return length (); }
};
```


Centralizing Option Processing

- *Problem*
  - Command-line options must be global to many parts of the sort program

- *Key forces*
  - Unrestricted use of global variables increases system coupling and can violate encapsulation
  - Initialization of static objects in C++ can be problematic

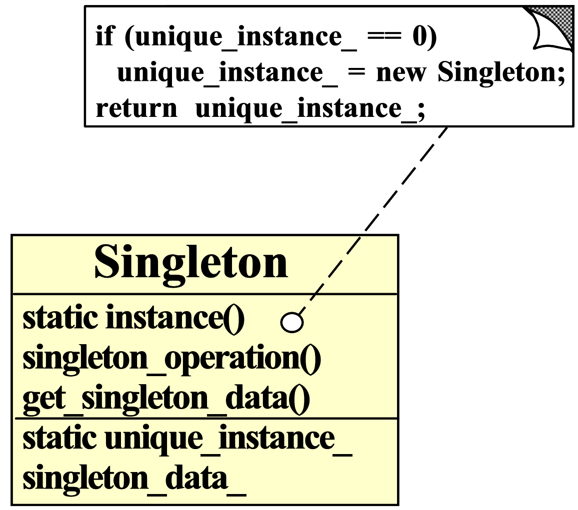
- *Solution*
  - Use the *Singleton* pattern to centralize option processing

Singleton Pattern

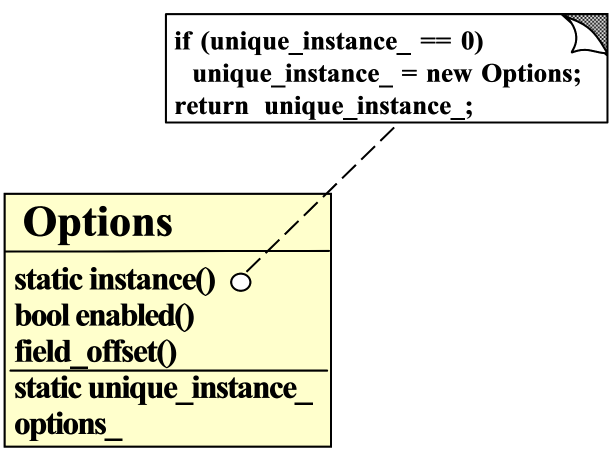
- *Intent*
  - "Ensure a class has only one instance, and provide a global point of access to it"

- This pattern resolves the following forces:
  1. Localizes the creation and use of "global" variables to well-defined objects
  2. Preserves encapsulation
  3. Ensures initialization is done after program has started and only on first use
  4. Allow transparent subclassing of Singleton implementation

Structure of the Singleton Pattern



Using the Singleton Pattern



Options Class

- This manages globally visible options

```
class Options
{
public:
  static Options *instance (void);
  void parse_args (int argc, char *argv[]);

  // These options are stored in octal order
  // so that we can use them as bitmasks!
  enum Option { FOLD = 01, NUMERIC = 02,
              REVERSE = 04, NORMAL = 010 };
  enum Pivot_Strategy { MEDIAN, RANDOM, FIRST };

  bool enabled (Option o);

  int field_offset (void); // Offset from BOL.
  Pivot_Strategy pivot_strat (void);
  int (*compare) (const char *l, const char *r);

protected:
  Options (void); // Ensure Singleton.

  u_long options_; // Maintains options bitmask...
  int field_offset_;
  static Options *instance_; // Singleton.
};
```

Using the Options Class

- The following is the comparison operator used by `sort`

```
int
Line_Ptrs::operator< (const Line_Ptrs &rhs)
{
  Options *options = Options::instance ();

  if (options->enabled (Options::NORMAL))
    return strcmp (this->bof_, rhs.bof_) < 0;

  else if (options->enabled (Options::FOLD))
    return strcasecmp (this->bof_, rhs.bof_) < 0;

  else
    // assert (options->enabled (Options::NUMERIC));
    return  numcmp (this->bof_, rhs.bof_) < 0;
}
```

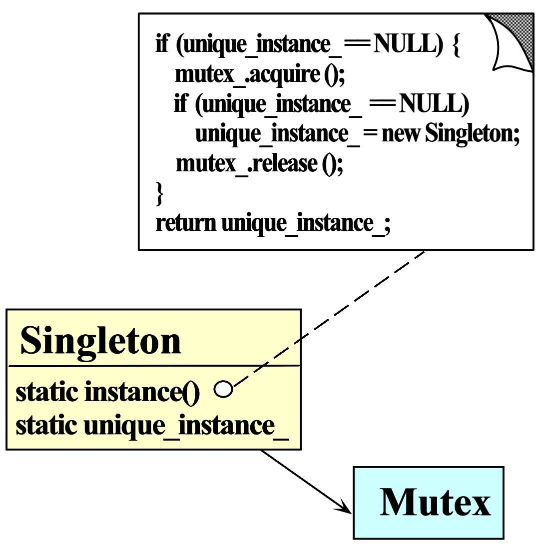

Efficiently Avoiding Race Conditions for Singleton Initialization

- *Problem*
  - A multi-threaded program might have execute multiple copies of `sort` in different threads
- *Key forces*
  - Subtle race conditions can cause Singletons to be created multiple times
  - Locking every access to a Singleton can be too costly
- *Solution*
  - Use the *Double-Checked Locking Optimization* pattern to efficently avoid race conditions when initialization Singletons

The Double-Checked Locking Optimization Pattern

- *Intent*
  - *Ensures atomic initialization or access to objects and eliminates unnecessary locking overhead*
- This pattern resolves the following forces:
  1. Ensures atomic initialization or access to objects, regardless of thread scheduling order
  2. Keeps locking overhead to a minimum
     - *e.g.*, only lock on first access, rather than for the entire Singleton `instance()` method

Structure of the Double-Checked Locking Optimization Pattern



Using the Double-Checked Locking Optimization Pattern

- Uses the Adapter pattern to turn ordinary classes into Singletons optimized automatically with the Double-Checked Locking Optimization pattern

```
template <class TYPE, class LOCK>
class Singleton {
public:
  static TYPE *instance (void);
protected:
  static TYPE *instance_;
  static LOCK lock_;
};

template <class TYPE, class LOCK> TYPE *
Singleton<TYPE, LOCK>::instance (void) {
  // Perform the Double-Check.
  if (instance_ == 0) {
    Guard<LOCK> mon (lock_);
    if (instance_ == 0) instance_ = new TYPE;
  }
  return instance_;
}
```

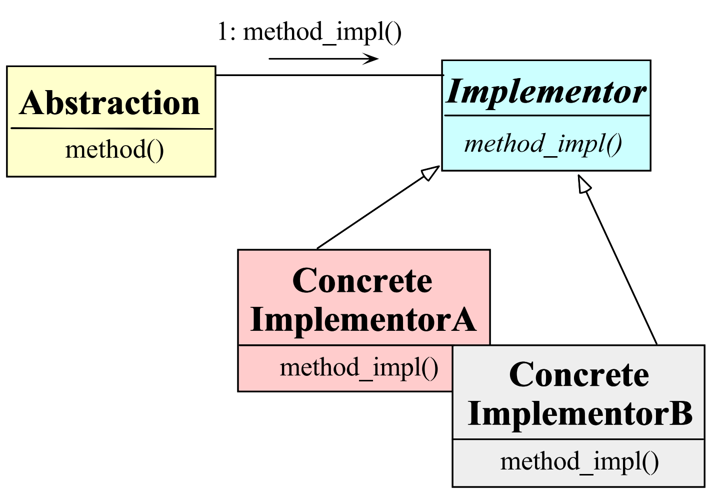

Simplifying Comparisons

- *Problem*
  - The comparison operator shown above is somewhat complex
- *Forces*
  - It's better to determine the type of comparison operation during the initialization phase
  - But the interface shouldn't change
- *Solution*
  - Use the *Bridge pattern* to separate interface from implementation

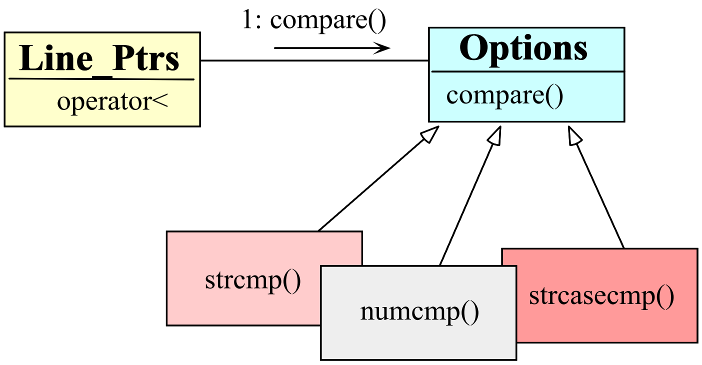
The Bridge Pattern

- *Intent*
  - "Decouple an abstraction from its implementation so that the two can vary independently"
- This pattern resolves the following forces that arise when building extensible software
  1. *How to provide a stable, uniform interface that is both closed and open, i.e.,*
     - *Closed* to prevent direct code changes
     - *Open* to allow extensibility
  2. *How to simplify the Line_Ptrs::operator< implementation*

Structure of the Bridge Pattern



Using the Bridge Pattern



Using the Bridge Pattern

- The following is the comparison operator used by `sort`

```
int
Line_Ptrs::operator<(const Line_Ptrs &rhs)
{
  return (*Options::instance ()->compare)
            (bof_, rhs.bof_);
}
```

- This solution is much more concise
- However, there's an extra level of function call indirection...
  - Which is equivalent to a virtual function call

Initializing the Comparison Operator

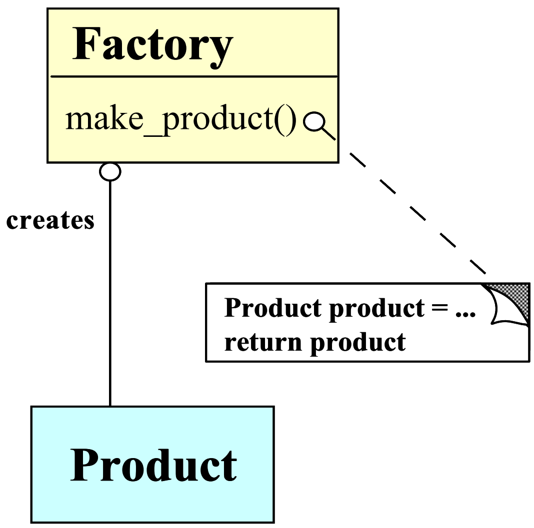
- *Problem*
  - How does the `compare` pointer-to-method get assigned?

    int (*compare) (**const char** *left, **const char** *right);
- *Forces*
  - There are many different choices for `compare`, depending on which options are enabled
  - We only want to worry about initialization details in one place
  - Initialization details may change over time
  - We'd like to do as much work up front to reduce overhead later on
- *Solution*
  - Use a *Factory* pattern to initialize the comparison operator

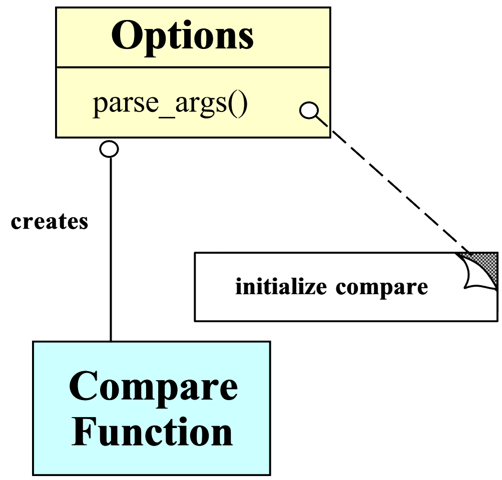
The Factory Pattern

- *Intent*
  - "Centralize the assembly of resources necessary to create an object"
    - Decouple object creation from object use by localizing creation knowledge
- This pattern resolves the following forces:
  - Decouple initialization of the `compare` operator from its subsequent use
  - Makes it easier to change comparison policies later on
    - *e.g.*, adding new command-line options

Structure of the Factory Pattern



Using of the Factory Pattern for Comparisons



Code for Using the Factory Pattern

- The following initialization is done after command-line options are parsed

```
Options::parse_args (int argc, char *argv[])
{
  Options *options = Options::instance ();
  // ...
  if (options->enabled (Options::NORMAL))
    options->compare = &strcmp;
  else if (options->enabled (Options::FOLD))
    options->compare = &strcasecmp;
  else if (options->enabled (Options::NUMERIC))
    options->compare = &numcmp;
  // ...

int numcmp (const char *s1, const char * s2)
{
  double d1 = strtod (s1, 0), d2 = strtod (s2, 0);

  if (d1 < d2) return -1;
  else if (d1 > d2) return 1;
  else // if (d1 == d2)
    return 0;
}
```

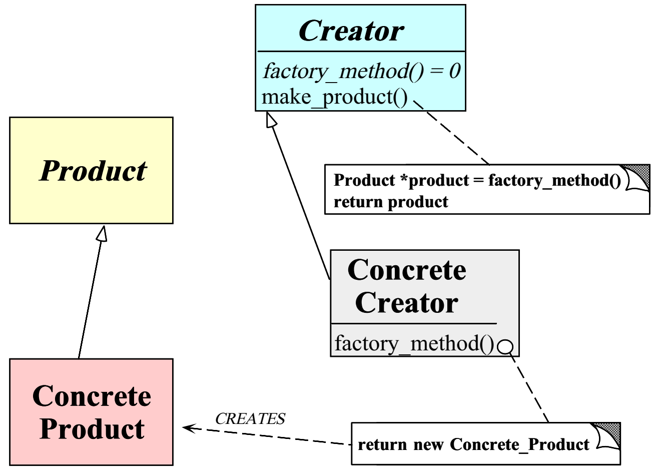

Initializing the Access_Table

- *Problem*
  - One of the nastiest parts of the whole system sort program is initializing the `Access_Table`
- *Key forces*
  - We don't want initialization details to affect subsequent processing
  - Makes it easier to change initialization policies later on
    - *e.g.*, using the Access_Table in non-sort applications
- *Solution*
  - Use the *Factory Method* pattern to initialize the `Access_Table`

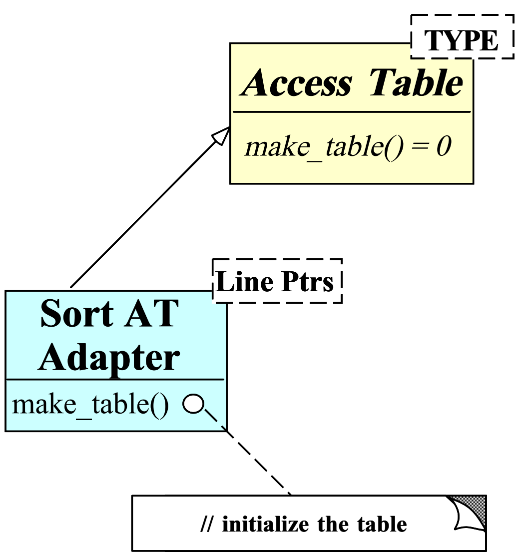
Factory Method Pattern

- *Intent*
  - Define an interface for creating an object, but let subclasses decide which class to instantiate
    - ∗ Factory Method lets a class defer instantiation to subclasses
- This pattern resolves the following forces:
  - Decouple initialization of the `Access_Table` from its subsequent use
  - Improves subsequent performance by pre-caching beginning of each field and line
  - Makes it easier to change initialization policies later on
    - ∗ *e.g.*, adding new command-line options

Structure of the Factory Method Pattern



Using the Factory Method Pattern for Access_Table Initialization



Using the Factory Method Pattern for the Sort_AT_Adapter

- The following iostream Adapter initializes the `Sort_AT_Adapter` access table

```
void operator>> (istream &is,
                 Sort_AT_Adapter &access_table)
{
  Input input;

  // Read entire stdin into buffer.
  char *buffer = input.read (is);

  // Determine number of lines.
  size_t num_lines = input.replaced ();

  // Factory Method initializes Access_Table<Line_Ptrs>.
  access_table.make_table (num_lines, buffer);
}
```

Implementing the Factory Pattern

- The Access_Table_Factory class has a Factory Method that initializes Sort_AT_Adapter

```
// Factory Method initializes Access_Table.
int Sort_AT_Adapter::make_table (size_t num_lines,
                                 char *buffer)
{
  // Array assignment op.
  this->access_array_.resize (num_lines);
  this->buffer_ = buffer; // Obtain ownership.

  size_t count = 0;

  // Iterate through the buffer and determine
  // where the beginning of lines and fields
  // must go.

  for (Line_Ptrs_Iter iter (buffer, num_lines);
       iter.is_done () == 0;
       iter.next ())
  {
    Line_Ptrs line_ptr = iter.current_element ();
    this->access_array_[count++] = line_ptr;
  }
}
```


Initializing the Access_Table with Input Buffer

- *Problem*
  - We'd like to initialize the Access_Table *without* having to know the input buffer is represented
- *Key force*
  - Representation details can often be decoupled from accessing each item in a container or collection
- *Solution*
  - Use the *Iterator* pattern to scan through the buffer

Iterator Pattern

- *Intent*
  - Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation
- The Iterator pattern provides a way to initialize the Access_Table without knowing how the buffer is represented:

```
Line_Ptrs_Iter::Line_Ptrs_Iter
  (char *buffer, size_t num_lines);

Line_Ptrs
Line_Ptrs_Iter::current_element (void)
{
  Line_Ptrs lp;

  // Determine beginning of next line and next field...
  lp.bol_ = // .....
  lp.bof_ = // .....

  return lp;
}
```


Iterator Pattern (cont'd)

- The Iterator pattern also provides a way to print out the sorted lines without exposing representation

```
void operator<< (ostream &os,
 const Sort_AT_Adapter &at)
{
  if (Options::instance ()->enabled (Options::REVERSE))
    for (size_t i = at.size (); i > 0; i--)
      os << at[i - 1].bol_;

  else
    for (size_t i = 0; i < at.size (); i++)
      os << at[i].bol_;
}
```

- Note that STL is heavily based on iterators

Summary of System Sort Case Study

- This case study illustrates using OO techniques to structure a modular, reusable, and highly efficient system
- Design patterns help to resolve many key forces
- Performance of our system sort is comparable to existing UNIX system sort
  - Use of C++ features like *parameterized types* and *inlining* minimizes penalty from increased modularity, abstraction, and extensibility

Case Study 3: Sort Verifier

- *Verify whether a sort routine works correctly*
  - *i.e.*, output of the sort routine must be an ordered permutation of the original input
- This is useful for checking our system sort routine!
  - The solution is harder than it looks at first glance...
- As before, we'll examine the key forces and discuss design patterns that resolve the forces

General Form of Solution

- The following is a general use-case for this routine:

```
template <class ARRAY> void
sort (ARRAY &a);

template <class ARRAY> int
check_sort (const ARRAY &o, const ARRAY &p);

int main (int argc, char *argv[])
{
  Options::instance ()->parse_args (argc, argv);

  Input_Array input;
  Input_Array potential_sort;

  cin >> input;

  copy (input, potential_sort);
  sort (potential_sort);

  if (check_sort (input, potential_sort) == -1)
    cerr << "sort failed" << endl;
  else
    cout << "sort worked" << endl;
}
```


Common Problems

unsorted	7	13	4	15	18	13	8	4
sorted, but not permuted	0	0	0	0	0	0	0	0
permuted, but not sorted	8	13	18	15	4	13	4	7
sorted and permuted	4	4	7	8	13	13	15	18

- Several common problems:
  - Sort routine may zero out data
    - * though it will appear sorted...;-)
  - Sort routine may fail to sort data
  - Sort routine may erroneously add new values

Forces

- Solution should be both time and space efficient
  - *e.g.*, it should not take more time to check than to sort in the first place!
  - Also, this routine may be run many times consecutively, which may faciliate certain space optimizations
- We cannot assume the existence of a "correct" sorting algorithm...
  - Therefore, to improve the chance that our solution is correct, it must be simpler than writing a correct sorting routine
    - Quis costodiet ipsos custodes?

Forces (cont'd)

- Multiple implementations will be necessary, depending on properties of the data being examined, *e.g.*,
  1. if data values are small (in relation to number of items) and integrals use ...
  2. if data has no duplicate values use ...
  3. if data has duplicate values use ...
- This problem illustrates a simple example of "program families"
  - *i.e.*, we want to reuse as much code and/or design across multiple solutions as possible

Strategies

- Implementations of search structure vary according to data, *e.g.*,
  1. *Range Vector*
     - O(N) time complexity and space efficient for sorting "small" ranges of integral values
  2. *Binary Search* (version 1)
     - O(n log n) time complexity and space efficient but does not handle duplicates
  3. *Binary Search* (version 2)
     - O(n log n) time complexity, but handles duplicates
  4. *Hashing*
     - O(n) best/average case, but O(n2) worst case, handles duplicates, but potentially not as space efficient

General OOD Solution Approach

- Identify the "objects" in the application and solution space
  - *e.g.*, use a *search structure* ADT organization with member function such as `insert` and `remove`
- Recognize common design patterns
  - *e.g.*, Strategy, Template Method, and Factory Method
- Implement a framework to coordinate multiple implementations
  - *e.g.*, use classes, parameterized types, inheritance and dynamic binding

General OOD solution approach (cont'd)

- C++ framework should be amenable to:
  - *Extension and Contraction*
    - May discover better implementations
    - May need to conform to resource constraints
    - May need to work on multiple types of data
  - *Performance Enhancement*
    - May discover better ways to allocate and cache memory
    - Note, improvements should be transparent to existing code...
  - *Portability*
    - May need to run on multiple platforms

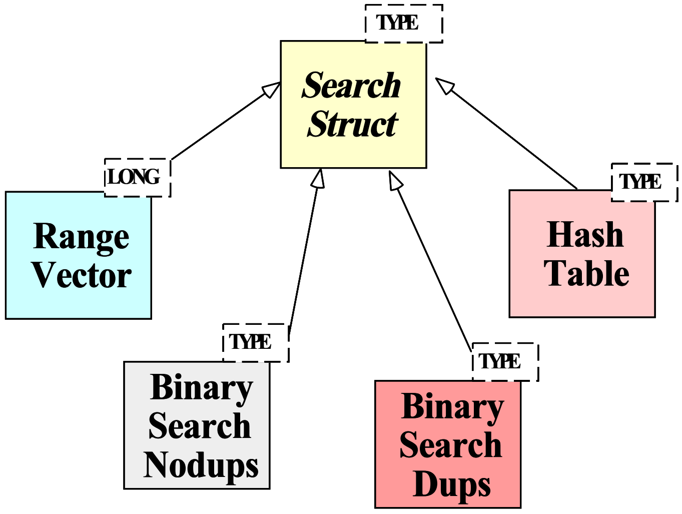
High-level Algorithm

- *e.g.*, pseudo code

template <**class** ARRAY>
int check_sort (const ARRAY &original,
        const ARRAY &potential_sort)
{
    Perform basic sanity check to see if the `potential_sort` *is actually in order (can also detect duplicates here)*

    if *basic sanity check succeeds* **then**
        Initialize search structure `srchstrct`
        for i $\leftarrow$ 0 **to size** $-$ 1 **loop**
            insert (`potential_sort[i]`)
                into `srchstrct`
        for i $\leftarrow$ 0 **to size** $-$ 1 **loop**
            if *remove* (`original[i]`) *from*
                `srchstrct` *fails* **then**
                return ERROR
        return SUCCESS
    else
        return ERROR
    end if
}

C++ Class Model



C++ Class Interfaces

- Search structure base class.

```
template <class T>
class Search_Struct
{
public:
  virtual int insert (const T &new_item) = 0;
  virtual int remove (const T &existing_item) = 0;
  virtual ~Search_Struct (void) = 0;
};
```

- Strategy Factory class

```
template <class ARRAY>
Search_Strategy
{
public:
  // Singleton method.
  static Search_Strategy *instance (void);

  // Factory Method
  virtual Search_Struct<ARRAY::TYPE> *
    make_strategy (const ARRAY &);
};
```

C++ Class Interfaces (cont'd)

- Strategy subclasses

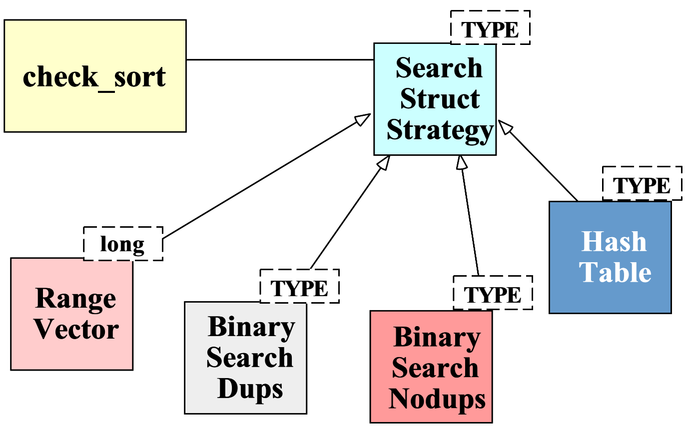
```
// Note the template specialization
class Range_Vector : public Search_Struct<long>
{ typedef long TYPE; /* ... */ };

template <class ARRAY>
class Binary_Search_Nodups : public Search_Struct<ARRAY::TYI
{
  typedef T TYPE; /* ... */
};

template <class ARRAY>
class Binary_Search_Dups : public Search_Struct<ARRAY::TYPE>
{
  typedef T TYPE; /* ... */
};

template <class T>
class Hash_Table : public Search_Struct<T>
{
  typedef T TYPE; /* ... */
};
```


Design Patterns in Sort Verifier

- Factory Method
  - "Define an interface for creating an object, but let subclasses decide which class to instantiate"
    - * Factory Method lets a class defer instantiation to subclasses

- In addition, the *Facade*, *Iterator*, *Singleton*, and *Strategy* patterns are used

Using the Strategy Pattern



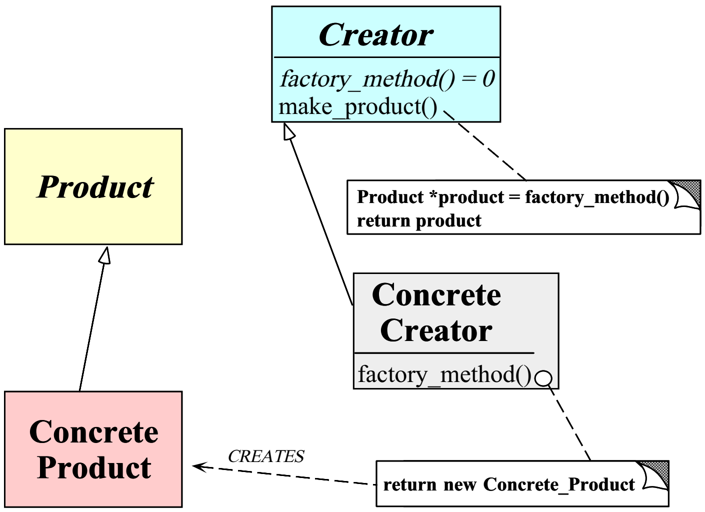
- This pattern extends the strategies for checking if an array is sorted without modifying the `check_sort` algorithm

The Factory Method Pattern

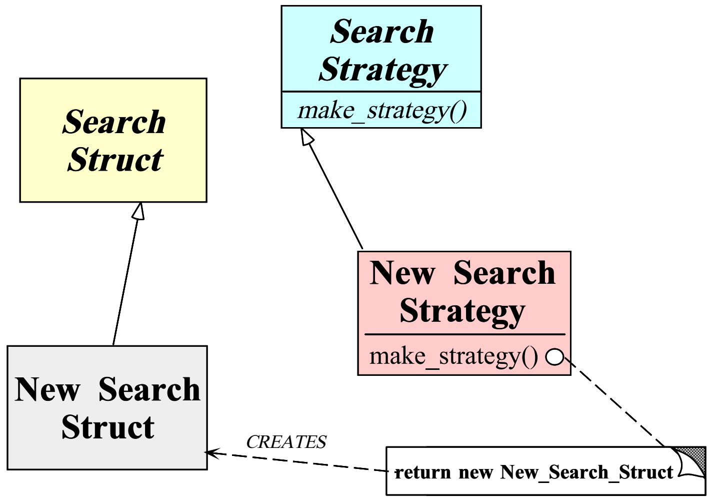
- *Intent*
  - Define an interface for creating an object, but let subclasses decide which class to instantiate
    - * Factory Method lets a class defer instantiation to subclasses

- This pattern resolves the following force:
  1. *How to extend the initialization strategy in the sort verifier transparently*

Structure of the Factory Method Pattern



Using the Factory Method Pattern



Implementing the check_sort Function

- *e.g.*, pseudo-code for the sort verification strategy

```
template <class ARRAY> int
check_sort (const ARRAY &orig, const ARRAY &p_sort)
{
  if (orig.size () != p_sort.size ())
    return -1;

  auto_ptr < Search_Struct<ARRAY::TYPE> > ss =
    Search_Strategy<ARRAY>::instance ()->make_strategy
      (p_sort);

  for (int i = 0; i < p_sort.size (); i++)
    if (ss->insert (p_sort[i]) == -1)
      return -1;

  for (int i = 0; i < orig.size (); i++)
    if (ss->remove (orig[i]) == -1)
      return -1;

  return 0;
  // auto_ptr's destructor deletes the memory...
}
```


Initializing the Search Structure

```
template <class ARRAY> Search_Struct<ARRAY::TYPE> *
Search_Strategy<ARRAY>::make_strategy
                         (const ARRAY &potential_sort)
{
  int duplicates = 0;

  for (size_t i = 1; i < size; i++)
    if (potential_sort[i] < potential_sort[i - 1])
      return 0;
    else if (potential_sort[i] == potential_sort[i - 1])
      duplicates++;

  if (duplicates == 0)
    return new Binary_Search_Nodups<ARRAY>
                (potential_sort);
  else if (size % 2)
    return new Binary_Search_Dups<ARRAY>
                (potential_sort, duplicates)
  else return new Hash_Table<ARRAY::TYPE>
                   (size, &hash_function);
}
```

Specializing the Search Structure for Range Vectors

```
template <Array<long> > Search_Struct<long> *
Search_Strategy<Array<long> >::make_strategy
  (const Array<long> &potential_sort)
{
  int duplicates = 0;

  for (size_t i = 1; i < size; i++)
    if (potential_sort[i] < potential_sort[i - 1])
      return 0;
    else if (potential_sort[i] == potential_sort[i - 1])
      duplicates++;

  long range = potential_sort[size - 1] -
            potential_sort[0];
  if (range <= size)
    return new Range_Vector (potential_sort[0],
                             potential_sort[size - 1])
  else if (duplicates == 0)
    return new Binary_Search_Nodups<long>
               (potential_sort);
  else if (size % 2)
    return new Binary_Search_Dups<long>
               (potential_sort, duplicates)
  else return new Hash_Table<long>
                  (size, &hash_function);
}
```


Summary of Sort Verifier Case Study

- The sort verifier illustrates how to use OO techniques to structure a modular, extensible, and efficient solution
  - The main processing algorithm is simplified
  - The complexity is pushed into the strategy objects and the strategy selection factory
  - Adding new solutions does not affect existing code
  - The appropriate ADT search structure is selected at run-time based on the Strategy pattern