Object-Oriented Design Case Studies with Patterns & C++

Douglas C. Schmidt

Professor
d.schmidt@vanderbilt.edu
Department of EECS
Vanderbilt University

www.dre.vanderbilt.edu/~schmidt/
(615) 343-8197

Case Studies Using Patterns

- The following slides describe several case studies using C++ & patterns to build highly extensible software

- The examples include
  1. Expression Tree
     - e.g., Adapter, Factory, Bridge
  2. System Sort
     - e.g., Facade, Adapter, Iterator, Singleton, Factory Method, Strategy, Bridge
  3. Sort Verifier
     - e.g., Strategy, Factory Method, Facade, Iterator, Singleton

Case Study: Expression Tree Evaluator

- The following inheritance & dynamic binding example constructs expression trees

  - Expression trees consist of nodes containing operators & operands
    - Operators have different precedence levels, different associativities, & 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 Behavior

Expression 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.,
  * Return the "value" of the expression tree.
  * Generate code.
  * Perform semantic analysis.

Print_Tree Function

A typical algorithmic implementation uses a switch statement & a recursive function to build & evaluate a tree, e.g.,

```c
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 ("\n");
    } break;
    case BINARY: {
      printf ("%s", root->op_[0]);
      print_tree (root->binary_.l_);
      printf ("%s", root->op_[0]);
      print_tree (root->binary_.r_);
      printf ("\n");
    } break;
    default: printf ("error, unknown type\n");
  }
}
```

Algorithmic Version

A typical algorithmic method for implementing expression trees involves using a struct/union to represent data structure, e.g.,

```c
typedef struct Tree_Node Tree_Node;
struct Tree_Node {
  enum { NUM, UNARY, BINARY } tag_;
  short use_; /* reference count */
  union {
    char op_[2];
    int num;
  } 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_;
```
**Limitations with Algorithmic Approach**

- Problems or limitations with the typical algorithmic approach include
  - Little or no use of encapsulation
- Incomplete modeling of the application domain, which results in
  1. Tight coupling between nodes & 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!
  3. Data structures are "passive" & functions do most processing work explicitly

**More Limitations with Algorithmic Approach**

- The program organization makes it difficult to extend, *e.g*.,
  - Any small changes will ripple through the entire design & implementation
    - *e.g.*, see the "ternary" extension below
    - Easy to make mistakes switching on type tags . . .
- Solution wastes space by making worst-case assumptions *wrt* structs & unions
  - 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 algorithmic 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
  - Discover 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
      - class Binary_Node: handles binary operators, *e.g.*, a + b, 10 - 30
    - * class Tree: "glue" code that describes expression-tree edges, *i.e.*, relations between Nodes
  - Note, these classes model entities in the application domain
    - *i.e.*, nodes & edges (vertices & arcs)
**OO Pattern Examples**

**Design Patterns in the Expression Tree Program**

- **Factory**
  - *Centralize the assembly of resources necessary to create an object*  
    *e.g.*, decouple Node subclass initialization from use

- **Bridge**
  - *Decouple an abstraction from its implementation so that the two can vary independently*  
    *e.g.*, printing contents of a subtree and managing memory

- **Adapter**
  - *Convert the interface of a class into another interface clients expect*  
    *e.g.*, make Tree conform C++ iostreams

---

**C++ Node Interface**

```cpp
class Tree; // Forward declaration

// Describes the Tree vertices
class Node {
  friend class Tree;
  protected: // Only visible to derived classes
    Node () : use_ (1) {} 

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

    // Important to make destructor virtual!
    virtual ~Node () {
      int use_; // Reference counter.
    };
```

**C++ Tree Interface**

```cpp
#include "Node.h"

// Bridge class that describes the Tree edges and acts as a Factory.
class Tree {
  public:
    // Factory operations
    Tree (int);
    Tree (const string &, Tree &);
    Tree (const string &, Tree &, Tree &);
    Tree (const Tree &);
    void operator= (const Tree &);
    void print (std::ostream & ) const;
  private:
    Node *node_; // pointer to a rooted subtree
  
```
C++ Int_Node Interface

```cpp
#include "Node.h"

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

C++ Unary_Node Interface

```cpp
#include "Node.h"

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

C++ Binary_Node Interface

```cpp
#include "Node.h"

class Binary_Node : public Node {
public:
    Binary_Node (const string &op,  
                 const Tree &t1,  
                 const Tree &t2); 
    virtual void print (std::ostream &s) const;
private:
    const string operation_;  
    Tree left_;  
    Tree right_;  
};
```
C++ Int_Node Implementations

```cpp
#include "Int_Node.h"

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

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

C++ Unary_Node Implementations

```cpp
#include "Unary_Node.h"

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

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

C++ Binary_Node Implementation

```cpp
#include "Binary_Node.h"

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

void Binary_Node::print (std::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 & 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 generalization of the GoF Factory Method pattern

Structure of the Factory Pattern

```
Factory
    make_product() ->

Product
```

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

Tree::Tree (const string &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, & dynamic binding, etc.
  - We don’t want to expose dynamic memory management details to application developers
- **Solution**
  - Use the Bridge pattern to shield the use of inheritance & 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 & open**, i.e.,
     - interface is *closed* to prevent direct code changes
     - Implementation is *open* to allow extensibility
  2. **How to manage dynamic memory more transparently & robustly**
  3. **How to simplify the implementation of operator<<**

Using the Bridge Pattern

Illustrating the Bridge Pattern in C++

- The Bridge pattern is used for printing expression trees:
  ```cpp
  void Tree::print (std::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++ istd::ostream operators

---

Structure of the Adapter Pattern

1: request()

client → Target

request()

Adapter

request()

Adaptee

specific_request()

2: specific_request()

---

Using the Adapter Pattern

1: operator<<

client → Target

operator<<

iostream

operator<<

2: print()

Tree

print()
Using the Adapter Pattern

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

```cpp
std::ostream &operator<< (std::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

```cpp
Tree::Tree (const Tree &t): node_ (t.node_) {
    // Sharing, ref-counting.
}
```

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

C++ Tree Implementation (cont’d)

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

C++ Main Program

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

int main (int, char *[]) {
    const Tree t1 = Tree("-", Tree("-", 5), Tree("+", 3, 4));
    cout << t1 << endl;  // prints (-5) * (3 + 4)  
    const Tree t2 = Tree("+", t1, t1);  
    // prints (((-5) * (3 + 4)) * ((-5) * (3 + 4))).
    cout << t2 << endl;  
    return 0;  
    // Destructors of t1 & t2 recursively
}  // delete entire tree when leaving scope.
```
Adding Ternary Nodes

Extending the existing program to support ternary operations:
- "if" statements, ternary operators, etc.

```cpp
#include "Node.h"

class Ternary_Node: public Node {
public:

    Ternary_Node(const string &op, const Tree &a, const Tree &b, const Tree &c);

    void print(std::ostream &stream) const;

private:

    const string operation_;
    Tree left_, middle_, right_;};

void Ternary_Node::print(std::ostream &stream) const {
    stream << this->operation_ << (this->left_ 
function recursive call

stream << this->middle_ 
function recursive call

stream << this->right_ 
function recursive call

stream << ";";
}

```
C++ Ternary_Node Implementation (cont’d)

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

Differences from Algorithmic Implementation

• On the other hand, modifying the original algorithmic approach requires
  changing (1) the original data structures, e.g.,

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

Differences from Algorithmic Implementation (cont’d)

• & (2) 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”
    & relations in expression trees
• Use of C++ language features simplifies the design and facilitates
  extensibility
  – e.g., implementation follows directly from design
• Use of patterns helps to motivate, justify, & generalize design choices
Potential Problems with OO Design

- Solution is very “data structure rich”
  - e.g., requires configuration management to handle many headers & .cpp files!
- May be somewhat less efficient than original algorithmic 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: 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
- 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 (e.g., ASCII)
- The ordering is affected globally by the following options:
  - Ignore case (-f)
  - Sort numerically (-n)
  - Sort in reverse (-r)
  - Begin sorting at a specified field (-k)
  - Begin sorting at a specified column (-c)
- Your program need not sort files larger than main memory

High-level Forces

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

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

```cpp
// Reusable function:
// template<typename ARRAY> void sort (ARRAY &a);

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

    cin >> input;
    sort (input);
    cout << input;
}
```

General OOD Solution Approach

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

Top-level Algorithmic View of the Solution (cont’d)

- Avoid the grand mistake of using top-level algorithmic view to structure the design . . .
  - Structure the design to resolve the forces!
  - Don’t focus on algorithms or data, but instead look at the problem, its participants, & their interactions!

C++ Class Model
C++ Class Components

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

- **Strategic components**
  - System_Sort
    - Facade that integrates everything . .
  - Sort_AT_Adapter
    - Integrates Array & Access_Table
  - Options
    - Manages globally visible options
  - Sort
    - * e.g., both quicksort & insertion sort

Detailed Format for Solution

- Note the separation of concerns

```cpp
// Prototypes
template <typename ARRAY> void sort (ARRAY &a);
void operator>>(std::istream &, Sort_AT_Adapter &);
void operator<<(std::ostream &, const Sort_AT_Adapter &);

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 1/2 size of main memory)

- **Forces**
  - To improve performance solution must minimize:
    1. Data copying & data manipulation
    2. Dynamic memory allocation

- **Solution**
  - Create an Input class that reads arbitrary input efficiently
**The Input Class**

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

```cpp
class Input {
public:
    // Reads from <input> up to <terminator>, replacing <search> with <replace>. Returns dynamically allocated buffer.
    char *read (std::istream &input, int terminator = EOF, int search = '\n', int replace = '\0');
    // Number of bytes replaced.
    size_t replaced () const;
    // Size of buffer.
    size_t size () const;
private:
    // Recursive helper method.
    char *recursive_read ();
};
```

**The Input Class (cont’d)**

```cpp
class Input {
    // Initialize all the data members...
    return recursive_read ();
}
```

```cpp
char *Input::recursive_read () {
    char buffer[BUFSIZE];
    // 1. Read input one character at a time, performing search/replace until EOF is reached or buffer is full.
    // 1.a If buffer is full, invoke recursive_read() recursively.
    // 1.b If EOF is reached, dynamically allocate chunk large enough to hold entire input
    // 2. On way out of recursion, copy buffer into chunk
}
```

**Design Patterns in the 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()` function 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` & `iostreams`
**Factory**
- Centralize assembly of resources needed to create objects
  - *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, & make them interchangeable
  - *e.g.*, allow flexible pivot selection

---

**Sort Algorithm**

- For efficiency, two types of sorting algorithms are used:
  1. *Quicksort*
     - Highly time & space efficient sorting arbitrary data
     - $O(n \log n)$ average-case time complexity
     - $O(n^2)$ worst-case time complexity
     - $O(\log n)$ space complexity
     - Optimizations are used to avoid worst-case behavior
  2. *Insertion sort*
     - Highly time & space efficient for sorting “almost ordered” data
     - $O(n)$ average- & 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, & make them interchangeable
    * Strategy lets the algorithm vary independently from clients that use it
- **Solution**
  - Use the *Strategy* pattern to select the pivot selection algorithm
  - Define a family of algorithms, encapsulate each one, & make them interchangeable
    * Strategy lets the algorithm vary independently from clients that use it

Structure of the Strategy Pattern

```
Context
  context_interface()

  Strategy
    algorithm_interface()

    Concrete Strategy A
      algorithm_interface()

    Concrete Strategy B
      algorithm_interface()

    Concrete Strategy C
      algorithm_interface()
```

Using the Strategy Pattern

```
quick_sort

  > pivot_strat->get_pivot (array, lo, hi)

  Select First

  Random

  Median of Three
```

Vanderbilt University
Implementing the Strategy Pattern

- ARRAY is the particular “context”

```cpp
template <typename ARRAY>
void sort (ARRAY &array) {
  Pivot_Strategy<ARRAY> *pivot_strat =
    Pivot_Factory<ARRAY>::make_pivot
    (Options::instance ()->pivot_strat ());
  std::auto_ptr <Pivot_Strategy<ARRAY>> holder (pivot_strat);

  // Ensure exception safety.
  ARRAY temp = array;
  quick_sort (temp, pivot_strat);
  // Destructor of <holder> deletes <pivot_strat>.
  array = temp;
}
```

Fixed-size Stack

- Defines a fixed size stack for use with non-recursive quicksort

```cpp
template <typename T, size_t SIZE>
class Fixed_Stack
{
  public:
    bool push (const T &new_item);
    bool pop (T &item);
    bool is_empty ();
    // . . .

  private:
    T stack_[SIZE];
    size_t top_;
};
```

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 & reuse
  - Conceptually, sorting only makes a few assumptions about the “array” it sorts
    - e.g., supports operator[] methods, size, & trait TYPE
  - We don’t want to arbitrarily limit types of arrays we can sort

- **Solution**
  - Use the Facade & 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[] & size` methods, & element `TYPE`
    2. Allows the implementation to be efficient and arbitrarily complex without affecting clients

**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 & 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, & provide a global point of access to it

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

Using the Singleton Pattern

```cpp
if (unique_instance_ == 0) {
    unique_instance_ = new Singleton;
    return unique_instance_;}
```

Options Class

- This manages globally visible options

```cpp
class Options {
public:
    static Options *instance();
    bool 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 };
```
Options Class (cont’d)

bool enabled (Option o);

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

protected:
Options (); // Ensure Singleton.

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

#define SET_BIT(WORD, OPTION) (WORD |= OPTION)
#define CLR_BIT(WORD, OPTION) (WORD &~ OPTION)

bool Options::parse_args (int argc, char *argv[]) {
for (int c;
(c = getopt (argc, argv, "nrfs:k:c:t:")) != EOF; ) {
    switch (c) {
    case 'n': {
        CLR_BIT (options_, Options::FOLD);
        CLR_BIT (options_, Options::NORMAL);
        SET_BIT (options_, Options::NUMERIC);
        break;
    } // ...
    case 'r':
        break;
    default:
        optarg = argv [c];
        break;
    }
}
}

Using the Options Class

• One way to implement sort() comparison operator:

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

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

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

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

• We’ll see another approach later on using Bridge

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 & open, i.e.,
     - Closed to prevent direct code changes
     - Open to allow extensibility
  2. How to simplify the `Line_Ptrs::operator<` implementation & reference counting for `Access_Table` buffer

Using the Bridge Pattern

- The following is the comparison operator used by `sort`
  ```cpp
  int Line_Ptrs::operator<(const Line_Ptrs &rhs) const {
    return (*Options::instance ()->compare) (bof_, rhs.bof_) < 0;
  }
  ```
- 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?
  ```c
  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 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++ istd::ostream operators

---

Structure of the Adapter Pattern

---

Using the Adapter Pattern

---
**Dynamic Array**

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

```cpp
template<typename T>
class Array {
public:
    Array (size_t size = 0);
    int init (size_t size);
    T &operator[](size_t index);
    size_t size () const;
    T *begin () const; // STL iterator methods.
    T *end () const;
    // ...
private:
    T *array_;
    size_t size_;
};
```

**The Access_Table Class**

- Efficiently maps indices onto elements in the data buffer

```cpp
template<typename T>
class Access_Table {
public:
    // Factory Method for initializing Access_Table.
    virtual int make_table (size_t lines, char *buffer) = 0;
    // Release buffer memory.
    virtual ~Access_Table ()
    T &element (size_t index); // Reference to <indexth> element.
    size_t length () const; // Length of the access_array.
    Array<T> &array (void) const; // Return reference to array.
protected:
    Array<T> access_array_; // Access table is array of T.
    Access_Table_Impl *access_table_impl_; // Ref counted buffer.
};
```

**The Access_Table_Impl Class**

- Part of the Bridge pattern

```cpp
class Access_Table_Impl {
public:
    Access_Table_Impl (void); // Default constructor
    Access_Table_Impl (char *buffer); // Constructor
    // Virtual destructor ensures subclasses are virtual
    virtual ~Access_Table_Impl ()
    void add_ref (void); // Increment reference count
    void remove_ref (void); // Decrement reference count
    char *get_buffer (void); // Get buffer from the class
    void set_buffer (char *); // Set buffer

private:
    char *buffer_; // Underlying buffer
    size_t ref_count_; // Refcount tracks deletion.
};
```

**The Sort_AT_Adapter Class**

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

```cpp
struct Line_Ptrs {
    // Comparison operator used by sort().
    int operator< (const Line_Ptrs &)
    char *bol_, *bof_; // Beginning of line & field/column.
};
```
The Sort_AT_Adapter Class

```cpp
class Sort_AT_Adapter { // Note 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 ...
            Line_Ptrs &operator[](size_t index);
            size_t size() const;
    };

    // Put these into separate file.
    Line_Ptrs &Sort_AT_Adapter::operator[](size_t i) {
        return element(i);
    }
    size_t Sort_AT_Adapter::size() const {
        return length();
    }
};
```

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

```
Factory
  make_product() o
      creates
Product
  Product product = ... o
      return product
```

Using the Factory Pattern for Comparisons

```
Options
  parse_args() o
      creates
Compare Function
  initialize compare
```

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

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

### 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 & line.
  - Makes it easier to change initialization policies later on.
    - e.g., adding new command-line options.
**Structure of the Factory Method Pattern**

- **Product**
- **ConcreteProduct**
- **Creator**
  - `factory_method() = 0`
  - `make_product()`
  - `Product *product = factory_method()
  - return product`
- **ConcreteCreator**
  - `factory_method()`
  - `return new Concrete_Product`

---

**Using the Factory Method Pattern for Access Table Initialization**

- **Access Table**
  - `make_table() = 0`
- **Sort AT Adapter**
  - `make_table()`
  - `// initialize the table`

---

**Using the Factory Method Pattern for the Sort_AT_Adapter**

- The following `std::ostream` Adapter initializes the `Sort_AT_Adapter` access table.

```cpp
void operator>>(std::istream &is, Sort_AT_Adapter &at)
{
    Input input;
    // Read entire stdin into buffer.
    char *buffer = input.read(is);
    size_t num_lines = input.replaced();

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

---

**Implementing the Factory Method Pattern**

- The `Access_Table.Factory` class has a Factory Method that initializes `Sort_AT_Adapter`.

```cpp
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;
}
```
Implementing the Factory Method Pattern (cont'd)

// Iterate through the buffer & determine
// where the beginning of lines & 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 C++ Standard Library (STL) is heavily based on the iterator pattern,
    e.g.,

```cpp
int main (int argc , char *argv[]) {
    std::vector <std::string> args ;
    for (int i = 1; i < argc ; ++i) {
        args.push_back (std::string (argv [i]));
    }
    for (std::vector<std::string>::iterator j = args.begin () ;
    j != args.end () ; ++j)
        cout << (*j) _ << endl ;
}
```

Iterator Pattern (cont'd)

- The Iterator pattern provides a way to initialize the Access Table without
  knowing how the buffer is represented

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

Line_Ptrs Line_Ptrs_Iter::current_element () {
    Line_Ptrs lp ;

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

    return lp ;
}
```
**Iterator Pattern (cont’d)**

- Iterator provides a way to print out sorted lines

```cpp
void operator<<(std::ostream &os, const Line_Ptrs lp) {
  os << lp.bol_;
}

void operator<<(std::ostream &os, const Sort_AT_Adapter &at) {
  if (Options::instance() -> enabled (Options::REVERSE))
    std::reverse_copy (at.array ().begin (),
                      at.array ().end (),
                      std::ostream_iterator<System_Sort::Line_Ptrs> (os, "\n"));
  else
    std::copy (at.array ().begin (),
                at.array ().end (),
                std::ostream_iterator<System_Sort::Line_Ptrs> (os, "\n"));
}
```

**Summary of System Sort Case Study**

- This case study illustrates using OO techniques to structure a modular, reusable, & 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, & extensibility

**Case Study: 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 & discuss design patterns that resolve the forces

**General Form of Solution**

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

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

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

int main (int argc, char *argv[]) {
  Options::instance () -> parse_args (argc, argv);
  Input original;
  Input potentially_sorted;
}
General Form of Solution (cont’d)

```cpp
cin >> input;

std::copy (original.begin (),
            original.end (),
            potentially_sorted.begin ());
sort (potentially_sorted);
if (check_sort (original, potentially_sorted) == -1)
    cerr << "sort failed" << endl;
else
    cout << "sort worked" << endl;
```

Common Problems

- 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 & 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 facilitate 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 custodiet ipsos custodes?
      - (Who shall guard the guardians?)

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) & 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 & space efficient for sorting “small” ranges of integral values
  2. Binary Search (version 1)
     - $O(n \log n)$ time complexity & 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 $O(n^2)$ worst case, handles duplicates, but potentially not as space efficient

General OOD Solution Approach

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

General OOD solution approach (cont’d)

- C++ framework should be amenable to:
  - Extension & 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 & 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 <typename 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 < 0 to size - 1 loop
        insert (potential_sort[i])
        into srchstrct
    for i < 0 to size - 1 loop
        if remove (original[i]) from srchstrct fails then
            return ERROR
        return SUCCESS
    else
        return ERROR
    end if
}
Using the Strategy Pattern

In addition, the Factory, Facade, Singleton, and Strategy patterns are used:

- **Factory Method**: Define an interface for creating an object, but let subclasses decide which class to instantiate. This lets classes defer instantiation to subclasses.

  ```cpp
class RangeVector {
  public:
    SearchStrategy<long> check_sort;
  
    // other member functions...
};
```

- **Facade**: A design pattern that exposes a simple interface to a more complex subsystem. Allows clients to interact with a complex subsystem using a simpler interface.

  ```cpp
class DesignPatternsInSortVerifier {
  public:
    void check_sort();
    
  private:
    RangeVector _range_vector;
};
```

- **Strategy**: Enables a client to select from a family of algorithms, using a common interface. The Strategy Pattern extends the strategies for checking if an array is sorted without modifying the check_sort algorithm.

  ```cpp
class HashTable {
  public:
    SearchStrategy<T> check_sort;
  
  private:
    template <typename ARRAY, typename T> class HashTable;
    
    // other member functions...
};
```

- **Singleton**: Ensures that a class has only one instance and provides a global point of access to it.

  ```cpp
class Singleton {
  private:
    static Singleton* _instance;
  
  public:
    Singleton();
    Singleton(Singleton const& other) = delete;
    Singleton& operator=(Singleton const& other) = delete;
  
    static Singleton& instance()
    {
      return _instance;
    }
};
```

- **Iterators**: Provide a uniform way of accessing the elements of any container (e.g., vectors, maps).

  ```cpp
class Iterator {
  public:
    void operator++();
  
  private:
    // iterator member functions...
};
```

- **Template Specialization**: This pattern extends the strategies for checking if an array is sorted without modifying the check_sort algorithm.

  ```cpp
class TemplateSpecialization {
  template <typename ARRAY>
  class SearchStrategy {
    public:
      void check_sort(ARRAY const& array);
  
  private:
    // other member functions...
};
```

- **Shrinky Subesces**:

  ```cpp
class ShrinkySubsequences {
  public:
    void shrinky_subsequences();
  
  private:
    // other member functions...
};
```

## Using the Strategy Pattern

```cpp
class BinarySearchDups {
  public:
    SearchStrategy<ARRAY, TYPE> check_sort;
  
  private:
    template <typename T> class HashTable;
    
    // other member functions...
};
```
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*

Using the Factory Method Pattern

- e.g., C++ code for the sort verification strategy

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

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

Implementing the check_sort Function
Implementing the check_sort Function (cont'd)

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

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

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

Initializing the Search Structure

- Factory Method

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

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

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

Initializing the Search Structure (cont'd)

```c++
if (typeid (potential_sort[0]) == typeid (long))
    & range <= size)
    return new Range_Vector (potential_sort[0],
        potential_sort[size - 1])
else 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<typename ARRAY::TYPE>
    (size, &hash_function);
```

Specializing the Search Structure for Range Vectors

```c++
template <Array<long> > Search_Strategy<long> *
Search_Struct<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];
```
Specializing the Search Structure for Range Vectors

```java
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, & efficient solution
  - The main processing algorithm is simplified
  - The complexity is pushed into the strategy objects & 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