Mayfly
A Pattern for Lightweight Generic Interfaces [*†]

Jeremy G. Siek Andrew Lumsdaine

July 21, 1999

Abstract

The Mayfly pattern describes an implementation approach to constructing interfaces for efficient data structures. In Nature, the mayfly is a creature well known for its short life span. Similarly, for our purposes, a Mayfly is a temporary object that resides on the stack or only in registers (never in the heap). All of its member functions are typically inlined and it is always passed by value. These characteristics make Mayfly objects ideal for providing lightweight interfaces to efficient array-based data structures such as compressed matrices, graphs, heaps, and trees. The Mayfly pattern cuts across several other patterns. For instance, many of the iterators in the Standard Template Library (STL [1, 19]) are Mayflies. Among the Mayflies described in this paper will be some Adapter [3] [1] and Aggregate [3, 14] objects.

1 Intent

Implement lightweight interfaces for efficient data structures using small temporary objects.

2 Motivation

Many times the most efficient implementation of an abstract data type is with a concrete representation that is quite different than the abstraction it is representing. For instance, even though a heap is a binary tree, the classical way to efficiently represent a heap is with an array and a special indexing scheme. An example of a heap with the corresponding array representation is shown in Fig 1.

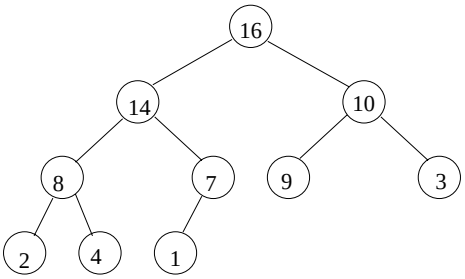
The problem with using such an array representation is that it is inherently low-level and not user-friendly. One must constantly translate the problem-level abstraction (such as "parent") to the array indexed version ($(i-1)/2$). Additionally, one cannot write generic algorithms (ala the STL [19]) using such low level indexing constructs. The implementation details are hard coded into the algorithm, so each algorithm will only work with a particular heap data structure. This same problem shows up time and again in the area of linear algebra, with the wide variety of matrix formats, and also in graph algorithms.

The textbook (e.g., [2]) definition of a heap is that it is an ordered tree, where the ordering is defined by the *heap property*. The heap property states that the value of the parent node p is greater than any of the children nodes, where c is a child.

$$\begin{aligned} value(p) &\geq \max\{value(c) \mid c \in children(p)\} \\ value(c) &\leq value(parent(c)) \end{aligned}$$

[*]This work was supported by NSF grants ASC94-22380 and CCR95-02710.

[1]There are the two alternate spellings "adapter" and "adaptor". It seems the design patterns community prefers to use the "e", while the generic programming community prefers to use the "o". For this paper we adopt the spelling "adapter".



$$
\begin{aligned}
parent(i) &= (i-1)/2 \\
rightChild(i) &= 2i+1 \\
leftChild(i) &= 2i+2
\end{aligned}
$$

16	14	10	8	7	9	3	2	4	1

Figure 1: Representation of a binary tree with an array.

This ordering makes the heap an efficient method for implementing a priority queue.

There are two core procedures that are used to operate on a heap: *push-heap* and *pop-heap*, which respectively add and remove elements from the heap. The implementation of these operations must update the heap so that the heap property is maintained. There are two operations that carry out the task of maintaining the heap. The first is *down-heap* which moves an element of the heap from the top towards the bottom. The second is *up-heap* which moves an element from the bottom towards the top. Below is the canonical STL implementation of *push-heap* (which just does an *up-heap*).

```
template <class RandomAccessIterator, class Distance, class T>
void __push_heap(RandomAccessIterator first, Distance holeIndex,
                 Distance topIndex, T value) {
  Distance parent = (holeIndex - 1) / 2;
  while (holeIndex > topIndex && (*(first + parent) < value)) {
    *(first + holeIndex) = *(first + parent);
    holeIndex = parent;
    parent = (holeIndex - 1) / 2;
  }
  *(first + holeIndex) = value;
}
```

This implementation, strictly speaking, is an *abstraction violation*. That is, the algorithm accesses the data structure at a level of abstraction that is lower than appropriate. By explicitly coding the heap algorithm in terms of array indices or equivalently in terms of iterators, instead of in terms of tree nodes, the algorithm is committing an abstraction violation. The disadvantages of this is that the algorithm is unnecessarily complicated, it can not be used as a generic algorithm, and any changes in implementation (array representation) require changes in the algorithm.

One solution is not to use arrays. The object-oriented approach for building a heap would be to represent the binary tree with node objects that have pointers to their children and parents. This succeeds in providing a nice interface but looses efficiency in a number of ways.

1. Storing the parent and child pointers requires additional memory.

2. Accessing the pointer values requires memory accesses, whereas index calculations do not. The trend on modern microprocessors is that memory accesses are becoming significantly more expensive relative to ALU operations.

3. The allocation for each node is often expensive. With the array the allocation is done once and amortized.

4. The nodes can be spread across the heap, giving poor data-locality and therefore poor cache performance and slower access times.

Nevertheless, because this approach is straightforward (in that the concrete representation directly maps to the data abstractions) many object-oriented libraries have chosen this route, despite the loss in efficiency. However, as a result, the libraries are not used in situations where high performance is a high priority. For example, many graph algorithms are still written in Fortran, using low-level array representations [4, 15, 18], even though there exist many easy to use graph libraries such as LEDA [12].

The second two problems listed above can be fixed using better allocation schemes. The first two problems concerning pointer overhead is more difficult. What is needed is some way to keep the array based representation and add an interface layer that makes the data structure easier to work with and suitable for use in generic algorithms.

The Mayfly pattern provides a solution to this problem. In the case of the heap data structure, we use the array representation and indexing scheme, but when an algorithm wishes to operate on the heap, temporary objects are created "on the fly" to provide the same kind of interface one would see from the explicitly stored node objects mentioned above. The temporary node object consists of a pointer to the array and an integer for indexing. The methods of the node objects carry out the index calculations. For example, here is the `parent()` method of the node class. `r` is the pointer to the array, `(i - 1)/2` is the index of the parent node, and `n` is the size of the array.

```
tree_node tree_node::parent() const {
  return tree_node(r, (i - 1) / 2, n);
}
```

We can now implement the heap algorithms in terms of the tree interface presented by the Mayfly node objects. Here is the algorithm equivalent to the `push_heap()` algorithm above.

```
template <class TreeNode>
TreeNode up_heap(TreeNode x) {
  while (x.has_parent() && (x.parent().key() < x.key()))
    swap(x, x.parent());
  return x;
}
```

The algorithm is now much easier to understand and looks like pseudo-code. Also it is easy to verify the correctness of the algorithm in that it maintains the heap property $value(c) \leq value(parent(c))$. In addition, this algorithm is suitable as a generic algorithm, since it can operate on any data structure implementation that exports a tree interface.

With the Mayfly approach, the efficiency of the array data structure is retained. The only difference with using the Mayflies is that there is an extra layer of function calls and the pointers and integers have been wrapped up in temporary objects. Many C++ optimizing compilers can reduce this overhead to zero, which will be discussed in more detail in the implementation section.

To summarize, the following characteristics define a Mayfly.

- A Mayfly implements an interface for some data structure.

- A Mayfly is light weight, with few data members (the maximum workable size for a Mayfly depends on the compiler).

- It is stored on the stack or even just in machine registers.
- It has no virtual member functions.
- It is passed by value.
- A Mayfly is usually only directly involved with functions that are inlined.

3 Applicability

Use the Mayfly pattern when all of the following are true:

- Execution speed or memory efficiency are high priorities.
- Low level details of a data structure should be hidden, for ease of programming or for constructing generic algorithms.
- Run-time flexibility is not needed. The Mayfly pattern relies heavily on *static polymorphism* (template functions) for efficiency, which requires the object types to be completely defined at compile time. This rules out the extra flexibility that can be had with *dynamic polymorphism* (virtual functions in C++).

4 Structure

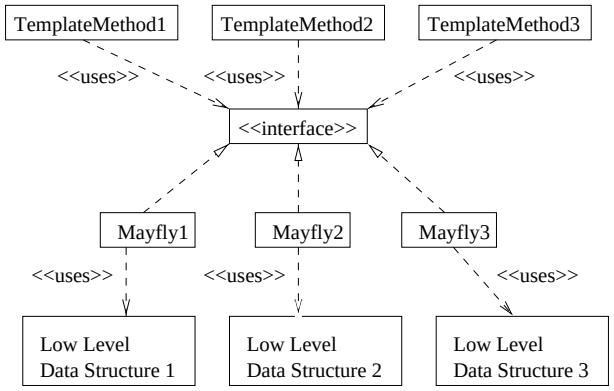


Figure 2: The structure for the Mayfly pattern.

5 Participants

- Interface
  - Codifies the minimal set of requirements needed by a Template Method [3] to operate on a family of data structures. For example, Table 1 shows the set of requirements that could define a TreeNode interface. We provide the `children()` method instead of the `left_child()` and `right_child()` methods to allow for generalization to k-ary trees (instead of just binary trees).
- Mayfly

expression	return type	note
`X::children_list`	A Container	The type for the children list
`X::key_type`	LessThan Comparable	The type for the keys to sort by
`x.key()`	`X::key_type`	The key value for this node
`x.parent()`	`X`	The parent node of node `x`
`x.has_parent()`	`bool`	Whether node `x` has a parent
`x.children()`	`X::children_list`	A list of the children of node `x`
`swap(x,y)`	`void`	Swap the position of `x` and `y` in the tree

Table 1: TreeNode interface requirements. `X` denotes the particular TreeNode class. `x` and `y` are objects of type `X`.

  - Implements a lightweight interface for an efficient data structure using temporary objects.

- Template Method
  - Implements some functionality in terms of the interface.
  - Makes few assumptions about the data structures, which allows the interface to be smaller.
  - Does not use the data structure directly (no abstraction violations).
- Low-level Data Structure
  - Provides efficient storage and fast access to data. It is the concrete data structure for which an interface is to be built. Often there are many data structures that can be accessed with the same interface. Each data structure will needs its own Mayfly (or family of Mayfly objects), to implement the interface.

6 Collaborations

- A Mayfly uses the Low-level Data Structure.
- The Mayfly implements a particular Interface.
- A Template Method can use many different Mayflies. All the Mayflies that implement the same interface can be substituted for one another in the Template Method.

7 Implementation

Much of the definition of the Mayfly design pattern relates to its implementation. Specifically, the pattern describes a way to implement interfaces with zero overhead. In this section we will demonstrate this claim.

The key idea is that we want use implementation techniques that are easy for modern compilers to optimize. Good optimizing compilers are now quite adept at completely removing the overhead associated with objects and functions as long as a few conditions are met:

1. The functions are
   - small.
   - not recursive.
   - not virtual.
2. The objects must be on the stack, not dynamically allocated.

7.1 Performance Optimization Discussion

When these conditions are met, compilers can apply *function inlining* and *lightweight object optimization* [7, 13] to completely remove the overhead due to the function calls and to the objects. A function call introduces overhead because it takes on the order of 30 instructions (most of which are loads and stores) to create and fill a new frame on the stack. This is significant when the function body contains only a few instructions. The overhead due to objects is a bit more complicated. In modern RISC processors efficient use of registers is of utmost importance for performance. Compilers for RISC processors have register allocation algorithms that map variables to registers, which drastically reduces the number of loads and stores needed to access the variable. The traditionally register allocation algorithms do not attempt to map data within structures (or objects) to registers. This means that the use of such objects as iterators will cause a significant increase in the memory traffic for a tight loop. Lightweight object optimization breaks an object up into its individual parts so that the traditional register allocation algorithms can be effective in the presence of objects. Many times the presence of objects can interfere with other optimizations as well, such as copy propagation and loop nest optimizations such as blocking for data locality, loop interchange, and loop fusion [6, 8, 9, 11].

7.2 Compiler Optimization Example

One compiler that does a excellent job with procedure inlining and lightweight object optimization is the KAI C++ compiler. The result of the optimizations can been seen by examining the intermediate C code output by the compiler (this is a cfront style C++ compiler that converts C++ to C). The following is the output from compiling the `up_heap()` function using the `array_tree` Mayfly class for the TreeNode template argument. The variable names have been demangled to make the intermediate code easier to read.

```
int i;
int* array;
int parent;
// ...
while (i && (array[(((-1) + i) / 2)] < array[i])) {
  int parent_val;
  parent = ((-1) + i) / 2;
  parent_val = array[parent];
  array[parent] = array[i];
  array[i] = parent_val;
  i = ((-1) + i) / 2;
}
```

The result is the same as that of the low-level implementation! This demonstrates how the Mayfly interface is in fact zero-overhead.

7.3 Language Choice

The number of languages that can be used to create Mayflies is quite small. This is because many languages introduce significant overhead in object creation and in method dispatch, so much so that there would be little point in using Mayflies over a straightforward object-oriented approach. For instance, methods in Java use dynamic dispatch, and object creation (even for small temporary objects) is quite expensive. Similarly, Mayflies can not be constructed in dynamic languages such as Smalltalk or Lisp for similar reasons. To be able to use Mayflies and still have some flexibility, the language must have type parameterization. Beyond the language, the compiler used must also perform the optimization discussed above.

8 Sample Code

In this section we will go into the details of an implementation for the `tree_node` class, which implements the tree interface on top of an array using the standard heap indexing scheme. We will also show some sample code demonstrating the use of Mayflies in the Matrix Template Library [16, 17] and in the Generic Graph Component Library (GGCL) [10], two high performance libraries that we have developed.

First we show the data contained within a `tree_node` object. It contains a pointer to the underlying array, an index to the location of the node in the array, and the size of the array. We have templated the class with RandomAccessIterator [19] so that `tree_node` can be used with many different kinds of arrays.

```
template <class RandomAccessIterator>
class tree_node {
  typedef RandomAccessIterator iter;
public:
  tree_node(iter array_start, iter array_end, iter pos)
    : r(start), i(pos - start), n(end - start) { }
  // ...
protected:
  iter r;             // pointer to the underlying array
  difference_type i;  // location of this node
  difference_type n;  // size of the array
};
```

Next we must implement the methods required by the TreeNode interface. We start with some of the easier methods to implement. The `key()` method merely returns the key value from the underlying array. The `parent()` method, as we have seen before, calculates the index of the parent node, and creates a new object to represent the parent. It is important that the parent object be just a temporary object, not allocated on the heap, so that we do not introduce unnecessary overhead. The `has_parent()` method just checks to see if this node is at the top of the tree.

```
template <class RandomAccessIterator>
class tree_node {
  // ...
  reference key() { return r[i]; }
  tree_node parent() const { return tree_node(r, (i - 1) / 2, n); }
  bool has_parent() const { return i != 0; }
  // ...
};
void swap(tree_node x, tree_node y) {
  value_type tmp = x.key();
  x.key() = y.key();
  y.key() = tmp;
  y.i = x.i;
}
```

Next we implement the `children()` method. First we show an example of how the `children()` method is used. The following algorithm, `down_heap()`, is the mirror image of the `up_heap()` algorithm described in the motivation.

```
template <class TreeNode>
TreeNode down_heap(TreeNode x) {
  while (x.children().size() > 0) {
    typename TreeNode::children_list::iterator child_iter;
    child_iter = std::max_element(x.children().begin(), x.children().end());
    if (x.key() < child_iter->key())
      swap(x, *child_iter);
    else
      break;
  }
  return x;
}
```

The `children()` method returns a `children_list` which behaves just like and STL container, complete with iterators. Here we use them in the STL function `max_element()`. We can implement the `children_list` as another kind of mayfly, nested within the `tree_node`. The data items from the `tree_node` must be duplicated within the `children_list`. This is not an overhead problem since the list is just a temporary object and the copy propagation will eliminate the extra copying. The following code shows most of the implementation for the `children_list`.

```
template <class RandomAccessIterator>
class tree_node {
  // ...
  struct children_list {
    struct iterator { ... };
    children_list(iter rr, difference_type ii, difference_type nn)
      : r(rr), i(ii), n(nn) { }
    iterator begin() { return iterator(r, 2 * i + 1, n); }
    iterator end() { return iterator(r, 2 * i + 1 + size(), n); }
    size_type size() const { ... }
  protected:
    iter r;
    difference_type i;
    difference_type n;
  };
  children_list children() { return children_list(r, i, n); }
  // ...
};
```

We have shown the implementation of a TreeNode based on an array, but there are several other data structures that can be used to implement heaps, such as binomial heaps, Fibonacci heaps, and binary-trees (pointer based). One can implement the TreeNode interface for any of these structures, and then use them with the generic heap algorithms defined above.

9 Known Uses

There are many uses of Mayflies under the guise of the Iterator, though there are also many iterators that are not Mayflies. Iterators that use virtual functions or that are allocated on the heap are not Mayflies. Mayfly iterators are used in the Standard Template Library, our libraries – the Matrix Template Library (MTL) [17, 16] and the Generic Graph Component Library (GGCL) [10] — and a host of others.

We use Mayflies in the MTL to represent rows of a matrix, submatrices, and different views of matrices themselves (such as the transpose of a matrix or the lower triangle of a matrix). In GGCL we use Mayflies to represent vertex and edge objects for graph representations where the vertices and edges are encoded

in some data structure but not stored as explicit objects, such as in adjacency list and adjacency matrix representation.

A much different type of Mayfly is used with a technique called expression templates [20], which provide an efficient implementation of operator overloading for arrays operations. Each arithmetic expression operating on an array object is explicitly turned into an object so that the evaluation of the expressions can be customized. The expression objects can be considered a Mayfly. Again, with a good optimizer these Mayfly objects disappear. The Blitz++ [21] and POOMA [22, 5] libraries make heavy use of expression templates.

Many of the classes associated with the C++ Standard `valarray` class are also Mayflies. The `slice` and `slice_array` are examples of such Mayfly classes.

Numeric classes, such as the Standard `complex<T>`, the extended precision `doubledouble` class, and the `interval<T>` class that we constructed, are all Mayfly objects. In these cases the underlying data structure is trivial, but each of these classes provide the same "numeric" interface for their underlying representations. They also fulfill the other requirements for Mayflies.

10 Related Patterns

Iterator: Mayfly is a generalization of Iterator, combined with a specific implementation method.

Adapter: One can view Mayflies as Adapters for data structures, though the purposes are somewhat different. In addition, various kinds of adapters can be implemented as Mayflies.

Composite: Mayflies can provide a lightweight interface for Composite structures. In addition, some lightweight Composite structures (such as the `children_list`) can be Mayflies.

Template Method: A Template Method (implemented with template functions instead of virtual functions) is implemented in terms of an abstract interface, which is implemented by the Mayfly.

References

[1] M. H. Austern. *Generic Programming and the STL*. Professional computing series. Addison-Wesley, 1999.

[2] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Publishing Company, October 1994.

[4] A. George, J. R. Gilbert, and J. W. Liu, editors. *Graph Theory and Sparse Matrix Computation*. Springer-Verlag New York, Inc, 1993.

[5] S. Karmesin, J. Crotinger, J. Cummings, S. Haney, W. Humphrey, J. Reynders, S. Smith, and T. J. Williams. Array design and expression evaluation in pooma ii. In D. Caromel, R. R. Oldehoeft, and M. Tholburn, editors, *ISCOPE*. Advanced Computing Laboratory, LANL, 1998.

[6] K. Kennedy and K. S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pages 301–320, Portland, Oregon, Aug. 12–14, 1993. Intel Corp. and the Portland Group, Inc., Springer-Verlag.

[7] Kuck and Associates. *Kuck and Associates C++ User's Guide*.

[8] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Fourth Intern. Conf. on Architectural Support for Programming Languages and Operating Systems (APLOS IV), Palo Alto, California*, pages 63–74, April 9-11 1991.

[9] M. S. Lam and M. E. Wolf. Automatic blocking by a compiler. In R. G. Dongarra, Jack; Kennedy, Ken; Messina, Paul; Sorensen, Danny C.; Voigt, editor, *Proceedings of the 5th SIAM Conference on Parallel Processing for Scientific Computing*, pages 537–542, Houston, TX, Mar. 1991. SIAM.

[10] L.-Q. Lee and J. G. S. A. Lumsdaine. The generic graph component library. In *OOPSLA*, 1999. submitted.

[11] K. S. McKinley and O. Temam. A quantitative analysis of loop nest locality. *ACM SIGPLAN Notices*, 31(9):94–104, Sept. 1996.

[12] K. Mehlhorn and S. Naeher. *LEDA*. http://www.mpi-sb.mpg.de/LEDA/leda.html.

[13] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[14] Object Management Group. *UML Notation Guide*, version 1.1 edition, September 1997. http://www.rational.com/uml/.

[15] Y. Saad. *Iterative Methods for Sparse Minear System*. PWS Publishing Company, 1996.

[16] J. G. Siek and A. Lumsdaine. The matrix template library: A generic programming approach to high performance numerical linear algebra. In *International Symposium on Computing in Object-Oriented Parallel Environments*, 1998.

[17] J. G. Siek and A. Lumsdaine. *Modern Software Tools in Scientific Computing*, chapter A Modern Framework for Portable High Performance Numerical Linear Algebra. Birkhauser, 1999.

[18] S. S. Skiena. *The Algorithm Design Manual*. Springer-Verlag New York, Inc, 1998.

[19] A. A. Stepanov and M. Lee. The Standard Template Library. Technical Report X3J16/94-0095, WG21/N0482, ISO Programming Language C++ Project, May 1994.

[20] T. L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995. Reprinted in C++ Gems, ed. Stanley Lippman.

[21] T. L. Veldhuizen and M. E. Jernigan. Will C++ be faster than Fortran. In *Scientific Computing in Object-Oriented Parallel Environments*. ISCOPE, December 1997.

[22] R. C. Whaley and J. Dongarra. Automatically tuned linear algebra software (ATLAS). Technical report, University of Tennessee and Oak Ridge National Laboratory, 1997.