Small Data Structures

Version 13/06/00 02:15 - 33 by Charles Weir

How can you reduce the memory needed for your data?

- The *memory requirements* of the data exceed the memory available to the system.
- You want to increase *usability* by allowing users to store as much of their data as possible.
- You need to be able to *predict* the program's use of memory.
- You cannot delete some of the data from the program.

The fundamental difference between code and data is that programmers care about code while users care about data. Programmers have some direct control over the size of their code (after all, they write it!), but the data size is often completely out of the programmers' control. Indeed, given that a system is supposed to store users' data, any memory allocated to code, buffers, or other housekeeping is really overhead as far as the user is concerned. Often the amount of memory available to users can make or break a systems *usability* — a word processor which can store a hundred thousand word document is much more useful than one which are only store a hundred words.

Data structures that are appropriate where memory is unrestricted may be far too prodigal where memory is limited. For example, a typical implementation of an address database might store copies of information in indexes as well as the actual data, effectively storing everything in the database twice. Porting such an approach to the Strap-It-On wrist-top PC would halve the number of addresses that could be stored in the database.

Techniques like COMPRESSION and using SECONDARY STORAGE can reduce a program's main memory requirements, but both have significant liabilities when used to manage the data a program needs to work on. Many kinds of compressed data cannot be accessed randomly; if random access is required the data must be uncompressed first, costing time, and requiring a large amount of buffer memory for the uncompressed data. Data stored on secondary storage is similarly inaccessible, and needs to be copied into main memory buffers before it can be accessed.

Therefore: *Choose the smallest structure that supports the operations you need.*

For any given data set there are many different possible data structures that might support it. Suppose, for example, you need an unordered collection of object references with no duplicates – in mathematical terms, a set. You could implement it using a linear array of pointers, using a hash table, or using a variety of tree structures. Most class libraries will provide several different implementations of such collections; the best one to choose depends on your requirements. Where *memory is limited*, therefore, you must be particularly careful to choose a structure to minimise the program's *memory requirements*.

You can think of data structure design as a three-stage process. First analyse the program's requirements to determine the information the program needs to store; unnecessary information requires no memory!

Second, analyse the characteristics of the data; what's its total volume; how does it vary over a single program run and across different runs; and what's its granularity – does it consist of a few large objects or many small objects? You can also analyse the way you'll access the data: whether it is read and written, or only ever read; whether it is accessed sequentially or randomly; whether elements are inserted into the middle of the data or only added at the end.

Third, choose the data structures. Consider as many different possibilities as you can – your standard class libraries will provide some basic building blocks, but consider also options like embedding objects (FIXED ALLOCATION), EMBEDDING POINTERS, or PACKING the DATA. For each candidate data structure design, work out the amount of memory it will require to store the data you need, and check that it can support all the operations you need to perform. Then consider the benefits and disadvantages of each design: for example a smaller data structure may require more processing time to access, provide insufficient flexibility or give insufficient real-time performance. You'll need also to evaluate the resulting memory requirements for each possibility against the total amount of memory available – in some cases you may need to do simple trials using scratch code. If none of the solutions are satisfactory you may need to go back and reconsider your earlier analysis, or even the requirements of the system as a whole. On the other hand there's no need to optimise memory use beyond your given requirements (see the THRESHOLD SWITCH pattern [Auer and Beck 1996]).

For example, the Strap-It-On^{TM} address program has enough memory to store the address records but not indexes. So its version of the address program uses a sorted data structure that does not need extra space for an index but that is slower to access than the indexed version.

Consequences

Choosing suitable data structures can reduce a program's *memory requirements*, and the time spent can increase the *quality of the program's design.*

By increasing the amount of users' information the program can store, careful data structure design can increase a program's *usability*.

However: analysing a program's requirements and optimising data structure design takes *programmer discipline* to do, and *programmer effort* and time to do well.

Optimising data structure designs to suit limited memory situations can restrict a program's *scalability* should more memory become available.

The *predictability* of the program's memory use, the *testing costs*, and the program's *time performance* may or may not be affected, depending upon the chosen structures.

❖ ❖ ❖

Implementation

Every data structure design for any program must trade off several fundamental forces: *memory requirements, time performance,* and *programmer effort* being the most important. Designing data structures for a system with tight memory constraints is no different in theory from designing data structures in other environments, but the practical tradeoffs can result in different solutions. Typically you are prepared to sacrifice *time performance* and put in more *programmer effort* than in an unconstrained system, in order to reduce the *memory requirements* of the data structure.

There are several particular issues to consider when designing data structures to minimise memory use:

1. Predictability versus Exhaustion

The *predictability* of a data structure's memory use, and ultimately of the whole program can be as important as the structure's overall memory requirements, because making memory use more predictable makes it easier to manage. Predictability is closely related to the need to deal with *memory exhaustion:* if you can predict the program's maximum memory use in advance then you can use FIXED ALLOCATION to ensure the program will never run out of memory.

2. Space versus Flexibility

Simple, static, inflexible data structures usually require less memory than more complex, dynamic, and flexibly data structures. For example, you can implement a one-to-one relationship with a single pointer or even an inline object (see FIXED ALLOCATION), while a one-to-many relationship will require collection classes, arrays, or EMBEDDED POINTERS. Similarly, flexible collection classes require more memory than simple fixed sized arrays, and objects with methods or virtual functions require more memory than simple records (C++ `structs`) without them. If you don't need flexibility, don't pay for it; use simple data structures that need less memory.

3. Calculate versus Store

Often you can reduce the amount of main memory you need by calculating information rather than storing it. Calculating information reduces a program's time performance and can increase its power consumption, but can also reduce its memory requirements. For example, rather than keeping an index into a data structure, you can traverse the whole data structure using a linear search. Similarly, the PRESENTER pattern [Vlissides 1998] describes how graphical displays can be redrawn from scratch rather than being updated incrementally using a complex object structure.

❖ ❖ ❖

Specialised Patterns

This chapter contains five specialised patterns that describe a range of techniques for designing data structures to minimise memory requirements. The following figure shows the relationships between the patterns.

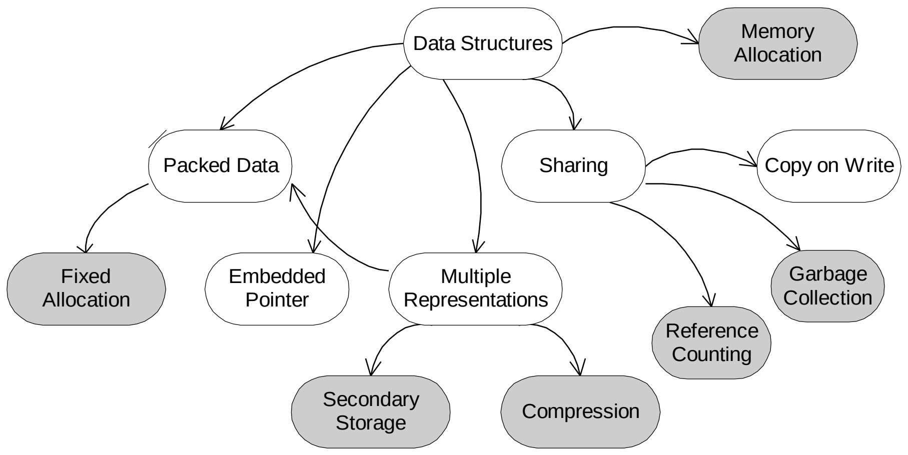


Figure 1: Data Structure Pattern Relationships

PACKED DATA selects suitable internal representations of the data elements in an object, to reduce its memory footprint.

SHARING removes redundant duplicated data. Rather than using multiple copies of functions, resources or data, the programmer can arrange to store only one copy, and use that copy wherever it is needed.

COPY-ON-WRITE extends SHARING so that shared objects can be modified without affecting other client objects that use the shared objects.

EMBEDDED POINTERS reduce the memory requirements for collection data structures, by eliminating auxiliary link objects and moving pointers into the data objects stored in the structures.

MULTIPLE REPRESENTATIONS are effective when no single representation is simultaneously compact enough for storage yet efficient enough for actual use.

Known Uses

Like Elvis, data structures are everywhere.

The classic example of designing data structures to save memory is the technique of allocating only two BCD digits to record the year when storing dates [Yourdon 2000]. This had unfortunate consequences, although not the disasters predicted in the lead-up to the millennium [Berry, Buck, Mills, Stipe 1987]. Of course these data structure designs were often made for the best of motives: in the 1960s disk and memory was much more expensive than it is today; and allocating two extra characters per record could cost millions.

An object-oriented database built using Smalltalk needed to be scaled up to cope with millions of objects, rather than several thousand. Unfortunately, a back-of-the envelope calculation showed that the existing design would require a ridiculous amount of disk space and thus buffer memory. Examination of the database design showed that Smalltalk Dictionary (hash table) objects occupied a large proportion of its memory; further investigation showed and that these Dictionaries contained only two elements: a date and a time. Redesigning the database to use Smalltalk Timestamp objects that stored a date and time directly, rather than the dictionary, reduced the number of objects needed to store each timestamp from at least eight to three, and made the scaled-up database project feasible.

See also

Once you have designed your data structures, you then have to allocate the memory to store them. The MEMORY ALLOCATION chapter presents a series of patterns describing how you can allocate memory in your programs.

In many cases, good data structure design alone is insufficient to manage your program's memory. The COMPRESSION chapter describes how memory requirements can be reduced by explicitly spending processor time to build very compact representations of data that generally cannot be used directly in computations. Moving less important data into SECONDARY STORAGE and constant data into READ-ONLY MEMORY can reduce the demand for writable primary storage further.

There are many good books describing data structure design in depth. Knuth [1997] remains a classic, though its examples are, effectively, in assembly language. Hoare [1972] is another seminal work, though nowadays difficult to find. Aho, Hopcroft and Ullman [1983] is a standard text for university courses, with examples in a Pascal-style pseudo-code, Cormen et al [1990] is a more in-depth Computer Science text, emphasising the mathematical analysis of algorithms. Finally Segewick's series beginning with *Algorithms* [1988] provide a more approachable treatment, with editions quoting source code in different languages – for example *Algorithms in C++: Fundamentals, Data Structures, Sorting, Searching,* [Segewick 1999]

Packed Data

Also known as: Bit Packing

How can you reduce the memory needed to store a data structure?

- You have a data structure (a collection of objects) that has significant memory requirements.
- You need fast random access to every part of every object in the structure.
- You need to store the data in these objects in main memory.

No matter what else you do in your system, sooner or later you end up having to design the low-level data structures to hold the information your program needs. In an object-oriented language, you have to design some key classes whose objects store the basic data and provide the fundamental operations on that data. In a program of any size, although there may be only a few key data storage classes, there can be a large number of instances of these classes. Storing all these objects can require large amounts of memory, certainly much more than storing the code to implement the classes.

For example, the Strap-It-On's Insanity-Phone application needs to store all of the names and numbers in an entire local telephone directory (200,000 personal subscribers). All these names and numbers should just about fit into the Strap-It-On's memory, but would leave no room for the program than displayed the directory, let alone any other program in the Wrist-Top PC.

Because these objects (or data structures) are the core of your program, they need to be easily accessible as your program runs. In a program of any complexity, the objects will need to be accessed randomly (rather than in any particular order) and then updated. Taken together, random access with updating requires that the objects are stored in main memory.

You might consider using COMPRESSION on each object or on a set of objects, but this would make processing slow and difficult, and makes random access to the objects using references almost impossible. Similarly, moving objects into SECONDARY STORAGE is not feasible if the objects need to be accessed rapidly and frequently. Considering the Insanity-Phone example again, the data cannot be placed in the Strap-It-On's secondary memory because that would be too slow to access; and the data cannot be compressed effectively while maintaining random access because each record is too small to compressed individually using standard adaptive compression algorithms.

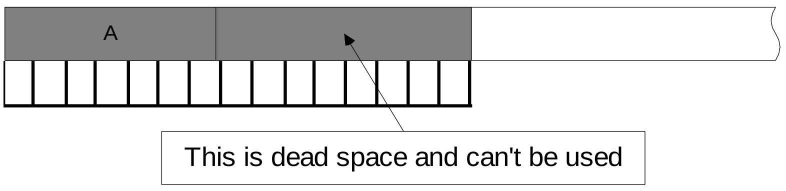
Therefore: *Pack data items within the structure so that they occupy the minimum space.*

There are two ways to reduce the amount of memory occupied by an object:

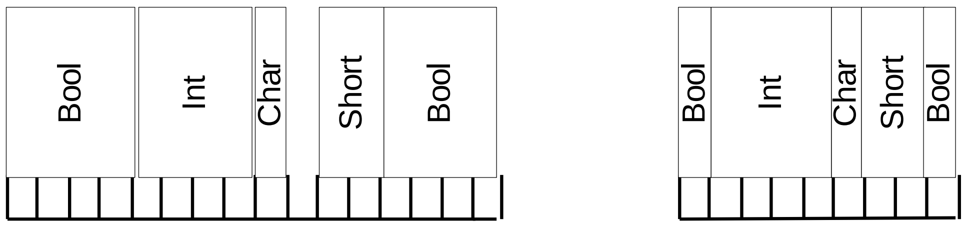
1. Reduce the amount of memory required by each field.
2. Reduce the amount of unused memory allocated between fields.

Consider each individual field in turn, and consider how much information that field really needs to store. Then, chose the smallest possible representation for that information. This may be the smallest suitable language level-data type, or even smaller, using different bits within, say, a machine word to encode different data items.

Once you have analysed each field, analyse the class as a whole to ensure that extra memory is not allocated between fields. Compilers or assemblers often ensure that fields are aligned to take advantage of CPU instructions that make it easier to access aligned data, so, for example, all two-byte fields be stored at even addresses, and all four-byte fields at addresses that are multiples of four. Aligning fields wastes the memory space between the end of one field and the start of the next.



The figure below shows how packing an object can almost halve the amount of memory that it requires. The normal representation on the left allocates four bytes for each Boolean variable (presumably to use faster CPU instructions) and aligns two and four-byte variables to two or four-byte boundaries; the packed representation allocates only one byte for each Boolean variable and dispenses with alignment for longer variables.



Normal representation Packed representation

Considering the Insanity-Phone again, the designers realised that local phone books never cover more than 32 area codes – so each entry requires only 5 bits to store the area code. A seven-digit decimal number requires 24 bits. Surnames are duplicated many times, so Insanity-Phone stores each surname just once – an example of SHARING – and therefore gets less than 30,000 unique names in each book; this requires 18 bits. Storing up to three initials (5 bits each – see STRING COMPRESSION) costs a further 15 bits. The total is 62 bits, and this can be stored in one 64 bit long integer for each entry.

Consequences

Each instance occupies less memory reducing the total *memory requirements* of the system, even though the same amount of data can be stored, updated, and accessed randomly. Choosing to pack one data structure is usually a *local* decision, with little *global* effects on the program as a whole.

However: The *time performance* of a system suffers, because CPUs are slower at accessing unaligned data. If accessing unaligned data requires many more instructions than aligned data, it can impact the program's *power consumption*. More complicated packing schemes like bit packing can have even higher overheads.

Packing data requires *programmer effort* to implement, produces less intuitive code which is harder to *maintain*, especially if you use non-standard data representations. More complicated techniques can increase *testing costs*.

Packing schemes that rely on particular aspects of a machine's architecture, such as word sizes or pointer formats, will reduce *portability*. If you're using non-standard internal representations, it is harder to exchange objects with other programs that expect standard representations.

Finally, packing can reduce *scalability*, because it can be difficult to unpack data structures throughout a system if more memory becomes available.

❖ ❖ ❖

Implementation

The default implementation of a basic type is usually chosen for time performance rather than speed. For example, Boolean variables are often allocated as much space as integer variables, even though they need only a single bit for their representation. You can pack data by choosing smaller data types for variables; for example, you can represent Booleans using single byte integers or bit flags, and you may be able to replace full-size integers (32 or 64 bits) with 16 or even 8-bit integers (C++'s short and char types).

Compilers tend to align data members on machine-word boundaries which wastes space (see the figure on p.N above). Rearranging the order of the fields can minimise this padding, and can reduce the overhead when accessing non-aligned fields. A simple approach is to allocate fields within words in decreasing order of size.

Because packing has significant overheads in speed and maintainability, it is not worthwhile unless it will materially reduce the program's memory requirements. So, pack only the data structures that consume significant amounts of memory, rather than packing every class indiscriminately.

Here are some other issues to consider when applying the PACKED DATA pattern.

1. Compiler and Language Support

Compilers, assemblers and some programming language definitions support packing directly.

Many compilers provide a compilation flag or directive that ensures all data structures use the tightest alignment for each item, to avoid wasted memory at the cost of slower run-time performance. Microsoft C++, the directive:

```
#pragma pack( n )
```

sets the packing alignment to be based on n-byte boundaries, so pack(1) gives the tightest packing; the default packing 8 [Microsoft 1997]. G++ provides a `pack` attribute for individual fields to ensure they are allocated directly after the preceding field [Stallman 1999].

2. Packing Objects into Basic Types

Objects can impose a large memory overhead, especially when they contain only a small amount of data. Java objects impose an allocation overhead of at least one an additional pointer, and C++ objects with virtual functions require a pointer to a virtual function table. You can save memory by replacing objects by more primitive types (such as integers or pointers), an example of MULTIPLE REPRESENTATIONS.

When you need to process the data, wrap each primitive type in a first-class object, and use the object to process the data; when you've completed processing, discard the object, recover the basic type, and store it once again. To avoid allocating lots of wrapper objects, you can reuse the same wrapper for each primitive data item. The following Java code sketches how a `BigObject` can be repeatedly initialised from an array of integers for processing. The `become` method reinitialises a `BigObject` from its argument, and the `process` method does the work.

```
BigObject obj = new BigObject(0);

    for (int i=1; i<10; i++)
    {
        obj.become(bigarray[i]);
        obj.process();
    }
```

In C++, we can define operators to convert between objects and basic types, so that the two can be used interchangeably:

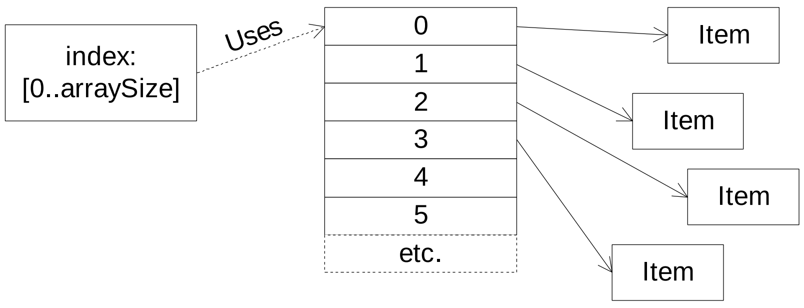
```
class BigIntegerObject
{
public:
   BigIntegerObject( int anInt=0 ) : i( anInt ) {}
   operator int() { return i; }
private:
   int i;
};

int main()
{
   BigIntegerObject i( 2 ), j( 3 );
   BigIntegerObject k = i*j;  // Uses conversion operators
```

3. Packing Pointers using Tables

Packing pointers is more difficult, because they don't obviously contain redundant data. To pack pointers you need to look at what they reference.

If a given pointer may point to only one of a given set of then it may be possible to replace the pointer with an index into an array; often, an array of pointers to the original item. Since the size of the array is usually much less than the size of all the memory in the system, the number of bits needed for an array index can be much less than the number of bits needed for a general pointer.



Take, for example, the Insanity-phone application above. Each entry apparently needs a pointer to the corresponding surname string: 32 bits, say. But if we implement an additional array of pointers into the string table, then each entry need only store an index into this array (16 bits). The additional index costs memory: 120Kb using 4-byte pointers.

If you know that each item is guaranteed to be within a specific area of memory, then you can just store offsets within this memory. This might happen if you've built a string table, or if you're using POOLED ALLOCATION, or VARIABLE ALLOCATION within a heap of known size. For example, if all the Insanity-Phone surname strings are stored in a contiguous table (requiring less than 200K with STRING COMPRESSION), the packed pointer needs only hold the offset from the start of the table: 18 bits rather than 32.

4. Packing Pointers using Bitwise Operations

If you are prepared to sacrifice portability, and have an environment like C++ that allows you to manipulate pointers as integers, then you have several possible ways to pack pointers. In some architectures, pointers contain redundant bits that you do not need to store. Long pointers in the 8086 architecture had at least 8 redundant bits, for example, so could be stored in three bytes rather than four.

You can further reduce the size of a pointer if you can use knowledge about the heap allocation mechanism, especially about alignment. Most memory managers allocate memory blocks aligned on word boundaries; if this is an 8-byte boundary, for example, then you can know that any pointer to a heap object will be a multiple of eight. In this case, the lowest three bits of each pointer are redundant, and can be reused or not stored. Many garbage collectors, for example, pack tag information into the low bits of their pointers [Jones and Lins 1996].

Example

Consider the following simple C++ class:

```
class MessageHeader {
    bool  inUse;
    int   messageLength;
    char  priority;
    unsigned short channelNumber;
    bool  waitingToSend;
};
```

With 8-byte alignment, this occupies 16 bytes, using Microsoft C++ on Windows NT. With the compiler packing option turned on it occupies just 9 bytes. Note that the packed structure does not align the integer `i1` to a four-byte boundary, so on some processors it will take longer to load and store.

Even without compiler packing, we can still improve the memory use just by reordering the data items within the structure to minimise the gaps. If we sort the fields in decreasing order of size

```
class ReorderedMessageHeader {
    int   messageLength;
    unsigned short channelNumber;
    char  priority;
    bool  inUse;
    bool  waitingToSend;
};
```

the class occupies just 12 bytes, a saving of four bytes. If you're using compiler field packing, both `MessageHeader` and `ReorderedMessageHeader` occupy 9 bytes, but there's still a benefit to latter since it puts all the member items on the correct machine boundaries where they can be manipulated fast.

We can optimise the structure even more using bitfields. The following version contains the same data as before:

```
class BitfieldMessageHeader {
    int      messageLength;
    unsigned channelNumber: 16;
    unsigned priority:      8;
    unsigned inUse:         1;
    unsigned waitingToSend: 1;

public:
    bool IsInUse() { return inUse; }
    void SetInUseFlag( bool isInUse ) { inUse = isInUse; }
    char Priority() { return priority; }
    void SetPriority( char newPriority ) { priority = newPriority; }
    // etc.
};
```

but occupies just 8 bytes, a further saving of four bytes – or one byte if you're using compiler packing.

Unfortunately compiler support for booleans in bitfields tends to be inefficient. This problem isn't actually a sad reflection on the quality of C++ compiler writers today; the real reason is that it requires a surprising amount of code to implement the semantics of, say, the `setB1` function above. We can improve performance significantly by using bitwise operations instead of bitfields, and implement the member functions directly to expose these operations:

```
class BitwiseMessageHeader {
    int      messageLength;
    unsigned short channelNumber;
    char priority;
    unsigned char flags;
public:
    enum FlagName { InUse = 0x01, WaitingToSend = 0x02 };
    bool GetFlag( FlagName f )   { return (flags & f) != 0; }
    void SetFlag( FlagName f )   { flags |= f;  }
    void ResetFlag( FlagName f ) { flags &= ~f;  }
};
```

This optimises performance, at the cost of exposing some of the implementation.

❖ ❖ ❖

Known Uses

Packed data is ubiquitous in memory-limited systems. For example virtually all Booleans in the EPOC system are stored as bit flags packed into integers. The Pascal language standard includes a special `PACKED` data type qualifier, used to implement the original Pascal compiler.

To support dynamic binding, a C++ object normally requires a vtbl pointer to support virtual functions [Stroustrup 1995, Stroupstrup 1997]. EPOC requires dynamic binding to support Multiple Representations for its string classes, but a vtbl pointer would impose a four bytes overhead on every string. The EPOC string base class (`TDesC`) uses the top 4 bits of its 'string length' data member to identify the class of each object:

```
class TDesC8 {  private:
   unsigned int iLength:28;
   unsigned int iType:4;
   /* etc... */
```

Dynamically bound functions that depend on the actual string type are called from `TDesC` using a switch statements on the value of the `iType` bitfield.

Bit array classes are available in both the C++ Standard Template Library and the Java Standard Library. Both implement arrays of bits using array of machine words. Good implementations of the C++ STL also provide a template specialisation to optimise the special case of an array of Booleans by using a bitset [Stroustrup 1997, Chan et al 1998].

Java supports object wrapper versions for many primitive types (Integer, Float). Programmers typically use the basic types for storage and the object versions for complicated operations. Unfortunately Java collections store objects, not basic types, so every basic type must be wrapped before it is stored into a collection [Gosling, Joy, Steele 1996]. To avoid storing multiple copies of the same information the Palm Spotless JVM carefully shares whatever objects it can, such as constant strings defined in different class files [Taivalsaari et al 1999].

See Also

EMBEDDED POINTERS provides a way to limit the space overhead of collections and similar data structures. FIXED ALLOCATION and POOLED ALLOCATION provide ways to reduce any additional memory management overhead.

Packing string data often requires STRING COMPRESSION.

The VISITOR and PRESENTER [Vlissides 1996] patterns can provide behaviour for collections of primitive types (bit arrays etc.) without having to make each basic data item into an object. The FLYWEIGHT PATTERN [Gamma et al 1995] allows you to process each item of a collection of packed data within the context of its neighbours.

Sharing

Also Known As: Normalisation.

How can you avoid multiple copies of the same information?

- The same information is repeated multiple times.
- Very similar objects are used in different components.
- The same functions can be included in multiple libraries
- The same literal strings are repeated throughout a program.
- Every copy of the same information uses memory.

Sometimes the same information occurs many times throughout a program, increasing the program's *memory requirements.* For example, the Strap-It-On user interface design includes many icons showing the company's bespectacled founder. Every component displaying the icon needs to have it available, but every copy of that particular gargoyle wastes memory.

Duplication can also enter the program from outside. Data loaded from RESOURCE FILES or from an external database must be recreated inside a program, so loading the same resource or data twice (possibly in different parts of the program) will also result in two copies of the same information.

Copying objects has several benefits. Architecturally, it is important that components take responsibility for the objects they use, so copying objects between components can simplify ownership relationships. Some language constructs (such as C++ value semantics and Smalltalk cloning) assume object copying; and sometimes copying objects to where they are required can avoid indirection, making systems run faster.

Unwanted duplication doesn't just affect data objects. Unless care is taken, every time a separately compiled or built component of the program uses a library routine, a copy of that routine will be incorporated into the program. Similarly every time a component uses a string or a constant a copy of that string may be made and stored somewhere.

Unfortunately, for whatever reason information is duplicated, every copy takes up memory that could otherwise be put to better use.

Therefore: *Store the information once, and share it everywhere it is needed.*

Analyse your program to determine which information is duplicated, and which information can be safely shared. Any kind of information can be duplicated — images, sounds, multimedia resources, fonts, character tables, objects, and functions, as well as application data.

Once you have found common information, check that it can be shared. In particular, ensure that it never needs to be changed, or that all its clients can cope whenever it is changed. Modify the information's clients so that they all refer to a single shared copy of the information, typically by accessing the information through a pointer rather than directly.

If the shared information can be discarded by its clients, you may need to use REFERENCE COUNTING or GARBAGE COLLECTION so that it is only released once it is no longer needed anywhere in the program. If individual clients may want to change the data, you may need to use COPY-ON-WRITE.

For example, the Strap-It-On PC really only needs one copy of Our Founder's bitmap. This bitmap is never modified, so a single in-memory copy of the bitmap is shared everywhere it is needed in the system.

Consequences

Judicious sharing can reduce a program's *memory requirements,* because only one copy is required of a shared object. SHARING also increases the *quality* of the design, since there's less chance of code duplication. SHARING generally does not affect a program's *scalability* when more memory is made available to the system, nor its *portability*. Since there's no need to allocate space for extra duplicate copies, SHARING can reduce *start-up times*, and to a lesser extent *run-time performance*.

However: *programmer effort* and *discipline*, and *team co-ordination* is required to design programs to take advantage of sharing. Designing sharing also increases the complexity of the resulting system, adding to *maintenance* and *testing* costs since shared objects create interactions between otherwise independent components.

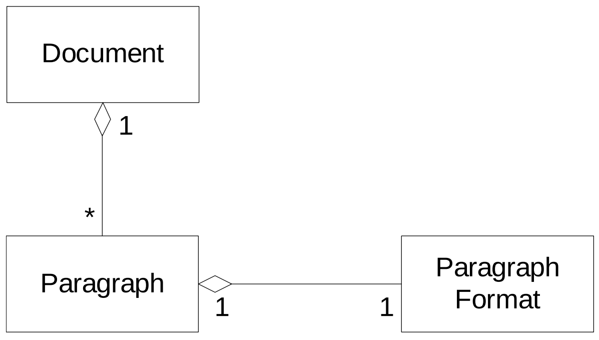
Although it does not affect the *scalability* of centralised systems, sharing can reduce the *scalability* of distributed systems, since it can be more efficient to make one copy of each shared object for each processor.

Sharing can introduce many kinds of aliasing problems, especially when read-only data is changed accidentally [Hogg 1991, Noble et al 1998], and so can increase *testing costs*. In general, sharing imposes a *global* cost on programs to achieve *local* goals, as many components may have to be modified to share a duplicated data structure.

❖ ❖ ❖

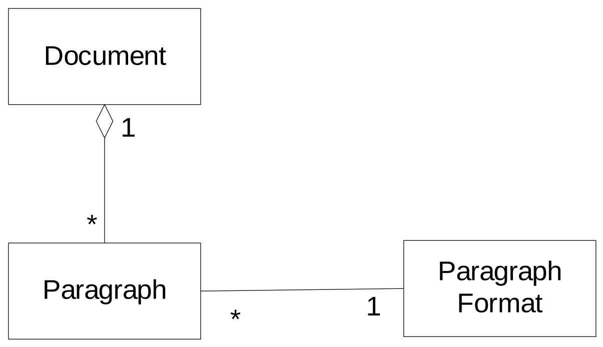
Implementation

Sharing effectively changes one-to-one (or one-to-many) relationships into many-to-one (or many-to-many) relationships. Consider the example below, part of a simple word processor, in UML notation [Fowler 1997]. A `Document` is made up of many `Paragraphs`, and each `Paragraph` has a `ParagraphFormat`.



Considering each class in turn, it is unlikely that `Documents` or `Paragraphs` will be duplicated, unless, for example, many documents have many identical paragraphs. Yet many paragraphs within a document will have the same format. This design, however, gives each `Paragraph` object has its own `ParagraphFormat` object. This means that the program will contain many `ParagraphFormat` objects (one for each `Paragraph`) whilst many of these objects will have exactly the same contents. `ParagraphFormats` are obvious candidates for sharing between different `Paragraphs`.

We can show this sharing as a one-to-many relationship.



In the revised design, there will only be a few `ParagraphFormat` objects, each one different, and many `Paragraphs` will share the same `ParagraphFormat` object.

The design activity of detecting repeated data and splitting it into a separate object is called *normalisation*. Normalisation is an essential part of relational database design theory, and boasts an extensive literature [Connolly and Begg 1999; Date 1999; Elmasri and Navathe 2000].

You should consider the following other issues when applying the SHARING **pattern**.

1. Making Objects Shareable

Aliasing problems make it difficult to share objects in object-oriented programs [Hogg 1991, Noble, Vitek, Potter 1998]. Aliasing problems are the side effects caused by changing a shared object: if a shared object is changed by one of its clients the change will affect any other client of the shared object, and such changes can cause errors in clients that do not expect them. For example, changing the font in a shared Paragraph Format object will change the fonts for all Paragraphs that share that format. If the new font is, say, a printer-only font, and is changed to suit one particular paragraph that will never be displayed on screen, it will break other paragraphs using that format which do need to be displayed on screen, because a printer-only font will not work for them.

The only kinds of objects that can be shared safely without side effects are *immutable* objects, objects that can never be changed. The immutability applies to the object itself – it's not enough just to make some clients read-only, since other, writing, clients may still change the shared object 'behind their back'. To share objects safely you typically have to change clients so that they make new objects rather than changing existing ones (see COPY-ON-WRITE). You should consider removing any public methods or fields that can be used to change shared objects' state: in general, every field should only be initialised in the object's constructor (in Java, all fields should be `final`). The FLYWEIGHT pattern [Gamma 1995] can be used to move dynamic state out of shared objects and into their clients.

2. Establishing Sharing

For two or more components to be able to share some information, each component must be able to find the information that is shared.

In small systems, where only a few distinguished objects are being shared, you can often use a global variable to store each shared object, or store shared instances within the objects' classes using the SINGLETON pattern [Gamma 1995]. The shared global variables can be initialised statically when the program starts, or the first time a shared objects is accessed using LAZY INITIALISATION [Beck 1997].

To provide a more general mechanism you can implement a *shared cache*, an in-memory database mapping from keys to shared objects. To find a shared object, a component checks the cache. If the shared object is already in the cache you use it directly; if not you create the object and store it back into the cache. For this to work you need a unique key for each shared object to identify it in the cache. A shared cache works particularly well when several components are loading the same objects from resource files databases, or networks like the world wide web. Typical keys could be the fully qualified file names, web page URLs, or a combination of database table and database key within that table – the same keys that identify the data being loaded in the first place.

3. Deleting Shared Objects

Once you've created shared objects, you may need to be able to delete them when no longer required.

There are three standard approaches to deleting shared objects: REFERENCE COUNTING, GARBAGE COLLECTION and object ownership. REFERENCE COUNTING keeps a count of the number of objects interested in a shared object; when this becomes zero the object can be released (by removing the reference from the cache and, in C++, deleting the object). A GARBAGE COLLECTOR can detect and remove shared objects without requiring reference counts. Note that if a shared object is accessed via a cache, the cache will always have a reference to the shared object, preventing the garbage collector from deleting it, unless you can use some form of weak reference [Jones and Lins 1996].

With object ownership, you can identify one other single object or component that has the responsibility of managing the shared object (see the SMALL ARCHITECTURE **pattern)**. The object's owner accepts the responsibility of deleting the shared object at the appropriate time; generally it needs to be an object with an overview of all the objects that 'use' the shared object [Weir 1996, Cargill 1996].

4. Sharing Literals and Strings

In many programs literals occupy more space than variables, so you can assign often used literals to variables and then replace the literals by the variables, effectively SHARING one literal in many places. For example, LaTeX uses this technique, coding common literals such as one, two, and minus one as the macros '`\@ne`', '`\tw@`', and '`\m@on`'. Smalltalk shares literals as part of the language environment, by representing strings as 'symbols'. A symbol represents a single unique string, but can be stored internally as if it were an integer. The Smalltalk system maintains a 'symbol table' that maps all known symbols to the strings they represent, and the compilation and runtime system must search this table to encode each string as a symbol, potentially adding a new entry if the string has not been presented before. The Smalltalk environment uses symbols for all method names, which both compresses code and increases the speed of method lookup.

5. Sharing across components and processes

It's more difficult to implement sharing between several components in different address spaces. Most operating systems provide some kind of shared memory, but this is often difficult to use. In concurrent systems, you need to prevent one thread from modifying shared data while another thread is accessing it. Typically this requires at least one semaphore, and increases code complexity and testing cost.

Alternatively, especially when data is shared between many components, you can consider encapsulating the shared data in a component of its own and use client-server techniques to access it. For example, EPOC accesses its relational databases through a single 'database server' process. This server keeps a cache of indexes for its open databases; if two applications use the same database they share the same index.

Example

This Java example outlines part of the simple word processor described above. Documents contain `Paragraphs`, each of which has a `ParagraphFormat`. `ParagraphFormats` are complex objects, so to save memory several `Paragraphs` share a single `ParagraphFormat`. The code shows two mechanisms to ensure this:

- When we duplicate a `Paragraph`, both the original and the new `Paragraph` share a single `ParagraphFormat` instance.
- `ParagraphFormats` are referenced by name, like "bold", "normal" or "heading 2". A Singleton `ParagraphFormatCatalog` contains a map of all the names to `ParagraphFormat` objects, so when we request a `ParagraphFormat` by name, the result is the single, shared, instance with that name.

The most important class in the word processor is Document: basically a sequence of Paragraphs, each of which as a (shared) `ParagraphFormat`.

```
class Document {
    Vector paragraphs = new Vector();
    int currentParagraph = -1;
```

The `Paragraph` class uses a `StringBuffer` to store the text of the paragraph, and also stores a reference to a `ParagraphFormat` object.

```
class Paragraph implements Cloneable {
    ParagraphFormat format;
    StringBuffer text = new StringBuffer();
```

A new `Paragraph` can be constructed either by giving a reference to a format object (which is then stored, without being copied, as the new Paragraph's format) or by giving a format name, which is then looked up in the `ParagraphFormatCatalog`. Note that neither initialising or accessing a paragraph's format copies the `ParagraphFormat` object, rather it is passed by reference.

```
Paragraph(ParagraphFormat format) {
        this.format = format;
    }

    Paragraph(String formatName) {
        this(ParagraphFormatCatalog.catalog().findFormat(formatName));
    }

    ParagraphFormat format() {return format;}
```

`Paragraphs` are copied using the clone method (used by the word-processor to implement its cut-and-paste feature). The clone method only copies one object, so the new clone's fields automatically point to exactly the same objects as the old object's fields. We don't want a `Paragraph` and its clone to share the `StringBuffer`, so we must clone that explicitly and install the cloned `StringBuffer` into the cloned `Paragraph`; however we don't want to clone the `ParagraphFormat` reference, because `ParagraphFormats` can be shared.

```
    public Object clone() {
        try {
            Paragraph myClone = (Paragraph) super.clone();
            myClone.text =  new StringBuffer(text.toString());
            return myClone;
        } catch (CloneNotSupportedException ex) {
            return null;
        }
    }
```

Paragraphs find their formats using the `ParagraphFormatCatalog`, The catalog is a SINGLETON [Gamma et al 1995].

```
class ParagraphFormatCatalog {
    private static ParagraphFormatCatalog systemWideCatalog
        = new ParagraphFormatCatalog();
    public static ParagraphFormatCatalog catalog() {
        return systemWideCatalog;
    }
```

that implements a map from format names to shared `ParagraphFormat` objects:

```
Hashtable theCatalog = new Hashtable();
    public void addNewNamedFormat(String name, ParagraphFormat format) {
        theCatalog.put(name,format);
    }

    public ParagraphFormat findFormat(String name) {
        return (ParagraphFormat) theCatalog.get(name);
    }
}
```

Since the `ParagraphFormat` objects are shared, we want to restrict what the clients can do with them. So `ParagraphFormat` itself is just an interface that does not permit clients to change the underlying object.

```
interface ParagraphFormat {
    ParagraphFormat nextParagraphFormat();
    String defaultFont();
    int fontSize();
    int spacing();
}
```

The class `ParagraphFormatImplementation` actually implements the `ParagraphFormat` objects, and includes a variety of accessor methods and constructors for these variables:

```
class ParagraphFormatImplementation implements ParagraphFormat {
    String defaultFont;
    int fontSize;
    int spacing;
    String nextParagraphFormat;
```

Each `ParagraphFormat` object stores the name of the `ParagraphFormat` to be used for the next paragraph. This makes it easier to initialise the `ParagraphFormat` objects, and will give the correct behaviour if we replace a specific `ParagraphFormat` in the catalogue with another.

To find the corresponding `ParagraphFormat` object, it must also refer to the catalogue

```
public ParagraphFormat nextParagraphFormat() {
        return ParagraphFormatCatalog.catalog().
            findFormat(nextParagraphFormat);
}
```

When the `Document` class creates a new paragraph, it use the shared `ParagraphFormat` returned by the format of the current paragraph: Note that `ParagraphFormat` objects are never copied, so the will be shared between all paragraphs that have the same format.

```
public Paragraph newParagraph() {
        ParagraphFormat nextParagraphFormat =
            currentParagraph().format().nextParagraphFormat();
        Paragraph newParagraph = new Paragraph(nextParagraphFormat);
        insertParagraph(newParagraph);
        return newParagraph;
    }
```

❖ ❖ ❖

Known Uses

Java’s String instances are immutable, so implementations share a single underlying buffer between any number of copies of the same String object [Gosling et al 1996]. And all implementations of Smalltalk use tokens for pre-compiled strings, as discussed above.

C++'s template feature often generate many copies of very similar code, leading to 'code bloat'. Some C++ linkers detect and share such instances of duplicated object code – Microsoft, for example, call this 'COMDAT folding' [Microsoft 1997]. Most modern operating systems provide Dynamic Link Libraries (DLLs) or Shared libraries that allow different processes to use the same code without needing to duplicate it in every executable [Kenah and Bate 1984; Card et al1998].

The EPOC Font and Bitmap server stores font and image data loaded from RESOURCE FILES in shared memory [Symbian 1999]. These are used both by applications and by the Window Server that handles screen output for all applications. Each client requests and releases the font and bitmap data using remote procedure calls to the server process; the server loads the data into shared memory or locates an already-loaded item, and thereafter the application can access it directly (read-only). The server uses reference counting to decide when to delete each item; an application will normally release each item explicitly but the EPOC operating system will also notify the server if the application terminates abnormally, preventing memory leaks.

See Also

SHARING was first described as a pattern in the *Design Patterns Smalltalk Companion* [Alpert, Brown, Woolf 1998].

COPY-ON-WRITE provides a mechanism to change a shared object as seen by one object, without impacting any other objects that rely on it. The READ-ONLY MEMORY pattern describes how you can ensure that objects supposed to be read-only cannot be modified. Shared things are often READ-ONLY, and so often end up stored on SECONDARY STORAGE.

The FLYWEIGHT PATTERN [Gamma et al 1995] describes how to make objects read-only so that they can be shared safely. Objects may also need to be moved into a SINGLE PLACE for modelling reasons [Noble 1997]. Ken Auer and Kent Beck [1996] describe techniques to avoid sharing Smalltalk objects by accident.

Copy-on-Write

How can you change a shared object without affecting its other clients?

- You need the system to behave as if each client has its own mutable copy of some shared data.
- To save memory you want to share data, or
- You need to modify data in Read-Only Memory.

Often you want the system to behave as though there are lots of copies of a piece of shared data, each individually modifiable, even though there is only one shared instance of the data. For example, in the Word-O-Matic word processor, each paragraph has its own format, which users can change independently of any other paragraph. Giving every paragraph its own `ParagraphFormat` object ensures this flexibility, but duplicates data unnecessarily because there are only a few different paragraph formats used in most documents.

We can use the SHARING pattern instead, so that each paragraph format object describes several paragraphs. Unfortunately, a change to one shared paragraph format will change all the other paragraphs that share that format, not just the single paragraph the user is trying to change.

There's a similar problem if you're using READ-ONLY MEMORY (ROM). Many operating systems load program code and read-only data into memory marked as 'read-only', allowing it to be shared between processes; In palmtops and embedded systems the code may be loaded into ROM or flash RAM. Clients may want changeable copies of such data; but making an automatic copy by default for every client will waste memory.

Therefore: *Share the object until you need to change it, then copy it and use the copy in future.*

Maintain a flag or reference count in each sharable object, and ensure it's set as soon as there's more than one client to the object. When a client calls any method that modifies a shared object's externally visible state, create a duplicate of some or all of the object's state in a new object, delegate the operation to that new object, and ensure that the the client uses the new object from then on. The new object will initially not be shared (with flag unset or reference count of one), so further modifications won't cause a copy until the new object in turn then gets multiple clients.

You can implement COPY-ON-WRITE for specific objects in the system, or implement it as part of the operating system infrastructure using PAGING techniques. The latter approach is particularly used with code, which is normally read-only but allows a program to modify its own code on occasion, in which case a paging system can make a copy of part of the code for that specific program instance.

Thus Word-O-Matic keeps a reference count of the number of clients sharing each `ParagraphFormat` object. In normal use many `Document` objects will share the same `ParagraphFormat`, but on the few occasions that a user modifies the format of a paragraph, Word-O-Matic makes a copy of its `ParagraphFormat` and keeps that separate to the modified `Paragraph` and to any other `Paragraphs` with the new format.

Consequences

COPY-ON-WRITE gives programmers the illusion of many copies of a piece of data, without the waste of memory that would imply. So it reduces the *memory requirements* of the system. In some cases it increases a program's *execution speed*, and particularly its *start-up time*, since copying can be a slow operation.

COPY-ON-WRITE also allows you to make it appear that data stored in read-only storage can be changed. So you can move infrequently changed data into read-only storage, reducing the program's *memory requirements.*

Once COPY-ON-WRITE has been implemented it requires little *programmer discipline* to use, since clients don't need to be directly aware of it.

However: COPY-ON-WRITE requires *programmer effort* or *hardware or operating system* support to implement, because the system must intercept writes to the data, make the copy and then continue the write as if nothing had happened.

If there are many write accesses to the data, then COPY-ON-WRITE can decrease *time performance*, since each write access must ensure the data's not shared. COPY-ON-WRITE can also lead to lots of copies of the same thing cluttering up the system, decreasing the *predictability* of the system's performance, making it *harder to test*, and ultimately increasing the system's *memory requirements.*

COPY-ON-WRITE can cause problems for object identity if the identity of the copy and the original storage is supposed to be the same.

❖ ❖ ❖

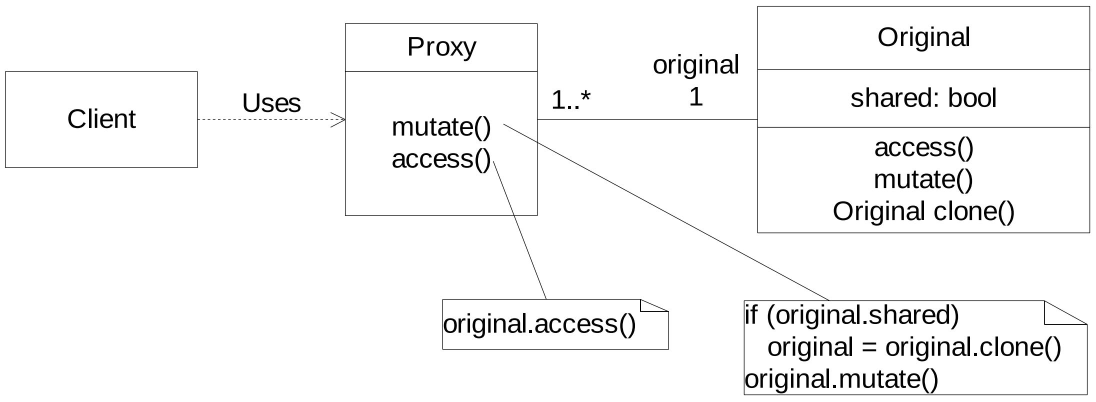
Implementation

Here are some issues to consider when implementing the COPY-ON-WRITE pattern.

1. Copy-On-Write Proxies

The most common approach to implementing COPY-ON-WRITE is to use a variant of the PROXY Pattern [Gamma et al 1995, Buschmann et al 1996, Coplien 1994]. Using the terminology of Bushman et al [1996], a PROXY references an underlying *Original* object and forwards all the messages it receives to that object.

To use PROXY to implement COPY-ON-WRITE, every client uses a different PROXY object, which distinguishes *accessors*, methods that merely read data, from *mutators* that modify it. The Original Object contains a flag that records whether it has more than one Proxy sharing it. Any updator method checks this flag and if the flag is set, makes a (new, unshared) copy of the representation, installs it in the proxy, and forwards the mutator to the that copy instead.



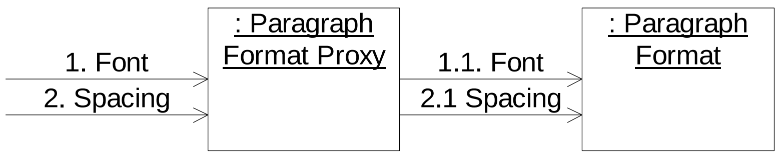
In this design, the `shared` flag is stored in the Original object, and it's the responsibility of the representation's `clone` method to create an object with the flag unset. A valid alternative implementation is to place the flag into the Proxy object; in this case the Proxy must reset the flag after creating and installing a new Original object. As a third option, you can combine the Client and Proxy object, if the Client knows about the use of COPY-ON-WRITE, and if no other objects need to use the Original (other than via the combined Client/Proxy, of course).

You can also combine the function of COPY-ON-WRITE with managing the lifetime of the underlying representation object by replacing the `shared` flag in the Representation object with a reference count. A reference count of exactly one implies the object is not shared and can be modified. See the REFERENCE COUNTING pattern for a discussion of reference counting in detail.

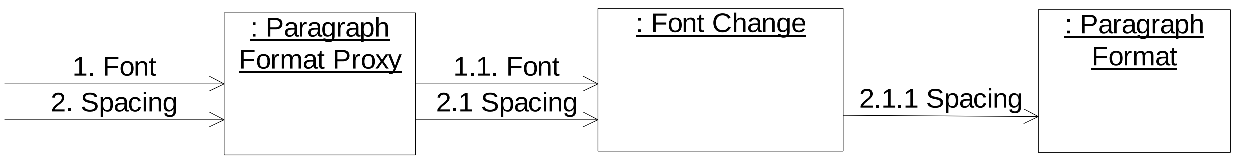
2. Copying Changes to Objects

You do not have to copy all of any object when it is changed. Instead you can create a 'delta' object that stores only the changes to the object, and delegates requests for unchanged data back to the main object. For example, when a user changes the font in a paragraph format, you can create a `FontChange` delta object that returns the new font when it is asked, but forwards all other requests to the underlying, and unchanged, `ParagraphFormat` object. A delta object can be implement as a DECORATOR on the original object [Gamma et al 1995]. The diagram below uses UML shows a possible implementation as a UML Collaboration Diagram [Fowler 1997].

Before:



After:



3. Writing to Objects in Read-Only Memory

You can use COPY-ON-WRITE so that shared representations in ROM can be updated. Clearly the `shared` flag must be set in the ROM instance and cleared in the copy, but otherwise this is no different from a RAM version of the pattern.

In C++ the rule is that only instances of classes without a constructor may be placed in ROM. So a typical implementation must use static initialisation for the flag, and must therefore have public data members. The restriction on constructors means that you can't implement a copy constructor and assignment operator; instead you'll need to write a function that uses the default copy constructor to copy the data.

Example

This example extends the word processor implementation from the SHARING pattern, to allow the user to change the format of an individual paragraph. In this example the `Paragraph` object combines the role of Proxy and Client, since we've restricted all access to the `ParagraphFormat` object to via the `Paragraph` object. We don't need to separate out the read-only aspects to a separate interface as no clients will ever see `ParagraphFormats` directly.

The `Document` class remains unchanged, being essentially a list of paragraphs. The `ParagraphFormat` class is also straightforward, but now it supports mutator methods and needs to implement the `clone` method. For simplicity we only show one mutator – to set the font.

```
class ParagraphFormat implements Cloneable {
    String defaultFont;
    int fontSize;
    int spacing;
    String nextParagraphFormat;

    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }

    void privateSetFont(String aFont) {defaultFont = aFont;}
}
```

As in the previous example, the Paragraph class must also implement cloning. This implementation keeps the shared flag in the `Paragraph` class (i.e. in the Proxy), as the member `paragraphFormatIsUnique`.

```
class Paragraph implements Cloneable {
    ParagraphFormat format;
    boolean paragraphFormatIsUnique = false;

    StringBuffer text = new StringBuffer();

    Paragraph(ParagraphFormat format) {
        this.format = format;
    }
    Paragraph(String formatName) {
        this(ParagraphFormatCatalog.catalog().findFormat(formatName));
    }
```

The `Paragraph` implementation provides two private utility functions: `aboutToShareParagraphFormat` and `aboutToChangeParagraphFormat`. The method `aboutToShareParagraphFormat` should be invoked whenever we believe it's possible that we may be referencing a `ParagraphFormat` object known to any other object.

```
    protected void aboutToShareParagraphFormat() {
        paragraphFormatIsUnique = false;
    }
```

If any external client obtains a reference to our `ParagraphFormat` object, or passes in one externally, then we must assume that it's shared:

```
    ParagraphFormat format() {
        aboutToShareParagraphFormat();
        return format;
    }

    public void setFormat(ParagraphFormat aParagraphFormat) {
        aboutToShareParagraphFormat();
        format = aParagraphFormat;
    }
```

And similarly, if a client clones this `Paragraph` object, we don't want to clone the format, but instead simply note that we're sharing it:

```
    public Object clone() {
        try {
            aboutToShareParagraphFormat();
            Paragraph myClone = (Paragraph) super.clone();
            myClone.text =  new StringBuffer(text.toString());
            return myClone;
        } catch (CloneNotSupportedException ex) {
            return null;
        }
    }
```

Meanwhile any method that modifies the `ParagraphFormat` object must first call `aboutToChangeParagraphFormat`. This method makes sure the `ParagraphFormat` object is unique to this Paragraph, cloning it if necessary.

```
protected void aboutToChangeParagraphFormat() {
    if (!paragraphFormatIsUnique) {
        try {
            format = (ParagraphFormat) format().clone();
        } catch (CloneNotSupportedException e) {}
        paragraphFormatIsUnique = true;
    }
}
```

Here's a simple example of a method that modifies a `ParagraphFormat`:

```
void setFont(String fontName) {
    aboutToChangeParagraphFormat();
    format.privateSetFont(fontName);
}
```

❖ ❖ ❖

Known Uses

Many operating systems use COPY-ON-WRITE in their paging systems. Executable code is very rarely modified, so it's usually SHARED between all processes using it, but this pattern allows modification when processes need it. By default each page out of an executable file is flagged as read-only and shared between all processes that use it. If a client writes to a shared page, the hardware generates an exception, and operating system exception handler then creates a writable copy for that process alone. [Kenah and Bate 1984; Goodheart and Cox 1994]

RogueWave's Tools.h++ library uses COPY-ON-WRITE for its `CString` class [RogueWave 1994]. A `CString` object represents a dynamically allocated string. C++'s pass-by-value semantics mean that the `CString` objects are copied frequently, but very seldom modified. So each `CString` object is simply a wrapper referring to a shared implementation. `CString`'s copy constructor and related operators manipulate a reference count in the shared implementation. If any client does an operation to change the content of the string; the `CString` object simply makes a copy and does the operation on the copy. One interesting detail is that there is only one instance of the null string, which is always shared. All attempts to create a null string, for example by initialising a zero length string, simply access that shared object.

Because modifiable strings are relatively rare in programs, Sun Java implements them using a separate class, `StringBuffer`. However `StringBuffer` permits it's clients to retrieve `String` objects with the method `toString`. To save memory and speed up performance the resulting `String` uses the underlying buffer already created by `StringBuffer`. However the `StringBuffer` object has a flag to indicate that the buffer is now shared; if a client attempts to make further changes to the buffer, `StringBuffer` creates a copy and uses that. [Chan et al 1998]

Objects in NewtonScript were defined using inheritance, so that common features could be declared in a parent object and then shared by all child objects that needed them. Default values for objects' fields were defined using copy-on-write slots. If a child object didn't define a field it would inherit that field's value from its parent object, but when a child object wrote to a shared field a local copy of the field was automatically created in the child object. [Smith 1999].

See Also

HOOKS provide an alternative technique for changing the contents of read-only storage.

Embedded Pointer

How can you reduce the space used by a collection of objects?

- Linked data structures are built out of pointers to objects
- Collection objects (and their internal link objects) occupy large amounts of memory to store large collections.
- Traversing through a linked data structure can require temporary memory, especially if the traversal is recursive.

Object-Oriented programs implement relationships between objects by using collection objects that store pointers to other objects. Unfortunately, collection objects and the objects they use internally can require a large amount of memory. For example, the Strap-It-On's 'Mind Reader' brainwave analysis program must receive brainwave data in real-time from an interrupt routine, and store it in a list for later analysis. Because brainwaves have to be sampled many times every second, a large amount of data can accumulate before it can be analysed, even though each brainwave sample is relatively small (just a couple of integers). Simple collection implementations based on linked lists can impose an overhead of at least three pointers for every object they store, so storing a sequence of two-word samples in such a list more than doubles the sequence's intrinsic memory requirements – see figure xx below.

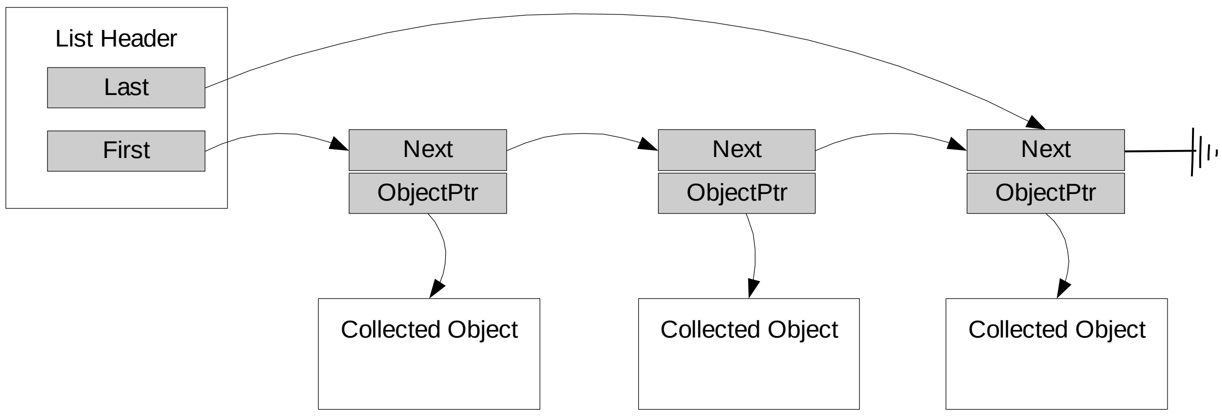


Figure 2: Linked list using external pointers

As well as this memory overhead, linked data structures have other disadvantages. They can use large numbers of small internal objects, increasing the possibility of fragmentation (see Chapter N). Allocating all these objects takes an unpredictable amount of time, making it unsuitable for real-time work. Traversing the structure requires following large numbers of pointer links; this also takes time, but more importantly, traversals of recursive structures like graphs and trees can also require an unbounded amount of temporary memory; in some cases, similar amounts of memory to that required to store the structure itself. Finally, any function that adds an object to such a collection may fail if there is insufficient memory, and so must carry all the costs of PARTIAL FAILURE.

Of course, linked structures have many compensating advantages. They can describe many different kinds of structures, including linked lists, trees, queues, all of a wide variety of different subtypes [Knuth 1997]. These structures can support a wide variety of operations quite efficiently (especially insertions and deletions in the middle of the data). Linked structures support VARIABLE ALLOCATION, so they never need to allocate memory that is subsequently unused. The only real alternative to building linked structures is to use some kind of FIXED ALLOCATION, such as a fixed-size array. But fixed structures place arbitrary limits on the number of objects in the collection, waste memory if they are not fully occupied, and insertion and deletion operations can be very expensive. So, how can you keep the benefits of linked data structures while minimising the disadvantages?

Therefore: *Embed the pointers maintaining the collection into each object.*

Design the collection data structure to store its pointers within the objects that are contained in the structure, rather than in internal link objects. You will need to change the definitions of the objects that are to be stored in the collection to include these pointers (and possibly to include other collection-related information as well).

You will also need to change the implementation of the collection object to use the pointers stored directly in objects. For a collection that is used by only one external client object, you can even dispense completely with the object that represents the collection, and incorporate its data and operations directly into the client. To traverse the data structure, use iteration rather than recursion to avoid allocating stack frames for every recursive call, and use extra (or reuse existing) pointer fields in the objects to store any state related to the traversal.

So, for example, rather than store the Brainwave sample objects in a collection, Strap-it-On's Brainwave Driver uses an embedded linked list. Each Brainwave sample object has an extra pointer field, called `Next`, that is used to link brainwave samples into a linked list. As each sample is received, the interrupt routine adjusts its `Next` field to link it into the list. The main analysis routine adjusts the sample object's pointers to remove each from the list in its own time for processing.

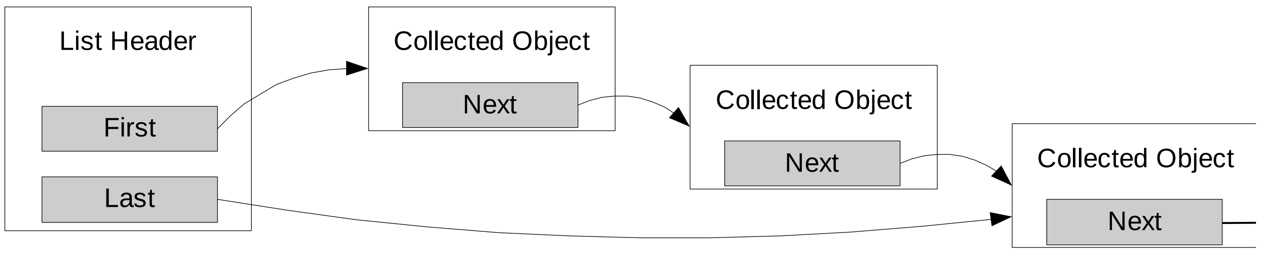


Figure 3: Linked list using embedded pointers

Consequences

Embedded pointers remove the need for internal link objects in collections, reducing the number of objects in the system and thus the system's *memory requirements*, while increasing the *predictability* of the systems memory use (especially if traversals are iterative rather than recursive). The routines to add and remove items from the linked list cannot suffer memory allocation failure.

Using embedded pointers reduces or removes the need for dynamic memory allocation, improving the *real-time performance* of the system. Some operations may have better *run-time performance*; for example with an embedded doubly-linked list you can remove an element in the collection simply by using a pointer to that element directly. With an implementation using external pointers (such as STL's Deque [Austern 1998]) you'd need first to set an iterator to refer to the right element, which requires a linear search.

However: Embedded pointers don’t really belong to the objects they are embedded inside. This pattern reduces those objects’ encapsulation, gaining a *local* benefit but reducing the *localisation* of the design. The pattern tightly couples objects to the container class that holds them, making it more difficult to reuse either class independently, increasing the *programmer effort* required because specialised collections often have to be written from scratch, reducing the *design quality* of the system and making the program harder to *maintain*.

In many cases a given collected object it will often need to be in several different collections at different times during its lifetime. It requires *programmer discipline* to ensure that the same pointer is never used by two collections simultaneously.

❖ ❖ ❖

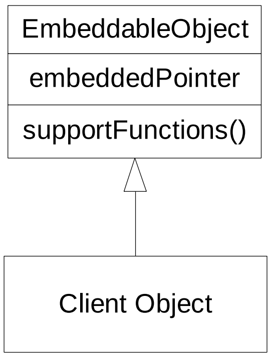
Implementation

Applying the Embedded Pointer pattern is straightforward: place pointer members into objects and build up linked data structures using those pointers, instead of using external collection objects. You can find the details in any decent textbook on data structures from Knuth [1997], which will describe the details, advantages, and disadvantages of the classical linked data structure designs, from simple singly and doubly linked lists to subtle complex balanced trees. See the SMALL DATA STRUCTURES pattern for a list of such textbooks.

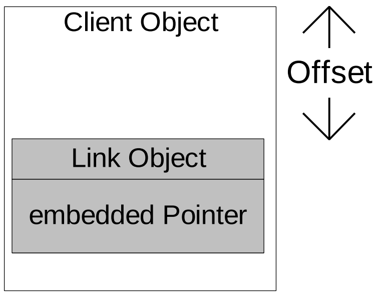
1. Reuse

The main practical issue when using embedded pointers is how to incorporate the pointers into objects in a way that provides some measure of reuse, to avoid re-implementing all the collection operations for every single list. The key idea is for objects to somehow present a consistent interface for accessing the embedded pointers to the collection class (or the functions that implement the collection operations). In this way, the collection can be used with any object that provides a compatible interface. There are three common techniques for establishing interfaces to embedded pointers: inheritance, inline objects, and preprocessor constructs.

1.1. Inheritance. You can put the pointers and accessing functionality into a superclass, and make the objects to be stored in a collection inherit from this class. This is straightforward, and provides a measure of reuse. However: you can’t have more than one instance of such a pointer for a given object; if the pointer is implementing a collection, this would limit. In single-inheritance languages like Smalltalk it also prevents any other use of inheritance for the same object, and so limits any object to be in only one collection at a time. In languages with multiple inheritance objects could be in multiple collections provided each collection accesses the embedded pointers through a unique interface, supplied by a unique base class (C++) or interface (Java).



1.2. Inline Objects. In languages with inline objects, like C and C++, you can embedded a separate 'link' object that contains the pointers directly into the client object. This doesn't suffer from the disadvantages of using Inheritance, but you need to be able to find the client object from a given link object and vice versa. In C++ this can be implemented using pointers to members, or (more commonly) as an offset in bytes.



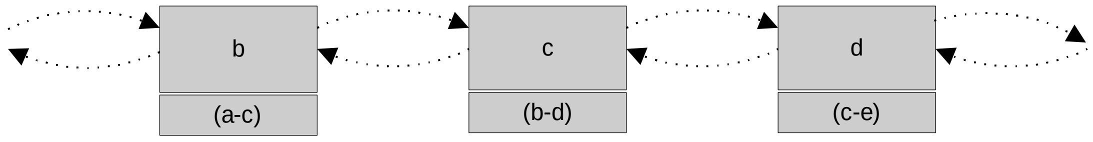
For example, EPOC's collection libraries find embedded pointers using byte offsets. Whenever a new collection is created, it must be initialised with the offset inside its client objects where its pointers are embedded.

1.3. Preprocessors. C++ provides two kinds of preprocessing: the standard preprocessor `cpp`, and the C++ template mechanisms. So in C++ a good approach is to include the embedded pointers as normal (possibly public) data members, and to reuse the management code via preprocessing. You can also preprocess code in almost any other language given a suitable preprocessor , which could be either a special purpose program like `m4`, or a general purpose program like `perl`.

2. Pointer Differences

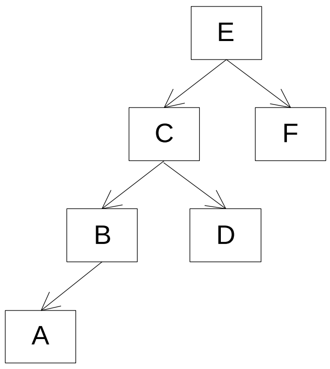
Sometimes an object needs to store two or more pointers; for example a circular doubly-linked list node needs pointers to the previous and next item in the list. You can reduce the amount of memory needed to store by storing the difference (or the bitwise exclusive or) of the two pointers, rather than the pointer itself. When you are traversing the structure forwards, for example, you take the address of the previous node and add the stored difference to find the address of the next node; reverse traversals work similarly.

For example, in Figure XXX, rather than node `c` storing the dotted forward and back pointers (i.e. the addresses of nodes `b` and `d`) node c stores only the difference between these two addresses. Given a pointer to node `b` and the difference stored within `c`, you can calculate the address of node d as (b – (d-c)). Similarly, traversing the list the other way, given the address of node d and (b-c) you can calculate the address of node b as (d+(b-d)). For this to work, you need to store two initial pointers, typically a head and tail pointer for a circular doubly-linked list [Knuth 1997].

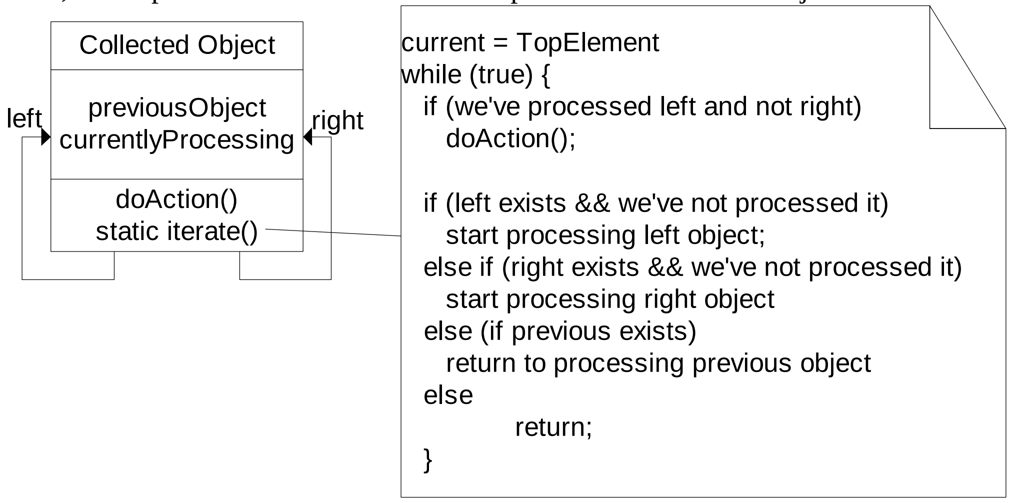


(3) Traversals

A related, but different, problem happens when you need to traverse an arbitrarily deep structure, especially if the traversal has to be recursive. Suppose for example you have an unbalanced binary tree, and you need to traverse through all the elements in order. A traversal beginning at E will recursively visit C, then F; the traversal at C will visit B and D, and so on. Every recursive call requires extra memory to store activation records on the stack, so traversing larger structures can easily exhaust a process's stack space.



3.1. Iterative traversals using extra embedded pointers. Consider the traversal more closely: at each object it needs to store one thing on the stack: the identity of the object its coming from (and possibly any working data or parameters passed through the iteration). So for example, when C invokes the operation on D, it must store that it needs to return to E on completion. You can use Embedded Pointers in each object to store this data (the parent, and the traversal state — two Boolean flags that remember whether the left and right leaves have been processed). This allows you to iterate over the structure using a loop rather than recursion, but imposes the overhead of an extra pointer or two in each object.

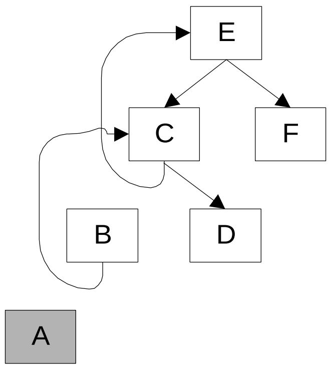


```
current = TopElement
while (true) {
  if (we've processed left and not right)
    doAction();

  if (left exists && we've not processed it)
    start processing left object;
  else if (right exists && we've not processed it)
    start processing right object
  else (if previous exists)
    return to processing previous object
  else
          return;
  }
```

3.2. Iterative traversals using pointer reversal. Consider the iteration process further. At any time one of the three pointers in each element, left leaf, right leaf or parent, is redundant. If there is no iteration, the parent pointer is redundant; if a left or right left is currently being processed that leaf pointer is redundant (because the traversal has already reached that leaf). Pointer reversal allows iterative traversals of linked structures by temporarily using pointers to leaf nodes as parent pointers: as the traversal proceeds around the object, the pointers currently being followed are 'reversed', that is, used to point to parent objects.

In the figure above, for example, when a traversal is at node A, node B's left leaf pointer would be reversed to point to its parent, node C; because B is C's left child, C's left child pointer would also be reversed to point to node E.



Example

Here's an example using Embedded Pointers to store a data structure and traverse it using pointer reversal. The example program implements a sorting algorithm using a simple binary tree. To start with we'll give the objects that are stored in the tree (the `BinaryTreeObjects`) a single character of local data. We also need a `greaterThan` operation and a operation called to do the action we need (`doIt`).

```
class BinaryTreeObject {
    char data;

    BinaryTreeObject(char data) {
        this.data = data;
    }

    Object doIt(Object param) {
        return ((String) param + data);
    }

    boolean greaterThan(BinaryTreeObject other) {
        return data > other.data;
    }
```

A binary tree needs a left pointer and a right pointer corresponding to each node. Using the Embedded Pointers pattern, we implement each within the structure itself:

```
    BinaryTreeObject left;
    BinaryTreeObject right;
```

Adding an element to the binary tree is fairly easy. We can traverse the tree starting at the top, going left when our new element is less than the current item, greater when it's greater, until we get to a vacant position in the tree.

```
static void insert(BinaryTreeObject top, BinaryTreeObject newItem) {
        BinaryTreeObject current = top;

        for (;;) {
            if (current.greaterThan(newItem)) {
                if (current.left == null) {
                    current.left = newItem;
                    return;
                } else {
                    current = current.left;
                }
            } else {
                if (current.right == null) {
                    current.right = newItem;
                    return;
                } else {
                    current = current.right;
                }
            }
        }
    }
```

Note that this method is not recursive and so should allocate no memory other than one stack frame with one local variable (`current`).

Traversing the tree is more difficult, because the traversal has to visit all the elements in the tree, and this means backtracking up the tree when it reaches a bottom-level node. To traverse the tree without using recursion, we can add two embedded pointers to every tree node: a pointer to the previous (parent) item in the tree, and a marker noting which action, left node or right node, the algorithm is currently processing.

```
BinaryTreeObject previous;
    static final int Inactive = 0, GoingLeft = 1, GoingRight = 2;
    int action = Inactive;
```

The `traversal` method, then, must move through each node in infix order. Each iteration visits one node; however this may mean up to three visits to any given node (from parent going left, from left going right, and from right back to parent); we use the stored action data for the node to see which visit this one is. The `traversal` method must also call the `doIt` method at the correct point – after processing the left node, if any.

```
static Object traversal(BinaryTreeObject start, Object param) {
        BinaryTreeObject current = start;

        for (;;) {

            if (current.action == GoingLeft ||
                (current.action == Inactive && current.left == null)) {
                param = current.doIt(param);
            }

            if (current.action == Inactive && current.left != null) {
                current.action = GoingLeft;
                current.left.previous = current;
                current = current.left;
            } else if (current.action != GoingRight && current.right != null) {
       current.action = GoingRight;
       current.right.previous = current;
       current = current.right;
       }  else {
       current.action = Inactive;
       if (current.previous == null) {
           break;
       }
       current = current.previous;
       }

        }
        return param;
}
```

Of course, a practical implementation would improve this example in two ways. First we can put the `left`, `right`, `action`, `previous` pointers and the `greaterThan` stub into a base class (`SortableObject`, perhaps) or into a separate object. Second we can make the `traversal()` method into a separate ITERATOR object [Gamma et al 1995], avoiding the need to hard-code the `doIt` method.

We can extend this example further, to remove the parent pointer from the data structure using Pointer Reversal. First, we'll need two additional methods, to save the parent pointer in either the left or the right pointer:

```
BinaryTreeObject saveParentReturningLeaf(BinaryTreeObject parent) {
        BinaryTreeObject leaf;

        if (action == GoingLeft) {
            leaf = left;
            left = parent;
        } else {
            leaf = right;
            right = parent;
        }
        return leaf;
    }
```

and then to restore it as required:

```
BinaryTreeObject restoreLeafReturningParent(BinaryTreeObject leafJustDone) {
        BinaryTreeObject parent;

        if (action == GoingLeft) {
            parent = left;
            left = leafJustDone;
        } else {
            parent = right;
            right = leafJustDone;
        }
        return parent;
    }
```

Now we can rewrite the `traversal` method to remember the previous item processed, whether it's the parent of the current item or a leaf node, and to reverse the left and right pointers using the methods above:

```
static Object reversingTraversal(BinaryTreeObject top, Object param) {

        BinaryTreeObject current = top;
        BinaryTreeObject leafJustDone = null;
        BinaryTreeObject parentOfCurrent = null;

        for (;;) {

            if (current.action == GoingLeft ||
                (current.action == Inactive && current.left == null)) {
                param = current.doIt(param);
            }

            if (current.action != Inactive)
                parentOfCurrent = current.restoreLeafReturningParent(leafJustDone);

            if (current.action == Inactive && current.left != null) {
                current.action = GoingLeft;
                BinaryTreeObject p = current;
                current = current.saveParentReturningLeaf(parentOfCurrent);
                parentOfCurrent = p;
            } else if (current.action != GoingRight && current.right != null) {
        current.action = GoingRight;
        BinaryTreeObject p = current;
        current = current.saveParentReturningLeaf(parentOfCurrent);
        parentOfCurrent = p;
            } else {
                current.action = Inactive;
                if (parentOfCurrent == null) {
                    break;
                }
                leafJustDone = current;
                current = parentOfCurrent;
            }

        }
        return param;
    }
```

We're still wasting a word in each object for the 'action' parameter. In Java we could perhaps reduce this to a byte but no further. In a C++ implementation we could use the low bits of, say, the left pointer to store it (see PACKED DATA – packing pointers), thereby reducing the overhead of the traversing algorithm to nothing at all.

❖ ❖ ❖

Known Uses

EPOC provides at least three different 'linked list' collection classes using embedded pointers [Symbian 1999]. The embedded pointers are instances of provided classes (`TSglQueLink`, for example) accessed via offsets; the main collection logic are in separate classes, which use the 'thin template idiom' to provide type safety: `TSglQueue<MyClass>`. EPOC applications, and operating system components, use these classes extensively. The most common reason for preferring them over collections requiring heap memory is that operations using them cannot fail; this is a significant benefit in situations where failure handling is not provided.

The Smalltalk `LinkedList` class uses inheritance to mix in the pointers; the only things you can store into a `LinkedList` are objects that inherit from class Link [Goldberg and Robson 1983]. Class `Link` contains two fields and appropriate accessors (`previous` and `next`) to allow double linking. Compared with other Smalltalk collections, for each element you save one word of memory by using concatenation instead of pointers, plus you save the memory overhead of creating a new object (two words or so) and the overhead of doing the allocation.

See Also

You may be able to use FIXED ALLOCATION to embed objects directly into other objects, rather than just embedding pointers to objects.

Pointer reversal was first described by Peter Deutsch [Knuth 1997] and Schorr and Waite [1967]. Embedded Pointers and pointer reversal are used together in many implementations of GARBAGE COLLECTION [Goldberg and Robson 1983, Jones and Lins 1996]. Jiri Soukup discusses using preprocessors to implement linked data structures in much more detail [1994].

Multiple Representations

How can you support several different implementations of an object?

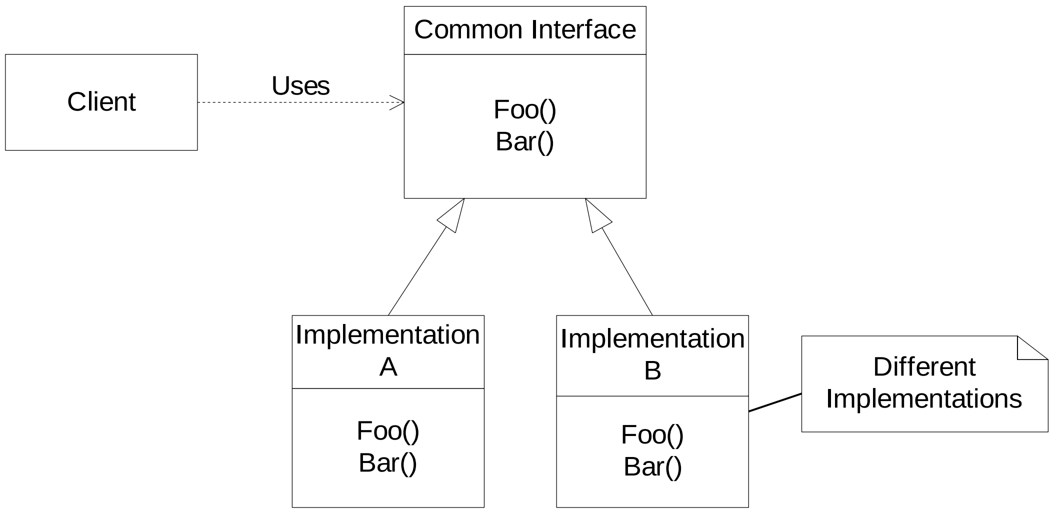
- There are several possible implementations of a class, with different trade-offs between size and behaviour.
- Different parts of your system, or different uses of the class, require different choices of implementation. One size doesn't fit all.
- There are enough instances of the class to justify extra code to reduce RAM usage.

Often when you design a class, you find there can be several suitable representations for its internal data structures. For example, in the Strap-It-On's word-processor (Word-O-Matic) a word may be represented as a series of characters, a bitmap, or a sequence of phonemes. Depending on the current output mechanism (a file, the screen, or the vocaliser) each of these representations might be appropriate.

Having to choose between several possible representations is quite common. Some representations may have small *memory requirements*, but be costly in processing time or other resources; others may be the opposite. In most cases you can examine the demands of the system and decide on a best SMALL DATA STRUCTURE. But what do you do when there's no single 'best' implementation?

Therefore: *Make each implementation satisfy a common interface.*

Design a common abstract interface that suits all the implementations without depending on a particular one, and ensure every implementation meets the interface. Access implementations via an ABSTRACT CLASS [Woolf 2000] or use ADAPTERS to access existing representations [Gamma et al 1995] so clients don't have to be aware of the underlying implementation.



For example, Word-O-Matic defines a single interface '`Word`', which is used by much of the word-processing code. Several concrete classes, `StorableWord`, `ViewableWord`, `SpokenWord`, that implement the `Word` interface. Each implementation has different internal data structures and different implementations of the operations that access those structures. The software creates whichever concrete class is appropriate for the current use of the Word object, but the distinction is only significant when it comes to outputting the object. The multiple implementations are concealed from most of the client code.

Consequences

The system will use the most appropriate implementation for any task, reducing the *memory requirements* and *processing time* overheads that would be imposed by using an inappropriate representation. Code using each instance will use the common interface and need not know the implementation, *reducing programmer effort* on the client side and increasing *design quality* and *reusability*.

Representations can be chosen *locally* for each data structure. More memory-intensive representations can be used when more memory is available, adding to the *scalability* of the system.

However: The pattern can also increase total *memory requirements*, since the code occupies additional memory.

MULTIPLE REPRESENTATIONS increases *programmer effort* in the implementation of the object concerned, because multiple implementations are more *complex* than a single implementation, although this kind of complexity is often seen as a sign of *high-quality* design because subsequent changes to the representation will be easier. For the same reason, it increases *testing costs* and *maintenance costs* overall, because each alternative implementation must be tested and maintained separately.

Changing between representations imposes a *space and time* overhead. It also means *more complexity* in the code, and *more complicated testing* strategies, increasing *programmer effort* and making memory use *harder to predict*.

❖ ❖ ❖

Implementation

There are number of issues to take into account when you are using MULTIPLE REPRESENTATIONS.

1. Implementing the Interface.

In Java the standard implementation of dynamic binding means defining either a Java class or a Java interface. Which is more suitable? From the point of view of the client, it doesn't matter; either can define an abstract interface. Using a Java interface gives you more flexibility, because each implementation may inherit from other existing classes as required; however, extending a common superclass allows several implementations to inherit common functionality. In C++ there's only the one conventional option for implementing the common interface: making all implementations inherit from a base class that defines the interface.

There's a danger that clients may accidentally rely on features on a particular implementation – particularly non-functional ones – rather than of the common interface. D'Souza and Wills [1998] discuss design techniques to avoid such dependencies in components.

2. Binding clients to implementations.

Sometimes you need to support several implementations, though a given client may only ever use one. For example, the C++ Standard Template Library (STL) iterator classes work on several STL collections, but any given STL iterator object works with only one [Stroustrup 1997, Austern 1998]. In this case, you can statically bind the client code to use only the one implementation — in C++ you could store objects directly and use non-virtual functions. If, however, a client needs to use several different object representations interchangeably then you need to use dynamic binding.

3. Creating dynamically bound implementations

The only place where you need to reference the true implementation classes in the code is where the objects are created. In many situations, it's reasonable to hard code the class names in the client, as in the following C++ example:

```
CommonInterface *anObject = new SpecificImplementation( parameters );
```

If there's a good reason to hide even this mention of the classes from the client, then the ABSTRACT FACTORY pattern [Gamma et al 1995] can implement a 'virtual constructor' [Coplien 1994] so that the client can specify which object to create using just a parameter.

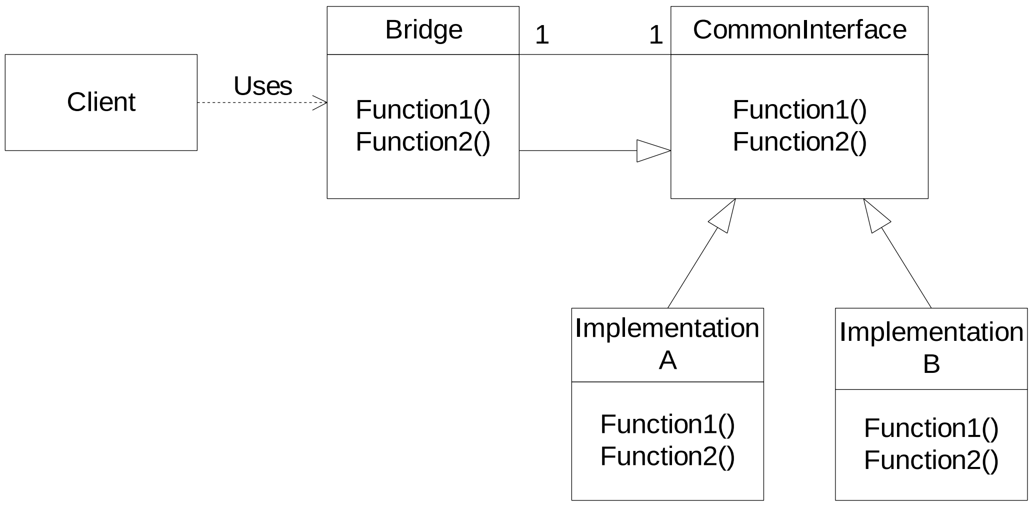
4. Changing between representations

In some cases, an object's representation needs to change during its lifetime, usually because a client needs some behaviour that is not supported well by the object's current representation. Changes to an object's representation can be explicitly requested by its client, or can be triggered automatically within the object itself. Changing representations automatically has several benefits: the client doesn't need knowledge of the internal implementation, improving *encapsulation*, and you can tune the memory use entirely within the implementation of the specific object, improving *localisation*. Changing representations automatically requires dynamic binding, so clients will use the correct representation without being aware of it. In some situations, however, the client can have a better knowledge of optimisation strategies than is available to the object itself, typically because the client is in a better position to know which operations will be required.

4.1. Changing representations explicitly. It is straightforward for an object to let a client change its representation explicitly: the object should implement a conversion function (or a C++ constructor) that takes the common interface as parameter, and returns the new representation.

```
class SpecificImplementation : public CommonInterface
{ public:
    SpecificImplementation( CommonInterface c ) {
    // initialise this from c
     }
};
```

4.2. Changing representations automatically. You can use the BRIDGE pattern to keep the interface and identity of the object constant when its internal structure changes (strictly speaking a 'half bridge', since it varies only the object implementation and not the abstraction it supports) [Gamma et al 1995]. The client sees only the bridge object, which delegates all its operations to an implementation object through a common interface:



The bridge class needs the same methods as the common interface; so it's reasonable (though unnecessary) in C++ and Java to make the Bridge class derive from the common interface. Each implementation object will need a construction function taking the common interface as parameter. Of course, some implementations may store more or less data, so there may be special cases with more specific constructors.

Some languages make it simple to implement the Bridge object itself. In Smalltalk, for example, you can override the `DoesNotUnderstand:` method to pass any unrecognised operation on to the implementation object [Lalonde 1994]. In C++ you can implement `operator->()` to do the same [Coplien 1994], or alternatively you can avoid deriving the Bridge class from the common interface and by make all its functions non-virtual and inline.

Example

This Java example implements a Word object with two representations: as a simple string (the default), and as a string with an additional cached corresponding sound. Both these representations implement the basic Word interface, which can return either a string or sound value, and allows clients to choose the most appropriate representation.

```
interface WordInterface
{
    public byte[] asSound();
    public String asString();
    public void becomeSound();
    public void becomeString();
}
```

The most important concrete class is the Word class, that acts as a bridge between the Word abstraction an its two representations, as a sound and as a text string.

```
class Word implements WordInterface {
    private WordInterface rep;

    public byte[] asSound()     {return rep.asSound();}
    public String asString()    {return rep.asString();}
    public void becomeSound()   {rep.becomeSound();}
    public void becomeString()  {rep.becomeString();}
```

The constructor of the `Word` class must select an implementation. It uses the method `Become`, which simply sets the implementation object.

```
public Word(String word) {
        become(new StringWordImplementation(this, word));
    }
    public void become(WordInterface rep) {
        this.rep = rep;
    }
```

The default implementation stores `Words` as a text string. It also keeps a pointer to its Word BRIDGE object, and uses this pointer to automatically change a word's representation into the other format. It has two constructors: one, taking a string, is used by the constructor for the `Word` object; the other, taking a `WordInterface`, is used to create itself from a different representation.

```
class StringWordImplementation implements WordInterface
{
    private String word;
    private Word bridge;

    public StringWordImplementation(Word bridge, String word) {
        this.bridge = bridge;
        this.word = word;
    }

    public StringWordImplementation(Word bridge, WordInterface rep) {
        this.bridge = bridge;
        this.word = rep.asString();
    }
```

It must also provide implementations of all the `WordInterface` methods. Note how it must change its representation to return itself as a sound; once the `asSound` method returns this object will be garbage:

```
public byte[] asSound()
    {
        becomeSound();
        return bridge.asSound();
    }
    public String asString() {return word;}

    public void becomeSound() {
        bridge.become(new SoundWordImplementation(bridge, this));
    }
    public void becomeString() {}
```

Finally, the sound word class is similar to the text version, but also caches the sound representation. Implementing the sound conversion function is left as an exercise for the reader!

```
class SoundWordImplementation implements WordInterface
{
    private String word;
    private Word bridge;
    private byte[] sound;

    SoundWordImplementation(Word bridge, WordInterface rep) {
        this.bridge = bridge;
        this.word = rep.asString();
        this.sound = privateConvertStringToSound(this.word);
    }

    public String asString() {return word;}
    public byte[] asSound()  {return sound;}
    public void becomeString() {
        bridge.become(new StringWordImplementation(bridge, this));
    }
    public void becomeSound() {}
}
```

❖ ❖ ❖

Known Uses

Symbian's EPOC C++ environment handles strings as *Descriptors* containing a buffer and a length. Descriptors provide many different representations of strings: in ROM, in a fixed-length buffer, in a variable length buffer and as a portion of another string. Each kind of descriptor has its own class, and users of the strings see only two base classes: one for a read-only string, the other for a writable string [Symbian 1999].

The Psion 5's Word Editor has two internal representations of a document. When the document is small the editor keeps formatting information for the entire document; when the document is larger than a certain arbitrary size, the editor switches to storing information for only the part of the document currently on display. The switch is handled internally to the Editor's 'Text View' component; clients of the component (including other applications that need rich text) are unaware of the change in representation.

Smalltalk's collection classes also use this pattern: all satisfy the same protocol, so a user need not be aware of the particular implementation used for a given collection [Goldberg and Robson 1983]. Java's standard collection classes have a similar design [Chan et al 1998]. C++'s STL collections also use this pattern: STL defines the shared interface using template classes; all the collection classes support the same access functions and iterator operations [Stroustrup 1997, Austern 1998].

Rolfe&Nolan's Lighthouse system has a 'Deal' class with two implementations: by default an instance contains only basic data required for simple calculations; on demand, it extends itself reading the entire deal information from its database. Since clients are aware when they are doing more complex calculations, the change is explicit, implemented as a `FattenDeal` method on the object.

MULTIPLE REPRESENTATIONS can also be useful to implement other memory saving patterns. For example the LOOM Virtual Memory system for Smalltalk uses two different representations for objects: one for objects completely in memory, and a second for objects PAGED out to SECONDARY STORAGE [Kaehler and Krasner 1983]. Format Software's PLUS application implements CAPTAIN OATES for images using three representations, which change dynamically: a bitmap ready to `bitblt` to the screen, a compressed bitmap, and a reference to a representation in the database.

See Also

The BRIDGE pattern describes how abstractions and implementations can vary independently [Gamma et al 1994]. The MULTIPLE REPRESENTATIONS pattern typically uses only half of the BRIDGE pattern, because implementations can vary (to give the multiple representations) but the abstraction remains the same.

Various different representations can use explicit PACKED DATA or COMPRESSION, be stored in SECONDARY STORAGE, be READ-ONLY, or be SHARED. They may also use FIXED ALLOCATION or VARIABLE ALLOCATION.