Shadows: A Type-safe Framework for Dynamically Extensible Objects

Jonathan J. Gibbons
Michael J. Day

SMLI TR-94-31 November 1994

Abstract:

In an object-oriented program, it is common to have a collection of interconnected objects of a variety of types. To manipulate such a collection, it is often desirable to be able to extend the functionality of the individual objects, in different ways for different and independent clients, and possibly for more than one client at a time. The complete set of potential clients may not be known when the code for the collection is compiled, or when the collection is actually built. Furthermore, it is desirable to be able to extend the functionality of the various objects in a type-safe manner.

"Shadowing" is a flexible way to solve this problem that permits a collection of objects to be projected from one type-space to another. Internally, a simple form of run-time typing is used to provide type-safety. Both the shadow technology and the run-time typing technology use a specialized utility called `autodefine` that automates many of the implementation details.

A Sun Microsystems, Inc. Business
M/S 29-01
2550 Garcia Avenue
Mountain View, CA 94043

email addresses:
jonathan.gibbons@eng.sun.com
michael.day@eng.sun.com

About this document

Chapter 1: Background **1**

The rationale for shadows is explored by examining alternative solutions, their advantages and disadvantages.

Chapter 2: Shadows **6**

A general and language-independent overview of the shadow mechanism.

Chapter 3: C++ Implementation. **11**

The various classes to implement shadows are presented, together with instructions on who gets to write what.

Chapter 4: Behind the Scenes **23**

The implementation is explained by tracing the execution of calls made by a client.

Chapter 5: Advanced Usage **27**

Extra tips and techniques that may prove useful in specific circumstances.

Chapter 6: Acknowledgements **31**

Appendix A: Narrow **32**

A brief description of a facility to provide run-time typing.

Appendix B: Autodefine **34**

Some of the internal mechanism of shadows is boiler-plate code defined on client classes. `autodefine` is a utility to mechanically generate such code.

Background 1

In an object-oriented program, it is common to have a collection of interconnected objects of a variety of types. To manipulate such a collection, it is often desirable to be able to extend the functionality of the individual objects, in different ways for different and independent clients, and possibly for multiple clients at a time. The complete set of potential clients may not be known when the code for the collection is compiled, or when the collection is actually built. Furthermore, it is desirable to be able to extend the functionality of the various objects in a type-safe manner.

There is another aspect to the problem. Even if it were possible to modify the objects to support all the desired extra functionality, it might not be desirable to do so. To add numerous methods (and the instance data required by their implementation) can lead to an effect

called *interface bloat*, in which objects pay the cost of all that functionality whether they use it or not.

The problem is not specific to any one programming language, but is exacerbated by the presence of inheritance. This is because inheritance introduces the difference between the static and the dynamic type of an object. In this note, C++ is used for the examples and for framing the eventual solution.

There are a variety of solutions to the problem, each with their own features and their own shortcomings.

1.1 Extending the data of nodes

1.1.1 Client data

The simplest way to provide extensible data for the nodes of a data structure is to permit each node to have an untyped data pointer available for client use. Such a solution provides no static type checking, although some suitable run-time type checking scheme could be used. As a single pointer, it either supports a single client, or relies on clients to coordinate their access to the pointer, perhaps through the use of a single shared data structure such as a property list.

1.1.2 Property lists

Property lists are the next step up from simple client data pointers, since they permit multiple clients to store data (or *properties*) on a node. As with the single client data solution, the properties cannot be statically typed, although the property name can often be used as the basis of a single adhoc run-time typing mechanism.

1.2 Extending the functionality of nodes

The use of client data or a property list on a node only addresses the question of extending the set of data on (or associated with) a node. When executing an algorithm that involves processing each of the nodes of a data structure, it is typical to need to perform different operations according to the types of the individual nodes. Ideally, such operations can be virtual methods on the types of the nodes, but this requires that the necessary methods be known when the code for the nodes of the data structure is compiled, which is not always possible.

1.2.1 Run-time type determination

To be able to perform operations specific to the types of the nodes, and without being able to make such operations into virtual functions of the

nodes, requires that it at least be possible to determine the type of the nodes. This can be done by an ad-hoc mechanism such as a virtual function that returns a user type-code. Type-codes can be represented by any suitable unique value, such as a string literal or a value of an enumerated type. Alternatively, a more sophisticated mechanism to determine the type of a node, such as a run-time type check, can be used if such is available.

Whichever mechanism is used, the algorithm can be written such that it examines each node it encounters. For each node, the algorithm can attempt to determine the type of the node, and based on that information, a switch statement can be executed that performs the desired statements. In Modula3, a `typecase` statement is provided for this exact purpose, although the same effect can be achieved with a little effort in C or C++.

The mechanism obviously works, but is somewhat clumsy to use. One problem is that whenever the algorithm must dispatch according to the type of the node, it has to go through the run-time type determination and switch statement, which may not be cheap.

Furthermore, such code is clumsy to maintain and upgrade. When a new type is added to the type system of the data structure, there is no easy way to see which switch statements need to be modified. At the very least, it is worth having a coding convention that calls `abort` if the default case of the switch statement is executed. This provides a run-time check if code has been accidentally omitted, but it relies on having good test cases for the data structure, so that if code has been omitted, the abort actually occurs.

1.2.2 Using factories to hide the details of object creation

It is often the case that the data structure to be processed is not held in persistent storage as such, but is generated by reading data which is held in some external form. A classic example is the parse tree of a program generated by a compiler front end as the result of reading a text file. It is then typically the case that, although many different algorithms might be written to process the data, only one will be used for any particular instantiation of the data structure. Continuing the example of the compiler front end, many different code generators may be written to process the syntax tree, but only one is normally used in any one version of the compiler.

When this is the case, a possible solution involving factory objects can be used. When reading in the data, nodes are not created directly (e.g., by calling `malloc` or `new T`). Instead, the nodes are created indirectly by calling virtual methods on a factory object. A particular implementation of the factory can perform appropriate allocation of the nodes, such as allocating instances of subtypes of the required nodes.

Here is an example.

```
// generic code to create the data structure
Foo *read_data_structure(Factory *factory)
{
    ...
    Foo *foo = factory->create_foo(...);
    ...
    return foo;
};
class Factory
{
  public:
    Foo *create_foo(...) = 0;
    ...
};
// specialization of the code to allocate specific types of nodes
class MyFoo: public Foo
{
  public:
    ...
};
class MyFactory: public Factory
{
  public:
    Foo *create_foo(...) { return new MyFoo(...); }
    ...
};
int main(int argc, char **argv)
{
    MyFactory *mf = new MyFactory;
    Foo *f = read_data_structure(mf);
};
```

To read a data structure, `read_data_structure` is called with a specific factory, and so although a pointer to a `Foo` object is returned, the result is really a pointer to a `MyFoo`, since it will have been allocated by `MyFactory`. Thus, the resultant pointer can be narrowed to a `MyFoo`, in which case extra methods and data of the `MyFoo` object can be accessed. Unfortunately, all the internal pointers of the data structure will still be in the `Foo` space (not the `MyFoo` space), and so will need to be narrowed whenever it is necessary to access the extra functionality. Furthermore, this technique can only be used when the data structure is being created on demand from some other form, and when just one extension is needed.

1.3 Transforming the types of nodes

The previous two sections examined the problems when the nodes in a data structure were created without extra data members or methods that a specific client would have added, given an opportunity. There is one further case to consider: that the types of the nodes may not be the most convenient to work with.

For example, a single type of node may have been used where a client would have preferred different subtypes to be used, according to instance data of the node. The use of subtypes might even have subsumed the need for the instance data. To solve this problem, a client needs to be able to extend the functionality of the node in a data-dependent manner. This is typically done with `switch` statements or their equivalent, such as a sequence of `if` statements.

Alternatively, the data structure may use more types of nodes than a client is interested in. The client might only be interested in various high-level base types, and not in all the most derived types. In this case, the functionality of the nodes has to be extended according to the appropriate base types of the nodes, and not just according to the most derived type of the nodes. This is difficult without being able to add appropriate virtual methods to the various types of the nodes.

Shadows 2

In the previous section, various ways of extending the functionality of data structures were discussed. All had some features that make them appropriate in various situations, and all had drawbacks that make them inappropriate in others.

Shadows are based on a more powerful mechanism which make it possible to project a data structure from one type space into another. The primary reason to do this is to add extra functionality to the nodes of the data structure, such as extra methods. More sophisticated use of the mechanisms may change the actual shape of the shadow type structure itself.

One particular feature of shadows is that they are designed to work well even when inheritance is used for the nodes of the data structures. Support is also provided for lazily evaluating the projection of the data structure.

2.1 A simple example

Here is an example of a simple data structure and its shadow. On the left is a simple data structure, involving nodes of type `A`, `B`, and `C`. On the right is the shadow data structure, in which the nodes are of type `A′`, `B′`, and `C′`. The shadow may perhaps have been created in order to extend the types of the nodes in the data structure, and because it was not possible to change the data structure itself.

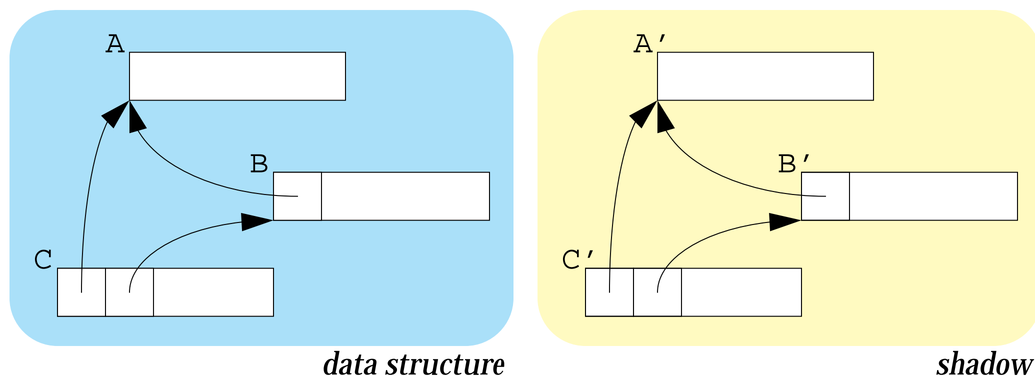


***Figure 1**: A simple data structure and its shadow*

In Figure 1, there is a 1-1 correspondence between the instances of the nodes in the data structure and its shadow, and there is a 1-1 correspondence between the types of the nodes in the data structure and the types of the nodes in its shadow. As we will see, neither of these conditions are required.

Note also the convention that is frequently used throughout this note: that `T'` stands for the type of the shadow of a node of type `T`.

2.2 The shadow map

The example in Figure 1 omits one important detail: how does the shadow get created? This is done by an object called a *shadow map*.

The shadow map provides a lazily evaluated mapping from the nodes in the original data structure to the corresponding nodes in the shadow data structure. Entries in the mapping are generated on demand by calling upon methods to fabricate nodes in the shadow data structure. These methods are called the *factory methods* of the map. Factory methods are supplied by a client according the the shadow that is to be created.

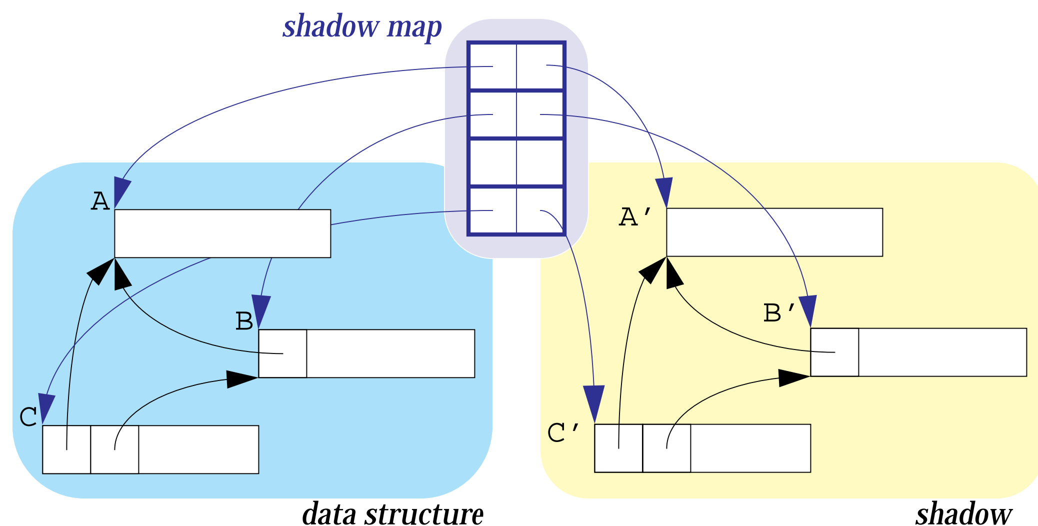


Figure 2: *A data structure and its shadow, showing the shadow map*

Thus, to get the shadow of a node, it is sufficient to ask the shadow map for the shadow, passing it the node in question. Internally, the shadow map sees if it already contains the shadow for the requested node. If it does, that

value is immediately returned; if not, it calls a factory method to create the shadow node. The result that is returned from the factory method is both remembered by the shadow map for future reference and also returned to the caller.

Once a shadow has been created, the shadow map often plays no further role and might even be deleted. Alternatively, it does contain pointers to all the nodes in the shadow data structure, and so it can prove helpful in performing operations involving the complete set of shadow nodes, such as deleting them when they are no longer needed.

2.3 Manipulating the data structure

In the examples so far, an unwritten implication of the figures has been that the nodes of the shadow data structure are very similar to the nodes in the original, merely having the types projected into the shadow type space. However, there is no reason for this to be so; the methods on the shadow map that generate shadow nodes can generate whatever form of shadow nodes they choose. The only restriction placed on these methods is that for each node in the original data structure, there should be a single corresponding node in the shadow data structure.[1]

For example, suppose that we are merely creating the shadow in order to add a method to each of the nodes, such as a `print` method. In this case, the shadow data structure might choose merely to store back pointers to the nodes in the original data structure, without shadowing the relationship between the nodes in the shadow space at all. The print method can get at the instance data[2] to be printed by following the back pointer to the nodes being shadowed; if needed, each shadow node can get at its shadow children by re-evaluating the shadows of the children in the original structure on demand.As well as specifying the instance data for the shadow nodes, the factory methods may also change the type of the nodes on a per-instance basis. Thus, two nodes in the original data structure may be represented by the same type, albeit with different instance data. In shadowing these two nodes, a factory method may examine the instance data and choose between two different representations for the shadowed node.

For example, consider the case where the data structure being shadowed is a parse tree generated by a compiler front end. A binary expression node in the parse tree might be represented by a single node type, `PT_BinaryExpr`, with an enumerated value indicating the operation of the expression: add subtract, multiply or divide. However, a back end may choose to shadow this one type of node by generating different types of

1. This is a slight over-simplification. A node in the original data structure can have no corresponding shadow node, in which case the shadow map will contain a null pointer in the appropriate entry.
2. There is an implicit assumption that any data to be shadowed is accessible, either directly or by functions.

shadow nodes depending on the operation involved, such as CG_AddExpr, CG_SubExpr, CG_MultExpr, or CG_DivExpr.

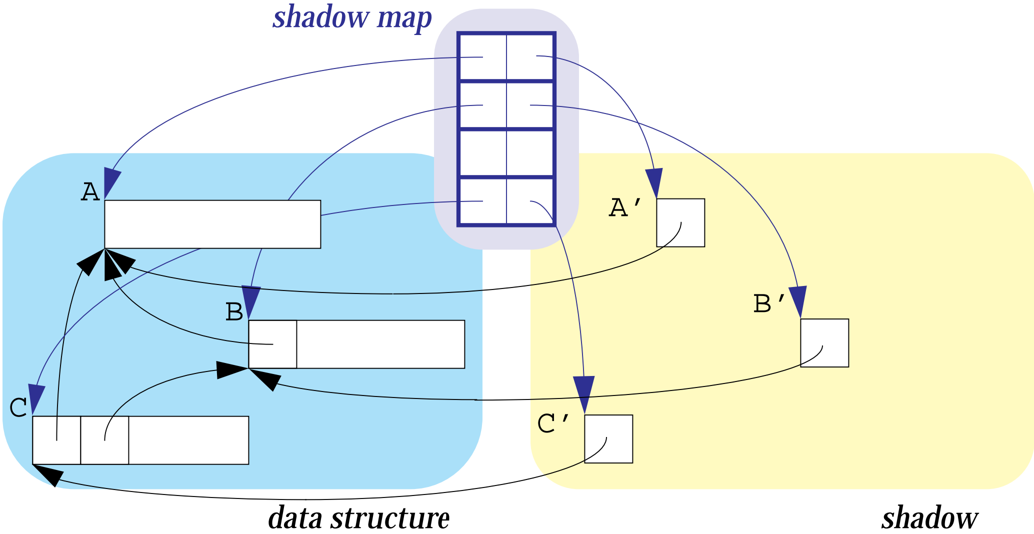


***Figure 3**: Using back pointers to reduce the size of the shadow data structure*

It may also be the case that not all of a data structure needs to be shadowed. Consider again the case of the parse tree generated by a compiler front end. An incremental code-generator may only need to generate code for (and hence shadow) the parse tree for one or two methods and not the whole tree.

2.4 Cyclic and acyclic data structures

In order to shadow a node, the algorithm described so far says that a client should simply ask the shadow map for the shadow, and that the shadow map will compute it if necessary by calling the appropriate factory method. However, if invoked, that method will often call upon the shadow map for the shadows of child nodes of the original, and this may in turn recursively invoke the factory method. All will be well—provided the data structure being shadowed is acyclic.

If the data structure contains cycles, slightly more care must be taken. Specifically, the (possibility of a) cycle must be known and the cycle explicitly broken.

In the preceding example, in the course of shadowing A, B needs to be shadowed, which needs C to be shadowed, which potentially needs A to be shadowed—and which may not have been completed yet, and so will not

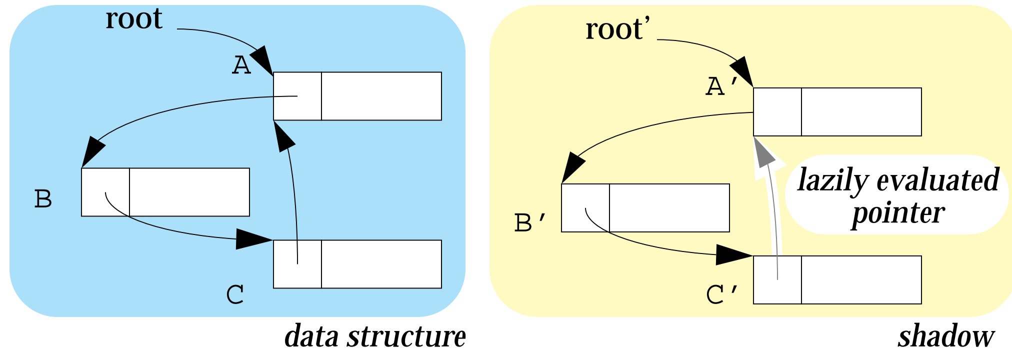


***Figure 4**: Shadowing cycles*

be available in the shadow map. Instead, `C'` stores a lazy pointer to `A'` which is only dereferenced to a real pointer when absolutely necessary. In particular, it can only be dereferenced after the shadowing of `A` has been completed, in which case the shadow map can provide the information that the result of shadowing `A` is `A'`. Once it has been converted to a real pointer, that value is remembered so that future accesses will be almost as fast as using the pointer itself.

In the C++ implementation, it serves to use a special type of pointer somewhere within the cycle, which is lazily evaluated. This is described in detail in Section 3.5, “Lazy shadows for cyclic data” on page 20.

C++ Implementation 3

Implementing a shadow map by calling upon a factory object and caching the results in a hash table may not seem very difficult. However, when inheritance is taken into account, the problem becomes a little more complicated. And, as we will see, the problem is also exacerbated by the rules and idiosyncracies of the C++ type system.

3.1 Inheritance

Consider two classes, `B` and `D`, such that `B` is a base class and `D` is derived from `B`. Now consider another class, `X`, which contains a pointer of declared type `B*`. When that pointer is initialized, it could be initialized with a pointer to an object of class `B`, or it could be initialized with a pointer to an object of class `D`, with the pointer being widened from type `D*` to type `B*`.

Now consider the shadow type space. Suppose `B′` is the shadow type for `B`, `D′` for `D` and `X′` for `X`. Then, in a simple shadow mapping, `X′` will contain a pointer of type `B′*`, and that pointer may point to objects of type `B′`, or `D′` if the pointer has been widened.

The problem, then, is to correctly shadow not only the dynamic (or *true*) type of the objects, but also the static (or *perceived*) type.

3.2 Shadowable, ShadowMap and T’::shadow.

A number of classes and methods collaborate to provide the shadowing mechanism. Figure 6 outlines the classes and primary methods involved.

Shadowable

Nodes to be shadowed must inherit from `Shadowable`. This provides an important virtual method that is used during the process of instantiating the shadow of a node.

```
class Shadowable
{
 public:
   virtual Narrowable *create_shadow(ShadowMap *map) = 0;
};
```

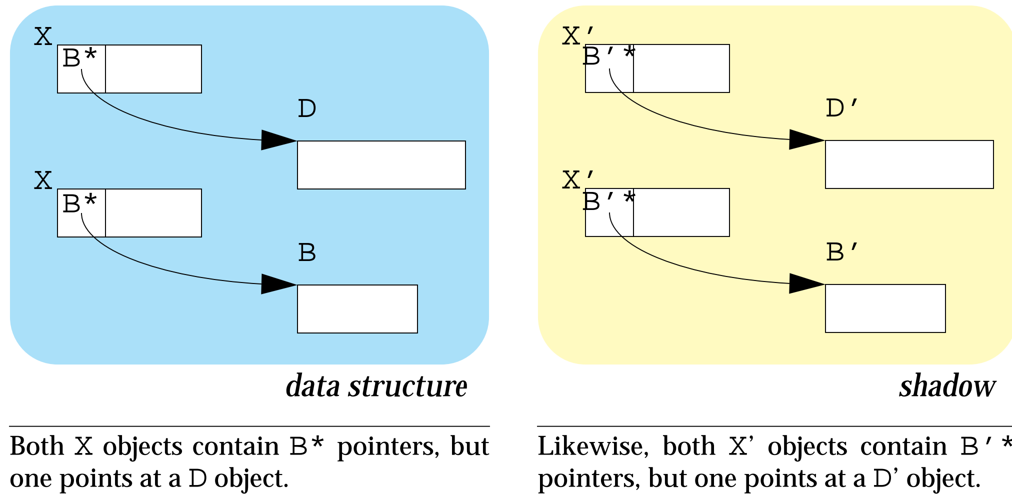


***Figure 5**: Shadowing in the face of inheritance*

Although all nodes to be shadowed must provide an implementation of `create_shadow`, this method is never directly called by a client. The implementation is boiler-plate code that invokes the appropriate factory method on an appropriate subtype of the shadow map argument. This boiler-plate code can be mechanically generated by `autodefine`. The `map` argument provides a context for the shadow, permitting different shadows of the same node to coexist.

ShadowMap

This represents the shadow map. It has one public method, to look up a shadow node in the map, and to create one if not found. However, even this method is not normally called directly by clients. Instead, clients should call the `shadow` method on the target class instead, which provides more convenient and type-safe access to shadow nodes.

There will normally be two levels of inheritance based on `ShadowMap`. The first extends the `ShadowMap` with a set of pure virtual methods that define the ability to shadow specific types of nodes; the second level provides the implementation of those pure virtual methods.

```
class ShadowMap
{
 public:
   Narrowable *lookup_or_create_shadow(Shadowable *orig);
           // returns the shadow node for 'orig'.
};
```

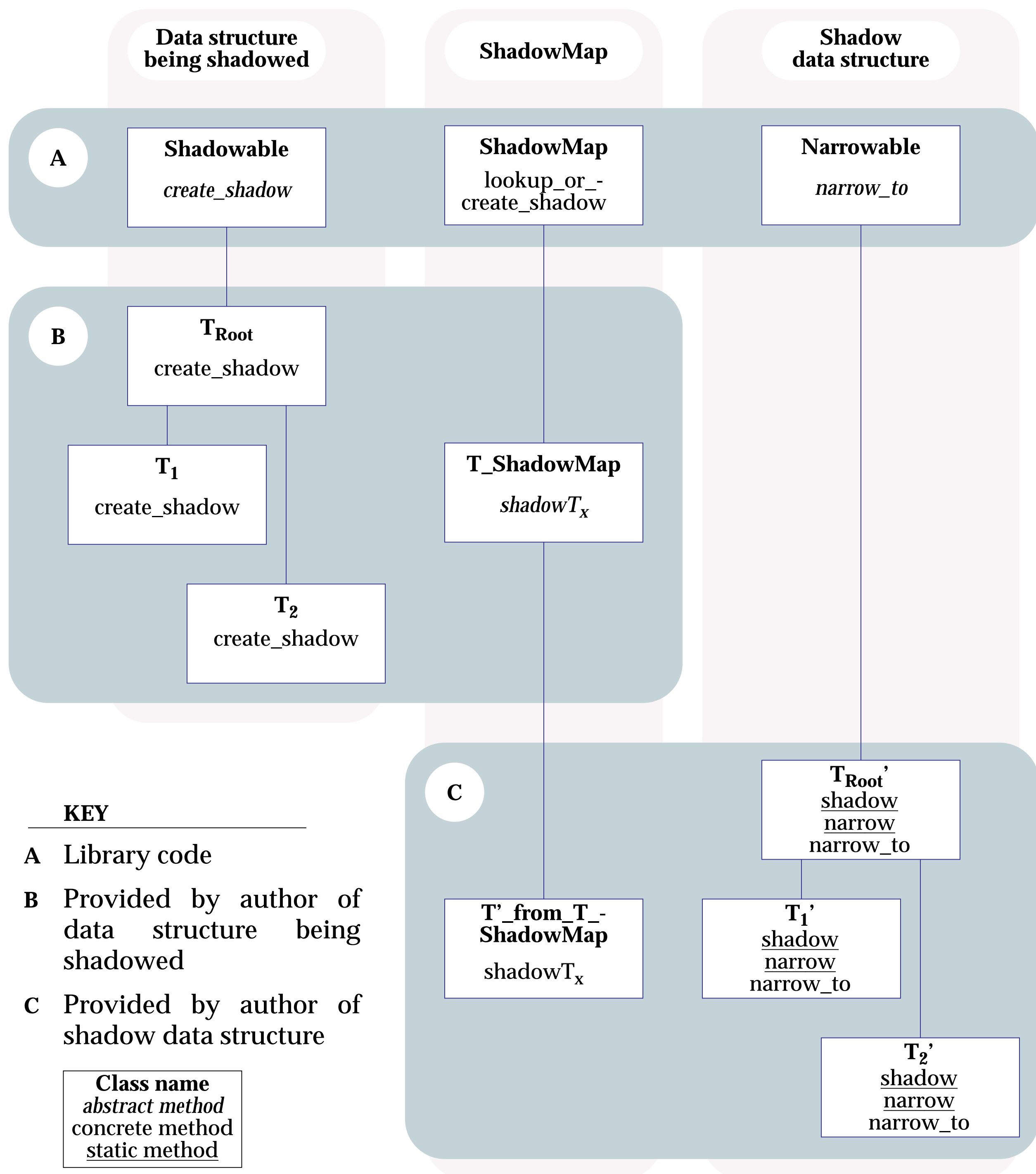


***Figure 6**: Primary classes and methods used to implement shadows*

T'::shadow

This is perhaps the most important method since it is the primary way a client creates or accesses the shadow of a node. It is a static method for the desired type, `T'`, of the shadow node. It takes two arguments: the node to be shadowed, of type `T`, and a shadow map.

```
static T' *T'::shadow(T *t, Roman_ShadowMap *map);
```

It returns the shadow of `t` according to `map`.

Ideally, this would be a method on a `Roman_ShadowMap`, as in

```
T' *t' = map->shadow(t);        // Wrong!
```

but of course there is no way to express the signature of such a method.

3.3 Making a data structure shadowable

This section describes the steps that must be taken to make a data structure shadowable. Sections 3.4 and 3.5 will describe the steps needed to implement a specific shadow, and finally, Section 3.6 will describe the steps to be taken by a client to actually instantiate a specific shadow.

To make a data structure shadowable, all the nodes to be shadowed in the data structure must inherit from a common root class, and this common root class must itself inherit from `Shadowable`.[3]

All nodes to be shadowed must implement the abstract `create_shadow` method inherited from `Shadowable`.

```
Narrowable *create_shadow(ShadowMap *);
```

These implementations of `create_shadow` are typically declared as private methods on the nodes to be shadowed, to emphasize that they are not to be called directly by clients. (The method is declared public on `Shadowable`.) The implementation is boiler-plate code that invokes the appropriate factory method on an appropriate subtype of the shadow map argument. This boiler-plate code can be mechanically generated by `autodefine.`

One more thing must be done to make a data structure shadowable: an abstract subtype of `ShadowMap` must be provided that defines a special method for each node type, *T*, to be shadowed:

```
Narrowable *shadowT(T *);
```

These methods will be referred to in the boiler-plate implementations of `create_shadow`.

3. Strictly speaking, the need for a common root class other than `Shadowable` is a requirement of the `autodefine` utility used to automatically generate parts of the shadow machinery.

3.3.1 *Example*

Suppose we have a data structure containing nodes of type `A`, `B`, `C` and `D`. Suppose `C` inherits from `A` and `D` inherits from both `A` and `B`. Let us suppose that `Roman` is the supertype for these four classes.

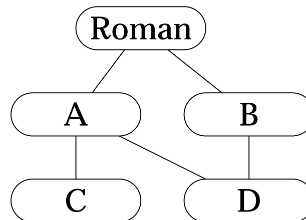


***Figure** 7: Class structure for example data structure*

Then we would expect to see the following patterns in the code:

```
class Roman: public virtual[4] Shadowable
{
 public:
   // any generic methods on all Roman node types can be declared here
 private:
   Narrowable *create_shadow(ShadowMap *);
};
class A: public virtual Roman
{
 public:
   // A methods ...
 private:
   Narrowable *create_shadow(ShadowMap *);
};
class B: public virtual Roman
{
 public:
   // B methods ...
 private:
   Narrowable *create_shadow(ShadowMap *);
};
```

4. Virtual inheritance is used in this example as a matter of style; it may not be necessary in all situations.

```
class C: public virtual B
{
 public:
   // C methods ...
 private:
   Narrowable *create_shadow(ShadowMap *);
};
class D: public virtual A, public virtual B
{
 public:
   // D methods ...
   Narrowable *create_shadow(ShadowMap *);
};
class Roman_ShadowMap: public virtual ShadowMap
{
 public:
   virtual Narrowable *shadowRoman(Roman *);
   virtual Narrowable *shadowA(A *);
   virtual Narrowable *shadowB(B *);
   virtual Narrowable *shadowC(C *);
   virtual Narrowable *shadowD(D *);
};
```

Given the name of the root class, `Roman`, `autodefine` can mechanically generate the definition of the class `Roman_ShadowMap` and default method bodies, and so we will skip the details of the default `shadow`*`T`* methods for now. In their simplest form, these methods can just be abstract methods.

3.4 *Making a specific shadow of a data structure*

For any data structure to be shadowed, it must be built with the conventions described in Section 3.3.

To permit a specific shadow to be made, the major task is to provide an implementation of the subtype of `ShadowMap` specific to the nodes in the data structure.

The shadow map subtype contains a collection of methods of the form:

```
Narrowable *shadowT(T *);
```

These methods will be implicitly invoked as a result of a client calling *`T'`*`::shadow(t, map);` for the first time. (Subsequent calls will return the same result, which will have been saved in the shadow map.) Thus, it is the task of the method to construct a *`T'`* object from the *`T*`* argument. This will typically involve shadowing the members of the *`T`* object and constructing a *`T'`* object from them.

In addition to providing an implementation of a shadow map with factory methods that actually create the nodes of the shadow data structure, a `shadow` method must be declared on each of the classes for nodes in the

shadow data structure. More accurately, it must be declared on those classes which can be generated as the result of shadowing a shadowable node. (See ***T'::shadow*** in Section 3.2.)

If the class being shadowed is `T`, and the corresponding shadow class is `T'`, then the shadow method is declared as:

```
class T'
{
 public:
   static T' *shadow(T *t, TRoot_ShadowMap *map);
};
```

The implementation of this method is boiler-plate, and can be generated by `autodefine`. It calls `map->lookup_or_create_shadow(t)` and narrows the result to be of type `T'*`.

3.4.1 Example: Shadow methods and ways of generating shadow nodes

Continuing the example of Section 3.3.1, involving classes `A`, `B`, `C`, and `D`, that inherit from `Roman`, suppose we wish to make a shadow involving nodes `Alpha`, `Beta`, `Gamma`, and `Delta`, that have a common root node, `Greek`. Then, picking `Alpha` as an example, its `shadow` method might be defined as:

```
class Alpha
{
 public:
   // other Alpha methods ...

   static Alpha *shadow(A *a, Roman_ShadowMap *map);
};
```

Now we need to define a factory class, `GreekFromRoman_ShadowMap` that is an implementation of the class `Roman_ShadowMap`. Note that that class merely declared abstract (or default) `shadowT` methods for the various subtypes of the root node, `Roman`. It did not embody any knowledge of the types of the shadow nodes to be created; that knowledge will be embodied in the implementation of the methods of `GreekFromRoman_ShadowMap`.

Here is what the definition of the class `GreekFromRoman_ShadowMap` might look like.

```
class GreekFromRoman_ShadowMap:
   public virtual Roman_ShadowMap
{
 public:
   // Roman_ShadowMap methods defined by this class:
   Narrowable *shadowA(A *);
   Narrowable *shadowB(B *);
   Narrowable *shadowC(C *);
   Narrowable *shadowD(D *);
};
```

If the shadow map being inherited has only pure virtual methods, then all of those methods will need to be implemented here. If there are any default implementations, then it may not be necessary to implement all of the methods here. Default factory methods for a shadow map are described in Section 5.1 on page 27.

There are several paradigms that can be used, and the examples that follow should be considered merely illustrative of the techniques that can be used. Which paradigm is appropriate in any particular situation will be a matter of personal taste and considerations of how to group the shadowing code: all together, or with the code for the nodes to be shadowed.

Shadowing instance data within the factory method for a shadow node

Suppose we wish to shadow an `A` object into an `Alpha` object, and that an `A` object has two members, of type `B*` and `C*` that are both to be shadowed into the `Alpha` object. Then, assuming suitable access functions and constructors, the factory method could be written in the following way:

```
Narrowable *GreekFromRoman_ShadowMap::shadowA(A *a)
{
   Beta *beta = Beta::shadow(a->get_b(), this);
   Gamma *gamma = Gamma::shadow(a->get_c(), this);
   return new Alpha(beta, gamma);
}
```

The method shadows the child members of `A`, recursively invoking the shadow machinery for the children and calling a constructor on the result. Not all children need be shadowed if there is no need for them in the shadow node. Some members may not be shadowable: for example, basic types, enumerated values, strings, and so on. These members can either be copied across to the shadow node directly, or a back pointer to the original node can be stored in the shadow, so that the shadow can get at these members by following the back pointer. The decision of which way to store them should be made by taking considerations of space and access time into account.

Just as not all members of the original node may be shadowed, so not all members of the shadow node may be the result of shadowing members from the original node. Storing a back pointer is a very simple example. Alternatively, some members of the shadow node may be the result of computing and caching a complex expression involving one or more members of the original node.

Shadowing instance data within a constructor for a shadow node

Instead of shadowing the children in the factory method, the shadowing can be done in the constructor for the shadow node itself:

```
Narrowable *GreekFromRoman_ShadowMap::shadowA(A *a)
{
   return new Alpha(a, this);
}
Alpha::Alpha(A *a, Roman_ShadowMap *map)
   :  beta(Beta::shadow(a->get_b(), map)),
      gamma(Gamma::shadow(a->get_c(), map())
{ }
```

Performing the shadowing in the constructor keeps the conversion code close to the rest of the code for `Alpha`, whereas performing the shadowing in the factory groups all of the shadowing code (for all classes) together.

Shadowing instance data within a static method of the shadow node

Having the factory method call the constructor for the shadow node implies that the factory knows exactly what the shadow type really will be, as indicated by the calls of `new Alpha(...)`. Another solution is to use a static `create` method on the `Alpha` class. This keeps the factory code simple, without binding in knowledge of a concrete type.

```
Narrowable *GreekFromRoman_ShadowMap::shadowA(A *a)
{
   return Alpha::create(a, this);
}
Alpha *Alpha::create(A *a, Roman_ShadowMap *map)
{
   Beta *beta = Beta::shadow(a->get_b(), map);
   Gamma *gamma = Gamma::shadow(a->get_c(), map);
   return new Alpha(beta, gamma);
}
```

3.4.2 Example: Generating different types of shadow node

In Section 2.3 on page 8, there was an example concerning a compiler parse tree node of type *`PT_BinaryExpr`* that is to be shadowed into one of a number of different types, depending on the binary operation.

Here is how that might be implemented, using a static create method in the shadow class.

```
CG_Expr *CG_BinaryExpr::create(PT_BinaryExpr *be,
                               PT_ShadowMap *map)
{
   CG_Expr *left = CGExpr::shadow(be->get_left(), map);
   CG_Expr *right = CGExpr::shadow(be->get_right(), map);
   switch (be->get_op()) {
      case Op::Add:
         return new CG_AddExpr(left, right);
      case Op::Sub:
         return new CG_SubExpr(left, right);
      case Op::Mult:
         return new CG_MultExpr(left, right);
      case Op::Div:
         return new CG_DivExpr(left, right);
      default:
         should_not_happen();
   }
}
```

Note that the body of this method could also be put into the factory method `shadowPT_BinaryExpr`. The choice of where to locate the code is mostly one of style, although by making it a method on `PT_BinaryExpr`, it is possible that a more sophisticated example would be able to make use of other methods defined in the class which might only be privately accessible.

3.5 Lazy shadows for cyclic data

The various examples of making shadows in Section *3.4* also used the primitive operation `T'::shadow`. As was noted in the explanation, this will likely cause recursive calls on shadow for all the children of the node being shadowed. The corollary is that when shadowing a node, the transitive closure of all children of that node will also be shadowed. This will work provided the data structure is acyclic.

Lazy shadows provide a way around these problems. They should be used if the data has (or might have) cycles; they can also be used if it is not necessary or desirable to shadow all of a data structure up front.

If `T` is a type of node to be shadowed and `T'` is the type of the shadow node for `T`, then `LazyShadow<T', T>` is a template for a smart pointer that lazily evaluates `T'::shadow`. The smart pointer is initialized with the two arguments for `T'::shadow`: a `T*` pointer and a `ShadowMap*`. It can be assigned to a `T'*` pointer, and can be transparently dereferenced as a `T'` pointer; both of these operations will cause the shadow to be evaluated. Consider the following code, written without using lazy shadows.

```
Narrowable *GreekFromRoman_ShadowMap::shadowA(A *a)
{
   return new Alpha(a, this);
}

class Alpha
{
  public:
   Alpha::Alpha(A *a, Roman_ShadowMap *map);
   void print();
  private:
   Beta *beta;
   Gamma *gamma;
};

Alpha::Alpha(A *a, Roman_ShadowMap *map)
  : beta(Beta::shadow(a->get_b(), map)),
   gamma(Gamma::shadow(a->get_c(), map())
{ }

void Alpha::print()
{
   Beta *beta2 = beta;
   gamma->print();
};
```

If for some reason it becomes necessary or desirable to use lazy shadows for the child nodes of class `Alpha`, here is how the code might change.

```
class Alpha
{
  public:
   Alpha::Alpha(A *a, Roman_ShadowMap *map);
   void print();
  private:
   LazyShadow<Beta,B> beta;       // was Beta *beta;
   LazyShadow<Gamma,C> gamma;     // was Gamma *gamma;
};

Alpha::Alpha(A *a, Roman_ShadowMap *map)
  : beta(a->get_b(), map),        // shadow call removed
   gamma(a->get_c(), map()        // shadow call removed
{ }
```

The method `GreekFromRoman_ShadowMap::shadowA` remains unchanged. The implementation of `Alpha::print` also remains unchanged, because `beta` can be transparently converted to and used as a `Beta*` pointer, just as `gamma` can be converted to a `Gamma*` one. The first time `beta` is converted to a `Beta*` pointer, the call of `Beta::shadow(a->get_b(), map)` will finally occur; however, subsequent uses of `beta` will bypass the call and will be almost as fast as using the evaluated pointer.

3.6 *Instantiating a shadow of a data structure (Or: What the end client actually needs to know!)*

We have already seen most of the apparatus needed to instantiate a shadow of a data structure. To shadow a data structure, a client needs the following:

- a root pointer to the data structure to be shadowed, or to a part of the data structure to be shadowed
- the shadow map subtype for the appropriate shadow type space

The following is a typical code sequence used to instantiate a shadow:

```
A *a = ...
    // get pointer to data structure to be shadowed

Roman_ShadowMap *map = new GreekFromRoman_ShadowMap;
Alpha *alpha = Alpha::shadow(a, map);
    // that's all there is to it
```

This works as follows: after getting a pointer to the node to be shadowed, the code creates an instance of the appropriate subtype of `ShadowMap`, and then simply shadows the root pointer, using exactly the same method we have already seen when shadowing the child members of a node being shadowed. The result is that `alpha` is a root pointer to a shadow data structure as defined by the `GreekFromRoman_ShadowMap` object.

Once the node has been shadowed, it depends on the specification of `GreekFromRoman_ShadowMap` as to when the map can be deleted. If the `alpha` shadow node has been fully evaluated, containing no internal lazy shadows and with no other implicit dependency on the map, then the map can indeed be deleted. However, a client may choose to keep the map available to shadow other nodes, in which case some or all of the saved entries in the map may speed up subsequent shadow operations.

Behind the Scenes 4

Section 2 gave an overview of the entire mechanism, and Section 3 gave details of what is needed both to make a data structure shadowable and to actually make a shadow. This section describes the glue code needed to make the various elements of code come together.

Let us follow the execution of the two lines of example code from the previous page:

```
Roman_ShadowMap *map = new GreekFromRoman_ShadowMap;
Alpha *alpha = Alpha::shadow(a, map);
    // that's all there is to it
```

4.1 Creating a shadow map

The call of new `GreekFromRoman_ShadowMap` constructs an instance of an implementation of a subtype of `ShadowMap` that can be used to generate a "greek" shadow of "roman" nodes. The call will automatically initialize a table in the `ShadowMap` base class that will be used to cache the results of calling the factory methods provided by the subtype.

4.2 Creating a shadow node

The shadow node is created by invoking the `shadow` method on the required type. This is a class-static method whose implementation is of the form:

```
T' *T'::shadow(T *x, TRoot_ShadowMap *m)
{
    return T'::narrow(m->lookup_or_create_shadow(x));
}
```

The map is interrogated for the shadow of `x` by the call of `m->lookup_or_create_shadow(x)`.

The result of `lookup_or_create_shadow` is of type `Narrowable*`, since the map cannot know the type that will actually be required. The result is therefore narrowed to the correct type by calling `T'::narrow`. Assuming the shadow has been created correctly, this will give the required result. Otherwise, zero will be returned.

Ignoring some minor complexity to deal with detecting cycles while creating shadows, the call of `lookup_or_create_shadow` checks to see if a shadow node is already registered for the argument, and if not, it takes steps to create the shadow node. Somewhat simplified, the code looks like the following:

```
Narrowable *ShadowMap::lookup_or_create_shadow(Shadowable *sp)
{
    Narrowable *np;

    if (sp == 0) return 0;

    if (!tbl->get_shadow(sp, np)) {
        np = sp->create_shadow(this);
        tbl->set_shadow(sp, np);
    }

    return np;
}
```

Notes:

`tbl`
: This represents the table that contains the results of earlier calls to `lookup_or_create_shadow`.

`tbl->get_shadow(sp, np)`
: This looks up the shadow of `sp`, sets `np` and returns `TRUE` if it is found, and otherwise leaves `np` alone and returns `FALSE`.

`tbl->set_shadow(sp, np)`
: This records `np` as the shadow of `sp`.

The major line that needs explaining is this one:

```
    np = sp->create_shadow(this);
```

This is the means by which the shadow mechanism correctly creates the appropriate type of shadow node, independently of the perceived type of the object. The client may have a `B*` pointer that in reality points to a `D` object. Nevertheless, the client may well invoke `Beta::shadow(b, map)` and will expect to receive a `Beta` object even though the real type of the object being shadowed is a `D` and the real type of the shadow node must be a `Delta`.

`create_shadow` is a virtual method of the node being shadowed. It is implemented for each type of object being shadowed, and looks like this:

```
Narrowable *D::create_shadow(ShadowMap *map)
{
    Roman_ShadowMap *m = Roman_ShadowMap::narrow(map);
    return m->shadowD(this, map);
}
```

Note that the argument is only of type `ShadowMap*` and not of type `Roman_ShadowMap*`. This is a regrettable consequence of `create_shadow` being inherited from `Shadowable`, which only knows

about ShadowMap.

When sp->create_shadow(this) is invoked, the message will be received by the true type of the object, in its implementation of create_shadow. This knowledge of the true type is passed on to the map by invoking a method specific to the true type. This is the purpose of the call:

```
m->shadowD(this)
```

Since this is a call to shadow a Roman node (D), this is a method on the Roman_ShadowMap, which is the reason for the narrow call in the previous line of code. Note that map is not narrowed to the actual implementation type of the shadow map, since that is neither known nor necessary; it is merely narrowed to the type that defines the abstract methods to shadow the various types.

Typically, the call of m->shadowD(this) will be implemented by a subtype of Roman_ShadowMap that provides the knowledge of how to shadow a D node into the appropriate shadow type. In our running example, the call would be handled by GreekFromRoman_ShadowMap::shadowD, of which we have already seen the like. (See the discussion on the implementation of shadowA in Section 3.4.)

The result of the shadow call on the map is of type Narrowable*. The result is passed back to ShadowMap::lookup_or_create_shadow, which saves the value for future use, and returns it, still as a Narrowable*, to the original call of shadow on the target type. This attempts to narrow to the appropriate type the value that has been passed back. If the narrow call succeeds, the client has the value that was wanted; if not, then zero is returned, which should be interpreted as meaning that either the item being shadowed was itself zero, or that the item cannot be shadowed to the requested type. If it is important to distinguish these cases, the client can easily check if the item being shadowed is zero or not.[5]

It is worth checking what happens in the face of inheritance and widened pointers. Assume the client has in his hand a B* pointer to a D object. As a B* pointer, it is to be expected that the shadow will be a Beta* pointer. Therefore, Beta::shadow(b, map) will be called. Assuming this is the first shadow call on b, ShadowMap::lookup_or_create_shadow will invoke create_shadow on b, which will be handled by the true type D, by way of the virtual function dispatch. This will call the factory to shadowD. Thus, the shadow node will be created with the correct shadow type. The pointer to the shadow node (e.g., a Delta*) will be passed back as a Narrowable*, and narrowed to Beta* in the code for Beta::shadow.

5. This interface might change when exceptions are available in C++.

Thus, the mapping between node type and shadow node type is maintained for both the true and perceived type of the objects involved.

4.3 Who does what: a quick reference guide

There are many different parts to the shadow system, with the code for the various parts being the responsibility of various different authors. There are at least three authors involved: the author of the standard shadow code, the author of the data structure being shadowed, and the author of the shadow data structure itself.

In addition, some of the code can be mechanically generated by a special utility called `autodefine` which is described in Appendix B.

The following table describes the various components of the shadow system and identifies the responsibility for the various parts. See also *Figure 1* on page 6.

Implementation responsibilities

Class or methods	*Responsibility*
`LazyShadow`	Standard shadow library code
`Shadowable`	Standard shadow library code
`ShadowMap`	Standard shadow library code
`T``::create_shadow` (declaration) (definition)	Data structure being shadowed Autodefine
`TRoot``_ShadowMap` (class definition and default methods)	Autodefine
`T'_from _T_ShadowMap` (declaration and methods)	Shadow data structure
`T'``::shadow` (declaration) (definition)	Shadow data structure Autodefine
`T'``::narrow` (declaration) (definition)	Shadow data structure Autodefine
`T'``::narrow_to` (declaration) (definition)	Shadow data structure Autodefine

Explanation of symbols

`T`	Prototype class in shadowed data structure
`TRoot`	Common root class for all classes in shadowed data structure
`T'`	Prototype class in shadow data structure
`T'_Factory`	Name of factory for shadow data structure

Advanced Usage 5

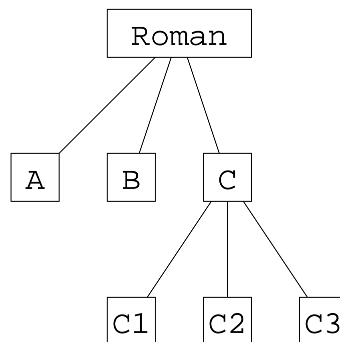
5.1 Default factory methods on a shadow map

Section 3.3 described the actions necessary to make a data structure shadowable, and it introduced the need for a subtype of `ShadowMap` that declared a `shadowT` method for each class *T* in the data structure that needed to be shadowed.

Other than saying that these methods could be pure virtual methods, and other than saying that the class could be automatically generated by `autodefine`, Section 3.3 did not go into any further details of the implementation of the methods. This section shows how default methods on the `ShadowMap` subtype may be used to advantage.

The examples so far have suggested that each type in the data structure has one or more corresponding types in the shadow data structure: we have seen examples of both *1-1* and *1-many* mappings. However, *many-1* mappings (and even *many-many* mappings) are also possible.

Consider the following inheritance hierarchy:



For a particular shadow of a data structure built from these data types, it may be the case that we do not need to shadow all of the information in types `C1`, `C2` and `C3`: it might be sufficient just to shadow the properties they have as `C` objects. In other words, rather than shadowing `C1` to `Gamma1`, `C2` to `Gamma2`, and `C3` to `Gamma3`, we wish to shadow all `C` nodes to `Gamma` nodes.

Since we wish to shadow nodes of these types to Gamma nodes, it is appropriate to call Gamma::shadow(C*, Roman_ShadowMap*). Assuming this is the first shadow call on the C* argument, this will call through the virtual create_shadow method, and that call will be received by the true type of the object: C1, C2 or C3. Whichever type receives the call will pass it on to the appropriate method on the factory: shadowC1, shadowC2, or shadowC3.

If the methods on the Roman_ShadowMap are all pure virtual methods, then they must all be separately implemented by the actual map used for the specific shadow. However, if the methods on Roman_ShadowMap have a default implementation, then they do not need to be overridden in the actual map used. In this case, what is wanted is for shadowC1, shadowC2, and shadowC3 all to delegate to shadowC. This means that in the actual map to be used, only shadowC need be provided, and this can do what is necessary to shadow any C objects, whether they be C, C1, C2, or C3 objects.

Based on this rationale, here is the rule implemented by autodefine when generating the *TRoot*_ShadowMap for a collection of types inheriting from *TRoot*.[6] The default implementation for each shadow*T* method is to delegate to the immediate parent type, in the case of single inheritance, or to the leftmost immediate parent, in the case of multiple inheritance.[7] The one exception is the shadow*TRoot* method, to which all methods will eventually delegate if not overridden, and this method will abort if invoked. It is up to the client to ensure that this abort does not get called: either by overriding it, or by ensuring that no instances of *TRoot* are allocated and that all subtypes of *TRoot* have overridden appropriate shadow*T* methods in the map.

5.2 Shadows of shadows

If some data structure is shadowed to yield a new data structure, there is no reason why that new data structure should not itself be shadowable. All that is required is that the base type of the nodes in the new data structure should inherit from Shadowable, and that the nodes should implement create_shadow.

A sequence of shadows could be used (for example) to represent the transformations involved in a compilation pipeline, going from syntax tree to machine code instructions. In this case, the job of fabricating the shadow nodes at each stage would probably be non-trivial, but there is no inherent reason why this should not be done.

6. autodefine also has an option to make the methods on the *TRoot*_ShadowMap class be pure virtual.
7. *Caveat*: autodefine does not notice inherited template types.

A variant of this is that a shadow might provide methods that, when invoked, cause a different data structure to be created, and this resultant data structure might itself be shadowable.

5.3 Passing extra parameters to the shadow call

When shadowing a node, it is sometimes useful to be able to pass additional information to the call that will fabricate the shadow node. This may be information needed globally by the whole shadow (such as a window handle or error stream), or it may be needed by a specific type and a specific shadow call.

If this is necessary, such information can be stored as instance data of the actual implementation of the `ShadowMap`. This means that the information will be available in any call of `shadowT` that may subsequently be invoked.

Information that is global to the shadowing process can be stored as instance data in the map when the map is constructed.

Information that is needed on a per-node basis requires a little more care. The easy solution is to provide access methods on the factory that directly allow the extra instance data to be accessed and modified throughout the shadowing process. However, if this technique is used, there is a serious potential for an undetected programmer error. Consider the following scenario:

> Inside some call of `Tx'::shadow(Tx *x, TRoot_ShadowMap *map)`
> – set instance data `d` on the map: `map->set_data(d)`
> – call `Ty'::shadow(y, map)` to shadow some child `y`

The problem is that there is an inherent guarantee that the final call of T_y`'::shadow(y, map)` will see the data `d`. If by some programmer error the shadow call has already been executed for the value `y`, then an entry will exist in the map and subsequent calls will return that value, and will not go through the factory method and pick up the data. In addition, it is clumsy for the client to have to call two methods: one to set the data, and another to shadow the desired node. What is required is a slightly different mechanism that is easier to use and which provides better semantic guarantees.

The solution is to provide methods on the map that set the instance data and shadow the node in a single operation. Since the reason for these methods is to have extra shadow-specific parameters, these are methods defined directly on the implementation of the shadow map, and are not inherited at all.

For example, the lines above could be replaced by a method defined as follows:

```
class TRoot'_from_TRoot_ShadowMap
{
 public:
   Ty' *create_Ty_shadow(Ty *y, Data *d);

 // rest of class ...

private:
 Data *data_for_create_Ty_shadow;
};
```

The implementation of

```
Ty' *create_Ty_shadow(Ty *y, Data *d)
```

can set the instance variable `data_for_create_Ty_shadow` and must then cause a shadow to be created. Whilst it can do that by explicitly calling

```
Ty'::shadow(y, map)
```

there is another method on `ShadowMap` that is more suitable. Internally, `Ty'::shadow(y, map)` calls

```
Ty'::narrow(map->lookup_or_create_shadow(y))
```

and as we have seen, this returns the value stored in the map if it already exists. The alternative is to replace the call of `Ty'::shadow(y, map)` by the following

```
y' = Ty'::narrow(map->create_shadow(y));
```

The call of `map->lookup_or_create_shadow(y)` has been replaced by a call to the sibling method `map->create_shadow(y)`, which still looks up `y` in the map, but this time to check that no value is already stored in there. It is a run-time error if a value is found. Assuming no entry is found, the code will then internally call

```
y->create_shadow(map)
```

which, as before, will call back into one of the `shadowT` methods on the map. These all have access to the data `d` and can fabricate the shadow node correctly, based on the type of the node and the contextual information passed in from the client.

The name `create_Ty_shadow` is the choice of the author of the shadow map implementation class, in this case `TRoot'_from_TRoot_ShadowMap`. It is specific to the implementation class and not inherited at all. It is suggested that it not be an overloaded name in case the implementation class is ever extended by implementation inheritance to shadow even more types.

Acknowledgements 6

This note describes work developed by Michael J. Day and Jonathan J. Gibbons, with invaluable feedback from Theodore C. Goldstein, Brian T. Lewis, and Michael J. Jordan. Jos Marlowe also helped with her reviews of the many earlier drafts of this document.

Appendix A: Narrow

This appendix describes an implementation of a design first put forward in *Narrow in C++* by Dwight Hare of SunSoft.

A.1 Narrowing

If class `DERIVED` inherits from class `BASE`, then a pointer to an object of class `DERIVED` may be used where a pointer to an object of class `BASE` is expected. Narrowing is the inverse operation of taking a pointer that appears to be a pointer to an object of class `BASE`, and determining whether the pointer really is to an object of class `BASE`, or whether it is a pointer to an object of class `DERIVED`, or indeed, of some other class that inherits from `BASE`.

A.2 The "narrow" method

Any class, *T*, that wishes to make this mechanism available to clients must provide a special method for clients to use:

```
static T *narrow(Narrowable *);
```

This method takes its argument and attempts to narrow it to be of type *T*. If the pointer argument does not point to an object of type *T* or to a type derived from type *T*, then a zero result is given. A zero argument also yields a zero result.

A.3 Implementation details

A.3.1 Narrowable

In addition to declaring `narrow`, each class *T* must inherit (directly or indirectly) from `Narrowable`. This declares a pure virtual method

```
void *narrow_to(TypeId *)
```

that must be implemented by each concrete class in the type-hierarchy. Although declared to be public on `Narrowable`, this method is internal to the mechanism and is typically declared to be private in the leaf subclasses.

A.3.2 TypeId typeid_T;

For any class *T* that declares `T::narrow`, the class must declare a corresponding global constant of type `TypeId`:

```
TypeId typeid_T;
```

The important thing here is the uniqueness of the address of `typeid_T` that is of significance, not its value. Thus, it is sufficient to use

```
typedef int TypeId;
```

and to let the linker allocate all `TypeId`s at unique addresses.

A.3.3 Implementation of "narrow"

Given a `typeid_T`, the definition of `narrow` is as follows:

```
T *T::narrow(Narrowable *x)
{
   return (x == 0 ? 0 : (T *)x->narrow_to(&typeid_T));
};
```

A.3.4 Implementation of "narrow_to"

`narrow_to` must be defined on all non-abstract classes that inherit from `Narrowable`. It is invoked from `T::narrow` for some class `T`, and will be passed `&typeid_T` as an argument. `narrow_to` checks the argument against the `typeid`s for all types that objects of this type might have been widened to, and if a match is found, it returns the value of the `this` pointer adjusted to be of type `T*`. Note that although the result is passed back as a `void *`, `narrow_to` expects that the caller will immediately cast it to the correct type.

For example, if `A` inherits from `B` and `C` and they both inherit from `D`, then the following would be appropriate:

```
void *A::narrow_to(TypeId *t)
{
   if (t == &typeid_D) return (D*)this;
   else if (t == &typeid_C) return (C*)this;
   else if (t == &typeid_B) return (B*)this;
   else if (t == &typeid_A) return (A*)this;
   else return 0;
};
```

It can be noted that individual lines may be omitted if the corresponding class does not support a `narrow` operation (for whatever reason).

A.4 Autodefine

The definition of `T::narrow`, `T::narrow_to` and `TypeId typeid_T;` can be automated by the use of the `autodefine` utility. (See Appendix B.)

Appendix B: Autodefine

`autodefine` is a utility that scans C++ header files to look for any declarations of a set of standard methods. For any such declarations that are found, it generates the corresponding standard definition.

Among the declarations it checks for are those needed by the shadow and narrow mechanisms.

Parsing C++ is a complex and difficult task. Instead of having a full C++ parser, autodefine uses a simple LR(1) parser written in YACC to implement "fuzzy parsing": it recognizes enough of the syntax of C++ to do its job, and skips over the rest. From that which it recognizes, it is a relatively easy matter to build up a simple syntax tree and to deduce and write out the generated definitions.

For example, it detects sequences like (but not limited to) the following:

```
class name [ : ( public virtual name )+ ]
{
 public :
   ...
   static name * narrow ( Narrowable * [name] ) ;
   ...
   static name * shadow ( Shadowable * [name] ,
                                   ShadowMap * [name] ) ;
   ...
} ;
```

It is a relatively simple matter to skip over the vagaries of whitespace, alternate orderings of method declarations or keywords like `public` and `virtual`, optional argument names, and even complete method bodies including balanced pairs of { and }.

One major simplification is the assumption that the input text is well-formed C++: since the input is C++ header files that will (presumably) be read by the C++ compiler as well, this is not an unreasonable assumption.

The current implementation has a restriction that only one family of types per directory is shadowable. To work around this restriction, an arbitrary class must be introduced from which all families inherit, thus reducing the many families of types in this directory to one extended family.

© Copyright 1994 Sun Microsystems, Inc. The SMLI Technical Report Series is published by Sun Microsystems Laboratories, a division of Sun Microsystems, Inc. Printed in U.S.A.

Unlimited copying without fee is permitted provided that the copies are not made nor distributed for direct commercial advantage, and credit to the source is given. Otherwise, no part of this work covered by copyright hereon may be reproduced in any form or by any means graphic, electronic, or mechanical, including photocopying, recording, taping, or storage in an information retrieval system, without the prior written permission of the copyright owner.

TRADEMARKS
Sun, Sun Microsystems, and the Sun logo are trademarks or registered trademarks of Sun Microsystems, Inc. UNIX and OPEN LOOK are registered trademarks of UNIX System Laboratories, Inc. All SPARC trademarks, including the SCD Compliant Logo, are trademarks or registered trademarks of SPARC International, Inc. SPARCstation, SPARCserver, SPARCengine, SPARCworks, and SPARCompiler are licensed exclusively to Sun Microsystems, Inc. All other product names mentioned herein are the trademarks of their respective owners.