The following paper was originally published in the
Proceedings of the Third USENIX Conference on Object-Oriented Technologies and Systems
Portland, Oregon, June 1997

Montana Smart Pointers: They're Smart, and They're Pointers

Jennifer Hamilton
Microsoft

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org

Montana Smart Pointers: They're Smart, and They're Pointers

Jennifer Hamilton*
Microsoft
jenh@microsoft.com

Abstract: The Montana C++ programming environment provides an API interface to the compiler, which allows the compilation process to be extended through programmer-supplied tools. This paper investigates the feasibility of that interface, using smart pointers as an example. Smart pointers are a powerful feature of the C++ language that enable a variety of applications, such as garbage collection, persistence, and distributed objects. However, while smart pointers can be used in much the same way as built-in pointers, they are not interchangeable. Using the Montana API, smart pointer functionality can be introduced for built-in pointers, thus enabling built-in pointers that act like smart pointers. We provide an overview of the Montana programming environment and describes how smart pointers can be implemented using the Montana API.

1. Introduction

The Montana[1] C++ programming environment is a joint development effort between IBM's Software Solutions and Research Divisions, and will be the base for a future release of IBM's VisualAge C++ product. Montana provides many unique features over traditional C++ compilers, most notably support for complete incremental compilation and an API interface [Nac96].

The purpose of this paper is to assess the feasibility of the Montana API interface for extending the compilation process to augment built-in language syntax. We have chosen the C++ smart pointer support as a basis of comparison. In this paper we present a partial smart pointer implementation using a Montana extension, where built-in pointer operations are modified as part of the compilation process, and summarize the results.

2. The Montana C++ Programming Environment

The Montana project grew from the recognition that current C++ development environments, while improving, were lacking in many areas, especially compared with those available for languages such as Smalltalk. One of the major frustrations in developing large C++ applications is the build turnaround time. The goal for Montana is to provide extremely fast incremental compilation, so that recompilation time required is proportional to the size of the change. In particular, changing a header files should not force recompilation of all files that happen to include it.

The design goal for the Montana architecture is that it can be extended in a variety of ways. A good example is the Montana object model[2] support. Most C++ compilers support a single native object model, the semantics for which are entrenched in the compiler itself, making it difficult to support different object, such as DirectToSOM C++ [Ham96] or other industry object models. Montana, however, was designed so that the object model is supported through a well-defined interface. A new object model can be added without requiring massive changes throughout the compiler. At the time of writing, the author was responsible for the design and development of such non-native object models.

Montana is designed around a system called *CodeStore* [Bar94]. CodeStore consists of a C++ parser, a database that contains the compiled C++ program representation, and a class library that provides an API interface to the compilation process and program representation. Using this class library interface, C++-knowledgeable tools such as browsers can query the program representation of a compiled C++ program. In addition, CodeStore tools called *extensions* can be written that interact with the compilation process.

There are three types of extensions [Sor96]: 1) *CodeStore*

---

* This work was performed while the author was a member of the C++ compiler development group at the IBM Toronto Laboratory.

[1] The name "Montana" originated from an architecture meeting in which the idea of developing a new compiler with a clean slate was referred to as a "blue sky" approach. Since "blue sky" was thought to be the motto for the state of Montana (it's actually "big sky"), that became the name of the project. [Nac96]

[2] By object model, we mean issues such as how objects are laid out in memory and the strategy used to support virtual functions and bases. See [Lip96] for a detailed discussion.

extensions, which add data to the CodeStore and have incremental update capability, 2) *incorporation*[3] *extensions*, that modify or observe the incorporation process directly and 3) *user interface extensions*, which allow additional artifacts such as buttons and menus to be added to the user interface display. An example of the first type of extension is a separate compiler that is triggered as part of the compilation process to handle different file types, while an example of an incorporation extension is a tool that interacts directly with the compilation process itself, querying or updating the result. In this paper, we will concentrate on the second form of extension.

3. Smart Pointers

Smart pointers are a powerful feature of the C++ language that enable a variety of applications, such as garbage collection, persistence, and distributed objects. They are used to augment the functionality of C++ pointer operations, allowing the programmer to perform additional work when pointers are created and used.

Smart pointers essentially allow a user-defined exit added to be pointer operations. A smart pointer [Stro89] itself is an instance of a class that wraps a built-in pointer, for which the dereference operator `->` has been overloaded, as shown in Figure 1 (smart pointers are typically defined using templates however). Such objects can be used in much the same way as a built-in pointer, but have additional functionality provided through operator overloading. In much the same way as inheritance, smart pointers can be used in C++ to extend the functionality of a class. However, while inheritance extends the functionality of class instances themselves, smart pointers are used to extend the environment containing the instance. In other words, smart pointers are used to modify how the programming environment operates on an object, rather than how the object operates on itself. Smart pointers have a wide variety of uses, from simple applications such as detecting null dereferences, debugging, and read-only pointers [Alg95], to more complex applications such as garbage collection [GC96], [Ede92a], and persistence [Coh96].

```
#include <iostream.h>

struct S {
    int i;
};

class SP {
    S *_p;
  public:
    SP(S *p) : _p(p) {}
    S* operator->() {
        cout << "dereferencing" << endl;
        return _p;
    }
};

int main()
{
    SP sp(new S);

    sp->i = 10;       // sp.operator->()->i
}
```

Figure 1 Simple Smart Pointer Class

In general, smart pointers can be used in exactly the same way as built-in pointers, however, as described in [Ede92b], there are some important differences between the two with respect to implicit type conversions performed by the compiler. These fall into two major categories: 1) class hierarchies and 2) types qualified with `const` or `volatile`. A further issue, described in [Mey96a] and [Mey96b], is testing for nullness. When using built-in pointers, the compiler implicitly performs a variety of conversions between pointer types. Examples are `T*` to `const T*`, `Derived*` to `Base*`, and `T*` to `void*`. These implicit conversions are not supported for smart pointer types.

If, however, the "smarts" of a smart pointer could be added to a built-in pointer, these problems would be alleviated. In this section, we describe the changes that would be needed to built-in pointer expressions in order that they operate as smart pointers. For the purpose of this example, we will implement a reference counting smart pointer. In subsequent sections, we will describe how to implement this model using a Montana incorporation extension.

3.1 A Reference Counting Smart Pointer

The basic model for a reference-counting smart pointer is as follows:

1) Whenever a new reference is made to a given object, the reference count for that object should be incremented.
2) If a reference to an object is removed, the reference count for that object should be decremented. If the reference count for an object goes to zero, delete the object.

These rules are illustrated by the functions `increment`

[3] *Incorporation* is the Montana term for recompiling a program, in which the changes to the source will be incorporated into the CodeStore database.

```
void decrement(ReferenceCounter *sp)
{
    if (!sp)
        return;
    if (! --sp->rc)
        delete sp;
}

void increment(ReferenceCounter *sp)
{
    if (!sp)
        return;
    ++sp->rc;
}
```

Figure 2 Reference Counter Functions

and `decrement` shown in Figure 2.

In order to add reference counting smart pointer functionality to built-in pointers operations, the following expression transformations are required:

Pointer assignment: Whenever an assignment is made to a designated smart built-in pointer, the reference count for the object originally pointed to should be decremented and that of the object now pointed to should be incremented. Thus, the expression `p1 = p2` becomes:

```
(p1 == p2 ? 0 : decrement(p1),
   p1 = p2, increment(p1), p1)
```

Pointer initialization: A designated smart built-in pointer must always be either explicitly initialized to a value, or to zero. (If a pointer were not initialized to zero and contained non-zero garbage, a subsequent assignment to that pointer using the previous expression would likely result in an exception).

The statement `SPC* p1;` becomes:

```
SPC* p1 = 0;
```

and `SPC** p1 = new SPC*;` becomes:

```
SPC** p1=new SPC*; p1 ? *p1=0 : 0;
```

If a smart pointer is initialized to a value, the reference count for the underlying object must be incremented. So the statement `SPC* p1 = p2;` becomes:

```
SPC* p1 = p2; increment(p1);
```

Object Initialization: When a designated smart built-in pointer object is created, the reference count must be initialized. For dynamically-created objects, the count should be initialized to 0, and for static or automatic objects, the reference count should be initialized to 1 so that the object can be used in reference counting contexts, but will never be deleted.

Pointer destruction: When a designated smart built-in pointer is destroyed the reference count for the referenced object must be decremented. There are several ways that a smart pointer will be destroyed, the most common being that it goes out of scope. Other possibilities are that a dynamically allocated smart pointer is deleted, or an exception occurs in which the containing block is unwound from the stack. Only the deletion of a dynamically-allocated smart pointer consists of an expression that can be transformed. The other two require modifications to the function itself so that the scope termination and exception handling code will include the decrement of any smart pointers declared therein.

3.2 Which Built-in Pointers Become Smart?

The above discussion raises the question of how to determine which built-in pointer operations should be transformed into smart pointer operations. One could blindly apply the transformation to all built-in pointer operations, but this would certainly be overkill. Rather, we would like to select only specific pointers for the transformation The approach that we have chosen is to define a special base class, `ReferenceCounter`. Expressions involving objects declared of, or pointers to, a class derived from `ReferenceCounter` will be transformed as described above.

For example, consider the built-in pointers declared of type `C*` in Figure 3. Because class `C` is derived from `ReferenceCounter`, several transformations should take place. `cp1` and `cp2` should be implicitly initialized to 0 at the point of declaration, and the assignment from `cp2` to `cp1` should be transformed as described earlier.

```
class C : public ReferenceCounter {};

int main()
{
    C *cp1, *cp2;

    cp1 = cp2;
}
```

Figure 3 Built-in Pointer Operations

4. Montana CodeStore Architecture

The previous section described the necessary

transformations to built-in pointer operations in order to implement a reference counting smart pointer. This section provides an overview of the Montana CodeStore architecture and describes in a generic sense how a transformation extension can be added. This will form the basis for the remainder of the paper, which describes our implementation of smart built-in pointers in Montana using an incorporation extension.

4.1 The Montana Incorporation Process

As part of the Montana incremental compilation process, the compiler separates a source file into *regions*, where each region consists of approximately one declaration. If a region, or something that region depends upon, has changed since the last incorporation, it is re-incorporated. Re-incorporation involves a number of standard steps: parsing, semantic analysis, transformation, error checking, code generation, and incremental linking. In addition, dependency arcs are added between CodeStore elements so that a change in one region can trigger a re-incorporation of a dependent region. For example, a region containing a derived class declaration will have a dependency on each region containing one of its base classes.

The Montana class `CS_CodeStore` is used to represent the underlying CodeStore database. This class supports a variety of routines to create, query and update the CodeStore. An application that operates on a CodeStore will contain exactly one instance of the `CS_CodeStore` class. If an incorporation is currently taking place against the database, this `CS_CodeStore` instance will contain a reference to an object of type `CS_IncorporationState`, which represents the current state of the incorporation.

4.2 Transformation

The transformation step involves simplifying expressions and statements into a C-like representation. In Montana, it is implemented through the class `CS_Transformer`, which is shown in Figure 4. `CS_Transformer` has three versions of the method `transform`, corresponding to the different types of transformations that are supported. Most calls to the `CS_Transformer::transform` methods are for statements or initializations. Expression transformations typically occur as part of the transformation of their containing statement.

When the compiler needs to transform an item, it obtains a transformer object from the CodeStore's incorporation state object. The incorporation state in turn retrieves the transformer from an implementation component factory (see Figure 5). The incorporation state maintains a list of implementation component factories, and selects the transformer returned by the front element in the list.

An implementation component can be one of four types: a type analyzer, a diagnostician, a transformer, or an optimizer. Each of these implementation components take part in a specific portion of the incorporation process, and can be overridden to modify the compilation process. Applications can provide custom implementation components by subclassing the implementation component factory class and providing overrides for the methods of interest. By inserting this new class at the front of the incorporation state list, the incorporation state will select the overridden component provided.

For example, to provide a transformer (incorporation) extension, the class `CS_ImplementationComponentFactory` would be derived from, supplying a transformer method that would return the custom transformer object. The incorporation state method `prependImplementationComponentFactory` would be called to add this factory to the front of the list. Any factory methods that are not overridden would return the result of invoking that method against the next factory in the list, as shown in Figure 5 with the method invocation against the result of the `next` method.

Montana supplies a default implementation component factory that provides the standard implementations for each component. When no extensions have been introduced, this factory will be at the front of the incorporation state's list. Figure 6 shows the relationship between the various classes discussed in this section.

5. Implementing Smart Pointers with Montana

In this section, we will present our implementation of smart pointers through built-in pointers using a Montana transformation incorporation extension. The complete implementation is included in the Appendix A, and we have extracted specific pieces to clarify the explanation. We will first describe how to add our specific transformation extension, and then present our implementation for reference counting smart pointers based on the required expression transformations discussed earlier.

```
class CS_Transformer : public CS_IncorporationComponentBase<CS_DepthFirstModifier>
{
public:
   CS_Transformer(CS_IncorporationState& s) :
CS_IncorporationComponentBase<CS_DepthFirstModifier>(s) { }

   // Transform a statement tree
   //
   virtual CS_bool transform(CS_Statement*& stmt, CS_bool emitMessages)
      { modifyStatement(*stmt); return CS_true; }

   // Transform a variable initializer
   //
   virtual CS_bool
         transform(CS_Initializer*& init, CS_VariableDeclaration& var, CS_bool emitMessages)
      { init = &modifyInitializer(*init, &var.typeDescriptor(), &var); return CS_true; }

   // Transform an expression tree
   //
   virtual CS_Expression& transform(CS_Expression& expr, CS_bool emitMessages)
      { return modifyExpression(expr); }
};
```

Figure 4 `CS_Transformer`

```
class CS_ImplementationComponentFactory : public CS_Link<CS_ImplementationComponentFactory>
{
public:
   virtual CS_TypeAnalyzer&   typeAnalyzer()  { return next()->typeAnalyzer();  }
   virtual CS_Diagnostician&  diagnostician() { return next()->diagnostician(); }
   virtual CS_Optimizer&      optimizer()     { return next()->optimizer();     }
   virtual CS_Transformer&    transformer()   { return next()->transformer();   }
};
```

Figure 5 `CS_ImplementationComponentFactory` class

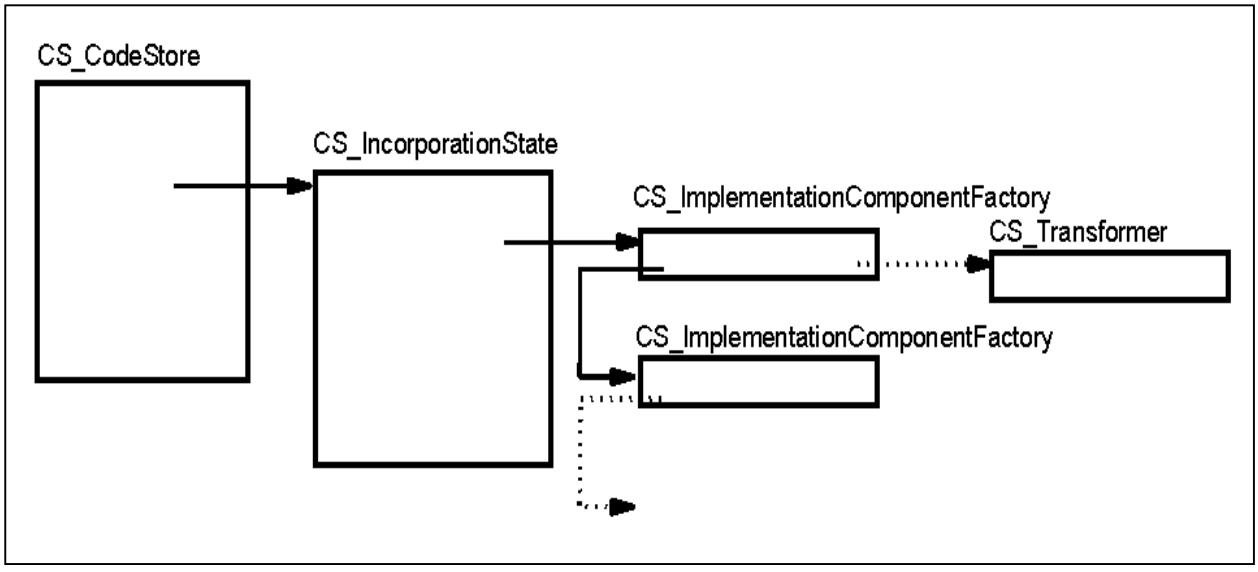

Figure 6 Relationship Between Classes

```
SmartPointerImplementationComponentFactory::
        SmartPointerImplementationComponentFactory(CS_IncorporationState& s) :
    _state(s),
    // The second argument to this constructor comes from pull on the
    // chain of components stored in the IncorporationState.
    //
    _transformer(new SmartPointerTransformer(_state,
                 _state.implementationComponentFactory().transformer()))
{
    assume(_transformer);
}

CS_Transformer& SmartPointerImplementationComponentFactory::
        transformer()
{
    assume(_transformer);
    return *_transformer;
}

SmartPointerImplementationComponentFactory::
        ~SmartPointerImplementationComponentFactory()
{
    delete _transformer;
}
```

Figure 7 SmartPointerImplementationComponentFactory implementation

```
class SmartPointerImplementationComponentFactory
   : public CS_ImplementationComponentFactory {
  public:
    SmartPointerImplementationComponentFactory(CS_IncorporationState&);
    virtual CS_Transformer& transformer();
    virtual ~SmartPointerImplementationComponentFactory();

  private:
    CS_IncorporationState& _state;
    CS_Transformer* _transformer;
};
```

Figure 8 `SmartPointerImplementationComponentFactory` class

5.1 Creating a Transformation Incorporation Extension

As described in the previous section, in order to implement a transformation extension, we must create a subclass of the `CS_ImplementationComponent-Factory` class and insert an object of this new type at the front of the incorporation state's factory list. Figure 8 shows the definition of the class `SmartPointerImplementationComponent-Factory` and Figure 7 shows the corresponding implementation. The method transformer is overridden to return our custom smart pointer transformer extension. The constructor for the factory initializes the `_transformer` member by creating a new object of class `SmartPointerTransformer`. This latter class will implement our smart pointer transformer extension, and will be discussed in more detail subsequently. Note that the transformer extension constructor is passed the incorporation state and the current transformer object, obtained from the front of the factory list.

5.2 Dynamically Loading a Transformation Incorporation Extension

We now have a factory implementation that will return our custom transformation extension. The next issue to address is how the factory object will be created and added to the front of the incorporation state's factory list. This will be achieved by loading a dynamic link library (DLL) that contains a static variable whose initialization will cause the factory to be created and inserted into the list. Then the question is, how is the DLL loaded? We will now examine the Montana support for defining and loading extensions.

Externally, a Montana extension is introduced using an *Incremental C++ Extension*, or *ice*, file. Montana searches for and applies ice files at load time according to a defined search order. Figure 9 shows an ice file which defines an extension called `SmartPointer`, for which the corresponding DLL to load is `smartp.dll`. The suffix and prefix information is used to associate a

specific file type with a given extension. This interface is for CodeStore extensions, but is used as a temporary measure until the final interface for incorporation extensions is defined.

Montana programs are compiled by providing a *configuration file* which supplies the various options for the compilation. For our purposes, the configuration file shown in Figure 10 is used.

This configuration file indicates that the source file to be compiled is `t.cpp`, the target executable will be `t.exe`, and that an additional source file called `dummy.sp` will also be processed. This latter source file, being an unsupported file type, will cause Montana to search the ice files for an appropriate extension that handles this file type, and load the extension DLL `smartp.dll`.

The next step is to register an *extension dynamic load point* using a statically-defined variable in the `smartp.dll` extension DLL as shown in Figure 11. An extension dynamic load point is used to register an extension with the compiler. For an incorporation extension, the final parameter to the extension dynamic load point constructor, the *incorporation startup function pointer* is most important. This function will be run at the start of every incorporation and can be used by an extension to plug in components into the incorporation state.

```
[SmartPointer]

type=extension
description=Smart Pointer
Extension
dll=smartp.dll
suffixes=sp SP
prefix=dummy
```

Figure 9 ice File for Smart Pointer Extension

```
source type(cpp) src0 = "t.cpp"
target "t.exe" { source src0 }

source type(sp) src1 = "dummy.sp"
```

Figure 10 Montana Configuration File

```
CS_ExtensionDynamicLoadPoint

SmartPointer::extension_load_point(
        SmartPointer::className(),
        SmartPointer::update,
        SmartPointer::isChanged,

SmartPointer::processOptions,
        EXTENSION_PRIORITY,

SmartPointer::incorporationStartup);
```

Figure 11 Extension Dynamic Load Point

```
class SmartPointer : public CS_InterfaceBase
{
public:
   static const char* className();

   static void SmartPointer::incorporationStartup(
      CS_ExtensionDynamicLoadPointLink&, CS_IncorporationState&);

   static CS_DependencyNode::UpdateResult
        update(CS_ExtensionSource* me, CS_IncorporationState& state,
               CS_bool emitMessages);

   static CS_bool isChanged(CS_ExtensionSource* me);

   static void processOptions(CS_ExtensionSource* me, CS_OptionList& options);

private:
   static CS_ExtensionDynamicLoadPoint  extension_load_point;
};
```

Figure 12 `SmartPointer` class

```
void SmartPointer::incorporationStartup(
        CS_ExtensionDynamicLoadPointLink&, CS_IncorporationState& state)
{
    cout << __FUNCTION__ << endl;

    SmartPointerImplementationComponentFactory* fac =
        new SmartPointerImplementationComponentFactory(state);
    assume(fac);  // (our version of "assert")

    // Push our new factory with its new Transformer onto the chain
    // stored in the IncorporationState.
    //
    state.prependImplementationComponentFactory(*fac);

    return;
}
```

Figure 13 `incorporationStartup` method

The main effect then, of constructing the static member variable `SmartPointer::extension_load_point` is that the method `SmartPointer::incorporationStartup` will be called prior to each incorporation. The class `SmartPointer` and the `incorporationStartup` method are shown in Figure 12 and Figure 13. In the `incorporationStartup` method the newly-created `SmartPointerImplementationComponentFactory` object is added to the front of the incorporation state's factory list (see Figure 6). Adding this custom factory object to the front of the queue will cause any requests made of the incorporation state for a transformer object to return the custom transformer, `SmartPointerTransformer.`

5.3 The SmartPointerTransformer class

At this point, whenever a transformation takes place, the `SmartPointerTransformer` class (see Figure 14) will have control. One of the three overridden `transform` methods shown at the beginning of the class will be called depending upon the type of transformation taking place: a statement, initialization, or an expression. The overridden versions of the `SmartPointerTransformer` methods are shown in Figure 15. These `transform` methods have fairly standard implementations. They first call an appropriate `modify` method, and then invoke the `transform` method of the previous element in the component chain (given by member variable `_parent.`)

Recall that the constructor for the `SmartPointerTransformer` class is passed the current transformer object, which is used to initialize the data member `_parent`. Calling the parent `transform` method allows the standard compiler transformations to take place after the extension has been run.

It is when the `modify` method is called that the transformer extension has an opportunity to modify the transformed expression. The compiler-supplied `modify` methods step through the underlying item and calls an appropriate `modifyxxx` method for each entity encountered. By overriding methods corresponding to expression of interest, the transformer extension can modify these expressions. In this case, we have overloaded `modifyAssignExpression, modifyExpressionInitializer, modifyImplicitInitializer`, and `modifyDestructorStateChangeExpression` (these methods will be explained in more detail in the next section). If the underlying expression or statement corresponds to one of these four, the overloaded method will be called. Each of these methods determines if any further expression transformation is necessary, based on the type of the object being operated on. If so, the expression is transformed according to the model described earlier for transforming built-in pointer operations into smart pointer operations.

5.4 CS_SmartPointerTransformer::modify

Now we will discuss the implementation of the `CS_SmartPointerTransformer::modify` methods. Each of these methods uses the `SmartPointerTransformer::transformerImplementation` method to determine if the current expression or statement deals with an object of interest. This method returns a `SmartPointerTransformationImplementation` that will perform implementation-specific transformations, depending upon the smart pointer type. We will discuss this latter class in the next section.

```
class SmartPointerTransformer : public CS_Transformer {
  public:
    SmartPointerTransformer(CS_IncorporationState& s, CS_Transformer& p)
       : CS_Transformer(s), _parent(p),
         _referenceCounterTransformerImplementation(0) {
    }
    ~SmartPointerTransformer();

    virtual CS_bool transform(CS_Statement*& stmt, CS_bool emitMessages);
    virtual CS_bool transform(CS_Initializer*&, CS_VariableDeclaration&, CS_bool);
    virtual CS_Expression& transform(CS_Expression&, CS_bool);

    virtual CS_Expression& modifyAssignExpression(CS_BinaryExpression&);
    virtual CS_Initializer& modifyExpressionInitializer(
        CS_ExpressionInitializer&, CS_TypeDescriptor*, CS_VariableDeclaration*);
    virtual CS_Initializer& modifyImplicitInitializer(
        CS_ImplicitInitializer&, CS_TypeDescriptor*, CS_VariableDeclaration*);
    virtual CS_Expression& modifyDestructorStateChangeExpression(
        CS_DestructorStateChangeExpression&);

    CS_Expression& typeAnalyze(CS_Expression&);
    CS_Initializer& typeAnalyze(CS_Initializer&);

  private:
    CS_Transformer& _parent;

    // classes for each smart pointer implementation
    ReferenceCounterTransformerImplementation* _referenceCounterTransformerImplementation;

    // Return the smart pointer implementation, if any, for the expression
    // The expression must be a pointer to a class derived from a SmartPointer class
    SmartPointerTransformerImplementation* transformerImplementation(CS_TypeDescriptor&);
};
```

Figure 14 `SmartPointerTransformer` **class**

```
CS_bool SmartPointerTransformer::transform(CS_Statement*& stmt, CS_bool emitMessages)
{
    modifyStatement(*stmt);
    _parent.transform(stmt, emitMessages);
    return CS_true;
}

CS_bool SmartPointerTransformer::
        transform(CS_Initializer*& init, CS_VariableDeclaration& var, CS_bool emitMessages)
{
    init = &modifyInitializer(*init, &var.typeDescriptor(), &var);
    _parent.transform(init, var, emitMessages);
    return CS_true;
}

CS_Expression& SmartPointerTransformer::transform(CS_Expression& expr, CS_bool emitMessages)
{
    CS_Expression *expr2 = &modifyExpression(expr);
    return _parent.transform(*expr2, emitMessages);
}
```

Figure 15 `SmartPointerTransformer transform` **methods**

```
CS_Expression& SmartPointerTransformer::modifyAssignExpression(CS_BinaryExpression& binary)
{
    if (! binary.expression1().typeDescriptor().isPointer())
        return binary;

    SmartPointerTransformerImplementation *ti =
        transformerImplementation(*binary.expression1().typeDescriptor().next());

    return ti ? typeAnalyze(ti->modifyAssignExpression(binary)) : binary;
}
```

Figure 16 `modifyAssignExpression` **method**

```
CS_Initializer& SmartPointerTransformer::modifyExpressionInitializer(
        CS_ExpressionInitializer& init, CS_TypeDescriptor* td, CS_VariableDeclaration* var)
{
    if (!td)
        return init;

    // need to handle both pointer and object initialization
    SmartPointerTransformerImplementation *ti =
        transformerImplementation(td->isPointer() ? *td->next() : *td);

    if (! ti)
        return init;

    return typeAnalyze(ti->modifyExpressionInitializer(init, td, var));
}
```

Figure 17 `modifyExpressionInitializer` method

```
CS_Initializer& SmartPointerTransformer::modifyImplicitInitializer(
        CS_ImplicitInitializer& init, CS_TypeDescriptor* td, CS_VariableDeclaration* var)
{
    if (!td)
        return init;

    // need to handle both pointer and object initialization
    SmartPointerTransformerImplementation *ti =
        transformerImplementation(td->isPointer() ? *td->next() : *td);

    if (! ti)
        return init;

    return typeAnalyze(ti->modifyImplicitInitializer(init, td, var));
}
```

Figure 18 `modifyImplicitInitializer` method

```
CS_Expression& SmartPointerTransformer::
        modifyDestructorStateChangeExpression(CS_DestructorStateChangeExpression& dsce)
{
    return SmartPointerTransformerImplementation::
        modifyDestructorStateChangeExpression(dsce);
}

CS_Expression& SmartPointerTransformerImplementation::
    modifyDestructorStateChangeExpression(CS_DestructorStateChangeExpression& dsce)
{
    // save most recent state table entry
    if (dsce.tableEntry() && dsce.tableEntry()->asDestructorStateTableEntry())
        _currentDestructorStateTableEntry = dsce.tableEntry()->asDestructorStateTableEntry();

    return dsce;
}
```

Figure 19 `modifyDestructorStateChangeExpression` methods

The `modifyAssignExpression` method is shown in Figure 16. For our smart pointer implementation, we want to detect any pointers assignments where the underlying type is derived from a special base class such as `ReferenceCounter`. `modifyAssignExpression` is passed a reference to a `CS_BinaryExpression` object, representing the assignment expression currently being transformed. If the type descriptor for the lhs of this expression, given by `CS_BinaryExpression::expression1()`, is not a pointer, then no further work is necessary. If it is a pointer, then the type to which it points, given by `CS_TypeDescriptor::next()` is passed to `transformerImplementation` to check if it points to an object derived from one of the special base classes. If a non-null value is returned, the `modifyAssignExpression` of the returned implementation object is called to modify the expression.

The `modifyExpressionInitializer` and `modifyImplicitInitializer` methods (Figure 17 and Figure 18) perform similar functions, but must handle both pointer and non-pointer initializations. These methods call the `transformerImplementation` with the `CS_TypeDescriptor` for the non-pointer variable being initialized. If the variable being initialized is a pointer, the `CS_TypeDescriptor` for the type pointed to, given by `td->next()`, is passed instead.

The `modifyDestructorStateChange-Expression` method, shown in Figure 19, does not actually perform any modification against the given expression. Rather, it calls the static method `SmartPointerTransformationImplementation::modifyDestructorStateChangeExpression`, which simply saves the most recently seen state table entry if that entry is for a destructor. This value will be used later to add information to the state table in the appropriate location.

5.5 SmartPointerTransformerImplementation class

As discussed earlier, the model for our smart pointer implementation is that a built-in pointer will be transformed into a smart pointer if the underlying type inherits from a special base class. In order to provide multiple smart pointer transformations, we have defined a common base class called `SmartPointer-TransformerImplementation` (Figure 20), from which a derived class will be defined for each smart pointer implementation. This derived class will handle the transformations specific to that smart pointer implementation.

When a method needs to determine if a smart pointer implementation applies to a given expression, it calls the method `SmartPointerTransformer::transformerImplementation`, shown in Figure 21. The `transformerImplementation` method is passed a reference to an object of type `CS_TypeDescriptor`, which describes the type of the object being operated on. If the object is of a class type, a reference to the associated `CS_ClassDeclaration` is assigned to the variable `decl`. A `CS_ClassDeclaration` provides complete information about a class declaration, such as the class name, members. etc. So at this point, `decl` will reference the class declaration for the object of interest.

Next, the `findClassDeclaration` method of the `ReferenceCounterTransformerImplementation` class is invoked. This method returns a pointer to the `CS_ClassDeclaration` object representing the class `ReferenceCounter`, if that declaration has been encountered in the program, and null otherwise. If non-null is returned, the method uses the `CS_ANSI_Queries::isBaseClassOf` method to determine if the class declaration for the object of interest is derived from `ReferenceCounter`. The `CS_ANSI_Queries` class provides a variety of functions that support querying of class declarations. If `ReferenceCounter` is a base class, the `_referenceCounterTransformer-Implementation` data member will be initialized with a new `ReferenceCounterTranformer-Implementation` object if it has not yet been initialized.

In the current implementation, we have defined one special base class, `ReferenceCounter`, and one corresponding specialization of `SmartPointerTransformerImplementation`. The programmer would include the declaration of `ReferenceCounter` class (see Figure 22) and derive from it to introduce reference-counting functionality for pointer operations against objects of that derived class. (The `name` member in `ReferenceCounter` is used for debugging purposes and will be discussed later). Additional smart pointer implementations could be introduced by inserting code at the end of the `transformerImplementation` method where indicated.

5.6 ReferenceCounterTransformerImplementation class

The ReferenceCounterTranformerImplementation (see Figure 23) class provides the transformer extension implementation for a reference counting smart pointer. It is a specialization of the `SmartPointer-TransformerImplementation`, and contains overrides of the `SmartPointerTransformer-Implementation::modify` methods, along with additional methods specific to implementing a reference counting smart pointer.

```
class SmartPointerTransformerImplementation : public
CS_IncorporationComponentBase<CS_InterfaceBase> {
  public:
    SmartPointerTransformerImplementation(CS_IncorporationState &state,
                                          SmartPointerTransformer &transformer) :
        CS_IncorporationComponentBase<CS_InterfaceBase>(state), _transformer(transformer) {}

    virtual CS_Expression& modifyAssignExpression(CS_BinaryExpression&) = 0;
    virtual CS_Initializer& modifyExpressionInitializer(
        CS_ExpressionInitializer&, CS_TypeDescriptor*, CS_VariableDeclaration*) = 0;
    virtual CS_Initializer& modifyImplicitInitializer(
        CS_ImplicitInitializer&, CS_TypeDescriptor*, CS_VariableDeclaration*) = 0;

    static CS_Expression& modifyDestructorStateChangeExpression(
        CS_DestructorStateChangeExpression& dsce);
    CS_DestructorStateTableEntry* currentDestructorStateTableEntry()
        { return _currentDestructorStateTableEntry; }
    void currentDestructorStateTableEntry(CS_DestructorStateTableEntry *ste)
        { _currentDestructorStateTableEntry=ste; }

    SmartPointerTransformer& transformer() { return _transformer; }

  private:

    SmartPointerTransformer& _transformer;
    static CS_DestructorStateTableEntry *_currentDestructorStateTableEntry;
};
```

Figure 20 SmartPointerTransformerImplementation class

```
SmartPointerTransformerImplementation *SmartPointerTransformer::
        transformerImplementation(CS_TypeDescriptor& td)
{
    if (! td.isNamedType() ||
          ! td.declaration().declarationKind() == CS_Declaration::IsClass)
        return NULL;
    CS_ClassDeclaration &decl = *td.declaration().asClassDeclaration();

    CS_ClassDeclaration *referenceCounter =
        ReferenceCounterTransformerImplementation(state(), *this).findClassDeclaration();
    if (referenceCounter &&
          CS_ANSI_Queries::isBaseClassOf(*referenceCounter, decl)) {
        cout << "got a pointer to class derived from "
             << referenceCounter->signature() << endl;
        if (! _referenceCounterTransformerImplementation) {
            _referenceCounterTransformerImplementation =
                new ReferenceCounterTransformerImplementation(state(), *this);
        }
        return _referenceCounterTransformerImplementation;
    }
    // insert code to look for other smart pointer class implementations here
    return NULL;
}
```

Figure 21 `transformerImplementation` method

```
class ReferenceCounter {
  private:
    int rc;
    char *_name;

    static void dtor(ReferenceCounter **sp, int);
    static void decrement(ReferenceCounter *sp);
    static void increment(ReferenceCounter *sp);

  public:
    ReferenceCounter(char *name) : _name(name) { rc = 0; }
    char *name() { return _name; }
    virtual ~ReferenceCounter();
};
```

Figure 22 ReferenceCounter class

```
class ReferenceCounterTransformerImplementation :
        public SmartPointerTransformerImplementation {
  public:
    ReferenceCounterTransformerImplementation(
        CS_IncorporationState& state, SmartPointerTransformer& transformer) :
        SmartPointerTransformerImplementation(state, transformer) {};

    CS_ClassDeclaration* findClassDeclaration();

    CS_Expression& modifyAssignExpression(CS_BinaryExpression& binary);
    CS_Initializer& modifyExpressionInitializer(
        CS_ExpressionInitializer&, CS_TypeDescriptor*, CS_VariableDeclaration*);
    CS_Initializer& modifyImplicitInitializer(
        CS_ImplicitInitializer&, CS_TypeDescriptor*, CS_VariableDeclaration*);

    private:
        const CS_Atom& referenceCountMember();
        virtual CS_Expression& decrementReferenceCounterExpression(CS_Expression &);
        virtual CS_Expression& incrementReferenceCounterExpression(CS_Expression &);
        virtual CS_Expression& createStateChangeExpression(
            CS_Expression&, CS_VariableDeclaration&);
        virtual CS_FunctionDeclaration& findOrCreateDecrement();
        virtual CS_FunctionDeclaration& findOrCreateIncrement();
        virtual CS_FunctionDeclaration& findOrCreateDtor();
        virtual CS_FunctionDeclaration& findOrCreateMemberFunction(char *);
        virtual CS_DestructorStateTableEntry&createDestructorStateTableEntry(
            CS_VariableDeclaration&, CS_TreeNode&);
        virtual void addDestructorCalls(
            CS_VariableDeclaration&, CS_DestructorStateTableEntry&, CS_TokenLocation&);
};
```

Figure 23 ReferenceCounterTransformerImplementation class

```
CS_Expression&
ReferenceCounterTransformerImplementation::modifyAssignExpression(
    CS_BinaryExpression &binary)
{
    // Don't transform expressions on temporaries
    CS_Expression &e = binary.expression1();
    if (e.expressionKind() == CS_Expression::IsName &&
            e.asNameExpression()->name().declaration() &&
            ! e.asNameExpression()->name().declaration()->mapsToASourceLocation()) {
        return binary;
    }

    CS_TokenLocation loc =
        binary.sourceLocation().sourceRegion()->tokenLocation();

    CS_Expression &expr =
        ef().createCommaExpression(loc,
            decrementReferenceCounterExpression(binary.expression1()),
            ef().createCommaExpression(loc,
                binary,
                incrementReferenceCounterExpression(binary.expression1())));
    return expr;

}
```

Figure 24 modifyAssignExpression method

```
CS_Initializer& ReferenceCounterTransformerImplementation::
        modifyExpressionInitializer(CS_ExpressionInitializer& init,
                                    CS_TypeDescriptor* td, CS_VariableDeclaration* var)
{
    assume(var);

    CS_TokenLocation loc =
        init.expression().sourceLocation().sourceRegion()->tokenLocation();

    if (td->isPointer()) {
        CS_BinaryExpression *be = init.expression().asBinaryExpression();
        assume(be);
        // C *c1(x) will already be transformed to C *c1 = x;
        assume(be->binaryExpressionKind() == CS_BinaryExpression::opAssign);

        CS_Expression &expr1 = be->expression1();

        // change C *c1; c1 = x to
        //        C *c1; c1 = (c1=x, c1 != 0 ? c1->rc++: 0, c1)

        init.setExpression(
            ef().createAssignExpression(loc,
                expr1,
            ef().createCommaExpression(loc,
                ef().createAssignExpression(loc,
                    ic().cloneExpression(expr1),
                    transformer().modifyExpression(be->expression2())),
                ef().createCommaExpression(loc,
                    incrementReferenceCounterExpression(
                        expr1),
                    ef().createCommaExpression(loc,
                        createStateChangeExpression(expr1, *var),
                    ic().cloneExpression(expr1))))));
        return init;
    }
    // For non-dynamic variables, initialize reference count
    // to 1 so that never get collected.

    init.setExpression(
        ef().createCommaExpression(loc,
            transformer().modifyExpression(init.expression()),
            ef().createAssignExpression(loc,
                ef().createDotExpression(loc,
                    ef().createNameExpression(loc, *var),
                    referenceCountMember()),
                ef().createLiteralExpression(
                    cs(), loc, intType(), 1))));
    return init;
}
```

Figure 25 modifyExpressionInitializer method

5.6.1 Transforming Pointer Assignments

As discussed earlier, when an assignment to a reference-counting smart pointer occurs, we want to transform an expression such as `x = y`, where x points to a type derived from `ReferenceCounter`, to the following:

```
(x == y ? 0 :
decrement(x), x = y,
increment(x), x)
```

This is achieved through the `modifyAssignExpression` method shown in Figure 24. The first thing the method does is check if the left-hand-side (lhs) of the expression (given by `binary.expression1()`), is a compiler-generated temporary. Unlike programmer-declared variables, the declaration of a temporary will not have a corresponding source location. Our implementation does not currently handle temporaries, and assignments to such are ignored. Temporaries are discussed in more detail in section 7.

If the target of the assignment is not a temporary, a new comma expression is created using the result of the `decrementReferenceCounter` and `incrementReferenceCounter` methods along with

the current assignment expression. The `findOrCreateDecrement` method called in the `decrementRefereneCounterExpression` method locates or creates a declaration corresponding to the `ReferenceCounter::increment` method declared earlier. Note that the right-hand-side (rhs) value is used three times in the resulting expression, in decrement, assignment, and increment expressions. We should generate a temporary to hold the rhs value so that expressions containing side-effects are not executed multiple times. Further, we currently do not generate code to handle the initial check for the lhs being equal to the rhs, which requires a temporary for both the lhs and the rhs, or the final expression containing just the rhs for the assignment value. This support would be added when temporaries are handled by our implementation (see section 7).

5.6.2 Transforming Initialization Expressions

The `modifyExpressionInitializer` method, shown in Figure 25, transforms initialization expressions corresponding to the model described earlier. Much of this method is fairly self-explanatory. For pointers, however, there is an additional action performed, which is to add state change information. If a pointer goes out of scope, either due to an exception or control implicitly returning from the function, the appropriate reference count decrement must take place. This is achieved through calling the `createStateChangeExpression` method, which will use the saved state change variable to create a state change node and insert it in the table in the appropriate place, based on the most recent state change that occurred within the function. This will cause code to be inserted at the end of the function on implicit scope termination to call the `dtor` method defined for the class `ReferenceCounter`, which will decrement the reference count. See the appendix for details.

6. An Example

To demonstrate the smart pointer transformer extension in action, Figure 26 shows a simple program containing several pointer declarations and assignments. Figure 27 shows the resulting execution output after compiling the program with the transformation extension. The built-in pointers act like smart pointers!

```
#include "ReferenceCounterInterface.h"

class C : public ReferenceCounter {
  public:
    int i;
    C(char *name) : ReferenceCounter(name) {}
};

int main()
{
    cout << "C c1;" << endl;
    C c1("c1");

    cout << "C *cp1 = &c1;" << endl;
    C *cp1 = &c1;

    cout << "C *cp2 = new C(\"new C 1\");" << endl;
    C *cp2 = new C("new C 1");

    cout << "cp2 = 0;" << endl;
    cp2 = 0;

    cout << "cp1 = cp2;" << endl;
    cp1 = cp2;

    cout << "C *cp3 = new C(\"new C 2\");" << endl;
    C *cp3 = new C("new C 2");

    cout << "C c2;" << endl;
    C c2("c2");

    cout << "cp1 = &c2;" << endl;
    cp1 = &c2;

    return 0;
}
```

Figure 26 Test program

```
C c1;
C *cp1 = &c1;
>> Incrementing count for c1 to 2
C *cp2 = new C("new C 1");
>> Incrementing count for new C 1 to 1
cp2 = 0;
>> Decrementing count for new C 1 to 0
>> Deleting new C 1 with 0 references
cp1 = cp2;
>> Decrementing count for c1 to 1
C *cp3 = new C("new C 2");
>> Incrementing count for new C 2 to 1
C c2;
cp1 = &c2;
>> Incrementing count for c2 to 2
>> Deleting c2 with 2 references
>> Decrementing count for new C 2 to 0
>> Deleting new C 2 with 0 references
>> Decrementing count for c2 to -1
>> Deleting c1 with 1 references
```

Figure 27 Test Program Output

```
transformed tree for: int main()
{
  __ef __fsm_tab = { 0xBEEFDEAD, 4, {
      { <offset of c1 + 0>, &C::__dftdt, 1, 16, 0, 0 },
      { <offset of @1 + 0>, &operator delete, -3, 16, 0, 1 },
      { <offset of @2 + 0>, &operator delete, -3, 16, 0, 2 },
      { <offset of c2 + 0>, &C::__dftdt, 1, 16, 0, 3 } } };
  __est __es = { 0, 0, &__fsm_tab, (long int *) 0, 0 };
  *ostream::operator<<(ostream::operator<<((ostream *) &cout, "C c1;"), endl);
  C c1; *C::C(&c1, "c1") , __es.__s = 1;
  *ostream::operator<<(ostream::operator<<((ostream *) &cout, "C *cp1 = &c1;"), endl);
  C *cp1; cp1 = &c1;
  *ostream::operator<<(ostream::operator<<((ostream *) &cout,
      "C *cp2 = new C(\"new C 1\");"), endl);
  C *cp2; cp2 = (( @1 = ::operator new(16) ?
      __es.__s = 2 , C::C(@1, "new C 1") , __es.__s = 1 : 0 ) , @1);
  *ostream::operator<<(ostream::operator<<((ostream *) &cout, "cp2 = 0;"), endl);
  cp2 = 0;
  *ostream::operator<<(ostream::operator<<((ostream *) &cout, "cp1 = cp2;"), endl);
  cp1 = cp2;
  *ostream::operator<<(ostream::operator<<((ostream *) &cout,
      "C *cp3 = new C(\"new C 2\");"), endl);
  C *cp3; cp3 = (( @2 = ::operator new(16) ?
      __es.__s = 3 , C::C(@2, "new C 2") , __es.__s = 2 : 0 ) , @2);
  *ostream::operator<<(ostream::operator<<((ostream *) &cout, "C c2;"), endl);
  C c2; *C::C(&c2, "c2") , __es.__s = 4;
  *ostream::operator<<(ostream::operator<<((ostream *) &cout, "cp1 = &c2;"), endl);
  cp1 = &c2;
  return @3 = 0 , (__es.__s = 3 , C::~C(&c2, 2, 0) , (__es.__s = 0 , C::~C(&c1, 2, 0))) , @3;
}
```

Figure 28 Transformed Expressions Without Smart Pointer Extension

```
transformed tree for: int main()
{
  __ef __fsm_tab = { 0xBEEFDEAD, 7, {
      { <offset of c1 + 0>, &C::__dftdt, 1, 16, 0, 0 },
      { <offset of cp1 + 0>, &ReferenceCounter::dtor, 1, 4, 0, 1 },
      { <offset of cp2 + 0>, &ReferenceCounter::dtor, 1, 4, 0, 2 },
      { <offset of @0 + 0>, &operator delete, -3, 16, 0, 3 },
      { <offset of cp3 + 0>, &ReferenceCounter::dtor, 1, 4, 0, 4 },
      { <offset of @1 + 0>, &operator delete, -3, 16, 0, 5 },
      { <offset of c2 + 0>, &C::__dftdt, 1, 16, 0, 6 } } };
  __est __es = { 0, 0, &__fsm_tab, (long int *) 0, 0 };
  *ostream::operator<<(ostream::operator<<((ostream *) &cout, "C c1;"), endl);
  C c1; *C::C(&c1, "c1") , __es.__s = 1 , c1.rc = 1;
  *ostream::operator<<(ostream::operator<<((ostream *) &cout, "C *cp1 = &c1;"), endl);
  C *cp1; cp1 = (cp1 = &c1, (ReferenceCounter::increment(static_cast<ReferenceCounter *> (cp1)),
      (cp1 , __es.__s = 2 , cp1)));
  *ostream::operator<<(ostream::operator<<((ostream *) &cout,
      "C *cp2 = new C(\"new C 1\");"), endl);
  C *cp2; cp2 = (cp2 = (( @0 = ::operator new(16) ? __es.__s = 4 , C::C(@0, "new C 1") ,
__es.__s = 3 : 0 ) , @0) , (ReferenceCounter::increment(static_cast<ReferenceCounter *> (cp2)),
      (cp2 , __es.__s = 3 , cp2)));
  *ostream::operator<<(ostream::operator<<((ostream *) &cout, "cp2 = 0;"), endl);
  ReferenceCounter::decrement(static_cast<ReferenceCounter *> (cp2)) ,
      (cp2 = 0 , ReferenceCounter::increment(static_cast<ReferenceCounter *> (cp2)));
  *ostream::operator<<(ostream::operator<<((ostream *) &cout, "cp1 = cp2;"), endl);
  ReferenceCounter::decrement(static_cast<ReferenceCounter *> (cp1)) ,
      (cp1 = cp2 , ReferenceCounter::increment(static_cast<ReferenceCounter *> (cp1)));
  *ostream::operator<<(ostream::operator<<((ostream *) &cout,
      "C *cp3 = new C(\"new C 2\");"), endl);
  C *cp3; cp3 = (cp3 = (( @1 = ::operator new(16) ? __es.__s = 6 , C::C(@1, "new C 2") ,
      __es.__s = 5 : 0 ) , @1) ,
      (ReferenceCounter::increment(static_cast<ReferenceCounter *> (cp3)) ,
      (cp3 , __es.__s = 5 , cp3)));
  *ostream::operator<<(ostream::operator<<((ostream *) &cout, "C c2;"), endl);
  C c2; *C::C(&c2, "c2") , __es.__s = 7 , c2.rc = 1;
  foo();
  *ostream::operator<<(ostream::operator<<((ostream *) &cout, "cp1 = &c2;"), endl);
  ReferenceCounter::decrement(static_cast<ReferenceCounter *> (cp1)) ,
      (cp1 = &c2 , ReferenceCounter::increment(static_cast<ReferenceCounter *> (cp1)));
  return @2 = 0 , (__es.__s = 6 , C::~C(&c2, 2, 0) , (__es.__s = 5 ,
      ReferenceCounter::decrement(static_cast<ReferenceCounter *> (cp3))) , (__es.__s = 3 ,
      ReferenceCounter::decrement(static_cast<ReferenceCounter *> (cp2))) , (__es.__s = 2 ,
       ReferenceCounter::decrement(static_cast<ReferenceCounter *> (cp1))) , (__es.__s = 0 ,
      C::~C(&c1, 2, 0))) , @2;
}
```

Figure 29 Transformed Expressions With Extension

Without the transformer extension, the transformed tree for the main function would be as shown in Figure 28. The first line of the transformed function contains a finite state machine table used for exception handling. Each of the 4 entries in the table specifies the action to take should an exception occur during execution of the function. There is an entry for the two local variables requiring destruction, along with the dynamically-allocated storage, in the order they occur in the function. Most of the remaining transformed function is fairly self-explanatory. The state variable `__es.__es` is updated to indicate the progress made, (in other words the state of the function), should an exception occur. The final line handles local destructors, updating the state as each destructor is called in reverse order of declaration. Note that there are no state table entries, state changes, or final destruction code, corresponding to the built-in pointers, as there is no cleanup necessary or possible for them.

Figure 29 shows the transformed tree with the reference counter transformer extension. The first difference is that the state table contains entries for each of the three built-in pointers, calling the `ReferenceCounter::dtor` method. In addition, state changes have been added throughout the function for the built-in pointer declarations. Each declaration or pointer assignment now includes the additional smart pointer functionality that was added as part of the transformation extension. And finally, at the end of the function, the built-in pointers are decremented as they go out of scope, in reverse order of

declaration.

7. Further Work

There are several areas that were not covered by this work, most notably temporaries. The problem with temporaries is that the API currently does not support them very well. There is no model for detecting when a temporary goes out of use, which is necessary in order to correctly apply reference counting. The current model does not include temporaries in the reference count, which is sufficient for some, but not all, cases. Consider an expression such as `cp2 = cp1++;` The initial value of `cp2` must be saved in a temporary prior to the increment of `cp2` in order to be assigned to `cp1,` so the expression would be transformed as follows:

```
cp2 = (@0 = cp1 , cp1 = cp1 + 1 , @0);
```

Applying the smart pointer transformation without taking into account the temporary would yield an incorrect result if the decrement against `cp1` caused the underlying storage to be deleted, resulting in a dangling reference being assigned to `cp2.` The API needs a mechanism for allowing a transformation to determine when a temporary goes out of use so that, for this example, the appropriate reference counting can take place. When such support for temporaries is avaiable, the smart pointer implementation must also be updated to generate temporaries for the modified expressions to avoid multiple evaluation of expressions containing side-effects, as discussed earlier.

Further work also needs to be done in the area of non-implicit scope termination, (for example through a return statement), and exception handling. The current transformation implementation handles reference decrementing only for implicit scope termination, through the state table additions. However, the API does support the capability to handle explicit scope termination, by detecting return statements and using the state information to determine what needs to be done. With respect to exception handling, the current extension implementation does work for simple examples, but not for all cases. Due to time constraints, we did not pursue these areas, but anticipate that the implementation would be fairly straightforward.

8. Conclusion

C++ smart pointers, while similar to built-in pointers, cannot be used interchangeably. Most notably, implicit compiler conversions are not supported for smart pointers. We have proposed that the “smarts” of smart pointers be added to built-in pointers, and presented the expression transformations that would be necessary to implement a reference-counting built-in pointer. Using the Montana API, we have demonstrated a working example of these ideas.

The Montana API interface has proven to be quite complete for the purposes of adding transformation extensions. Other work in this area [Car97] supports this conclusion. We found the API interface and design to be reasonably straightforward and understandable, particularly given the complexity of the problem we were attempting to solve, that of modifying compiler-generated expressions. Nonetheless, adding a transformation extension is not a trivial undertaking, and is more likely to be expected of a class library vendor rather than a casual programmer.

While there is some additional work necessary to allow full support for a reference counting smart pointer implementation, it is clear that the interface is quite capable of handling such language-level extensions. The API definitely need better support for temporaries, both those generated by the compiler and by extensions such as the one we have demonstrated. However, given the flexibility of the interface, this does not seem like a difficult design issue.

9. References

[Alg95] Alger, Jeff; *Secrets of the C++ Masters*, London, England: Academic Press, 1995.

[Bar94] Barton, John; Charles, Phillipe; Chee, Yi-Min; Karasick, Michael; Lieber, Derek; and Nackman, Lee; "CodeStore: Infrastructure for C++-Knowlegeable Tools", Unpublished position paper submitted to OOPSLA 1994 Workshop on Object-Oriented Compilation. http://www.research.ibm.com/softwaretechnology/papers/oopsla94_codestore/position.html

[Car97] Carmichael, Ian; Unpublished working example of using a Montana transformation extension and global program knowledge to de-virtualize virtual function calls.

[Chu94] Churchill, Steve; "Exception Recovery with Smart Pointers", *C++ Report*, January 1994.

[Coh96] Cohen, Shimon; "Lightweight Persistence in C++", *C++ Report*, May 1996.

[Det92] Detlefs, David; "Garbage Collections and Run-time Typing as a C++ Library", *Usenix C++ Technical Conference Proceedings*, June 1992.

[Ede92a] Edelson, Daniel R.; "A Mark-and-Sweep Collector for C++", *Conference Record of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1992.

[Ede92b] Edelson, Daniel R.; "Smart Pointers: They're Smart, but They're Not Pointers", *USENIX C++ Conference Proceedings*, 1992.

[Ede92c] Edelson, Daniel R.; "Precompiling C++ for Garbage Collection", I*nternational Workshop on Memory Management Proceedings*, September 1992.

[GC96] "Great Circle Technology Overview"; Geodesic Systems, 1996.

[Ham96] Hamilton, Jennifer; *Programming with DirectoToSOM C++*, New York, NY: John Wiley, 1996.

[Har96] Harvey, David; "Smart Pointer Templates in C++", 1996, http://web.ftech.net/~honeyg/articles/smartp.htm

[Lip96] Lippman, Stanley; *Inside the C++ Object Model*, Reading, Mass: Addison-Wesley, 1996.

[Mey96a] Meyers, Scott; *More Effective C++*, Reading, Mass: Addison-Wesley, 1996.

[Mey96b] Meyers, Scott; "Refinements to Smart Pointers", *C++ Report*, Nov-Dec, 1996.

[Mil96] Miller, Justin; "Clean Up: C++ Garbage Collection", *Byte*, January 1996.

[Stro91] Stroustrup, Bjarne; *The C++ Programming Language, 2nd Edition*, Reading, Mass. Addison-Wesley, 1991.

[Stro89] Stroustrup, Bjarne; "The Evolution of C++: 1985 - 1989", *Computing Systems*, Summer 1989.

[Stro94] Stroustrup, Bjarne; *The Design and Evolution of C++*, Reading, Mass: Addison-Wesley, 1994.

10. Special References

The following referenced papers are IBM internal documents that we expect will be made publicly available in some form as part of the Montana product. We provide a brief synopsis here.

[Kar96] Karasick, Michael; "So You Want To Write a Tool?", 1996. *Introduces the structure of CodeStore tools that use the query interface and explains the CodeStore programming conventions and idioms.*

[Nac96] Nackman, Lee; "An Overview of Montana", 1996. *Provides an introduction to Montana, it's history, goals, and architecture.*

[Sor96] Soroker, Danny; "Montana White Paper: Extensions", 1996. *Describes the Montana interface for defining and loading extensions.*

[Stre96] Streeter, Dave; "Building Programs with Montana", 1996. *Describes how to use Montana to compile programs and build the Montana database.*