C++ in a Changing Environment

Andrew J. Palay
Silicon Graphics Computer Systems
ajp@sgi.com

Abstract

Current C++ systems have been designed without considering the requirements of environments that make use of shared libraries or dynamic loading. In these environments it must be possible to release new compatible versions of libraries or dynamically loaded components without recompiling portions of the system that make use of the classes defined in these new components. This paper describes our initial work on developing a new C++ system, called ΔC++, that supports class changes with minimal recompilation. With ΔC++, applications linked against a shared library will continue to run, without recompilation, even when a new version of the shared library is released. ΔC++ can also be used to reduce the edit/compile/debug development cycle and provides a clean way to separate interface specifications from implementation.

Introduction

The computer industry is moving toward developing software environments that make heavy use of shared libraries and dynamic loading. Shared libraries allow applications to reuse large amounts of code, thus reducing both memory and disk usage, Shared libraries can also be used by library providers as a mechanism by which they can release new compatible versions of their libraries without requiring applications to be rereleased. Existing applications will automatically see the benefit of using the new library. Dynamic loading can be used to build extensible software systems where developers are encouraged to build relatively small components that can easily be incorporated into already existing applications. This technique has been used in both the Andrew Toolkit [1] and NextStep [2] to support extensible multi-media user interface systems. For systems like this to be successful it must be possible for a user to install and use a component created after the original application into which it is going to be linked.

Releasing new compatible versions of code is one of the keys to supporting both shared libraries and dynamic loading. In the case of shared libraries it must be possible to release a new version of a library without invalidating the applications that use that library. In the case of dynamic loading it must be possible to release a new application without invalidating the loadable components. Similarly, it must be possible to release a new version of a loadable component without invalidating other components that use it. This is not the case with code generated by current C++ systems. A developer must be able release a new library or component that includes compatible changes to a class interfaces. Unfortunately current C++ systems require the recompilation of all code that depend on those changed classes. This results from the fact that current C++ systems resolve all object references at compile time, while in the above environments the information needed to do that resolution is not known until link time..

Over the past year we have been working on a new C++ system, called ΔC++, that solves this problem. We developed a prototype version of ΔC++ that has been used to understand the problems that arise when trying to support C++ in a changing environment. In particular, we were interested in understanding the set of object references that must be resolved at link time and how they should be resolved. We were also interested in understanding whether we can support this level of dynamics without changes to the C++ language (the answer is yes). This paper describes the initial results of this work. We discuss the types of class changes we wish to support and the specific problems that must be solved by C++ systems in order to support these changes. We also describe the runtime solution used by our prototype version of ΔC++, followed by a brief discussion of future work we plan to do toward developing a linktime version of ΔC++ that solves the same problems but without the performance penalty incurred by the prototype. Finally we discuss some other problems that can be solved using the ΔC++ technology.

Types of Compatible Class Changes

Changes that must be supported

In discussing the types of changes we begin with the following definitions for the classes Alpha and Beta:

```
class Alpha {
    public:
        long a;
        long A();
        virtual long B();
};
```

```
class Beta {
    public:
        long x;
        virtual long X();
};
```

There are four major types of changes to class interfaces that we believe must be supported. The simplest change is *member-extension*. It must be possible to add both member functions and variables to a class without forcing the recompilation of any code that uses that class. This must be true for public, protected and private members. For example we must be able to release a new version of `Alpha` that has the following interface without requiring code that uses `Alpha` to be recompiled:

```
class Alpha {
    public:
        long a;
        long b;
        long A();
        virtual long B();
        virtual long C();
};
```

A second form of extension that must be supported is *class-extension*. It must be possible to add a new base class to an already existing class. In our example, we must be able to add `Beta` as a base class to `Alpha`. While `Alpha` will now support additional functionality, it still supports its original interface.

Another type of modification that needs to be supported is *member-promotion*. This is the moving of functionality from a derived class to a base class. Given that `Alpha` is derived from `Beta`, a developer must be free move some of the members from `Alpha` into `Beta`. The new version of `Alpha` will still provide a compatible interface. Users of `Alpha` should not be interested in how the functionality of `Alpha` is provided only that it is provided. Thus a developer should be able to release the following new versions of `Alpha` and `Beta` without requiring any code that uses either class to be recompiled:

```
class Alpha : public Beta {
    public:
        long a;
        long A();
        virtual long B();
};
```

```
class Beta {
    public:
        long x;
        long b;
        virtual long X();
        virtual long C();
};
```

The last major form of modification that we need to support is *override-changing*, especially for member functions. A user of `Alpha` should be unconcerned whether `Alpha` overrides the member function `X()`, originally declared in `Beta`. The function that will get called when invoking `X()` on an instance of `Alpha` will change but the code should still work. A similar case can be made for member variables, although we think supporting *override-changing* of member variables needs to be examined in greater detail.

Given we need to support the above types of modifications, two other modifications, *member-reordering* and *class-reordering*, can be easily be supported. The location of a member in a class need not be fixed for all time. A developer might wish to reorder the member variables in order to make more efficient use of space, or may choose to reorder members in order to group members by their protection level. This reordering has no effect on the interface being provided by the class.

Changes that need not be supported

There are a number of modifications that we have chosen not to support. In each case, supporting such a change would require code modifications that can not be done in an efficient manner These modifications include:

- changing the inheritance of a class from non-virtual to virtual or from virtual to non-virtual.
- changing a method from non-virtual to virtual or from virtual to non-virtual.
- widening the type of a member (eg. from short to long).

Changes that we believe should be supported

There are a set of modifications that we can possibly handle, although there is some disagreement as to whether we should. In each of these cases we currently believe that the modification should be supported but we also understand that a case can be made for not supporting it.

Changes to inline functions are the first such modification which can not be handled without modifying the code that is generated. The only effective method for handling changes to inline member functions is not to inline the functions in the first place. This will have a detrimental effect on the performance of some programs, thus the disagreement on whether it is right to support inline modifications. In reality, we probably can not completely eliminate the use of inline functions, so we will need to educate developers when it is appropriate to use them.

Changing the value of a default parameter to a function is a second controversial type of modification. The controversy is not over performance considerations but with the programmer's understanding of default parameters. A developer can either view the default parameter specification as a statement that the compiler will provide a default (which just happens, for implementation reasons, to be listed in the class specification) or that the compiler will provide the specific default. In the former case the default specification is just a shorthand for declaring and defining a several methods. In the latter case the developer uses the default specification as a shortcut when entering a program. The developer could have typed in the full call with all the parameters specified, but as long as the defaults provide the proper values, those parameters can be skipped. The *C++* language, with its syntax for specifying default parameters, makes it impossible to determine which of these interpretations is correct.

Changing the assignment of constant values within a class is another controversial type of modification. This problem arises when an enumeration with assigned values is declared within a class. The controversy is similar to the problem with default parameters. There is no way to decide whether the values of those constants are part of the visible interface.

Is handling all these modifications necessary?

It can be argued that we are being too adventurous relative to the types of modifications that we wish to support. When people initially look at this problem they focus on the problem of supporting changes to just the private portion of an class interface. This is quite understandable, since the private portion should have no effect on users of a class. A typical solution to this problem is to move what was the private portion of a class into a separate implementation class, and have the original class's private portion contain just a pointer to an instance of that implementation class. While this solves the problem of handling changes to the private portion of the class, it does not solve the problem of adding functionality to a class, nor changing the overriding behavior of the class. When dealing with releasing new versions of shared libraries and dynamic loadable components, functionality improvements are at least as important as handling changes to the private implementation of a class. Arguing this is similar to arguing that new releases of an operating system should not add any functionality, but only include performance improvements that can be made without effecting the public interface provided by the operating system.

Problems to be Addressed

As stated earlier, the basic problem with current C++ systems, is that object resolution is done at compile time. In order to support the types of modifications presented in the previous section the earliest object resolution can be done is link time. The first problem we must solve is determining the exact layout of a class instance. Object code must contain enough information about the class definitions to do that layout. This includes the set of base classes and the member variables to be added by the class. It must be possible to determine the size and required alignment of each member variable. For basic types the size can be provided by the compiler, while for instances of other classes that size must be resolved dynamically. The process of determining the layout of a class will determine the following:

- the size of a class instance
- the offset for each class member
- the location (if any) of any vtable pointers
- the location (if any) of any virtual base class pointers.
- offsets to move between classes.

The offset for each class member will be used to resolve any reference to a member variable. For example, returning to the example presented in the previous section, the expression `pa->a`, where `pa` is a pointer to an instance of `Alpha`, will need to access data at a different offset from `pa` depending on which definition of `Alpha` is being used. The vtable pointers are needed for calling virtual methods and for the code added to constructors and destructors. The virtual base class pointers pointers are needed to access data in any virtual base classes and also for code in the constructors. The interclass offsets are needed to support casts and are also used to determine the offsets associated with member functions.

Just as we need to be able to determine the layout of a class instance, we also must be able to determine the layout and values stored in the vtable for a class. Again, this requires that the object code contains the list of base classes and the member functions provided by the class. From this information, the following will be done:

- allocation of the class's vtable
- determination of offsets for each member function
- initialization of function and offset fields for each vtable entry.

The offset for a member function is used to determine the proper slot in the vtable to be used when calling a virtual member function. The initialization of the vtable requires that we are able to determine the overriding behavior of the classes involved and, in the case of multiple inheritance, be able to set the offset field for each function entry to adjust the pointer to the instance appropriately.

In the process of doing object resolution we must also be able to determine the actual functions that will be called when invoking either a non-virtual member function or a static member function. In the case of a non-virtual member function and multiple inheritance we must also determine the offset associated with that function. In both of these cases we need to adjust any calls to those functions to invoke the correct function. In a similar fashion we need to resolve references to static member variables.

Proper function resolution must also be done for potentially generated functions like constructors, destructors, and operator =, as well as for operator new and operator delete. Further the code added to constructors and destructors must be ready to handle an arbitrary set of base classes (both non-virtual and virtual) since we will not know *a priori* a class's set of base classes.

Another problem that must be solved is the allocation of global, static or automatic class instances. Since the size of the class is not known until at least link time the allocation of those instances must be delayed. For global and static instance this is not a major problem, whereas automatic instance are. Changes in the size of an automatic instance will potentially change the offset for other, perhaps non-class, variables that are to be allocated on the stack.

Runtime ΔC++

The prototype runtime version of ΔC++ provides a solution to the above problems, but with a substantial performance penalty (about a factor of 2 in both size and speed). We built this prototype in an attempt to understand the problems that must be addressed and also to understand how we should build a link time solution to these problems. It is not intended to be used as a real solution to the problems being addressed in this paper. By describing some of the code translations from C++ to C that we used, we hope to provide a better understanding of the problems that must be addressed when adding dynamics to the C++ environment..

Runtime ΔC++ utilizes a set of *offset* variables (one per member) to resolve references to class members. These variables are initialized as part of the process that resolves class definitions which takes place as part of the initialization phase before a C++ application really executes. In the following example:

```
class Alpha {
    public
        long a;
        long A();
        virtual long B();
};

Alpha *pa;
```

the code generated for the expression `pa->a` looks like:

```
*((long *) (((char *) pa) + __mtable_Alpha[__a__Alpha.vo])).
```

The vector `__mtable_Alpha` contains the offsets that are needed to access members of a class and the variable `__a__Alpha.vo` indicates the slot in that table that will holds the offset value for the member `a`. In this case `__mtable_Alpha[__a__Alpha.vo]` is initialized to be the number of bytes that the member `a` is away from the start of the class `Alpha(0)`. It may seem that the use of `__mtable_Alpha` is unnecessary, however it is needed if we wish to allow for the overriding of member data. For example, if the class `Gamma` is derived from `Alpha` and we wish to allow `Gamma` to be redefined at a later time to also have a member `long a`, and have references of the form `pg->a` where `pg` is a pointer to a `Gamma` now refer to the `a` in `Gamma` instead of in `Alpha`, then using the vector `__mtable_Alpha` is necessary. If we do not want to allow overriding of member data then the expression `pa->a` would result in the following code:

```
*((long *) (((char *) pa) + __a__Alpha.offset))
```

where `__a__Alpha.offset` would contain the number of bytes that the member `a` is away from the start of the class `Alpha`.

The expression `pa->A()` generates:

```
(*((long (*)(struct Alpha *)) (__vtable_Alpha[__A__Alpha.mo].f))
  (((struct Alpha *)(((char *) pa)
    + __A__Alpha.so
    + __vtable_Alpha[__A__Alpha.mo].d)))
```

and the expression `pa->B()` generates:

```
(*((long (*)(struct Alpha *)) ((*((struct __mptr **)
  (((char *) pa) + __vptr__Alpha))[__B__Alpha.mo].f)
    (((struct Alpha *)(((char *) pa) + __B__Alpha.so
      + (*((struct __mptr **) (((char *) pa)
      + __vptr__Alpha)))[__B__Alpha.mo].d))).
```

The scheme for calling virtual and non-virtual member functions is quite similar. In each case we are choosing to do the function lookup through a method table. In the non-virtual case we are doing a lookup through the method table that is associated with the declared class of the calling object. In the virtual member function case we are using the method table attached to the calling object. In each case the function that is going to be called is determined by a variable, `__A__Alpha.mo` and `__B__Alpha.mo`, respectively. These variables give the offset into the method table that holds the appropriate function and the value to add to the variable `pa` to handle the case of multiple inheritance. The variables `__A__Alpha.so` and `__B__Alpha.so`, which would be set to 0 in this example, are needed to handle changes in the class structure that introduce multiple base classes. The variable `__vptr__Alpha` gives the offset (in bytes) to the vptr for the class `Alpha`.

The above code may seem to be overly complex but each field is necessary in order to handle the modifications described above. For example the above code must continue to work if the definition of Alpha is changed to:

```
class Alpha {
    public:
        long al;
        virtual long C();
        long a;
        long A();
        virtual long B();
};
```

In this case the value assigned to `__a__Alpha.vo` would change from `0` to `1`, the value of `__mtable[__a__Alpha.vo]` would change from 0 to 4, the variable `__A__Alpha.mo` would change from 0 to 1, and the variable `__B__Alpha.mo` would change from 1 to 2. With these changes the above code fragments will continue to function correctly.

These fragments will continue to work even if we choose to further modify our example so that the definition of Alpha looks like:

```
class Aleph {
    public:
        long al;
        virtual long C();
};
```

```
class Beth {
    public:
        long bt;
        long a;
        long A();
        virtual long D();
        virtual long B();
};
```

```
class Alpha : public Aleph , public Beth {
    public:
        long B();
};
```

In this case the value of `__a__Alpha.vo` would change to 3 and the value `__mtable_Alpha[__a__Alpha.vo]` would change to 12 (the 2nd slot of `__mtable_Alpha` would hold the location of the vptr which is located 4 bytes into the instance). The value of `__A__Alpha.so` would change to 8, and the function stored in `__vtable_Alpha[__A__Alpha.mo].f` will be changed from `Alpha::A` to `Beth::A`. The change to the value `__A__Alpha.so` makes sure that the function Beth::A will be called with the correct value for `this`. The value of the variables `__B__Alpha.mo` and `__B__Alpha.so will be` changed to 3 and 8 and the value of `__vtable__Alpha[__B__Alpha.mo].d` will be changed to -8. Thus, if `pa` is really of type `Alpha` then the code `pa->B()` will invoke the function `Alpha::B` with `pa` passed as the first parameter,

So far we have shown how we can handle, for non-static members, *member-extension*, *class-extension*, and *member-promotion*. *Member-reordering* and *class-reordering* are simple, since the actual determination of a member's location is always done via a variable. Changing the position of a member is handled by changing the value of the appropriate offset variables and tables. The same is true for changing the order of the base classes.

The above code fragments also handle *override-changing*. If we change the last definition of `Alpha` so that it has the following definition:

```
class Alpha : public Aleph , public Beth {
    public:
        long a;
        long A();
        long B();
};
```

The value of `__a__Alpha.vo` remains the same, however the value of `__mtable_Alpha[__a__Alpha.vo]` will change to 20. Similarly the values of __vtable_Alpha[`__A__Alpha.mo].f` and `__vtable_Alpha[A__Alpha.mo].d` will change to `Alpha::A` and -8, respectively.

In the previous examples we have examined the code that must be generated for non-static members of a class. It must also be possible to handle the same types of modifications to the static portion of a class definition. If we extend the class Alpha to contain two static members, `static long s` and `static long S()`, then the code generated for `pa->s` is:

```
(*((long *) __stable__Alpha[__sa__Alpha.vo]))
```

where `__stable_Alpha` is a vector that contains pointers to the static variables for the class `Alpha`. Thus , in this example `__s__Alpha.vo` would be `0` and `__stable_Alpha[0]` contains &`Alpha::s`. Similarly the expression `pa->S()` generates the following code:

```
((*(((long (*)(void ))(__vtable_Alpha[__S__Alpha.mo]).f))))( )
```

The use of the vectors, `__stable__Alpha` and `__vtable__Alpha`, and the offset variables, `__s__Alpha.vo` and `__S_Alpha.mo`, provide the level of indirection that is necessary to handle the required modifications. If we promote the member `s` into the class `Aleph`, the value of the variable `__s__Alpha.vo` may change (depending on whether we have added other static member variables), but the value of `__stable__Alpha[__sa__Alpha.vo]` will change to &`Aleph::s`.

The above examples give a slightly misleading picture of the code that is generated for the above class definitions. The code fragments listed above would only have been generated if the code was compiled using the original definition of `Alpha`. If we recompile the same code after we added the base classes of `Aleph` and `Beth`, the original code fragments `pa->a`, `pa->A()`, and `pa->B()` change to the following:

```
*((long *) (((char *) pa)
    + __mtable_Alpha[__Beth__Alpha.vo + __a__Beth.vo])).

(*((long (*)(struct Alpha *))
  (__vtable_Alpha[__Beth__Alpha.mo + __A__Beth.mo].f))
    (((struct Alpha *)(((char *) pa)
      + __Beth__Alpha.offset
      + __A__Beth.so
      + __vtable_Alpha[__Beth__Alpha.mo + __A__Beth.mo].d)))
```

```
(*((long (*)(struct Alpha *)) ((*((struct __mptr **)
  (((char *) pa) + __vptr__Alpha))[__Beth__Alpha.mo + __B__Beth.mo].f)
    (((struct Alpha *)(((char *) pa)
      + __Beth__Alpha.offset + __B__Beth.so
      + (*((struct __mptr **) (((char *) pa)
          + __vptr__Alpha)))[__Beth__Alpha.mo + __B__Beth.mo].d))).
```

where the values of the variables are as follows:

```
__Beth__Alpha.vo = 1
__a__Beth.vo = 1
__mtable_Alpha[2] = 12
__Beth__Alpha.mo = 1
__A__Beth.mo = 0
__vtable_Alpha[1].f = Beth::A
__Beth__Alpha.offset = 8
__vtable_Alpha[1].d = 0
__B__Beth.mo = 2
__vptr__Alpha = 4
__vtable_Alpha[3].f = Alpha::B
__vtable_Alpha[3].d = -8.
```

In this example we have introduced another structure `__Beth__Alpha` which contains information that is used to move up the class hierarchy from `Alpha` to `Beth`. The `offset` field provides the number of bytes that must be added to an instance of an `Alpha` to change it into a `Beta`. The `vo` and `mo` fields are used to allow extensions to a base class to be made without recompiling any derived class. We want to be able to add a new member to the class `Beth` and, without recompiling the code for `Alpha`, and be able to reference that new member from an instance of an `Alpha`. When we add a member to a class, we require that the code that provides the offset variables for that class be recompiled. This guarantees that the appropriate offset variables will be defined. This is not true for derived classes, which need not be recompiled when a base class changes. For example, if we add the member `long Bt()` to the class `Beth` then the structure `__Bt__Beth` will exists and be properly initialized, whereas, until `Alpha` is recompiled, the structure `__Bt__Alpha` won't exist. The `vo` and `so` fields of the `__Beta__Alpha` structure make it possible to access the proper parts of the `__mtable_Alpha` and `__vtable__Alpha` tables.

The need to handle this kind of change also explains why we must reference all members through a function table. If we could assume that the structure `__Bt__Alpha` exists whenever `Alpha` contains the member `long Bt()`, even if it is inherited from a base class, than a call to a non-virtual member function could look like:

```
(*((long (*)(struct Alpha *)) (__A__Alpha.mo.f))
  (((struct Alpha *)(((char *) pa)
    + __A__Alpha.so
    + __A__Alpha.d)))
```

Unfortunately the above assumption can not be made, thus dictating the use of the function table.

C++ allows a programmer the ability to directly reference a member of a class by providing its qualified name. Thus we need to be able to handle expressions of the form `pa->Beth::a` and `pa->Aleph::C()`. Each of these cases result in code that first casts the pointer `pa` to the qualified class and then uses the code given above for instance variables and non-virtual member functions. The generated code for casting an `Alpha` to a `Beth` looks like:

```
((struct Beth *) (((char *) pa) + __Beth__Alpha.offset))
```

and the code generated for the expressions `pa->Beth::a` and looks `pa->Aleph::C()` looks like:

```
*((long *) (((char *) pa) + __Beth__Alpha.offset
    + __mtable_Beth[__a__Beth.vo]))

(*((long (*)(struct Beth *)) (__vtable_Beth[__C__Beth.mo].f))
  (((struct Alpha *)(((char *) pa)
    + __Beth__Alpha.offset
    + __C__Beth.so
    + __vtable_Beth[__C__Beth.mo].d))).
```

Virtual base classes further complicate this picture in that it introduces another level of indirection. The major problem with virtual base classes arises when a derived class overrides either a non-virtual member function or a member variable. Since the location of the virtual base class instance relative to the top of the class instance can only be computed dynamically, there is no easy way to get the `this` to point to the proper place. If we extend our example so that both `Aleph` and `Beth` derive virtually from the class `Gimmel`, which contains an non-virtual member function `long G()`then calling the function:

```
long test(Aleph *pal)
{
    return pa->G();
}
```

should result in calling the function `Gimmel::G` with a pointer to a `Gimmel`. If in a later release `Aleph` overrides `G()` then the call will have to result in a call to `Aleph::G` with a pointer to an `Aleph`. The code generated for `test` must handle both of these cases. The way the runtime version of ΔC++ handles this problem is by generating code that casts `pal` to a `Gimmel` and then calling the appropriate method. Unfortunately there is not a fixed offset that should be added to the casted version of `pa` in order to convert it back into an `Aleph`. The offset that is used when passing in an `Aleph` is different than the offset used when passing in an `Alpha`. Determining the appropriate offset, requires that we select the offset using the type of the object passed in as a parameter. The type of an instance is stored in the first entry in vtable attached to the object (this does require that every instance have a vtable pointer).

Future Work

As we stated earlier, the runtime version of ΔC++ is not an adequate solution to the problems of supporting shared libraries and dynamic loading in C++ as the performance overhead is too great. No developer will accept a factor of 2 decrease in performance in order to solve these problems. We are developing a compile/linktime solution to this problem that provides the above functionality without any major performance penalty. The basic concept is to replace the multitude of variables used in the runtime solution with a set of relocation types in a linktime solution. Efficient code for member access will be generated, and modified at link time with the proper offsets. The same work with respect to the generation and initialization of vtables will also be done by the linker.

We have also started to look at how we support templates in this same environment. The mechanisms outlined in this paper solve one of the problems associated with templates. Using ΔC++, a developer will be able to change the template definition in exactly the same ways as normal classes. The problem with templates that still needs to be addressed is how and when should the template be instantiated. Using the current *cfront 3.0* solution, where instantiation is done when the application is linked together, is not viable. With shared libraries, linking is done when the user runs an application. Since the user may not have access to the compiler nor would the user want to wait for the compiler to run, generating code at that time is not feasible.

Other Uses of ΔC++ Technology

We started working on ΔC++ in order to solve the problems of shared libraries and dynamic loading. Duing the development of the prototype we started to realize that there are other places where this technology can be used. We have looked at using this technology to solve several problems in the application development process. Using ΔC++, developers need not recompile large portions of their application whenever they change (or someone else changes) a class definition. When a class definition is changed, only the code that implements that class, that uses the new parts of the definition, or that used parts of the definition that has been removed need to be recompiled. This should greatly reduce the edit/compile/debug cycle. In a similar fashion ΔC++ will allow base class developers the ability to modify class definitions and test their changes against already existing code. With current C++ systems, the cost of changing a class definition high up in the hierarchy is often prohibitive. In order to verify that the change does not cause any problems all the code that uses that base class must be recompiled. In developing the Andrew Toolkit, which had a similar compile time model as C++, we discovered that developers eventually refused to make changes in important base classes, since the cost of testing was so high. The result of this that developers would work around problems in the most basic parts of the system that should have been fixed. With ΔC++ the developer can change the class definition, recompile a few files and relink the resulting applications to test the change. This fundamentally changes the cost of making basic changes in the system.

Another use of ΔC++ is delayed binding of class implementations. ΔC++ allows us to completely separate interface and implementation thus solving the problem discussed in [3] without creating lots of additional classes nor using multiple inheritance. Using either shared libraries or dynamic loading a user can choose which implementation of a class should be used when running an application. The choice can include using implementations that were not released with the application but provided later by another developer. For example, a user could select the presentation style to be used by an application by pointing the application at different libraries, each which supports the proper interfaces but with different class interfaces and implementations. In this way an application could be shipped with a Motif look and feel and someone else could provide an OpenLook look and feel at a later time. Further subclassing with the application code from those components would still work.

Using ΔC++ also allows library providers the option of eliminating the private portion of a class interface when releasing their product. The code compiled with an abridged version of a class definition will always work with an implementation compiled with a complete version.

Related Work

The issue of dynamic loading in C++ has been addressed in previous papers [4, 5], but ignored the issues raised in this paper. We believe that the combination of ΔC++ and the ideas presented in these papers can lead to a usable dynamic loading system.

Environments such as Lisp and Smalltalk that support the problems described in this paper have existed for a long time. This work differs from these systems in that the ultimate goal is to develop an environment that supports the fuill C++ language specification [6] with little performance degradation over standard C++ systems.

Conclusion

Current C++ systems have been designed with the assumption that the end-user of an application has access to both the sources of the application (and possibly its supporting libraries) and to a C++ compiler. Current systems also have been designed with the assumption that the cost of running the compiler is essentially zero. While it may be argued that these assumptions were valid in the past, they are no longer valid today. In the today's world, library and application developers have no desire to release source code for their product. Even if they did end-users probably have not purchased the compilation environment nor would they want to run the compiler if it was bundled on their system. We must develop software that can be purchased, installed and run with limited overhead. Our work with ΔC++ is an initial attempt to look at these problems. We have attempted to solve what we believe to be the biggest problem, the changing of class definitions with limited recompilation. We already know that template instantiation is another problem area we need to investigate. In the future any proposed extension to C++ must consider the problems raised by environments that make use of shared libraries and dynamic loading.

Acknowledgments

I would like to thank David (Bean) Anderson and Mark Linton for allowing me to bend their ears about this work. This was especially helpful when trying to make sense of the complexities of C++ and cfront. I would also like to thank Miles Bader (a former colleague of mine when I worked at the Information Technology Center, Carnegie Mellon Univerity), who did some very early work at solving similar problems in the object system used in the Andrew Toolkit.

References

[1] Mark Sherman, David Anderson, Wilfred J. Hansen, Thomas P. Neuendorffer, Andrew J. Palay, and Zalman Stern, "Allocation of User-Interface Resources in the Andrew Toolkit". In *Proceedings of the International Conference on Multimedia Information Systems '91*, McGraw Hill, 1991, pages 261-272.

[2] NeXT System Reference Manual, NeXT Inc.

[3] Bruce Martin, "The Separation of Interface and Implementation in C++". In *Proceedings of the 1991 USENIX C++ Conference*, 1991, pages 51-62.

[4] Sean M. Dorwood, Ravi Sethi, and Jonathan Shopiro, "Adding New Code to a Running C++ Program". In *Proceedings of the 1990 USENIX C++ Conference*, 1990, pages 279-292.

[5] B. Stroustrup, "Possible Directions for C++", *1987 USENIX C++ Workshop*, 1987, pages 399-416

[6] M Ellis and B. Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, 1990.