# Lund C++ Seminars, June 13–14, 1991

Brück, Dag M.

1991

*Document Version:*
Publisher's PDF, also known as Version of record

**Department of Automatic Control**
**Lund Institute of Technology**
P.O. Box 118
S-221 00 Lund Sweden

*Document name*
INTERNAL REPORT

*Date of issue*
August 1991

*Document Number*
CODEN: LUTFD2/(TFRT-7479)/1–150/(1991)

*Author(s)*
Dag M. Brück (Editor)

*Supervisor*

*Sponsoring organisation*

*Title and subtitle*
Lund C++ Seminars — June 13–14, 1991

*Abstract*

The Lund C++ Seminars were given in connection with the ISO and ANSI C++ standardization meetings in Lund, Sweden. This report contains written documentation of most talks presented at the Lund C++ Seminars, June 13–14 1991.



*Key words*
Programming languages, C++

*Classification system and/or index terms (if any)*

*Supplementary bibliographical information*

*ISSN and key title*

*ISBN*

*Language*
English

*Number of pages*
150

*Recipient's notes*

*Security classification*

CODEN: LUTFD2/(TFRT-7479)/1–150/(1991)

# Lund C++ Seminars
# June 13–14, 1991

Dag M. Brück (Ed.)

Department of Automatic Control
Lund Institute of Technology
August 1991

# Contents




Lund C++ Sign

# Building Reusable Components in C++

*Michael J. Vilot*
ObjectWare, Inc
16 Warton Road, Nashua NH 03062-2537
mjv@objects.mv.com
(603)888-4729

The Lund C++ Seminars
Lund, Sweden
13 Jun 91



## N O T E S

This 60 minute presentation is necessarily limited.

Further details are available:

Booch, G., *Software Components with Ada*, Benjamin/Cummings, Menlo Park CA, 1987.

Booch, G., and M. Vilot, "The Design of the C++ Booch Components," OOPSLA/ECOOP '90 Conference, *SIGPLAN Notices*, vol 25(10), October 1990, p. 1.

Booch, G., "Using the OOD Notation," *Object Magazine*, vol 1(2), July/August 1991.

Booch, G., and M. Vilot, *Software Components in C++*, Addison Wesley, Reading MA, in preparation.

# Reusable Components

- Context
  - ➢ Programming Systems Products
    example: UNIX
  - ➢ Program Generators
    example: RCS/SCCS, CASE tools
  - ➢ Programs
    example: AWK, program editors
  - ➢ Libraries
    example: C-ISAM



## N O T E S

The term "programming systems product" is from:
Brooks, F., *The Mythical Man Month: Essays on Software Engineering*, Addison-Wesley, Reading MA, 1975, p. 4.

Programming Systems Products tend to be very large and complex. Because of the investment involved, they are usually exclusive — it is rare to find two or more operating systems running on the same machine.

Program Generators are not quite as large, but do represent an investment. The adoption of a CASE tool's "methodology" inhibits changing to another product.

Individual programs can be designed to be customizable and/or extensible. An example of this kind of program is the `emacs` editor.

Libraries tend to be specialized, and thus more easily replaced. The typical "subroutine library" does not export much in the way of data structures. The usual approach is to make them "opaque," using pointers as arguments to the functions.

**Reusable Components**

# Design Issues

- Focus & Content
  - ➢ Application (problem domain)
  - ➢ General-purpose (mechanism)
- Structure
  - ➢ Single "tree"
  - ➢ Individual classes
  - ➢ "Forest"
- Performance
  - ➢ Compile, Link, Run time
  - ➢ Library, Executable space



## N O T E S

Application-oriented component libraries often take the form of "application frameworks" and try to encourage a consistent approach by applications written for that problem space.

General-purpose component libraries try to be usable in many different problem contexts.

The choice of structure depends on the degree to which the library designer wants to enforce the use of certain mechanisms and policies:

- — The NIH class library uses a single "tree" structure.
- — The AT&T SLE library provides separate classes.
- — The GNU `libg++` library uses a "forest" approach.

Meiler Page-Jones uses the term "encumbrance" to refer to the various performance overhead aspects of using a particular library component.

**Design Issues**

# Concrete Data Types (CDTs)

- Characteristics
  - Self-contained
  - Not intended as a base class
- Design Goals
  - close match to a particular concept and implementation strategy
  - run-time and space efficiency comparable to "hand-crafted" code
  - minimal dependency on other classes
  - understandable and usable in isolation



N O T E S

[Stroustrup 91] § 13.2

The term "concrete data type" is meant to contrast with abstract data type. Examples of CDTs are: `date`, `string`, and `complex`.

**Design Issues**

# Node Classes

- Characteristics
  - ➢ Provides a foundation (as a base class)
  - ➢ Often part of a larger framework
- Design Goals
  - ➢ relies on bases classes for implementation and supplying services
  - ➢ uses virtual functions for most (all) operations
  - ➢ adds to services of base classes
  - ➢ can be a base class for further derivation
  - ➢ understandable only in context of base classes



N O T E S

[Stroustrup 91] § 13.4

Node classes are strongly coupled to the base classes they derive from.

Often, the protected interface is a key design aspect.

Examples of node classes include: `ios` and `streambuf` from the iostreams library, and `process` and `task` from the task library.

Node classes are essential elements of application frameworks, including user interface frameworks such as `MacApp` or `ET++`.

**Design Issues**

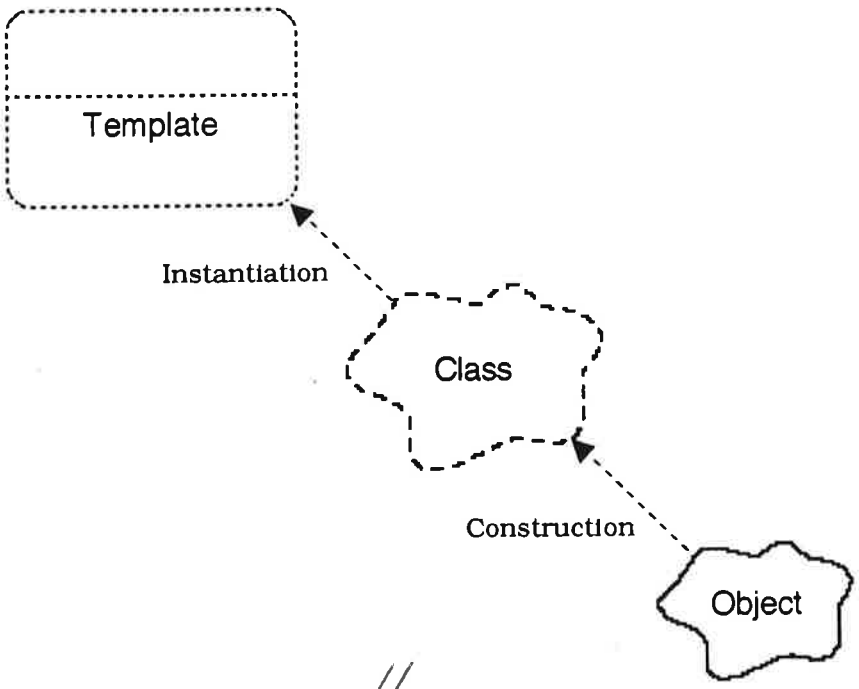
# Templates

- Name Spaces
  - ➢ Class templates are in a separate name space
  - ➢ Template model is macro-expansion
- Template Handling
  - ➢ Implementation should detect the need to expand declarations
  - ➢ Extra work needed to expand definitions
  - ➢ Transitive dependencies
  - ➢ Shared implementations



## N O T E S

Classes are instances of templates, similar to the way objects are instances of classes.




//

# Other Issues

- Technical
  - Library maintenance
  - Cataloging & Retrieval
- Managerial
  - Reuse incentives
  - Library builders vs. library users
- Social
  - Trust (correctness & performance)
  - Perceptions
- Legal
  - Ownership & intellectual rights
  - Liability



## N O T E S

Technical issues:

Keeping a large collection of components organized and usable is a significant challenge. We need better tools for cataloging and retrieving components.

Managerial issues:

Current software management practice does not encourage/incentivize reuse.

Social issues:

Current approaches to documenting components do not provide enough information to establish whether a given component will perform to expectations.

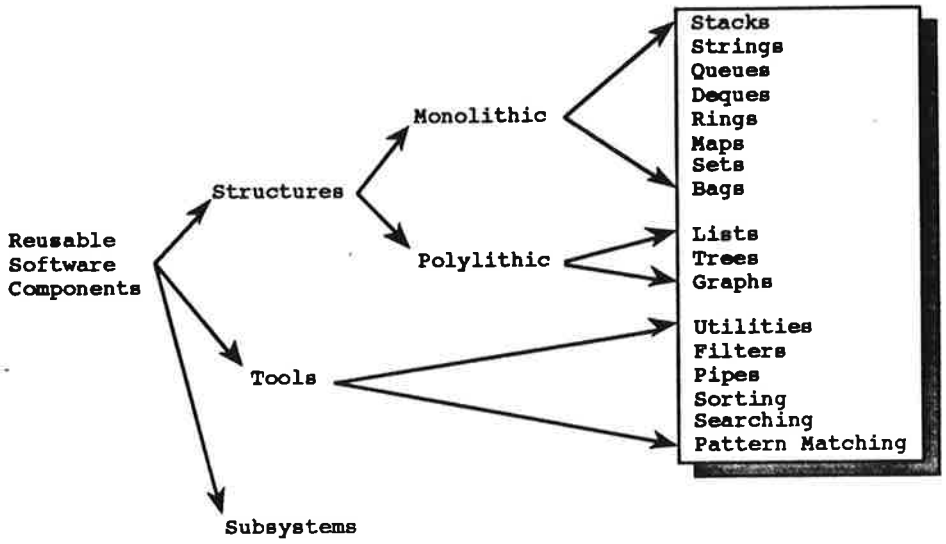
Perception of the effort to find and use a component is usually **over** estimated, while the perception of the effort to build from scratch is usually **under** estimated.

Legal issues:

The legal system is moving to adapt to the needs of software. National and international laws are clarifying some aspects. It is a slow process.

**Overview**

# Structures and Tools






## N O T E S

Grady described this taxonomy in his '87 text. It represents the core set of abstractions embodied in the library.

As you can see, there are not a lot of them. What surprised me was the fact that it took over 100,000 lines of Ada code to implement them! Of course, the number of lines was a direct result of providing the various time/space forms of each component.

The distinction between monolithic and polylithic involves the notion of *structural sharing*. Basically, you can manipulate sub-lists, sub-trees, and sub-graphs recursively. All other components are handled as an indivisible unit.

# C++ Library Design

## *Key Abstractions*

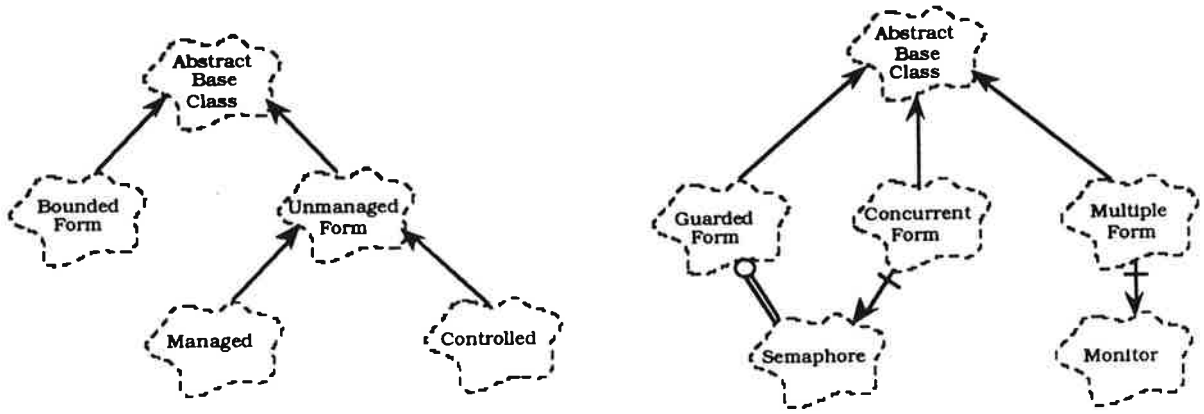
- Design Decisions
  - Inheritance
  - Type parameterization
  - Exceptions
- Core Data Structures
  - Iterators
- Utilities & Tools
- Support
  - Factor out storage & concurrency



N O T E S

**Key Abstractions**

# Core Data Structure Classes






## N O T E S

This slide, another OOD class diagram, illustrates the heart of the design.

The hierarchy on the left represents the storage forms, and the hierarchy on the right represents the concurrency forms.

There is no inherent superior/subordinate relationship between the two hierarchies.

Deriving the concurrent forms from the "concrete" classes had two benefits:

- their implementation was trivial
- they can be packaged as an optional layer

Our use of the term "monitor" involves a critical region with more than one entry — in this case, one for reading and one for writing. This use of the term should not be surprising, see:

Holt, R., G. Graham, E. Lazowska, and M. Scott, *Structured Concurrent Programming with Operating Systems Applications*, Addison-Wesley, Reading MA, 1978.

Hoare, C.A.R., " Monitors: an operating system structuring concept," *Communications of the ACM*, vol 17(10), October 1974, pp. 549–557.

# Internal Design

## *Mechanisms in C++*

- Inheritance
  - Abstract Base Classes
  - Public & private derivation
  - Use of mixin classes
- Support
  - Iterators
  - Exceptions
- Templates
  - Template classes
  - Template expansion filter



N O T E S

# Abstract Base Class

- Establishes interface
  - Pure virtual functions
- Kinds of public members:
  - Constructive/destructive
  - Modifiers
  - Selectors
  - Private
- Cache state
  - Changes some *O(n)* computations to *O(1)*
- Express relationships among abstract types



N O T E S

```
template<class Item>
class Queue {
public:
  Queue() : rep_length(0) {}
  Queue(const Queue<Item>& s) : rep_length(s.rep_length) {}
  Queue<Item>& operator=(const Queue<Item>&);
  virtual ~Queue() {}

  // Modifiers
  virtual Queue<Item>& clear();
  virtual Queue<Item>& add(Item);
  virtual Queue<Item>& pop();

  // Selectors
  int operator ==(const Queue<Item>&) const;
  int operator !=(const Queue<Item>&) const;
  virtual unsigned length() const;
  virtual int      empty()  const;
  virtual Item     front()  const =0;

protected:
  unsigned rep_length;
};
```

**Mechanisms**

# Managed and Controlled

- Derived from Unmanaged class
- Allocate different types of Nodes
  - Managed, Controlled mixins
  - Managed_Node, Controlled_Node classes
- Provides storage managers for operators new and delete
  - Class Storage_Manager
  - One per template instance (static)



N O T E S

```
#include "unmanaged_queue.h"

template<class Item>
class Managed_Queue : public Unmanaged_Queue<Item>
{
public:
  Managed_Queue() {}
  Managed_Queue(const Unmanaged_Queue<Item>& q)
    : Unmanaged_Queue<Item>(q) {}
  // Generated constructors and assignment
  // inherit Modifiers
  // inherit Selectors
protected:
  Node<Item>* allocate(Item);
};
```

```
//  managed_queue.cp
#include "managed_queue.h"
#include "managed_storage.h"    // static Storage_Manager

template<class Item>
Node<Item>* Managed_Queue<Item>::allocate(Item item)
{
  return new Managed_Node<Item>(item);
}
```

**Mechanisms**

# Utility & Tool Mixins

- Interconnection mechanism
  - ➢ Declare, use pure virtual
- Requires derivation to use
  - ➢ Derived class provides virtual's definition
  - ➢ Usually multiple inheritance
- Alternative:
  - ➢ Template argument(s)



N O T E S

```
// sort.h

template<class Item>
class Quick_Sort {
protected:
virtual  Item&    elem(unsigned i) const =0;
public:
  sort(unsigned size);

private:
  void exchange (unsigned left, unsigned right);
  sort_recursive(unsigned left, unsigned right);
};
```

```
// client.h

template<class Item>
class Sort_Vec : public Vector<Item>, public Quick_Sort<Item> {
public:
  Sort_Vec(unsigned size) : Vector<Item>(size) {}
  Sort_Vec(const Vector<Item>& v) : Vector<Item>(v) {}
protected:
  Item& elem(unsigned i) const { return Vector<Item>::elem(i); }
};
```

**Mechanisms**

# Exceptions

- Anticipated language support
  - ➢ Delay over resumption
  - ➢ November X3J16 decision
- Hierarchy of exception classes
  - ➢ Natural classification
  - ➢ Adds to global name space
- Simple function call `_catch()`
  - ➢ Also `terminate()` and `set_terminate()`



N O T E S

We do not provide an exception handling mechanism.

The source code uses *throw-expression* syntax, and we compile with a simple preprocessing trick:

```
-Dthrow=_catch
```

to turn them all into calls to our library function:

```
extern void _catch(const Exception&);
```

The base `Exception` class provides some simple information:

```
class Exception {
  friend ostream& operator<<(ostream&, const Exception&);
public:
  Exception(const char* name, const char* who, const char* what);
  virtual ~Exception() {}

  virtual void  display(ostream&) const;
  virtual const char* name() const;
  virtual const char* who()  const;
  virtual const char* what() const;
...
};
```

It uses a fixed-length representation for the strings, so it won't run out of heap trying to throw an exception.

**Conclusions**

# Project Results

- Code sharing effects
  - Ada: 150,000+ NCSL
  - C++: < 20,000
  - Unusual, due to repetitive structure
- Element count:
  - Ada: 501 packages
  - C++: 380 classes
  - No Discrete types in C++
  - No fixed/float distinction in C++



N O T E S

Some causes for the code reduction:

- Eliminating half the data structures components, due to iterators.
- Factoring out the support classes.
- Eliminating duplication by factoring common functions into base classes and inheriting them.
- Eliminating duplication in the Managed and Controlled forms with the `allocate()` "virtual constructor" function.*
- Eliminating duplication and manual exception handling in the concurrent forms with the `Lock` class.
- No tasking.
- No discrete components.
- No fixed-point math types.

* Jordan, D., "Class Derivation and Emulation of Virtual Constructors," *C++ Report*, vol 1(8), September 1989, p. 1.

**Conclusions**

# Observations on C++

- ✧ Public derivation describes interface (protocol)
- ✧ Private derivation & data members describe representation
  - ➢ Might explore generalizing template arguments
- ✧ Reference semantics & template arguments
- ✧ Inline function definitions & nested classes reduce template encumbrance
- ✧ Template environment deliberately underspecified
  - ➢ Implementations will have to be creative at handling instantiations
  - ➢ Context-dependent expansion
  - ➢ Static objects & static members



N O T E S

We did this:

```
template < class T >
class Unmanaged_X : public X<T>, private List<T> { … };
```

Using private derivation makes adding new representation classes awkward.

We would have liked to have done this:

```
template < class T, class List<T> = our_List<T> >
class Unmanaged_X : public X<T> {
  List<T> rep_list;
public:
...
};
```

This can't quite be simulated at the instance level without introducing globals:

```
template < class T >
class Unmanaged_X : public X<T> {
  List<T>& rep_list;
public:
  Unmanaged_X(List<T>& list = _______ );
...
};
```

Not having a default obligates clients to create a `List` object for every instance.

# Templates as interfaces

*Andrew Koenig*

AT&T Bell Laboratories
184 Liberty Corner Road; Warren NJ 07059; ark@europa.att.com

### Introduction

Consider an on-line student registration system. Part of such a system will have to deal with the fact that its purpose is to register students; another part will be there because it is an online system; still other code will talk to the underlying database, and so on.

It would be nice if the parts of the system that deal with conceptually different characteristics are themselves kept carefully separate. That, for example, would make it possible to use the online and database parts of the system in building, say, an online library circulation system.

To understand how the parts of such a system might be kept separate, we will write a tiny program and separate it into pieces. When studing those pieces, think about how the techniques used there might apply to larger programs as well.

### The first example

Here is a simple function that adds the elements of an array of integers:

```
int sum(int* p, int n)
{
        int result = 0;
        for (int i = 0; i < n; i++)
                result += p[i];
        return result;
}
```

Here is an example of how to use this function:

```
#include <stream.h>

main()
{
        int x[10];
        for (int i = 0; i < 10; i++)
                x[i] = i;
        cout << sum(x, 10) << "\n";
        return 0;
}
```

This example prints `45`, which is the sum of the non-negative integers less than 10.

The `sum` function "knows about" three things:

- it is adding a bunch of things together;
- the things it is adding are integers; and
- the integers it is adding are stored in a particular way.

Let's see how we can divide up the program so that each one of these characteristics is contained in a different part.

### Separating the iteration

The first thing we will remove is the knowledge that elements being added are contained in an array. To do this, we follow the standard C++ technique: make it into a class.

What kind of class might encapsulate the notion of iterating over a collection of integers? An object of that class would evidently have to represent the state of the iteration, so the relevant operations are:

- a constructor, which establishes the data to be handled;
- a way of asking for the next element in sequence;
- a way of telling when the iteration is complete; and
- an assignment operator, copy constructor, and destructor to enable one of these objects to be used as a value.

Such objects are often used in C++ class libraries; they are ususally called *iterators*. There are many kinds of iterators possible; what we have just seen is a fairly fundamental outline of how they work.

Let's write down what we know so far about our iterator:

```
class Int_iterator {
public:
      Int_iterator(int*, int);
      int valid() const;
      int next();
      Int_iterator(const Int_iterator&);
      Int_iterator& operator=(const Int_iterator&);
      ~Int_iterator();
};
```

This class is called `Int_iterator` rather than `Iterator` because indeed it expresses an iteration through a sequence of `int` values. We will generalize it later to values of arbitrary type.

The constructor takes two arguments: the address of the initial element of the sequence and the number of elements. The `valid` member returns a nonzero value if there are still elements left in the sequence; the `next` member fetches the next element in the sequence if there is one. We will assume that `next` is not called unless there is known to be a value remaining in the sequence.

Before we fill in the definition of our iterator, let's rewrite the `sum` function:

```
int sum(Int_iterator ir)
{
      int result = 0;
      while (ir.valid())
            result += ir.next();
      return result;
}
```

Notice that this function does indeed use the `Int_iterator` copy constructor and destructor because its formal parameter is an `Int_iterator` rather than a `const Int_iterator&`. Because an iterator contains a state, iteration necessarily changes the iterator's value. Thus making the parameter a `const` is out of the question. Moreover, if the parameter were just an `Int_iterator&`, that would restrict the arguments to being lvalues, because only an lvalue argument can be passed to a non-constant reference parameter. The `sum` function does not use the iterator assignment operator.

A main program that uses this `sum` function might look like this:

```
#include <stream.h>

main()
{
        int x[10];
        for (int i = 0; i < 10; i++)
                x[i] = i;
        cout << sum(Int_iterator(x, 10)) << "\n";
        return 0;
}
```

The only difference is that instead of calling `sum(x, 10)` it is now necessary to call `sum(Int_iterator(x, 10))`. Of course, it is possible to write an overloaded `sum` function that preserves the original interface:

```
inline int sum(int* p, int n)
{
        return sum(Int_iterator(p, n));
}
```

It is time to return to the definition of the `Int_iterator` class. What information is actually necessary in an object of that class? We need to be able to locate the next element in the sequence, presumably through an `int*`, and we need to know when the sequence is finished. This latter information could be obtained by remembering either how many elements are left or the address that is one past the last element.* If we arbitrarily choose to remember the count, the private members of the `Int_iterator` class then look like this:

```
private:
        int* data;
        int len;
```

The constructor is straightforward. All it has to do is to initialize the `data` and `len` members from its corresponding parameters:

```
        Int_iterator(int* p, int c):
                data(p), len(c) { }
```

The destructor, assignment operator, and copy constructor are even easier. Because their actions are equivalent to the corresponding actions on the data members, these three member functions can be omitted altogether. The compiler will automatically insert appropriate definitions for them.

Finally, we must define `valid` and `next`. After doing so, we arrive at this:

---

* I strongly prefer storing a pointer to one past the last element rather than storing a pointer to the last element itself, because if the sequence has no elements at all, there is no last element to point to. For more detail about this style of counting, see my book *C Traps and Pitfalls*.

```
class Int_iterator {
public:
      Int_iterator(int* p, int c): data(p), len(c) { }
      int valid() const {
            return len > 0;
      }
      int next() {
            --len;
            return *data++;
      }
private:
      int* data;
      int len;
};
```

**Iterating over arbitrary types**

The name `Int_iterator` is a dead giveaway that we have missed an opportunity for abstraction. We can readily turn the `Int_iterator` class into a general `Iterator` template:

```
template<class T> class Iterator {
public:
      Iterator(T* p, int c):
            data(p), len(c) { }
      int valid() const {
            return len > 0;
      }
      T next() {
            --len;
            return *data++;
      }
private:
      T* data;
      int len;
};
```

The only thing the least bit difficult about this is figuring out which of instances of `int` in the `Int_iterator` class should become `T` and which should remain `int`.

The rest of our program can remain completely unchanged if we say that the type `Int_iterator` is now equivalent to `Iterator<int>`:

```
typedef Iterator<int> Int_iterator;
```

Alternatively, we can change the rest of the program by simply substituting `Iterator<int>` everywhere `Int_iterator` formerly appeared:

```
int
sum(Iterator<int> ir)
{
        int result = 0;
        while (ir.valid())
                result += ir.next();
        return result;
}

main()
{
        int x[10];
        for (int i = 0; i < 10; i++)
                x[i] = i;
        cout << sum(Iterator<int>(x, 10)) << "\n";
        return 0;
}
```

**Adding other types**

If we can iterate over collections of arbitrary type, it would be nice to be able to find the sum of values of arbitrary type as well. This is easily done by making the `sum` function into a template:

```
template<class T>
T sum(Iterator<T> ir)
{
        T result = 0;
        while (ir.valid())
                result += ir.next();
        return result;
}
```

This now makes it possible to calculate the sum of arrays of objects of other classes. What classes? Any class for which

- it is possible to convert `0` to an object of that class;
- The += operator is defined on objects of that class; and
- The objects have value-like semantics to that the `sum` function can return them as its value.

All the numeric types meet this requirement, of course, and it is easy to define other classes that do as well.

**Abstracting the storage technique**

We have separated out two of the three kinds of knowledge from our little program. What about the third?

We are looking for different kinds of iterators to reflect different data structures. So far we have only one iterator, which lets us get at things stored in arrays. But what if our values are in a linked list? What if they are on a file? Is it possible to abstract those as well?

The "standard" way of solving this problem is to turn the `Iterator` class into an abstract base class that can represent any one of a number of different iterator classes:

```
template<class T> class Iterator {
public:
        virtual int valid() const = 0;
        virtual T next() = 0;
        virtual ~Iterator() { }
};
```

Next we say that an `Array_iterator<T>` is a kind of `Iterator<T>`:

```
template<class T> class Array_iterator: public Iterator<T> {
public:
        Array_iterator(T* p, int c): data(p), len(c) { }
        int valid() const {
                return len > 0;
        }
        T next() {
                --len;
                return *data++;
        }
private:
        T* data;
        int len;
};
```

Finally, we define the `sum` function to take a *reference* to an `Iterator` as its argument, to allow dynamic binding:

```
template<class T> T sum(Iterator<T>& ir)
{
        T result = 0;
        while (ir.valid())
                result += ir.next();
        return result;
}
```

Then, to add the elements of an array, we merely create an appropriate `Array_iterator` to describe it and pass that to `sum`:

```
#include <stream.h>

main()
{
        int x[10];
        for (int i = 0; i < 10; i++)
                x[i] = i;
        Array_iterator<int> it (x, 10);
        cout << sum(it) << "\n";
        return 0;
}
```

The difference here is that we have explicitly created an `Array_iterator<int>` object named `it` to express the iteration over the elements of `x`. As mentioned before, we cannot simply say

```
        cout << sum(Array_iterator<int>(x, 10)) << "\n";
```

because the subexpression `Array_iterator<int>(x, 10)` is not an lvalue and therefore cannot have a non-constant reference attached to it.

That is a minor inconvenience. Another, somewhat less minor, inconvenience is that because dynamic binding is used, every iteration of the inner loop in the `sum` function requires a virtual

function call. This could be expensive compared to the inline expansion of our previous examples, especially if the objects being added are simple things like integers.

We will deal with both of those objections by dispensing with dynamic binding. That means that when we want to add the elements of some collection we will have to know at compile time what kind of collection it is, but this is probably not a major hardship.

Evidently the way to do this is to make the `sum` function take *two* type parameters: the type of the iterator and the type of the objects being added. Unfortunately, this runs into a problem. If we try to define the `sum` function in the obvious way:

```
template<class Iter, class Obj> Obj sum(Iter it)
{ /* ... */ }
```

we find that we have defined a function whose return type is unrelated to its argument type. That is, the type of `sum(x)` is independent of the type of `x`. This is illegal in C++, because otherwise there would be no way to determine the type of an expression without examining a potentially unbounded amount of context.

The way out of this corner is again to follow the standard technique: make it a class! In other words, instead of defining a `sum` function, define a `Sum` class whose purpose is to add values expressed to it as iterators. It is easier to show the class itself than to describe it:

```
template<class S, class It> class Sum {
public:
        Sum(It iter): ir(iter) { }
        operator S();
private:
        It ir;
};
```

We create a `Sum` object by telling it we will use it to add objects of type `S` and handing it an iterator of an appropriate, but otherwise unknown, type. We cause the `Sum` object to perform its calculation by *converting* it to type `S`. That calculation will involve using the iterator to step through the collection. To ensure that it can be done more than once, we will copy the iterator to preserve the original:

```
template<class S, class It> Sum<S, It>::operator S() {
        It i = ir;
        S result = 0;
        while (i.valid())
                result += i.next();
        return result;
}
```

You can see that this `operator S()` is not much different from the `sum` function in the previous examples. Using it is not much different either:

```
#include <stream.h>

main()
{
        int x[10];
        for (int i = 0; i < 10; i++)
                x[i] = i;
        cout << Sum<int,iterator<int> >(iterator<int>(x, 10)) << "\n";
        return 0;
}
```

The complicated expression in the middle is easiest to understand if we read it from the inside out. First we construct `iterator<int>(x, 10)` to create an `iterator<int>` object that represents the ten elements of the array `x`. Next we construct a `Sum<int,iterator<int> >` object that on command will sum the elements of that array. Finally, by trying to print the `Sum<int,iterator<int> >` object, we implicitly do the actual addition.

It is true that this style requires a little extra coding. However, in exchange for that we obtain a lot of extra flexibility. For example, if we had said `Sum<double,iterator<int> >` that would have been enough to evoke double precision floating point addition instead of integer addition. Of course, it is still possible to abbreviate common cases:

```
template<class T> T sum (T* p, int n)
{
      return Sum<T,iterator<T> > (iterator<T>(p,n));
}
```

which makes this whole setup as easy to use as our first example:

```
main()
{
      int x[10];
      for (int i = 0; i < 10; i++)
            x[i] = i;
      cout << sum(x, 10) << "\n";
      return 0;
}
```

**The proof of the pudding**

To illustrate the flexibility of this approach, let's use the `Sum` class to add a potentially unbounded collection of numbers read from an *istream*.

First we need a class we can use as an iterator. Because it behaves so much differently from our previous `Iterator` class, we'll call it a `Reader`. In other words, a `Reader<T>` object will read a sequence of `T` objects from some `istream` in such a way that the `Sum` class can use it as an iterator.

This presents a small problem. The way we defined iterators, it is necessary to be able to test whether there are data left before reading it. That's not how `istreams` work, however; the way to tell if data is left in an `istream` is to read from it and see if it worked.

We will solve this problem by reading ahead one element in the `Reader` class and remembering whether it was successful. Because that must be done both in the constructor and in the `valid` member function, we'll make it a separate, private member function called `advance`:

```
template<class T> class Reader {
public:
        Reader(istream& is): i(is) { advance(); }
        int valid () const { return status; }
        T next() {
                T result = data;
                advance();
                return result;
        }
private:
        istream& i;
        int status;
        T data;
        void advance() {
                i >> data;
                status = i != 0;
        }
};
```

Each `Reader` object binds a reference to an `istream` that is given to its constructor. Thus `Reader<double>(cin)` is a `Reader` object that will fetch a sequence of `double` values from `cin`.

With this class and the `Sum` class from the previous example, we can now add numbers read from the input:

```
main()
{
        cout << Sum<double,Reader<double> >(Reader<double> (cin)) << "\n";
        return 0;
}
```

The remarkable thing about this example is that the `Sum` class is precisely the same as was used to add the elements of an array.

**Summary**

The key to successfully building a system of any significant size is to be able to break it up into pieces on which people can work independently. The key to that, in turn, is to define clear interfaces between those pieces.

This has been an illustration of how to use C++ class definitions as interfaces, in order to reduce the amount of information that one part of a system has to know about another. Specifically, we took a simple `sum` function and split each of its three parts off into an independent piece of code.

As templates come into wider use, techniques of this kind will give us ways of improving our system designs.

# Experiences in Using the C++ Task System

Philippe Gautron
Rank Xerox France & LITP, Institut Blaise Pascal
Université Paris VI
4 place Jussieu, 75252 PARIS CEDEX 05, France
gautron@rxf.ibp.fr

Short Paper
C++ Workshop – Lund, June 1991

The purpose of this paper is to describe the use of the C++ Task System through different applications.

A didactic example, the fibonnacci series, is first presented as introductory approach. The second example is the simulation of a multithread kernel as basic system architecture for a distributed object-oriented system prototype. The third example is the communication layer support for the development of musical software.

## 1 What the C++ Task System Is

The C++ Task System is a library designed to support concurrent programming. The library is part of the standard AT&T distribution [AT&T 89] and is available on different architectures, such as 3B, VAX, 680x0 and SPARC.

The library was designed in 1980 [Stroustrup 80] and is a typical example of object-oriented development: the current interface is (almost) similar to its first design, whilst the implementation has been revised at several occasions [Stroustrup and Shopiro 87, Shopiro 87].

The task system is an appropriate support for simulation in a pure coroutine style. The notion of *task force* can be introduced to define a collection of cooperative activities computing towards a common goal. The different tasks of a task force share a single address space, typically a UNIX process. The task system scheduling is based on a single strategy: no preemption, no priority, FIFO mode.

## 2 A Didactic Example

The fibonnacci series will be our introductory example [Gautron 89]. Its definition is:

$$U_{n+2} = U_{n+1} + U_n$$

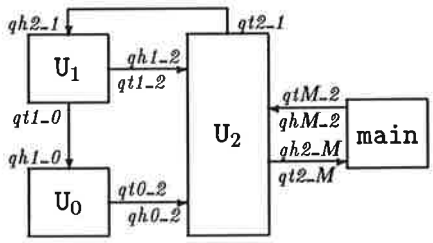
with:

$$U_0=0,\ U_1=1$$

Recursive programming is a simple (and inefficient) way to write this function:

```
unsigned int fib (unsigned int n){
    return n < 2 ? n : fib (n-1) + fib (n-2);
}
```

A more efficient alternative is to consider each component of the series as a separate task. Three tasks, $U_i$, hold the series values at times $n$, $n+1$ and $n+2$. In order to allow the calculus to proceed, $U_{n+2}$ and $U_{n+1}$'s values become respectively $U_{n+1}$ and $U_n$'s, and the new value of $U_{n+2}$ is then calculated. A fourth task, the `main` task, implicitly created by the task system when the application is started, controls the process on its whole. The task force can be shown as follows (the communication links are explained later):




A new task is a user-defined class derived from class `task`. For example:

```
class U0 : public task {
    public:
        U0 (qtail *qt0_2, qhead *qh1_0);
        void behavior();
};

U0::U0 (qtail *qt0_2, qhead *qh1_0) : task ("U0") {
    ...
}

class U1 : public task { ... };

class U2 : public task { ... };

main () { ... }        // task main
```

A new executing environment (stack, registers) is allocated when the `task` constructor is called. A new coroutine is created when the derived class constructor is executed.

Communication among tasks can be implemented through global variables (global to the process containing the task force) or by message passing. The non-preemptive nature of the task system favours the message passing style: the library does not provide any support for monitoring shared data.[1]

The library supplies three classes –`object`, `qtail` and `qhead`– for the management of inter-task communications. In our example:

- instances of class `Message` carry out integer values:

```
// define a user message
  class Message : public object {
      public:
          Message (unsigned int value = 0);
          void setValue (unsigned int value);
          unsigned int getValue();
  };
```

[1] Access synchronization can be easily added to the library [Shopiro 87].

- communication channels are pairs of qhead/qtail pointers:

```
// create a one-way channel, by allocating its head and associating it a tail:
  qhead *qh = new qhead;
  qtail *qt = qh->tail();
```

- series values are messages sent through a ring network:

```
// instantiate a Message initialized to handle value
  Message send = new Message (value);

// insert it in the queue
  qh->put (send);

// retrieve it from the queue
  Message *receive = (Message*) qt->get();
```

The main function, executed as activity of the main task:

1. allocates the different channels for the task force execution,
2. creates the three tasks with the appropriate channel links as arguments,
3. waits for $U_2$'s result.

$U_2$ computes the sum of $U_0$ and $U_1$'s values while $U_0$ and $U_1$ behave like buffers, initialized respectively to 0 and 1. $U_2$ returns its result to main which, in turn, sends a dummy message to $U_2$. The values held by $U_1$ and $U_2$ are then shifted and a new calculus can proceed. This loop is performed as long as main re-starts the computation: the number of loops is the series argument minus two.

## 3 More About the Interface

Task scheduling can occur either explicitly, when specific member functions of class task are invoked, or implicitly, on queue overflow and queue underflow.

task member functions include:

- task::resultis: terminates the invoked task and causes the task force scheduling.
- task::delay: suspends the invoked task for the time argument. A class timer provides a similar behavior.
- task::cancel: destroys the invoked task.
- task::wait: wait on an arbitrary instance of class object. The waiting condition is the value returned by a call to the virtual function object::pending. Classes qtail and qhead publicly derive from object.

Actions performed when queue overflow or queue underflow occurs depend on the mode of the queue. The default and common use is to suspend the task which invoked the queue and to cause the task force scheduling.

# 4 About the Implementation

The implementation uses assembly code for the basic operations of task management. Different implementations have been proposed to take advantage of specific architectures [Doeppner and Gebele 87], or of lightweight processes [Birrell 89, Gautron 91].

The use of inheritance for creating a new task is not without constraints. Indeed, in C++, a base class constructor is called within the body of a derived class constructor. The executing environment for a new coroutine is thus allocated *after* the start of its execution. Prohibiting inheritance for task creation should not be enough. In any case, flexibility –task creation with an arbitrary number of arguments of arbitrary type– must be paid with non-portable code.

The task system has been implemented with UNIX as privileged target system. Asynchronous events handling [Shopiro 87] is supported. The task force is contained within a single UNIX process: a task blocked on I/O events causes the whole process to sleep and does *not* involve the scheduling of the task force.

The current implementation suffers from:

- compiler-dependent and machine-dependent codes: porting is not easy
- painful debugging
- non-protected resources (stacks are heap-allocated)
- no check for stack overflow

Positive points are:

- performances suitable for efficient simulation
- high-level interface
- convenient facilities for inter-task communications

# 5 Prototyping a Distributed Object-Oriented System

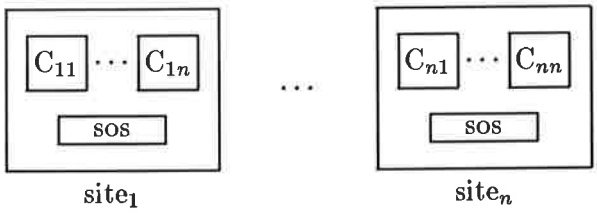
SOS is a prototype distributed object-oriented operating system built on top of UNIX [SOR 89, Shapiro 89]. SOS was designed to support distributed or fragmented objects: the data of an object can be localized on different sites and object fragments can remotely invoke each other.

## 5.1 System Resources

System resources supplied by the SOS kernel include:

- contexts: separate address spaces implemented as UNIX processes
- lightweight threads: implemented within a context as tasks of the task system
- (local and remote) inter-context communications through UNIX sockets

Each site runs the kernel (a specific context –`sos`–), an arbitrary number of pre-defined contexts (a naming service for example), and user-created contexts:




The kernel is in charge of context management (creation, deletion) and of initialization of point-to-point connections between contexts.

Within a context, tasks can be created:

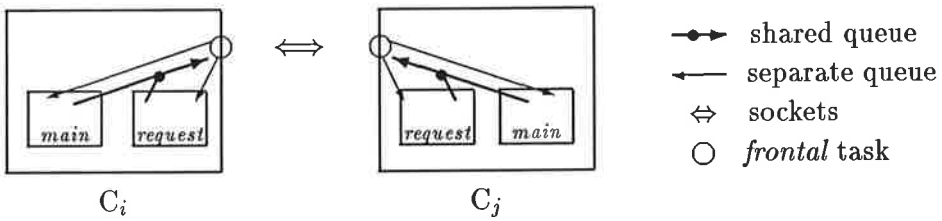
- by the system, to process incoming remote invocations
- by the application, to split its own execution.

## 5.2 Inter-context Communications

The basic communication protocol managed by the system distinguishes between *reply* messages (reply to remote invocations on the application's initiative) and *request* messages (for local execution result of remote invocation).

When a new context is allocated, at least three tasks are created: the `main` task for executing the application, a `frontal` task for the management of inter-context communications, and a `request` task for processing the first incoming message.

Two communication channels are associated to each task: a particular channel for input messages, a channel shared between all the tasks for output messages. The communication links can be summarized as follows:




The `frontal` task performs a UNIX system call, `select`, to poll the sockets for incoming remote invocations. Output messages are automatically arranged on the shared queue: no poll is required for these messages.

To optimize task creation, a pool of 8 pre-defined `request` tasks is allocated when the context is started. A new task is created when this pool is insufficient.

## 5.3 Support for Building Applications

To create a new task, the application must define a class derived from `SosTask`. This latter derives from `task` and its constructor allocates the communication channels, as required by the system, for the newly created task. System and application tasks are not distinguished by the task scheduler.

UNIX I/O management was of a particular concern during the design of the system: a realistic prototype requires a blocking I/O to invoke the task scheduler and not to cause the whole task force to sleep. SOS supplies a specific class, `UnixChannel`, allowing applications to manage I/O requests

in conjunction with the task system. A `UnixChannel` instance is initialized with a file descriptor (including 0, 1 or 2) and the class interface provides `read/write` member functions behaving like the similar UNIX system calls. I/O requests are sent to the `frontal` task through channels of the task system and the file descriptors are introduced as needed into the mask of the call to `select`. `read/write` operations are thus changed into `get/put` operations. The `read/write` system calls are performed by the `frontal` task and the application can be advisedly scheduled by the task scheduler.

### 5.4 Conclusion

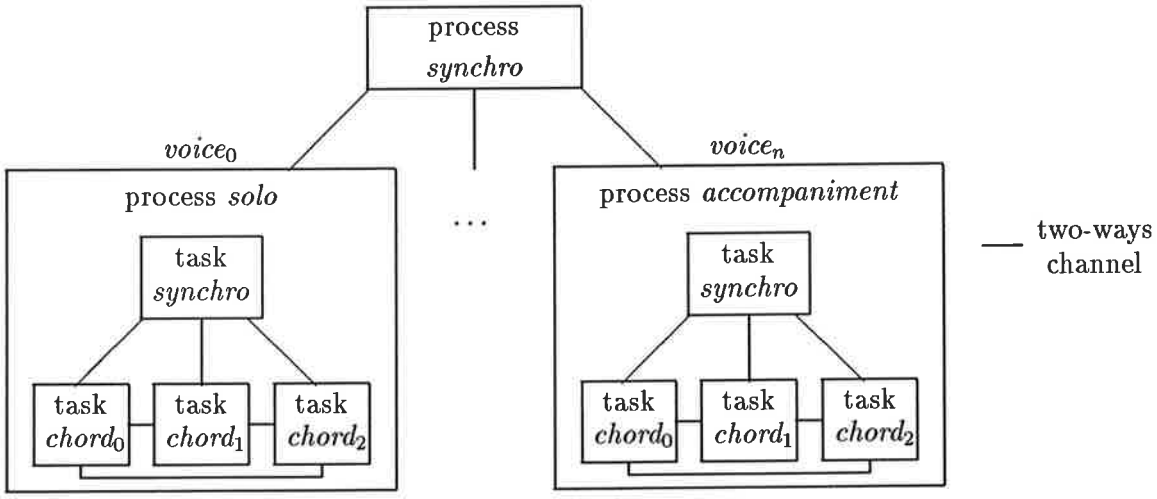
This approach simulates a multithread kernel, with threads managed by the operating system. This simulator was an appropriate support for studying distributed communication protocols [Makpangou and Shapiro 88], experimenting with fragmented objects [Gourhant and Shapiro 90], and building distributed applications [Marques and al. 88]. The major drawbacks of the use of the task system were the lack of preemptive scheduling (the prototype requires some collaboration between the application and the system to time-multiplex the UNIX process), and the lack of priorities for the scheduling (to favour the application for example).

As related work, the task system was also used to support the simulation of a multiprocessor operating system modeled on the Choices family [Johnston and Campbell 88].

## 6 A Musical Application

UNIX processes and C++ tasks were the system support we used to implement a simulation of jazz improvisation [Gautron 85]. Scores were performed in two stages: score generation in batch mode on a VAX, score playing in real-time on a dedicated machine (4X).

Parallelism is of particular interest for musical development. In our application, parallelism occurs at two levels: loose coupling between voices (solo(s), accompaniment(s)) and strong coupling between chords of a same voice. Each voice is implemented as a UNIX process and each chord as a task of the task system. An arbitrary number of voices can be created, limited by the availability of the UNIX resources. The overall organization reflects theses requirements:




Inter-process communications are achieved through UNIX pipes, and inter-task communications through queues of messages. Communication protocols allow both transfer of data and synchronization. Chord synchronisation is under control of the `synchro` task, and voice synchronisation under

control of the **synchro** process. Data can be directly transferred between any task within a same process. Data exchanged between tasks of different processes are sent via the **synchro** tasks and the **synchro** process, which behave like the master tasks and the master process.

Three different softwares (blues and ballad) were successively developped on top of this architecture. Each voice is first initialized with a large set of pre-defined musical phrases, shared by the chords of a same voice. Score generation, written in C++, amounts to articulating the musical phrases, a mix of heuristic rules and random. A new score results from a new seed for the random number generator.

Splitting the application into processes helps debugging: in our application, inter-process communications could be easily faked in order to individually test each voice. The task system has proved to be an appropriate and efficient support for our developments. **chord** tasks within a same process are *cloned*: they run exactly the same code and their executions differentiate by the arguments passed to the constructor.

## Acknowledgments

SOS was the contribution of many people. Marc Shapiro inspired the design. Dima Abrossimov wrote the kernel. Yvon Gourhant refreshed my memory about the system.
The musical softwares were developped at IRCAM, on David Wessel's responsability. Indelible thanks are addressed to Andre Hodeir without whom this work could not have been possible.

## References

[AT&T 89] *AT&T C++ Language System Release 2.0: Product Reference Manual.* 1989. Select Code 307–146.

[Marques and al. 88] José Alves Marques, Luis Pinto Simoes, Nuno M. Guimaraes, and Luis Carrico. Interface Manager and Generator for SOMIW. *SOMIW (Esprit 367) Report 10/R4*, INESC, Rue Alves Redol 9-2, 1000 - Lisboa, (Portugal), January 1988.

[Birrell 89] Andrew D. Birrell. An Introduction to Programming with Threads. *Technical Report 35*, Digital Systems Research Center, Palo Alto, CA (USA), January 1989.

[Doeppner and Gebele 87] Thomas W. Doeppner and Alan J. Gebele. C++ on a Parallel Machine. In *Proceedings and Additional Papers, USENIX C++ Workshop*, pages 95–107, Santa-Fe, NM (USA), November 1987.

[Johnston and Campbell 88] Gary M. Johnston and Roy H. Campbell. A Multiprocessor Operating System Simulator. In *Proc. USENIX C++ Conference*, pages 169–181, Denver, CO (USA), October 1988.

[Gautron 85] Philippe Gautron. Unix et multiprocessus, C++ et multitâche : Une approche logicielle de la simulation de l'improvisation dans le jazz. *PhD thesis*, Université Paris XI-Orsay, IEF, Paris (France), October 1985. Also available as Technical Report LITP 86-16, LITP, Université Paris VI - PARIS.

[Gautron 89] Philippe Gautron. An Introduction to the C++ Task System. In *The C++ Report*, 1(10), November 1989.

[Gautron 91] Philippe Gautron. Porting and Extending the C++ Task System with the Support of Lightweight Processes.
In *Proc. USENIX C++ Conference*, pages 135–146, Washington, D.C. (USA), April 1991.

[Gourhant and Shapiro 90] Yvon Gourhant and Marc Shapiro. FOG/C++: a Fragmented-Object Generator.
In *USENIX C++ Conference*, pages 63–74, San Francisco, CA (USA), April 1990.

[Makpangou and Shapiro 88] Mesaac Makpangou and Marc Shapiro. The SOS Object-Oriented Communication Service.
In *Proc. 9th Int. Conf. on Computer Communication*, Tel Aviv (Israel), October–November 1988.

[SOR 89] SOR. SOS reference manual for prototype V4.
*Technical Report 108*, INRIA, June 1989.

[Shapiro 89] Marc Shapiro. Prototyping a Distributed Object-Oriented Operating System on UNIX.
*Workshop on Experiences with Building Distributed and Multiprocessor Systems (WEBDMS)*, Ft. Lauderdale, FL (USA), October 1989.

[Shopiro 87] Jonathan E. Shopiro. Extending the C++ Task System for Real-time Control.
In *Proceedings and additional papers, USENIX C++ Workshop*, pages 77–94, Santa-Fe, NM (USA), November 1987.

[Stroustrup and Shopiro 87] Bjarne Stroustrup and Jonathan E. Shopiro. A Set of C++ Classes for Co-routine Style Programming.
In *Proceedings and Additional Papers, USENIX C++ Workshop*, pages 417–439, Santa-Fe, NM (USA), November 1987.

[Stroustrup 80] Bjarne Stroustrup. A Set of C Classes for Co-routine Style Programming.
*Technical Report CSRT 90*, AT&T, Murray Hill NJ (USA), November 1980. Revised (1) July 1982, (2) November 1984.

# C++ SYMBOLIC DEBUGGING

**Dmitry Lenkov**
**Shankar Unni**

Hewlett-Packard Company
California Language Laboratory
19447 Pruneridge Avenue, MS: 47LE
Cupertino, CA 95014
E-mail: {dmitry|shankar}%hpda@hplabs.hp.com

**ABSTRACT**

Many software developers using C++ have had painful experiences in symbolically debugging C++ programs. Absence of C++-specific debuggers is one of major reasons for this. Building a C++ debugger is a non-trivial task that requires one to define functionality appropriate for debugging C++ programs and design an implementation to support this functionality.

This paper discuses debugger functionality necessary for symbolically debugging C++ programs. It then introduces a practical approach for an implementation supporting this functionality.

## 1. Introduction

The introduction of C++ into the market has placed new requirements on symbolic debuggers. C++ [1] is a C-based object-oriented language with a number of advanced features. Three parts of the C++ definition in the previous statement emphasize three groups of challenges which a C++ symbolic debugger is supposed to address. Since C++ is a C-based language, a C++ debugger should support functionality offered by C debuggers in common use. C++ is an object-oriented language [2]. This means a C++ debugger should support new syntactic conventions, for example, specific to class scope or to member pointers. It also should be able to display C++ objects as the user declared them and allow their modification. This is complicated by the fact that a pointer to a class in C++ can actually point at runtime to an object belonging to any of the derived classes of the said class, in addition to objects belonging to that class. This poses a challenge to the debugger: to try to distinguish which class the object really belongs to, and display the object using the appropriate type template. In addition C++ has a number of advanced features [1] such as overloaded functions and operators, inline functions, etc.

Until recently there were no C++-specific symbolic debuggers. And even now many hardware platforms have either no such debuggers or C++ debuggers with rudimentary functionality. Without a good symbolic debugger, C++ can be very difficult to debug [4]. Besides the complexity of the language, there are other factors such as the fact that C++ compilers mangle variable and function names in order to support overloading and type-safe linkage, and that in many debuggers, the user is effectively debugging some intermediate C code that is very hard to read and extrapolate back to the original C++ source.

In choosing the right approach to the definition of functionality and to the implementation of a C++ debugger, it is appropriate to consider two groups of debuggers currently in wide use. Debuggers in the first group have been primarily aimed at traditionally structured programming languages like

### Overloaded functions

When a function or operator is overloaded, the debugger should be able to recognize this fact, and provide a convenient interface to refer to a set of overloaded functions, either individually or collectively. The same specification format as above can be used to specify a set of overloaded functions. However the debugger should provide a special interface to allow the user to select a particular function, or a subset of functions, out of a set of overloaded functions (for breakpoints on individual functions). This interface can either display full declarations (prototypes) for all functions in the set and ask for a selection or prompt for function parameter types and make selection internally.

For example, one can ask the debugger to set a breakpoint at:

- All the constructors for class Foo,
- All functions called `print()` for the class `Foo` and its descendants.

and so on.

### Member function of an object

Given an object, or a pointer to one, the user can select a member function of the class to which it belongs to set a breakpoint. This is subtly different from the previous cases: if this member function is a virtual member function, then the debugger sets a breakpoint at the actual function that the above expression refers to (based on the real type of the object at runtime). For example:

```
class a {
   public:
     a();
     virtual void print ();
};

class b : public a {
   public:
     b();
     void print ();
};

.......
a* p_a = new b();
```

After the last statement is executed, `p_a` points to an object of the class `b`. So if `p_a->print` is specified to the debugger to set a breakpoint, the breakpoint will be set on `b::print`.

## 2.1.2. Data breakpoints

Data breakpoints are those that are triggered on access or modification of an object. For the case of C++, there are two specific types of data breakpoints that are quite useful:

### Class breakpoints

In order to easily debug the entire interface of a class, it is desirable to be able to set breakpoints at all member functions of a given class. This sort of breakpoint allows the programmer to watch accesses to a whole group of objects. We support two variations of class breakpoints:

(i) breakpoints that are set on member functions actually declared in a given class, and

(ii) breakpoints that are set on all member functions, including those inherited from base classes.

The class name serves as a specification in both cases. To distinguish between these cases, two different command names are used.

### 2.2.1. Symbolic display of the type of a class instance

Most symbolic debuggers maintain symbol tables that contain type information sufficient to display the type of a variable or a function parameter. However, this becomes more complex in the case of a pointer (or reference) variable (or parameter) referencing objects of a class which has subclasses. When the user wants the type of an object referenced by such a variable to be displayed, `p_b` in the example above, there are two alternatives in regard to what he actually wants to be displayed:

(a) The statically declared type of `p_b`. Since every `c` is a `b`, this does make sense.

(b) The actual type of the object that `p_b` points to at run time (in this case, class `c`). This requires run-time access to the object type information.

Both alternatives should be supported by a C++ symbolic debugger. Certain restrictions on the second alternative are considered in details in a separate section below.

### 2.2.2. Symbolic display of a class instance

The two alternatives considered in the previous section apply to the symbolic display of a class instance. In the case of the first alternative, the display of an object referenced by `p_b` may look like:

```
class b : public a {
   j = 3;
}
```

For the second alternative it may look like:

```
class c :
   public b {
      j = 3;
   } {
   k = 1;
}
```

In addition the user should be able to display an entire object, including members of all base classes. In this case the display may look like:

```
class c :
   public b :
      public a {
         i = 10;
      } {
      j = 3;
   } {
   k = 1;
}
```

Besides these variations, there are others based on displaying instance variables based on scoping and access levels.

### 2.2.3. Modification of class instance members

The user should be able to modify instance variables of an object. This can be done by following the syntax of C++ assignment statements. For example

### 3.3. User modification of flow of control

The user should be able to change the program counter at any time when the program is stopped. For instance, he/she may want the debugger to execute some sequence of statements that it had (or is about to) skip over because of some incorrect program condition.

## 4. Implementation issues

We established in the previous section that accessing type information at run-time is necessary to support functionality described there. This is probably most non-trivial challenge that we discovered during implementation of our C++ debugger.

Another problem results from the fact that many C++ products are based on the AT&T C++ translator which produces C code that is difficult to read and process. This is a problem because much of the symbolic information about the C++ program is lost in the translation. For instance, a lot of the information about access restrictions, inline functions, enumeration constants, and the like is lost in the translation. Also, the exact inheritance tree is difficult to reconstruct from the intermediate C structures that are generated.

Since our implementation is based on AT&T's translator, we had to find a way around and generate correct information about C++ symbols, their types, and their location in C++ code. This section outlines our solution to these two problems.

### 4.1. Generating debug information

In a true integrated C++ compiler, generating debug information is a relatively straightforward exercise: the only problem to be solved is the actual representation of the debug information which the debugger can use when debugging the program. However, in a classic translator-based environment, this is a much harder problem. As mentioned above, the basic difficulty is that by the time the C++ source is translated to C, we have lost much of the original structure of the original C++ source.

There are several C++ environments today that attempt to present the user with a crudely reconstructed "C++ debug environment" [4], but these are demonstrably difficult to use.

Thus, we have to rely on the translator to generate enough information on the side to supplement the debug information emitted by the C compiler so as to be able to re-create (in the debug information) the original C++ source. This is what we have done in our implementation (hpc++).

In this environment, when the user requests debug information to be generated, both the translator (*cfront*) and the C compiler generate some debug information. The translator generates information about types, variables and functions (in other words, everything that it can determine in the absence of code-generation-time information like stack layout and code offsets).

There is a "merger" program that runs after the C compiler, which gathers the debug information from the C compiler and cfront, and merges it to produce complete debug information for the original C++ compilation unit, as shown in figure 1.

```
 public:
   void print ();                  // prints a_i and b_i
};
.............................

a* p_a;

if (expression)
  p_a = new a();
else p_a = new b();

p_a -> print();
```

Regardless of whether p_a points to an instance of the class a or class b, at the execution of the last statement a::print is always called. Since a reasonable symbolic debugger would have the type information for p_a, it would also find information about the class a. Thus we can get to the right function print and display the value of a_i. On the other hand, there is no way to access the field b_i by using pointers or references to the class a without using casts. This eliminates a possibility of mkaing an error involving b_i if casts are not used. Suppose now that a cast is used:

```
((b*) p_a) -> print();          // prints a_i and b_i
```

In this case we can specify the same cast to a debugger (our debugger understands this). The situation changes significantly if the function print is virtual. In this case `p_a->print()` becomes a generic call and may invoke either `a::print()` or `b::print()` depending on which instance `p_a` points to. However , as mentioned above, if print is a virtual member function then instances of the class `a` and instances of the class `b` contain pointers to different virtual tables. Thus they can be used to find correct information about the class of an instance pointed to by `p_a`. There are at least two approaches to provide access to class information through virtual pointers.

(a) The first one [8] requires us to associate an additional virtual function that would return a pointer or reference to the class information generated by a compiler or a special tool before execution. These functions can be generated automatically, either by the compiler or translator, or by a separate tool.

This approach definitely provides a powerful general mechanism. However, we have to be careful that this mechanism is always in place, even when not compiling the code specifically for debugging. Otherwise, there arises the possibility that the execution of the program will differ based on whether it was compiled with or without debugging information.

(b) We have developed a different mechanism. We are not aware of a previous use of this approach for accessing C++ class information at run-time. The basic idea is to use virtual table addresses as run-time class identifiers and search for class descriptions generated before the execution through a hash table mapping virtual table addresses into references to class descriptions. Below we discuss details of this approach. We assume that virtual tables are implemented as described in [1] and [9].

### 4.2.1. Virtual table implementation by AT&T C++ translator

Given a class that has at least one virtual member function, the objects belonging to this class, or any of its derived classes, have one or more "virtual table pointers" embedded in them, which can serve as identifying handles for the objects. However, this scheme has several drawbacks:

an amount equal to the size of an object of class A. Later, when attempting to determine the type of the object using the latter handle (i.e. a pointer to class B), we will have to determine (a) that it belongs to class C, and (b) that we have to subtract the size of class A from the value of the handle in order to get to the base of the object. Thus, the table for this program fragment may look like figure 2:

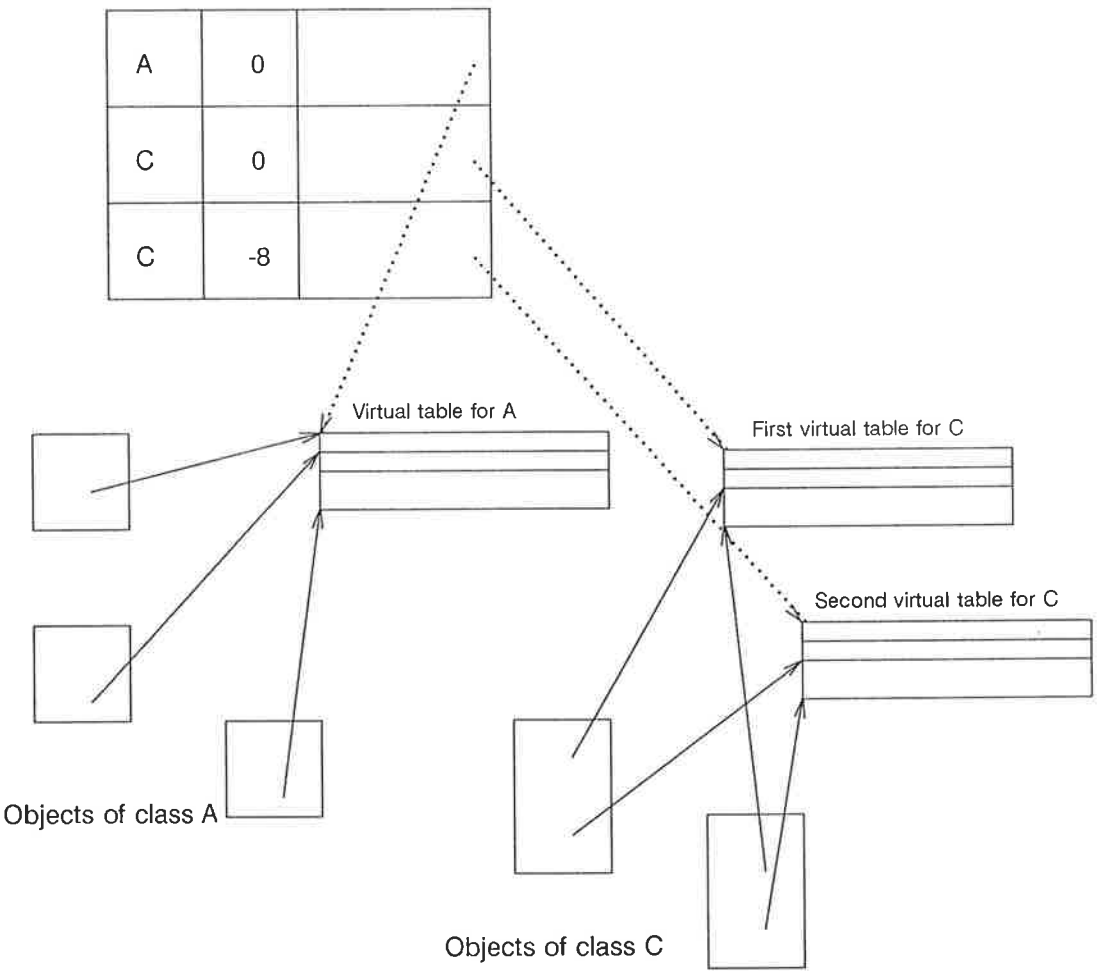



**Figure 2. Lookup table used by debugger.**

The debugger has enough information to know, given a declared type, where to look for the virtual table pointer inside the object. Armed with this information, the debugger extracts the value from the object and looks it up in this table, and can thus determine the real type of the object.

# Type Identification in C++

*Dmitry Lenkov*
*Michey Mehta*
*Shankar Unni*

California Language Laboratory,
Hewlett-Packard Company,
19447 Pruneridge Avenue,
Cupertino, CA 95014.

E-Mail: {dmitry|mnm|shankar}@cup.hp.com

*ABSTRACT*

Many applications and class libraries require a mechanism for run-time type identification and access to type information. This paper describes a general type identification mechanism consisting of language extensions and library support. We introduce the following language extensions to support type identification uniformly for all types: a new built-in type called **typeid**, and three operators **stype** (static type), **dtype** (dynamic type), and **subtype** (subtype inquiry). We also describe a library class called **TypeInfo**, which is used to access compiler generated type information. Special member functions of the **TypeInfo** class are used to extend the compiler generated type information. An implementation strategy is presented to demonstrate that the proposed extensions can be implemented efficiently. We compare our proposal with previous work on runtime type identification mechanisms.

## 1. Introduction

There have been various attempts made in C++ to implement a method of type identification for objects and a mechanism to access additional type information[3][4][5]. There are several reasons why such identification is needed.

- Support for accessing derived class functionality

  Many of the commonly available C++ class libraries (such as NIH[4], InterViews[6], and ET++[5]) consist of an inheritance hierarchy with a root class (such as the Object class in NIH). When dealing with pointers to this root class, a common operation in these toolkits is to determine if a pointer points to an object of a derived class. If so, the pointer is castdown to the derived class so that a derived class member function may be invoked. Since C++ performs its type checking at compile time, type information is not available at runtime, and each toolkit uses different mechanisms for determining the actual type of the object being dereferenced. When the root class is a virtual base class (as in NIH), since the castdown is not permitted by C++, the library must invent mechanisms to circumvent this restriction. We show that our type identification scheme supports subtype queries and castdowns.

- Support for Exception Handling.

  The exception handling mechanism[1][2] requires type identification at run time, in order to match the thrown object with the correct **catch** clause. The exception handling mechanism is

Here are some examples of how the **subtype** operator can be used.

Example 1:

```
List* l_p = // initialize
...
l_p = // point to some other list
...
if( subtype( SortedList, l_p)) {
   Key k = (SortedList*) l_p -> least_key();
   ...
}
...
if( subtype( LenSortedList, l_p))
   cout << ((LenSortedList*) l_p -> length());
```

Another example is calling a function that requires an actual parameter which is a derived class.

Example 2:

```
void func( LenSortedList *);
...
if( subtype( LenSortedList, l_p))
   func( (LenSortedList*) l_p);
```

In the previous two examples the castdown operation was used to allow functionality defined on subtypes to be used. However, the subtype operator also has applications that do not require a castdown operation. Consider:

Example 3:

```
void sort( List*);
...
List *l_p = // initialize
...
if( !subtype( SortedList, l_p))
   sort( l_p);
```

Consider another example:

Example 4:

```
void other_func( OtherType *);
...
OtherType p = // initialize
...
if( subtype( SortedList, l_p))
   other_func( p);
```

In this example, the functionality associated with the SortedList subtype is invoked as in example 2. However actual actions take parameters of types other than SortedList. Thus the castdown operation is not needed.

C++ types fall under three different categories with regard to the subtype operator: polymorphic classes (those that have virtual functions), simple types, and non-polymorphic classes. For polymorphic classes the behavior of the **subtype** operator is illustrated above. A simple type (int, int (*)(), etc.) has no subtype (other than itself). Thus the **subtype** operator establishes equality for them with the result defined statically at compile time. For example,

an object of a class which is not a subtype of LenSortedList? Currently if one attempts:

```
B* b_p = //initialize
C* c_p = (C*) b_p;
```

where B and C are unrelated but have a common parent, an unchanged value of b_p is assigned to c_p. It is reasonable to do the same in the case of dynamic casting.

### 2.3. The Type Identification Scheme

Some of the applications described in the introduction would require a unique identifier to be associated with a type. The primary component of this type identification scheme is the predefined type called **typeid**.

#### 2.3.1. The typeid Type

The **typeid** type is a simple predefined type, similar to **int** or **void***, with a few operations defined on it. Expressions evaluating to the **typeid** type can be compared for equality and inequality. Variables of the **typeid** type can be assigned or initialized with an expression of the **typeid** type. No other operations are allowed. Each unique type in an application has a unique value of the **typeid** type associated with it. We define two operators which return values of type **typeid**.

**stype** returns the type identifier (**typeid** value) for the static type of an expression. It can also be applied to a type name and returns the type's **typeid** value. The **dtype** operator can be applied to any expression that evaluates to a pointer to a type. If the pointer points to a polymorphic class, **dtype** returns the type identifier (**typeid** value) of the actual type of an object pointed to by this pointer. Note that this type must be determined dynamically. If the pointer does not point to a polymorphic class, **dtype** returns the **typeid** value of the static type pointed to by the pointer definition.

Example 6:

```
List* l_p = new SortedList;
int num_Sorted_Lists = 0;
...
typeid t  = dtype(l_p);
if (t == stype(SortedList)) num_Sorted_Lists++;
```

The reason that **stype** and **dtype** are not predefined member functions is the same reason that **sizeof** is not a member function: both identify a fundamental property of types, as opposed to an operation on objects of those types. On the other hand, both can be applied to any types including types such as (int* (*) ()).

An alternative to the **stype** operator is to allow an explicit conversion of any type to **typeid**. However this would also require the conversion of type names to **typeid**. The above example would look like:

```
List* l_p = new SortedList;
int num_Sorted_Lists;
...
typeid t  = dtype(l_p);
if (t == typeid(SortedList)) num_Sorted_Lists++;
```

may extend the type information associated with a class. We believe that it is best to allow the class library creators and users to specify what information needs to be associated with each type.

The following mechanism is used to extend the type information associated with a type.

- We provide a member function called "add_aux_typeinfo" in the **TypeInfo** class. This member function is used to attach additional type information to the minimal type information generated for a type.
- We provide a member function called "get_aux_typeinfo" in the **TypeInfo** class. This member function is used to retrieve any additional type information that a user may have attached to a type.
- It is reasonable to expect that multiple users may wish to attach auxiliary type information to the same type. Therefore, the notion of a "key" is required. A "key" is used to distinguish between multiple auxiliary type information objects attached to the same type.

Consider an example:

```
// User wants to add a "name" field to the TypeInfo for class Widget

//See section 3.3 for an explanation of the AuxTypeInfo class
class NameInfo : AuxTypeInfo {
   char *name;
public:
   NameInfo(char* n): name(n){};
};

NameInfo NameInfoObject = "Widget";
// Attach additional type information for "Widget"
get_typeinfo( stype(Widget)) -> add_aux_typeinfo( &NameInfoObject, stype(NameInfo));

// Assuming the user has installed name information in Widget, and
// all classes derived from it, here is how a user could dynamically
// find out the name of a class.
Widget* w = // initialized to something;
char* name = (NameInfo*) (get_typeinfo( dtype(w)) ->
                  get_aux_typeinfo( stype(NameInfo))) -> name;
```

The extensibility scheme we have proposed is essentially a convenient method of adding a static member (in fact, a virtual static member) to an existing type, without having to modify the type in any way. Individual users can certainly come up with various methods of accomplishing the same result, but the goal here is to propose a *uniform* method for extending type information.

### 3.3. The AuxTypeInfo Class

Any additional type information should be defined as a class derived from AuxTypeInfo. Instances of this are used to link the auxiliary type information objects. See section 4.5 on additional information about the implementation of extensibility. The AuxTypeInfo class is defined as follows:

- Reasonable space and execution performance should be expected when using type inquiry operators.
- When a program uses type inquiry operators the execution cost should be paid only when (and if) these are actually used at runtime. Any startup cost should be minimized.
- We wanted a scheme that would work with both "munch" and "patch" (see next section).

### 4.1. Terminology

The implementation section of this paper uses terminology that may not be familiar to everyone.

- *Munch* and *Patch* : Munch and Patch are schemes used by AT&T C++ front end based implementations to ensure that all static objects are appropriately initialized before the main program begins. After a program is linked, a "munch" implementation scans the resulting executable for special symbols and constructs additional data structures which are then relinked into the program. In a "patch" implementation, the executable resulting from a link is also scanned for these special symbols. But instead of constructing additional data structures, "patch" fixes existing data structures, for example, linking some of them together.
- *vtables*: "vtables" are tables, or data structures, which support virtual function calls. A polymorphic object will contain one or more pointers to one or more such tables.

### 4.2. Implementation Details

We now provide some details of a possible implementation scheme. In section 2 we described the built-in **typeid** type and in section 3 we described a library routine **get_typeinfo** which will convert a **typeid** into a **TypeInfo***. A **typeid** is really equivalent to a **TypeInfo***, and in the rest of this section we will always use the **TypeInfo** class name.

Our overall implementation strategy is:

- One **TypeInfo** object per type:
  The type inquiry operators return a pointer to a *unique* **TypeInfo** object associated with the type. The reason we need to guarantee one unique object is so that pointer comparisons can be used to determine whether two types are the same.
- **TypeInfo** objects are only allocated if necessary:
  Our implementation scheme attempts to minimize the number of **TypeInfo** objects which are allocated, since we do not need to allocate one for every single type. Allocating a **TypeInfo** object for every single type we encounter in a program is not necessary, since a compiler can determine whether or not the **TypeInfo** object for a type is accessible at runtime.

#### 4.2.1. Allocation of TypeInfo objects

**TypeInfo** objects can be referenced at runtime for any of the following reasons:

1. We must have **TypeInfo** objects for the static types of any types used in type inquiry operators. For types which are classes we must also allocate **TypeInfo** objects for each ancestor in the class hierarchy. This is needed to allow traversal of the ancestor hierarchy of a class in order to support **subtype** inquiries and the **TypeInfo** class functionality. This also supports the exception handling mechanism.
2. We must have **TypeInfo** objects for all derived classes of polymorphic base classes on which the user performs a dynamic type inquiry operation (i.e **dtype** or **subtype**). Since a derived class can be defined in a compilation unit which is not visible to the compilation unit containing a type inquiry operator, **TypeInfo** objects have to be emitted for all polymorphic classes.

- Consider a class X for which we cannot find a unique place to initialize the **TypeInfo** information. Assume we allocate and initialize **TypeInfo** objects in file1 and file2 called TypeInfo_X_file1 and TypeInfo_X_file2, respectively.
- For the expression **stype(X)**, the equivalent C code we generate is:

```
C++ code                    C code
========                    ======
stype(X)                    TypeInfo_X_file1.RealTypeInfo (if in file1)
stype(X)                    TypeInfo_X_file2.RealTypeInfo (if in file2)
```

- The "patch" tool will notice that there are two TypeInfo objects for X, arbitrarily pick one of them, and make the "RealTypeInfo" fields of both objects point to the chosen version. Note that since we have allocated initialized objects, patch is allowed to modify them in the object file itself.

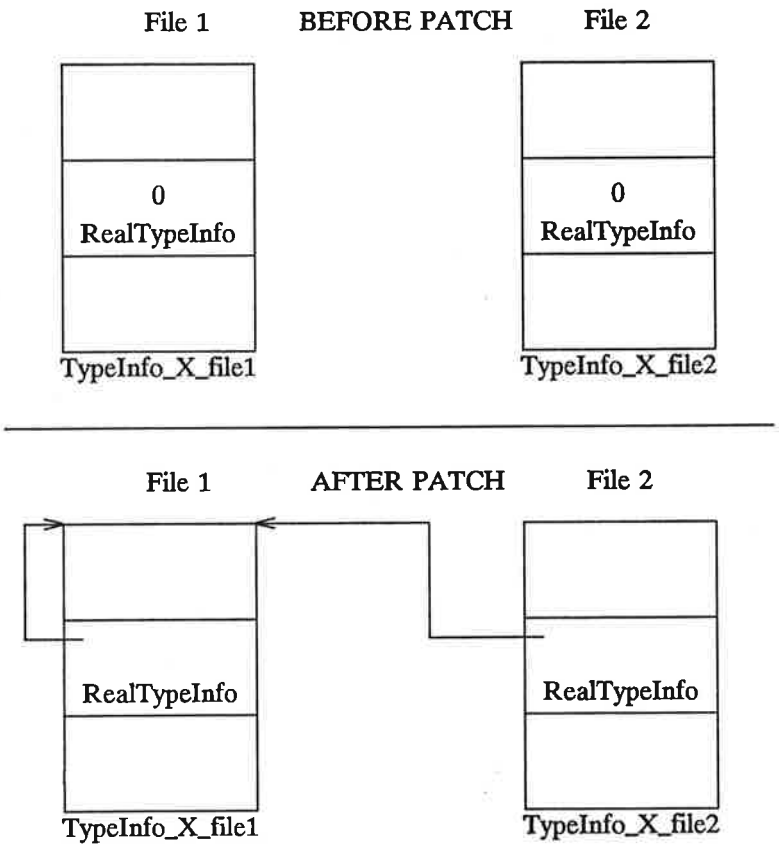



Figure 1: Patch Implementation

The primary advantage of this scheme is that there is no startup cost. Note that this same scheme cannot be used by munch without incurring some runtime cost, since there is no way for munch to initialize the "RealTypeInfo" fields without generating some code to execute at runtime. The next section describes a scheme for munch which involves no runtime initializations.

### 4.4. Shared Library Considerations

Shared libraries are a mechanism for multiple programs to share the same copy of routines linked into a shared library; linking with shared libraries will generally result in much smaller executables files than linking with archive libraries. Shared libraries add complexity to the implementation of type identification. Although we do not go into any details of how shared libraries work (since vendors differ in their implementations), we make the following assumptions about shared libraries:

- At link time, a tool like "munch" or "patch" will not get a complete picture of all object files which belong in this executable, since shared libraries can be explicitly loaded at runtime (for example you may may load a set of graphics routines which depend on the output device you are using).
- When a shared library is created, we assume that there will be some mechanism which will allow us to run "munch" or "patch" on the shared library.
- We assume that the shared library mechanism allows us to specify a routine that will be executed when the library is first loaded.

The "patch" and "munch" schemes previously described rely on being able to process a "complete" executable; since a link involving shared libraries results in an "incomplete" executable we need to make modifications to our schemes.

The previous section describes two conditions under which we cannot initialize a unique **TypeInfo** object at compile time:

1. Polymorphic classes which do not have unique vtables
2. Non-polymorphic classes

We now describe how this would be handled in a shared library implementation. Both implementations suggested below will have some runtime initializations being performed.

#### 4.4.1. Patch Implementation for Shared Libraries

The scheme of having a "RealTypeInfo" field does not work for shared libraries, because when a shared library is loaded it would be difficult (and expensive) to make each such field point to the appropriate **TypeInfo** object. The shared library patch algorithm looks like this:

1. At compile time we emit a tentative definition for a **TypeInfo** object which cannot be initialized in a unique file. We also emit an initialized definition for this **TypeInfo** object.
2. At "patch" time, we create a chain of initializations which should be performed; each entry in this chain will contain a pointer to each **TypeInfo** object which needs to be initialized, and a pointer to the corresponding initialized object. If there are be multiple initialized objects available, one is chosen arbitrarily. This algorithm applies when patching executables as well as shared libraries.
3. When "_main" is executed and when a shared library is loaded, all the **TypeInfo** initializations are performed before any other initialization code is executed. An initialization of a **TypeInfo** object is quick because we simply need to store a pointer to the initialized object within the **TypeInfo** object.

Patch cannot initialize **TypeInfo** objects during the patch phase itself, because of the method used by most linkers to implement tentative definitions. If a linker needs to allocate space for an uninitialized tentative definition, it will usually simply update the size of the uninitialized area, and the loader will be responsible for initializing this area to 0. Since there is no real image in the object file which contains the initialization data for uninitialized tentative definitions, there would be no way to "patch" it to a different value.

The scheme we have described does have some runtime cost, and this cost is the initialization of one word for each **TypeInfo** object allocated for non-polymorphic classes, and polymorphic classes

use different syntaxes for accessing this member. The Dossier scheme uses the ::, ., and -> operators, whereas we propose introducing two new operators **stype** and **dtype**. The main reason for our choice is to provide a consistent syntax for accessing type information for all types, not just classes.

Extensibility

Regardless of how much information is made available for a type automatically (for example, a list of ancestor classes), there will always be some applications which need additional type information. Our paper discusses a method for extending the standard type information generated by the compiler (the developer will have to take steps to ensure that this additional type information gets associated with the type).

Implementation

The Dossier mechanism relies on a tool to process the sources for an application and generate Dossiers. Although the sources may be partitioned into multiple sets (to handle libraries), care must be taken to ensure that the same Dossier is not generated twice. Our paper describes some implementation schemes in which the compiler automatically generates the necessary information, and no additional processing is necessary.

## 6. Open Issues

The type identification mechanism presented in this paper provides a reasonably complete set of functionality for type related operations and handling type information. In developing this mechanism we discovered some issues that require further discussion.

- Non-Polymorphic Classes

  Non-polymorphic classes inherently possess a certain inconsistency with regard to type identification. Although they form a subtype hierarchy in the same way as polymorphic classes, given a pointer to a non-polymorphic base class it is difficult to determine the true type identity of the actual object being dereferenced at runtime. There are various alternatives available:
  - make all non-polymorphic classes polymorphic;
  - make all non-polymorphic classes, except for "extern C" classes, polymorphic;
  - introduce a pragma to control this;
  - introduce a compiler option;

  Each of these options has serious disadvantages.

- ptr_cast operator

  The use of casts is usually unsafe. The idea of an alternate cast operator[7] that is supposed to be applied only when a legal conversion is possible, is attractive. This operator would raise an exception if applied incorrectly.

## 7. Conclusion

We have described a general type identification mechanism consisting of language extensions and library support. The language extensions introduced support a reasonably full set of type inquiries. The library class called **TypeInfo** has been introduced to allow access to compiler generated type information. While providing access to basic information about types, it also contains member functions which can be used to extend the compiler-generated type information. An implementation strategy has been presented to demonstrate that the proposed extensions can be implemented efficiently.

The proposed type identification mechanism should satisfy the requirements of application and class library developers for type identification, access to a subtype query mechanism, and run-time access to type information.

# C++ Standardization


Steve Carter — Chair, Int'l Concerns Working Group
X3J16, C++ Committee

Interim Convener,
WG21, C++ Working Group

| | |
|---|---|
| Postal Address | Bellcore<br>444 Hoes Lane, RRC 4A-737<br>Piscataway, NJ 08854<br>United States of America |
| Telephone | +1 908 699-6732 |
| Facsimile | +1 908 463-1965 |
| Electronic Mail | uel!attunix!bellcore!bcr!slc2<br>slc2%bcr.cc.bellcore.com@bellcore.com |


SLC - 5/31/91

# C++ Standardization

## Overview

- Why Standardize C++ ?
- Who Is Standardizing C++ ?
- What Are X3J16 and WG21 Doing?
- How will X3J16 and WG21 Work Together?
- When Will Standardization Complete?

SLC - 5/31/91

# *Why Standardize C+ +?*

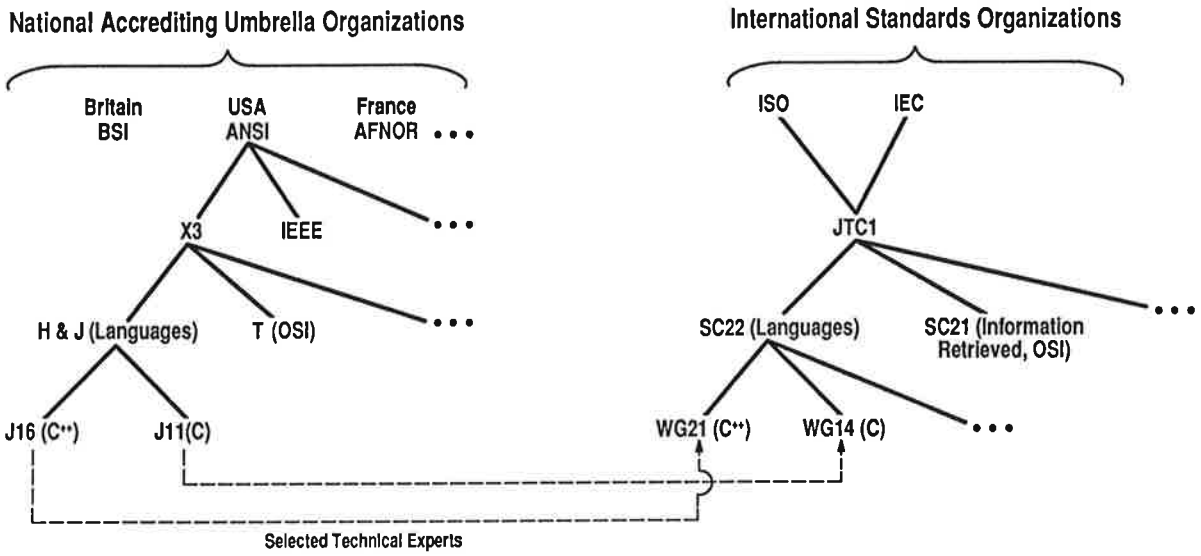
## Credibility, Reduced Cost and Increased Availability

- User Benefits
  - Easier to write vender independent source code
  - Wider product availability
  - Commodity pricing
  - Improved chances for support tools
- Benefits to Venders
  - Even playing field
  - Expanded market
- Vender and User Benefits
  - C+ + and associated products gain credibility

SLC - 5/31/91

# *Who Is Standardizing C++?*

## X3J16 and WG21




* Countries indicating they will participate in WG21: Canada, France, Japan, Netherlands, Sweden, USA, and USSR.

# What Are X3J16 and WG21 Doing?

- X3J16
  - To Date: 1.5+ years, 5 meetings, 70 members
  - Participants: Amdahl, Apple, AT&T, Borland, DEC, Glockenspiel, HP, IBM, Kubota, Microsoft, NCR, SCO Canada, Sun, Unisys, Zortec
  - Committee Goals: What they want to achieve
  - Committee Priorities: Their value system
- WG21
  - To Date: 0 years/meetings
  - Participants: Canada, France, Japan, USA Netherlands, Sweden
  - New Work Item Proposal: The Charter
  - First Meeting Agenda: June 18-19 Business

# What Is X3J16 Doing?

## X3J16 Committee Goals

- Base documents:
  C++ Reference Manual (Ellis & Stroustrup),
  ANSI X3J11 C Std,
  ISO C Std addendum (when final)
- Specify C++ program syntax and semantics without preprocessor references
- Specify a minimum set of C++ libraries
- Consider major extensions,
  e.g. parameterized types and exception handling
- Suitable for international community

# *What Is X3J16 Doing?*

## X3J16 Committee Goals

- Standardize C+ + environment elements
  - preprocessing, lexical analysis, startup,
  - termination, compatible implementations,
  - target & host environment differences, linkage,
  - freestanding & hosted implementation differences
- A high level of compatibility with ANSI X3J11 C
- Two deliverables: Draft proposed std & Rationale

SLC - 5/31/91

# *What Is X3J16 Doing?*

## X3J16 Committee Priorities

- Clear and unambiguous specification
- Compatibility with the C+ + Reference Manual
- Compatibility with the other base documents
- Consistency
- Favorable user and implementor experience
- Portability, efficiency, expressiveness
- Ease of implementation including translatability into C

SLC - 5/31/91

## *What Is WG21 Doing?*

C+ + New Work Item Proposal: The Charter

- Solicits input: formal specification, international character handling
- Goal: ISO and ANSI C+ + standards are the same
- June 1991: first WG21 meeting
- Early 1995: Register Committee Draft
- Ballots synchronized through X3J16 type I development
- C and C+ + differences documented according to ISO Technical Reference 10176
- USA provides C+ + WG convener
- X3J16 documents circulated in WG21

SLC - 5/31/91

## *What Is WG21 Doing?*

WG21 Business, June 18 - 19

- Resolve ballot comments
  - C+ + should be an amendment to the C standard in the sense that C+ + is a superset of C
  - Synchronization of X3J16 and WG21 documents must be improved/resolved
- Synchronization of technical expert efforts
- Establish meeting schedule

SLC - 5/31/91

## *How Will X3J16 and WG21 Work Together?*

Synchronized Efforts of Technical Experts

Several Possibilities

- WG21 contracts technical development to X3J16
  - one or two nearly identical documents
- WG21 absorbs all members of X3J16 and does own technical development
  - one document
- WG21 does own technical development with input from X3J16 representatives
  - probably two separate slightly different documents

SLC - 5/31/91

## *When Will C++ Standardization Complete?*

Milestones: 1989 and 1990

| | | |
|---|---|---|
| 9/89 | WG21 | USA requested to submit C++ proposal |
| 12/89 | X3J16 | first meeting |
| 3/90 | X3J16 | base documents: X3J11 C Std and ARM |
| 7/90 | X3J16 | – Templates model adopted<br>– One non-USA meeting/year endorsed |
| 11/90 | X3J16 | – Exception handling model adopted<br>– Overriding/Renaming rejected<br>– Consensus favors *iostream* and *string* classes as a minimum |

## *When Will C++ Standardization Complete?*

Milestones: 1991

| | | |
|---|---|---|
| 3/91 | X3J16 | Adopts second draft of document |
| 4/91 | JTC1 | Approves C++ Proposal |
| 6/91 | WG21 | first meeting |
| 6/91 | X3J16 | first meeting outside USA – Lund |

SLC - 5/31/91

## *When Will C++ Standardization Complete?*

Milestones: 1992 and beyond

| | | |
|---|---|---|
| 3/92 | X3J16 | second meeting outside USA – London |
| 93 | X3J16 | submits document for public review |
| 93 | WG21 | registers and circulates document (CD) |
| 94 | WG21 | document elevates to Draft Int'l Std |
| 95 | WG21 | document elevates to Int'l Std |

SLC - 5/31/91

# *C++ Standardization*

## Acronyms

| | |
|---|---|
| AFNOR | Association Française de Normalisation |
| ANSI | American National Standards Institute |
| ARM | *The Annotated C++ Reference Manual* |
| BSI | British Standards Institute |
| CD | Committee Draft (CD --> DIS --> IS) |
| DIS | Draft International Standard (DIS --> IS) |
| IEC | International Electrotechnical Committee |

# *C++ Standardization*

## Acronyms

| | |
|---|---|
| IEEE | Institute for Electrical and Electronics Engineers |
| IS | International Standard |
| ISO | International Standards Organization |
| JTC1 | Joint Technical Committee on Information Technology |
| OSI | Open Systems Interconnection reference model |
| SC | Study Committee |
| WG | Working Group |

## *How Will X3J16 and WG21 Work Together?*

USA Proposal: X3J16–WG21 Cooperation Plan

The USA (X3J16) proposes that WG21

- adopt the latest draft document accepted by X3J16 as the base document
- designate Jonathan Shopiro as WG21 technical editor
- assign technical development responsibility to X3J16
- refer all technical development proposals to X3J16
- hold joint meetings with X3J16

SLC - 5/31/91

## *How Will x3J16 and WG21 Work Together?*

USA Proposal: X3J16–WG21 Cooperation Plan

The USA (X3J16) agrees to

- accept and give serious consideration to technical recommendations from the WG21
- report on the disposition of all WG21 recommendations and requests
- report on the progress of technical development reflected in the document
- distribute USA (X3J16) working papers to WG21 heads of delegation for redistribution within their country as desired

SLC - 5/31/91

# *How Will x3J16 and WG21 Work Together?*

## USA Proposal: X3J16–WG21 Cooperation Plan

The USA (X3J16) agrees to

- distribute early versions of the document to WG21 heads of delegation. Distribute later versions of the document to the entire WG21 membership. Prior to CD registration, distribute the latest version of the document to the entire SC22 membership.
- attempt to hold USA (X3J16) meetings outside the USA annually.

SLC - 5/31/91

CD – Committee Document, ISO progression: CD ---> DIS ---> IS

# Implementing Multiple Inheritance in C++

Michael S. Ball

TauMetric Corporation

8765 Fletcher Parkway, Suite 301

La Mesa, CA 91942, USA

mike@taumet.com

16



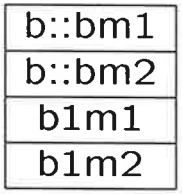
A simple class is the same as a C struct.

```
class b {    // no bases
    int bm1;
    int bm2;
};
```


```
struct b {
    int bm1;
    int bm2;
};
```

Base class fields go at the start of the derived class.

```
class b1: b {
    int b1m1;
    int b1m2;
};
```

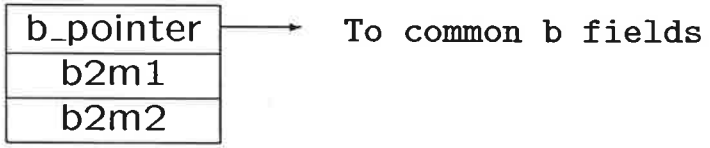



```
struct b1 {
    b b_fields;
    int b1m1;
    int b1m2;
};
```



Virtual base classes are allocated outside of the class and a pointer kept in the class.

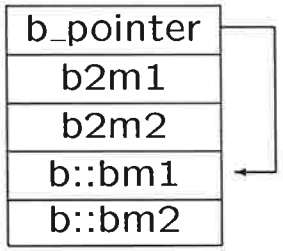
```
class b2: virtual b{
    int b2m1;
    int b2m2;
};
```




```
struct b2 {
    b* b_pointer;
    int b2m1;
    int b2m2;
};
```

When an instance of a `b2` is allocated, the space for the instance of a b is placed at the end of the structure. The result looks like:

```
struct b2 {
    b* b_pointer;
    int b2m1;
    int b2m2;
    b b_value;
};
```

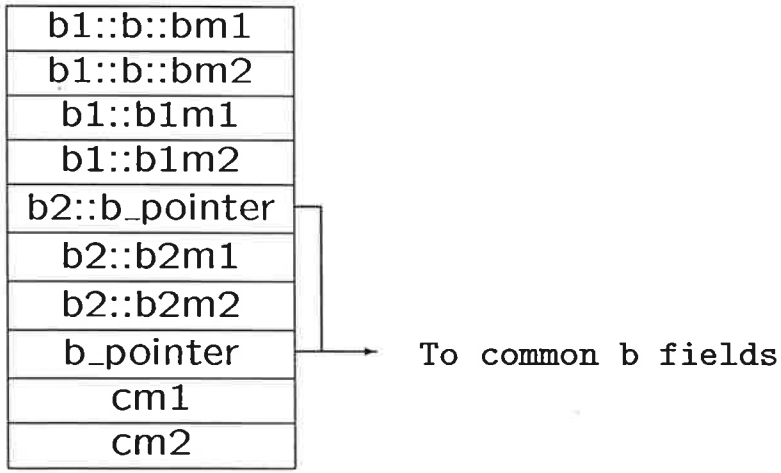



`b_pointer` will be set up in the constructor.



A final example shows it all

```
class c: b1, b2, virtual b {
    int cm1;
    int cm2;
};
```



Side note: casting a pointer from derived to base class is now a complex operation.

For the first base class it's still simple:

`(b1*)cp` becomes `(b1*)cp`

For a later class, we adjust the offset:

`(b2*)cp` becomes `&(cp->b2_fields)`

For a virtual base class:

`(b*)cp` becomes `cp->b_pointer`

Actually, a `NULL` must remain a `NULL`, so:

```
(b1*)cp becomes cp ? &(cp->b2_fields)
                   : (b1*) cp
```



Virtual functions are called through a vector associated with the actual type of an object. Each class has a virtual vector and each instance of a class has a pointer to that vector.

Each virtual function is assigned a number (virtual index).

```
class b {
    int i;          // a typical member
public:
    virtual f1();   // virtual index 0
    virtual f2();   // virtual index 1
    virtual f3();   // virtual index 2
};
```

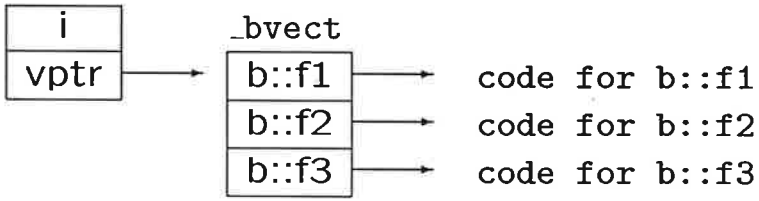
The virtual vector for `b` would look like

```
typedef int (*vvent)(); // func pointer
vvent _bvect[] =
    {b::f1, b::f2, b::f3};
```

The storage allocated for `b` is

```
struct b {
    int i;
    vvent *vptr;
};
```

The virtual pointer `vptr` would be initialized to point to `_bvect`.



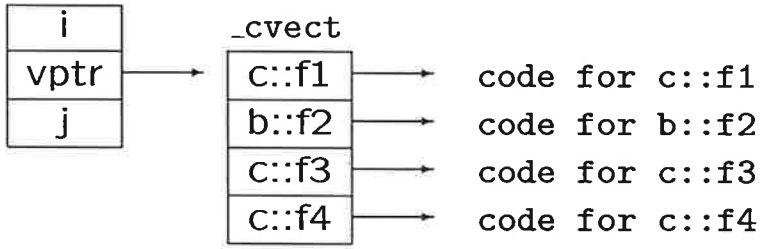



In a derived class

```
class c: public b {
    int j;
public:
    f1();           // virtual index 0
    f3();           // virtual index 2
    virtual f4();   // virtual index 3
};
```

Objects of type `c` have the virtual vector

```
vvent _cvect[] =
   {c::f1, b::f2, c::f3, c::f4};
```



A simple table lookup finds the correct virtual function.

```
b* bp;

bp->f3(); // virtual index 2
```

The call to f is translated to

```
(bp->vptr[2])(bp)
```

So if `bp` points to a `b` this calls `b::f3` and if it points to a `c` it calls `c::f3`.



This won't work with multiple inheritance.

```
class b1 {
    int k;
public:
    virtual f3();  // virtual index 0
    virtual f4();  // virtual index 1
    virtual f5();  // virtual index 2
    f6();    // no virtual index
};

class c1: public b, public b1 {
    int j;
public:
    f1();          // virtual index 0
    f3();          // virtual index 2
    f4();          // virtual index 3
};
```

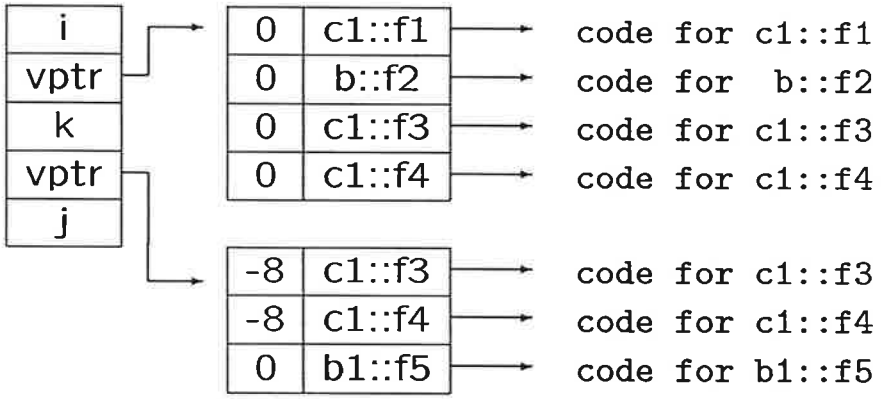
Problems:

- The fields of `b1` don't begin at the same location as the fields of `c1`, but a function member of `b1` expects `this` to point to the start of its fields.
- The virtual indices assigned in `b1` have no relationship to those assigned in `b`. `b1` needs a separate virtual vector.



The virtual vector entry becomes

```
struct vvent {
    int delta;    // adj. to "this"
    int (*func)();// function pointer
};
```




and the call becomes (conceptually)

```
(bp->vptr[vi].func)
     (bp+bp->vptr[vi].delta)
```

Multiple inheritance slows virtual calls even when not used. This penalty is unnecessary if we don't have to generate C code.

Generate single inheritance virtual vectors and a single inheritance call.

If `this` needs adjustment, generate code to do the adjustment and place its address in the vector. Such code is called a "thunk".

A code would look like:

```
thunk_ent:
    this += offset;
    goto routine;
```

which is, of course, not legal C.



## Code from 2 Compilers

```
 optimizing                    non-optimizing

Ordinary call:
  movl this,sp@-               movl this,d0
  jsr  routine                 movl d0,sp@-
                               jsr routine

Single Inheritance Virtual call
  movl this,a0                 movl this,d0
  movl a0,sp@-                 movl d0,sp@-
  movl a0@(0x8),a0             movl this,a0
  movl a0@(0x10),a0            movl a0@(0x8),a0
  jsr  a0@                     movl a0@(0x10),a0
                               jsr  a0@

Multiple Inheritance Virtual call
  movl this,a0                 movl this,a0
  movl a0,a1                   movl a0@(0x8),a0
  movl a0@(0x8),a0             movl a0@(0x20),d0
  addl a0@(0x20),a1            addl this,d0
  movl a1,sp@-                 movl d0,sp@-
  movl a0@(0x24),a1            movl this,a0
  jsr  a1@                     movl a0@(0x8),a0
                               movl a0@(0x24),a0
                               jsr  a0@

Stub, if needed
  addl #offset,sp@(0x4)        addl #offset,sp@(0x4)
  jmp  routine                 jmp  routine
```

Call Speed and Size Comparison

| call type | delta zero | optim | | non-optim | |
|---|---|---|---|---|---|
| | | size | refs | size | refs |
| simple | N/A | 12 | 10 | 14 | 11 |
| SI | N/A | 18 | 17 | 24 | 22 |
| MI (delta) | yes | 24 | 22 | 38 | 35 |
| MI (delta) | no | 24 | 22 | 38 | 35 |
| MI (stub) | yes | 18 | 17 | 24 | 22 |
| MI (stub) | no | 18 | 25 | 24 | 30 |

# Implementing the Dark Corners of C++

*Martin J. O'Riordan*
*Microsoft Corporation*

*(uunet!microsoft!martino)*

*Lund - Sweden*
*(14 June 1991)*

This paper briefly explores some of the more hidden parts of the C++ language, which are difficult to understand, recognise and implement. Much of what is presented here has been brought to light through the implementation efforts of the Microsoft C++ Compiler Development team, and represents the work of many people.

## Introduction

Since approximately April 1985, implementations of compilers for the C++ programming language have been available. Originally the AT&T translator '**cfront**' and subsequently implementations by other vendors. Since that time, the language has been evolving, and although the primary goals and mechanisms of the language are apparent, there are many nooks and crannies which have fallen into the shade. It is these nooks and crannies I wish to explore, rather than the regular and obvious functionality. And how these shady corners impact the design of object mapping, code generation, and compilation strategies in the implementation of a compiler. I will introduce each of these issues with a description of the problem, how the issue may be addressed, and where applicable, how it is the Microsoft model tries to resolve them.

# 1 The "One Definition Rule"

The "One Definition Rule" is a statement in the ARM, and the working draft document for the X3J16 ANSI standards sub-committee. The essence of this statement is that although a type or inline function may be defined in many translation units, they must all agree to the same actual definition, as if the whole was translated as a single unit. The concepts of "Type Safe Linkage", and "Inline Functions" rely on this axiom, but for different reasons. Type safe linkage allows multiple translation units to see a definition of a type, and where externally linked components involving such a type are shared, the definitions must agree exactly. Inline functions on the other hand, must be semantically indistinguishable from using a regular function.

These are very difficult goals to achieve, and at best, current implementations only approximate the ideal.

## 1.1 Type Safe Linkage

Since C++ uses name equivalence for types, it is possible for the definitions of a type of a given name to differ between translation units. Traditionally, such types are defined in header files which are included by each translation unit, and such differences occur rarely. But when they do occur, they are extremely difficult to detect.

Possible solutions could involve the exporting of some internal abstract representation of named types, which could be analysed by traditional linking tools for identity. However, in practice, such a mechanism is very expensive, and more typically, this level of type identity is ignored for performance of translation reasons. Instead, elaborate schemes of "Name Decoration (Mangling)" are required. These name decoration mechanisms work well provided the underlying types do indeed agree, but are incapable of enforcing such agreement.

Since these kinds of type mismatch are expected to be rare, the name decoration schemes are adequate. At Microsoft, the name decoration scheme is extended to cover all aspects of symbolic identity, except for verifying that the actual definitions of named types do indeed agree. However, by decorating the names of all externally linked names, including data variable names, it is possible to ensure that object names are of the same type, functions accept the same arguments types and return the same types, and that linkage specifications agree across translation units. Our particular environment needs are further aggravated by the need to support different memory models and function calling conventions.

Consider the following fragment of code :-

```
// Translation Unit #1

extern double p;
void bar () { p = 9.5; }


// Translation Unit #2

#include <stdio.h>

extern void bar ();
char* p = "Hello World\n";

int main ()
{
    printf ( p );
    bar ();
    printf ( p );   // What will this do ?
    return 0;
}
```

The behaviour of the above example is unpredictable. Since the external data variable '**p**' is considered to be of different types by the different translation units. Ideally, such a program would be rejected by the translation system, but often the reality of delivering a usable implementation results in some trade-offs. Indeed, current implementations of the C++ programming language do contain such prudent trade-offs.

There are two principal name decoration strategies that can be applied. Either decorate the name '**p**' or don't. If it is decorated, then in the absence of special linker support, there will be two names, one called '**f(double,p)**' and the other called '**f(char*,p)**', where '**f(t,n)**' is the decoration transformation function. Consequently, the data name is essentially overloaded, despite the fact that such overloading would be forbidden within the same translation unit. If the name is not decorated, then the fact that it is considered to be of different types in each translation unit will not be detected, and a different type of error can occur.

In the absence of linker support, it is probably better not to decorate the data names (and function return types), since the errors that can occur are familiar to the 'C' programmer. Decorating would produce a new class of error, with which the traditional 'C' programmer would not be familiar.

However, when linker support is available, decorating data variables and return types, is distinctly superior, since it is possible to detect the type mismatch, and produce an appropriate diagnostic. The approach adopted by Microsoft is to decorate all symbols with external linkage.

Ultimately, name decoration schemes are too complex, since it gets ever more expensive to encode information for such things as nested types, exception signatures, and parameterised type information. Local names also present a significant problem, especially when local static variables are involved in inline functions. Consider the following code fragment :-

```
inline int Function ()
{
    static int i = 9;
    // ...
    if ( condition )
    {
        static int i = 77;
        // ...
    }
    // ...
}
```

The principal problem is that the inline function may be expanded many times within a translation unit, and indeed possibly in several translation units. However, to adhere to the "*as if a real function*" semantics, the local name needs to have some form of reproducable external name. This name must be decorated to allow the two local names '**i**', to be distinguished from each other. Furthermore, the function '**Function**' may be overloaded, and each overloaded form may have its own local static variable '**i**', which must be distinguished from all other '**i**'s. This problem is further complicated when the initialisation of such local static variables is dynamic, requiring the use of some externally linked flag (or similar mechanism) to ensure once only initialisation. Similarly, if the local static has destruction semantics, further externally linked helper functions may be required.

Again, a name decoration scheme can be described to encompass these names in a reproducable fashion, but it is definitely stretching the practical limitations of achieving type safe linkage using any encoded naming technique.

One the bright side, most 'C' compilers support a rich set of information produced for the purposes of debugging. As C++ compilers become more common, they too will be enriched with information necessary for describing the complexities of names, overloading and the C++ types involved. This information is a valuable resource, not only for debugging, but also for enabling the development of stronger linkers, capable of interpreting this information, and producing comprehensive diagnostics of inter-module type violations and inconsistancies. The use of such information could eliminate the need for formal name decoration.

## 1.2 Inline Functions

These present a special problem to an implementation. Traditional implementations are capable of expanding inline functions that can be represented by expressions. This

makes representing inlines functions which embody loops, switch selections and resursion quite difficult. To get over these limitations, it is necessary to describe a function in some more abstract form, that can be readily incorporated into the abstract representation of a using function. That aside, even in the presence of extensive inlining capability, inline functions must sometimes be instantiated. That is, for one reason or another, the inline function must exist as a real function. This presents some difficult semantic problems for C++.

Typically, such inline functions are defined in an included header file. The one definition rule requires that all definitions of the inline function in separate translation units be the same definition. This is very difficult to achieve in practice, since traditional techniques involve the generation of an internally linked function, and current linking techniques are not generally capable of making any correlation between these inline function instantiations.

As described in section 1.1, local static variables in such inline functions can be cicumvented using a complex name encoding scheme, or type extended linking technology. However, the functions themselves can present problems. Firstly, there are the "gray-edges" as to what the one definition rule actually means, for instance :-

```
// Header file "type.h"
class X {
    int i;
    int j;
public:
    int bar () { return i; };
    int foo () { return j; };
};
extern int groan ( int ( X::* )(), X& );

// Translation Unit #1
#include "type.h"

int groan ( int ( X::*pmfX )(), X& rX ) {
    int temp;
    if ( pmfX == &X::foo ) {
        temp = ( rX.*pmfX )();
        pmfX = &X::bar;
    }
    else
        temp = 0;
    return temp;
}

// Translation Unit #2
#include "type.h"

main () {
    X localX;
    while ( groan ( &X::foo, localX ) == 0 )
        ;
}
```

Because the instantiations are local to each translation unit, the address of the two functions 'X::foo' are different, even though the intent is probably that they be the

same. This presents two problems. The multiple instantiations of a function intended to be the same function means that there is a semantic breach from the requirement that an inline function behave semantically as if it were a real function. Secondly, the multiple instantiations lead to a wasteful redundancy in the produced program. This example is contrived, but it does illustrate a problem with the multiple instance model for implementing inline function definitions.

By giving these types of functions 'external' linkage, multiple instances can be examined for identity by the "One Defintion Rule", and also coalesced to remove redundant copies. This leads to a greater match between programmer expectations and the reality of the implementation. Further more, the one definition rule can be enforced more easily. The ruling that inline functions get internal linkage is counter intuitive and is defined to permit restricted implementation models to be defined.

For 'external' linkage on inlines functions to be supported, a small amount of linker help is required, to recognise that a requested symbol has been resolved already, and that another occurence of the inline function need only be checked for identity with the one already found. However, when multiple instances are unavoidable, the problem is not great, since only address identity is compromised, and that is not a frequent use of such functions anyway.

## 2 Virtual Functions and Multiple Inheritance

Prior to multiple inheritance, virtual functions presented no real problem. All objects of a given type had a single address point, which was typically that of the first byte of the space allocated for the instance. The '**this**' address passed to virtual functions could only be the common address point, so the tables describing the set of virtual functions comprised simply of an array of the addresses of the virtual functions.

However, multiple inheritance introduces the idea that an object may have more than one address point. If a type has two or more base types, then it will have at least two possible address points. That of each of its base classes, and perhaps a new address point for itself (although not typically). If there is a difference between the address point of the deriving class, and that of a base class which introduces a given virtual function signature, and that function is over-ridden in the deriving class; then some form of adjustment between the address supplied when that function is called in the context of that base class, and the actual address expected by the over-riding function is required. For instance :-

```
struct Base1 {
        int b1;
virtual int foo ();
};

struct Base2 {
        int b2 ();
virtual int foo ();
};

struct Derived : Base1, Base2 {
        int d;
virtual int foo ();  // Over-rides both Base1::foo and Base2::foo
};
```

Instances of the type 'Derived' have at least two address points. The 'Base1*' address point and the 'Base2*' address point. It may also have an address point of its own, but typically this is the same as one of the base address points. If a Derived instance is used polymorphically in the context of 'Base1' and 'Base2', then at least one of the contexts must differ in address point to that of the 'Derived' type. Thus :-

```
void bar () {
    Derived dd;
    Base1*  pB1 = &dd;
    Base2*  pB2 = &dd;

    pB1->foo ();    // Calls 'Derived::foo()'
    pB2->foo ();    // Calls 'Derived::foo()'
```

Both applications call 'Derived::foo', but since there is only one 'Derived::foo', only one address point must reach that function. On either the 'pB1' application, or the 'pB2' application (or indeed both), an adjustment to the 'Derived*' address point is required.

Some implementations achieve this by encoding the adjustment in the table with the virtual functions, and the caller performs a computation involving this adjustment before providing the address point to the associated function. Other implementations use a special intermediate function called a '**thunk**' which performs the necessary adjustment to the '**this**' pointer before transfering control to the over-riding function. The advantage of the adjustor-thunk form, is that the thunk need only be supplied when the adjustment required is non-zero. It also reduces the complexity of the call site.

## 2.1 The Introducing Class

In an implementation which uses tables of functions (I will call them '**vftable**'s), it is desirable to eliminate this computation when the adjustment is zero. It is also desirable to increase the probability of these adjustments being zero. For this reason, I introduce the idea of "The Introducing Class". The introducing class is the class which first introduces a virtual function of a given signature to a deriving type. This is determined as being the class indicated by a transitive, left-to-right, pre-traversal of the inheritance DAG, as described by the declaration order of the base classes.

The introducing class defines the address point which will be anticipated by the over-riding virtual function. If this is not the address point of the deriving class, then the necessary adjustment from the anticipated address to the actual address point is known to the over-riding function at compile-time, and is typically folded into local member references made by that function. The advantage of this, is that for any given virtual function signature, the introducing path has always got a zero displacement to perform at the call.

The other non-introducing paths, have an adjustment made from the address supplied to that of the introducing class address point. The Microsoft implementation uses the thunk-adjustor approach, but due to the concept of the introducing class, the number of these thunks is reduced to a minumum since most adjustments will be zero. Furthermore, the degenerate single inheritance form always reduces to the simpler call sequence, allowing for the multiple inheritance solution to be implemented without losing anything in the single inheritance heirarchy.

# 3 Virtual Functions and Virtual Inheritance

Virtual inheritance is a form of multiple inheritance used to permit sharing of components of the inheritance DAG. The simplest configuration of classes which illustrates this sharing consists of three classes. This is described as follows :-

```
struct VBase {
    int v;
};
struct Base : virtual VBase {
    int b;
};
struct Derived : Base, virtual VBase {
    int d;
};
```

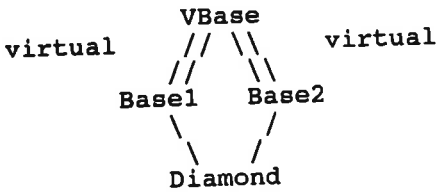
However, this example is not very illustrative of the virtual mechanism.

## 3.1 The Diamond

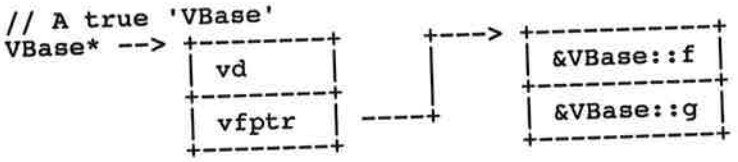
The most typical example of virtual inheritance is commonly known as the "Virtual Diamond". The diamond is an arrangement of four classes, whose DAG resembles a diamond. The diamond embodies all of the facilities of virtual inheritance and all of its problems, and is an excellent example for illustrating the mechanism (although not its application). The following is an example of the diamond which I will use for future reference and examples :-

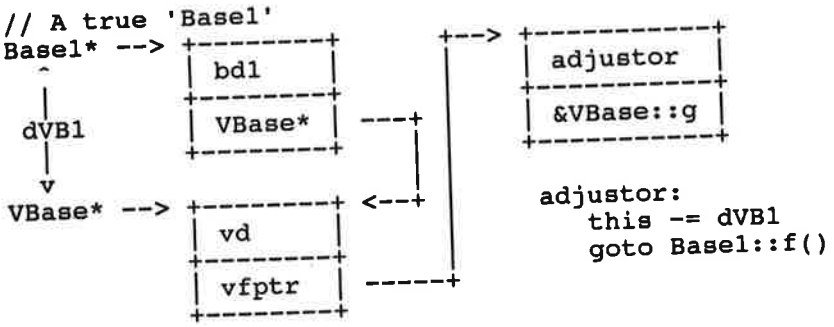
```
struct VBase {
    double  dv;
    virtual int f ();
    virtual int g ();
    VBase ();
};

struct Base1 : virtual VBase {
    double db1;
    virtual int f ();
    Base1 ();
};

struct Base2 : virtual VBase {
    double db2;
    virtual int f ();
    Base2 ();
};

struct Diamond : Base1, Base2 {
    double dd;
    virtual int f ();
    Diamond ();
};
```

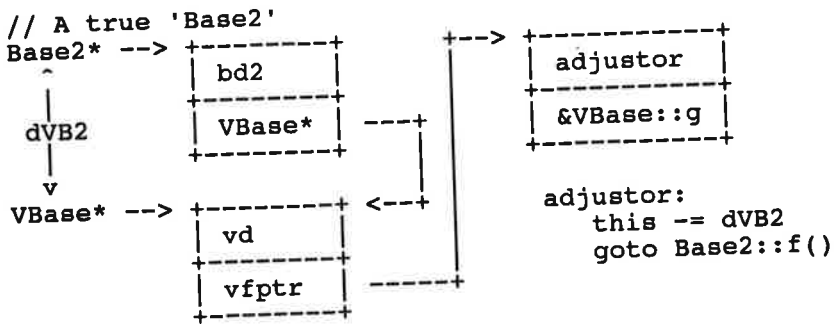
A graphic illustration of the DAG could be :-

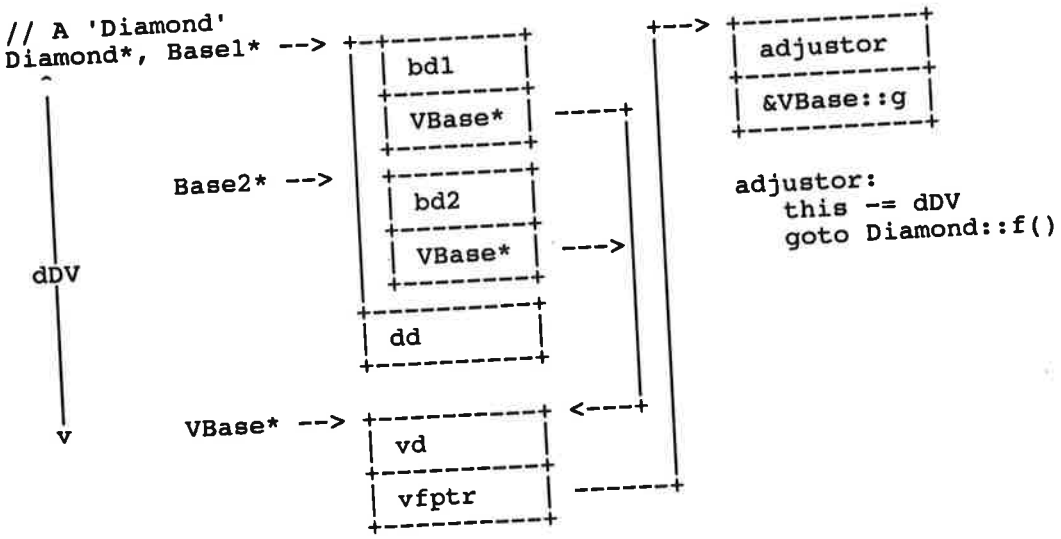



Virtual inheritance introduces a new complication to the DAG. Since there is only one occurence of the virtual base class embodied in the derived classes, the problem of "SHAPE" is introduced. SHAPE is the name I use to describe the way instances of a type are mapped. In the absence of virtual inheritance, the relative SHAPE of a class does not change. That is, the shape of an actual 'Base1' is no different to the relative shape of the 'Base1' occurrence in the derived class. Similarly, the shape of 'Base2'. However, when the class 'VBase' is virtually inherited by the classes 'Base1' and 'Base2', then the relative shape of the 'Base1' occurrence within the class 'Diamond' is not necessarily the same as that of a true 'Base1' object. Similarly, the shape of 'Base2'. Consider the following possible mapping for the diamond example :-






In this example, neither the shape of a 'Base1' as a base of a 'Diamond' the same as a true 'Base1', nor is the shape of a 'Base2' as a base of a 'Diamond' the same as that of a true 'Base2'.

This difference in shape is accounted for when the virtual methods are called. The call site knows to use the '**VBase***' hidden member to find the location of the virtual base part of the 'Diamond', and having found the 'VBase' part of the instance, it can locate the '**vfptr**' hidden member and select the entry in the table corresponding to the desired virtual function, in this case, element '**0**' for 'f()' and element '**1**' for 'g()'. The adjustor '**thunk**' relocates the address point as necessary for function 'f()'.

A refinement of the 'Introducing Class' can be used here too. The address expected may be that of the location a 'Diamond' would expect the 'VBase' part to be, so the adjustment in this case could be zero. Subsequently deriving classes which did not over-ride 'f()', would have a modified '**vftable**' which did contain an adjustor thunk to compensate for any change in the Diamond's SHAPE. The effect is again to increase the number of times the adjustment is zero, and hence reducing the number of computations and the number of adjustor thunks. The Microsoft implementation does extend the "Introducing Class" concept in this way.

## 3.2 Construction/Destruction and SHAPE

Normally the different shape of the base classes is not a concern, since the '**vftable**'s and adjustor thunks are set up to compensate for the altered layout of the actual instance. However, construction and destruction are a different story.

During construction, each base class is constructed before the derived class. While a base class is being constructed, it is considered to be an instance of that base type, and not the derived type. This is essential to prevent virtual functions over-ridden by the

derived class from performing operations involving as yet uninitialised bases or members of the derived class. Typically, this is achieved by each constructor loading the address of the '**vftable**' it would use for actual instances of the base type. As a consequence, virtual functions called indirectly by the constructor will be prevented from accessing uninitialised information.

However, the tables for actual bases do not accomodate the SHAPE change caused by subsequent derivations. For instance, an actual 'Base1' has an adjsutor which adjusts '**this**' by 'dVB1' before calling 'Base1::f', but an adjustment of 'dVD' is needed, since the position of the 'VBase' component is different. To illustrate, consider the following function definitions for the Diamond member functions :-

```
int VBase::g () { return f (); }

Base1::Base1 () { (void)g (); }
Base2::Base2 () { (void)g (); }
```

Irrespective of what the over-riding function 'f()' does, there is an immediate problem. The definition of 'Vbase::g()' is unaware of future derivations, and during the construction of the 'Base1' and 'Base2' parts of the 'Diamond', the polymorphic context is correct, the constructor finds the 'VBase' through the '**VBase***' hidden member. It then calls the method 'VBase::g()' using the 1th element of the virtual function table pointed to by the '**vfptr**' hidden member of the 'VBase' componenent.

However, once into the method 'VBase::g()', the context is lost. It no is longer aware of any deriving context, and knows only about 'VBase's. Thus when it calls 'f()', it passes it's own '**this**' address. The over-riding function (in this case, either 'Base1::f' or 'Base2::f') is expecting a different address point. But since the call occurs before the ultimately derived object is initialised, the '**vftable**' that is on-line is that of the intermediate base class, whose shape is different to that of the ultimate derived class. Consequently, the adjustment 'dVB1' or 'dVB2' is used, when the displacement 'dVD' is required.

Solving this transient construction problem (and its inverse during destruction), is not at all simple. One possibility is for each derived class to generate a flavour of the '**vftable**'(s) for the base classes, so as to compensate for the incorrect adjustment. This is difficult but possible. The costs are that some means of communicating the need to substitute a different '**vftable**' to the constructor is required. In a complex DAG, this involves the generation of some descriptor which associates the different phases of construction with the appropriately shaped 'vftable's. It also involves an '$N^2$' set of '**vftable**'s, where 'N' is the number of classes derived from the class which first over-rides a method introduced by a virtual base class. This is a very complicated and expensive resolution. Other solutions involving writable copies of '**vftable**'s suffer from difficulties with re-entrancy and also in the Intel mixed model environments.

### 3.2.1 *Construction Displacement*

An alternative approach to resolving this problem is the use of a displacement compensation value, which may be stored in the instance. This displacement is a value which describes the displacement from where a virtual base is actually mapped, to where the current context expects it to be. In the fully constructed object, this displacement is always zero. A displacement is associated with each virtual base class, and may be placed immediately preceding, or immediately following the actual occurence of the virtual base class. In the case of the Diamond, the ultimately derived constructor places the value 'dVD-dVB1' into the displacement field prior to calling the 'Base1' constructor. It then places the value 'dVD-dVB2' into the displacement field prior to calling the 'Base2' constructor.

Intermediate constructors which are not the ultimately derived constructor can perform an arithmetic adjustment of the value in the displacement field, by the amount they have shifted the relative position of the virtual base, before calling the constructors of classes which have that virtual base.

The virtual function tables for the classes which have a virtual base class place a '**thunk**' on each over-ridden virtual method of that virtual base's class, which retrieves the constructor displacement value, and adjusts the 'this' value by that, in addition to the static adjustment required. Thus, in the previous problem, when the 'Base2' constructor indirectly calls 'f()' though 'g()', it performs the displacement 'dVB2+(dVD-dVB2)', or simply 'dVD' which is the correct compensation for the altered shape of the instance.

Optimisations can be made to reduce the number of times this mechanism needs to be invoked. For instance, the displacement field is only required when a virtual method in the virtual base is first over-ridden, since the indicated possibility of the shape and the polymorphic context conflicting cannot exist. Furthermore, generated constructors and destructors are immune from the problem, since they do not call any methods.

Ideally, in an environment where complete program type knowledge is available, the displacement can be dispensed with, since the compensation can be made only when such a call tree can actually exist. However, for now and the immediate future, the separate compilation environment is a reality, and the constructor displacement solution provides a viable compromise. Further performance optimisations can be achieved using '**vftable**'s tuned to the ultimately-derived constructor and only having the displacement fetching form, on the non-ultimately-derived constructor. This potentially doubles the number of tables for the virtual base parts of an instance, but it is still a lot better than the N-squared solution.

[Note: The same, but inverse problem, exists during destruction]

# 4 Pointer to Members

Pointers to members are relatively complicated objects. For single inheritance, a pointer to data member need only know the displacement from the common address point to the member. A pointer to function member needs to know the address of the function to be called, or if it is a virtual function, the index into the '**vftable**' to where the function address can be found. Contravariant conversions involve no adjustment to the representation, and are purely changes to the type system.

## 4.1 Pointers to Members and Multiple Inheritance

In multiple inheritance, the situation is a bit more difficult. Again, a data member need remember only the displacement from the address point to the data member. If a contravariant conversion takes place, then the value of the displacement can be adjusted by a constant which represents the difference between the derived class' address point and that of the base class from which the member pointer is converted. Functions are more complex.

For non-virtual member functions, the displacement from the address point to that expected by the method is required. This is first added(or subtracted) to the given '**this**' address before calling the actual function. The address of the actual function is needed as for the single inheritance case. Thus a couple is required to represent a pointer to a non-virtual function member. The displacement field is essentially a pointer to a data member, as the address point is in effect a pointer to the first member (whether user named, or hidden) of the class or base class.

A pointer to a virtual member function involves more work. The displacement to the address point is required, just as for the pointer to the non-virtual function. The index of the virtual method in the '**vftable**' is also needed. Finally, a third field is required to determine the displacement from the expected address point to the corresponding '**vfptr**' hidden member. Contravariant conversions are as for the pointer to non-virtual member function, involving a simple arithmetic adjustment of the address point displacement field. Consider a possible 'C' mapping for such a pointer to member function :-

```
// It's shape, for example "void (T::*)()"
struct PointerToVirtualMember
{
    offset_t  thisAdj;
    offset_t  vfptrDisp;
    ptrdiff_t vIndex;
} pVMF;
```

```
// It's application, "expr" is the expression supplied left of the
// '.*' or '->*' operator, as in "((expr).*pVMF )();"

char* tmp = (char*)( expr );  // Prevent side effects

tmp += pVMF.thisAdj;     // Compute the address to pass as 'this'

( *( tmp + pVMF.vfptrDisp )[ pVMF.vIndex ])( tmp );  // Apply
```

Further difficulties need to be considered. For instance, a representation for a NULL pointer to member pointer needs to be determined. Also, since a pointer to member function can contain the address of both virtual and non-virtual function members, the 'PointerToMember' structure needs to discriminate in some way. The following section deals with a reduction in the complexity of pointers to members, and a consequent simplification in the representation of both NULL and the discrimination.

### *4.1.1 A Canonical Mapping for the 'vfptr'*

One of the fields needed to represent a pointer to a virtual member function, is the 'vfptrDisp' field, which describes the displacement from the expected address point to the '**vfptr**' member of the base class. One of the complexity reductions Microsoft adopted, was the convention of always mapping the '**vfptr**' as the first member of a class, and making the address of the '**vfptr**' the address point for that class. Thus, the displacement from the address point to the '**vfptr**' member is always zero, and the 'vfptrDisp' field may be eliminated. Thus, the composite pointer to member function representation is reduced to a couple. The displacement to the address point of the expected class, and either an index or a function pointer address.

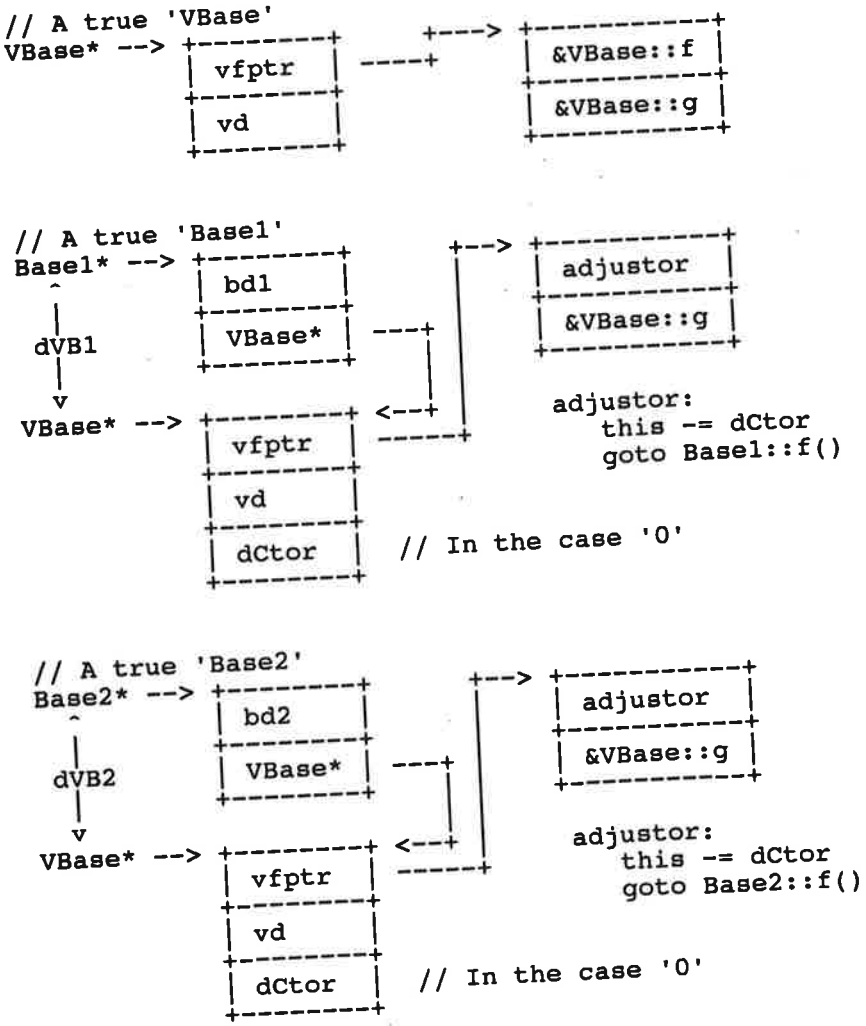
A further advantage in using a canonical location for the 'vfptr', is that the calling of the '$N^{th}$' virtual method for any type can be expressed by a canonical sequence, thus :-

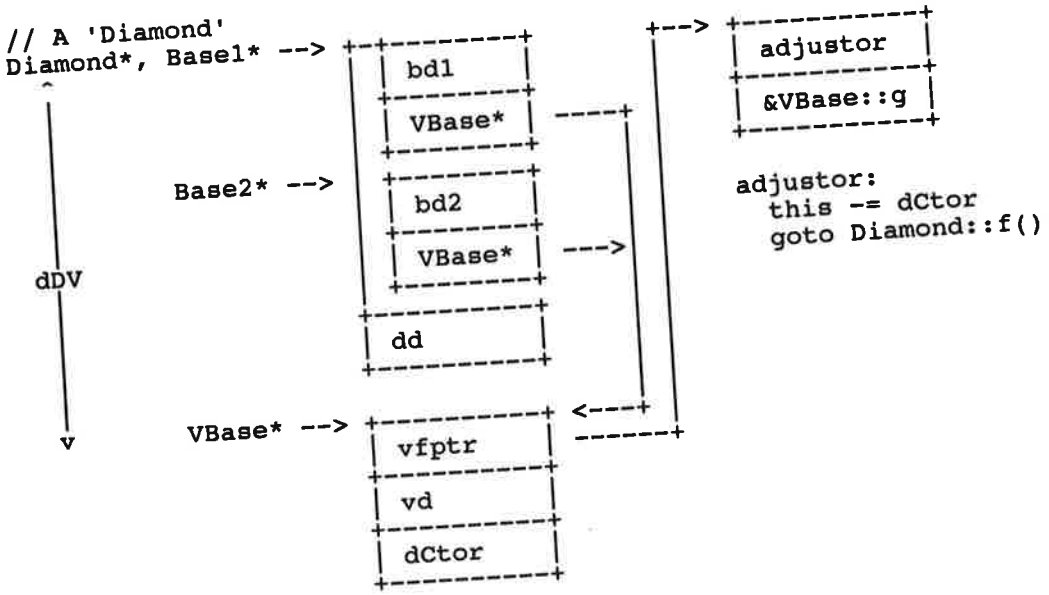
```
char* tmp = (char*)( expr );

( *( *tmp )[ N-1 ])( tmp, args );
```

For pointers to virtual methods, this means that a '**thunk**' can be created for calling a method at any particular index, and that '**thunk**' can be used by all types. Then, instead of storing the index in the pointer to member function structure, just the address of the thunk is required. Thus, the discrimination occurs automatically between virtual and non-virtual methods, and only a single form of pointer to member function is required.

## 4.2 Pointers to Members and Virtual Inheritance

This is a domain of considerable complexity and discussion by the X3J16 standards sub-committee. The problem has to do with the difficulty in modelling such pointers to members. Opinion is generally based on the implementation models in current use, and the complexity in accomodating pointers to members, and contravariant conversions when a given pointer to member could contain the address of a member in a virtual base class. Before continuing, I will re-describe the mapping of the "Diamond" using the canonical mapping for the '**vfptr**', as described earlier. I will also accomodate the "Introducing Class" changes to alter the adjustor values, and represent the "Constructor Displacement" field for SHAPE correction during construction. Note that the position of the hidden member '**dCtor**' may be mapped either immediately before, or immediately after the mapping of the virtual base to which it is bound. There are no advantages specific to either, so it is just a matter of convention. The revised mapping for the "Diamond" example is as follows :-






Consider first, pointers to data members. A pointer to a data member of 'VBase' need only consist of the displacement from the address point of a 'VBase' to the appropriate member. However, consider now a pointer to a member of a 'Base1' object. If the pointer is to the member 'Base1::bd1', then a displacement from the 'Base1' address point to the member is all that is needed. But, the member 'VBase::vd' is also a data member of a 'Base1' object, and the member pointer could just as easily refer to that member. Since the 'VBase' part of a 'Base1' is movable (through subsequent derivation), a displacement alone cannot be used. To represent this kind of pointer to member, a displacement from the address point of a 'Base1' to the appropriate virtual base class pointer (hidden member), is required. In addition, the displacement from the VBase address point to the member 'VBase::vd' is needed. The contravariant conversion from a 'VBase::*' to a 'Base1::*' is just a matter of extending the pointer to member type to include the displacement to the hidden member which contains the pointer to the virtual base.

This can of course be optimised by recognising that the class has only one virtual base, and requiring that the translator supplies the displacement upon application. However there are greater problems. If the virtual base class has itself got a virtual base class, then the representation must contain a displacement to the hidden pointer to the immediate virtual base class. An offset to the hidden pointer in the immediate virtual base class to the hidden pointer contained therein, to the indirect virtual base class. And in addition, contain the actual displacement to the appropriate member. Contravariance is achieved at each level by adding a new field for each level of virtual base class. This is an unbounded solution for pointers to members.

Pointers to function members of virtual base classes are similar, since they are essentially a composite of a function address, and a pointer to data member which refers to the address point required by the associated function.

### 4.2.1 *Bounding Virtual Base Classes*

The "unboundedness" of the problem stems from the potential cascading of virtual base pointers in the heirarchy. This can be eliminated by modifying the model in two ways.

First, instead of incorporating a hidden member for each virtual base class, a single pointer to a table containing displacements to the virtual bases can be used. This is a lot like the virtual function table, except that instead of containing pointers to functions, it contains virtual base displacements. This table called the '**vbtable**' is shared by all instances of the given type, just as the '**vftable**' is shared for all instances of a given type. Similarly, as new virtual base classes are introduced during subsequent inheritance, the displacements to the new virtual bases are appended to one of the '**vbtable**'s already present, just as the addresses of new virtual functions are traditionally appended to one of the '**vftable**'s. This has the effect of improving sharing and reducing the per-instance cost of supporting virtual base classes, as a single pointer is all that is required of an instance when more than one virtual bases class is introduced. It also reduces the complexity of initialising and destructing instances which contain virtual base classes, since no address arithmetic is involved, just the loading of the '**vbptr**' by the constructor for the ultimately derived class.

The second change to the model is to introduce redundancy into the nearest '**vbtable**'. This means that for the '**vbtable**' most often used by the deriving class (the one chosen for appending new virtual base displacements), has extended information to allow for better performance when accessing indirect virtual bases. Essentially it goes like this. For each virtual base, ensure that there exists an entry which contains a direct displacement to that virtual base. Now at a small static cost, the entire set of virtual base classes, whether direct or indirect, may be accessed using a single table lookup. This results in a constant performance cost for virtual inheritance, as well as bounding the accessing of members of virtual base classes.

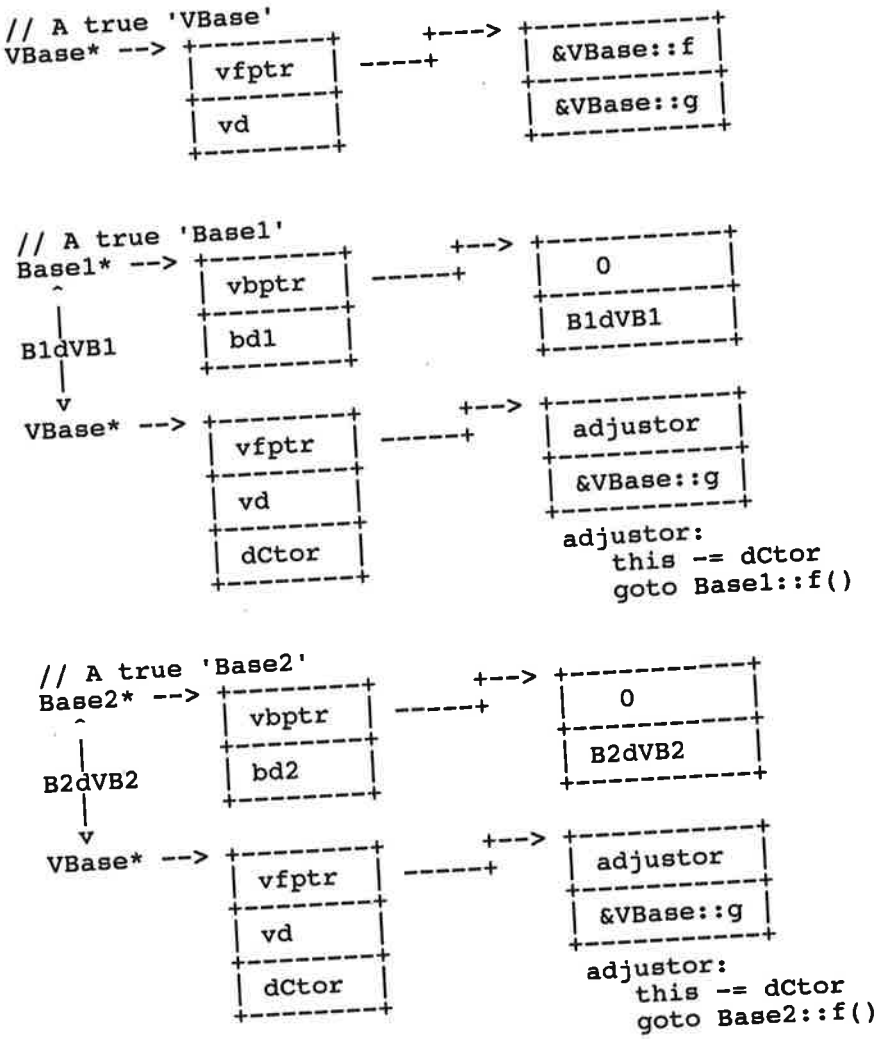
### 4.2.2 *How it Applies to Pointers to Members of Virtual Base Classes*

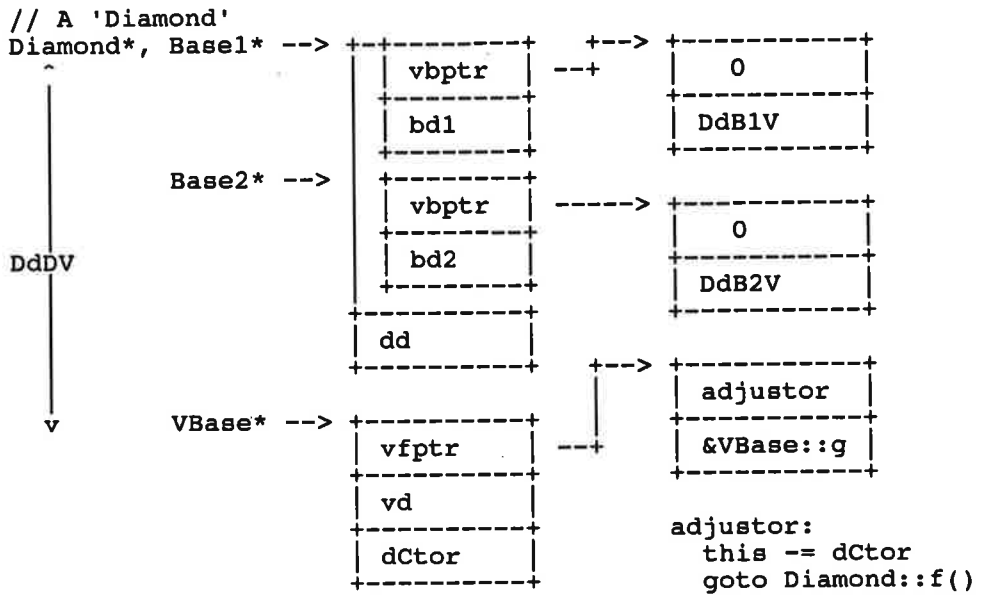
But how is all of this pertinent to the bounding of pointers to members. Well first of all, instead of a new field being added for each level of virtual-base of virtual-base'ness, only a single displacement field is needed, and a single index into the '**vbtable**' to the associated displacement. The complexity of application is also bounded. A further simplification arises from the fact that for any given type, only one '**vbptr**' is needed to access all displacements to virtual bases, and this has a location which is known statically at compile time. For this reason, the displacement to the '**vbptr**' field can be omitted. Thus, the representation of a bounded pointer to data members involving virtual bases, is reduced to a couple. A '**vbtable**' index, and the displacement from that address point to the

appropriate member. Similarly, the bounded pointer to function member is reduced to its pointer to data member component, and the function address itself.

However, there are still some complications. Since the pointer to member may contain the address of a member in the non-virtual part of the class, no displacement is needed. For this reason, the index zero is reserved, and the entry in the **'vbtable'** at location zero is itself always zero. This results in uniform application of a pointer to member.

Contravariant conversions now require the presence of a small lookup table to perform the mapping between the representation in the base class, and its possibly different representation in the derived class. Yet another revision of how the 'Diamond' might be mapped will reveal the purpose of these alterations to the model.






Note, the name 'DdB1V' describes the difference between where the 'VBase' address point is in a 'Diamond' with respect to the 'Base1' address point, versus where it would be in a true instance of a 'Base1'. Similarly, 'DdB2V' with respect to 'Base2'.

Note also, that the constants 'B1dB1V' and 'B2dB2V' represent where the 'VBase' part of a 'Base1' or a 'Base2' respectively, are with where they would be expected to be in a true 'Base1' or 'Base2'. Since these are truelly 'Base1's and 'Base2's, these values are actually zero.

In the example of the diamond, there is only one virtual base, so the illustration shows the mechanism, but not the sharing. Also, in this case, since the range of possible value for pointers to members of the base classes 'Base1' and 'Base2', the contravariant conversion to a 'Diamond::*' does not require a lookup table, since the '**vbtable**' index mapping is the same. However, if the virtual base index part is zero, then a regular displacement must be done in the case of converting a 'Base2::*' to a 'Diamond::*', just as for simple multiple inheritance.

### *4.2.3 Why Not Combine the 'vbtable's and the 'vftables' ?*

One of the questions I am most often asked, is "Why not combine the two tables?". It would be very desirable to combine these two tables, since they do not change during the active life of an object, and the overhead of tables and hidden pointers could be further reduced if they were combined.

Certainly, during the time after construction and before destruction, a single unified table could be used, since the correct shape and virtual function sets are

static for the object. However, during the construction and destruction phases of an object, this is not true. While any given base class is being constructed or destructed, it is polymorphically considered to be of the type statically indicated by the constructor or destructor, and not of any type derived from it. Thus the set of virtual functions must be that of the type of the base class. But the SHAPE of the instance is that of the derived class, and not of the type indicated by the static type of the constructor or destructor.

For this reason, the SHAPE is maintained by a separate table to that which determines its polymorphic behaviour. Since the shape is established by the ultimately derived constructor, prior to the construction of any bases or members, the problem is immediately eliminated. The constructors for the base classes will not change the shape, but will bind to the tables which correctly describe their behaviour. In the earlier illustration of the indirect activation of virtual methods during construction and destruction, this dichotomy between shape and behaviour was examined, and the construction displacement hidden member was added to address the problem. The use of separate a '**vftable**' and '**vbtable**' eliminates the same type of problem, but in the direct case rather than the indirect case.

### *4.2.4 Pointers to Members and Signatures*

Further optimisation for pointers to members may be achieved. The most startling of these is using a "signature" based pointer to member representation. Since pointers to members are not subject to type conversions (explicit or implicit) other than regular contravariance, there is no way they can ever contain the address of anything other than that of a member which matches their signatures. For instance :-

```
struct X
{
    int i;
};
float X::*pmf;
```

In this case, the class 'X' has no members of type 'float', so the declaration cannot be initialised with the address of any member of 'X'. Thus, it can only have two possible states. The NULL state, and an undefined state. Since the undefined state could also be the NULL state, no representation is needed for such a pointer to member. This represents a very special case of pointer to member. However, consider the following example :-

```
struct X
{
    int i;
};
int X::*pmi;
```

This is a little less degenerate than the last example. In this case, there are only three states that the pointer to member can contain. The two legal states are the NULL state, and the address of 'X::i'. The undefined state can be arbitrarily mapped to either of the two legal states, allowing the pointer to member to be represented by a single boolean value.

The above examples are purely academic, and are unlikely to find such an implementation in reality. However, the same idea of determining the most suitable representation for a member pointer can be used to reduce the representation of a pointer to member from the most general, to something more specific, from a set of possible pointer to member representations. For instance, consider the following class declarations :-

```
struct VBase {
    int   vi;
    float vf;
};

struct Base1 : virtual VBase {
    int   b1i;
    long  b1l;
};

struct Base2 : Base1 {
    long b2l;
};

int   Base2::*pmInt;
float Base2::*pmFloat;
long  Base2::*pmLong;
```

The example illustrates three different pointers to members of the same class. Each of these pointers to members is different in type signature. Each can benefit from a different representation.

- The first pointer to member '**pmInt**' could point to 'VBase::vi' or 'Base1::b1i'. In this case, the pointer to member must presume the worst case, and a representation involving the virtual base lookup is required.

- The second pointer to member '**pmFloat**' can only point to the member 'VBase::vf', so although it is a member of the virtual base, this knowledge can be inferred from the signature, and the pointer to member representation need only store the displacement to the member within the 'VBase' part.

- Finally, the pointer to member '**pmLong**' can only point to members within the non-virtual part of the class, and thus the representation does not need to contain information pertinent to accessing members in a virtual bases class.

These are only examples of the kinds of reduction of representation that could be performed, and many more exist. Of course these types of reduction in representation are accompanied by an increase in the complexity of the translator, since multiple representations of NULL may be required. Multiple representations of pointers to members are required. And a significant amount of inferred type specific knowledge is needed. But the performance gains of such increases in translator complexity could be warranted by applications which heavily utilise the pointer to member paradigm.

# Sixteen Ways to Stack a Cat

*Bjarne Stroustrup*

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

*ABSTRACT*

This paper presents a series of examples of how to represent stacks in a program. In doing so it demonstrates some of the fundamental techniques and tradeoffs of data hiding as seen in languages such as C, Modula2, and Ada. Since all the examples are written in C++ it also demonstrates the flexibility of C++'s mechanisms for expressing data hiding and access.

## 1 Introduction

Consider representing a stack of objects of some type in a program. Several issues will affect our design of the stack class: ease of use, run time efficiency, cost of recompilation after a change. We will assume that messing around with the representation is unacceptable so that data hiding for the representation is a must. We will assume that many stacks are necessary. The type of the elements on the stacks is of no interest here so we will simply call it `cat`. The implementations of the various versions of stacks are left as an exercise for the reader.

Please note that the purpose is to illustrate the diversity of data hiding techniques, not to show how best to represent stacks in C++. The techniques shown here apply to a variety of types – most of which are more interesting than stacks – and will be used in conjunction with other techniques. For example, if you actually wanted to build a better stack for C++ you would probably start by making `cat` a parameter, that is, use templates†.

This paper is a fairly lighthearted play with C++ features and techniques. I think it has something to delight and possibly horrify novices and seasoned C++ programmers alike. Most of the techniques shown have serious uses, though.

## 2 Files as Modules

Consider the traditional C notion of a file as a module. First we define the interface as a header file:

```
// stack interface, stack.h:

typedef int stack_id;

stack_id create_stack(int);
void destroy_stack(stack_id);

void push_stack(stack_id,cat);
cat pop_stack(stack_id);
```

The integer argument to `create_stack` is the maximum size of the desired stack. We can now use stacks like this:

† See Chapter 14 of Ellis&Stroustrup: The Annotated C++ Reference Manual. Addison Wesley. 1990.

```
#include "stack.h"

void f(int sz, cat kitty)
{
        stack_id s = create_stack(sz);
        push_stack(s,kitty);
        cat c2 = pop_stack(s);
        destroy_stack(s);
}
```

This is rather primitive, though. Stacks are numbered, rather than named; the concept of the address of a stack is ill defined; copying of stacks is undefined; the lifetimes of stacks are exclusively under control of the users; the technique relies on the convention of `.h` files and on comments to express the concept of a stack; the names of the stack functions are clumsy.

From a C++ perspective, the problem is that there are no stack objects defined. These stacks do not obey the general language rules for naming, creation, destruction, access, etc. Instead, little "cookies" (the `stack_ids`) are passed to functions that manipulate stack representations.

Note that in many contexts it would be reasonable not to impose a maximum size on a stack. Not imposing a maximum would allow a noticeable simplification of the stack interface. However, this would decrease the value of the stack example as a vehicle for discussion of general data hiding issues because most types do require arguments to the operations that create objects.

## 3 Stack Identifiers

We can do a little better. First let us make `stack_id` a genuine type:

```
// stack interface, stack.h:

struct stack_id { int i; };

stack_id create_stack(int);
void destroy_stack(stack_id);

void push_stack(stack_id,cat);
cat pop_stack(stack_id);
```

This at least will prevent us from accidentally mixing up stack identifiers and integers:

```
#include "stack.h"

void f(int sz, cat kitty)
{
        stack_id s = create_stack(sz);
        push_stack(s,kitty);
        cat c2 = pop_stack(sz); // error: stack_id argument expected
        destroy_stack(s);
}
```

This error would not have been caught by the compiler given the first definition of `stack_id`. These first two versions both have the nice property that the implementation is completely hidden so that it can be changed without requiring recompilation of user code as long as the interface remains unchanged.

## 4 Modules

Now let us use a class to specify this stack module. Doing that will allow us to avoid polluting the global name space and relying on convention and comments to specify what is and isn't part of a stack's interface:

```
// stack interface, stack.h:

class stack {
public:
        struct id { int i; };

        static id create(int);
        static void destroy(id);

        static void push(id,cat);
        static cat pop(id);

};
```

We can use this module like this:

```
#include "stack.h"

void f(int sz, cat kitty)
{
        stack::id s = stack::create(sz);
        stack::push(s,kitty);
        cat c2 = stack::pop(s);
        stack::destroy(s);

}
```

This looks very much like a Modula-2 module. We don't have a 'with' or 'use' construct to avoid the repetition of "`stack::`". For example:

```
void f(int sz, cat kitty)
{
        use (stack) {                          // pseudo code
                id s = create(sz);
                push(s,kitty);
                cat c2 = pop(s);
                destroy(s);

        }
}
```

However, such a construct for merging name spaces is a syntactic detail and does not always lead to more readable code. Further, an even more radical simplification of the notation is achieved in section 10.

Note that `static` member functions were used to indicate that the member functions do not operate on specific objects of class `stack`; rather, class `stack` is only used to provide a name space for `stack` operations. This will be made explicit below.

## 5 Modules with Sealed Identifiers

To make the correspondence to Modula-2 modules more exact, we need to stop people from messing around with the stack identifiers and stop them from trying to create (C++ style) stack objects:

```
// stack interface, stack.h:

class stack {
public:
        class id {
                friend stack;
        private:
                int i;
        };

        static id create(int);
        static void destroy(id);

        static void push(id,cat);
        static cat pop(id);
private:
        virtual dummy() = 0;
};
```

The representation of an *id* is now accessible only to class `stack`, "the implementation module for `stacks`," and class `stack` is an abstract class so that no objects of class `stack` can be created:

```
stack x;  // error: declaration of object of abstract class stack
```

The use of a pure virtual function to prevent the creation of objects is a bit obscure, but effective. An alternative technique would have been to give `stack` a private constructor.

Naturally, the example of stack usage from section 4 works exactly as before.

## 6 Packages

In the style of Modula-2, we are now passing around "little cookies" (opaque types) used by the implementation module to identify the objects. If we want we can pass around the objects themselves (or pointers to them) in the style of Ada:

```
// stack interface, stack.h:

class stack {
public:
        class rep {
                friend stack;
        private:
                // actual representation of a stack object
        };

        typedef rep* id;

        static id create(int);
        static void destroy(id);

        static void push(id,cat);
        static cat pop(id);
private:
        virtual dummy() = 0;
};
```

The typedef is redundant, but it allows our user code to remain unchanged:

```
#include "stack.h"

void f(int sz, cat kitty)
{
        stack::id s = stack::create(sz);
        stack::push(s,kitty);
        cat c2 = stack::pop(s);
        stack::destroy(s);
}
```

That `rep` is very much like an Ada private type. Users can pass it around but not look into it.

One disadvantage is that by placing the representation of stacks in the declaration of class `stack` we force recompilation of user code when that representation changes. This can lead to a major increase in compilation time. Actually, this recompilation isn't necessary, because the user code *and* that code's use of information about the implementation was left unchanged. A reasonably smart dependency analyser could deduce that no recompilation of user code is necessary after even a radical change to the representation. However, a dumb (that is, time stamp on source-file based) dependency analyser will not deduce that and will recompile user code just because the representation was changed. A smart dependency analyser will rely on smaller units of change, on understanding of the semantics of C++, or on both to minimize recompilation. Smart dependency analysers are rumored to exist, but they are not widely available.

On the positive side, the representation information is now available to the compiler when compiling user code so that genuine local variables can be declared and used:

```
#include "stack.h"

void g(int sz, cat kitty)
{
        stack::rep s;
        stack::push(&s,kitty);
        cat c2 = stack::pop(&s);
}
```

This is unlikely to work without additional code in the implementation, though, unless some mechanism for initializing a stack representation exists. This could be done by adding suitable constructors and destructors to class `rep`, but it would be more in the spirit of packages to provide an explicit initialization function. It would also be natural to support use reference arguments to eliminate most explicit pointers.

```
// stack interface, stack.h:

class stack {
public:
        class rep {
                friend stack;
                // actual representation of a stack object
        };

        static rep* create(int);
        static void destroy(rep&);

        static void initialize(rep&,int);
        static void cleanup(rep&);

        static void push(rep&,cat);
        static cat pop(rep&);
private:
        virtual dummy() = 0;
};
```

The `create()` function is now redundant (a user can write one without special help from the class), but I have left it in to cater for code and coding styles that relies on it. If needed, it could be made to

return a reference instead of a pointer.

```
#include "stack.h"

void h(int sz, cat kitty)
{
        stack::rep s;
        stack::initialize(s,sz);
        stack::push(s,kitty);
        cat c2 = stack::pop(s);
        stack::cleanup(s);
}
```

The `cleanup()` operation is needed because the `initialize()` operation and/or the `push()` operation are likely to grab some free store to hold the elements. Unless we want to rely on a garbage collector we must clean up or accept a memory leak.

Now enough information is available to the compiler to make inlining of operations such as `initialize()`, `push()`, and `pop()` feasible even in an implementation with genuine separate compilation. The definitions of the functions we want inlined can simply be placed with the type definition:

```
// stack interface, stack.h:

class stack {
public:
        class rep {
                friend stack;
                // actual representation of a stack object
        };

        // ...

        static cat pop(rep& x)
        {       // extract a cat from the representation of stack x
                // return that cat
        }

        // ...
};
```

Inlining and genuine local variables can be essential to make data hiding techniques affordable in applications where run time efficiency is at a premium.

## 7 Packages with Controlled Representations

Alternatively, if we did not want users to allocate `rep`s directly we could control the creation and copying of `rep`s by making these operations private:

```
// stack interface, stack.h:

class stack {
public:
        class rep {
                friend stack;

                // actual representation of a stack object

                rep(int);                       // constructor
                rep(const rep&);                // copy constructor
                void operator=(const rep&); // assignment operator
        };

        static rep* create(int);
        static void destroy(rep&);

        static void initialize(rep&,int);
        static void cleanup(rep&);

        static void push(rep&,cat);
        static cat pop(rep&);
private:
        virtual dummy() = 0;
};
```

This ensures that only the `stack` functions can create `rep`s:

```
stack::rep* stack::create(int i)
{
        rep* p = new rep(i);    // fine: create is a member of stack
        // ...
        return p;
}

f()
{
        stack::rep s(10); // error: f() cannot access rep::rep(): private member
}
```

Naturally, the example of stack usage from section 6 works exactly as before.

## 8 Packages with Implicit Indirection

If we are not interested in inlining but prefer to minimize recompilation costs even when we don't have a smart dependency analyser, we can place the representation "elsewhere:"

```
// stack interface, stack.h:

class stack_rep;

class stack {
public:
        typedef stack_rep* id;

        static id create(int);
        static void destroy(id);

        static void push(id,cat);
        static cat pop(id);
private:
        virtual dummy() = 0;
};
```

This scheme keeps implementation details not only inaccessible to users but also out of sight. With this definition (alone) a user cannot allocate `stack_reps`. Unfortunately C++ does not allow you to define a class "elsewhere" *and* have its name local to another class. Consequently, the name `stack_rep` must be global.

The indirection (implied by the use of `id`) is implicit to the users of stacks and explicit in the implementation of stacks.

## 9 Simple Minded Types

One simple improvement would be to put the operations that create and destroy `stack_reps` into class `stack_rep`. Actually, for a C++ programmer it would be natural to put all the operations into the `rep` and rename it "`stack`:"

```
// stack interface, stack.h:

class stack {
        // actual representation of a stack object
public:
        typedef stack* id;

        static id create(int);
        static void destroy(id);

        static void initialize(id,int);
        static void cleanup(id);

        static void push(id,cat);
        static cat pop(id);
};
```

so that we can write:

```
#include "stack.h"

void f(int sz, cat kitty)
{
        stack s;
        stack::initialize(&s,sz);
        stack::push(&s,kitty);
        cat c2 = stack::pop(&s);
        stack::cleanup(&s);
}
```

The redundant use of the typedef ensures that our original program still compiles:

```
#include "stack.h"

void g(int sz, cat kitty)
{
        stack* p = stack::create(sz);
        stack::push(p,kitty);
        cat c2 = stack::pop(p);
        stack::destroy(p);
}
```

The likely difference between `f()` and `g()` is two memory management operations (a `new` in `create` and a `delete` in `destroy`).

## 10 Types

Now all we have to do is to make the functions non-static and give the constructor and destructor their proper names:

```
// stack interface, stack.h:

class stack {
        // actual representation of a stack object
public:
        stack(int size);
        ~stack();

        void push(cat);
        cat pop();
};
```

We can now use the C++ member access notation:

```
#include "stack.h"

void f(int sz, cat kitty)
{
        stack s(sz);
        s.push(kitty);
        cat c2 = s.pop();
}
```

Here we rely on the implicit calls of the constructor and destructor to shorten the code.

## 11 Types with Implicit Indirection

If we want the ability to change the representation of a stack without forcing the recompilation of users of a stack we must reintroduce a representation class `rep` and let `stack` objects hold only pointers to `rep`s:

```
// stack interface, stack.h:

class stack {

        struct rep {
                // actual representation of a stack object
        };

        rep* p;
public:
        stack(int size);
        ~stack();

        void push(cat);
        cat pop();
};
```

The indirection is invisible to the user.

Naturally, to take advantage of this indirection to avoid re-compilation of user code after changes to the implementation we must avoid inline functions in `stack`. If our dependency analyser is dumb we might have to pace representation class `rep` "elsewhere" as was done in section 8 above.

## 12 Multiple Representations

In all of the examples above, the binding between the name used to specify the operation to be performed (e.g. `push`) and the function invoked were fixed at compile time. This is not necessary. The following examples show different ways to organize this binding. For example, we could have several kinds of stacks with a common user interface:

```
// stack interface, stack.h:

class stack {
public:
        virtual void push(cat) = 0;
        virtual cat pop() = 0;
};
```

Only pure virtual functions are supplied as part of the interface. This allows stacks to be used, but not created:

```
#include "stack.h"

void f(stack& s, cat kitty)
{
        s.push(kitty);
        cat c2 = s.pop();
}
```

Since no representation is specified in the stack interface, its users are totally insulated from implementation details.

We can now provide several distinct implementations of stacks. For example, we can provide a stack implemented using an array

```
// array stack interface, astack.h:

#include "stack.h"

class astack : public stack {
        // actual representation of a stack object
        // in this case an array
        // ...
public:
        astack(int size);
        ~astack();

        void push(cat);
        cat pop();
};
```

and elsewhere a stack implemented using a linked list:

```
// linked list stack interface, lstack.h:

#include "stack.h"

class lstack : public stack {
        // actual representation of a stack object
        // in this case a linked list
        // ...
public:
        lstack(int size);
        ~lstack();

        void push(cat);
        cat pop();
};
```

We can now create and use stacks:

```
#include "astack.h"
#include "lstack.h"

void g()
{
        lstack s1(100);
        astack s2(100);

        cat ginger;
        cat blackie;

        f(s1,ginger);
        f(s2,snowball);
}
```

## 13 Changing Operations

Occasionally, it is necessary or simply convenient to replace a function binding while a program is running. For example, one might want to replace `lstack::push()` and `lstack::pop()` with new and improved versions without terminating and restarting the program. This is fairly easily achieved by taking advantage of the fact that calls of virtual functions are indirect through some sort of table of virtual functions (often called the `vtbl`).

The only portable way of doing this requires cooperation from the `lstack` class; it must have a constructor that does no work except for setting up the `vtbl` that all constructors do implicitly:

```
class noop {};

lstack::lstack(noop) {} // make an uninitialized lstack
```

Fortunately such a constructor can be added to the program source text without requiring recompilation that might affect the running program that we are trying to update (enhance and repair).

Using this extra constructor we define a new class which is identical to `lstack` except for the redefined operations:

```
// modified linked list stack interface, llstack.h:

#include "stack.h"

class llstack : public lstack {
public:
        llstack(int size) : lstack(size) {}
        llstack(noop x) : lstack(x) {}

        void push(cat);
        cat pop();

};
```

Given an `lstack` object we can now update its pointer to its table of virtual functions (its `vtbl`) thus ensuring that future operations on the object will use the `llstack` variants of `push()` and `pop()`:

```
#include "llstack.h"

void g(lstack& s)
{
        noop xx;
        new(&s) llstack(xx);     // turns s into an llstack!
}
```

Naturally, we must rely on some form of dynamic linking to make it possible to add `g()` to a running program. Most systems provide some such facility.

This use of `operator new` assumes that

```
// place an object at address 'p':
void* operator new(size_t,void* p) { return p; }
```

has been defined and relies on the `llstack::llstack(noop)` constructor for suppressing (re)initialization of the data of `s` and for updating `s`'s `vtbl`.

This trick allows us to change the `vtbl` for particular objects without relying on specific properties of an implementation (that is, portably). There is no language protection against misuse of this trick.

Changing the `vtbl` for every object of class `lstack` in one operation, that is, changing the contents of the `vtbl` for class `lstack` rather than simply changing pointers to `vtbl`s in the individual objects, cannot be done portably. However, since that operation will be messy and non-portable I will not give an example of it.

## 14 Changing Representations

A more interesting – and probably more realistic – challenge is to replace both the representation and the operations for an object at run time. For example, convert a stack from an array representation to a linked list representation at run time without affecting its users. To ensure that the cutover from one representation to another can be done by a single assignment we reintroduce the `rep` type and make `push()` and `pop()` simple forwarding functions to operations on the `rep`:

```
// stack interface, stack.h:

class stack {
        rep* p;
public:
        stack(int size);
        ~stack();

        rep* get_rep() { return p; }
        void put_rep(rep* q) { p = q; }

        void push(cat c) { p->push(c); }
        cat pop() { return p->pop(); }

        int size() { return p->size(); }
};
```

The idea is to have operations that convert between the different representations and then let them use `get_rep()` and `put_rep()` to update the pointer to the representation. In a real system `get_rep()` and `put_rep()` would most likely not be publicly accessible functions.

First we define `rep` exactly as we did `stack` before :

```
// stack interface, rep.h:

class rep {
public:
        virtual void push(cat) = 0;
        virtual cat pop() = 0;
        virtual int size() = 0;
};
```

and use it as a base for the different implementations:

```
// array stack interface, astack.h:

#include "rep.h"

class astack : public rep {
        // actual representation of a stack object
        // in this case an array
        // ...
public:
        astack(int size);
        ~astack();

        void push(cat);
        cat pop();

        int size();
};
```

Elsewhere we can define a stack implemented using a linked list:

```
// linked list stack interface, lstack.h:

#include "rep.h"

class lstack : public rep {
        // actual representation of a stack object
        // in this case a linked list
        // ...
public:
        lstack(int size);
        ~lstack();

        void push(cat);
        cat pop();

        int size();
};
```

Now we can convert the representation of a `stack` from a `astack*` to a `lstack*` by changing `stack::p` using `stack::get_rep()` and `stack::put_rep()` and (in general) also copying the elements:

```
rep* convert_from_a_to_l(stack& s)
{
        rep* rp = s.get_rep();
        astack* ap = new astack(s.size());
        // copy s's elements to *ap
        s.put_rep(ap);
        return rp;
}
```

In other words, we solve the problem by introducing yet another indirection†. This assumes that the size argument from the original constructor has been stored away somewhere so that it can be used as the argument to the new `astack`. In a real system we would also need to check that `s` really had an appropriate representation and would most likely also have to provide some further consistency checking and interlocking. However, the fundamental idea is illustrated.

## 15 Changing the Set of Operations

Finally, I will show a version of the stack example that is somewhat un-C++-like in that it dispenses with static type checking of the operations. The idea is to completely disconnect the users and the implementers and simulate a dynamically type-checked language. Overreliance on such techniques can make systems slow and messy. However, the flexibility offered can be important in localized contexts where the inevitable problems caused by lack of formal structure and of run-time checking can be contained.

In this and the following examples a bit of scaffolding is needed to make the programming techniques convenient. This makes the toy examples somewhat longer to define but does not, in fact, noticeably increase the size of a realistic system relying on them. The most primitive building block for this example is lists of (operation_identifier,function) pairs:

† The first law of computer science: Every problem is solved by yet another indirection.

```
typedef cat (*PcatF)(void*,cat);

struct oper_link {
        oper_link* next;
        int oper;
        PcatF fct;

        oper_link(int oo, PcatF ff, oper_link* nn)
                : oper(oo), fct(ff), next(nn) {}

};
```

Using `oper_links` we can specify a class `cat_object` that allows us to invoke an unspecified set of functions on an unspecified representation:

```
class cat_object {
        void* p;        // pointer to representation
        oper_link* oper_table; // list of operations

public:
        cat_object(oper_link* tbl = 0, void* rep = 0)
                : oper_table(tbl), p(rep) { }

        cat operator()(int oper, cat arg = 0);

        void add_oper(int, PcatF);
        void remove_oper(int);

};
```

Default arguments are user to spare the programmer the bother of specifying arguments where they are not in fact necessary for a particular operation. This technique is elaborated in section 16 below.

The application operator simply looks for an operation in its list and executes it (if found):

```
cat cat_object::operator()(int oper, cat arg)
{
        for (oper_link* pp = oper_table; pp; pp = pp->next)
                if (oper == pp->oper) return pp->fct(p,arg);
        return bad_cat;
}
```

If the operation fails the distinguished object `bad_cat` is returned.

Given this feeble framework we can now build a stack:

```
// stack interface, stack.h:
enum stack_oper { stack_destroy = 99, stack_push, stack_pop };

cat_object* make_stack(cat_object* = 0);
```

As usual, the implementation of the stack is left as an exercise to the reader. However, here is a hint:

```
#include "stack.h"

struct rep {
        // ...
        void push(cat);
        cat pop();
};
```

```
static cat stack_push_fct(void* p, cat c)
{
        ((rep*) p)->push(c);
        return bad_cat;
}

static cat stack_pop_fct(void* p, cat)
{
        return ((rep*) p)->pop();
}

static cat stack_destroy_fct(void* p, cat) { /* ... */ }

cat_object* make_stack(cat_object* p)
{
        if (p == 0) p = new cat_object(0,new rep);  // get a clean object
        p->add_oper(stack_push,&stack_push_fct);    // and make it
        p->add_oper(stack_pop,&stack_pop_fct);      // behave
        p->add_oper(stack_destroy,&stack_destroy_fct); // like a stack
        return p;
}
```

We can now create and use stacks:

```
#include "stack.h"

void g(cat kitty)
{
        cat_object* s = make_stack();
        s(stack_push,kitty);
        cat c2 = s(stack_pop);
        s(stack_destroy);
}
```

We can add operations to a stack at run time. For example, we might want an operation for peeking at the top `cat` without popping:

```
enum { stack_peek = stack_pop+1 };
cat stack_peek_fct(void*,cat);

void h(cat_object& s)
{
        s.add_oper(stack_peek,&stack_peek_fct);

        cat top = s(stack_peek);
}
```

Note that the operations on these stacks do not involve addresses; they can be transmitted between address spaces without special effort. This technique for invoking operations is often called message passing.

## 16 Tailoring

The dynamically typed stack above could be improved in many ways. For example, the operation lookup could be made faster, an inheritance mechanism could be added, the naming of operations could be made more general and safer, the method for passing arguments could be made more general and safer, etc. Here I will just demonstrate two techniques of more general interest.

Firstly, the message passing mechanism can be hidden behind an interface that provides notational convenience and type safety for the key stack operations. The point is that combinations of the techniques described in this paper can be used to handle more delicate cases. In particular, any data abstraction technique can be used to hide ugliness in an implementation:

```
// improved stack interface, Stack.h:

#include "stack.h"
class stack : public cat_object {
public:
        stack() { make_stack(this); }
        ~stack() { (*this)(stack_destroy); }
        void push(cat c) { (*this)(stack_push,c); }
        cat pop() { return (*this)(stack_pop); }
};
```

This allows us to write

```
#include "Stack.h"

void g(int sz, cat kitty)
{
        stack s;

        // compile time checked uses:

        s.push(kitty);
        cat c2 = s.pop();

        // run time checked uses:

        s.add_oper(stack_peek,&stack_peek_fct);
        cat top = s(stack_peek);
}
```

This uses the unchecked "message passing" notation only where needed.

Secondly, in the dynamically typed stack example I dodged the issue of argument types and argument type checking by simply providing a fixed number of arguments of fixed type. Similarly, I simply had all operations return a `cat`. That is surprisingly often a viable choice for the sort of interfaces for which you actually need "message passing." A larger class of problems can be handled by allowing arguments of a fixed number of types. For example:

```
class argument {
        enum type_indicator { non_arg, int_arg, ptr_arg, string_arg, cat_arg };
        type_indicator t;
        union {
                int i;
                void* p;
                char* s;
                cat c;
        };
public:
        argument(noop) :t(non_arg) { }
        argument(int ii) : i(ii), t(int_arg) { }
        argument(void* pp) : p(pp), t(ptr_arg) { }
        argument(char* ss) : s(ss), t(string_arg) { }
        argument(cat cc) : c(cc), t(cat_arg) { }

        operator int() { return t==int_arg ? i : -1; }
        operator void*() { return t==ptr_arg ? p : 0; }
        operator char*() { return t==string_arg ? s : 0; }
        operator cat() { return t==cat_arg ? c : bad_cat; }
};
```

This assumes that `cat` is declared so that == and `?:` can be applied.

The error handling for the conversion operators could be improved, but even as it stands this would allow the cat_object class to be written:

```
typedef argument (*PargumentF)(void*,argument);

struct oper_link {
        oper_link* next;
        int oper;
        PargumentF fct;

        oper_link(int oo, PcatF ff, oper_link* nn)
                : oper(oo), fct(ff), next(nn) {}

};
```

and can be used like this:

```
#include "stack.h"

void g(cat kitty)
{
        argument_object& s = *make_stack();
        s(stack_push,kitty);    // converts 'kitty' to argument
        cat c2 = s(stack_pop);  // converts argument to cat
        s(stack_destroy);

}
```

The object `no_arg` is passed (by default) to indicate that no argument was specified by the user.

You may have noticed that the size argument to the stack create operation disappeared when we moved to the message passing style. The reason was to avoid the complication of dealing with arguments of different types until we had the mechanism to do so. Putting that argument back in is now left as an exercise to the reader.

Adding "list of `arguments`" to the list of acceptable `argument` types is left as yet another exercise to the reader. Allowing such lists again increases the range of applications for which the message passing technique is acceptable.

## 17 Dynamic Classes

Note that the concept of a class was almost lost in the message passing examples above. Each object had its own list of acceptable operations that could be modified independently of the lists of any other object. This could be seen as too flexible for many applications and also not sufficiently amenable to space and time optimizations. These problems can be alleviated by re-introducing the notion of a class as an object containing information common to a set of objects. Here we will only represent the set of acceptable operations on an object of one of these "dynamic classes."

```
class argument_class_rep {
public:
        oper_link* oper_table; // list of operations

        void add_oper(int, PargumentF);
        void remove_oper(int);

};
```

The reader can easily extend this notion, though.

The objects looks much as they did before. The only change is that an indirection through an object representing a class has been introduced on the path to the table of operations:

```
class argument_object {
        void* p;                        // pointer to representation
        argument_class_rep* crep; // class
public:
        argument_object(argument_class_rep* cc, void* rep = 0)
                : crep(cc), p(rep) { }

        argument operator()(int oper, argument arg = no_arg);
};
```

Given this we can create an object representing stacks and provide an operation for gaining access to it:

```
static argument_class_rep stack_class;  // the object representing stacks

argument_class_rep* get_stack_class()
{       if (stack_class->oper_table == 0) {
                stack_class.add_oper(stack_push,&stack_push_fct);
                stack_class.add_oper(stack_pop,&stack_pop_fct);
                stack_class.add_oper(stack_destroy,&stack_destroy_fct);

        }
        return &stack_class;
}
```

Finally, we can provide a function for making stacks

```
argument_object* make_stack()
{       return new argument_object(get_stack_class());
}
```

and use it exactly as the previous versions:

```
void g(cat kitty)
{       argument_object& s = *make_stack();
        s(stack_push,kitty);    // converts 'kitty' to argument
        cat c2 = s(stack_pop);  // converts argument to cat
        s(stack_destroy);

}
```

This version, however, does not allow operations for an individual object to change. It does however, open the possibility of trivially changing the operations on all objects of a dynamic class.

## 18 Conclusions

C++ covers the spectrum of data hiding techniques from C (files as modules) through Modula-2 (modules) and Ada (packages) to C++ (classes), and beyond. Given a free choice a C++ programmer would naturally choose one of the class-based strongly-typed techniques (sections 10, 11, or 12), but the other techniques can occasionally be useful.

## 19 Acknowledgements

Andrew Koenig, Brian Kernighan, and Doug McIlroy lent ears and blackboard to some of these little puzzle programs. Jim Coplien suggested the extension of the range of examples to include function binding examples. The nine-plus cat lives in this paper are dedicated to Dave McQueen who once in desperation proposed the death penalty for presenting "yet another stack example." Also thanks to Andy for giving me a practical demonstration of the difficulty of stacking cats.

# MEMORY MANAGEMENT TECHNIQUES IN C++

William M. Miller
Glockenspiel, Ltd.

Internet: wmm@world.std.com
CompuServe: 72105,1744
BIX: wmiller



`new` and `delete` operators:

```
int* pi = new int;
char* pc = new char[count];
delete pi;
delete [count] pc;
```

Type secure:

- returns `type*`, not `void*`
- size calculation done automatically
- no cast required

Integrated with constructor/destructor system

Well-defined error handling

`new C(args)` allocates storage, invokes constructor with resulting address as `this`:

```
C* p = malloc(sizeof(C));
C_ctor(p, args);
```

`delete objectptr` invokes destructor with `objectptr` as `this`, then frees storage:

```
C_dtor(objectptr);
free(objectptr);
```

`new C[count]` allocates enough storage for `count` copies of `C`, then invokes the default (i.e., no-argument) constructor for each copy:

```
C* p = malloc(count*sizeof(C));
for (i = 0; i < count; i++)
   C_ctor(p + i);
```

`delete [count] objectptr` invokes the destructor for each of `count` copies of the object, then frees the storage:

```
for (i = count-1; i >= 0; i--)
   C_dtor(objectptr + i);
free(objectptr);
```

Types of `new C` and `new C[count]` are identical: `C*`

Order of destruction of array elements is inverse of order of construction

Multi-dimensional arrays are supported:

```
C (*p) [J][K];

p = new C[I][J][K];

delete [I] p;
```

The leftmost dimension (`[I]`) can be any arbitrary expression; the remaining dimensions must be constant expressions.

Notes:

1) Can pass arguments to constructor in non-array case: `new C(arg1, arg2)`
2) Cannot pass arguments to constructor in array case; must have default (no-argument) constructor
3) `[count]` is required for invoking destructors, redundant for types without destructors
4) `delete` is only polymorphic (i.e., able to delete a derived class object via a pointer to a base class) if the destructor is declared virtual
5) `delete` of an array MUST be used with the correct type pointer, not a pointer to a base type

New for 2.1:

1) Cannot delete via a pointer to constant, i.e., `const C*`

2) It is permitted to allocate an array of a class whose constructor has all default arguments:

```
class complex {
public:
   complex(float re = 0.0, float im = 0.0);
   };

new complex[100];
```

3) `count` should be omitted from array deletion, e.g., `delete [] p` instead of `delete [100] p`

Error handling:

```
typedef void (*PF)();
extern PF _new_handler;
extern PF set_new_handler(PF);

void f() {
   PF old = set_new_handler(mine);
   //...
   (void) set_new_handler(old);
   }
```

When storage is exhausted, the function pointed to by `_new_handler` is invoked. It can:

- print a message and `exit()`
- free some storage and return, causing allocation to be retried

If no `_new_handler` is defined, `new` returns 0.

Can replace allocation strategy globally:

```
void* operator new(size_t);

void operator delete(void*);
```

Per-class allocation control (OBSOLETE TECHNIQUE):

```
class foo {
   foo() {
      if (!this)
         this = my_alloc();
      //...
      }
   ~foo() {
      //...
      this = 0;
      }
   };
```

If constructor assigns to `this`:

- `this` will be 0 on entry if object was allocated via `new`
- Base and member constructors called only after assignment to `this`

Problems with assignment to `this`:

1) Error-prone. If `this` is not assigned on some execution paths, base and member constructors will not be called under some conditions.

2) Base class cannot provide allocation strategy for derived class because `this` is guaranteed to have a non-zero value before the base class constructor is executed.

CONCLUSION: obsolete as of 2.0, support can be removed at any time.

Per-class `operator new()` and `operator delete()`:

```
class foo {
   //...
   void* operator new(size_t);
   void operator delete(void*);
   // or, (void*, size_t)
   };
```

Invoked whenever an object of class `foo` or a class derived from `foo` is allocated via `new`.

Is NOT invoked for arrays.

Deals only with storage, not object:

- `void*`, not `foo*`
- no `this` pointer

Overloaded `operator new()` (global or member):

```
void* operator new(size_t, args);
```

First argument must be `size_t`; remaining arguments are unconstrained.

If have multi-argument `operator new()` member function, some implementations require that default version also be provided because of name hiding.

Additional arguments are supplied in a parenthesized list between `new` and the type name:

```
new (args) type(init)
```

Useful for:

Creating objects in existing storage, e.g., for batching file writes by allocating in a buffer

```
inline void* operator new(size_t, void* p) {
   return p;
   }

p = new (buff_ptr) foo;
buff_ptr += sizeof(foo);
```

Communicating additional information about the allocation strategy:

```
void* operator new(size_t, enum where);

p = new (NEAR_HEAP) foo;
```

Cannot overload `operator delete()`; corresponding feature is explicit invocation of destructor:

```
p->foo::~foo();
```

Storage deallocation, if needed, is handled separately via explicit code after the destructor has run.

Version issue: current and previous versions required full qualification of destructor name. Full qualification normally suppresses virtual overriding. A future version will allow `p->~foo()` for virtual functions and the full form will have its expected meaning.

Smart pointers:

```
struct foo {
   int bar;
   };

class fooptr {
   //...
   foo& operator*();
   foo* operator->();
   };

fooptr p;
p->bar = 5;
```

Useful for virtual memory, remote access, etc.

Static data members:

- Only one allocated for class, not one per object
- Can be accessed as `C::member` as well as `object.member` or `objectptr->member`
- Must be defined outside of class (has external linkage); initializer is evaluated in class's scope

```
class foo {
   static int member;
   };

int foo::member = -32767;
```

Useful for object counter, head/tail of linked list, etc.

Static member functions:

```
class foo {
   //...
   static void mbrfunc();
   };

void foo::mbrfunc() { ... }
```

Access with or without object ( `foo::mbrfunc()` )

Has no `this` pointer, i.e., can access only static data members and member functions without use of explicit object qualification

`const` and/or `volatile` member functions:

```
class foo {
   int i;
public:
   int f() const { return i; }
   };
```

Objects declared `const` allow access only to member functions declared `const`. (Unqualified objects can use `const` member functions.)

Member functions declared `const` are not allowed to modify data members or call non-`const` member functions.

Can overload member functions based on `const`.

`volatile` is treated analogously.

PROBLEM: need to insure certain initialization code is run before any static objects of type `foo` are constructed, but order of construction across modules is arbitrary.

SOLUTION: in the header file which declares class `foo`, define a `foo_init` class and a static object of that type:

```
class foo { ... };

class foo_init {
   static int obj_count;
public:
   foo_init();
   ~foo_init();
   };

static foo_init xfoo;
```

Constructor performs needed initialization, destructor handles cleanup:

```
foo_init::foo_init() {
   if (obj_count++ == 0) {
      // initialization for class foo
      }
   }

foo_init::~foo_init() {
   if (--obj_count == 0) {
      // cleanup for class foo
      }
   }
```

The reason this works is that static allocation is guaranteed to be in order of appearance WITHIN a module.

Putting a static `foo_init` object into the header file guarantees that it will be constructed before any possible use of `class foo`.

This trick was invented by Jerry Schwarz, formerly of AT&T, for the implementation of io_streams.

Temporaries can cause performance problems:

```
struct foo {
   foo();
   foo(foo&);
   ~foo();
   foo some_op(foo);
   };

foo foo::some_op(foo f) {
   foo ret_val;
   // compute ret_val
   return ret_val;
   }

void f() {
   foo a, b, c;
   // calculate values for b & c
   a = b.some_op(c);
   }
```

Execution for this example:

1) Copy value of `c` to argument temporary (constructor)

2) Construct `ret_val` in `foo::some_op`

3) Copy `ret_val` to function return value temporary (constructor)

4) Destroy `ret_val`

5) Copy function return value to `a` (assignment operator)

6) Destroy function return value

7) Destroy argument temporary

**Reduced cost code:**

```
struct foo {
   foo();
   foo(foo&);
   foo(const foo&, const foo&);
   };
foo::foo(const foo&, const foo&) {
   // compute like rat_val
   }

foo b, c;
foo a(b, c);
```

- Use `const foo&` instead of `foo` parameters to avoid argument copying
- Use initialization and a constructor instead of a normal function and assignment

**Acknowledgement: Jonathan Shopiro**

**Temporaries can cause bugs:**

```
class Str {
   char* data;
   Str(const Str&, const Str&);
public:
   //...
   operator char*() { return data; }
   Str operator+(const Str& s) {
      return Str(*this, s);
      }
   };

void foo(const char*);

void f() {
   Str a, b;
   //...
   foo(a + b);  // generates temporary
   // that may be deleted BEFORE f is called
   }
```