C++ ABI Open Issues

Revised 14 September 1999

Revisions

[990914] Additions to B-1, D-*, D-9.

[990908] New issue D-9. Additions to B-1, D-*, D-2, D-4, D-5, D-6.

[990901] Additions to A-6, B-6.

[990825] Additions to A-6, B-6, C-5, D-*.

[990813] Closed A-11. Additions to A-6, B-1, B-6, B-8, C-2, G-5.

[990810] New issue G-5. Additions to B-6, C-2, C-3.

[990805] Closed A-12, A-14, B-3, B-4, B-7, C-7. Additions to A-6, A-11, B-1, B-6, F-1.

[990729] Closed A-7. Additions to A-11, A-12, C-2. Summary added for A-12. New issue A-14.

[990727] Closed B-2, C-8. Additions to A-9 (closed), C-2. Summaries added for C-4, C-6, D-1 to D-4.

[990720] Additions to B-2, B-5, C-2, D-1.

[990701] Closed A-3, A-5, A-10, A-13. Additions to A-6, B-6, B-7, B-8, C-2, C-7.

[990625] Closed A-1, A-2, A-4, A-8, A-9. Additions to A-3, A-5, A-7, B-4, B-5, B-7, G-3, G-4. New issues B-6, B-7, B-8, C-7, C-8.

[990616] Added HP summaries. Added sketchy notes from 990610 discussions (A and B issues). A-10 was intended by HP as something different than I described, so it was renamed, and a new issue A-13 opened as an SGI issue. HP did not submit A-12, so relabeled as Sun's (is that right?). Added library interface issues, H-1 and H-2.

Definitions

The issues below make use of the following definitions:

empty class
: A class with no non-static data members, no virtual functions, no virtual base classes, and no non-empty non-virtual base classes.)

nearly empty class
: A class, the objects of which contain only a Vptr.

vague linkage
: The treatment of entities -- e.g. inline functions, templates, vtables -- with external linkage that can be defined in multiple translation units, while the ODR requires that the program behave as if there were only a single definition.

Issue Status

In the following sections, the ***class*** of an issue attempts to classify it on the basis of what it likely affects. The identifiers used are:

call	Function call interface, i.e. call linkage
data	Data layout
lib	Runtime library support
lif	Library interface, i.e. API
g	Potential gABI impact
ps	Potential psABI impact
source	Source code conventions (i.e. API, not ABI)
tools	May affect how program construction tools interact

Object Layout Issues

#	Issue	Class	Status	Source	Opened	Closed
A-1	Vptr location	data	closed	SGI	990520	990624

Summary: Where is the Vptr stored in an object (first or last are the usual answers).

Resolution: First.

#	Issue	Class	Status	Source	Opened	Closed
A-2	Virtual base classes	data	closed	SGI	990520	990624

Summary: Where are the virtual base subobjects placed in the class layout? How are data member accesses to them handled?

Resolution: Virtual base subobjects are normally placed at the end (see issue A-9). The Vtable will contain an offset to the beginning of the base object for use by member accesses to them (see issue B-6).

#	Issue	Class	Status	Source	Opened	Closed
A-3	Multiple inheritance	data	closed	SGI	990520	990701

Summary: Define the class layout in the presence of multiple base classes.

Resolution: See the class layout description in closed issue A-9. Briefly, empty bases will normally go at offset zero, non-virtual base classes at the beginning, and virtual base classes at the end.

#	Issue	Class	Status	Source	Opened	Closed
A-4	Empty base classes	data	closed	SGI	990520	990624

Summary: Where are empty base classes allocated?

Resolution: At offset zero if possible. See A-9.

<table>
<tr><th>#</th><th>Issue</th><th>Class</th><th>Status</th><th>Source</th><th>Opened</th><th>Closed</th></tr>
<tr><td>A-5</td><td>Empty parameters</td><td>data</td><td>closed</td><td>SGI</td><td>990520</td><td>990701</td></tr>
<tr><td colspan="7"><b>Summary</b>: When passing a parameter with an empty class type by value, what is the convention?</td></tr>
<tr><td colspan="7"><b>Resolution</b>: Except for cases of non-trivial copy constructors (see C-7), and parameters in the variable part of varargs lists, no parameter slot will be allocated to empty parameters.</td></tr>
</table>

<table>
<tr><th>#</th><th>Issue</th><th>Class</th><th>Status</th><th>Source</th><th>Opened</th><th>Closed</th></tr>
<tr><td>A-6</td><td>RTTI .o representation</td><td>data call ps</td><td>open</td><td>SGI</td><td>990520</td><td></td></tr>
<tr><td colspan="7"><b>Summary</b>: Define the data structure to be used for RTTI, that is:
• for user <code>type_info</code> calls;
• for dynamic_cast implementation; and
• for exception-handling.</td></tr>
</table>

[990701 All] Daveed will put together a proposal by the 15th (action #13); the group will discuss it on the 22nd.

[990805 All] Daveed should have his proposal together for discussion. Michael Lam will look into the Sun dynamic cast algorithm.

It was noted that appropriate name selection along with the normal DSO global name resolution should be sufficient to produce a unique address for each class' RTTI struct, which address would then be a suitable identifier for comparisons.

[990812 Sun -- Michael] Sun has provided a description, in a separate page, describing their implementation. They are filing for a patent on the algorithms described.

[990819 EDG -- Daveed]

Run-time type information

The C++ programming language definition implies that information about types be available at run time for three distinct purposes:

a. to support the typeid operator,
b. to match an exception handler with a thrown object, and
c. to implement the dynamic_cast operator.

(c) only requires type information about polymorphic class types, but (a) and (b) may apply to other types as well; for example, when a pointer to an int is thrown, it can be caught by a handler that catches "int const*".

Open questions

- What is the type of base class offsets?
- What is the limit on the number of classes?
- Should the common dynamic_cast case be optimized by adding an entry to the vtable (to indicate a polymorphic base subobject appears on more than one derivation path to the complete object type)?

Place of emission

It is probably desirable to minimize the number of places where a particular bit of RTTI is emitted. For polymorphic types, a similar problem occurs for virtual function tables, and hence the information can be appended at the end of the primary vtable for that type. For other types, they must presumably be emitted at the location where their use is implied: the object file containing the typeid, throw or catch. Basic type information (such as for "int", "bool", etc.) could be kept in the run-time support library but the benefits of that may be limited.

The typeid operator

The typeid operator produces a reference to a std::type_info structure with the following public interface:

```
  struct std::type_info {
     virtual ~type_info();
     bool operator==(type_info const&) const;
     bool operator!=(type_info const&) const;
     bool before(type_info const&) const;
     char const* name() const;
  };
```

Assuming that after linking and loading only one type_info structure is active for any particular type symbol, the equality and inequality operators can be written as address comparisons: to type_info structures describe the same type if and only if they are the same structure (at the same address). In a flat address space (such as that of the IA-64 architecture), the before() member is also easily written in terms of an address comparison. The only additional piece of information that is required is the NTBS that encodes the name. The type_info structure itself can hold a pointer into a read-only segment that contains the text bytes.

Matching throw expressions with handlers

When an object is thrown a copy is made of it and the type of that copy is TT. A handler that catches type HT will match that throw if:

- HT is equal to TT except that HT may be a reference and that HT may have top-level cv qualifiers (i.e., HT can be "TT cv", "TT&" or "TT cv&"); or
- HT is a reference to a public and unambiguous base type of TT; or
- HT has a pointer type to which TT can be converted by a standard pointer conversion (though only public, unambiguous derived-to-base conversions are permitted) and/or a qualification conversion.

This implies that the type information must keep a description of the public, unambiguous inheritance relationship of a type, as well as the const and volatile qualifications applied to types.

The dynamic_cast operator

Although dynamic_cast can work on pointers and references, from the point of view of representation we need only to worry about polymorphic class types. Also, some kinds of dynamic_cast operations are handled at compile time and do not need any RTTI. There are then three kinds of truly dynamic cast operations:

- dynamic_cast, which returns a pointer to the complete lvalue,
- dynamic_cast operation from a base class to a derived class, and
- dynamic_cast across the hierarchy which can be seen as a cast to the complete lvalue and back to a sibling base.

RTTI layout

1. The RTTI layout for a given type depends on whether a 32-bit or 64-bit mode is in effect.

2. Every vtable shall contain one entry describing the offset from a vptr for that vtable to the origin of the object containing that vptr (or equivalently: to the vptr for the primary vtable). This entry is directly useful to implement dynamic_cast, but is also needed for the other truly dynamic casts.

3. Every vtable shall contain one entry pointing to an object derived from std::extended_type_info; the possible types are:
   - std::__fundamental_type_info
   - std::__pointer_type_info
   - std::__reference_type_info
   - std::__array_type_info
   - std::__function_type_info
   - std::__enum_type_info
   - std::__class_type_info
   - ~~std::__qualifier_type_info~~

   std::extended_type_info derives from std::type_info but adds no fields to the latter; it is introduced solely to follow the suggestion of the C++ standard.

4. std::type_info contains just two pointers:
    - its vptr
    - a pointer to a NTBS representing the name of the type

5. std::__pointer_type_info adds two fields:

    - a word describing the cv-qualification of what is pointed to (e.g., "int volatile*" should have the "volatile" bit set in that word)

    - a pointer to the std::type_info derivation for the unqualified type being pointed to

    Note that the first bits should not be folded into the pointer because we may eventually need more qualifier bits (e.g. for "restrict").

6. std::__reference_type_info is similar to std::__pointer_type_info but describes references.

7. ~~std::__qualifier_type_info is similar to std::__pointer_type_info but describes top level qualifiers as in "int const" and "char *const".~~

8. std::__class_type_info introduces a variable length structure. The variable part that follows consists of a sequence of base class descriptions having the following structure:

```
    struct std::__base_class_info {

       std::type_info *type; /* Null if unused. */
       std::ptrdiff_t offset;
       std::__base_class_index next; /* Hash table link. */
       int is_direct: 1;
       int is_floating: 1; /* I.e., virtual or base of virtual
subobject. */
       int is_virtual: 1; /* Implies is_floating. */
       int is_shared: 1; /* Implies is_floating and the virtual
subobject
                            appears on multiple derivation paths. */
       int is_accessible: 1;
       int is_ambiguous: 1;

    };
```

    The fixed length introduction adds the following fields to std::type_info:
    - a word with flags describing details about the class such as whether it is a class/struct/union and whether it is polymorphic.
    - a hashvalue that can be used for quick lookups in a variable length structure describing base classes.
    - the number of base class descriptions that follow it (a power of two).

std::type_info::name()

The NTBS returned by this routine is the mangled name of the type.

The dynamic_cast algorithm

Dynamic casts to "void cv*" are inserted inline at compile time. So are dynamic casts of null pointers and dynamic casts that are really static.

This leaves the following test to be implemented in the run-time library for truly dynamic casts of the form "dynamic_cast(v)": (see [expr.dynamic_cast] 5.2.7/8)

- If, in the most derived object pointed (referred) to by v, v points (refers) to a public base class sub-object of a T object [note: this can be checked at compile time], and if only one object of type T is derived from the sub-object pointed (referred) to by v, the result is a pointer (an lvalue referring) to that T object.

- Otherwise, if v points (refers) to a public base class sub-object of the most derived object, and the type of the most derived object has an unambiguous public base class of type T, the result is a pointer (an lvalue referring) to the T sub-object of the most derived object.

- Otherwise, the run-time check fails.

The first check corresponds to a "base-to-derived cast" and the second to a "cross cast". These tests are implemented by std::__dynamic_cast.

```
void* std::__dynamic_cast(void *sub, std::__class_type_info *src,
                                     std::__class_type_info *dst,
                                     std::ptrdiff_t src2dst_offset)
{

  // Pick up vtable pointer from given object:
  void *vptr = *(void**)sub;
  if (src2dst_offset>=0 && NO_VBASE(sub, vptr)) {
    // If the type of "v" was not an accessible nonvirtual base type of
    // "T", src2dst_offset should have been set to -1 if it was an
    // accessible but floating base of "T" and to -2 if it was not at all
    // an accessible base of "T".
    // In addition, the vtable should contain an entry to indicate that
    // the complete object has no virtual bases (e.g., a count of the
    // vbase locator entries).
    return (char*)sub+src2dst_offset;
  } else {
    // Slower case
    void *result = 0;
    if (src2dst_offset==-1) {
      // Possibly a "floating" base-to-derived cast:
      result = floating_base2derived(sub, src, dst);
    }
    if (result==0) {
      // The base-to-derived case did not succeed, so we should attempt
      // the cross-cast (which is really a derived-to-base cast from the
      // complete object):
      result = derived2base(complete, dst);
    }
  }
  return result;
}
```

The exception handler matching algorithm

```
bool __eh_match(std::type_info *thrown_type,
                std::type_info *handled_type) {
  if (thrown_type == handled_type) {
    return true;
  } else if (IS_REFTYPE(handled_type)) {
    std::type_info *caught_type = REMOVE_REFTYPE(handled_type);
    if (IS_CVQUAL(caught_type)) {
      caught_type = REMOVE_CVQUAL(caught_type);
    }
    return thrown_type==caught_type;
  } else if (IS_PTRTYPE(handled_type)) {
    cvqual_match(&thrown_type, &handled_type);
    if (thrown_type==handled_type) {
      return true;
    } else if (IS_CLASSTYPE(thrown_type) &&
               IS_CLASSTYPE(handled_type)) {
      return derived2base_check(thrown_type, handled_type);
    } else {
      return false;
  }
 } else {
  return false;
 }
}
```

[990826 All] Discussion centered on whether the representation should include all base classes or just the direct ones, and in the former case how hashing might be handled. It was agreed that the __qualifier_type_info variant is not needed, and it is now striken in the above proposal. Also, a pointer-to-member variant is needed. Christophe will provide a description of the HP hashing approach, and Daveed will update the specification.

#	Issue	Class	Status	Source	Opened	Closed
A-7	Vptr sharing with primary base class	data	closed	HP	990603	990729
Summary: It is in general possible to share the virtual pointer with a polymorphic base class (the *primary* base class). Which base class do we use for this?						
Resolution: Share with the first non-virtual polymorphic base class, or if none with the first nearly empty virtual base class.						

#	Issue	Class	Status	Source	Opened	Closed
A-8	(Virtual) base class alignment	data	closed	HP	990603	990624
Summary: A (virtual) base class may have a larger alignment constraint than a derived class. Do we agree to extend the alignment constraint to the derived class?						
Resolution: The derived class will have at least the alignment of any base class.						

#	Issue	Class	Status	Source	Opened	Closed
A-9	Sorting fields as allowed by [class.mem]/12	data	closed	HP	990603	990624
Summary: The standard constrains ordering of class members in memory only if they are not separated by an access clause. Do we use an access clause as an opportunity to fill the gaps left by padding?						
Resolution: See closed issue list.						

[990722 all] The precise placement of empty bases when they don't fit at offset zero remained imprecise in the original description. Accordingly, a precise layout algorithm is described in a separate writeup of Data Layout.

#	Issue	Class	Status	Source	Opened	Closed
A-10	Class parameters in registers	call	closed	HP	990603	990701
Summary: The C ABI specifies that small structs are passed in registers. Does this apply to small non-POD C++ objects passed by value? What about the copy constructor and `this` pointer in that case?						
Resolution: Non-POD C++ objects are passed like C structs, except for cases with non-trivial copy constructors identified in C-7.						

#	Issue	Class	Status	Source	Opened	Closed
A-11	Pointers to member functions	data	closed	Cygnus	990603	990812
Summary: How should pointers to member functions be represented?						
Resolution: As a pair of values, a "pointer" and a this adjustment. See the closed list for a more detailed description.						

#	Issue	Class	Status	Source	Opened	Closed
A-12	Merging secondary vtables	data	closed	Sun	990610	990805
Summary: Sun merges the secondary Vtables for a class (i.e. those for non-primary base classes) with the primary Vtable by appending them. This allows their reference via the primary Vtable entry symbol, minimizing the number of external symbols required in linking, in the GOT, etc.						
Resolution: Concatenate the Vtables associated with a class in the same order that the corresponding base subobjects are allocated in the object.						

#	Issue	Class	Status	Source	Opened	Closed
A-13	Parameter struct field promotion	call	closed	SGI	990603	990701
Summary: It is possible to pass small classes either as memory images, as is specified by the base ABI for C structs, or as a sequence of parameters, one for each member. Which should be done, and if the latter, what are the rules for identifying "small" classes?						
Resolution: No special treatment will be specified by the ABI.						

#	Issue	Class	Status	Source	Opened	Closed
A-14	Pointers to data members	data	closed	SGI	990729	990805
Summary: How should pointers to data members be represented?						
Resolution: Represented as one plus the offset from the base address.						

Virtual Function Handling Issues

#	Issue	Class	Status	Source	Opened	Closed
B-1	Adjustment of "this" pointer (e.g. thunks)	data call	open	SGI	990520	
Summary: There are several methods for adjusting the *this* pointer for a member function call, including thunks or offsets located in the vtable. We need to agree on the mechanism used, and on the location of offsets, if any are needed. To maximize performance on IA64, a slightly unusual approach such as using secondary entry points to perform the adjustment may actually prove interesting.						

[990623 HP -- Christophe]

Open Issues Relevant To This Discussion

1. Keeping all of a class in a single load module. The vtable contains the target address and one copy of the target GP. This implies that it is not in text, and that it is generated by dld.

2. Detailed layout of the virtual table.

3. How can we share class offsets?

1. Scope and "State of the Art"

The following proposal applies only to calls to virtual functions when a this pointer adjustment is required from a base class to a derived class. Essentially, this means multiple inheritance, and the existence of two or more virtual table pointers (vptr) in the complete object. The multiple vptrs are required so that the layout of all bases is unchanged in the complete object. There will be one additional vptr for each base class which already required a vptr, but cannot be placed in the whole object so that it shares its vptr with the whole object. Note: when the vptr is shared, the base class is said to be the "primary base class", and there is only one such class.

For the primary base class, no pointer adjustment is needed. For all other bases, a pointer to the whole object is not a pointer to the base class, so whenever a pointer to the base class is needed, adjustment will occur.

In particular, when calling a virtual function, one does not know in advance in which class the function was actually defined. Depending on the actual class of the object pointed to, pointer adjustment may be needed or not, and the pointer adjustment value may vary from class to class. The existing solution is to have the vtable point not to the function itself, but to a "thunk" which does pointer adjustment when needed, and then jumps to the actual function. Another possibility is to have an offset in the vtable, which is used by the called function. However, more often than not, this implies adding zero.

Virtual bases make things slightly more complicated. In that case, the data layout is such that there is only one instance of the virtual base in the whole object. Therefore, the offset from a this pointer to a same virtual base may change along the inheritance tree. This is solved by placing an offset in the virtual table, which is used to adjust the this pointer to the virtual base.

2. Proposal and Rationale

My proposal is to replace thunks with offsets, with two additional tricks:

- Give a virtual function two entry points, so as to bypass the adjustment when it's known to be zero.
- Moving the adjustment at call-site, where it can be scheduled more easily, using a "reasonable" value, so that the adjustment is bypassed even more often.

The thunks are believed to cost more on IA64 than they would on other platforms. The reason is that they are small islands of code spread throughout the code, where you cannot guarantee any cache locality. Since they immediately follow an indirect branch, chances are we will always encounter both a branch misprediction and a I-cache miss in a row.

On the other hand, a virtual function call starts by reading the virtual function address. Reading the offset immediately thereafter should almost never cause a D-cache miss (cache locality should be good). More often than not, no adjustment is needed, or the adjustment will be done at call site correctly. In the worst case scenario, we perform two adjustments, one static at call site, and one dynamic in the callee, but this case should be really infrequent.

3. New Calling Convention

The new calling convention requires that the 'this' pointer on entry points to the class for which the virtual function is just defined. That is, for A::f(), the pointer is an A* when the main entry of the function is reached. If the actual pointer is not an A*, then an adjusting entry point is used, which immediately precedes the function.

In the following, we will assume the following examples:

```
struct A { virtual void f(); };
struct B { virtual void g(); };
struct C: A, B { }
struct D : C { virtual void f(); virtual void g(); }
struct E: Other, C { virtual void f(); virtual void g(); }
struct F: D, E { virtual void f(); }
```

```
void call_Cf(C *c) { c->f(); }
void call_Cg(C *c) { c->g(); }
void call_Df(D* d) { d->f(); }
void call_Dg(D* d) { d->g(); }
void call_Ef(E* e) { e->f(); }
void call_Eg(E* e) { e->g(); }
void call_Ff(F *ff) { ff->f(); }
void call_Fg(F *ff) { ff->g(); }    // Invalid: ambiguous
```

a) Call site:

The caller performs adjustment to match the class of the last overrider of the given function.

- call_Cf will assume that the pointer needs to be cast to an A*, since C::f is actually A::f. Since A is the primary base class, no adjustment is done at call site.
- call_Cg is similar, but assumes that the actual type is a B*, and performs the adjustment, since B is not the primary base class.
- call_Df and call_Dg will assume that the pointer needs to be cast to a D*, which is where D::f is defined. No adjustment is performed at call site.

b) Callee

- A::f and B::g are defined in classes where there is a single vptr. They don't define a secondary entry point. Because of call-site conventions, they expect to always be called with the correct type.
- D::f is defined in a class where there is more than one vptr, so it needs a secondary entry point and an entry 'convert_to_D' in the vtable. That's because it can be potentially called with either an A* or a B*. There are two vtables, one for A in D, one for B in D. The D::f entry in A in D points to the non-adjusting entry point, since A shares its vptr.
- D::g requires a secondary entry point, that will read the same offset 'convert_to_D' from the vtable.
- E also will require a 'convert_to_E' entry in the vtable, but this time, the vtable for A in C will have to point to an adjusting entry point, since A no longer shares the vptr with E (assuming Other has a vptr). This vtable is also the vtable of C in E.

c) Offsets in the vtable

Offsets have to be placed in the vtable at a position which does not conflict with any offset in the inheritance tree.

convert_to_D and convert_to_E are likely to be at the same offset in the vtable. This is not a problem, even if D and E are used in the same class, such as F, because this is the same offset in different vtables.

- call_Fg is invalid, because it is ambiguous.
- A notation such as ((E*) ff)->g() can be used to disambiguate, but in that case, we don't use the same vtable (either the E in F or D in F vtable). The E in F vtable uses that offset as 'convert_to_E', whereas the D in F vtable uses that offset as 'convert_to_D'.
- Similarly, call_Cf called with an F object will actually be called with the E in F or D in F, which disambiguates which C is actually used. The actual C* passed will have been adjusted by the caller unambiguously, or the call will be invalid.
- For functions overriden in F, an entry 'convert_to_F' is created anyway. This entry will not overlap with either convert_to_E or convert_to_D.

The fact that an offset is reserved does not mean that it is actually used. A vtable need to contain the offset only if it refers to a function that will use it. An offset of 0 is not needed, since the function pointer will point to the non-adjusting entry point in that case.

4. Cases where adjustment is performed

- For call_Cf: No adjustment is done at call site. No adjustment is done at callee site if the dynamic type is C, or D, or

D in F (that is, F casted to an E).

- For call_Cg: Adjustment to B* is done at call-site. No further adjustment is needed if the dynamic type is C, D, or D in F. On the other hand, a second adjustment may happen for an E or E in F, because C is not their primary base.

In other words, adjustment is made only when necessary, and at a place where it is better scheduled than with thunks. The only bad case is double adjustment for call_Cg called with an E*. This case can probably be considered rare enough, compared to calls such as call_Cg called with a C*, where we now actually do the adjustment at the call-site.

5. Comparing the code trails

Currently, the sequence for a virtual function call in a shared library will look as follows. I'm assuming +DD64, there would be some additional addp4 in +DD32. The trail below is the dynamic execution sequence. In bold and between #if/#endif, the affected code.

```
        // Compute the address of the vptr in the object,
        // from the this pointer
        // Optional, since vptroffset is often 0.
        // This also adjusts to the class of the final overrider
        addi            Rthis=vptroffset_of_final_overrider,Rthis
        ;;
        // Load the vptr in a register
        ld8             Rvptr=[Rthis]
        ;;
        // Add the offset to get to the function descriptor pointer
        // in the vtable.  Never zero, this instruction is always generated
        addi            Rfndescr=fndescroffset,Rvptr
        ;;
        // (Assuming inlined stub) Load the function address and new GP
        ld8             Rfnaddr=[Rfndescr],8
        ;;
        // Load the new GP
        ld8             GP=[Rfndescr]
        mov             BRn=Rfnaddr
        ;;
        // Perform the actual branch to the target

        // ...
        // ... Branch misprediction almost always, followed by
        // ... I-Cache miss almost always if jumping to a thunk
        br.call B0=BRn

#if OLD_ADJUST
thunk_A::f_from_a_B:
        // If the 'adjustment_from_B_to_A is the 'adjustment_to_A' above,
        // then in the new case, the vtable directly points to A::f
        addi            Rthis,adjustment_from_B_to_A

        // In most cases, we can probably generate a PC-relative branch here
        // It is unclear whether we would correctly predict that branch
        // (since it is assumed that we arrive here immediately following
        // a misprediction at call site)
        br              A::f
#endif // OLD_ADJUST

// This occurs less often than OLD_ADJUST
// (it does not happen when call-site adjustment is correct)
#if NEW_ADJUST
adjusting_entry_A::f
        // Can't be executed in less than 3 cycles?
        addi            Rvptr=class_adjustment_offset,Rvptr
        ;;
        // This loads data which is close to the fn descriptor,
        // so it's likely to be in the D-cache
        ld8             Rvptr=[Rvptr]
        ;;
        add             Rthis=Rthis,Rvptr
#endif

A::f:
        alloc   ...
```

[990812 All] Discussion of B-6 raises questions of impact on the above approach. Christophe will look at the issues.

[990826 Cygnus -- Jason] [An alternative suggestion from Jason via email.]

Rather than per-function offsets, we have per-target type offsets. These offsets (if any) are stored at a negative index from the vptr. When a derived class D overrides a virtual function F from a base class B, if no previously allocated offset slot can be reused, we add one to the beginning of the vtable(s) of the closest base(s) which are non-virtually derived from B. In the case of non-virtual inheritance, that would be D's vtable; in simple virtual inheritance, it would be B's. The vtables are written out in one large block, laid out like an object of the class, so if B is a non-virtual base of D, we can find the D vtable from the B vptr.

D::f then recieves a B*, loads the offset from the vtable, and makes the adjustment to get a D*. The plan is to also have a non-adjusting vtable entry in D's vtable, so we don't have to do two adjustments to call D::f with a D*; the implementation of this is up to the compiler. I expect that for g++, we will do the adjustment in a thunk which just falls into the main function.

The performance problems with classic thunks occur when the thunk is not close enough to the function it jumps to for a pc-relative branch. This cannot be avoided in certain cases of virtual inheritance, where a derived class must whip up a thunk for a new adjustment to a method it doesn't override.

In this case, we will only ever have one thunk per function, so we don't even have to jump. Except in the case of covariant returns, that is, where we will have one per return adjustment. But we know all necessary adjustments at the point of definition of the function, so they can all be within pc-relative branch range.

[Extensive discussion followed by email -- this suggestion is not completely correct, but may be the basis of a workable solution.]

[990831 Cygnus -- Ian] A couple of observations ...

On the state of the art:

The Microsoft approach is worth mentioning. (I haven't seen it discussed -- though perhaps that is because of the patent situation.)

It allows zero-adjusting (i.e. non-thunking) calls for (almost) every virtual function call in a non-virtual, multiple inheritance hierarchy.

For those that are unfamiliar, the idea is that all calls go via the base class vft and overriding functions expect a pointer to the base class type. (That is, if D::f overrides B::f, it expects the first parameter to be of type B*, not D*.) The callee does the necessary static adjustment to get to the derived class 'this' pointer as needed.

It avoids requiring a thunk, and it's often the case that the cost is zero in the callee because the this-adjustment can be folded into other offset computations.

On the balance, it could well win over all the other approaches being discussed here. [Though, it may lose in some specific cases vs. Christophe's approach where one would create additional extra entries in the derived class vft.]

On when to make extra virtual function table entries for functions:

One of Cristophe's suggestions is sort-of separate from the rest of the discussion: making extra entries in the derived class' vft for some overridden virtual functions. It has the benefit of giving you a faster calls if you happen to be in (or near) the derived class -- at the expense of space in the vft.

Of course, you can always make the call through the introducing base class, so these extra entries are a pure space/time performance trade off (w/ some unpredictable D-cache effects) and the cost/benefit analysis will depend a little on what the rest of the strategy looks like.

The same idea is potentially applicable, no matter what strategy you actually use for vft layout, and different criteria for deciding what extra entries to make are possible. For example, creating an extra entry when overriding a function introduced in a virtual base has the added benefit of avoiding a cast to a virtual base at the call site.

[990909 All] We are getting closer -- understanding of the alternatives is improving, and Christophe may agree with the Jason/Brian proposal after more thought. To make sure we really understand what we're agreeing to, Jason and Christophe will write up more precise proposal(s).

#	Issue	Class	Status	Source	Opened	Closed
B-2	Covariant return types	call	closed	SGI	990520	990722
Summary: There are several methods for adjusting the 'this' pointer of the returned value for member functions with covariant return types. We need to decide how this is done. Return thunks might be especially costly on IA64, so a solution based on returning multiple pointers may prove more interesting.						
Resolution: Provide a separate Vtable entry for each return type.						

#	Issue	Class	Status	Source	Opened	Closed
B-3	Allowed caching of vtable contents	call	closed	HP	990603	990805
Summary: The contents of the vtable can sometimes be modified, but the concensus is that it is nonetheless always allowed to "cache" elements, i.e. to retain them in registers and reuse them, whenever it is really useful. However, this may sometimes break "beyond the standard" code, such as code loading a shared library that replaces a virtual function. Can we all agree when caching is allowed?						
Resolution : Caching is allowed within a member function.						

#	Issue	Class	Status	Source	Opened	Closed
B-4	Function descriptors in vtable	data	closed	HP	990603	990805
Summary: For a runtime architecture where the caller is expected to load the GP of the callee (if it is in, or may be in, a different DSO), e.g. HP/UX, what should vtable entries contain? One possibility is to put a function address/GP pair in the vtable. Another is to include only the address of a thunk which loads the GP before doing the actual call.						
Resolution : The Vtable will contain a function address/GP pair.						

#	Issue	Class	Status	Source	Opened	Closed
B-5	Where are vtables emitted?	data	open	HP	990603	
Summary: In C++, there are various things with external linkage that can be defined in multiple translation units, while the ODR requires that the program behave as if there were only a single definition. From the user's standpoint, this applies to inlines and templates. From the implementation's perspective, it also applies to things like vtables and RTTI info. (We call this *vague linkage*.)						

[990624 Cygnus -- Jason] There are several ways of dealing with vague linkage items:

1. Emit them everywhere and only use one.
2. Use some heuristic to decide where to emit them.
3. Use a database to decide where to emit them.

4. Generate them at link time.

#3 and #4 are feasible for templates, but I consider them too heavyweight to be used for other things.

The typical heuristic for #2 is "with the first non-inline, non-abstract virtual function in the class". This works pretty well, but fails for classes that have no such virtual function, and for non-member inlines. Worse, the heuristic may produce different results in different translation units, as a method could be defined inline after being declared non-inline in the class body. So we have to handle multiple copies in some cases anyway.

The way to handle this in standard ELF is weak symbols. If all definitions are marked weak, the linker will choose one and the others will just sit there taking up space.

Christophe mentioned the other day that the HP compiler used the typical heuristic above, and handled the case of different results by encoding the key function in the vtable name. But this seems unnecessary when we can just choose one of multiple defns.

A better solution than weak symbols alone would be to set things up so that the linker will discard the extra copies. Various existing implementations of this are:

1. The Microsoft PE/COFF defn includes support for COMDAT sections, which key off of the first symbol defined. One copy is chosen, others are discarded. You can specify conditions to the linker (must have same contents, must have same size).

2. The IBM XCOFF platform includes a garbage-collecting linker; sections that are not referenced in a sweep from main are discarded. In xlC, template instantiations are emitted in separate sections, with encoded names; at link time, one copy is renamed to the real mangled name, and the others are discarded by garbage collection.

The GNU ELF toolchain does a variant of #1 here; any sections with names beginning with ".gnu.linkonce." are treated as COMDAT sections. It seems more sensible to me to key off of the section name than the first symbol name as in PE.

The GNU linker recently added support for garbage collection, and I've been thinking about changing our handling of vague linkage to make use of it, but haven't.

I propose that the ia64 base ABI be extended to provide for either COMDAT sections or garbage collection, and that we use that support for vague linkage.

I further propose that we not use heuristics to cut down the number of copies ahead of time; they usually work fine, but can cause problems in some situations, such as when not all of the class's members are in the same symbol space. Does the ia64 ABI provide for controlling which symbols are exported from a shared library?

A side issue: What do we want to do with dynamically-initialized variables? The same thing, or use COMMON? I propose COMMON.

See also G-3, for vague linkage of inlined routines and their static variables.

[990624 SGI summarizing others] HP uses COMDAT for many cases, keying from the symbol names. HP also uses some heuristics. HP observes that IA-64 objects will already be large. From the base ABI discussions, any use of WEAK or COMMON symbols will need to take care not to depend on vendor-specific treatment.

Defining a COMDAT mechanism doesn't preclude using heuristics to avoid some copies up front. A COMDAT mechanism should also specify how to get rid of associated sections like debugging info, unless the identical mechanism works.

[990629 HP -- Christophe] First, the "usual" heuristic (which is usual because it dates back to Cfront) is to emit vtables in the translation unit that contains the definition of the first non inline, non pure virtual function. That is, for:

```
        struct X {
                void a();
                virtual void f() { return; }
                virtual void g() = 0;
                virtual void h();
                virtual void i();
        };
```

the vtable is emitted only in the TU that contains the definition of h().

This breaks and becomes non-portable if:

- There is no such thing. In that case, you generally emit duplicate versions of vtables
- There is a "change of mind", such as having the above class followed by:

```
        inline void X::h() { f(); }
```

Now, the COMDAT issue is as follows: a COMDAT section is, in some cases, slightly more difficult to handle (at least, that's the impression Jason gave me). For statics with runtime initialization, what you can do is reserve COMMON space ('easier'), then initialize that space at runtime. As I said, the problem is if two compilers disagree on whether this is a runtime or a compile time initialization, such as in :

```
        int f() { return 1; }
        int x = f();    // Static (COMDAT) or Dynamic (COMMON) initialization?
```

So I personally recommend that we put everything in COMDAT.

[990715 All] Consensus so far: use a heuristic for vtable and typeinfo emission, based on the definition of the key function. (The first virtual function that is not declared inline in the class definition.) The vtable must be emitted where the key function is defined, it may also be emitted in other translation units as well. If there is no key function then the vtable must be emitted in any translation unit that refers to the vtable in any way.

Implication: the linker must be prepared to discard duplicate vtables. We want to use COMDAT sections for this (and for other entities with vague linkage.)

Open issue: the elf format allows only 16 bits for section identifiers, and typically two of those bits are already taken up for other things. So we've only got 16k sections available, which is unacceptable if we're creating lots of small sections.

Jason - COMDATs disappear into text and data at link time, so the issue is really only serious if we've got more than 16k vtables (or template instantiations, etc.) in a single translation unit.

Daveed - HP has gotten around this problem by hacking their ELF files to steal another 8 bits from somewhere else.

Jack - a new kind of section table could be a viable solution. However, it would break everything if we did it for ia32. Is a solution that only works on ia64 acceptable? Note also that the elf section table has its own string table, which we wouldn't be able to share with the new kind of section table. Index and link fields often point into section table, we would have to figure out how to deal with this. (Jack is not opposed to the idea of an alternate section table, he is just pointing out some of the issues we will have to resolve.)

[990805 All] We need a specific proposed representation for COMDAT. IBM's version is restricted to one symbol per section. Jim will look for Microsoft's PECOFF definition. Anyone else with a usable definition should send it.

#	Issue	Class	Status	Source	Opened	Closed
B-6	Virtual function table layout	data	open	SGI	990520	
Summary: What is the layout of the Vtable?						

[990624] Issue split from A-1.

[990630 HP - Christophe]

Here is a short description of the virtual tables layout. This is mostly taken out of the existing Taligent runtime.

Virtual tables

Virtual tables (vtables) are used to dispatch virtual functions, to access virtual base class subobjects, and to access information for runtime type identification (RTTI). Each class that has virtual member functions or virtual bases has an associated set of vtables. The vtable pointers within all the objects (instances) of a class point to the same set of vtables.

Vtable layout

Each vtable consists of the following parts, in the order listed. All offsets in a vtable are of type ptrdiff_t.

"Virtual base offsets" are used to access the virtual bases of an object and to non virtual bases when pointer adjustment is required for some overriden virtual function. Each entry is a displacement to a virtual base subobject from the location within the object of the vtable pointer that addresses this vtable. These entries are only necessary if the class directly or indirectly inherits from virtual base classes, or if it overrides a virtual function defined in a base class and if adjustment from the based to the derived class is needed. The values can be positive or negative.

"Offset to top" holds the displacement to the top of the object from the location within the object of the vtable pointer that addresses this vtable. A negative value indicates the vtable pointer is part of an embedded base class subobject; otherwise it is zero. The offset provides a way to find the top of the object from any base subobject with a vtable pointer. This is necessary for dynamic_cast in particular. The virtual pointer points to the "offset to top" field.

"Typeinfo pointer" points to the typeinfo object used for RTTI. All entries in each of the vtables for a given class point to the same typeinfo object.

"Duplicate base information pointer" points to a table used to perform runtime disambiguation of duplicate base classes for dynamic casts. The entries in each of the vtables for a given class do not all point to the same table of duplicate base information.

Function pointers are used for virtual function dispatch. Each pointer holds either the address of a virtual function of the class, or the address of a secondary entry point that performs certain adjustments before transferring control to a virtual function. [To be discussed] In the case of shared library builds, [Solution 1] a function pointer entry contains the address of a function descriptor, which contains the target GP value and the actual function address [Solution 2] a function pointer entry contains two words: the value of the target GP value and the actual function address [Solution 3] a function pointer points to an import table generated by the dynamic loader, which will transfer control to the target function.

The function pointers are grouped so that the entries for all virtual functions introduced by a base class are kept together, in declaration order. Function pointers from a derived class immediately follow those for the base class. The order of function pointers therefore depends on the static type of subobject whose vtable pointer points to that virtual table.

A vtable pointer in an object addresses the "offset to top" field of a vtable. This location is known as the address point of the vtable. Note that the virtual base offsets are at negative displacements from the address point.

The typeinfo and duplicate base information are covered in separate discussions.

Types of vtables

The following categories describe the rules for how vtables of a class are constructed from the vtables of its bases.

Category 1

This category is for a class that inherits no base classes. If the class declares no virtual functions, then it has no vtable, and its objects have no vtable pointers. If the class declares virtual functions, then it has a single vtable containing RTTI fields followed by function pointer entries. There is one function pointer entry for each virtual function declared in the class. The vtable pointer in an object of the class addresses this vtable.

Category 2

This category is for a class that inherits only non-virtual base classes, and the non- virtual base classes have no base classes, and there is no adjustment in any of the overriden virtual functions.

If none of the base classes have vtables, a vtable is constructed for the class according to the same rules as in Category 1. Otherwise the class has a vtable for each base class that has a vtable. The class's vtables are constructed from copies of the base class vtables. The entries are the same, except:

- The RTTI fields contain information for the class, rather than for the base class.
- The function pointer entries for virtual functions inherited from the base class and overridden by this class are replaced with the addresses of the overridden functions (or the corresponding adjustor secondary entry points),
- At negative offsets, offsets to the base classes are generated if used by adjustor secondary entry points.

Informally, each of these vtables has a name in the form Base-in-Derived vtable, where Base is a base class and Derived is the derived class. Each vtable pointer in an object addresses one of these vtables for the class. The vtable pointer of an A subobject within a B object would address the A-in-B vtable.

The vtable copied from the primary base class is also called the primary vtable; it is addressed by the vtable pointer at the top of the object. The other vtables of the class are called secondary vtables; they are addressed by vtable pointers inside the object.

Aside from the function pointer entries that correspond to those of the primary base class, the primary vtable holds the following additional entries at its tail: - Entries for virtual functions introduced by this class - Entries for overridden virtual functions not already in the vtable. (These are also called replicated entries because they are already in the secondary vtables of the class.) [I wonder if this is actually needed, this seems to be a consequence of RRBC]

The primary vtable, therefore, has the base class functions appearing before the derived class functions. The primary vtable can be viewed as two vtables accessed from a shared vtable pointer.

Note Another benefit of replicating virtual function entries is that it reduces the number of this pointer adjustments during virtual calls. Without replication, there would be more cases where the this pointer would have to be adjusted to access a secondary vtable prior to the call. These additional cases would be exactly those where the function is overridden in the derived class, implying an additional thunk adjustment back to the original pointer. Thus replication saves two adjustments for each virtual call to an overridden function introduced by a non-primary base class.

Category 3

This category is for a class that inherits only virtual base classes, and the virtual base classes have no base classes.

The class has a vtable for each virtual base class that has a vtable. These are all secondary vtables and are constructed from copies of the base class vtables according to the same rules as in Category 2. The vtable pointers of the virtual base subobjects within the object address these vtables.

The class also has a vtable that is not copied from the virtual base class vtables. This vtable is the primary vtable of the class and addressed by the vtable pointer at the top of the object, which is not shared. It holds the following function pointer entries:

- Entries for virtual functions introduced by this class
- Entries for overridden virtual functions. (These are also called replicated entries, because they are already in the secondary vtables of the class)

The primary vtable also has virtual base offset entries to allow finding the virtual base subobjects. There is one virtual base offset entry for each virtual base class. For a class that inherits only virtual bases, the entries are in the reverse order in which the virtual bases appear in the class declaration, that is, the entry for the leftmost virtual base is closest to the address point of the vtable.

Category 4

This category is for a class that directly or indirectly inherits base classes that are either virtual or non-virtual.

The rules for constructing vtables of the class are a combination of the rules from Categories 2 and 3, and for the most part can be determined inductively. However the rules for placing virtual base offset entries in the vtables requires elaboration.

The primary vtable has virtual base offset entries for all virtual bases directly or indirectly inherited by the class. Each secondary vtable has entries only for virtual bases visible to the corresponding base class. The entries in the primary vtable are ordered so that entries for virtual bases visible to the primary base class appear below entries for virtual bases only visible to this class.

For virtual bases only visible to this class, the entries are in the reverse order in which the virtual bases are encountered in a depth-first, left-to-right traversal of the inheritance graph formed by the class definitions. Note that this does not follow the order that virtual bases are placed in the object.

Vtables for partially constructed objects

In some situations, a special vtable, called a construction vtable is used during the execution of base class constructors and destructors. These vtables are for specific cases of virtual inheritance.

During the construction of a class object, the object assumes the type of each of its base classes, as each base class subobject is constructed. RTTI queries in the base class constructor will return the type of the base class, and virtual calls will resolve to member functions of the base class rather than the complete class. Normally, this behavior is accomplished by setting, in the base class constructor, the object's vtable pointers to the addresses of the vtables for the base class.

However, if the base class has direct or indirect virtual bases, the vtable pointers have to be set to the addresses of construction vtables. This is because the normal base class vtables may not hold the correct virtual base index values to access the virtual bases of the object under construction, and adjustment addressed by these vtables may hold the wrong this parameter adjustment if the adjustment is to cast from a virtual base to another part of the object. The problem is that a complete object of a base class and a complete object of a derived class do not have virtual bases at the same offsets.

A construction vtable holds the virtual function addresses and the RTTI information associated with the base class and the virtual base indexes and the addresses of adjustor entry points with this parameter adjustments associated with objects of the complete class.

To ensure that the vtable pointers are set to the appropriate vtables during base class construction, a table of vtable pointers, called the VTT, which holds the addresses of construction and non-construction vtables is generated for the complete class. The constructor for the complete class passes to each base class constructor a pointer to the appropriate place in the VTT where the base class constructor can find its set of vtables. Construction vtables are used in a similar way during the execution of base class destructors.

[990701 All] The above arrived to late for everyone to read it carefully. It was agreed that we would consider it outside the meetings, discuss any issues noted by email, and attempt to close on 22 July. (Christophe is on vacation until that week, and Daveed leaves on vacation the next week.)

[990811 SGI -- Jim] I've put a reworked version of Christophe's writeup in the data layout document, along with a number of questions it raises.

[990812 All] Extensive discussion of this issue produced the observations that

- The number of virtual base offsets changes for vtables embedded in derived vtables.
- Therefore, one cannot reference one by a compile-time-constant offset from another (within the set associated with a type).
- Therefore, one cannot omit vfunc pointers from a derived vtable just because they appear in one of the base class vtables.

Christophe will look at the implications of these observations. Others should too.

[990820 IBM -- Brian]

Re: vtable layout, sharing vtable offsets

I'm going to write the exam on this to see how well I am understanding the issue.

If I understand it correctly, the proposal under consideration is tied to the decision to replicate virtual function entries in vtables. It requires replicating in the vtable for base class B all virtual functions that are overridden in B; more replication that this implies will be wasted since a function is always called through a vtable of an introducing or overriding class.

When a non-pure virtual function X::f() is compiled it is possible to determine whether it requires a secondary entry point. It will require one if that function may be virtually called (i.e., is the final overrider) in any class in which f() appears in more than one vtable; this needs to be decidable knowing only X. A rule that works is: X::f() overrides one or more f()'s from base classes of X, and either one or more of those base classes are virtual or X fails to share its vptr with all instances of them.

[Though a virtual base may happen to share its vptr with X in an object of complete type X, that relationship may fail to hold in further derived classes, so we need to generate the secondary entry point just in case.] ["Sharing a vptr" is the condition under which no adjustment is necessary; if the bases involved are all nonvirtual then subsequent class derivation won't change this.]

Each vtable that requires a nonzero adjustment will have a "convert to X" offset mixed in with its virtual base offsets. It is necessary that a "convert to X" appears in the same position in each vtable that references X::f()'s secondary entry; it is desirable that the "convert to X" also be unique in each vtable.

Assume that X has nonvirtual nonprimary bases Nx (x=1,2,...), and virtual bases Vx, all of which have a virtual f(). Then vtables for Nx in X, or in anyclass derived from X that does not further override f(), will reference X::f()'s secondary entry. Vtables for Vx in X or any derived class where Vx does not share a vptr with X, will also reference X::f()'s secondary entry; note this will occur in a construction vtable even if the derived class does further override f().

The question, then, is whether a position for the "convert to X" offset can be chosen, knowing only X and its parentage, that can be used consistently in all those vtables and that won't collide with a "convert to Y" position chosen on account of some other hierarchy where Y::g() overrides an Nx::g() or Vx::g().

If Y derives from X, we will be able to select a "convert to Y" position that doesn't conflict, so we can restrict our attention to cases where X and Y are unrelated. Also, if the base involved is nonvirtual (Nx) then we are safe, because no instance of Nx will be a subobject of both X and Y, so no Nx vtable will require both "convert to X" and "convert to Y" offsets.

The remaining case is where X and Y are unrelated but both have a virtual base Vx:

```
struct V1 { virtual void f();  virtual void g(); };
struct Other1 { virtual void ignore1(); }
struct X : Other1, virtual V1 { virtual void f(); }

struct Y : Other1, virtual V1 { virtual void g(); }

struct ZZ: X, Y { }
```

The vtable for N1 in ZZ does require both offsets. The only way I see to accomplish this is to preallocate an adjustment slot for each virtual function in V1. That is, X::f() uses the first slot position, and Y::g() the second, based on the order that f() and g() are declared in V1. This only needs to be done in hierarchies where V1 is virtual, but the same offset has to be used for any Nx tables in X too.

Is this close?

Re: Concatenating vtables

I don't understand the comment that varying numbers of virtual base offsets make it impossible to concatenate vtables and refer to them via a single symbol. The only code that refers by name to X's vtable and the vtables of N1 in X etc. is X's constructor and destructor, and maybe some derived classes that find they are able to reuse some pieces. All that code is aware of X's declaration and can map out its tables. What am I missing?

[990826 All] There is still considerable confusion about what will work. Key questions are (1) whether member functions can share offsets to base classes, or each need their own; and (2) when we need a no-this-adjustment override entry.

[990901 SGI -- Jim] Being confused myself by all the discussion, I've constructed a new page containing (initially) an example of a class hierarchy supplied by Christophe, and attempted to identify possible function calls, the class data layout, and the class vtable layout based on Christophe's original proposal. Please provide corrections, and if you're proposing alternative vtable constructions, describing them for this example might help (me, at least). Also feel free to provide additional examples illustrating other points.

<table>
<tr><th>#</th><th>Issue</th><th>Class</th><th>Status</th><th>Source</th><th>Opened</th><th>Closed</th></tr>
<tr><td>B-7</td><td>Objects and Vtables in shared memory</td><td>data</td><td>closed</td><td>HP</td><td>990624</td><td>990805</td></tr>
<tr><td colspan="7"><b>Summary</b>: Is it possible to allocate objects in shared memory? For polymorphic objects, this implies that the Vtable must also be in shared memory.</td></tr>
<tr><td colspan="7"><b>Resolution</b> : No special representation is useful in support of shared memory.</td></tr>
</table>

<table>
<tr><th>#</th><th>Issue</th><th>Class</th><th>Status</th><th>Source</th><th>Opened</th><th>Closed</th></tr>
<tr><td>B-8</td><td>dynamic_cast</td><td>data</td><td>open</td><td>SGI</td><td>990628</td><td></td></tr>
<tr><td colspan="7"><b>Summary</b>: What information to we put in the vtable to enable (a) dynamic_cast from pointer-to-base to pointer-to-derived (including detection of ambiguous base classes) and (b) dynamic_cast to void*?</td></tr>
</table>

[990701 All] This should be part of the proposal Daveed will put together by the 15th (action #13); the group will discuss it on the 22nd.

[990812 Sun -- Michael] Sun has provided a description, in a separate page, describing their implementation. They are filing for a patent on the algorithms described.

Object Construction/Destruction Issues

#	Issue	Class	Status	Source	Opened	Closed
C-1	Interaction with .init/.fini	lif ps	open	SGI	990520	

Summary: Static objects with dynamic constructors must be constructed at intialization time. This is done via the executable object initialization functions that are identified (in ELF) by the DT_INIT and DT_INITARRAY dynamic tags. How should the compiler identify the constructors to be called in this way? One traditional mechanism is to put calls in a .init section. Another, used by HP, is to put function addresses in a .initarray section.

The dual question arises for static object destructors. Again, the extant mechanisms include putting calls in a .fini section, or putting function addresses in a .finiarray section.

Finally, which mechanism (DT_INIT or DT_INITARRAY, or the FINI versions) should be used in linked objects? The gABI, and the IA-64 psABI, will support both, with DT_INIT being executed before the DT_INITARRAY elements.

#	Issue	Class	Status	Source	Opened	Closed
C-2	Order of ctors/dtors w.r.t. link	lif ps	open	HP	990603	

Summary: Given that the compiler has identified constructor/destructor calls for static objects in each relocatable object, in what order should the static linker combine them in the linked executable object? (The initialization order determines the finalization order, as its opposite.)

[990610 All] Meeting concensus is that the desirable order is right to left on the link command line, i.e. last listed relocatable object is initialized first.

[990701 SGI] We propose that global constructors be handled as follows:

- The compiler shall emit global constructor calls as one or more entries in an SHF_INIT_ARRAY section.
- The linker shall combine them according to the rules of the base gABI, namely as a concatenated array of entries, in link argument order, pointed to by a DT_INIT_ARRAY tag. (The linker may intersperse entries from command line flags or modules from other languages, but that is beyond the C++ ABI scope.)

This does not address the global destructor problem. That solution needs to deal not only with the global objects seen by the compiler, but also interspersed local static objects. This treatment seems to be tied up in the question of how early unloading of DSOs is handled, and the data structure used for that purpose (issue C-3).

[990715 All] Cygnus scheme: priorities are 16-bit unsigned integers, lower numbers are higher priority. In each translation unit, there's a single initialization function for each priority. Anything that's prioritized has a higher priority than anything that isn't explicitly assigned a priority.

IBM scheme: priorities are 32-bit signed integers, higher numbers are higher priority. Something that isn't explicitly assigned a priority effectively gets a priority of 0.

Consensus: nobody is sure that negative priorities are very important, but also nobody can think of a reason not to allow them. We accept the idea that priorities are 32-bit signed integers. On a source level Cygnus will keep lower numbers as higher priority, but that's a source issue, not an ABI issue.

Status: No real technical issues, we have consensus on everything that matters. We need to write up the finicky details.

[990722 all] It was decided to follow the IBM approach, including:

- The source pragma will use a 32-bit signed priority. The default will map to 0, and larger numbers are lower priority.
- Priorities MIN_INT .. MIN_INT+1023 are reserved to the implementation.
- The object representation will use a 32-bit unsigned priority, obtained from the source priority by subtracting INT_MIN.
- Initialization priorities are only relevant within a DSO. Between DSOs, the normal ELF ordering based on object order applies.

To be resolved are the precise source pragma definition (possibly IBM's), and the ELF file representation.

[990729 all] SGI suggested an object representation involving (in relocatables) a new section type, containing pairs <priority, entry address>. The linker would merge all such sections, include any initialization entries specified by other means, and leave one or more DT_INITARRAY entries for normal runtime initialization, either building a routine to call the entries, or referencing a standard runtime routine.

IBM noted that they combine their equivalent data structures in the linker, but don't sort them, leaving that to a runtime routine. This can be done without explicit linker support, but involves runtime overhead.

Cygnus suggested that if we are going to require linker sorting, we should make the facility more general.

Jim will write up a more precise proposal.

[990804 SGI -- Jim]

Proposal

My objectives are:

- Simple representation in relocatable objects.
- No new representation in executable objects.
- Simple static linker processing (general if possible).
- Minimal unnecessary runtime cost.
- Minimal library interface.
- Integration with other initialization (at source priority zero).

Object File Representation

Define a new section type, e.g. **SHT_CXX_PRIORITY_INIT**. Its elements are structs:

```
typedef struct {
  ElfXX_Word    pi_pri;
  ElfXX_Addr    pi_addr;
} ElfXX_Cxx_Priority_Init;
```

The semantics are that **pi_addr** is a function pointer, with an **unsigned int** priority parameter, which performs some initialization at priority **pi_pri**. Each of these functions will be called with the GP of the executable object containing the table. The section header field **sh_entsize** is 8 for ELF-32, or 16 for ELF-64.

Runtime Library Support

Each implementation shall provide a runtime library function with prototype:

```
void __cxx_priority_init ( ElfXX_Cxx_Priority_Init *pi, int cnt );
```

It will be called with the address of a **cnt**-element (sub-)vector of the priority initialization entries, and will call each of them in order. It will be called with the GP of the initialization entries.

Linker Processing

The linker must take the collection of SHT_CXX_PRIORITY_INIT section entries from the relocatable object files being linked, and other initialization tasks specified in other ways (and treated as source priority 0 or object priority -MIN_INT), and produce an executable object file which executes the initialization tasks in priority order using only **`DT_INIT`**, **`DT_INIT_ARRAY`**, and **`__cxx_priority_init`**. Priority order is first according to the priority of the task, and then according to the order of relocatable objects and options in the link command. The order of tasks specified by other methods, relative to SHT_CXX_PRIORITY_INIT tasks of priority zero, is implementation defined. There are several possible implementations. Two extremes are:

- The linker sorts the SHT_CXX_PRIORITY_INIT sections together. If it inserts entries for initialization tasks specified in other ways, it may make a single DT_INIT_ARRAY entry pointing to __cxx_priority_init. If not, it must break it into subranges, interspersing DT_INIT_ARRAY entries for the other tasks with entries for the SHT_CXX_PRIORITY_INIT entries. (This implementation will minimize runtime overhead.)

- The linker simply appends the SHT_CXX_PRIORITY_INIT sections. It inserts DT_INIT_ARRAY entries before and after the entries for other initialization tasks which sort this vector and then execute the negative-priority calls on the first call, and the positive-priority ones on the second call. (I believe this is much like today's IBM implementation.) However, to be conforming, the routine which performs these tasks must be linked with the resulting executable object, or shippable with it as an associated DSO.

Note that if one is linking ELF-32 objects into a 64-bit program, the entries must be expanded as part of this process.

Sorting Sections

Jason suggested that if we base this feature on sorting sections, we should provide a general mechanism. Following is a proposal for that purpose.

Define a new section header flag, **`SHF_SORT`**. If present, the linker is required to sort the elements of the concatenated sections of the same type, where the elements are determined by **`sh_entsize`**. The sort is controlled by fields in **`sh_info`**:

`#define SH_INFO_KEYSIZE(info) (info & 0xff)`
: The size of the sort key (bytes).

`#define SH_INFO_KEYSTART(info) ((info>>8) & 0xff)`
: The start byte of the sort key within element, from 0.

`#define SH_INFO_SORTKIND(info) ((info>>16) & 0xf)`
: The kind of sort data: 0 for unsigned integer, 1 for signed integer.

The sort must be stable. The sort key must be naturally aligned.

Other conceivable options would be to allow sorting strings (like SHF_MERGE, this would be indicated by setting SHF_STRING and putting the character size in **`sh_entsize`**), or floating point data. Also, note that if we don't anticipate using such a general mechanism, it becomes possible to avoid padding words in the ELF-64 format by separating the priority and address vectors.

[990810 HU-B -- Martin] Global destructor ordering must not only interleave with static locals, but also with atexit. This gives two problems: atexit is only guaranteed to support 32 functions; and dynamic unloading of DSOs break when functions are atexit registered.

[990810 SGI -- Matt] Yes, the interleaving is required by the C++ standard. It's a nuisance, and I don't think there's any good reason for it, but the requirement is quite explicit.

The relevant part of the C++ standard is section 3.6.3, paragraph 3:

> *"If a function is registered with atexit (see , 18.3) then following the call to exit, any objects with static storage duration initialized prior to the registration of that function shall not be destroyed until the registered function is called from the termination process and has completed. For an object with static storage duration constructed after a function is registered with atexit, then following the call to exit, the registered function is not called until the execution of the object's destructor has completed. If atexit is called during the construction of an object, the complete object to which it belongs shall be destroyed before the registered function is called."*

What this implies to me is that atexit, and the part of the runtime library that handles destructors for static objects, must know about each other.

[990812 All] Some people would prefer a sorting scheme based on the section name instead of the data, and also less linker impact. Jim will look into alternatives.

#	Issue	Class	Status	Source	Opened	Closed
C-3	Order of ctors/dtors w.r.t. DSOs	ps	open	HP	990603	

Summary: Given the constructor/destructor calls for each executable object comprising a program, what is the order of execution between objects? For constructors, there is not much question: unless we choose some explicit means of control, file-scope objects will be initialized by the DT_INIT/DT_INITARRAY functions in the order determined by the base ABI order rules, and local objects will be initialized in the order their containing scopes are entered.

For destructors, the Standard requires opposite-order destruction, which implies a runtime structure to keep track of the order. Furthermore, the potential for dynamic unloading of a DSO (e.g. by dlclose) requires a mechanism for early destruction of a subset.

[990804 SGI -- Jim]

Proposal

My objectives are:

- Simple library interface.
- Efficient handling during construction.
- Standard-conforming treatment during normal program exit.
- Reasonable treatment during early DSO unload (e.g. dlclose).
- Minimal dynamic and static linker impact.

Runtime Data Structure

The runtime library shall maintain a list of termination functions with the following information about each:

- A function pointer (a pointer to a function descriptor on IA-64).
- A void* operand to be passed to the function.
- A void* handle for the *home DSO* of the entry (below).

The representation of this structure is implementation defined. All references are via the API described below.

Runtime API

A. Object construction:

   When a global or local static object is constructed, which will require destruction on exit, a termination function is *registered* as follows:

   ```
   int __cxx_atexit ( void (*f)(void *), void *p, dso_handle d );
   ```

   This registration, e.g. `__cxx_atexit(f,p,d)`, is intended to cause the call `f(p)` when DSO d is unloaded, before all such termination calls registered before this one. It returns zero if registration is successful, nonzero on failure. Should we use exceptions instead?

   The registration function is called separate from the constructor.

B. User atexit calls:

   When the user registers exit functions with `atexit`, they should be registered with NULL parameter and DSO handle, i.e.

   ```
   __cxx_atexit ( f, NULL, NULL );
   ```

   Should we also allow user registration with a parameter? With a home DSO?

C. Termination:

When linking any DSO containing a call to **`__cxx_atexit`**, the linker should define a hidden symbol **`__dso_handle`**, with a value which is an address in one of the object's segments. (It doesn't matter what address, as long as they are different in different DSOs.) It should also include a call to the following function in the FINI list (to be executed first):

```
void __cxx_finalize ( dso_handle d );
```

The parameter passed should be **`__dso_handle`**.

Note that the above can be accomplished either by explicitly providing the symbol and call in the linker, or by implicitly including a relocatable object in the link with the necessary definitions, using a .fini_array section for the FINI call. Also, note that these can be omitted for an object with no calls to **`__cxx_atexit`**, but they can be safely included in all objects.

Finally, a main program should be linked with a FINI call to **`__cxx_finalize`** with NULL parameter.

When **`__cxx_finalize(d)`** is called, it should walk the termination function list, calling each in turn if **`d`** matches **`__dso_handle`** for the termination function entry. If **`d == NULL`**, it should call all of them. Multiple calls to **`__cxx_finalize`** should not result in calling termination function entries multiple times; the implementation may either remove entries or mark them finished.

Issue: By passing a NULL-terminated vector of DSO handles to **`__cxx_finalize`** instead of one, we could deal with unloading multiple DSOs at once. However, **`dlclose`** closes one at a time, so I'm not sure the extra complexity is worthwhile.

Since **`__cxx_atexit`** and **`__cxx_finalize`** must both manipulate the same termination function list, they must be defined in the implementation's C++ runtime library, rather than in the individual linked objects.

<table>
<tr><th>#</th><th>Issue</th><th>Class</th><th>Status</th><th>Source</th><th>Opened</th><th>Closed</th></tr>
<tr><td>C-4</td><td>Calling vfuncs in ctors/dtors</td><td>call</td><td>open</td><td>Cygnus</td><td>990603</td><td></td></tr>
<tr><td colspan="7"><b>Summary</b>: When calling a virtual function from the constructor/destructor of a base subobject, the version specific to the base type is required, unlike the typical case when calling such a vfunc for the full object from some other context. Since the pointer for that vfunc in the full object's vtable (or even the subobject's sub-vtable) is the full object version, some other means is required for accessing the correct vfunc.</td></tr>
</table>

<table>
<tr><th>#</th><th>Issue</th><th>Class</th><th>Status</th><th>Source</th><th>Opened</th><th>Closed</th></tr>
<tr><td>C-5</td><td>Calling destructors</td><td>call</td><td>open</td><td>Sun</td><td>990603</td><td></td></tr>
<tr><td colspan="7"><b>Summary</b>: What is the calling convention for destructors? Do virtual destructors require special treatment? Is delete() integrated with the destructor call or separate? How is delete() handled when invoked on a base subobject?</td></tr>
</table>

[990729 all] Some implementations combine destructors with deletion, checking a flag in the destructor to determine whether to delete. This produces somewhat less code, especially if there are many delete() calls. However, it adds overhead to any destructor which does not require deletion, e.g. base and member objects, automatic objects. There is some concern that a runtime test is sometimes required, but noone has yet identified why.

[990819 Cygnus -- Jason] The [above] questions the usefulness of calling op delete from the destructor. But it's required by the language, in case the derived class defines its own op delete. This only applies to virtual dtors, of course.

One option would be to have two dtor slots, one which performs deletion and one which doesn't. The advantage of this sort of approach would be avoiding pulling in all the memory management code if you never actually touch the heap.

Microsoft has a patent on this device, but the old Sun ABI also talks about it, which seems to qualify as prior art.

#	Issue	Class	Status	Source	Opened	Closed
C-6	Extra parameters to ctors/dtors	call	open	Cygnus	990603	

Summary: When calling constructors and destructors for classes with virtual bases, information about the virtual base subobjects in the full class must be transmitted somehow to the ctor/dtor.

#	Issue	Class	Status	Source	Opened	Closed
C-7	Passing value parameters by reference	call	closed	All	990624	990805

Summary: It may be desirable in some cases where a type has a non-trivial copy constructor to pass value parameters of that type by performing the copy at the call site and passing a reference.

Resolution : Whenever a class type has a non-trivial copy constructor, pass value parameters of that type by performing the copy at the call site and passing a reference.

#	Issue	Class	Status	Source	Opened	Closed
C-8	Returning classes with non-trival copy constructors	call	closed	All	990625	990722

Summary: How do we return classes with non-trivial copy constructors?

Resolution: The caller allocates space, and passes a pointer as an implicit first parameter (prior to the implicit *this* parameter).

Exception Handling Issues

For reference, we have design information as follows:

- [990818 Intel -- Priti] (PowerPoint document)
- [990818 HP -- Christophe] (PDF document)

[990902 All] We observed that there are three levels at which we can discuss EH compatibility.

The first, minimal level is effectively that of the definition in the IA-64 Software Conventions document. It describes a framework which can be used by an arbitrary implementation, with a complete definition of the stack unwind mechanism, but no significant constraints on the language-specific processing. In particular, it is not sufficient to guarantee that two object files compiled by different C++ compilers could interoperate, e.g. throwing an exception in one of them and catching it in the other.

The second level is the minimum that must be specified to allow interoperability in the sense described above. This level requires agreement on:

- Standard runtime initialization, e.g. pre-allocation of space for out-of-memory exceptions.
- The layout of the exception object created by a throw and processed by a catch clause.
- When and how the exception object is allocated and destroyed.
- The API of the personality routine, i.e. the parameters passed to it, the logical actions it performs, and any results it returns (either function results to indicate success, failure, or continue, or changes in global or exception object state), for both the phase 1 handler search and the phase 2 cleanup/unwind.

- How control is ultimately transferred back to the user program at a catch clause or other resumption point. That is, will the last personality routine transfer control directly to the user code resumption point, or will it return information to the runtime allowing the latter to do so?

- Standard runtime initialization, e.g. pre-allocation of space for out-of-memory exceptions.

- Multithreading behavior.

The third level is a specification sufficient to allow all compliant systems to share the relevant runtime implementation. It includes, in addition to the above:

- Format of the C++ language-specific unwind tables.

- APIs of the functions named __allocate_exception, __throw, and __free_exception (and likely others) by HP, or their equivalents.

- API of landing pad code, and of any other entries back into the user code.

- Definition of what HP calls the exception class value.

The vocal attendees at the meeting wish to achieve the third level, and we will attempt to do so. Whether or not that is achieved, however, a second-level specification must be part of the ABI.

[990909 All/Jim] With much further discussion, we are starting to get better understanding of one another, but there are still obviously (in my mind) mismatched underlying assumptions. To resolve this, Christophe agreed to attempt to get us the HP APIs for the exception handling routines. I have also started a document on a more complete EH specification, though it hasn't gone beyond specifying more of the underlying base ABI part. I will go farther once I get back from my trip.

#	Issue	Class	Status	Source	Opened	Closed
D-1	Language-specific data area format	lib ps	open	SGI	990520	
Summary: The IA-64 runtime conventions describe language-independent descriptors for restoring registers when unwinding the stack. The do not specify how C++ performs language-specific unwinding for exception handling, i.e. locating a handler and destroying automatic objects. Note that this can be handled by agreeing on common descriptors, or by agreeing on per-frame personality routines with common APIs.						

[990715 Cygnus -- Jason] The languge-specific part of the EH stack in g++ contains these elements:

```
void *value; // pointer to the thrown object, or the thrown value
             // itself if a pointer
void *type;  // pointer to the type_info node for the thrown object
void (*cleanup)(void *, int) // pointer to the dtor for the object
bool caught; // has this exception been caught since its last throw?
long handlers; // how many catch handlers are active for this exception
```

Both 'caught' and 'handlers' are needed to handle rethrowing and catching within a catch block.

Language interaction is handled by recording the language of both the exception region and the thrown exception. Each thrown exception also includes a pointer to a language-specific matching function which is called to compare the types of the exception and handler.

#	Issue	Class	Status	Source	Opened	Closed
D-2	Unwind personality routines	lib ps	open	SGI	990520	
Summary: The IA-64 runtime conventions provide for a personality routine pointer for language-specific actions when unwinding the stack. They do not specify its interface. There are typically two required actions for C++: locating a handler (non-destructively) and destroying automatic objects while unwinding. This issue involves specification of the API (see also D-3).						

[990826 Intel/HP] The Software Conventions document is claimed to specify the interface, with the parameters indicating which action is required. (I can't find it, but this would be an acceptable solution -- Jim.)

#	Issue	Class	Status	Source	Opened	Closed
D-3	Unwind process clarification	lib ps	open	SGI	990520	
Summary: The IA-64 runtime conventions provide for a personality routine pointer for language-specific actions when unwinding the stack. However, they are quite muddy about the precise sequence of calls. This issue involves specification of unwind process (see also D-2).						

#	Issue	Class	Status	Source	Opened	Closed
D-4	Unwind routines nested?	lib ps	open	SGI	990520	
Summary: The IA-64 runtime conventions call for the unwind personality routine to behave like a routine nested in the routine raising an exception. Is that the preferred definition?						

[990902 All] Discussion reveals that Intel and HP have very different models of how cleanup actions are handled.

Intel builds one or more routines which are called from the unwind runtime, based on action descriptors in the unwind tables, and acting on the stack contents or objects to be destroyed without actually modifying the stack pointer until the final transfer of control to the user handler. This approach avoids actually restoring registers until the final transfer to the handler.

HP transfers control back to a user landing pad whenever anything needs to be done -- descriptors or handlers -- and reenters the unwind runtime if further processing is required. They believe this approach to use much less space than the action descriptors would, and most importantly, that it allows arbitrary fixup for code motion around the call that throws.

#	Issue	Class	Status	Source	Opened	Closed
D-5	Interaction with other languages (e.g. Java)	lib ps	open	HP	990603	
Summary: The IA64 exceptions handling framework is largely language independent. What is the behaviour of a C++ runtime receiving, for instance, an exception thrown from Java? Does it call terminate()? Does it allow the exception to pass through C++ code with destructors if there is no catch clause? Does it allow the exception to be caught in a catch(...) provided this catch(...) ends with a rethrow? Does it allow even more?						

[990908 SGI -- Jim] We propose that this be resolved by identifying the source language in the exception descriptor and specifying that the personality routine be able to perform cleanup actions during handling of foreign-language exceptions, but not attempt to catch them.

#	Issue	Class	Status	Source	Opened	Closed
D-6	Allow resumption in other languages?	lib ps	open	HP	990603	
Summary: The exception handling framework requires the interaction of the runtime of all the languages "on the stack" during exception processing. Some of these languages may have very different exception handling semantics. What are the constraints we impose on the C++ exception handling runtime to preserve the relative language neutrality of the EH framework? Example: do we allow a handler to cleanup and resume at the point where the exception was thrown?						

[990908 SGI -- Jim] The typical case of cleanup and resume is floating point trap handling, which is normally handled entirely in the original FP trap handler. Is there an example where stack walkback must occur to identify the handler, but resumption at the point-of-exception is required? I can't think of any, and I think the model of registering a trap handler is preferable for such purposes.

#	Issue	Class	Status	Source	Opened	Closed
D-7	Interaction with signals or asynch events	lib ps	open	HP	990603	
Summary: The Standard says that the behavior of anything other than "pure C code" (POF) is implementation defined, and warns (in a note) against using EH in a signal handler. We should define what is supported, possibly explicitly stating that signal handler code must be a POF. We could allow any feature but exception handling to be used. We could allow some EH routines to be called (for instance, **`uncaught_exception()`**). Or we could allow even an exception to be thrown, if it does not exit the handler.						

#	Issue	Class	Status	Source	Opened	Closed
D-8	Interaction with threads packages	lib ps	open	SGI	990603	
Summary: What happens when an exception is not caught in the thread where raised? What does uncaught_exception() return if another thread is currently processing an exception?						

#	Issue	Class	Status	Source	Opened	Closed
D-9	longjmp interaction	lib ps	open	IBM	990908	
Summary: Does longjmp run destructors?						

[990908 IBM -- Mendell] Does longjmp run destructors? I believe that the C ABI makes this optional. I would like to propose that it does run destructors.

[990908 SGI -- Wilkinson] The C++ standard, 18.7 paragraph 4, says a call to longjmp has undefined behavior if any automatic objects would have been destroyed by a throw/catch with the same source and destination. I don't see that this is something we need to fix.

[990908 IBM -- Thomson] Yes it does, but ANSI is not my customer. Meeting the bare minimum of function that ANSI requires doesn't necessarily mean that users can build robust applications. How can they know to avoid longjmp in their C code, because some third party library they are using has C++ buried in it?

[990908 SGI -- Dehnert] Implementation is a significant issue. The normal longjmp implementation is very simple -- setjmp stores the register/stack state, and longjmp copies it back and branches. There is normally no traceback involved, so what you suggest is a dramatic change, and probably would make C people very unhappy. Furthermore, C++ users have the option of using C++ exceptions, which have the effect you seek.

[990908 SGI -- Boehm] The problem is that on the C side:

1. A number of thread packages use setjmp/longjmp to perform context switches. In this case, the target sp is not on the same stack as the original sp, and there should not be any destructor invocations, since the original thread will be resumed, and the original sp will eventually be restored. (This isn't the optimal way to do thread switching, but it's the only one that's semi-portable, and hence it's moderately common.)

2. Some variants of longjmp are often used to jump out of signal handlers, which may not be invoked on the original user stack (cf. sigaltstack on most Unix systems). Thus unwinding may have to cross stack boundaries.

3. Setjmp is often used to capture the register state, e.g. for garbage collectors. (The collector I'm responsible for optionally does this. Last I looked, Guile did it unconditionally.) A straightforward stack-unwinding implementation of setjmp/longjmp would break this.

I don't know whether it's possible to avoid breaking these clients while providing the stack-unwinding semantics.

[990908 IBM -- Mendell/Thomson] [VisualAge C++] on OS/2 and Windows does do the unwinding. This is probably because unwinding support is in the OS. Also OS/390 and I believe AS/400 too. Our AIX implementation does not do the unwinding.

[990909 DEC -- Brender] In addition to the systems already mentioned by others, these systems also do exception-handling compatible unwinding for C's setjmp/longjmp:

- VMS/VAX and VMS/Alpha: Tru64 Unix/Alpha [not originally, but at least as of V4]
- Microsoft Visual C on W95&WNT/IA32: [to support SEH (structured exception handling) extensions] (probably also on IA64 for compatibility reasons)
- Microsoft Visual C on WNT/Alpha (RIP): [to support SEH]

If you believe in safe and compatible multi-language systems, there really is no choice but to do EH compatible unwinding for setjmp/longjmp -- at least by default.

I suppose it would be OK for an implementation to offer an alternate setjmp/longjmp that could be linked in for those who either know that it is safe in particular cases or are happy to trade safety for speed...

[990909 All] A brief discussion agreed that concensus is not absolutely necessary. An implementation could replace setjmp/longjmp with a version that either unwinds or justs restores and jumps, without breaking any code except that which assumed one or the other. (Ed.: In fact, if setjmp stores enough information to either restore or to catch an exception, one could just swap longjmp, although that would not be optimal for the unwind and catch case, since setjmp doesn't need to save much information in that case as most of what is needed is in the unwind descriptors.)

Template Instantiation Model Issues

#	Issue	Class	Status	Source	Opened	Closed
E	**Template Instantiation Model**					
E-1	When does instantiation occur?	tools	open	SGI	990520	
Summary: There are two principal models for instantiation. The *early instantiation* (or Borland) model performs all instantiation at compile time, potentially resulting in extra copies which are removed at link time. The *pre-link instantiation* model identifies the required instantiations prior to linking and instantiates them via a special compile step.						

#	Issue	Class	Status	Source	Opened	Closed
E-2	Separate compilation model	tools	open	SGI	990520	
Summary: [SGI]						

#	Issue	Class	Status	Source	Opened	Closed
E-3	Template repository	tools	open	HP	990603	
Summary: Independent of the template instantiation model, we need to make sure that whatever template persistent storage is used by one vendor does not interact negatively with other vendors' mechanisms. Issues: (1) Avoiding conflict on the name of any repository. (2) If .o files are used, describe how this information is to be preserved, ignored, etc. (3) Evaluate if tools such as make, ld, ar, or others, can break because .o files get written at unexpected times.						

Name Mangling Issues

#	Issue	Class	Status	Source	Opened	Closed
F-1	Mangling convention	call	open	SGI	990520	
Summary: What rules shall be used for mangling names, i.e. for encoding the information other than the source-level object name necessary to resolve overloading?						

[990806 SGI -- Jim] Naming to be resolved under this issue includes:

- Global and member operator names
- Global and member function names
- Vtable names (primary and initialization)
- RTTI struct names
- Template instance names
- Namespace effects

#	Issue	Class	Status	Source	Opened	Closed
F-2	Mangled name size	call g	open	SGI	990520	
Summary: Typical name mangling schemes to date typically begin to produce very long names. SGI routinely encounters multi-kilobyte names, and increasing usage of namespaces and templates will make them worse. This has a negative impact on object file size, and on linker speed. SGI has considered solutions to this problem including modified string tables and/or symbol tables to eliminate redundancy. Cygnus, HP, and Sun have also considered or implemented approaches which at least mitigate it.						

#	Issue	Class	Status	Source	Opened	Closed
F-3	Distinguish template instantiation and specialization	call g	open	SGI	990520	
Summary: In order to allow detection of conflicting template instantiation and specialization (in different translation units), should we name them differently? If we do so in an easily recognizable way, the linker could check for conflicts and report the ODR violation.						

Miscellaneous Issues

#	Issue	Class	Status	Source	Opened	Closed
G-1	Basic command line options	tools	open	HP	990603	
Summary: Can we agree on basic command line options (compiler and linker) for fundamental functionality, possibly allowing portable makefiles?						

#	Issue	Class	Status	Source	Opened	Closed
G-2	Detection of ODR violations	call	open	Sun	990603	
Summary: [Sun] (See also F-3.)						

<table>
<tr><th>#</th><th>Issue</th><th>Class</th><th>Status</th><th>Source</th><th>Opened</th><th>Closed</th></tr>
<tr><td>G-3</td><td>Inlined routine linkage</td><td>call</td><td>open</td><td>Sun</td><td>990603</td><td></td></tr>
<tr><td colspan="7"><b>Summary</b>: Inline routines with external linkage require a method of handling vague linkage (see B-5 for definition) for the out-of-line instance, as well as for any static data they contain. The latter includes string constants per [7.1.2]/4.</td></tr>
</table>

[990624 Cygnus -- Jason] How should we handle local static variables in inlines? G++ currently avoids this issue by suppressing inlining of functions with local statics. If we don't want to do that, we'll need to specify a mangling for the statics, and handle multiple copies like we do above.

<table>
<tr><th>#</th><th>Issue</th><th>Class</th><th>Status</th><th>Source</th><th>Opened</th><th>Closed</th></tr>
<tr><td>G-4</td><td>Dynamic init of local static objects and multithreading</td><td>call</td><td>open</td><td>SCO</td><td>990607</td><td></td></tr>
<tr><td colspan="7"><b>Summary</b>: The Standard requires that local static objects with dynamic constructors be initialized exactly once, the first time the containing scope is entered. Multi-threading renders the simple check of a flag before initialization inadequate to prevent multiple initialization. Should the ABI require locking for this purpose, and if so, what are the necessary interfaces? In addition to the locking of the initialization, special exception handling treatment is required to deal with an exception during construction.</td></tr>
</table>

[990607 SCO -- Jonathan] The standard is mute on multiple threads of control in general, so there is no requirement in the language to support what I'm talking about. But as a practical matter compilers have to do it (Watcom gave a paper on their approach during the standardization process, if I remember). This example using UI/SVR4 threads will usually show whether a compiler does it or not:

```
thr5.C:
// static local initialization and threads

#include
#define EXIT(a) exit(a)
#define THR_EXIT() thr_exit(0)

#include

int init_count = 0;
int start_count = 0;

int init()
{

        ::thr_yield();
        return ++init_count;
}

void* start(void* s)
{

        start_count++;
        static int i = init();
        if (i != 1) EXIT(5);
        THR_EXIT();
        return 0;
}

int main()
{
        thread_t t1, t2;
        if (::thr_create(0, 0, start, 0, 0L, &t1) != 0) EXIT(1);
        if (::thr_create(0, 0, start, 0, 0L, &t2) != 0) EXIT(2);
        if (::thr_join(t1, 0, 0) != 0) EXIT(3);
        if (::thr_join(t2, 0, 0) != 0) EXIT(4);
        if (start_count != 2)
                EXIT(6);
        if (init_count != 1)
                EXIT(7);
        THR_EXIT();
}
```

When compiled with CC -Kthread thr5.C on UnixWare 7, for instance, it passes by returning 0. When compiled with CC -mt thr5.C on Solaris/x86 C++ 4.2 (sorry don't have the latest version!), it fails by returning 5.

[990607 Sun -- Mike Ball] As far as I can tell, the language says that the automatic blocking issue isn't a valid approach. It says what has to happen, and it isn't that.

If you look at the entire statement you find that it reads: "Otherwise such an object is initialized the first time control passes through its declaration; such an object is considered initialized upon the completion of its initialization. If the initialization exits by throwing an exception, the initialization is not complete, so it will be tried again the next time control enters the declaration. If control re-enters the declaration (recursively) while the object is being initialized, the behavior is undefined."

The word "recursively" is normative, so eliminates that sentence from consideration.

One can, of course, make any extension to the language, but in this case I think the extension invalidates some otherwise valid code.

The sentence I'm referring to is that the object is considered initialized upon the completion of its initialization. This is explicit, and the reason for it is covered in the following sentence, which discusses an initialization that terminates with an exception. A person catching such an exception has the right to try again without danger that the static variable will be initialized in the meantime.

I don't see anything at all to justify semantics that say, "after initialization is started, Any other threads of control are blocked until that thread completes the initialization, unless, of course, it executes by an exception, in which case the other thread can do the initialization before the exception handler gets a chance to try again, except...." Take an attempt to define the semantics as far as you like.

The problem is that there is no way for the compiler writer to know what the programmer really wanted to do. I can (and will at some other date, if necessary) come up with scenarios justifying a variety of mutual exclusion policies, including none.

The solution is to let the programmer write the mutual exclusion, the same as we do for every other potential race condition. It's a real mess, and, I claim, an unwise one to put in as an extension.

[990608 HP -- Christophe] The semantics currently implemented in the HP aC++ compiler is as follows:

- No two thread can enter a static initialization at the same time
- Threads are blocked until immediately after the static initialization either succeeds or fails with an exception.

There are details of our implementation that I disagree with, but in general, the semantics seem clear and sane, not as convoluted as you seemed to imply. In particular, it correctly covers the case where the static initialization fails with an exception. Any thread at that point can attempt the initialization.

[990608 SCO -- Jonathan] Here's what the SCO UnixWare 7 C++ compiler does for IA-32, from a (slightly sanitized) design document. It meets Jim's goal of having no overhead for non-threaded programs and minimal overhead for threaded programs unless actual contention occurs (infrequent), and meets Mike's goal of handling exceptions in the initialization correctly (although it doesn't guarantee that the thread getting the exception is the one that gets next crack at initializing the static). It's also worth noting that dynamic initialization of local variables (static or otherwise) is very common in C++, since that's what most object constructions involve, so I don't think this case is as rare as Jim does.

[...] This is in local static variables with dynamic initialization, where the compiler generates out a static one-time flag to guard the initialization. Two threads could read the flag as zero before either of them set it, resulting in multiple initializations.

[...] Accordingly, when compilation is done with -Kthread on, a code sequence will be generated to lock this initialization. [...] the basic idea is to have one guard saying whether the initialization is done (so that multiple initializations do not occur) and have another guard saying whether initialization is in progress (so that a second thread doesn't access what it thinks is an initialized value before the first thread has finished the initialization). [...]

When compiled with -Kthread, the generated code for a dynamic initialization of a local static variable will look like the following. guard is a local static boolean, initialized to zero, generated by the [middle pass of the compiler]. Two bits of it are used: the low-order 'done bit' and the next-low-order 'busy bit'.

```
.again:
        movl    $guard,%eax
        testl   $1,(%eax)       // test the done bit
        jnz     .done           // if set, variable is initialized,
done
        lock; btsl  $1,(%eax)   // test and set the busy bit
        jc      .busy
        < init code >           // not busy, do the initialization
        movl    $guard,%eax
        movl    $3,(%eax)       // set the done bit
        jmp     .done
.busy:
        pushl   %eax            // call RTS routine to wait, passing address
        call1   __static_init_wait      // of guard to monitor
        testl   %eax,%eax       // 1 means exception occurred in init code,
        popl    %ecx
        jnz     .again                  // start the whole thing over
.done                                   // 0 means wait finished
```

The above code will work for position-independent code as well. The complication due to exceptions is: what happens if the initialization code throws an exception? The [compiler] EH tables will have set up a special region and flag in their region table to detect this situation, along with a pointer to the guard variable. Because the initialization never completed, when the RTS sees that it is cleaning up from such a region, it will reset the guard variable back to both zeroes. This will free up a busy-waiting thread, if any, or will reset everything for the next thread that calls the function.

The idea of the __static_init_wait() RTS routine is to monitor the value of guard bits passed in, by looping on this decision table:

```
    done    busy
    0       0       return 1 in %eax        (EH wipe-out)
    1       1       return 0 in %eax        (no longer busy)
    0       1       continue to wait        (still busy)
    1       0       internal error, shouldn't happen
```

As for how the wait is done [... not relevant for ABI, although currently we're using thr_yield(), which may or may not be right for this context].

[990608 SGI -- Hans] I'd like to make some claims about function scope static constructor calls in multithreaded environments. I personally can't recall ever having used such a construct, which somewhat substantiates my claims, but also implies some lack of certainty. I'd be interested in hearing any arguments to the contrary.

I believe that these arguments imply that this problem is not important enough to warrant added ABI complexity or overhead for sequential code.

Consider the following skeletal example:

```
f(int x) { static foo a(...); ... }
```

1. If the constructor argument doesn't depend on the function parameter, and the code behaves reasonably, it should be possible to rewrite this as

   ```
   static foo a(...);
   f(int x) { ... }
   ```

2. If I read the standard correctly (and that's a big disclaimer), the compiler is entitled to perform the above transformation under conditions that are usually true, but hard for the compiler to deduce. Thus code that relies on the initialization occurring during the execution of f is usually broken.

3. Thus the foo constructor cannot rely on its caller holding any locks. It must explicitly acquire any locks it needs.

4. It is far preferable to write the transformed form with a file scope static variable to start with. The initial form risks deadlock, since f may be called with locks held which the constructor can't assume are held. If it needs one of those locks it will need to reacquire it. With default mutex semantics that results in deadlock with itself. (If locks may be reentered, it may fail in a more subtle manner since the foo constructor may acquire a monitor lock whose monitor invariant doesn't hold.)

5. File scope static constructor calls aren't a problem and require no locking, since they are executed in a single thread before main is called or before dlopen returns. (Forking a thread in a static constructor should probably be disallowed. Threads may not have been fully initialized, among other issues.)

6. Static function scope constructor calls which depend on function arguments are likely to involve a race condition anyway, if multiple instances of the function can be invoked concurrently. Any of the calls might determine the constructor parameters. Thus these aren't very interesting either. And if they are really needed, they can be replaced with a file scope static constructor call plus an assignment.

#	Issue	Class	Status	Source	Opened	Closed
G-5	Varargs routine interface	call	open	HU-B	990810	
Summary: The underlying C ABI defines conventions for calling varargs routines. Does C++ need, or would it benefit from, any modifications or special cases? How should we pass references or class objects? Is any runtime library support required?						

[990810 HU-B Martin] I'd like to see an indirection in vararg lists, so they can be passed through thunks. This is necessary at least for the covariant returns, but might have other applications as well.

[990810 HU-B Martin] Since there already was the decision not to return a list of pointers from a covariant method, the only alternative to real thunks is code duplication (as done in Sun Workshop 5). *(Or alternate entrypoints... Jim)*

With real thunks, you have to copy the argument list. That is not possible for a varargs list, so here is my proposal for varargs in C++:

In the place of the ellipsis, a pointer to the first argument is passed. In case of a thunk for covariant returns, this pointer can be copied to the destination function. The variable arguments are put on the stack as they normally would.

With that, the issue is in which cases to use such a calling convention:

1. only for vararg calls to virtual methods, or
2. only for vararg calls to functions with C++ linkage, or
3. for all vararg calls. That would probably require a change to the C ABI

Option (1) could be further restricted to methods returning a pointer or reference to class type.

[990812 All] In response to a question, it was observed that passing one variant of a class hierarchy in a varargs list and referencing another variant in the va_arg macro is undefined, and we don't need to worry about a mechanism for doing the conversion.

Library Interface Issues

#	Issue	Class	Status	Source	Opened	Closed
H-1	Runtime library DSO name	tools	open	SGI	990616	
Summary: Determine the name of the common C++ runtime library DSO, e.g. **`libC.so`**. If there are to be vendor-specific support libraries which must coexist in programs from mixed sources, identify naming convention for them.						

#	Issue	Class	Status	Source	Opened	Closed
H-2	Runtime library API	lif	open	SGI	990616	
Summary: Define the required entrypoints in the common C++ runtime library DSO, and their prototypes.						

Please send corrections to Jim Dehnert.