C++ ABI Summary

Revised 6 October 1999

Links

- Full open issues list.
- Full closed issues list.
- Draft data layout specification.
- Draft exception handling specification.
- Vtable layout examples.
- 64 Bit Runtime Architecture and Software Conventions for IA-64, Intel document SC - 2791, Rev. No. 2.4E.
- IA-64 Instruction Set Architecture

Meetings

When		Where	Phone
30 September	10:00-12:00 PDT	completed	
7 October	10:00-12:00 PDT	SGI Sapphire 20L	650-933-7952
14 October	10:00-12:00 PDT	SGI Sapphire 20L	650-933-7952
21 October	10:00-12:00 PDT	SGI ???	650-933-7952
28 October	10:00-12:00 PDT	SGI ???	650-933-7952

Note: When calling the SGI telephone bridges, the first caller continues to ring until the second party joins. To get rid of it, you can call from a second phone, and hang it up once someone else calls.

Participants

Company	Name	Telephone	Fax	Email
	overall reflector			cxx-abi@corp.sgi.com
	Jim Dehnert	(650) 933-4272	(650) 932-4272	dehnert@sgi.com

SGI	Matt Austern	(650) 933-4196	(650) 932-4196	austern@engr.sgi.com
	Shin-Ming Liu	(650) 933-4287	(650) 932-4287	shin@engr.sgi.com
	John Wilkinson	(650) 933-4298	(650) 932-4298	jfw@engr.sgi.com
	reflector			cxx-abi-sgi@engr.sgi.com
CodeSourcery	Mark Mitchell	(650) 365-3064	(650) 375-7694	mark@codesourcery.com
Compaq	Ron Brender	(603) 884-2088	(603) 884-0153	brender@zko.dec.com
	Coleen Phillimore	(603) 884-0939	(603) 884-0153	coleen@philli.zko.dec.com
	Dave Plummer	(603) 884-3936	(603) 884-0153	Dave.Plummer@Compaq.com
Cygnus	Jason Merrill	(408) 542-9665	(408) 542-9765	jason@cygnus.com
	Benjamin Kosnik	(408) 542-9643	(416) 542-9765	bkoz@cygnus.com
	Ian Carmichael	(416) 482-3946	(416) 482-6299	iancarm@cygnus.com
	Ulrich Drepper	(408) 765-4699	?	drepper@cygnus.com
	reflector			c++abi@cygnus.com
EDG	Daveed Vandevoorde	(650) 592-5768	(650) 592-5781	daveed@edg.com
EPC	Colin McPhail	+44 (131) 225-6262	+44 (131) 225-6644	colin@epc.co.uk
Hewlett- Packard	Cary Coutant	(408) 447-5759	(408) 447-4244	cary@cup.hp.com
	Christophe de Dinechin	(408) 447-5491	(408) 447-4244	ddd@cup.hp.com
	Sassan Hazeghi	(408) 447-5007	(408) 447-4244	sassan@cup.hp.com
	reflector			cxx-abi-hp@adlmail.cup.hp.com

Humboldt-Universität zu Berlin	Martin von Löwis	+49 30 2093 3118	+49 30 2093 3112	loewis@informatik.hu-berlin.de
IBM	Mark Mendell	(416) 448-3485	(416) 448-4414	mendell@ca.ibm.com
	Allan H. Kielstra	(416) 448-3558	(416) 448-4414	kielstra@ca.ibm.com
	reflector			CxxABI-ADTC-CAN@ca.ibm.com
Intel	Sunil Saxena	(408) 765-5272	(408) 653-8511	Sunil.Saxena@Intel.com
	Suresh Rao	(408) 765-5416	(408) 765-5165	Suresh.K.Rao@Intel.com
	Priti Shrivastav	(408) 765-4699	(408) 765-5165	Priti.Shrivastav@Intel.com
	reflector			cxx-abi@unix-os.sc.intel.com
SCO	Jonathan Schilling	(908) 790-2364	(908) 790-2426	jls@sco.com
Sun	George Vasick	(650) 786-5123	(650) 786-9551	george.vasick@eng.sun.com
	Michael Lam	(650) 786-3492	(650) 786-9551	michael.lam@eng.sun.com
	Michael Ball	(650) 786-9109	(650) 786-9551	michael.ball@eng.sun.com
	Reza Monajjemi	(650) 786-6175	?	reza.monajjemi@eng.sun.com

Objectives

- Interoperable C++ compilation on IA-64: we want users to be able to build relocatable objects with different compilers and link them together, and if possible even to ship common DSOs. This objective implies agreement on:
  - Data representation
  - Object file representation
  - Library API
- ISO Standard C++: highest priority is functionality and performance of standard-compliant code. It should not be sacrificed for the benefit of language extensions or legacy implementations (though

considering them as tie-breakers is fine).

- Some areas will be easier to agree on than others. Our priorities should be based on achieving as much interoperability as possible if we can't attain perfection. That is, it is better to end up with a few restrictions being required for interoperable code, than to have no interoperability at all. This suggests priorities as follows:
    1. Items requiring base ABI changes that might affect other languages, and will therefore become impossible soon. Examples include exception handling / stack unwind, or ELF changes (not extensions).
    2. Core features where differences will prevent virtually any C++ object code from porting. Examples include data layout and calling conventions.
    3. Limited usage features, where users can achieve portability by avoiding the feature. An example might be multi-threading.
    4. Peripheral features, where the requirements on users to achieve portability are clear and easy to implement. An example is non-explicit inlining, where compilers would presumably allow it to just be suppressed.
    5. Tool interfaces, which affect how users build code, rather than what they build. An example is the compilation command line.
- Mechanisms/methods which allow coexistence of incompatible implementations may be suitable in some cases. For instance, packaging vendor-specific compiler support runtimes in DSOs occupying distinct namespaces might allow multiple such DSOs to be loaded for mixed objects and avoid requiring that all vendors have the same support runtimes.

Action Item Status

#	Action	Who	Status	Opened	Closed
1	Distribute Sun C++ ABI	Mike Ball	closed	990603	990930
2	Distribute Sun C++ ABI Rationale	Mike Ball	closed	990603	990930
3	Distribute Taligent C++ ABI	Cary Coutant	open	990603	
4	Expedite IA-64 RT Arch doc release	Cary Coutant	closed	990603	990720
5	Set up n-way NDA for eligible members	Priti Shrivastav	open	990603	
6	Organize/summarize object layout issues and alternatives	Matt Austern	closed	990603	990624
7	Write-up of Vfunc call protocol	Christophe de Dinechin	closed	990610	990805
8	Write-up of object layout strawman	Matt Austern	closed	990610	990624
9	Check with c++-core about empty base placement	Jason Merrill	closed	990610	990618

10	Describe dynamic cast / inaccessible base issue	Daveed Vandevoorde	closed	990617	990701
11	Summarize ctor/dtor issues	Michael Lam	closed	990617	990729
12	Describe Intel exception model	Priti Shrivastav	closed	990624	990818
13	Propose RTTI representation	Daveed Vandevoorde	closed	990701	990819
14	Open EDG exception stack?	Daveed Vandevoorde	moot	990715	990930
15	Priority scheme descriptor for C-2	Jim Dehnert	closed	990715	990804
16	Covariant return scheme	Jason Merrill	closed	990715	990729
17	Validate Christophe's B-6 Vtable layout	Jim Dehnert	closed	990729	990811
18	Check Sun dynamic cast algorithm	Michael Lam	closed	990805	990812
19	Write up C-3 destructor proposal	Jim Dehnert	closed	990805	990811
20	Look at implications of discussion on B-1, B-6	Christophe de Dinechin	closed	990812	990830
21	Look at alternative implementations of C-2	Jim Dehnert	open	990812	
22	Describe HP hash processing for RTTI	Christophe de Dinechin	moot	990826	990930
23	Complete RTTI proposal	Daveed Vandevoorde	closed	990826	991006
24	Summarize Vtable issues	Jason Merrill	closed	990909	990930
25	Update traceback personality API	Christophe de Dinechin	open	990923	

Issue Status

In the following table, the ***class*** column attempts to classify the issue 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

#	Issue	Class	Status	Source	Opened	Closed
A	**Object Layout**					
A-1	Vptr location	data	closed	SGI	990520	990624
A-2	Virtual base classes	data	closed	SGI	990520	990624
A-3	Multiple inheritance	data	closed	SGI	990520	990701
A-4	Empty base classes	data	closed	SGI	990520	990624
A-5	Empty parameters	data	closed	SGI	990520	990701
A-6	RTTI (`type_info`) .o representation	data call ps	open	SGI	990520	
A-7	Vptr sharing with primary base class	data	closed	HP	990603	990729
A-8	(Virtual) base class alignment	data	closed	HP	990603	990624
A-9	Sorting fields as allowed by [class.mem]/12	data	closed	HP	990603	990624
A-10	Class parameters in registers	call	closed	HP	990603	990701
A-11	Pointers to member functions	data	closed	Cygnus	990603	990812
A-12	Merging secondary vtables	data	closed	Sun	990610	990805
A-13	Parameter struct field promotion	call	closed	SGI	990603	990701
A-14	Pointers to data members	data	closed	SGI	990729	990805
B	**Virtual Function Handling**					
B-1	Adjustment of "this" pointer (e.g. thunks)	data call	open	SGI	990520	
B-2	Covariant return types	call	closed	SGI	990520	990722
B-3	Allowed caching of vtable contents	call	closed	HP	990603	990805
B-4	Function descriptors in vtable	data	closed	HP	990603	990805
B-5	Where are vtables emitted?	data	open	HP	990603	
B-6	Virtual function table layout	data	open	SGI	990520	
B-7	Objects and Vtables in shared memory	data	closed	HP	990624	990805
B-8	dynamic_cast	data	open	SGI	990628	
C	**Object Construction/Destruction**					
C-1	Interaction with .init/.fini	lif ps	open	SGI	990520	
C-2	Order of ctors/dtors w.r.t. link	lif ps	open	HP	990603	

C-3	Order of ctors/dtors w.r.t. DSOs	ps	open	HP	990603	
C-4	Calling vfuncs in ctors/dtors	call	open	Cygnus	990603	
C-5	Calling virtual destructors	call	open	Sun	990603	
C-6	Extra parameters to ctors/dtors	call	open	Cygnus	990603	
C-7	Passing value parameters by reference	call	closed	All	990625	990805
C-8	Returning classes with non-trival copy constructors	call	closed	All	990625	990722

D	**Exception Handling**					
D-1	Language-specific data area format	lib ps	open	SGI	990520	
D-2	Unwind personality routines	lib ps	open	SGI	990520	
D-3	Unwind process clarification	lib ps	open	SGI	990520	
D-4	Unwind routines nested?	lib ps	open	SGI	990520	
D-5	Interaction with other languages (e.g. Java)	lib ps	open	HP	990603	
D-6	Allow resumption in other languages?	lib ps	open	HP	990603	
D-7	Interaction with signals or asynch events	lib ps	open	HP	990603	
D-8	Interaction with threads packages	lib ps	open	SGI	990603	
D-9	longjmp interaction	lib ps	open	HP	990908	

E	**Template Instantiation Model**				
E-1	When does instantiation occur?	tools	open	SGI	990520
E-2	Separate compilation model	tools	open	SGI	990520
E-3	Template repository	tools	open	HP	990603

F	**Name Mangling**				
F-1	Mangling convention	call	open	SGI	990520
F-2	Mangled name size	call g	open	SGI	990520
F-3	Distinguish template instantiation and specialization	call g	open	SGI	990520
F-4	Empty throw specs	call g	open	HP	990930

G	**Miscellaneous**

G-1	Basic command line options	tools	open	HP	990603
G-2	Detection of 1-def rule violations	call	open	Sun	990603
G-3	Inlined routine linkage	call	open	Sun	990603
G-4	Dynamic init of local static objects and multithreading	call	open	SCO	990607
G-5	Varargs routine interface	call	open	HU-B	990810
H	**Runtime Library Interface**				
H-1	Runtime library DSO name	tools	open	SGI	990616
H-2	Runtime library API	lif	open	SGI	990616

Procedure Notes from 3 June 1999

- Meetings: 10-12 Thursdays at SGI for the near term.
- Intel NDA: Generally unnecessary. Priti will set up n-way for eligible members for cases where needed. Cary expects RT architecture/software conventions document to be released in the next month or two, removing most of the issues.
- Communication: Use of reflector encouraged for discussion. NDA communication will be handled with password-protected PDF once Intel sets up n-way.
- Available documents: Parties with existing, relevant documents (includes Sun, HP) will send them to group.
- Intellectual property: Participants don't expect problems with release of any of their IP. Microsoft has extensive patents in the area, but they are excessively broad (covering obvious ideas and prior art), so expectation is that they are not a problem. Nonetheless, we should be aware of them.

Please send corrections to Jim Dehnert.

C++ ABI Open Issues

Revised 6 October 1999

Revisions

[990905] New issue F-4. Additions to A-6, B-5, B-6, C-4, D-5, D-7.

[990929] Additions to D-*, D-9.

[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.						

#	Issue	Class	Status	Source	Opened	Closed
A-5	Empty parameters	data	closed	SGI	990520	990701
Summary: When passing a parameter with an empty class type by value, what is the convention?						
Resolution: 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.						

#	Issue	Class	Status	Source	Opened	Closed
A-6	RTTI .o representation	data call ps	open	SGI	990520	

Summary: Define the data structure to be used for RTTI, that is:

- for user `type_info` calls;
- for dynamic_cast implementation; and
- for exception-handling.

[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] (Proposal replaced by later version on 6 October.)

[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.

[991006 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:

. 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*".

Deliberations

The following conclusions were arrived at by the attending members of the C++ IA-64 ABI group:

- The exact layout for type_info objects is dependent on whether a 32-bit or 64-bit model is supported.
- Advantage should be taken of COMDAT sections and symbol preemption: two type_info pointers point to equivalent types if and only if the pointers are equal.
- A simple dynamic_cast algorithm that is efficient in the common case of base-to-most-derived cast case is preferrable over more sophisticated ideas that handle deep-base-to-in-between-derived casts more efficiently at a slight cost to the common case. Hence, the original scheme of providing a hash-table into the list of base classes (as is done e.g. in the HP aC++ compiler) has been dropped.
- The GNU egcs development team has implemented an idea of this ABI group to accelerate dynamic_cast operations by a-posteriori checking a "likely outcome". The interface of std::__dynamic_cast therefore keeps the src2dst_offset hint.
- std::__extended_type_info is dropped.

The full proposal has been incorporated in the ABI data layout document.

#	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.

[991004 SGI]

C++ ABI: COMDAT Proposal

Introduction

C++ has many situations where the compiler may need to emit code or data, but may not be able to identify a unique compilation unit where it should be emitted. The approach chosen by the C++ ABI group to deal with this problem, is to allow the compiler to emit the required information in multiple compilation units, in a form which allows the linker to remove all but one copy. This is essentially the idea called COMDAT in several existing implementations.

Our objectives include:

- Use existing structures as far as possible, to minimize impact on existing tools.
- Minimize impact on the linker by defining the unit of duplication as a section.
- Maximize generality. The C++ needs are rather varied, and similar needs from other languages should also be handled.
- In general, duplicated code or data sections are accompanied by additional duplicated sections, e.g. containing debug information. We want to define a mechanism which can deal with arbitrary such associations, without predicting them in advance.
- It is conceivable that duplicates may occur for something which is not recognized as a full duplicate at compile time. A possible example is a template instantiation which duplicates a full specialization in some compilation unit. It is desirable to deal with this case if possible.

Proposal

The proposal below is based on the HP definition, with minor modifications and more precise definitions.

SHF_COMDAT: Potentially Duplicate Sections

A section which may be duplicated, and for which only one copy is required, is identified by a new section header attribute:

SHF_COMDAT

> This section is subject to duplication in other relocatable objects, and the linker should choose only one copy to retain. This section must be referenced in a SHT_COMDAT_GROUP section (see below).

This attribute flag may be set in any section header, and no other modification or indication is made in the potentially duplicated sections. All additional information is contained in the associated SHT_COMDAT_GROUP section (see below).

SHT_COMDAT_GROUP: COMDAT Group Definition

COMDAT sections are assumed to occur in interrelated groups. For instance, an out-of-line definition of an inline function might require, in addition to its .text section, a read-only data section containing literals referenced, one or more debug information sections, and/or other informational sections. Furthermore, there may be internal references among these sections that would not make sense if one of them were replaced by a duplicate from another object. Therefore, we assume that such groups are to be included or omitted from the linked object as a unit. To facilitate this, we define a SHT_COMDAT_GROUP section:

The section header attributes of a COMDAT Group Section are:

name	`.COMDAT_group`
`sh_type`	`SHT_COMDAT_GROUP`
`sh_link`	`.symtab` section index
`sh_info`	symbol index
`sh_flags`	none
`sh_entsize`	size of section indices (4)
requirements	may not be stripped

The COMDAT group's `sh_link` field identifies a symbol table section, and its `sh_info` field the index of a symbol in

that section. The name of that symbol is treated as the identifier of the COMDAT group. If multiple COMDAT groups in different object files are identified by symbols with the same name, the linker should remove all but one of the groups. If the identifying symbol is defined in a non-COMDAT section in some object, the linker should remove all of the COMDAT groups identified by that symbol.

The section data of a SHT_COMDAT_GROUP section is a flag word followed by a sequence of section indices. The flag word may contain the following flags:

<To be defined>

The section indices in the SHT_COMDAT_GROUP section identify the sections which make up the group. Either none of them or all of them will be removed by the linker.

The `sh_size` value is `sh_entsize` times one plus the number of sections in the group.

Requirements

- References to the sections comprising a COMDAT group, from sections outside the group, must be made via global symbols. They may not reference local symbols for addresses in the group's sections, including section symbols.
- There may not be non-symbol references to the sections comprising a COMDAT group from sections outside the group, e.g. in sh_link fields. For example, relocations of one of the group's sections must be in a relocation section which is also part of the group.

  The above rules allow a COMDAT group to be removed without leaving dangling references, with only minimal processing of the symbol table.
- An entry in a symbol table section not in the COMDAT group, defined as an address in one of the group's sections, should be handled as follows:
  - If global but not referenced, remove it.
  - If global and referenced, change it to UNDEF. (This may cause a link error if the retained version of the COMDAT group does not define the same symbol.)
  - If local, remove it. (This will cause a link error if the symbol is referenced.)
- The SHT_COMDAT_GROUP section must precede the sections in the group (in the section table).

Questions

- Do we want flags to specify checking prior to removal of duplicates, e.g. for identical sections, same defined global symbols, etc.? If so, should there be one flags word per section index, instead of per group? (We don't see a need, but this was suggested in other proposals.)
- Do we want more control over when global symbols are removed vs. being converted to UNDEF? Alternatively, should we simply require that all symbols defined as addresses in the group be removed, and that references to them from outside do so via distinct UNDEF global symbols?
- Do we want to replace the symbol rule by simply requiring that any symbols defined as addresses in the group be defined in a .symtab section that is itself in the group?

[991005 SGI]

gABI: Section Indices

Background

The following proposal attempts to remove the limitation of 64K sections. Obviously, even if the problem is real, it will actually arise in very few compilation units. Therefore, the elements of the proposed solution are defined so as to leave unchanged object files which do not encounter the problem. We consider this compatibility objective as primary -- much more important than performance or clean definitions for the problematic object files -- particularly as it should allow vendors to merge the solution into existing tool chains at convenient times without disrupting existing programs.

Proposed ABI wording is in normal font; commentary is in italics. Section numbers are from the Intel IA-64 psABI.

Proposed gABI Changes

General Approach

The range of section indices from 0xff00 (SHN_LORESERVE) to 0xffff (SHN_HIRESERVE) is reserved for special purposes, and the gABI already forbids real sections with these indices. Our approach is to deal with situations where section indices cannot be compatibly expanded to a full 32 bits by using one of these indices as an escape value indicating that the actual index will be found elsewhere.

4.1 Elf Header

The ELF header has two relevant 16-bit fields: e_shnum contains the section count, and e_shtrndx the index of a string section. We modify their descriptions, and add two new optional fields as follows:

`ElfXX_Half e_shnum;`

> This member holds the number of entries in the section header table. Thus the product of `e_shentsize` and `e_shnum` gives the section header table's size in bytes. If a file has no section header table, `e_shnum` holds the value zero.
>
> If the number of sections is greater than `SHN_LORESERVE` (`0xff00`), this member has the value `SHN_XINDEX` (`0xffff`), and the actual number of section header table entries is in the member `e_xshnum` (below).

`ElfXX_Half e_shstrndx;`

> This member holds the section header table index of the entry associated with the section name string table. If the file has no section name string table, this member holds the value `SHN_UNDEF`. See ``Sections'' and ``String Table'' below for more information.
>
> If the section name string table index is greater than `SHN_LORESERVE` (`0xff00`), this member has the value `SHN_XINDEX` (`0xffff`), and the actual index of the section name string table is in the member `e_xshstrndx` (below).

`ElfXX_Word e_xshnum;`

> This member holds the number of entries in the section header table, if that number is too large to fit in `e_shnum`. It need not be present in the ELF header if it is not needed, and `e_ehsize` should reflect its presence or absence.

`ElfXX_Word e_xshstrndx;`

> This member holds the section header table index of the entry associated with the section name string table, if that index is too large to fit in `e_shstrndx`. It need not be present in the ELF header if it is not needed (in which case `e_xshnum` is also needed), and `e_ehsize` should reflect its presence or absence.

4.2 Sections

We define a new special section index as an escape value for large section indices, as referenced above:

`SHN_XINDEX (0xffff)`

> This special section index means, conventionally, that the actual section index is too large to fit in the field where it appears, and is to be found in another location (specific to the structure where it appears).

We note here that the section header contains two fields commonly used to hold section indices, `sh_link` and `sh_info`, but they are already defined as ElfXX_Word, and require no change.

A new section type is defined:

`SHT_SYMTAB_IDX (17)`

> A section of this type is paired with an SHT_SYMTAB section, if any of the symbols in that section reference a section index larger than 16 bits. It contains a table of 32-bit section indices, one for each symbol in the symbol table section, in the same order.
>
> The `sh_link` field of this section contains the index of the associated `SHT_SYMTAB` section.

A new special section name is defined:

`.symtab_idx`

> This section holds a section header index table for an associated `.symtab` section. The section's attributes will include the `SHF_ALLOC` bit if the associated `.symtab` section does; otherwise, that bit will be off.

There is no available field to point from the `.symtab` section to its associated `.symtab_idx` section, so we use the `sh_link` field in the latter to point back. It is recommended (but not required) that implementations place the `.symtab_idx` section immediately after its associated `.symtab` section (in the section header table) to make it easy for the linker to find.

4.x Symbol Table

The symbol table is the most problematic. It has no convenient location for an expanded section index. Therefore, we propose that the escape value imply redirection to a separate, parallel table containing full-size section indices.

Modify the definition of `st_shndx` as follows:

`st_shndx`

> Every symbol table entry is defined in relation to some section. This member holds the relevant section header table index.
>
> As the `sh_link` and `sh_info` interpretation table and the related text describe, section indexes in the range 0xff00 to 0xffff indicate special meanings. In particular, `SHN_XINDEX (0xffff)` indicates that the real index is too large to fit in this field, and must be found in the associated SHT_SYMTAB_IDX table (above).

If any of the `st_shndx` fields in a symbol table section contain the value `SHN_XINDEX (0xffff)`, there must be an associated `SHT_SYMTAB_IDX` section, with a `sh_link` field containing the index of this `SHT_SYMTAB` section. That section contains an array of 32-bit section indices, matching the symbol table entries 1-1 in the same order. Entries corresponding to `SHN_XINDEX (0xffff)` values of `st_shndx` in the symbol table must contain the actual section header index to be used. Others should contain either the correct section header index (i.e. duplicating the value in `st_shndx`), or zero.

Compatibility

There should be no compatibility impact on existing environments, since only very large section counts require object file changes. Individual vendors can postpone implementation until convenient, with no impact on typical programs.

#	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]

The current full proposal has been incorporated in the ABI data layout document.

[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.

[990930 Cygnus -- Jason] Jason has updated the Vtable layout description in abi-layout.html to reflect the approach from Cygnus and IBM.

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

#	Issue	Class	Status	Source	Opened	Closed

B-8	dynamic_cast	data	open	SGI	990628

Summary: 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*?

[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

. 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.

#	Issue	Class	Status	Source	Opened	Closed
C-4	Calling vfuncs in ctors/dtors	call	open	Cygnus	990603	
Summary: 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.						

[990630 HP -- Christophe] A rough idea from Christophe's original vtable layout proposal has been incorporated in the ABI data layout document.

#	Issue	Class	Status	Source	Opened	Closed
C-5	Calling destructors	call	open	Sun	990603	
Summary: 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?						

[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.
- [990922 HP -- Christophe]

Here is a quick description of the personality routine interface and semantics. This description is a slight extension of the existing personality routine implemented by HP for IA64. The extension is to allow multiple runtimes from possibly different vendors or for possibly different languages to cooperate in processsing an exception.

This document assumes that the chapter 11 of the Intel/HP "IA-64 = Software Conventions and Runtime Architecture" document is known to = the reader.

INTERFACE:

The complete exception processing framework consists of at least the = following routines: `_RaiseException`, `_ResumeUnwind`, = `_DeleteException`, `_Unwind_getGR`, = `_Unwind_setGR`, `_Unwind_getIP`, `_Unwind_setIP`, = `_Unwind_getLanguageSpecificData`, = `_Unwind_getRegionStart`. In addition, a language and vendor = specific personality routine will be stored by the compiler in the = unwind descriptor for the stack frames requiring exception = processing.

UNWIND RUNTIME ROUTINES:

The unwind runtime routines have the following interface and = semantics (all routines are `extern "C"`):

```
uint64 _RaiseException(uint64 exception_class, void = *exception_object);
```

> Raise an exception, passing along the given exception class and = exception object. The exception object has been allocated by the = language-specific runtime, and has a language-specific format. = `_RaiseException` does not return, unless an error condition is = found (such as no handler accepting to handle the exception, bad stack = format, etc).
>
> The first 4 words (32 bytes) of the exception object = are allocated for use exclusively by the unwinder, and should not be = written by the personality routine or other parts of the = language-specific runtime. The first word is used to store the exception = class. The second word points to the personality routine of the frame = that threw the exception intially. The two next words are reserved for = use by the unwinder. [Note: Typical use is to keep the state of the = unwinder while executing user code, such as our current frame_handle = pointer.]

```
void _ResumeUnwind (void = *exception_object);
```

> Resume propagation of an = existing exception. [Note: _ResumeUnwind should not be used to implement = rethrowing. To the unwinding runtime, the catch code that rethrows was a = handler, and the previous unwinding session was terminated before = entering it.] [Note: Compared to HP runtime, the exception class = and frame handle arguments have been removed. They also need no longer = be passed to the landing pads. Instead, the unwinder will store the = information in one of its 2 reserved words.]

```
void _DeleteException(void = *exception_object);
```

> If a given runtime resumes = normal execution after catching a foreign exception, it will not know = how to delete that exception. This exception will be deleted by calling = `_DeleteException`, which in turn will delegate the task to the = original personality routine (see `EH_DELETE_EXCEPTION_OBJECT` =

below).

```
uint64 _Unwind_getGR(void *context, int index);
uint64 = _Unwind_getIP(void *context);
void _Unwind_setGR(void *context, int = index, uint64 new_value);
void _Unwind_setIP(void *context, uint64 = new_value);
```

Get or set registers from the given = unwinder context. The 'context' argument is the same argument passed to = the personality routine (see below).[Note: Minor changes compared to the = existing unwinding interface, mostly to hide the register = classes]

```
uint64 _Unwind_getLanguageSpecificData(void = *context)
```

Get the address of the language-specific = data area for the current stack frame. The '`context`' argument = is the same argument passed to the personality routine.[Note: This is = not stricly required: it could be accessed through getIP using the = documented format of the UnwindInfoBlock, but since this work has been = done for finding the personality routine in the first place, it makes = sense to cache the result in the context, as we currently = do]

```
uint64 _Unwind_getRegionStart(void = *context)
```

Get the address of the beginning of the = current procedure or region of code. [Note: This is required for us = because we store data relative to the beginning of the code. So let's = make it mandatory ;-]

PERSONALITY ROUTINE:

The personality routine is defined with the following = interface:

```
int = PersonalityRoutine
    (int = version,
     int = phase,
     UInt64 = exceptionClass,
     void * = exceptionObject,
     void = *context);
```

[Note: the frame_handle argument has been removed: it was used only = once in the runtime, and the cost of reading it back from the exception = object is really minimal, compared to the cost of having to spill it in = all landing pads... The context argument type has been made opaque]

The arguments have the following role and meanings:

- **version**: Version number that the compiler and personality = routine agree on, identifying for instance language-specific table = format. This version number is read from the unwind information block = (unwind tables)
- **phase**: Indicates what processing the personality routine is = supposed to perform. The possible actions are described below under = '**UNWINDING PHASES**'
- **exceptionClass**: An 8-bytes identifier specifying the type of = the thrown exception. By convention, the high 4 bytes indicate the = vendor (for instance HP\0\0), and the low 4 bytes indicate the language = (for instance C++\0.) [Note: For C++, it is expected that agreement will = be reached on a common 'exceptionObject', but different vendors may = still chose to have different personality routines with different table = formats.]
- **exceptionObject**: The pointer to a memory location recording = the necessary information for processing the exception according to the = semantics of a given language. [Note: For C++, it is assumed that the = format of this exception object can be agreed upon, even if we disagree = on the LSDA and/or landing pad registers or similar = details.]
- **context**: Unwinder state information for use by the personality = routine. This is used by the personality routine in particular to access = the frame's registers. [Note: I don't see how anything could work = without a minimal common unwinder interface - which is why it has been = defined above]

- **return value**: The return value from the personality routine = indicates how further undinwind should happen, as well as possible error = conditions. See "**UNWINDING PHASES**" below for = details.

UNWINDING PHASES

Unwinding is a 2-phases process.

- PASS 1 unwinds through the stack, looking for a "handler", = that is a code that has the potential to stop the exception propagation. = For C++, this would be a 'catch' clause. The first pass can do a = "quick" unwind, meaning it does not need to maintain full = registers state.

- PASS 2 starts once a handler has been found. For each stack frame = that requires some cleanup, it performs that cleanup. For C++, this = would be destructors in addition to catch clauses. If compensation code = for some optimization is required, this is also the pass this code will = be executed. During that pass, the stack is actually unwound, and full = register state is restored prior to executing any cleanup, compensation = or handler code.

[Note: Cleanup code is code doing some user-defined cleanup such as = destructors. Compensation code is code inserted by the compiler to = compensate for an optimization that moved code past the throwing call. = Handler code is user-defined code that possibly can resume normal = execution]

The unwinding phase argument to the personality routine is a bitwise = or of the following constants:

- `EH_SEARCH_PHASE` =3D = 1: Indicates that the personality routine should check if the current = frame contains a handler, and if so return `EH_HANDLER_FOUND`, or = otherwise return `EH_CONTINUE_UNWIND`. `EH_SEARCH_PHASE` = cannot be set at the same time as = `EH_CLEANUP_PHASE`.
- `EH_CLEANUP_PHASE` =3D = 2: Indicates that the personality routine should perform cleanup for the = current frame. The personality routine can perform this cleanup itself, = by calling nested procedures, and return `EH_CONTINUE_UNWIND` [= Note: This is required to support the Intel nested procedures model]. = Alternatively, it can setup the registers (including the IP) for = transferring control to a "landing pad", and return = `EH_INSTALL_HANDLER` (See "**TRANSFERRING CONTROL TO A = LANDING PAD**" below).
- `EH_HANDLER_FRAME` = =3D 4: During pass 2, indicates to the personality routine that the = current frame is the one which was flagged as the handler frame during = pass 1.
- `EH_DELETE_EXCEPTION_OBJECT` =3D 8: = During pass 2, indicates that the runtime that actually caught the = exception does not know how to delete it, and called = `_DeleteException`. '`context`' should not be used in that = case.
- `EH_FATAL_PASS2_ERROR` =3D 16: During = pass 2, indicates that a fatal unwinding error occured. In that case, = the personality routine should not return. This is sent to the original = personality routine associated to the initial exception object. [Note: = This is required if we want to ensure that `_ResumeUnwind` never = returns, and if we also want to be able to call `terminate()` in = the case a stack inconsistency is found during pass 2. An error detected = during pass 1 is reported by returning from = `_RaiseException`.]

- `EH_FORCE_UNWIND` =3D 32: During pass 2, indicates that = no language is allowed to "catch" the exception. This flag is = set while unwinding the stack for setjmp or during thread cancellation. = User-defined code in a catch clause may still be executed, but the catch = clause has to resume unwinding at its end.

TRANSFERRING CONTROL TO A LANDING = PAD:

In the case the personality routine wants to transfer control to a = landing pad, it setups registers (including IP) to suitable values for = entering the landing pad. Prior to executing code in the landing pad, = registers not altered by the personality routine will be restored to the = exact state they were in that frame before the call that threw the = exception.

The landing pad can either resume execution to normal (as, for = instance, at end of a C++ `catch`), or resume unwinding by = calling the `_ResumeUnwind` function and passing it the = '`exceptionObject`' argument received by the personality routine. = `_ResumeUnwind` will never return.

`_ResumeUnwind` should be called if and only if the = personality routine did not return `EH_HANDLER_FOUND` during =

phase 1. In other words, the unwinder can allocate some resources (for = instance memory) and keep track of them in the exception object reserved = words. It should then free these resources before transferring control = to the last (handler) landing pad. It does not need to free the = resources before entering non-handler landing-pads, since = `_ResumeUnwind` will ultimately be called.

The landing pad will receive various arguments from the runtime, = typically passed in registers set using `_Unwind_setGR` by the = personality routine. For a landing pad that can lead to = `_ResumeUnwind`, one argument must be the = `exceptionObject` pointer, which must be preserved to be passed = to `_ResumeUnwind`. [Note: Thanks to the 4 reserved words in the = exception object, 2 landing-pad arguments have been eliminated.] The = landing pad may receive other arguments, for instance a 'switch value' = indicating the type of the exception being caught.

RULES FOR CORRECT INTER-LANGUAGE = OPERATION:

The following rules must be observed for correct operation between = languages and/or runtimes from different vendors:

- An exception which has an unknown class must not be altered by the = personality routine. The semantics of foreign exception processing = depend on the language of the stack frame being unwound. This covers in = particular how exceptions from a foreign language are mapped to the = native language in that frame.
- If a runtime resumes normal execution, and the caught exception was = created by another runtime, it should call `_DeleteException`. = This is true even if it understands the exception object format (such as = would be the case between different C++ runtimes). [Note: This is = because the other runtime might have to update some global variables = that point to the exception being deleted.]
- A runtime is not allowed to catch an exception if the = EH_FORCE_UNWIND flag was passed to the personality = routine.

CATCHING FOREIGN EXCEPTIONS IN C++

Foreign exception can be caught in a `catch(...)`. They can = also be caught as if they were of a `__foreign_exception` class, = defined in `<exception>`. [Note: The = `__foreign_exception` may have subclasses, such as = `__java_exception` and `__ada_exception`, if the runtime = is capable of identifying some of the foreign languages.]

The behavior is undefined in the following cases:

- A `__foreign_exception` catch argument is accessed in any way = (including taking its address).
- A `__foreign_exception` is active at the same time as another = exception (either there is a nested exception while catching the foreign = exception, or the foreign exception was itself = nested)
- `uncaught_exception()`, `set_terminate()`, = `set_unexpected()`, `terminate()` or `unexpected()` = is called at a time a foreign exception exists (for instance, calling = `set_terminate`() during unwinding of a foreign = exception)

[Note: All these cases might involve accessing the C++ specific = content of the thrown exception, for instance to chain active = exceptions]

Otherwise, a catch block catching a foreign exception is = allowed:

- To resume normal execution, thereby stopping propagation of the = foreign exception and deleting it,
- Or to rethrow the foreign exception. In that case, the original = exception object should have been unaltered in any way by the = C++ runtime.

A catch-all block may be executed during forced unwinding. For = instance, a setjmp may execute code in a catch(...) during stack = unwinding. However, if this happens, unwinding will proceed at the end = of the catch-all block, whether or not there is an explicit = rethrow.

Setting the low 4 bytes of exception class to C++\0 is = reserved for use by C++ runtimes compatible with the common = C++ ABI.

- [990923 All] Extensive discussion at the meeting was generally positive about the HP proposal. Several changes came up, ranging from editorial to substantive. Christophe will modify the specification.
  - Use *doubleword* instead of *word* for 8-byte items.
  - By the time the personality routine is called, the runtime either knows where the language-specific data area is, or can get it trivially. Therefore, pass it to the personality routine, instead of providing _Unwind_getLanguageSpecificData.
  - Most references to `setjmp` in the document should be to `longjmp`.
  - The description of EH_DELETE_EXCEPTION_OBJECT was unclear.
  - Clarify the distinction between passes 1 and 2, and the final steps (which are generally referred to as pass 2).
  - The group agreed that the performance benefit of allowing a simplified setjmp which supports only a full-unwinding longjmp is outweighed by the interoperability benefit of having a single setjmp which will support either a C-style direct or a C++ full-unwind longjmp.
  - We will follow the lead of IBM (?) and specify a distinct longjmp call which is defined to do a full-unwind longjmp.
  - Pthreads cancellation will be supported by specifying:
    - New exceptions (cancel and exit).
    - catch(...) always rethrows.
    - catch(...) catches anything, including foreign languages.

#	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.

[991006 All] The concensus of the group, from the discussion of the low-level exception API, is:

- A foreign-language exception, passing through a C++ frame, will cause normal destructor invocations exactly as a C++ exception would.

- Since catch(...) is commonly used as a cleanup mechanism (with a rethrow) rather than as a true catch, it is appropriate to catch and rethrow even foreign-language exceptions.
- Function exception specifications present a more complex issue. A foreign exception will generally not match any of the specified allowed exceptions (though implementations might deal with special cases). For consistency, it is expected that a foreign exception will be caught by a throw specification and not passed through, i.e. that unexpected() will be called.

#	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.						

[991006 All] This common ABI will not allow throwing exceptions from a signal 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.)

[990923 All] We agreed that:

- We will use a single setjmp which retains enough information for a traditional C direct longjmp.
- We will define a new longjmp call which always does full unwinding.
- Implementations may implement longjmp as either the direct or the full-unwind form, as a default, or using a user option.
- catch(...) will catch all exceptions, including foreign-language ones. It will always rethrow.

See the HP low-level exception writeup at the beginning of the exception issues section.

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.

#	Issue	Class	Status	Source	Opened	Closed
F-4	Empty throw specs	call g	open	HP	990930	

Summary: It is useful to be able to identify functions with empty throw specifications, to allow calling of unexpected() from the runtime during unwinding. Can this specification be merged into the function's name mangling?

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.)

#	Issue	Class	Status	Source	Opened	Closed

G-3	Inlined routine linkage	call	open	Sun	990603
Summary: 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.					

[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.

#	Issue	Class	Status	Source	Opened	Closed
G-4	Dynamic init of local static objects and multithreading	call	open	SCO	990607	
Summary: 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.						

[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.

C++ ABI Closed Issues

Revised 17 August 1999

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

A. 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).						

[990610 All] Given the absence of addressing modes with displacements on IA-64, the consensus is to answer this question with "first."

[990617 All] Given a Vptr and only non-polymorphic bases, which (Vptr or base) goes at offset 0?

- HP: Vptr at end, but IA-64 is different because no load displacement
- Sun: Vptr at 0 probably preferred
- g++: Vptr at end today

Tentative decision: Vptr always goes at beginning.

[990624 All] Accepted tentative decision. Rename, close this issue, and open separate issue (B-6) for Vtable layout.

#	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?

[990610 Matt] With regard to how data member accesses are handled, the choices are to store either a pointer or an offset in the Vtable. The concensus seems to be to prefer an offset.

[990617 All] Any number of empty virtual base subobjects (rare) will be placed at offset zero. If there are no non-virtual polymorphic bases, the first virtual base subobject with a Vpointer will be placed at offset zero. Finally, all other virtual base subobjects will be allocated at the end of the class, left-to-right, depth-first.

[990624 All] Define an empty object as one with no non-static, non-empty data members, no virtual functions, no virtual base classes, and no non-empty non-virtual base classes. Define a nearly empty object as one which contains only a Vptr. The above resolution is accepted, restated as follows:

Any number of empty virtual base subobjects (rare, because they cannot have virtual functions or bases themselves) will be placed at offset zero, subject to the conflict rules in A-3 (i.e. this cannot result in two objects of the same type at the same address). If there are no non-virtual polymorphic base subobjects, the first nearly empty virtual base subobject will be placed at offset zero. Any virtual base subobjects not thus placed at offset zero will be allocated at the end of the class, in left-to-right, depth-first declaration order.

#	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.						

[990617 All] At offset zero is the Vptr whenever there is one, as well as the primary base class if any (see A-7). Also at offset zero is any number of empty base classes, as long as that does not place multiple subobjects of the same type at the same offset. If there are multiple empty base classes such that placing two of them at offset zero would violate this constraint, the first is placed there. (First means in declaration order.)

All other non-virtual base classes are laid out in declaration order at the beginning of the class. All other virtual base subobjects will be allocated at the end of the class, left-to-right, depth-first.

The above ignores issues of padding for alignment, and possible reordering of class members to fit in padding areas. See issue A-9.

[990624 All] There remains an issue concerning the selection of the primary base class (see A-7), but we are otherwise in agreement. We will attempt to close this on 1 July, modulo A-7.

[990701 All] This issue is closed. A full description of the class layout can be found in issue A-9. (At this time, A-7 remains to be closed, waiting for the Taligent rationale.)

#	Issue	Class	Status	Source	Opened	Closed
A-4	Empty base classes	data	closed	SGI	990520	990624
Summary: Where are empty base classes allocated? (An empty base class is one with no non-static data members, no virtual functions, no virtual base classes, and no non-empty non-virtual base classes.)						

[990624 All] Closed as a duplicate of A-3.

#	Issue	Class	Status	Source	Opened	Closed

A-5	Empty parameters	data	closed	SGI	990520	990701
Summary: When passing a parameter with an empty class type by value, what is the convention?						

[990623 SGI] We propose that no parameter slot be allocated to such parameters, i.e. that no register be used, and that no space in the parameter memory sequence be used. This implies that the callee must allocate storage at a unique address if the address is taken (which we expect to be rare).

[990624 All] In addition to the address-taken case, care is required if the object has a non-trivial copy constructor. HP observes that in (some?) such cases, they perform the construction at the call site and pass the object by reference.

[990625 SGI -- Jim] I understand that the Standard explicitly allows elimination of even non-trivial copy construction in some cases. Is this one of them? Where should I look? Also, of course, varargs processing for elided empty parameters would need to be careful.

I have opened a new issue (C-7) for passing copy-constructed parameters by reference. Since doing so would turn an empty value parameter into a non-empty reference parameter, this issue can ignore such cases.

[990701 All] An empty parameter will not occupy a slot in the parameter sequence unless:

1. its type is a class with a non-trivial copy constructor; or
2. it corresponds to the variable part of a varargs parameter list.

Daveed and Matt will pursue the question of when copy constructors may be ignored for parameters with the Core committee, and if they identify cases where the constructors may clearly be omitted, those (empty) parameters will also be elided.

#	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.						

[990617 All] It will be shared with the first polymorphic non-virtual base class, or if none, with the first nearly empty polymorphic virtual base class. (See A-2 for the definition of *nearly empty*.)

[990624 All] HP noted that Taligent chooses a base class with virtual bases before one without as the primary base class), probably to avoid additional "this" pointer adjustments. SGI observed that such a rule would prevent users from controlling the choice by their ordering of the base classes in the declaration. The bias of the group remains the above resolution, but HP will attempt to find the Taligent rationale before this is decided.

[990729 All] Close with the agree resolution. If a convincing Taligent rationale is found, we can reconsider.

#	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? (An alternative for virtual bases: allow the virtual base to move in the complete object.)						

[990623 SGI] We propose that the alignment of a class be the maximum alignment of its virtual and non-virtual base classes, non-static data members, and Vptr if any.

[990624 All] Above proposal accepted. (SGI observation: the size of the class is rounded up to a multiple of this alignment, per the underlying psABI rules.)

#	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 separate writeup of Data Layout.						

[990610 all] Some participants want to avoid attempts to reorder members differently than the underlying C struct ABI rules. Others think there may be benefit in reordering later access sections to fill holes in earlier ones, or even in base classes.

[990617 all] There are several potential reordering questions, more or less independent:

1. Do we reorder whole access regions relative to one another?
2. Do we attempt to fill padding in earlier access regions with initial members from later regions?
3. Do we fill the tail padding of non-POD base classes with members from the current class?
4. Do we attempt to fill interior padding of non-POD base classes with later members?

There is no apparent support for (1), since no simple heuristic has been identified with obvious benefits. There is interest in (2), based on a simple heuristic which might sometimes help and will never hurt. However, it is not clear that it will help much, and Sun objects on grounds that they prefer to match C struct layout. Unless someone is interested enough to implement and run experiments, this will be hard to agree upon. G++ has implemented (3) as an option, based on specific user complaints. It clearly helps HP's example of a base class containing a word and flag, with a derived class adding more flags. Idea (4) has more problems, including some non-intuitive (to users) layouts, and possibly complicating the selection of bitwise copy in the compiler.

[990624 all] We will not do (1), (2), or (4). We will do (3). Specifically, allocation will be in modified declaration order as follows:

1. Vptr if any, and the primary base class per A-7.
2. Any empty base classes allocated at offset zero per A-3.
3. Any remaining non-virtual base classes.
4. Any non-static data members.
5. Any remaining virtual base classes.

Each subobject allocated is placed at the next available position that satisfies its alignment constraints, as in the underlying psABI. This is interpreted with the following special cases:

1. The "next available position" after a non-POD class subobject (base class or data member) with tail padding is at the beginning of the tail padding, not after it. (For POD objects, the tail padding is not "available.")
2. Empty classes are considered to have alignment and size 1, consisting solely of one byte of tail padding.
3. Placement on top of the tail padding of an empty class must avoid placing multiple subobjects of the same type at the same address.

After allocation is complete, the size is rounded up to a multiple of alignment (with tail padding).

[990722 all] The precise placement of empty bases when they don't fit at offset zero remains imprecise in the above.

Accordingly, a precise layout algorithm is described in a separate writeup of Data Layout.

[990729 all] The layout writeup was accepted, with the first choice for empty base placement. That is, if placement at offset zero doesn't work, it will be placed like a normal base/member. The concensus was that this won't happen often, and such bases will often overlap with the preceding tail padding or following components anyway. Jim will modify the writeup accordingly.

#	Issue	Class	Status	Source	Opened	Closed
A-10	Class parameters in registers	call	closed	HP	990603	990710
Summary: The C ABI specifies that 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?						

[990701 all] A separate issue (C-7) deals with cases where a non-trivial copy constructor is required; we ignore those cases here. Our conclusion is that, without a non-trivial copy constructor, we need not be concerned about the class object moving in the process of being passed, and there is no need to use a mechanism different from the base ABI C struct mechanism. At the same time, if we do use the underlying C struct mechanism, the user has complete control of the passing technique, by choosing whether to pass by value or reference/pointer.

Therefore, except in cases identified by issue C-7 for different treatment, class parameters will be passed using the underlying C struct protocol.

#	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, described below.						

[990729 All] Jason described the g++ implementation, which is a three-member struct:

1. The adjustment to *this*.
2. The Vtable index plus one of the function, or -1. (Zero is a NULL pointer.)
3. If (2) is an index, the offset from the full object to the member function's Vtable. If -1, a pointer to the function (non-virtual).

A concern about covariant returns was raised. It was observed that, given our decision to use distinct Vtable entries for distinct return types, no further concern is required here. Others will describe their representations. IBM has an alternative, but it is believed to be patented by Microsoft.

[990805 All] It is agreed that a two-element struct will be used for a pointer to a member function, with elements as follows:

`ptr`:

> For a non-virtual function, this field is a simple function pointer. (Under current base IA-64 psABI conventions, this is a pointer to a GP/function address pair.) For a virtual function, it is 1 plus twice the Vtable offset of the function. The value zero is a NULL pointer.

`adj`:

> The required adjustment to *this*.

Although we agreed to close this, SGI suggests a minor modification. Since the Vtable offset of a virtual function will always be even, we suggest that it not be doubled before adding 1. This is because shifts are more restricted on many

processors than other integer ALU operations (shifters are large structures), so an XOR or NAND will often be cheaper than a right shift.

[990812 All] Close this issue with the suggested modification.

#	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.						

[990701 Michael Lam] Michael will check what the Sun ABI treatment is and report back.

[990729 All] A separate issue raised in conjunction with A-7 is whether to include Vfunc pointers in the primary Vtable for functions defined only in the base classes and not overridden. If the primary and secondary Vtables are concatenated, this is no longer an issue, since all can be referenced from the primary Vptr.

[990805 All] All of the Vtables associated with a class will be concatenated, and a single external symbol used (to be identified as part of the mangling issue F-1). The order of the tables will be the same as the order of base class subobjects in an object of the class, i.e. first the primary Vtable, then the non-virtual base classes in declaration order, and finally the virtual base classes in depth-first declaration order.

#	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.						

[990701 all] Define no special treatment for this case in the ABI. A translator with control over both caller and callee may choose to optimize.

#	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.						

[990729 SGI] We suggest an offset from the base address of the class, represented as a `ptrdiff_t`.

[990805 All] Such pointers are represented as one plus the offset from the base address of the class, as a `ptrdiff_t`. NULL pointers are zero.

B. Virtual Function Handling Issues

#	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.						

[990610 Matt] One possibility is to have two Vtable entries, which might point to different functions, different entrypoints, or a real entrypoint and a thunk. Another is to return two result pointers (base/derived), and have the caller select the right one.

[990715 All] Daveed presented his multiple-return-value scheme, including an example that involved virtual base classes, return values that are pointers to nonpolymorphic classes, and other equally horrible things.

Consensus: we need to get the horrible cases correct, but speed only matters in the simple case. The simple case: class B has a virtual function f returning a B1* and class D has a virtual function f returning a D1*, where all four classes are polymorphic, B is a primary base of D, and B1 is a primary base of D1. (The really important case is where B1 is B and D1 is D, but that simplification doesn't make any difference.)

Jason: Would the usual multiple-entry-point scheme work just as well? That is, would it be just as fast as Daveed's scheme in the simple case, and still preserve enough information for the more complicated cases? It appears so, but we don't have a proof. Jason will try to provide one.

[990716 Cygnus -- Jason] Proof? You always know what types a given override must be able to return, and you know how to convert from the return type to those base types. You know from the entry point which type is desired. Seems pretty straightforward to me.

[990716 Cygnus -- Jason] The alternative I was talking about yesterday goes something like this:

When we have a non-trivial covariant return situation, we create a new entry in the vtable for the new return type. The caller chooses which vtable entry to use based on the type they want.

This could be implemented several ways, at the discretion of the vendor:

1. Multiple entry points to one function, with an internal flag indicating which type to return.
2. Thunks which intercept the function's return and modify the return value. Note that unlike the case of calling virtual functions, for covariant returns we always know which adjustments will be needed, so we don't have to pay for a long branch. We do, however, lose the 1-1 correspondence between calls and returns, which apparently affects performance on the Pentium Pro.
3. Function duplication.

The advantage of this approach to the complex case is that we don't have to do a dynamic_cast when faced with multiple levels of virtual derivation. It is also strictly simpler; Daveed's model already requires something like this in cases of multiple inheritance.

Of course, we can always mix and match; we could choose to only do this in cases of virtual inheritance, or use Daveed's proposal and do this only in cases of repeated virtual inheritance. In that case, the multiple returns would just be an optimization for the single virtual inheritance case.

Since we don't seem to care about the performance of anything but single nonvirtual inheritance, it seems simpler not to bother with multiple returns.

The remaining question is how to handle the case of nontrivial nonvirtual inheritance: do we use multiple slots or have the caller do the adjustment? My inclination is to have the caller adjust.

WRT patents, the idea of having the function return the base-most class and having the caller adjust is parallel to the patented Microsoft scheme whereby they pass the base-most class as the 'this' argument to virtual functions, but the word 'return' does not appear anywhere in the patent, so it seems safe.

[990722 All] The group was generally agreed that the simplicity of multiple entries in the vtable outweighed any space/performance advantage of more complex schemes (e.g. the method Daveed described on 15 July). Discussion focussed on whether it is worthwhile to eliminate some of the entries in cases where they are unnecessary because the caller knows the required conversion, namely when the return type has a unique non-virtual subobject of the original return type.

Agreement was reached to avoid the complication of eliminating some of the Vtable entries. Thus, the Vtable will have one entry for each accessible return type of a covariant virtual function. These may be implemented in a variety of ways, e.g. duplicated functions, separate entrypoints, or stubs, and the ABI need not specify the choice. The location of the Vtable entries is part of the separate Vtable layout issue B-6.

#	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.						

[990604 HP -- Christophe] Mike (Ball) gave me what I believe is an excellent definition of when caching is allowed. I'd like him to present it.

[990805 All] Christophe explained that the rule is simply that, within a call to a member function of the class, the class Vtable may not be modified. Between such calls, no assumption may be made. With this observation, the issue is closed.

[990812 All] The rule is even simpler. Once a program changes the type of a pointer's target, the pointer is invalidated, and its value may not be reused. Therefore, a code sequence which repeatedly refers to the same pointer value is invalid if the pointee's vtable has been changed.

#	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.						

[990624 All] Note that putting GP in the Vtable prevents putting it in shared memory. See B-7.

[990805 All] It was decided that special representations to accomodate shared memory would be expensive and therefore undesirable. Therefore, the decision is to put the function address/GP pair in the vtable, avoiding the cost of an extra indirection in using it.

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

[990624 All] Note that putting GP in the Vtable prevents putting it in shared memory. This interacts with B-4.

[990624 HP -- Cary] For a C++ object to be placed into shared memory, its vtable pointer must be valid in all processes that are sharing that object.

1. If the vtable can be placed in text, that would be fine, but the vtable contains function pointers (or descriptors) that require runtime relocation, so it must be in data.
2. We can place the vtables in shared memory, but only if the function pointers/descriptors are valid in all processes. The entry point addresses, which refer to shared text, should be shareable, but the gp values may not be identical for all processes. (RTTI pointers are also an issue, and could be solved by putting the RTTI information in shared memory as well.)
3. We can place the vtables in private memory, provided they are at the same address in all processes.

One way or another, we need a way of ensuring that a pointer from shared memory to private memory is valid in all processes, which means that we will need a means to ensure that certain shared library data segments can get mapped at the same address in all processes that load those certain libraries.

My wild idea a few years ago was to put the vtables in shared memory (by allocating and building them at load time, as Taligent did), and store a shared library index in place of the gp value in each function descriptor. Each process would have its own table of gp values, indexed by this shared library index, but the index space would be managed system-wide. The C++ runtime library would have been responsible for allocating a new index for each unique C++ shared library loaded on the system, then storing the process-local copy of the gp pointer in the appropriate slot of the table.

[990628 SGI -- Jim] Note a further problem with vtables in shared memory (Cary's point 2). If a virtual function comes from another DSO, it may be pre-empted differently in different programs. Hence, the function pointer itself is a problem even if the GP isn't.

[990701 All] An extensive discussion boiled down to a few points:

- The primary issue is objects in shared memory -- vtables aren't interesting in themselves, but rather because putting the object in shared memory implies having the vtable at the same address in all sharing processes.
- Many of us have a few customers asking for this. It is not clear just how extensive a facility they need, or how automatic it needs to be. We should attempt to gauge the need.
- Noone thinks we should penalize the non-shared case for the rare instances of shared demand.
- It is questionable whether we can define an ABI mechanism which will work on all of our systems, but we'd like not to preclude OS-specific extensions to do this if we can't.
- One possible approach would be an API allowing a user to place an object in shared memory, and then "install" it by setting its vtable pointers, possibly to copies also placed in the same shared memory.
- A more automatic approach would be something which allocated certain objects/vtables to shared memory, gave up at link time if not all pointers were internal to the object being linked, attempted to place the relevant segments at the same runtime addresses to allow sharing, and gave up on sharing if this was not possible. Such an approach

would perhaps still require some care on the part of the user to prevent problematic runtime situations.

These ideas are very fuzzy. Participants should think about the need and possibilities and attempt to identify more concrete approaches.

[990805 All] It was determined (largely based on consideration by Jason) that the only practical approach to putting objects in shared memory is to force the objects, Vtables, functions, etc. to the same addresses in the various processes involved. If this is done, data representation issues are irrelevant. Therefore, this issue is closed as moot.

Note that the base psABI defines a flag, EF_IA_64_ABSOLUTE, which forces an executable object to the addresses specified in ELF, so at least one method of representing this is already available.

C. Object Construction/Destruction Issues

#	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.						

[990701 All] Daveed and Matt will attempt to pin down the copy requirements with the Core committee, i.e. when a non-trivial copy constructor may be elided. The relevant Standard requirement is 12.8/15, and there is an open defect report related to this question. For cases where the ctor may not be elided, we expect to perform the copy at the call site, and pass a reference.

[990729 All] Matt will produce a clear proposal for when the ABI will elide the constructor (and therefore pass the class object like a normal C struct), based on the Standard's exceptions.

[990805 All] There are no cases where a non-trivial copy constructor can be simply elided for all instances of a particular parameter. Therefore, we shall use the consistent convention that, if a value parameter's (class) type has a non-trivial copy constructor, the caller will allocate space for it, perform the copy, and pass a reference.

Note that the standard does allow the caller, if the value being passed is a temporary, to construct the temporary directly into the parameter memory and elide the copy constructor call.

#	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).						

D. Exception Handling Issues

E. Template Instantiation Model Issues

F. Name Mangling Issues

G. Miscellaneous Issues

H. Library Interface Issues

Please send corrections to Jim Dehnert.

C++ ABI for IA-64: Data Layout

Revised 6 October 1999

Contents

Revisions

[991006] Added RTTI proposal.

[990930] Updated to new vtable layout proposal.

[990811] Described member pointer representations, virtual table layout.

[990730] Selected first variant for empty base allocation; removed others.

General

In what follows, we define the memory layout for C++ data objects. Specifically, for each type, we specify the following information about an object O of that type:

- the *size* of an object, *sizeof*(O);
- the *alignment* of an object, *align*(O); and
- the *offset* within O, *offset*(C), of each data component C, i.e. base or member.

For purposes internal to the specification, we also specify the *data size* of an object, *dsize*(O), which intuitively is sizeof(O) minus the size of tail padding.

Definitions

The descriptions below make use of the following definitions:

alignment of a type T (or object X)

A value A such that any object X of type T has an address satisfying the constraint that &X modulo A == 0.

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.

polymorphic class

A class requiring a virtual table pointer (because it or its bases have one or more virtual member functions or virtual base classes).

primary base class

For a polymorphic class, the unique base class (if any) with which it shares the Vptr at offset 0.

POD Data Types

The size and alignment of C POD types is as specified by the base (C) ABI. Type bool has size and alignment 1. All of these types have data size equal to their size. (We ignore tail padding for PODs because the Standard does not allow us to use it for anything else.)

Member Pointers

A pointer to data member is an offset from the base address of the class object containing it, represented as a `ptrdiff_t`. It has the size, data size, and alignment of a `ptrdiff_t`.

A pointer to member function is a pair as follows:

`ptr`:

For a non-virtual function, this field is a simple function pointer. (Under current base IA-64 psABI conventions, that is a pointer to a GP/function address pair.) For a virtual function, it is 1 plus twice the Vtable offset of the function. The value zero is a NULL pointer.

`adj`:

The required adjustment to *this*, represented as a `ptrdiff_t`.

It has the size, data size, and alignment of a class containing those two members, in that order. (For 64-bit IA-64, that will be 16, 16, and 8 bytes respectively.)

Non-POD Class Types

For non-POD class types C, assume that all component types (i.e. base classes and non-static data member types) have been laid out, defining size, data size, and alignment. Layout (of type C) is done using the following procedure.

I. **Initialization**

1. Initialize sizeof(C) to zero, align(C) to one, dsize(C) to zero.
2. If C is a polymorphic type:
   . If C has a polymorphic base class, attempt to choose a primary base class B. It is the first non-virtual polymorphic base class, if any, or else the first nearly empty virtual base class. Allocate it at offset zero, and set sizeof(C) to sizeof(B), align(C) to align(B), dsize(C) to dsize(B).
   b. Otherwise, allocate the vtable pointer for C at offset zero, and set sizeof(C), align(C), and dsize(C) to the appropriate values for a pointer (all 8 bytes for IA-64 64-bit ABI).

II. **Non-Virtual-Base Allocation**

For each data component D (i.e. base or non-static data member) except virtual bases, first the non-virtual base classes in declaration order and then the non-static data members in declaration order, allocate as follows:

1. If D is not an empty base class, start at offset dsize(C), incremented if necessary to alignment align(type(D)). Place D at this offset unless doing so would result in two components (direct or indirect) of the same type having the same offset. If such a component type conflict occurs, increment the candidate offset by align(type(D)), and try again, repeating until success occurs (which will occur no later than sizeof(C) incremented to the required alignment).

   Update sizeof(C) to max (sizeof(C), offset(D)+sizeof(D)). Update align(C) to max (align(C), align(D)). If D is a base class (not empty in this case), update dsize(C) to offset(D)+dsize(D). If D is a data member, update dsize(C) to max (offset(D)+dsize(D), offset(D)+1).
2. If D is an empty base class, its allocation is similar to the first case above, except that additional candidate offsets are considered before starting at dsize(C). First, attempt to place D at offset zero. If unsuccessful (due to a component type conflict), proceed with attempts at dsize(C) as for non-empty bases. As for that case, if there is a type conflict at dsize(C) (with alignment updated as necessary), increment the candidate offset by align(type(D)), and try again, repeating until success occurs.

   Once offset(D) has been chosen, update sizeof(C) to max (sizeof(C), offset(D)+sizeof(D)). Note that align(D) is 1, so no update of align(C) is needed. Similarly, since D is an empty base class, no update of dsize(C) is needed.

III. **Virtual Base Allocation**

Finally allocate any virtual base classes (except one selected as the primary base class in I-2a, if any) as we did non-virtual base classes in step II-1, in declaration order. Update sizeof(C) to max (sizeof(C), offset(D)+sizeof(D)). If non-empty, also update align(C) and dsize(C) as in II-1.

IV. **Finalization**

Round sizeof(C) up to a non-zero multiple of align(C).

Virtual Table Layout

General

A *virtual table* (*vtable*) is a table of information 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. There may be multiple vtables for a particular class, if it is used as a base class for other classes. However, the vtable pointers within all the objects (instances) of a particular most-derived class point to the

same set of vtables.

A vtable consists of a sequence of offsets, data pointers, and function pointers, as well as structures composed of such items. We will describe below the sequence of such items. Their offsets within the vtable are determined by that allocation sequence and the natural ABI size and alignment, just as a data struct would be. In particular:

- Offsets are of type `ptrdiff_t` unless otherwise stated.
- Data pointers have normal pointer size and alignment.
- Function pointers remain to be defined. One possibility is that they will be <function address, GP address> pairs, with pointer alignment.

In general, what we consider the address of a vtable (i.e. the address contained in objects pointing to a vtable) may not be the beginning of the vtable. We call it the *address point* of the vtable. The vtable may therefore contain components at either positive or negative offsets from its address point.

Components

Each vtable consists of the following components:

- *Base offsets* are used to access the virtual bases of an object, Such an 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. The values can be positive or negative.
- *vcall offsets* are used to perform pointer adjustment for overridden virtual functions. These entries are allocated when the class is used as a virtual base. They are referenced by the virtual functions to find the necessary adjustment from the base to the derived class, if any. These values may be positive or negative.
- The *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, as a `ptrdiff_t`. 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 *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.
- *Virtual 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.) In the case of shared library builds, a virtual function pointer entry contains a pair of components (each 64 bits in the 64-bit IA-64 ABI): the value of the target GP value and the actual function address. That is, rather than being a normal function pointer, which points to such a two-component descriptor, a virtual function pointer entry is the descriptor.
- *Secondary vtables* are copies of the vtables for base classes of the current class (copies in the sense that they have the same layout, though virtual function pointers may point to overriding functions).

The virtual pointer in the object points to the first virtual function pointer.

Virtual Table Order

A virtual table's components are laid out in the following order, analogous to the corresponding object layout.

1. If vcall offsets are required, they come first, with ordering as defined in categories 3 and 4 below.
2. If virtual base offsets are required, they come next, with ordering as defined in categories 3 and 4 below.
3. The offset to top field is next. It is present if the class has virtual functions. Question: Should we include the RTTI fields for classes with no virtual functions (only virtual bases), too?
4. The typeinfo pointer field is next. It is present if the class has virtual functions.
5. The vtable address point points here, i.e. this is the address of the vtable contained in an object's vptr.

6. Virtual function pointers come next, in order of declaration of the corresponding member function in the class. They appear both for newly introduced functions and overridden functions.
7. The secondary vtables are last. They are laid out in the same order used for the bases themselves in the object.

Virtual Table Construction

In this section, we describe how to construct the vtable for an class, given vtables for all of its base classes. To do so, we divide classes into several categories, based on their base class structure.

Category 0: Trivial

Structure:

- No virtual base classes.
- No virtual functions.

Such a class has no associated vtable, and its objects contain no vptr.

Category 1: Leaf

Structure:

- No inherited virtual functions.
- No virtual base classes.
- Declares virtual functions.

The vtable contains an RTTI field followed by virtual function pointers. There is one function pointer entry for each virtual function declared in the class.

Category 2: Non-Virtual Bases

Structure:

- Only non-virtual base classes.
- Inherits virtual functions.

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 overriding functions (or the corresponding adjustor secondary entry points).

For a base class `Base`, and a derived class `Derived` for which we are constructing this set of vtables, we shall refer to the vtable for `Base` as `Base-in-Derived`. The vptr of each base subobject of an object of the derived class will point to the corresponding base vtable in this set.

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.

Following 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.)

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: Virtual Bases Only

Structure:

- Only virtual base classes.
- Base classes are not empty or nearly empty.

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, except that the vtable for a virtual base A also includes a vcall offset entry for each virtual function represented in A's primary vtable and the secondary vtables from A's non-virtual bases. The vcall offset entries are allocated from the inside out, in the same order as the functions appear in A's 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 is 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 at increasing negative offsets from the address point of the vtable in the 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: Complex

Structure:

- None of the above, i.e. directly or indirectly inherits both virtual and non-virtual base classes, or at least one nearly empty virtual base class.

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 are placed closest to the vtable address point, i.e. at higher addresses, while entries for virtual bases only visible to this class are further from the vtable address point, i.e. at lower addresses.

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.

Issues:

- *A-6: Duplicate base structure representation.*
- *B-2: Will contain one entry per return type for covariant returns.*

Vtables During Object Construction (open issue C-4)

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.

Run-Time Type Information (RTTI)

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

. 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*".

Deliberations

The following conclusions were arrived at by the attending members of the C++ IA-64 ABI group:

- The exact layout for type_info objects is dependent on whether a 32-bit or 64-bit model is supported.
- Advantage should be taken of COMDAT sections and symbol preemption: two type_info pointers point to equivalent types if and only if the pointers are equal.
- A simple dynamic_cast algorithm that is efficient in the common case of base-to-most-derived cast case is preferrable over more sophisticated ideas that handle deep-base-to-in-between-derived casts more efficiently at a slight cost to the common case. Hence, the original scheme of providing a hash-table into the list of base classes (as

is done e.g. in the HP aC++ compiler) has been dropped.

- The GNU egcs development team has implemented an idea of this ABI group to accelerate dynamic_cast operations by a-posteriori checking a "likely outcome". The interface of std::__dynamic_cast therefore keeps the src2dst_offset hint.
- std::__extended_type_info is dropped.

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.) can be kept in the run-time support library. Specifically, this proposal is to place in the run-time support library type_info objects for the following types:

- void*, void const*; and
- X, X* and X const*, for every X in: bool, wchar_t, char, unsigned char, signed char, short, unsigned short, int, unsigned int, long, unsigned long, long long, unsigned long long, float, double, long double.

(Note that various other type_info objects for class types may reside in the run-time support library by virtue of the preceding rules; e.g., that of std::bad_alloc.)

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. This entry is located two words ahead of the location pointed to by the vptr (i.e., entry "-2").
3. Every vtable shall contain one entry pointing to an object derived from std::type_info. This entry is located at the word preceding the location pointed to by the vptr (i.e., entry "-1").

   std::type_info contains just two pointers:

   - its vptr
   - a pointer to a NTBS representing the name of the type

   The possible derived 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::__ptr_to_member_type_info
4. std::fundamental_type_info adds no fields to std::type_info;
5. std::__pointer_type_info adds two fields (in this order):
   - 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"). The bit 0x1 encodes the "const" qualifier; the bit 0x2 encodes "volatile".
6. std::__reference_type_info is similar to std::__pointer_type_info but describes references.
7. std::__array_type_info and std::__function_type_info do not add fields to std::type_info (these types are only produced by the typeid operator; they decay in other contexts). std::__enum_type_info does not add fields either.
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;
        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;
    };
```

(Ed. note: to avoid endianness confusion, the above should be defined in terms of a flags field and masks.)

9. The std::__ptr_to_member_type_info type adds two fields to std::type_info:
    - a pointer to a std::__class_type_info (e.g., the "A" in "int A::*")
    - a_pointer to a std::type_info corresponding to the member type (e.g., the "int*" in "int A::*")

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);
/* sub: source address to be adjusted; nonnull, and since the
 *      source object is polymoprhic, *(void**)sub is a vptr.
 * src: static type of the source object.
 * dst: destination type (the "T" in "dynamic_cast(v)").
 * src2dst_offset: a static hint about the adjustment needed
 *    on sub; since this adjustment cannot be 1, 2 or 3,
 *    those special values mean:
 *       1: no hint
 *       2: src is not a public base of dst
 *       3: src is a multiple public base type but never a
```

```
*          virtual base type
*    otherwise, the src type is a unique public nonvirtual
*    base type of dst at offset -src2dst_offset from the
*    origin of dst.
*/
```

The exception handler matching algorithm

Since the RTTI related exception handling routines are "personality specific", no interfaces need to be specified in this document (beyond the layout of the RTTI data).

External Names (a.k.a. Mangling)

<To be specified.>

Please send corrections to Jim Dehnert.

C++ ABI for IA-64: Exception Handling

Revised 9 September 1999

Revisions

[990909] Original version.

General

In what follows, we define the C++ exception handling ABI, at three levels:

- the base ABI, interfaces common to all languages and implementations;
- the C++ ABI, interfaces necessary for interoperability of C++ implementations; and
- the specification of a particular runtime implementation.

Definitions

The descriptions below make use of the following definitions:

Base ABI

The minimal level of specification 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.

It is intended that nothing in this section be specific to C++, though some parts are clearly intended to support C++ features.

Data Structures

Exception Handle

An exception handle is an address pointing to an exception object consisting of an exception control header followed by user exception information. The handle points to the user information, so the control information is accessed at a negative offset from it. The control information consists of:

```
    language-specific information
Unexpected_Handler      unexpectedHandler;
Terminate_Handler       terminateHandler;
Exception_Destructor    destructor;
```

The runtime may at any time call unexpectedHandler to report an unexpected exception; if NULL, it should call terminateHandler. The runtime may at any time call terminateHandler to terminate execution; if NULL, it should call exit. The runtime or the eventual user code handler may call destructor to destroy and if necessary deallocate the exception object; it may be NULL if no destruction is necessary.

Unwind State Handle

The stack unwind process maintains its current state via an opaque structure containing unwound machine state, accessible only via a procedural API, TBD.

Exception Handler Framework

The standard ABI exception handling / unwind process begins with a call to __eh_throw, described below. This call specifies an exception object, and an exception class (a __uint_64).

The runtime framework then starts a two-phase process:

- In the search phase, the framework repeatedly calls the personality routine, with the EH_SEARCH_PHASE API described below, first for the current PC and register state, and then unwinding a frame to a new PC at each step, until the personality routine reports either success (a handler found) or failure (no handler). It does not actually restore the unwound state, and the personality routine must access the state through the API. If failure is reported, it ... (calls terminateHandler?).
- If the search phase reports success, the framework restarts in the cleanup phase. Again, it repeatedly calls the personality routine, with the EH_CLEANUP_PHASE API described below, first for the current PC and register state, and then unwinding a frame to a new PC at each step, until the personality routine reports success. At that point, it restores the register state, and branches to the PC, which has been set by the personality routine to the landing pad address.

Throwing an Exception

Exceptions are initially thrown, after creating an exception object, by calling:

```
typedef enum {
    EH_THROW_INITIAL,
    EH_THROW_RETHROW,
    EH_THROW_RESUME
} EH_THROW_KIND;

void __eh_throw (
    EH_THROW_KIND kind,
    __uint_64   exception_class,
    void *      exception_object,
    ...
);
```

<To Be Specified>

Personality Routine

The personality routine has the following prototype:

```
typedef enum {
    EH_SEARCH_PHASE,    // Search phase call
    EH_CLEANUP_PHASE    // Cleanup phase call
```

```
} EH_PHASE;
typedef enum {
    EH_HANDLER,         // Handler landing pad found
    EH_PROCEED,         // Proceed to next frame
    EH_UNEXPECTED,      // Unexpected exception
    EH_TERMINATE,       // Terminate thread/program
    ...
} EH_RESULT;

EH_RESULT personality (
    int         version,
    EH_PHASE    phase,
    __uint_64   exception_class,
    void *      exception_object,
    void *      language_specific_data_area,
    Unwind_State *unwind_state
);
```

If called with phase==EH_SEARCH_PHASE, the personality routine may:

- Determine that control should be passed to user code, for any purpose including a real catch or cleanup and resume, in which case it returns EH_HANDLER. In this case, the framework starts over with the unwind process, now passing EH_CLEANUP_PHASE.
- Determine that the current exception is unexpected and processing cannot proceed, in which case it returns EH_UNEXPECTED. In this case, the framework ...
- Determine that the program must be terminated, in which case it returns EH_TERMINATED. In this case, the framework calls terminateHandler if non-NULL, or exit() otherwise, neither of which will return.
- Determine that nothing interesting happens in this frame, in which case it returns EH_PROCEED. In this case, the framework unwinds a frame and calls it again.

If called with phase==EH_CLEANUP_PHASE, the personality routine may:

- Determine that control should be passed to user code, for any purpose including a real catch or cleanup and resume, in which case it uses the API to set the PC and any registers which need to contain data, and returns EH_HANDLER. In this case, the framework restores the register state and transfers control to the PC in Unwind_State. This user code may either simply proceed with program execution, or may perform cleanup actions and rethrow/resume.
- Determine that the current exception is unexpected and processing cannot proceed, in which case it returns EH_UNEXPECTED. In this case, the framework ...
- Determine that the program must be terminated, in which case it returns EH_TERMINATED. In this case, the framework calls terminateHandler if non-NULL, or exit() otherwise, neither of which will return.
- Determine that nothing interesting happens in this frame, in which case it returns EH_PROCEED. In this case, the framework unwinds a frame and calls it again.

Unexpected Exception Handler

Define the prototype and behavior of the Unexpected_Handler. Delete the exception object? Rethrow?

Terminate Handler

Define the prototype and behavior of the Terminate_Handler. Delete the exception object or pass it to the handler? Perhaps pass all of the personality parameters to the handler so debug traceback is easy?

Unexpected Exception Handler

Define the prototype and behavior of the Destructor. Do we need a size in the standard exception object header?

Debugging

An idea: If we were to define a special value of exception_class to identify a debugging unwind, and put a debug entry in the exception object, the personality routines could support language-specific debug traceback.

C++ ABI

The second level of specification is the minimum required 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.

Common C++ Runtime Implementation

The third level is a specification sufficient to allow all compliant C++ 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. <*To be specified.*>

Please send corrections to Jim Dehnert.