Document number: N2271=07-0131
Date: 2007-04-27
Reply to: Paul Pedriana
Electronic Arts
ppedriana at ea.com

EASTL -- Electronic Arts Standard Template Library

Paul Pedriana
Electronic Arts
ppedriana at ea.com

Abstract

Gaming platforms and game designs place requirements on game software which differ from requirements of other platforms. Most significantly, game software requires large amounts of memory but has a limited amount to work with. Gaming software is also faced with other limitations such as weaker processor caches, weaker CPUs, and non-default memory alignment requirements. A result of this is that game software needs to be careful with its use of memory and the CPU. The C++ standard library's containers, iterators, and algorithms are potentially useful for a variety of game programming needs. However, weaknesses and omissions of the standard library prevent it from being ideal for high performance game software. Foremost among these weaknesses is the allocator model. An extended and partially redesigned replacement (EASTL) for the C++ standard library was implemented at Electronic Arts in order to resolve these weaknesses in a portable and consistent way. This paper describes game software development issues, perceived weaknesses of the current C++ standard, and the design of EASTL as a partial solution for these weaknesses.

Introduction

The purpose of this document is to explain the motivation and design of EASTL so that it may help the C++ community better understand the needs of game software development. This document is not a proposal, though some of EASTL's changes and extensions could form the basis for such discussions. The large majority of EASTL would be useful to any kind of C++ software development.

This document describes an STL implementation (EASTL) developed within Electronic Arts as an alternative to and extension of the STL defined by the C++ standard library. By STL, we mean the container, iterator, and algorithm components of the C++ standard library, hereafter referred to as std STL (with *std* referring to the std namespace, whereas the S in STL refers to standard C++). By C++ standard, we mean ISO 14882 (1998) and the 2003 update. The large majority of the design of std STL is excellent and achieves its intended purpose. However, some aspects of it make it hard to use and other aspects prevent it from performing as well as it could. Among game developers the most fundamental weakness is the std allocator design, and it is this weakness that was the largest contributing factor to the creation of EASTL. Secondarily was the lack of std STL containers designed to be memory-friendly. There are additional reasons that will be discussed below.

We hope that those reading this document have an open mind to the idea that std STL may not be ideal for all purposes. Before this document was written, sketches of it were shown to some outside of the game development industry. In some cases we found that there was an initial reaction to dismiss an alternative STL and assume that the somebody must be misunderstanding or misusing the STL. But upon explaining game development and high performance software issues and comparing these to std STL's design and implementation by current vendors,

people usually reduce their skepticism. Indeed we have found that those have the most extensive and deep STL experience have been those most enthusiastic about EASTL. We nevertheless have a great respect for the C++ standard library and the great work that has gone into its design and implementation, especially after having gone through the long and difficult process of implementing it.

This document is divided into the following sections. The first section summarizes the motivation for the creation of EASTL and its design; the subsequent sections flow from this.

Throughout this document there are references to the Appendix. The Appendix contains supplementary material which provides more detail about some item of discussion. This material is placed there in order to avoid getting in the way of the primary text, as the material is a bit verbose and is sometimes tangential to the discussion of EASTL. It was nevertheless felt to be important that the Appendix exist in order to provide a better understanding of practical game development issues.

Motivation for EASTL

The following is a listing of some of the reasons why std STL and its current implementations are not currently ideal for game development. There are additional reasons, but the list here should hopefully convey to you some sense of the situation. Each of the items listed below deserves a document of its own, as a single sentence alone cannot fully convey the nature or significance of these items. Some of the items refer to the STL design, whereas some of the items refer to existing STL implementations. It would be best if these were discussed independently, but to many users this distinction is often of little practical significance because they have little choice but to use the standard library that comes with their compiler.

- std STL allocators are painful to work with and lead to code bloat and sub-optimal performance. This topic is addressed separately within this document.
- Useful STL extensions (e.g. slist, hash_map, shared_ptr) found in some std STL implementations are not portable because they don't exist in other versions of STL or are inconsistent between STL versions (though they are present in the current C++09 draft).
- The STL lacks functionality that game programmers find useful (e.g. intrusive containers) and which could be best optimized in a portable STL environment. See Appendix item 16.
- Existing std STL implementations use deep function calls. This results in low performance with compilers that are weak at inlining, as is the currently prevalent open-source C++ compiler. See Appendix item 15 and Appendix item 10.
- Existing STL implementations are hard to debug. For example, you typically cannot browse the contents of a std::list container with a debugger due to std::list's usage of void pointers. On the other hand, EASTL allows you to view lists without incurring a performance or memory cost. See Appendix item 2.
- The STL doesn't explicitly support alignment requirements of contained objects, yet non-default alignment requirements are common in game development. A std STL allocator has no way of knowing the alignment requirements of the objects it is being asked to allocate, aside from compiler extensions. Granted, this is part of the larger problem of the C++ language providing minimal support for alignment requirements. Alignment support is proposed for C++09.

- STL containers won't let you insert an entry into a container without supplying an entry to copy from. This can be inefficient in the case of elements that are expensive to construct.
- The STL implementations that are provided by compiler vendors for the most popular PC and console (box connected to TV) gaming platforms have performance problems. EASTL generally outperforms all existing STL implementations; it does so partly due to algorithmic improvements but mostly due to practical improvements that take into account compiler and hardware behavior. See Appendix item 20.
- Existing STL implementations are hard to debug/trace, as some STL implementations use cryptic variable names and unusual data structures and have no code documentation. See Appendix item 2.
- STL containers have private implementations that don't allow you to work with their data structures in a portable way, yet sometimes this is an important thing to be able to do (e.g. node pools). See Appendix item 22.
- Many current versions of std STL allocate memory in empty versions of at least some of their containers. This is not ideal and prevents optimizations such as container memory resets that can significantly increase performance in some situations. An empty container should allocate no memory.
- All current std STL algorithm implementations fail to support the use of predicate references, which results in inefficient hacks to work around the problem. See Appendix item 3.
- The STL puts an emphasis on correctness before practicality and performance. This is an understandable policy but in some cases (particularly std::allocator) it gets in the way of usability and optimized performance.

Differences between std STL and EASTL

First, EASTL provides a set of containers, iterators, and algorithms that are identical in interface and behavior to std STL versions with one exception: allocators. EASTL has a different allocator specification which is simpler, more efficient, more flexible, and easier to use than std::allocator. Both std::allocator and eastl::allocator are described below. EASTL follows the defect reports and TR1 as well. EASTL additionally provides and uses some of the TR1 functionality as well, including most significantly the smart pointers, type traits, and hashing containers.

Second, EASTL provides extension functionality to the above containers, iterators, and algorithms. An example of this is the push_back(void), set_capacity(size), and validate() functions added to eastl::vector. The majority of these extensions are driven by the need for higher performance (push_back(void)), higher clarity (set_capacity), or higher debuggability (validate()). There are about 30 such extensions to the various entities in the library. These are described in detail below.

Third, EASTL provides additional containers and algorithms that don't correspond to std STL. These include intrusive_list, vector_map, fixed_hash_set, slist, ring_buffer, radix_sort, has_trivial_relocate, and others. These are described in detail below.

There are additional differences not related to the functional specification. They include a programming philosophy that emphasizes readability, consistency, and optimal performance on limited hardware. See Appendix item 23.

EASTL functionality summary

EASTL coverage of std STL (not including TR1)

EASTL covers the following parts of the C++ standard library.

Entity	Comments
list	EASTL has a small number of additional requirements for containers beyond those

vector deque basic_string set multiset map multimap bitset	std STL containers. An example of this is the requirement that a newly constructed or reset container allocates no memory. These are discussed in the next section. EASTL has additional functionality in most of the containers in order to obtain higher performance. These are discussed after the next section. EASTL uses a different allocator model, as described below. Otherwise, the EASTL versions logically behave the same as std STL versions. The EASTL implementations of the containers listed on the left are not modifications of any existing std STL implementation but are largely complete reimplementations.
queue stack priority_queue	
iterator	EASTL defines an additional iterator_tag in addition to bidirectional_iterator, random_access_iterator, etc: contiguous_iterator_tag. A contiguous iterator has all the properties of a random_access_iterator but additionally specifies that the range is contiguous in memory as with an array.
memory numeric algorithm utility functional	

EASTL augmentations/amendments to std STL

Here we list extension functionality that EASTL provides over std STL along with a rationale for each item. In some cases additional information can be found in the appendix and will be noted. All of these changes were prompted by real issues that were encountered in the use of std STL and EASTL. None were the result of speculation or "premature optimization."

Extension	**Rationale**
All containers have a node_type typedef and kNodeAlignment constant.	Much as containers define types such as value_type and key_type, EASTL defines node_type for all containers. node_type is the storage type that the container uses, as opposed to the user-specified contained type. It is the type that the container allocates. Explicitly defining this type allows users to implement allocators such that the allocated type is known at compile-time. Additionally, kNodeAlignment is a size_t constant which defines the alignment of the node_type. For containers such as hash tables which have multiple allocated types, additional node types are explicitly defined for the class.
Similarly, all containers understand and respect object alignment requirements.	As discussed elsewhere in this document, game development platforms explicitly or implicitly require the use of non-default alignments. A typical example is VMX vector types, which explicitly require 16 byte alignment because they are an array of 4 floats which are processed in parallel in a single 128 bit register.
All containers have	std STL lets you set the allocator for a class only during class construction.

a get_allocator and set_allocator function, which return the actual allocator instead of a copy of it.	Additionally, std STL doesn't let you access its allocator; you can only get a copy of it. This and other weaknesses of std STL allocators are discussed elsewhere in this document in more detail.
All containers guarantee that there is no memory allocation upon being newly empty-constructed.	Since EASTL containers have a set_allocator function and allow the user to set the container's allocator after the container's construction, this requirement naturally follows. It's also simply more efficient to avoid memory allocation whenever possible. Lastly, there is the game development policy that memory should never be allocated "behind a user's back" or when the user doesn't expect it.
All containers have a reset function, which unilaterally resets the container to an initialized (and unallocated) state, avoiding container teardown and reallocation.	A common high-performance technique is to create a temporary hash table with a fixed-size memory buffer, do processing with it, and then "nuke" the hash table when done instead of going through the process of clearing and deallocating its nodes. EASTL explicitly supports this programming pattern via the reset function. The supplied allocator generally must cooperate with this, lest memory leaks occur. However, it can work safely only with value types that have a trivial destructor.

All containers have explicit validate and validate_iterator functions.	EASTL provides an option to do implicit automatic iterator and container validation, but full validation (which can be potentially extensive) has too much of a performance cost to execute implicitly, even in a debug build. So EASTL provides these explicit functions which can be called by the user at the appropriate time and in optimized builds as well as debug builds.
	
	`bool validate() const;`
	`iterator_status_flag validate_iterator(const_iterator i) const;`
	
	See the EASTL container section for more.

Containers should be viewable in a basic debugger to the extent possible.	A common complaint by users is that std STL implementations use void pointers for linked list items and makes the container overly difficult to view while debugging. EASTL rectifies this with no overhead. See Appendix item 2.
vector has a data() function which acts similar to basic_string::data()	data() is not the same thing as &v[0], as the latter will result in an invalid dereference assertion failure when v is empty. This is also in the current C++09 draft.
vector<bool> is an array of bool. It is not a bit vector.	The user can use eastl::bitvector if the user wants a bit vector, and vector<bool> is deprecated in the current C++09 draft.
vector and basic_string have set_capacity(size), which sets the capacity to exactly	People have been recommending the "swap trick" for years, but this is an obfuscated and easy-to-forget workaround in place of what should be built-in functionality. If a workaround is well-known enough that there are hundreds of pages on the Internet devoted to it and it has its own nickname, it probably

what the user specifies.	should be built into the standard library. set_capacity is the same thing as resize_capacity which has been suggested by some.
vector::iterator and basic_string:: iterator are pointers.	Primarily this allows for easier debugging by the user and easier optimization by the compiler. A random access iterator is not the same thing as a pointer, which refers to contiguous memory. Even in the presence of type_traits optimizations there are things a compiler can do with known-contiguous pointers that it cannot do with random access iterators. See Appendix item 25. The downside to vector and string iterators as pointers is that some types of automated runtime validation can't be done. So far it doesn't seem to have been greatly missed.
All containers have a push_back(void) and/or similarly useful functions.	Existing std STL implementations implement insertion operations by copying from an element. For example, resize(size() + 1) creates a throw-away temporary object. There is no way in existing std STL implementations to add an element to a container without implicitly or explicitly providing one to copy from (aside from some existing POD optimizations). For expensive-to-construct objects this creates a potentially serious performance problem. A typical suggested workaround to this problem with std STL is to store pointers instead of objects; however, such external memory allocation is undesirable for reasons described elsewhere in this document. list and deque have both push_back(void) and push_front(void). slist has push_front(void). map, multimap, hash_map, and hash_multimap have insert(key). insert(key) is the same thing as the lazy_insert function that has been suggested by some. Other related proposals have been put forward.
There are sprintf, append_sprintf, trim, compare_i, make_lower, and make_upper functions for the basic_string class.	These are practical useful functions. Game developers prefer to use sprintf instead of stream and strstream functionality where possible because the latter is unacceptably slow in practice. The obscurity and lack of readability of stream operations adds to their unpopularity. The compare_i, make_lower, and make_upper functions are by definition not locale-savvy. They are nevertheless useful practical functions due to the large amount of non-localized string usage in utilities and applications. Not all strings are destined to be viewed by the end-user.

deque allows the user to specify the node size as a template parameter.	std STL deque guesses the node size for the user; the user cannot control it. This can result in undesirably large or undesirably small memory allocations that the user cannot control. Making the node size available like this does represent a leaking of the deque abstraction.
	
	`template <typename T, typename Allocator, size_t kNodeSize>`
	`class deque { };`

Associative containers (e.g.	If you have a map<string, int> and you want to look up an entry, std STL

map, hash_map) have the find_as function, which allows fast key lookups for expensive key types.	requires you to supply a string object to the map::find function. This is overly expensive for the common case whereby you have a string literal which you want to look up, because passing such a literal to the map::find function silently creates a temporary string object and allocates memory.
	
	EASTL solves this problem by defining the following member template function, which allows you to, for example, look up a string object via a lightweight string literal:
	
	`template <typename U, typename BinaryPredicate>`
	`iterator find_as(const U& u, BinaryPredicate predicate);`
	
	This is similar to the lazy_find that has been suggested by some.

An additional iterator_tag beyond random_access _iterator_tag is defined: contiguous_ iterator_tag.	random_access_iterator_tag defines a range which has random access, but isn't necessarily contiguous. This may lead to missed optimization opportunities. type_traits can be used to work around some of these opportunities but not in all cases and not in as explicit a way. *Need to provide an example here.*

eastl::pair has a single argument constructor. It accepts the first element as a constructor argument and default-constructs the second argument.
 	
`pair();`
`pair(const T1& x);`
`pair(const T1& x,`
`const T2& y);`
	The usefulness comes about because the map::value_type is pair<const key_type, mapped_type>. As a result, inserting a value into a map requires the user to create a temporary pair whose contents must be set at its construction. This turns out to be inefficient when the mapped_type is expensive to construct or copy, as the pair constructor forces you to provide a mapped_type to copy from. It's more efficient to write code like this:
	
	`map<int, expensive>::value_type value(37);`
	`value.second.blah = blah;`
	`m.insert(value);`

bitset uses uint32_t instead of unsigned long.	unsigned long is not a portable type. On game development platforms it can be any of 32, 64 or 128 bits. On platforms where it is 64 or 128 bits, it can be inefficient to work with these types, as it is on the PlayStation 2 platform. Types such as short, int, long, and long long cannot be used in a portable way and thus EASTL and many game programming standards disallow their usage in API interfaces. Sized types such as int16_t, int32_t, etc. are used instead.
bitset has neither string nor iostream functionality.	string and iostream are unrelated to bitset and embedding knowledge about them into bitset bloats the bitset implementation and dependency graph in a way that yields marginal and limited benefits.
bitset has find_first, find_next functions.	These are of course more efficient for the user than manually checking every bit in turn.

Three additional heap algorithms are provided: remove_heap, change_heap, is_heap. Similarly, priority_queue has change and remove member functions.	These are useful practical additions. remove_heap and change_heap provide functionality that can't be efficiently implemented externally.
eastl::list, slist, intrusive_list, and intrusive_slist don't cache the list size. Thus the list::size is O(n). However, there is a configuration option to cache it an make it O(1).	The C++ standard specifies that list::size is O(1), and some STL implementations cache the list size and implement it as O(1). The SGI STL branch of STL doesn't cache the list size and list::size is thus O(n). EASTL can be configured to work either way, but defaults to O(n). The rationale for this is that it adds extra memory and processing cost to the list container, whereas users of linked lists often don't care about the size of the list. The user can always maintain a list size cache on their own. A future revision of eastl::list will likely make this a template policy option.

EASTL algorithms respect user-supplied template argument types and are guaranteed to do so. No existing std STL currently does so.	The following code does not work as one might expect or hope with any current std STL implementation:
	
	`struct ExpensiveToConstructCompare{ };`
	`ExpensiveToConstructCompare compare;`
	`std::sort<int*, ExpensiveToConstructCompare&>(begin, end, compare);`
	
	The problem is that std STL implementations ignore the user-specified compare reference type and convert it to a value type and proceed to make (possibly very many) copies of it. This can be prohibitively expensive. The conventional workaround for this problem is to create a proxy compare class that references the real compare object. Why not simply have the algorithms obey the user's request and avoid such workarounds and their resulting overhead? It's not clear if the C++ standard requires that the user's request be respected, but it would be nice if it did. The currently proposed std::ref adapater could help this situation with existing std STL implementations, but it would be nice if they didn't require this.

EASTL type traits can be explicitly user-controlled.	Some type traits cannot be discerned by compilers, and in some cases you may want to override the compiler's automatic view of a type trait. EASTL provides a supported mechanism for explicitly setting a type_trait: EASTL_DECLARE_*. An example of this is EASTL_DECLARE_TRIVIAL_RELOCATE(x).
EASTL's shared_ptr/weak_ptr allow the user to specify an allocator	As described in the game software issues section, global new usage is often verboten in game software development, at least for console platforms. Thus

instead of implicitly using global new.	any library facility which uses global operator new or any memory allocation that cannot be controlled by the user is unacceptable.
Allocators are named. All containers have a default name in case one isn't supplied by the user.	Naming allocators and allocations is a common and useful practice for software development where all memory must be accounted for. Tracking allocations by file/line doesn't work well with libraries - especially fundamental libraries such as STL - and EASTL builds it into the API, though allocators can ignore it when appropriate. This is a case of a feature that might not be popular if it was in std STL, but since EASTL is developed for use by a game development company it works well.
queue, priority_queue, and stack have a get_container function.	It is sometimes simply practical to have this. A typical case is one whereby the creator of the queue is different from the user of the queue. It is also useful for diagnostics.
EASTL doesn't use char and wchar_t but instead uses char8_t, char16_t, and char32_t.	wchar_t isn't portable. See Appendix item 13.
EASTL defines eastl_size_t and eastl_ssize_t, and container size_type is typedef'd to eastl_size_t.	These default to size_t and signed size_t respectively but can be configured to be uint32_t and int32_t on 64 bit platforms, as a 64 bit size_t wastes space in practice in most cases.

Changes that aren't related to API specifications:

hash and tree containers work by inheritance instead of by composition in order to reduce a layer of indirection.	It is common for std STL implementations to implement std::set by having it own a member rbtree and route most of its functionality through the member. This has some theoretical advantages but results in asking the compiler to deal with a layer of indirection which it may or may not optimize away.

EASTL supplies per-platform optimizations in some cases.	For example, the min function specialized for float can be implemented in a way that avoids branching.
	
	`float min(float a, float b)`
	`{`
	`  float result, test(a - b);`
	`  __asm__ ("fsel %0, %1, %2, %3" :`
	`  "=f" (result) :`
	`  "f" (test), "f" (b), "f" (a));`
	`  return result;`
	`}`

	Such optimizations are implementation details, but are described here because they provide a practical performance benefit to the performance-conscious user. It would be nice if std STL implementations provided such things, though Metrowerks has been known to do so in some cases.
Instead of just <algorithm>, EASTL has <algorithm.h>, <sort.h>, <algoset.h>, and <heap.h>.	std STL <algorithm> is a large file that is slow to compile and creates large object files for compilers that use the Borland template model (most compilers) as opposed to the Cfront model.
EASTL containers avoid function calls to the extent possible, even if such calls might be inlined by the compiler.	This makes it easier for the compiler to optimize and easier for the user to trace. Debug game builds that heavily use std STL with all its function calls can be unacceptably slow. See Appendix item 9 and Appendix item 10.
EASTL is argument-dependent-lookup safe and guaranteed to be so.	This allows you to safely call STL functions with code from other namespaces. See Appendix item 11.
EASTL compiles without warning on the highest warning levels available by the compiler.	This might seem like an obvious idea, but the most common commercial STL implementation does not follow it. Nearly all shared libraries and most games within EA are developed with the highest possible warning levels, and many teams set warnings to be errors. As it currently stands, usage of the C++ standard library and STL provided by the aforementioned vendor must be wrapped in warning disabling clauses wherever they are used. This results in more messy code and more fragile builds.
Exception handling can be disabled in EASTL by explicitly supported configuration directives.	Almost all game development is done with exception handling disabled. The discussion of this policy is outside the scope of this paragraph, but is handled in Appendix item 19. It useful if users can explicitly disable exception handling in the libraries that they use, independently of how the compiler is configured for exception handling.
Header files are explicitly in an EASTL header directory and have a .h suffix. Thus we have #include <EASTL/list.h> instead of #include <list>.	Putting header files in an explicit directory gives them a "namespace" of sorts. This goes a long way towards avoiding header conflicts between shared libraries and is the standard design within Electronic Arts. Suffix-less header names (e.g. <list>) don't work well in practice as well as suffixed names in user and development environments. This is a practical reality, and users prefer suffixed file names anyway.
EASTL is implemented in a highly readable way that	All existing STL implementations with the possible exception of Metrowerks are hard to read. They have virtually no code documentation and use variable and function names that are cryptic

allows non-experts to follow it.	and/or use unusual formatting. This has been a surprisingly negative point in EA team evaluations of STL for their use.

EASTL coverage of TR1 (C++ library extensions)

EASTL covers the parts of TR1 that relate to containers, algorithms, and iterators. Most significantly, it does not cover random number generation and regular expressions.

Entity	Comments
array	
type_traits	EASTL provides many of the type_traits defined in TR1 to the extent possible, and includes some extension traits and functionality as well. EASTL algorithms and containers make extensive use of these type traits.
unordered_set unordered_multiset unordered_map unordered_multimap	EASTL names these hash_set, hash_multiset, hash_map, and hash_multimap. A primary reason they were renamed to unordered_* in TR1 was to avoid collision with existing std STL extensions. This is not an issue for EASTL because it has its own namespace: eastl. EASTL provides some extension functionality to the TR1 proposal, the most significant of which is the find_as function.
shared_ptr weak_ptr	These are fundamentally the same as TR1 with the exception that they allow the user to provide an allocator instead of using global new, which is often verboten in game library development. Also, the use of virtual functions is avoided as well, given the potential cache miss they may cause. See Appendix item 19.

EASTL additional functionality (not found in std STL or TR1)

The following lists EASTL containers, algorithms, functors, and meta-templates that are not found in either std STL nor in TR1. The non-allocating containers and smart pointers below are viewed by many as the most important part of EASTL and some users use these more often than regular containers or even exclusively.

Entity	Comments

fixed_list
fixed_slist
fixed_vector
fixed_string
fixed_set
fixed_multiset
fixed_map
fixed_multimap
fixed_hash_set
fixed_hash_multiset
fixed_hash_map
fixed_hash_multimap
	Fixed containers are fixed-size containers with their memory stored right within the container itself. Fixed containers allocate no dynamic memory and their memory tends to be cache-friendly due to its contiguity and proximity to the container's housekeeping data. The user declares the max container size as a template parameter, and can also specify that if the container overflows that an auxiliary dynamic allocator is used. The overflow allocation feature is best used for pathological cases or during development when the container's fixed size is being tuned. All fixed containers have high-water mark tracking to assist in size tuning.
	
	The following is the template declaration for fixed_vector:
	
	`template <typename T, size_t nodeCount, bool enableOverflow = true,`
	`typename Allocator overflowAllocator = EASTLAllocator>`
	`class fixed_vector`
	`{`
	`...`
	`};`

fixed_substring	This is a string which is a view on an arbitrary span of characters. Thus if you have a paragraph of text but want to treat a single sentence of it as a (const) string, fixed_substring can be used. This allows for efficient const string operations without allocating memory and copying string data. A fixed_substring can be resized, upon which it will then allocate memory and copy characters.
vector_set vector_multiset vector_map vector_multimap	These are the same thing as sorted vectors as described in Effective STL. The underlying container is not limited to being a vector but can be any random access container, such as deque or array. Recall that the characteristic of these containers is that they provide improved memory efficiency and locality at the cost of dynamic container modification efficiency.
intrusive_list intrusive_slist intrusive_set intrusive_multiset intrusive_map intrusive_multimap intrusive_hash_set intrusive_hash_multiset intrusive_hash_map intrusive_hash_multimap	Intrusive containers are containers whereby the user provides the nodes and thus no memory is allocated and cache behavior is improved during container manipulations. Another benefit is that elements can be removed from containers without referencing the container (i.e. without calling container member functions). This is useful for when a class hands out element pointers to clients which the client will eventually hand back. Another benefit of intrusive containers is that is they doesn't require elements to be copyable, and sometimes element copying is not possible. EASTL intrusive containers allow elements to safely reside in multiple unrelated containers simultaneously. See Appendix item 16 for the intrusive_list interface.
slist	slist is a singly-linked list that is much like the slist extension found in the SGI STL branch of STL implementations.

ring_buffer	This implements a constant or variable capacity ring buffer which is templated on a user-supplied bidirectional container. Typically you would use it with a list or a deque, but a constant-capacity ring_buffer might use vector or array instead.
	
	`template <typename T, typename Container = vector<T> >`
	`class ring_buffer { };`

linked_ptr linked_array intrusive_ptr safe_ptr	These are smart pointers which have similar characteristics to shared_ptr, but don't allocate memory. safe_ptr is an explicitly standalone weak pointer which doesn't allocate memory. linked_ptr uses a linked list to store references and thus also doesn't allocate memory. intrusive_ptr uses an object-supplied addref/release function to manage object lifetime.
shared_array scoped_array	These are array versions of shared_ptr and scoped_ptr. The conventional explanation for why the TR1 doesn't provide such array versions is that vector already supplies this. That's a fine argument, though vector and shared_array are not identical.
binary_search_i find_first_not_of find_last_of	These are additional algorithms.

find_last_not_of identical change_heap remove_heap is_heap median	binary_search_i is an alternative to binary_search which returns an iterator instead of bool. It turns out that users often want the result of the binary search and not the just status. This is the same as the binary_find function suggested by some. identical is an algorithm which is like the equal algorithm except it doesn't assume/require that the input ranges are of equal length. identical efficiently compares ranges for both length equality and element equality.

is_sorted
radix_sort
bucket_sort
shell_sort
insertion_sort
merge_sort
merge_sort_buffer
comb_sort
bubble_sort
	These are additional sort algorithms.
	
	radix_sort is implemented via the following interface:
	
	`template <typename RandomAccessIterator, typename ExtractKey, typename IntegerType>`
	`void radix_sort(RandomAccessIterator first, RandomAccessIterator last,`
	`RandomAccessIterator buffer, ExtractKey extractKey, IntegerType);`
	
	merge_sort_buffer is a merge_sort whereby the user supplies the scratch buffer instead of relying on dynamic allocation to provide it.
	
	bubble_sort has curiously been shown in practice to be a good sort for very small element counts (<5), and so it is retained for both practical and instructional purposes, though insertion_sort also works well for small sizes.
	
	comb_sort is a fast sort which is almost as fast as the (introspective) sort algorithm, but has the benefit of using much less code and thus may be more friendly with the instruction cache.

equal_to_2 not_equal_to_2 less_2 str_equal_to str_equal_to_i	These are additions to the functional module. equal_to_2 compares two different types T and U for equality. less_2 is similar. These are useful for things such as the find_as extension. str_equal_to compares two C strings represented by character pointers. It is like strcmp.
is_aligned has_trivial_relocate EASTL_DECLARE_*	EASTL has a couple additional type traits in addition to the TR1 set. Of note is the has_trivial_relocate trait, which defines whether a class instance can be moved (memcpy'd) to another location in memory safely. This is useful for classes that are expensive to copy construct and assign but are easy to memcpy. The classic example is a reference count. A vector of reference counts is expensive to manipulate because it triggers what we call "ref count storms" or "ref count thrashing" which slows down execution and makes some types of debugging very difficult. However, a reference count is merely an integer and so can memcpy'd to its new location by the vector. Note that a relocatable object is not the same thing as a POD. A POD may or may not be relocatable, and a non-POD may or may not be relocatable. relocation is very much related to move semantics proposed for the C++ language, but is called "relocate" in order to avoid confusion with evolving

	definition of move semantics. EASTL has a formal mechanism which allows the user to explicitly ascribe a type trait to a class. This can be applied to most of the type traits but is particularly targeted at the type traits which the compiler cannot determine for itself. For example, the EASTL_DECLARE_HAS_TRIVIAL_RELOCATE(T) tells the compiler that type T should have the has_trivial_relocate type trait.
uninitialized_copy_ptr uninitialized_fill_ptr uninitialized_fill_n_ptr uninitialized_copy_fill uninitialized_fill_copy uninitialized_copy_copy	These are the same as equivalents found in libstdc++ and its predecessors. These do operations on uninitialized memory such a filling to it or copying to it and thus initializing it as it goes.
generic_iterator	EASTL defines a generic iterator which can be wrapped around a pointer to make it act like a contiguous iterator.
use_self use_first use_second make_pair_ref	These are utility templates for the pair template. make_pair_ref is a version of make_pair that explicitly uses object references instead of values in order to prevent potentially expensive copy operations.
compressed_pair call_traits	These are as per existing similar extensions found elsewhere. compressed_pair is useful for implementing the "empty base class" optimization.

Game software issues

The discussion here involves gaming platforms currently developed for by EA. These include primarily console, hand-held, and desktop platforms. Virtually all game development within EA and other companies is done in C++; this isn't likely to change for many years. A number of characteristics distinguish game software requirements from desktop productivity software requirements. These include:

- Game software always maximizes the hardware and is subject to Nathan's Laws.
- Game source and binaries tend to be rather large. See Appendix item 5.
- Game applications work with large amounts of application data. See Appendix item 5.
- Console-based games run off of DVD drives, which are much slower than hard drives.
- Game applications must execute very smoothly; there can't be unplanned execution pauses.
- Non-desktop platforms don't have paged memory. If the application exhausts memory, it dies.
- The lack of paged memory means that memory fragmentation can kill an application.
- Non-desktop platforms have a fixed amount of memory, and it is smaller than desktop platforms. See Appendix item 27.
- Non-desktop platforms have special kinds of memory, such as "physical memory," "non-local memory," "non-cacheable memory," etc.
- Non-desktop platforms have processors and memory caches are less forgiving than those on desktops. See Appendix item 28.
- Debug builds of game software need to be reasonably fast. See Appendix item 9.
- Game platforms sometimes require the use of non-default memory alignment.

The above conditions lead to the following results:

- No matter how powerful any game computer ever gets, it will never have any free memory or CPU cycles.
- Game developers are very concerned about software performance and software development practices.
- Game software often doesn't use conventional synchronous disk IO such as <stdio.h> or <fstream> but uses asynchronous IO.
- Game applications cannot leak memory. If an application leaks even a small amount of memory, it eventually dies.
- Every byte of allocated memory must be accounted for and trackable. This is partly to assist in leak detection but is also to enforce budgeting.
- Game software rarely uses system-provided heaps but uses custom heaps instead. See Appendix item 26.
- A lot of effort is expended in reducing memory fragmentation.
- A lot of effort is expended in creating memory analysis tools and debugging heaps.
- A lot of effort is expended in improving source and data build times.
- Application code and libraries cannot be very slow in debug builds. See Appendix item 9.
- Memory allocation of any type is avoided to the extent possible.
- Operator new overrides (class and global) are the rule and not the exception.
- Use of built-in global operator new is verboten, at least with shareable libraries.
- Any memory a library allocates must be controllable by the user.
- Game software must be savvy to non-default memory alignment requirements.
- Memory pools are sometimes used in order to avoid fragmentation, even though they necessarily waste some memory themselves. See Appendix item 26.
- Branching (if/else/while/for/do) is avoided to the extent possible, especially mispredicted branches. See Appendix item 28.
- Virtual functions are avoided to the extent possible, especially in bottleneck code. See Appendix item 19.
- Exception handling is usually disabled. See Appendix item 17.
- RTTI is usually disabled or at least unused in shipping code.

Many of the above items are related to memory management and performance. As a result a lot of effort is put into optimizing the use of memory. Some games have been known to run with only a few KiB of system memory free. Some games run with no memory free at all and install an out-of-memory callback to free memory from elsewhere to satisfy the current request. See Appendix item 18. The memory fragmentation is not solved, as many of the analyses that have been used to measure memory fragmentation have largely applied to desktop software memory usage patterns, requirements, and environments. Game development has further memory constraints which make problems harder to solve. EA would like to welcome researchers to work with us on some of these problems, as they represent difficult problems that don't appear to be going away.

We also have the following practical considerations regarding C++ game programmers:

- C++ isn't taught much any more in college. It's hard enough finding people who know C++, and harder finding people who understand templates of the kind you find in STL.
- An increasing number of game developers are young and generic programming is foreign to them. Readers of this paper may have no trouble navigating a std STL header file, but this can be a daunting task for a less experienced programmer. An STL implementation that is very clear is worth more than experts may intially think.
- Game developers (all developers, really) need to be able to examine and trace STL containers and algorithms. It's not a matter of debugging the containers themselves (which are already debugged) but a matter of debugging the user's use of the containers.
- C++ templates are disliked by some because they are "tricky" and have "gotchas." You shouldn't have to be a language lawyer to use a programming language.

std::allocator

Section 20.1.5 of the C++ standard describes the requirements for an allocator, and the result looks more or less like below. The requirement that the allocator be a templated class is not explicitly stated in the standard, though the requirements for rebind make it virtually mandatory that allocators be templated. Items of interest to this discussion are colored in blue.

```
template <typename T>
class allocator
{
public:
    typedef size_t     size_type;
    typedef ptrdiff_t  difference_type;
    typedef T*         pointer;
    typedef const T*   const_pointer;
    typedef T&         reference;
    typedef const T&   const_reference;
    typedef T          value_type;

    allocator();
    allocator(const allocator<T>&);
   ~allocator();

    template <class U> allocator(const allocator<U>&);
    template <class U> struct rebind { typedef allocator<U> other; };

    T*        address(T&) const;
    const T*  address(const T&) const;
    T*        allocate(size_type, const void* = NULL);
    void      deallocate(T*, size_type);
    void      construct(T*, const T&);
    void      destroy(T*);
    size_type max_size() const;
};

bool operator ==(const allocator<T>&, const allocator<T>&);
bool operator !=(const allocator<T>&, const allocator<T>&);
```

Unfortunately, the std allocator design and its use by std containers make it hard to work with and leads to suboptimal performance and code bloat. These are some serious charges, but in the realm of high performance game development they are real and significant. The following is a list of some of the issues of std allocators and std containers' use of std allocators:

- As described in the Halpern proposal, std allocators are class-based and not instance-based. This results in some awkward attempts to make them act as if instance-based.
- Allocators are virtually required to be templates, and rebind is required to be a member template. Unless the compiler is very good at inlining (and some compilers are not -- see Appendix item 15), this can result in an explosion of templates, code, and performance problems.
- Allocators are rebound by containers to one or more additional types -- types which the user cannot know in any portable way. Thus the author of an allocator cannot know what it will actually be asked to allocate and construct. It is ironic that user-supplied allocators to STL containers in practice don't allocate objects of the user-specified type. It is unfortunate that there is no portable way for the container user or allocator implementor to know what the allocator will be asked to allocator or what it will be rebound to. This somewhat defeats the purpose of the allocator being templated on some type.
- Allocators don't understand alignment requirements. Objects allocated by a std allocator are assumed to be of default alignment (usually equal to sizeof double) or it is assumed that the implementor of the allocator knows what the alignment is. Neither of these assumptions are satisfactory, as the former can be simply untrue and the latter can be impractical or impossible.
- Allocators are required to construct and destroy objects as well as allocate them. This forces the mixing of two separate concepts: allocation and construction.
- Containers require you to set an allocator at container construction time; you cannot set it later.

- Containers do not let you access their allocators; they only give you copies of their allocators.
- Current STL implementations such as libstdc++ (up to libstdc++ v4.1, apparently rectified with v4.2) copy allocators and create temporary allocators during empty construction and copy construction, which can result in performance problems in the case of expensive allocators. The user is thus steered towards making allocators lightweight classes which are references to the actual allocators in use, which results in unwanted indirection overhead and extra code.

eastl::allocator

EASTL allocators bear a resemblance to std allocators but have one fundamental difference: they malloc memory rather than allocate and construct objects. EASTL allocators are thus more like the malloc and free functions than like the new and delete operators. EASTL allocators also bear a marked resemblence to the allocators described in the Towards a Better Allocator Model proposal though they were developed independently without knowledge of the former. A fundamental difference between the two is that EASTL allocators don't have virtual functions; these have a performance cost that is to be avoided whenever possible.

The following illustrates the interface requirements for EASTL allocators. Items of interest are colored in blue.

```
class allocator
{
public:
    allocator(const char* name = "EASTL");
    allocator(const allocator& x);
    allocator(const allocator& x, const char* name);

    allocator& operator=(const allocator& x);

    void* allocate(size_t n, int flags = 0);
    void* allocate(size_t n, size_t alignment, size_t offset, int flags = 0);
    void  deallocate(void* p, size_t n);

    const char* get_name() const;
    void        set_name(const char* name);
};

bool operator==(const allocator& a, const allocator& b);
bool operator!=(const allocator& a, const allocator& b);
```

Some notes regarding eastl allocators:

- Allocators are required to be assignable. The user may define a swap function for the allocator as well.
- allocator is not necessarily a template, nor does it necessarily have member templates.
- The flags parameter describes hints for the allocation, such as whether the memory is temporary or permanent. This is very useful for high performance non-fragmenting memory allocation. See Appendix item 7.
- The alignment and offset parameters specify the alignment requirements of the allocation. This is important on platforms that require non-default memory alignment. It's not feasible for the container to expect the allocator know the required alignment ahead of time. This is because allocators may be shared between containers and because containers may use an allocator to allocate multiple types, as is the case with deque and hashtable containers.
- The name-related functions allow the user to name the allocator, which in turn results in all allocations from that allocator being tagged with a user-supplied name. This is important in game software whereby all allocated memory must be accountable. See Appendix item 12.
- The allocator is assumed to save the name pointer as opposed to copying its contents.
- allocator comparisons are significant, and equality is defined as meaning that equal containers can free memory allocated by each other.

- There is no max_size function, though such a thing might be useful for fixed-size pool allocators and their containers.
- An allocator which is a copy of another allocator compares as equal to the original allocator and acts as such.

EASTL containers

All EASTL containers follow a set of consistent conventions. Here we define the prototypical container which has the minimal functionality that all (non-adapter) containers must have. Items of interest are colored in blue.

```
enum iterator_status_flag
{
    isf_none            = 0x00,
    isf_valid           = 0x01,
    isf_current         = 0x02,
    isf_can_dereference = 0x04
};

template <class T, class Allocator = eastl::allocator>
class container
{
public:
    typedef container<T, Allocator>            this_type;
    typedef T                                  value_type;
    typedef T*                                 pointer;
    typedef const T*                           const_pointer;
    typedef T&                                 reference;
    typedef const T&                           const_reference;
    typedef ptrdiff_t                          difference_type;
    typedef impl_defined                       size_type;
    typedef impl-defined                       iterator;
    typedef impl-defined                       const_iterator;
    typedef reverse_iterator<iterator>         reverse_iterator;
    typedef reverse_iterator<const_iterator>   reverse_const_iterator;
    typedef Allocator                          allocator_type;
    typedef impl-defined                       node_type;

    static const size_t kNodeAlignment       = impl-defined;
    static const size_t kNodeAlignmentOffset = impl-defined;

public:

    container(const allocator_type& allocator = allocator_type());
    container(const this_type& x);

    this_type& operator=(this_type& x);
    void swap(this_type& x);
    void reset();

    allocator_type&       get_allocator();
    const allocator_type& get_allocator() const;
    void                  set_allocator(allocator_type& allocator);

    iterator       begin();
    const_iterator begin() const;
    iterator       end();
    const_iterator end() const;

    bool                 validate() const;
    iterator_status_flag validate_iterator(const_iterator i) const;
};
```

```
template <class T, class Allocator>
bool operator==(const container<T, Allocator>& a, const container<T, Allocator>& b);

template <class T, class Allocator>
bool operator!=(const container<T, Allocator>& a, const container<T, Allocator>& b);
```

Notes:

- EASTL guarantees that existing std STL container usage will behave the same under EASTL. Changes in functionality are introduced only by new member functions, extra template parameters, or simply new containers. This allows for easy conversion to EASTL, and often from EASTL to std STL as well.
- The get_allocator function allows the user to access the container's allocator as opposed to a copy of it. Without this, the user can only set the container's allocator on container construction, and this is impractical under some situations.
- Note that allocator parameters are specified by value and not by pointer, unlike the Halpern proposal. A pointer-based parameter was initially considered but it was determined that a class can encapsulate a pointer and thus act as a superset of a pointer-based design.
- Swapped containers do not swap their allocators. See Appendix item 29 regarding LWG #431.
- Some operations that involve multiple containers (such as list::splice) are invalid when container allocators are unequal and an exception is thrown if the allocators are unequal.
- Newly constructed empty containers must do no memory allocation. Some STL and other container libraries allocate an initial node from the class memory allocator. EASTL containers by design never do this. If a container needs an initial node, that node should be made part of the container itself or be a static empty node object.
- Empty containers (new or otherwise) contain no constructed objects, including those that might be in an 'end' node. Similarly, no user object (e.g. of type T) should be constructed unless required by the design and unless documented in the container/algorithm contract.
- The reset function is a special extension function which unilaterally resets the container to an empty state without freeing the memory of the contained objects. This is useful for very quickly tearing down a container built into scratch memory and is a useful feature for high performance programming. No memory is allocated by reset, and the container has no allocated memory after the reset is executed. It is inherently unsafe for types that have non-trivial destructor.
- There is no max_size function, though such a thing might be useful for fixed-size containers. The reason it doesn't exist is simply because in practice it is rarely used.
- The validate and validate_iterator functions provide explicit container and iterator validation. EASTL provides an option to do implicit automatic iterator and container validation, but full validation (which can be potentially extensive) has too much of a performance cost to execute implicitly, even in a debug build. So EASTL provides these explicit functions which can be called by the user at the appropriate time and in optimized builds as well as debug builds.
- EASTL does not solve the out of memory problem. See Appendix item 18.

EASTL flaws

Some aspects of EASTL turned out to be not quite right. The following is a list of some of the things that might be done differently.

- The basic_string implementation propagates the problems with std::basic_string. Two fundamental problems with basic_string are that it has too many member functions and that end() should have been used instead of npos for member algorithms. EASTL fixes neither of these and in fact adds a few additional member functions. A better string class would fix the above two problems and would also implement optional behavior policies such as copy-on-write.
- Some people want list::size to be O(1) and some want it to be O(n). eastl::list should solve this by providing a template parameter to let the user choose, but currently it has no such option. A future revision of eastl::list will likely make this a template policy option.

- fixed-substring is currently a subclass of basic_string, which creates some unsafe situations. However, this is an implementation detail which can be rectified.
- The container reset function should perhaps have been given a less simple name. Some new users confuse it with clear.
- EASTL does not solve the out-of-memory problem any differently than std STL (aside from it providing intrusive containers and fixed size containers). The std STL solution (exception handling) is supported by EASTL but is not favored and is usually disabled. See Appendix item 18.
- The EASTL compile-time and runtime checking is not as thorough as that of libstdc++, as libstdc++ and possibly other std STL implementations have compile-time checking for some types of requirements. This too is something that can be improved in EASTL if deemed necessary.

Appendix

This appendix contains supplementary material referenced by the document body. In many cases the items here describe in more detail what is meant in the body. It is placed here in order to avoid getting in the way of the primary text.

1 - (removed)

2 - A debuggable list container

To make a list container debuggable, whereby the user can easily inspect and traverse it with a traditional debugger, we use Curiously Recurring Template Pattern like so in EASTL.

```
template <typename LN>
struct ListNodeBaseProxy
{
    LN* mpNext;
    LN* mpPrev;
};

template <typename T>
struct ListNode : public ListNodeBaseProxy< ListNode<T> >
{
    T mValue;
};

template <typename T, typename Allocator>
class ListBase  // Typically the list class inherits from a base class such as this.
{
public:
    typedef T value_type;
    typedef ListNode<T> node_type;
    typedef ListNodeBaseProxy< ListNode<T> > base_node_type;

protected:
    base_node_type mNode;
    . . .
};
```

3 - Algorithm type preservation problem

The following example demonstrates a problem with current std STL implementations. What the TableBasedSorter specifically does is somewhat meaningless and arbitrary.

```
struct TableBasedSorter {
    TableBasedSorter(const int values[128]) {
        for(int i = 0; i < 128; ++i)
```

```
            mTable[i] = ((values[i] ^ 0xff80) + 128) - i;
        }

        bool operator()(int a, int b) const {
            return mTable[b] < mTable[a];
        }

        int mTable[128];
    };


    std::sort(v, v + 128, TableBasedSorter(values));
```

The problem is that the std STL implementations pass the Compare function object around internally by value, triggering copying of it. This is compounded by debug STL implementations which call a Compare checker that evaluates the comparison twice instead of once so that it can enforce !(Compare(x, y) && Compare(y, x)). It thus ends up calling the copy constructor O(n log(n)) times. The user might think that the following should rectify the problem:

```
HuffmanHistoSorter compare;
std::sort<int*, TableBasedSorter&>(v, v + 128, compare);
```

But it doesn't work, because the sort function ignores the compare reference template parameter and makes copies of the compare internally.

The conventional workaround for this problem is to create a proxy compare class that references the real compare object, but this is clumsy and creates a performance dragging memory indirection. Why not simply have the algorithm obey the user's request? It's not clear if the C++ standard requires that the user's request be respected, but it would be nice if it did. EASTL does so.

4 - (removed)

5 - Game source and binary sizes

Game software for desktop platforms and for 2005+ console platforms is often large-scale software.

- The source code for large PC games is usually in the range of 500,000 to 2 million lines.
- Application binaries are in the range of 5 to 20 MB.
- Game source data (graphics, maps, search graphs, UI, movies, animations, scripting) source is in the range of 5 to 50 GB.
- Compiled game data is usually in the range of 500 MB to 4 GB (not including movie files).

In addition to the code for the games themselves, game software tends to have a lot of supporting tool or "pipeline" code. Since games tend to have a lot of data, a lot of custom tools are written to generate and process this data. The size of these tools can be larger than the size of the game applications themselves. However, these tools usually don't need to have the same attention to performance as does the game code itself. While game code is almost exclusively C++, tools are written in a variety of languages, including most commonly C#, Python, C++, and Perl.

6 - Garbage collected allocators

Garbage collected memory is not currently considered an option for game development, though current iterative generational garbage collection is well-done. A full discussion of this deserves a paper of its own, but suffice it to say that the realities of limited memory and specialized memory as well as the somewhat real-time requirements of game software make garbage collected memory currently not viable. However, EA is interested

in the results of research in this area and is interested in the possibility of collaborative work with researchers on this topic.

7 - Permanent and temporary memory

A useful technique for reducing memory fragmentation which is often used by game developers is to separate allocated memory into two classes: temporary memory and permanent memory. Some EA games have consumed system memory completely enough that that they would fail to run without this optimization. The way it works is that memory which is allocated once on startup is deemed to be permanently allocated. Similarly, memory which is allocated once for a new game level is deemed to be permanently allocated. Memory which is dynamically allocated and freed in a arbitrary or out-of-order way is deemed to be temporarily allocated. Memory which is permanent is allocated from the top of the heap downward, and memory which is temporary is allocated from the low end of the heap.

```
[temporary memory -->           (free space)           <-- permanent memory]
(low memory)                                                  (high memory)
```

The result is that permanent allocations are tightly packed together in high memory and don't create dead spots in low memory. Sometimes permanent and temporary memory are sometimes referred to as high and low memory, for reasons that should be obvious. This technique is useful on both fixed-memory systems and on mapped virtual memory systems such as Windows. The custom heaps in use at EA explicitly support the concept of permanent and temporary (high and low) memory.

8 - (removed)

9 - Debug builds can't be slow

Game software is notoriously difficult to test, and it is particularly difficult to automate game software testing. Individual modular components can be automatically unit tested (as is EASTL) but the interactive gameplay of an application is much harder test. There is currently no satisfactory solution to this problem and no solution appears to be imminent. This is a topic of research on its own, and until it may be some day solved, the majority of game testing is done interactively by humans.

The interactive nature of game software means that slow debug builds are hard to test and can waste a lot of human testers' time. Since much of the testing involves testing the interactivity of the game, a slow game can make interactivity testing virtually impossible. Also, a slow application can waste a lot of programmer time spent waiting for the application to get to some desired state. Such a wait occurs for any programmer testing any kind of application, but it often happens in game programming that the programmer needs to follow a potentially long series of gameplay steps in order to get the application to some state, and there might not shortcuts or cheats to get there.

When we were using STLPort on some of our earlier STL-based PC games at EA, we enabled inlining in debug builds in order to avoid all the function calls that aren't being inlined in debug builds. Enabling inlining wasn't desirable (as it made debugging more difficult) but it made the debug builds more responsive.

10 - Function call avoidance

EASTL containers avoid function calls to the extent possible, even if such calls might be inlined by the compiler. The avoidance of function calls makes it easier for the compiler to optimize and easier for the user to trace at runtime. Many of the improved benchmark results of EASTL over std STL are due to the compiler not being able to inline functions that the std STL author assumed or hoped that it would. Compiler inlining weaknesses are discussed in Appendix item 15 . Additionally, builds with inlining disabled can yield sluggish applications that become unsuitable for testing. See Appendix item 9.

The downside to function call avoidance is that it makes container authoring and maintenance more difficult for the author, though containers tend to have little work done on them once implemented and debugged. There is a maxim that EASTL attempts to follow: write for your users first, your peers second, and yourself last.

The following demonstrates the function calls involved in the vector::resize function in EASTL and libstdc++. The EASTL version does some math and then calls insert or erase. The libstdc++ version class functions which themselves call functions, and so on. Consider that the code calling resize may itself be subject to inlining. This layering of function calls puts a burden on the compiler to inline optimally which it may or may not be able to do.

EASTL vector::resize

```
void resize(size_type n)
{
    if(n > (mpEnd - mpBegin))
        insert(mpEnd, n - (mpEnd - mpBegin), value_type());
    else
        erase(mpBegin + n, mpEnd);
}
```

libstdc++ vector::resize

```
void resize(size_type __new_size)
{
    resize(__new_size, value_type());
}

void resize(size_type __new_size, const value_type& __x)
{
    if(__new_size < size())
        erase(begin() + __new_size, end());
    else
        insert(end(), __new_size - size(), __x);
}

size_type size() const
{
    return size_type(end() - begin());
}

const_iterator begin() const
{
    return const_iterator(this->_M_impl._M_start);
}

const_iterator end() const
{
    return const_iterator(this->_M_impl._M_finish);
}

__normal_iterator(const _Iterator& __i) : _M_current(__i) { }
```

11 - Argument-dependent lookup safety

If the user calls a std STL function with an argument that happens to exist in a namespace that has functions of the same name as those in the std namespace, what should std STL do if it needs to call a function of that name? Is the C++ standard clear on the requirement here? Should std STL explicitly call the std version of that function or should it call the version unqualified? In the large majority of cases, the STL should explicitly call the std version, with exceptions being made only for functions that were intended to be overridden, such as swap.

The following code may fail to compile if the std::sort implementation makes unqualified min calls, as the compiler will use ADL to find the test::min function in addition to std::min.

```
namespace test
{
    int min(int x, int y);

    class X { };
}

std::sort(XArray, XArray + 10);
```

At least one major commercial STL implementation has recently not been ADL-safe, despite having otherwise excellent standards support. The vendor has been notified and agreed to fix the problem future releases.

12 - Memory tagging

Memory tagging is the process of associating some useful information with a memory allocation. It is typically done in one or more of three ways:

1	file / line	The __FILE__ and __LINE__ values of the allocation request are stored.	This is lightweight but doesn't work well when the allocation comes from within a shared library, such as STL. Another problem with it is that it doesn't tell you anything about the nature of the allocation, such as who did the allocation or what the allocation is used for.
2	call stack	The call stack of the allocation request is stored.	This works well but is not as lightweight as file / line tagging. As with file / line tagging, call stack tagging doesn't tell you much about the allocation aside from where it was done.
3	name	The allocation is given a name, such as "VehicleInfo".	This has the advantage of telling you what the allocation is used for. For categorization and budgeting purposes, the name can be hierarchical, such as with "Automata/VehicleInfo". Allocation names can also be used in many cases to tell you where the allocation occurred as well. Memory tagging combined with detailed heap reports can tell you if similarly used memory is physically proximate. A disadvantage of this scheme is that it requires explicit support in user libraries; it cannot be automatically generated like a call stack can.

Each has its strengths and weaknesses and users will often favor one technique over another. However, they aren't mutually exclusive. Typically file/line and name will be used or callstack and name will be used together, as naming is somewhat complementary with the other two. EASTL explicitly supports memory name tagging, as it is so useful. Memory heaps in EA explicitly support all three of the above methods of memory tagging, as well as others: allocation time, allocation number, allocation group ids, arbitrary user-supplied data, and others.

13 - wchar_t portability

The large majority of commercial game software uses 8 bit UTF8-encoded text or 16 bit UCS2-encoded text. Different operating systems have different conventions for encoding text, but these have little bearing on what a game chooses to use for its text. The discussion of the strengths and weaknesses of text encoding options is outside the scope of this document, but the discussion of how to portably support a given encoding in a library is pertinent.

char16_t and char32_t are sized character types, much like uint16_t and uint32_t are sized integer types. These sized character types are extensions defined by EA which allow string code and data to be more portable, though they don't solve the string literal problem. wchar_t is not considered portable, as it may be 8, 16, or 32 bits, depending on the system (BEOS, Windows, Unix, respectively). The Unicode Standard recommands against using wchar_t for this same reason (version 4, section 5.2, paragraph 2).

If your application wants to use a 16 bit string but your compiler defines wchar_t as 32 bits, you cannot use the standard string library provided with the compiler. And even if you could recompile the standard string library to be 16 bit, you'd then be breaking any code you have that wants to use a 32 bit wchar_t. Another problem is that there is no way to declare a 16 bit string literal if wchar_t is 32 bits. Also, if wchar_t is defined as 8 bit UTF-8, there will be difficulties if you do a lot of string processing, which we can say with confidence because we've done this. So what we do is define char8_t, char16_t, and char32_t and rewrite the standard string library to explicitly support all three types. A side benefit is that the resulting string library is more efficient than the one provided with the compiler. See Appendix item 14.

Sized character types would be useful to have in C++, as they allow for portable and efficient usage of strings in a multi-platform development environment. Instead of prefixing a string with L, how about prefixing it with L16 or L32 instead? There are proposals for such sized char types in C++09.

14 - Why replace standard library functions?

Often you will see game programmers avoid using functions from the C and C++ standard library and instead use re-implemented versions. The reasons for this depend on the individual case but are usually driven by some practical performance issue. We provide a table of commonly re-implemented functionality and the most common reasons for it. We hope that this may help library designers understand the requirements of game software.

Replaced functionality	Rationale
memcpy, memmove, memset	memcpy implementations for existing well-established desktop platforms are known to be well-optimized and are rarely replaced. However, it turns out that library vendors for new hardware sometimes don't come up with optimized implementations until well after the hardware's initial availability. Also, in many cases memory manipulation functions can be improved for the case of memory aligned on page-sized or cache line-sized boundaries and thus game software often implements specialized memory manipulation functions for these cases.
printf, sprintf	sprintf is a function that is often re-implemented by game software. Typical reasons are thus: • sprintf is slow because it locks mutexes and reads internal locale information. • sprintf brings in enough extenal code to make it too big to be usable on some machines such as the new Sony/IBM Cell processors. • wsprintf can't be relied upon to work with 16 bit strings, as described in Appendix item 13. • Some sprintf implementations may allocate memory from the global heap. • Some sprintf implementations (e.g. Playstation 2) are very slow with double precision floating point math. • sprintf doesn't support explicitly sized types such as int32_t. C99's PRId32 is a poor workaround, as it is obfuscational and non-portable. • sprintf supports decimal (%d), octal (%o), and hexidecimal (%x) output, but not binary output (%b). • sprintf sometimes isn't provided by the compiler vendor. Or more often, some useful variation of it such as vsnprintf isn't provided by the compiler vendor.

wcscpy, wcslen, etc.	As explained in Appendix item 13, wchar_t is not portable in practice. If your game uses 16 bit UCS2-encoded strings -- as many do -- then you can't rely on the standard library wide string functions being usable. So games will commonly re-write the entire string library in order to be portable. This is unfortunate, as the vendor-provided C string functions are usually well-implemented.
fopen, fstream	The FILE-based standard library functions for manipulating files is fine for many purposes and applications. However, it isn't very good for game applications, particularly console-based applications. Even in those cases where sychronous fopen/fread behavior is appropriate, games almost always use the lower level operating system-provided equivalent functionality. As such, FILE use is uncommon in console-based games and often in PC-based games as well. The primary reasons for this are: • FILE-based IO is inherently blocking, and games work best if IO is asynchronous and priority and head proximity-driven • FILE-based IO is inherently buffered, whereas this is often not what game software wants. • FILE-based IO is slower (e.g. mutex locking) than equivalent system-provided functionality and usually more limited (e.g. no buffer control) in capabilities.
rand	The problems with the standard library rand are well-known and are likely part of the impetus for improving random number generation with TR1. A brief summary of the weaknesses is: • Limited to 16 bit values in practice. • Poor randomness. • Can't have a private implementation. • Poor performance.
atoi, strtol, etc.	A problem with functions such as atoi and strtol is that they work with non-portable types such as long and wchar_t. On game platforms, a long may be an inefficient data type (as it is on the Playstation 2). It is preferable to use APIs that work with sized types such as uint32_t and uint64_t, as they guarantee portable behavior and in practice are at least as efficient as types such as int and long. See Appendix item 21.
strncpy, strncat	These functions are too hard to use safely. We have replaced them with strlcpy and strlcat respectively, though much game software inside and outside EA still use these. Also, the fixed_string or fixed_substring class can be safely used to replace these functions in a friendly way.
strftime, set_locale	C-provided locale functionality suffers from being neither portable nor overridable. There is no portable way to tell what locale to use, as compilers implement it differently. However, the differences can often be abstracted away with macros. But once you get over this you still have the problem of API functions not always being helpful. The strftime function, for example, doesn't do time/date format localization but instead requires the user to do it. Again the user is required to abstract this away. Another problem is that the user has no way of extending or modifying the behavior of functions like strftime to accomodate some practical requirement. An end result of this is that game software often just rewrites functions like strftime to be simpler, more flexible, and usually more efficient as well.
new/delete, malloc/free	Custom heaps fragment memory less, use memory more efficiently, and handle aligned memory better that system provided heaps. Custom heaps provide built-in debugging features such as

	tagging and strong validation, among others. Additionally, gaming platforms often require usage of different kinds of memory for which there is no portable API to manipulate.
assert	The fundamental problem with assert is that there is no way to override it or intercept it and redirect it to application-provided facilities. The result is that assert usage is verboten because it operates outside the application's control and in practice usually unilaterally causes the application to exit. This is unfortunate because it goes against the purpose of standard library facilities: to provide a portable standard implementation of useful universal functionality. What we would like to see in the C++ standard is a portable way for users to be able to override the assert function and redefine its behavior.

15 - Compiler inlining problems

The GCC C++ compiler is a great compiler with a lot of flexibility and portability. However, it has one weakness of significance that affects templated libraries such as the STL: it is not very good at inlining. There are a number of compiler options to tweak the inlining parameters, but it seems to be fairly difficult to get the compiler to do the "right thing." This appears to be a known problem and it is hoped that a future version of the compiler will address this. The Microsoft C++ compiler does a rather good job of inlining and might be worth using as a reference.

A full discussion of compiler inlining characteristics is outside the scope of this document, but some Internet discussions regarding GCC inlining problems can be found at:

- http://groups.google.com/group/comp.lang.c++/browse_frm/thread/b74eed16bd48d42e
- http://groups.google.com/group/fa.linux.kernel/browse_frm/thread/1861b2634cdfa68a/
- http://www.pixelglow.com/lists/archive/macstl-dev/2005-September/000154.html

16 - EASTL intrusive_list

As an example of an intrusive container, we show the interface for eastl::intrusive_list. The actual implementation breaks intrusive_list into intrusive_list_base and intrusive_list in order to place some of the non-templated functions into a base class. Other EASTL intrusive containers (e.g. intrusive_hash_map) have a similar philosophy. Recall that the primary advantage of intrusive containers is that they allow the user to provide node memory.

```
struct intrusive_list_node      // Users can use this or provide their own.
{
    intrusive_list_node* pNext;
    intrusive_list_node* pPrev;
};


template <typename T, typename Pointer, typename Reference>
class intrusive_list_iterator
{
    public:
    typedef intrusive_list_iterator<T, Pointer, Reference> this_type;
    typedef intrusive_list_iterator<T, T*, T&>             iterator;
    typedef intrusive_list_iterator<T, const T*, const T&> const_iterator;
    typedef T                                              value_type;
    typedef T                                              node_type;
    typedef ptrdiff_t                                      difference_type;
    typedef Pointer                                        pointer;
    typedef Reference                                      reference;
    typedef bidirectional_iterator_tag                     iterator_category;
```

```
public:
    intrusive_list_iterator();
    explicit intrusive_list_iterator(pointer pNode);
    intrusive_list_iterator(const iterator& x);

    reference operator*() const;
    pointer operator->() const;

    intrusive_list_iterator& operator++();
    intrusive_list_iterator& operator--();

    intrusive_list_iterator operator++(int);
    intrusive_list_iterator operator--(int);
};


template <typename T = intrusive_list_node>
class intrusive_list
{
public:
    typedef intrusive_list<T>                                this_type;
    typedef T                                                node_type;
    typedef T                                                value_type;
    typedef eastl_size_t                                     size_type;
    typedef ptrdiff_t                                        difference_type;
    typedef T&                                               reference;
    typedef const T&                                         const_reference;
    typedef T*                                               pointer;
    typedef const T*                                         const_pointer;
    typedef intrusive_list_iterator<T, T*, T&>               iterator;
    typedef intrusive_list_iterator<T, const T*, const T&> const_iterator;
    typedef eastl::reverse_iterator<iterator>                reverse_iterator;
    typedef eastl::reverse_iterator<const_iterator>          const_reverse_iterator;

public:
    intrusive_list();
    intrusive_list(const this_type& x);
    this_type& operator=(const this_type& x);

    iterator               begin();
    const_iterator         begin() const;
    iterator               end();
    const_iterator         end() const;

    reverse_iterator       rbegin();
    const_reverse_iterator rbegin() const;
    reverse_iterator       rend();
    const_reverse_iterator rend() const;

    reference       front();
    const_reference front() const;
    reference       back();
    const_reference back() const;

    bool empty() const;
    eastl_size_t size() const;
    void clear();
    void reverse();

    void push_front(T& x);
    void pop_front();
    void push_back(T& x);
    void pop_back();

    iterator locate(T& x);
    const_iterator locate(const T& x) const;
```

```
        iterator insert(iterator pos, T& x);
        iterator erase(iterator pos);
        iterator erase(iterator pos, iterator last);
        void swap(intrusive_list&);

        static void remove(T& value); // Erases an element from a list without having the list; O(1).

        void splice(iterator pos, T& x);
        void splice(iterator pos, intrusive_list& x);
        void splice(iterator pos, intrusive_list& x, iterator i);
        void splice(iterator pos, intrusive_list& x, iterator first, iterator last);

        void merge(this_type& x);

        template <typename Compare>
        void merge(this_type& x, Compare compare);

        void unique();

        template <typename BinaryPredicate>
        void unique(BinaryPredicate);

        void sort();

        template<typename Compare>
        void sort(Compare compare);

        bool validate() const;
        int  validate_iterator(const_iterator i) const;
    };
```

17 - Disabling of exception handling

Game software usually disables exception handling. A full analysis of the strengths and weaknesses of C++ exception handling in game software is outside the scope of this document, but a few comments can be made here to help clarify some issues.

- Exception handling incurs some kind of cost in all compiler implementations, including those that avoid the cost during normal execution.
- However, in some cases this cost may arguably offset the cost of the code that it is replacing.
- Exception handling is often agreed to be a superior solution for handling a large range of function return values.
- However, avoiding the creation of functions that need large ranges of return values is superior to using exception handling to handle such values.
- Using exception handling correctly can be difficult in the case of complex software.
- The execution of throw and catch can be significantly expensive with some implementations.
- Exception handling violates the don't-pay-for-what-you-don't-use design of C++, as it incurs overhead in any non-leaf function that has destructable stack objects regardless of whether they use exception handling.

The approach that game software usually takes is to avoid the need for exception handling where possible; avoid the possibility of circumstances that may lead to exceptions. For example, verify up front that there is enough memory for a subsystem to do its job instead of trying to deal with the problem via exception handling or any other means after it occurs.

However, some game libraries may nevertheless benefit from the use of exception handling. It's best, howerver, if such libraries keep the exception handling internal lest they force their usage of exception handling on the rest of the application.

18 - Out of memory

EASTL does not solve the out-of-memory problem any differently than std STL (aside from it providing intrusive containers and fixed size containers). The std STL solution (exception handling) is supported by EASTL but is not favored and is usually disabled (see Appendix item 17). There are four common solutions in game software to dealing with out-of-memory conditions within libraries:

The library throws an exception if memory cannot be allocated.	This isn't used very often because it is has more overhead and is much harder to safely use than the following two methods. One forgotten "try" by the user and the application can unexpectedly crash.
The allocator calls a user-callback upon failure, whereby the user callback frees up memory in an application-specific way.	This is a reliable method because it allows the failure to be handled in a single place in a consistent way and it only needs to be written once and the user doesn't have to worry.
The user simply guarantees up front that there is enough memory.	This is the simplest and most reliable method and is the one that's used the most. A common way of guaranteeing the availability of the memory is to allocate it up-front and hand it to the system that needs it.
The library propagates and error value up to the caller.	This solution is non-optimal, especially if there are multiple return values that are specifically handled. Exception handling is often considered superior to this solution.

Dealing with exception handling at the level of the container user is tedious and error-prone; dealing with it at the level of the allocator is easier for the user and is more reliable. Using exception handling to properly and safely deal with out-of-memory conditions is a daunting and tedious task, especially in an environment of custom allocators. This is not a criticism of C++ exception handling in general but rather is an observation about using it with STL containers.

19 - Benchmarks often miss cache effects

Typically a benchmark of comparing two functions will go through some effort to make sure that the measurements aren't polluted by memory cache effects such as cache misses. This is a fine approach for comparing functions that are roughly equivalent in their memory effects. A typical such benchmark will call a function once, then start the timer, call it 1000 times, and finally stop the timer. However, such a testing approach may yield misleading results for cases whereby the compared functions are not equivalent in their memory effects. The benchmark intentionally chose to ignore cache misses that occurred during the first function call, but in real-world execution such cache misses may well affect application performance because these functions aren't actually called 1000 times in a row one after another. A better way of measuring performance is to have a 1000 different functions which are called one after another and which exercise the code and data memory caches like a real-world application might.

The Technical Report on C++ Performance recognizes cache effects at one point in the document but seems to dismiss them in its own measurements and conclusions. For example, it states that the timing analysis of virtual functions is straightforward and concludes that the cost of virtual functions is "a fixed number of machine instructions." This analysis is somewhat misleading, as it implies that there are no cache misses or that memory caches are so large that a cache miss for any code will only ever occur once on startup. See section 7.2.1 of the report. A full discussion of the usage of virtual functions in game development is outside the scope of this document, but it is sufficient to say here that the cost of virtual functions in practice is likely greater than implied by the Technical Report on C++ Performance, largely due to practical cache effects.

20 - Performance comparison

A couple of comparisons are presented here which compare EASTL with a commonly used commercial version of std STL. EASTL is generally faster, though in many cases the differences are minimal. In some cases speed improvements aren't very important; nobody is going to know or care if min_element executes 10% faster or slower in EASTL. In other cases -- such as with the sort algorithm -- speed improvements are important. The source code to the benchmark is distributed with this document. A reported result of 1.40 means that EASTL ran 40% faster; a ratio of 0.75 means that EASTL ran 33% (1/.75) slower.

In practice, with GCC it turns out that EASTL performs a little better than the benchmarks indicate due to EASTL being easier to inline. The benchmark code is very simple and is relatively easy for the compiler to inline, but real-world code in practice often turns out to not inline as well.

A: Windows / Pentium 4 x86 / VC8 -Og -Oi -Ot -Oy -Ob2 -GF -Gy / compiler-supplied STL
B: Console / 64 bit PowerPC / VC7 -Ox -Ob2 -Oi -Os / compiler-supplied STL
C: Linux / Pentium 4 x86 / GCC 4.1 -O3 / libstdc++
D: Mac OS X / G5 PowerPC 32 bit mode / GCC 4.0 -Os / libstdc++

Green: EASTL 15% or more faster (>= 1.15)
Blue: Std STL 15% or more faster (<= 0.87)

Test	Windows	Console	Linux	Mac
algorithm/adj_find/vector<TestObject>	1.00		0.98	1.25
algorithm/copy/vector<LargePOD>	1.09	1.24	0.99	0.92
algorithm/copy/vector<uint32_t>	0.95	1.15	1.28	1.01
algorithm/copy_backward/vector<LargePOD>	1.03	1.82	1.06	1.10
algorithm/copy_backward/vector<uint32_t>	0.78	1.07	1.23	1.10
algorithm/count/vector<uint64_t>	1.00	1.91	0.93	1.77
algorithm/equal_range/vector<uint64_t>	1.34	1.14	0.89	1.74
algorithm/fill/bool[]	9.43	1.25	1.00	1.00
algorithm/fill/char[]/'d'	7.36	17.05	1.02	1.00
algorithm/fill/vector<char>/'d'	7.33	17.27	7.38	88.80
algorithm/fill/vector<char>/0	9.39	30.75	8.84	100.00
algorithm/fill/vector<uint64_t>	1.07	30.60	1.06	1.08
algorithm/fill/vector<void*>	1.02	1.25	0.88	1.95
algorithm/fill_n/bool[]	13.71	1.81	1.00	0.98
algorithm/fill_n/char[]	10.67	19.41	1.01	1.01
algorithm/fill_n/vector<uint64_t>	1.00	19.18	1.10	0.90
algorithm/find_end/string/end	31.63	0.98	0.89	0.58
algorithm/find_end/string/middle	3.42	21.77	0.83	0.63
algorithm/find_end/string/none	1.51	2.74	0.82	0.66
algorithm/lex_cmp/schar[]	1.88	1.36	1.47	4.01
algorithm/lex_cmp/vector<TestObject>	0.99	1.26	0.70	1.16
algorithm/lex_cmp/vector<uchar>	1.88	0.75	1.45	15.54
algorithm/lower_bound/vector<TestObject>	1.43	2.22	1.07	1.63
algorithm/min_element/vector<TestObject>	1.13	1.07	1.47	1.44
algorithm/rand_shuffle/vector<uint64_t>	1.17	1.09	0.96	1.16
algorithm/reverse/list<TestObject>	1.02	1.05	1.20	0.86
algorithm/reverse/vector<TestObject>	1.03	0.91	0.96	1.27
algorithm/search/string<char>	2.48	1.01	0.77	1.49
algorithm/search_n/string<char>	1.39	7.32	1.07	2.58
algorithm/unique/vector<TestObject>	0.98	14.53	0.95	1.26
algorithm/unique/vector<uint32_t>	1.07	1.08	1.13	2.41
algorithm/unique/vector<uint64_t>	1.13	1.04	0.97	1.74
algorithm/upper_bound/vector<uint32_t>	1.47	1.15	1.13	1.99
		1.06		

bitset<15>/>>=/1	1.04	1.10	0.82	1.61
bitset<15>/count	0.97	0.97	0.94	1.00
bitset<15>/flip	1.87	1.08	0.80	2.00
bitset<15>/reset	1.00	1.00	0.91	1.00
bitset<15>/set()	1.25	1.08	0.80	2.63
bitset<15>/set(i)	1.17	1.00	0.91	1.96
bitset<15>/test	1.02	1.10	1.17	1.87
bitset<35>/>>=/1	0.57	1.93	1.19	2.64
bitset<35>/count	0.98	1.63	1.71	1.27
bitset<35>/flip	1.39	2.14	1.00	1.97
bitset<35>/reset	1.50	1.54	0.95	0.90
bitset<35>/set()	1.40	3.02	1.05	3.40
bitset<35>/set(i)	1.00	1.05	0.90	2.29
bitset<35>/test	1.14	1.10	1.22	1.79
bitset<75>/>>=/1	0.94	0.60	0.81	1.86
bitset<75>/count	1.68	0.65	1.05	1.00
bitset<75>/flip	1.00	1.82	0.91	1.29
bitset<75>/reset	1.80	2.16	1.00	0.49
bitset<75>/set()	1.50	3.02	0.91	1.46
bitset<75>/set(i)	1.00	0.94	0.89	1.94
bitset<75>/test	0.88	0.91	1.37	2.20
bitset<1500>/>>=/1	0.99	1.74	1.21	1.13
bitset<1500>/count	0.99	1.90	1.00	1.00
bitset<1500>/flip	1.08	1.89	1.44	0.87
bitset<1500>/reset	0.95	2.18	0.99	1.61
bitset<1500>/set()	1.14	1.89	1.71	1.37
bitset<1500>/set(i)	0.98	0.95	0.82	2.28
bitset<1500>/test	1.00	0.90	1.21	1.78
deque<ValuePair>/erase	2.48	1.05	1.19	0.86
deque<ValuePair>/find	1.56	1.85	1.06	1.05
deque<ValuePair>/insert	2.97	1.64	1.03	1.24
deque<ValuePair>/iteration	0.94	1.80	1.06	0.98
deque<ValuePair>/operator[]	1.30	1.40	1.55	2.66
deque<ValuePair>/push_back	8.67	6.43	0.98	1.65
deque<ValuePair>/push_front	9.73	5.03	1.03	1.68
deque<ValuePair>/sort	1.51	1.61	1.01	1.00
hash_map<string, uint32_t>/clear	0.58	1.01		1.16
hash_map<string, uint32_t>/count	2.23	2.75	0.75	1.19
hash_map<string, uint32_t>/erase pos	0.67	0.69	1.24	2.57
hash_map<string, uint32_t>/erase range	1.47	1.98	1.15	1.09
hash_map<string, uint32_t>/erase val	1.18	1.22	0.82	0.98
hash_map<string, uint32_t>/find	1.66	2.19	0.75	1.58
hash_map<string, uint32_t>/find_as/char*	1.96	2.34	1.21	4.61
hash_map<string, uint32_t>/insert	0.59	0.65	3.52	0.73
hash_map<string, uint32_t>/iteration	1.29	1.29	0.54	3.49
hash_map<string, uint32_t>/operator[]	1.60	2.11	3.00	2.21
			1.53	

hash_map<uint32_t, TestObject>/clear	0.89	1.02	0.82	1.22
hash_map<uint32_t, TestObject>/count	2.95	2.78	1.02	1.38
hash_map<uint32_t, TestObject>/erase pos	0.90	1.34	1.00	1.62
hash_map<uint32_t, TestObject>/erase range	3.41	5.30	1.05	1.03
hash_map<uint32_t, TestObject>/erase val	1.94	1.89	1.05	1.24
hash_map<uint32_t, TestObject>/find	2.72	1.89	0.76	1.24
hash_map<uint32_t, TestObject>/insert	2.55	2.04	0.91	1.05
hash_map<uint32_t, TestObject>/iteration	1.04	0.74	2.29	4.03
hash_map<uint32_t, TestObject>/operator[]	2.20	1.75	1.37	1.96
heap (uint32_t[])/make_heap	1.21	1.07	0.97	0.93
heap (uint32_t[])/pop_heap	1.09	1.01	0.98	0.98
heap (uint32_t[])/push_heap	1.59	1.08	1.05	1.13
heap (uint32_t[])/sort_heap	0.96	1.02	1.12	1.01
heap (vector<TestObject>)/make_heap	0.93	1.05	1.11	1.02
heap (vector<TestObject>)/pop_heap	0.93	0.98	1.07	1.02
heap (vector<TestObject>)/push_heap	0.97	1.03	0.88	1.12
heap (vector<TestObject>)/sort_heap	0.95	0.97	0.99	1.06
list<TestObject>/ctor(it)	1.10	1.09	1.03	1.00
list<TestObject>/ctor(n)	1.04	1.15	0.91	1.18
list<TestObject>/erase	1.13	0.96	0.97	1.47
list<TestObject>/find	1.24	0.97	1.00	2.33
list<TestObject>/insert	1.34	0.99	1.00	1.35
list<TestObject>/push_back	1.60	1.17	1.04	1.35
list<TestObject>/remove	1.23	1.13	0.90	1.61
list<TestObject>/reverse	1.06	1.25	1.09	1.33
list<TestObject>/size/1	1.00	1.00	1.00	1.07
list<TestObject>/size/10	1.00	1.00	1.01	1.00
list<TestObject>/size/100	1.00	1.00	0.96	1.00
list<TestObject>/splice	2.28	2.27	1.30	3.56
map<TestObject, uint32_t>/clear	1.00	1.01	1.00	1.02
map<TestObject, uint32_t>/count	2.15	2.03	0.94	1.21
map<TestObject, uint32_t>/equal_range	1.75	1.56	1.35	1.81
map<TestObject, uint32_t>/erase/key	1.38	1.53	1.26	1.66
map<TestObject, uint32_t>/erase/pos	1.26	1.31	1.38	1.09
map<TestObject, uint32_t>/erase/range	1.10	1.08	1.01	0.97
map<TestObject, uint32_t>/find	1.42	1.12	0.99	1.25
map<TestObject, uint32_t>/insert	1.03	1.08	1.04	1.17
map<TestObject, uint32_t>/iteration	1.88	1.83	2.05	1.36
map<TestObject, uint32_t>/lower_bound	1.18	1.13	0.92	1.29
map<TestObject, uint32_t>/operator[]	1.16	1.06	0.92	1.12
map<TestObject, uint32_t>/upper_bound	1.26	1.09	0.83	1.30
set<uint32_t>/clear	1.00	0.98		1.01
set<uint32_t>/count	1.98	2.18	1.00	1.31
set<uint32_t>/equal_range	1.59	1.78	1.05	2.12

set<uint32_t>/erase range	1.06	1.09	1.49	1.00
set<uint32_t>/erase/pos	0.92	1.13	1.10	0.99
set<uint32_t>/erase/val	1.37	1.49	1.11	1.88
set<uint32_t>/find	1.08	1.10	1.62	1.39
set<uint32_t>/insert	1.04	1.11	1.02	1.29
set<uint32_t>/iteration	0.86	1.62	1.13	1.33
set<uint32_t>/lower_bound	1.06	1.17	1.55	1.11
set<uint32_t>/upper_bound	1.07	1.13	1.06	1.04
			0.93	
sort/q_sort/TestObject[]	1.50	2.39	1.14	1.15
sort/q_sort/TestObject[]/sorted	1.31	1.60	1.01	1.32
sort/q_sort/vector<TestObject>	1.49	2.40	1.28	1.14
sort/q_sort/vector<TestObject>/sorted	1.38	1.60	1.34	1.36
sort/q_sort/vector<ValuePair>	1.25	2.42	1.04	1.33
sort/q_sort/vector<ValuePair>/sorted	1.46	1.49	1.25	1.75
sort/q_sort/vector<uint32>	1.42	3.53	0.96	1.40
sort/q_sort/vector<uint32>/sorted	1.27	2.31	1.10	1.72
string<char16_t>/compare	1.85	1.01	1.64	1.45
string<char16_t>/erase/pos,n	1.00	1.00	1.00	1.01
string<char16_t>/find/p,pos,n	1.00	0.96	5.03	4.22
string<char16_t>/find_first_not_of/p,pos,n	0.95	1.17	0.78	1.63
string<char16_t>/find_first_of/p,pos,n	0.97	1.13	0.99	1.00
string<char16_t>/find_last_of/p,pos,n	0.78	1.25	0.46	1.64
string<char16_t>/insert/pos,p	1.01	1.01	1.10	1.06
string<char16_t>/iteration	1.33	1.53	1.07	3.67
string<char16_t>/operator[]	1.45	1.83	1.30	5.16
string<char16_t>/push_back	2.14	6.39	1.22	1.35
string<char16_t>/replace/pos,n,p,n	0.99	1.01	1.18	1.06
string<char16_t>/reserve	1.16	0.92	100.00	100.00
string<char16_t>/rfind/p,pos,n	1.92	2.04	1.08	2.36
string<char16_t>/size	0.92	0.93	0.91	0.85
string<char16_t>/swap	1.94	1.40	1.10	1.38
string<char8_t>/compare	0.99	1.00	0.99	0.91
string<char8_t>/erase/pos,n	1.07	1.00	1.01	1.03
string<char8_t>/find/p,pos,n	0.58	1.00	5.06	3.69
string<char8_t>/find_first_not_of/p,pos,n	0.85	0.94	3.20	1.17
string<char8_t>/find_first_of/p,pos,n	1.34	1.00	10.11	1.45
string<char8_t>/find_last_of/p,pos,n	1.57	1.00	8.27	1.15
string<char8_t>/insert/pos,p	1.15	1.00	0.77	1.05
string<char8_t>/iteration	1.00	1.79	0.86	4.05
string<char8_t>/operator[]	2.23	2.10	1.66	5.66
string<char8_t>/push_back	2.28	5.09	2.28	2.05
string<char8_t>/replace/pos,n,p,n	1.00	1.02	1.03	1.06
string<char8_t>/reserve	1.19	1.70	100.00	100.00
string<char8_t>/rfind/p,pos,n	1.86	2.12	8.70	4.22
string<char8_t>/size	1.08	1.00	0.94	1.12
string<char8_t>/swap	1.95	1.31	1.45	3.99
vector<uint64>/erase	0.99	3.28	1.00	1.02
vector<uint64>/insert	0.99	1.01	1.03	1.12
vector<uint64>/iteration	0.93	1.16	1.15	2.69
vector<uint64>/operator[]	1.39	1.02	1.29	1.70

vector<uint64>/push_back	1.63	2.18	1.02	1.59
vector<uint64>/sort	1.01	1.00	0.92	1.27

21 - int is not a machine word

The int data type was originally intended to represent a machine word. The C99 standard states:

> *A "plain" int object has the natural size suggested by the architecture of the execution environment*
> (C99 6.2.5 p5)

But in practice int is nearly always 4 bytes (int32_t), even on 64 bit and 128 bit platforms.

22 - How do you make a node pool for a linked list?

If you want to make a std::list that uses a fixed-size node pool, there is no portable way to do it, as you don't know what your allocator needs to allocate:

```
template <size_t allocSize, size_t nodeCount>
class NodePool{ };

std::list<int, NodePool<???, 100> > someList;
```

In EASTL you can do this:

```
const size_t allocSize = sizeof eastl::list<int>::node_type;
eastl::list<int, NodePool<allocSize, 100> > someList;
```

23 - Algorithmic optimization vs. hardware optimization

When designing and implementing an algorithm you will achieve higher performance if you consider optimizing for the hardware as well as optimizing for the logical execution. You can come up with a very efficient looking algorithm that in practice executes poorly on practical hardware. When designing an algorithm there are often small changes you can make which don't modify the algorithm but significantly improve its performance. Such effects are often magnified on console and embedded hardware or with weaker compilers, as we've seen with EASTL and console platforms. Here are some things to consider:

Item	Notes
Branch misprediction slows performance.	The compiler optimizes for if statements to evaluate as true, so write your code to follow this when possible. Better yet, write code that avoids branching to the extent possible. See the eastl::min example above.
Some CPU operations are expensive.	For example, most PowerPCs are very slow at shifting an integer by a variable amount.
Table lookups may generate cache misses.	It may be faster to use ((unsigned)(x - '0') <= 9) than to use isdigit(x) because the latter may result in a data cache miss, which is much worse than a branch mispredction.
Function calls may generate instruction cache misses.	Large functions or functions that call a lot of other functions may generate costly cache misses.
Division is much slower than multiplication.	Convert divides into multiplies where possible.
Intermixed floating	The problem is that values need to be shuffled back and forth between

point and integer math is slow.	registers. This problem is exacerbated on PowerPC because such shuffling is required to go through system memory; there is no way to move a value from an integer register to a floating point register.
Compilers may not inline ideally.	Some C++ compilers aren't very good at inlining. Just because a function is declared inline doesn't mean the compiler will do it, even if seems obvious to you that it should.
Some data types are expensive.	For example, 16 bit integers perform significantly slower on many RISC processors than larger types.
Floating point constants may not be cheap.	PowerPC processors necessarily read floating point constants from memory as if they were a variable and thus are slower than you might expect and might generate unexpected cache misses.
Variable writes followed by reads may be expensive.	The PowerPC archicture has a weakness whereby you get penalized by a large number of clock ticks if you write a variable to memory and then shortly thereafter attempt to read it.
Vectorized math may be much more efficient than simpler looking math.	VMX, SIMD, and Cell do parallel floating point operations very fast. Sometimes a vectorized algorithm that loads registers with dummy unused values can outperform a non-vectorized algorithm that superficially looks more efficient.

24 - The swap trick

The vector::reserve function may increase the capacity of a vector, but it will not decrease it. Additionally, the reserve function may set the capacity to be greater than the amount specified by the user. However, the user may want to explicitly reduce the capacity of the vector or may want to set the capacity exactly. The swap trick forces the vector to do what you want:

```
std::vector<int>(v).swap(v); // Clear the excess capacity of v.
std::vector<int>().swap(v);  // Clear v and v's capacity.
```

We would like to propose that this functionality be built into to the C++ standard vector and string classes, as the above is less efficient in some cases, is hard to remember, and is not as flexible as would be an explicit function. See the EASTL set_capacity function. As stated earlier, if a workaround is well-known enough that there are hundreds of pages on the Internet devoted to it and it has its own nickname with the word "trick" in it, it probably should be built into the standard library.

25 - random_access_iterator != pointer

A random access iterator (random_access_iterator_tag) is an iterator which provides the same operations as do ordinary C pointers. However, it lacks one specification that true pointers have: element contiguity. You cannot tell if a given random access iterator refers to congituous elements in memory. A vector has contiguous elements, while a deque generally does not. But you can't tell this from the C++ iterator types. If you want to make optimizations based on memory contiguity, your only option typically is to use type traits. But even so, it still turns out that with some operations compilers optimize better when the compiler sees true pointers instead of a class acting like a pointer. First there is the usual problem of iterator abstractions and type traits being harder for the compiler to inline and optimize. Secondly, the compiler may be able to remove unnecessary divides by keeping the values as byte offsets, but with a random access iterator it is forced to use element offsets.

26 - Memory allocators used in game software

Here we provide a table of allocator types that are typically used in game software. With the exception of the non-local allocator, the other types are commonly found in many other types of software.

Name	Description
generalized allocator	Allocator that is a full generic replacement of malloc/free.
fixed-size allocator	Allocator that allocates blocks of a single size and alignment. Useful for implementing container node pools.
small block allocator	Allocator that is optimized to allocate only small sized blocks. Useful for limiting external fragmentation.
stack allocator (a.k.a. linear allocator, obstack)	Allocator that allocates by simply moving a pointer forward within a chunk of memory. The user cannot free individually allocated blocks.
page protected allocator	Allocator that returns blocks of memory that abut write-protected memory pages and thus trigger hardware exceptions upon writing out of bounds.
non-local allocator	Allocator that is optimized to deal with memory that is slow to read from the CPU, such as graphics memory. The heap management structures are in standard system memory but memory returned to the user resides elsewhere.
handle allocator	Allocator that has a handle-based interface whereby allocated blocks are relocatable when the user unlocks them.
page allocator	Allocates system pages, like mmap under Unix or VirtualAlloc under Windows.

27 - Game platform memory metrics

Here we provide a table showing some popular gaming platforms and their memory as publicly documented on sites such as Wikipedia. The console-based systems have much less memory than currently available desktop and server systems, but additionally are constrained by the lack of swappable memory. In some cases the memory specifications as displayed are oversimplified because the memory is split into multiple types of separate memory systems. A lot of memory in video games is taken by graphics data such as object meshes and textures. In many cases accounts for as much as half of the total memory used by the application.

Platform	System memory	Video memory
Nintendo DS	4 MiB	656 KiB
Sony PlayStation 2	32 MiB	4 MiB
Sony PSP	36 MiB	2 MiB
Nintendo GameCube	40 MiB	3 MiB
Microsoft XBox	64 MiB shared system / video	
Nintendo Wii	88 MiB	3 MiB
Sony Playstation 3	256 MiB	256 MiB
Microsoft XBox 360	512 MiB shared system / video	
PC / Macintosh	128 MiB - 4 GiB	32 MiB - 512 MiB

28 - How game console processors are weaker than desktop and server processors

The following is a list of some of the ways console processors (and their associated memory caches) are weaker than desktop and server processors. This list primarily refers to CPUs as opposed to GPUs. Not all items apply to all game console processors.

- Lack of out-of-order execution.
- Lack of speculative execution.
- Lack of branch prediction capabilities.
- Lack of double-precision floating point math support.
- Lack of unaligned memory read/write hardware and generate unhandled exceptions instead.
- Some instructions are microcoded and thus significantly slower.
- Small caches.
- Less intelligent caches.
- Lack of a memory management unit.
- Lack of mapped memory.

29 - A small problem with LWG #431 option #3

LWG (Library Working Group) issue #431 discusses the issue of swapping containers with unequal allocators. This is a difficult problem with no perfect solution. The submitter suggests three options:

1. This operation is illegal. Perhaps we could say that an implementation is required to check and to throw an exception, or perhaps we could say it's undefined behavior.
2. The operation performs a slow swap (i.e. using three invocations of operator=, leaving each allocator with its original container. This would be an O(n) operation.
3. The operation swaps both the vectors' contents and their allocators. This would be an O(1) operation.

Option #1 is unsatisfying and thus is going to be less popular than the other two.

Option #2 has problems due to the fact that the swapping of containers of equal allocators acts differently from the swapping of containers of unequal allocators.

Option #3 has the problem whereby it requires the allocators to have potentially unknowable lifetime requirements. It may have been implemented in CodeWarrior without reported problems, but it would fail to work with some existing game software. Consider the following, which represents a common programming pattern in game software:

```
struct X {
    X() : myList(BufferAlloc(myBuffer)) { }

    char myBuffer[512];
    std::list<int, BufferAlloc> myList;
};

void DoSomething(X& x1, X& x2) {
    std::swap(x1.myList, x2.myList);
}
```

The swap will compile without warning but the result will most likely be a program crash, due to x1 referring to the memory of x2.

Option #2 has subtle problems, but they can be worked around by the attentive user. However, option #3 has subtle problems that are much harder to work around and which prevent the usage of a common category of high performance programming patterns.

30 - Problems created by global memory allocators

It is mentioned elsewhere in this document that game software often strives to avoid global memory allocation (e.g. global operator new). We take a moment here to point out some of the practical problems in high performance software.

- Built-in global operator new and malloc provide no way to tell where the memory request is coming from and thus all memory requests are necessarily treated equivalently.
- As a result of the above, memory usage per subsystem is hard to measure and enforce because you have no way to categorize it.
- As a result of the above, memory allocations often get haphazardly strewn across the memory space, resulting in poor locality and cache misses.
- Fragmentation occurs more easily because you have no knowledge of temporal allocation locality.
- You need to protect the global heap from thread collisions and false sharing. Solutions such as Hoard help with this but at an often unacceptable wasted memory overhead.
- Allocation speed is slowed by systems having to deal with other systems' free blocks. System X allocates a lot of size N, system Y allocates a lot of size M. Mix these together in free pools and it takes longer to wade through them.
- The user of a given application subsystem can't tune the allocator behavior for its needs, as anything that is done to it modifies the behavior globally for all subsytems.

Acknowledgements

Thanks to Avery Lee, Thorsten Ottosen, Scott Meyers, Cliff Hammerschmidt, Jaap Suter, and Russ Tront for providing useful input on this document. Thanks to the numerous people at EA who have contributed to EASTL, especially Avery Lee and Talin.

References

Note that some of the C++ Standard documents listed below may have been updated since this writing.

Title	Links	Comments
The C language standard	http://www.open-std.org/jtc1/sc22/wg14/	The current C language standard is often referred to as as C99.
The C++ language standard	http://www.open-std.org/jtc1/sc22/wg21/ http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=38110 http://webstore.ansi.org/ansidocstore/product.asp?sku=ISO%2FIEC+14882%3A2003 ISBN: 978-0-470-84674-2	Also known as ISO/IEC 14882 1998-09-01. An update incorporating corrections was published in 2003.
C++ Standard Library Defect Report List	http://www.open-std.org/jtc1/sc22/wg21/docs/lwg-defects.html	Defect reports for the C++ standard library. EASTL largely follows the current defect report conclusions.
C++ Standard Library Wishlist	http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1901.html	A list of potential future C++

		standard library features.
Technical Report on C++ Performance	http://www.open-std.org/jtc1/sc22/wg21/docs/18015.html	An analysis of C++ performance in theory and practice. It is referenced in Appendix item 19.
C++ Library Extensions (a.k.a. TR1)	http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1836.pdf	A report on well-regarded standard library extensions for the next version of C++.
14 crazy ideas for the standard library in C++0x	http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1870.html	A list of proposed standard library extensions, a number of which are functionally equivalent to some of the extensions in EASL.
Universal Character Names in Literals	http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2007/n2170.html	A proposal to add char16_t, char32_t and associated string literals.
Boost	http://www.boost.org/	A library that provides additional functionality beyond the C++ standard library and which has served as a basis for a number of C++09 additions.
A Proposal to Add Move Semantics Support to the C++ Language	http://std.dkuug.dk/jtc1/sc22/wg21/docs/papers/2002/n1377.htm	Move semantics allow for some operations to be more efficient. Move semantics are a superset of what EASTL calls "trivial relocate."
Towards a Better Allocator Model Pablo Halpern	http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1850.pdf	A proposal for a new C++ standard allocator model which has a number of similarities in both motivation and design to the EASTL allocator mode.

Intrusive C++ Containers	http://freenet-homepage.de/turtle++/intrusive.html	A proposal for intrusive containers in the C++ standard. The proposal is similar in motivation to EASTL intrusive containers and intrusive smart pointers.
Adding Alignment Support to the C++ Programming Language	http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2007/n2165.pdf	A proposal to add explicit alignment support to the C++ standard.
Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library	http://www.awprofessional.com/bookstore/product.asp?isbn=0201749629&rl=1 http://www.aristeia.com/ ISBN: 978-0201749625	Recommendations for efficient use of STL.
Extended STL, Volume 1 Matthew Wilson	ISBN: 978-0321305503	Includes a description of contiguous_iterator that is similar or identical to the one defined in EASTL.
The Memory Fragmentation Problem: Solved? Mark S. Johnstone, Paul R. Wilson	http://citeseer.ist.psu.edu/johnstone97memory.html	This paper says that modern heaps solve the memory fragmentation for a large class of programs (section 6, paragraph 5). For the game software class, modern heaps improve memory fragmentation but by no means solve the problem. This is fundamentally due to alignment requirements and the fact that games run with nearly all system memory allocated and little free memory.

3 - Curiously Recurring Template Pattern	http://en.wikipedia.org/wiki/Curiously_Recurring_Template_Pattern	A parent class is templated on a subclass of itself.
Borland template model Cfront template model	http://gcc.gnu.org/onlinedocs/gcc/Template-Instantiation.html	Where is templated code compiled to? There are two common mechanisms: Borland and Cfront.
Small Memory Software: Patterns for Systems with Limited Memory	http://www.smallmemory.com/ ISBN: 978-0201596076	How to write write software for limited memory systems.
Empty base class optimization	http://www.cantrip.org/emptyopt.html	The C++ standard requires that all classes have a size of at least one byte, even if the class is empty. However, there is a way to make an empty class act as if its size is zero.
C++ Exception Handling	http://portal.acm.org/citation.cfm?id=614253	This document describes exception handling implementations currently in use.
GCC inlining problems	http://groups.google.com/group/comp.lang.c++/browse_frm/thread/b74eed16bd48d42e http://groups.google.com/group/fa.linux.kernel/browse_frm/thread/1861b2634cdfa68a/ http://www.pixelglow.com/lists/archive/macstl-dev/2005-September/000154.html	Some discussions of GCC inlining problems, which apply to GCC versions up to at least GCC 4.0.
Swapping containers with unequal allocators	http://www.open-std.org/jtc1/sc22/wg21/docs/lwg-active.html#431 http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1599.html	Library Working Group issue #431
Parkinson's Law	http://en.wikipedia.org/wiki/Parkinson%27s_law	Work expands so as to fill the time available for its completion. However, Parkinson's observation pertained to the

		development of bureaucracies and inefficiency.
The Garbage Collection Page	http://www.cs.kent.ac.uk/people/staff/rej/gc.html Book ISBN: 0-471-94148-4	A comprehensive guide to garbage collection, links to garbage collection references, and an associated book: *Garbage Collection: algorithms for automatic dynamic memory management.*
The Unicode Standard	http://www.unicode.org/	The definitive guide to the Unicode Standard.