Jan 2016

msetl is a c++(11) library intended to help make c++ code safer by providing compatible substitutes for common c++ data types. Currently the library includes safer substitutes for native pointers, int, size_t, bool, and std::vector<>. The library is particularly suitable for making existing and legacy code safer.

Probably the best way to start is to download the library and peruse the example file, msetl_example.cpp. It may help to step through it with your favprite interactive debugger.

Q: How easy is it to learn and use?

A: Very easy. Msetl data types are generally direct compatible substitutes for their corresponding native and stl counterparts. Converting existing code is mostly just a matter of “search and replace”.

Q: What are the safety benefits?

A: Well, the ultimate goal is to make c++ code as safe as, say java. The biggest benefit probably comes from replacing native pointers with msetl's “registered” pointers. Registered pointers should throw an exception on any attempt to access invalid memory. The replacement for std::vector<> and its iterators is also very safe – bounds checked and memory managed. The int, size_t and bool replacements have default initialization values.

Q: Anything else I should know?

A: A couple of notes about compling: g++(4.8) requires the -fpermissive flag. And in some cases msvc may complain about something about too many “inline” items. The error code will direct you to instructions to resolve the issue.

That's pretty much all you need to know. The rest of this document is just details and discussion.

Comtemplating msetl's registered pointers:

The benefits of using registered pointers to replace native pointers in legacy code is clear. But in a world that already has std::shared_ptr and std::unique_ptr, is there a reason to use registered pointers in new code? Well I think one can make some arguments for it.

First let's note that registered pointers are inherently not “thread safe”. But from a language safety point of view, this is not really an issue because the general practice of sharing of objects between threads is inherently dangerous and, from a safety perspective, should only be done through safety-proven, well tested mechanisms.

Let's also recognize that std::shared_ptr in particular, is a great data type from a safety perspective, and for heap allocated objects, the performance overhead is quite reasonable. That said, std::shared_ptr and std::unique_ptr assure us that objects will “stay alive” at least as long as the smart pointers that own them are alive (and still own them). This means that if you make a programming mistake, instead of the smart pointer possibly pointing to a deceased object, the object might remain alive longer than intended. I suggest that this is not always the preferred consequence of a programming mistake. Sometimes it may be important that an object be terminated on schedule, even if that results in an exception being thrown as a result of attempting to access a deceased object.

Also, the fact that std::shared_ptr and std::unique_ptr eliminate the redundant management of pointer and object lifespans is often touted as a benefit. But having the programmer express the intended lifespan of an object and it's pointers separately is a redundancy that can sometimes help catch programming bugs. For example, any situation that requires the assistance of std::weak_ptrs might be one where you're better off using any redundancy that might help make sure you got it right.

And then there's the fact that registered pointers can point to “stack allocated” objects. I'm not sure how often one needs a pointer to stack allocated objects these days, but, for example, if you want to write a function that takes as a parameter a safe reference to an object that could have been allocated either on the heap or the stack, registered pointers will do the trick.

If you are concerned with performance, registered pointers should do well as far as “safe” pointers go. Consider the case of one std::shared_ptr pointing to one object, this involves data structures at three different locations in memory - the pointer, the object, and a reference counting object. The shared_ptr could sometimes be allocated on the stack, but the other two are on the heap, so they will be using at least two more cpu cache lines. These days performance is a lot about the cpu cache. In comparison, one registered pointer pointing to one object involves data structures at only two locations in memory – the pointer and the object. The object is slightly enlarged because the pointer-tracking data structure has been grafted on to it, but it's all at the same location. If the number of pointers to the object exceeds a fixed amount (the default is currently two), then the pointer tracking data is moved to the heap, but presumably this will be a minority of cases. Furthermore, both the registered pointer and the object could be allocated on the stack (which is presumably already in the cache), so the number of extra cache lines used could very well be zero. So in most cases, the number of extra cache lines used will be less than the shared_ptr would use. In the worst cases it would use more.

Registered pointers are essentially a complete functional replacement for native pointers (minus pointer arithmetic, by intent of course). I'm not sure that std::shared_ptr and std::unique_ptr are.

More about msetl's vectors:

A couple of cases where safety cannot be guaranteed is when using the constructor or insert() member function that takes "non-safe" (standard stl) iterators or pointers as parameters. For example:

```
#include "msetl.h"

#include <iostream>

...

	{

		double a1[3] = { 1.0, 2.0, 3.0 };

		double *d_pointer1 = &(a1[0]);

		double a2[3] = { 4.0, 5.0, 360 };

		double *d_pointer2 = &(a2[0]);

		mse::mstd::vector<double> v1;

		//v1.insert(v1.begin(), d_pointer1, d_pointer2); /* not good */

		/* std::vector supports "naked" pointers as parameters to the
		insert() member

		function so mse::mstd::vector does also. Unfortunately there is no
			way to ensure

		that the naked pointer parameters have valid values. */


		mse::mstd::vector<double> v2 = { 1.0, 2.0, 3.0 };

		mse::mstd::vector<double> v3 = { 4.0, 5.0, 360 };

		mse::mstd::vector<double> v4;

		try {

			v4.insert(v4.begin(), v2.begin(), v3.begin());

		}

		catch (...) {

			std::cerr << "expected exception";

			/* The exception is triggered by a comparision of
				incompatible "safe" iterators. */
```

```
        }
    }
```

mse::mstd::vector iterators are also safe against reference of deallocated vectors. That is, kind of like java, deallocation of mse::mstd::vectors are intrinsically "managed" (delayed if necessary) so that you can't accidentally end up in a situation where an iterator is pointing to an mse::mstd::vector that has already been deallocated/deleted/freed:

```
    {
        typedef mse::mstd::vector<int> vint_type;
        mse::mstd::vector<vint_type> vvi;
        {
            vint_type vi;
            vi.push_back(5);
            vvi.push_back(vi);
        }
        auto vi_it = vvi[0].begin();
        vvi.clear();
        /* At this point, the vint_type object is cleared from vvi, but it
            has not been deallocated/destructed yet because it
        "knows" that there is an iterator, namely vi_it, that is still
            referencing it. At the moment, std::shared_ptrs are being
        used to achieve this. */
        auto value = (*vi_it); /* So this is actually ok. vi_it still points
        to a valid item. */
        assert(5 == value);
        vint_type vi2;
        vi_it = vi2.begin();
        /* The vint_type object that vi_it was originally pointing to is now
        deallocated/destructed, because vi_it no longer
        references it. */
    }
```

Arguably, it would be better to have the iterator throw an exception when trying to reference a container that has already been "deallocated/freed", rather than keep the container in memory and treat references to the container as valid. With this in mind, do not assume that such references will continue to be treated as valid in future versions of the library.

mse::mstd::vector does have some overhead, so it's more appropriate for use in situations where safety and compatibility are prioritized over performance and memory efficiency.

Note that at the moment, mse::mstd::vector supports a mix of the c++98 and c++11 standards. It does not quite support the entire c++11 standard yet. In practice, it shouldn't be a problem for the vast majority of cases.

Multi-modify vectors:

Stl iterators were basically designed as thinly veiled pointers. As a result, vector iterators behave differently from, say, list iterators with respect to insertions and removals on the container. For example, let's say you have an iterator pointing to the fifth element of a container and you then remove the third element. If the container is a list, then the iterator remains valid and now points to the fourth element. However, if the container is a vector, the iterator becomes "invalid".

In the case of insertion it's even worse. Upon any insertion, any vector iterator may become invalid "if a reallocation occurs".

This makes vector iterators less useful (or more tedious), and (in our opinion) more error prone than list iterators. So the mestl library also provides a version of the vector class that supports an iterator that behaves like a list iterator. That is, an iterator that remains valid (and pointing to the same item) after a container modification, as long as the item it points to has not been removed. This also means that any reallocations are transparent to the user. This new version of the vector is "mse::msevector", and it's "list-style" iterator is "mse::msevector::ipointer" ("mse::msevector::cipointer" for the "const" version).

mse::msevector also supports two other iterator types: "mse::msevector::iterator" for (high performance) compatibility with stl code, and "mse::msevector::ss_iterator_type" which is essentially just a "safe" version of mse::msevector::iterator.

```
	{
		mse::msevector<int> v = { 1, 2, 3, 4 };
		mse::msevector<int>::ipointer ip_vit1(v);
		/*ip_vit1.set_to_beginning();*/ /* This would be redundant as
		       ipointers are set to the beginning at initialization. */
		ip_vit1.advance(2);
		assert(3 == ip_vit1.item());
		auto ip_vit2 = v.ibegin(); /* ibegin() returns an ipointer */
		v.erase(ip_vit2); /* remove the first item */
		assert(3 == ip_vit1.item());
		ip_vit1.set_to_previous();
		assert(2 == ip_vit1.item());
	}
```

mse::msevectors are slightly less safe than mse::mstd::vectors in that mse::msevector iterators are not safe with respect to reference of deallocated vectors (i.e. java-style delay of object destruction). In exchange however, mse::msevectors have less overhead and should be a little bit faster. If you're looking to improve safety with very little compromise with respect to compatibility and not too much compromise with respect to performance, mse::msevectors may be the way to go.

The mestl library provides a third version of the vector class, mse::ivector, which is very safe, like mse::mstd::vector, but only supports the mse::ivector::ipointer "list-style" iterators. mse::ivector is more appropriate for situations where safety is a priority and you feel (as we do) that stl vector iterators are intrinsically problematic. mse::ivector::ipointers can still be used with stl algorithms (like std::sort).

Safer c++:

The msetl was created to support a new style/convention of c++ programming that attempts to approach a "java like" level of safety.

Some potential dangers in c++:

- native (aka "naked") pointers in general, including native arrays and function pointers

- references to deallocated variables/objects

- uninitialized variables/objects

- std::unique_ptr doesn't check dereferences

- std::shared_ptr used on something other than a newly allocated pointer

- unsafe stl in general

- non-virtual destructors

- reinterpret casts (and unions)

We are not going to fully define a "Safer c++" programming convention here, but considering the above list, some of the rules of a "Safer c++" programming convention might be:

i) A prohibition on the use of native pointers in general, including native arrays and function pointers. Smart pointers like std::shared_ptr or mse::TRegisteredPointer can be used instead.

ii) A prohibition on the use of types that do not have a default initialization value. The msetl library provides substitutes for common primitive types.

iii) A prohibition on the use of reinterpret casts (and unions).

iv) A prohibition on the use of std::unique_ptr. mse::TRegisteredPointer can be used instead.

v) A prohibition on declarations of std::shared_ptr that are not in the form of "std::shared_ptr<> x = std::make_shared<>(...);"

The stl issue should be solved by future development of "safe" implementations of the stl. Visual Studio already supports "checked iterators".

C++ references are potentially as dangerous as pointers when applied to dynamically allocated objects. The problem with prohibiting them outright is that they are just so convenient. Although not an exact substitute, perhaps consider using mse::TRegisteredPointers instead.

Why not just use java?:

Java is probably a good option in most cases. It might be considered to have a couple of potential drawbacks though:

i) Reliance on garbage collection:

- a) Maximum memory use is not deterministic (or deterministically bounded).
- b) Potential gc "clean up pauses" are not compatible with some real time applications.
- c) Non-deterministic destructors. This means that raii programming is not supported. The raii paradigm is arguably very useful in reducing many "higher level" programming errors.

ii) Reliance on the jvm/jre:

- a) The jvm/jre require resources (memory) at runtime.
- b) There could be (and have been) security bugs in popular implementations of the jvm/jre.