A brief introduction to C++'s model for type- and resource-safety

Bjarne Stroustrup (Morgan Stanley)
Herb Sutter (Microsoft)
Gabriel Dos Reis (Microsoft)

Abstract

You can write C++ programs that are statically type safe and have no resource leaks. You can do that simply, without loss of performance, and without limiting C++'s expressive power. This model for type- and resource-safe C++ has been implemented using a combination of ISO standard C++ language facilities, static analysis, and a tiny support library (written in ISO standard C++). This supports the general thesis that garbage collection is neither necessary nor sufficient for quality software. This paper describes the techniques used to eliminate dangling pointers and to ensure resource safety. Other aspects – also necessary for safe and effective use of C++ – have conventional solutions so they are mentioned only briefly here.

The techniques and facilities presented are supported by the Core C++ Guidelines [Stroustrup,2015] and enforced by a static analysis tool for those [Sutter,2015].

This is a minor revision of the October 2015 version. There are no significant technical changes, but in response to reader comments a few explanations have been improved and a few examples added. Please note that this is an introduction rather than a detailed technical report. For details, see [Sutter,2015] and our enforcement tools.

0. Overview

This paper is organized like this

- *1. The problem*: Resource leaks and memory corruption
- *2. Constraints on any solution*: Generality, performance , and compatibility
- *3. Memory safety*: Eliminate dangling pointers and access to deleted objects
- 4. Resource safety: Eliminate resource leaks
- *5. Other problems*: Concurrency, casts, range errors, etc.
- *6. Historical note*: Where did the solutions come from?

1. Introduction

Consider

```
void use(FILE* p);

int some_code(int x)
{
        FILE* f = fopen("foo","w");
        if (f==0) error("cannot open file");
        if (0<x) {
                use(f);
```

```
            // … do a lot of work …
        }
        else
            return -1;
        long pos = ftell(f);                // read through f
        // …
        fprintf(f, "Die, die, die!\n");     // read and write through f
        fclose(f);
        return 0;
};
```

This looks innocent enough, but the definition of **use()** is

```
void use(FILE* p)
{
        // …
        free(p);
        // …
}
```

Thick books could be written about what – in general – is wrong with this simple code. Experienced C and C++ programmers would quickly eliminate the problems in this small example, but we have decades of experience showing that the kind of errors exhibited by this simple example are extremely hard to eliminate in large code bases. We deliberately used a pure C example to emphasize that the problems we attack are long standing (40+ years) and not limited to C++. They are in various forms shared by all languages with manual resource management (such as Java and C#). C++'s facilities are part of our solution, rather than part of the problem.

Our aim is to show how to systematically prevent the classes of errors demonstrated by this small code sample, without compromising the fundamental strengths of C++.

Consider the fundamental errors in the example:

1. *Resource leaks:* If this code takes the early **return**, the file is never closed and the memory and operating system resources are not released.
2. *Access through an invalid pointer:* If this code doesn't take the early **return**, **free()** makes the pointer invalid (assuming the pointer points to a **FILE** that was dynamically allocated using **malloc()**). The memory pointed to will most likely be reused. In particular, the **ftell()** attempts to read through the dangling **FILE*** and accesses memory that no longer holds a **FILE**. This is a type violation.
3. *Memory corruption:* If this code doesn't take the early **return**, **free()** makes the pointer invalid (assuming the pointer points to a **FILE** that was dynamically allocated using **malloc()**). The memory pointed to will most likely be reused. In particular, a write through an invalid pointer, as in the **fprintf()**, is likely to overwrite memory that hold parts of completely unrelated objects. This is a type violation.
4. *Confusion between static and dynamic objects:* Somewhere in the documentation, it says whether the **FILE*** returned by **fopen()** points to a **FILE** that is statically or dynamically allocated. However, compilers do not read manuals, and calling **free()** on a statically-allocated object or

failing to call **free()** on a dynamically-allocated object leads to serious errors (especially in large, complex systems).

5. *Type-incorrect deallocation:* Somewhere in the documentation, it says that you should use **fclose()** rather than **free()** to properly close a file. However, compilers do not read manuals, and there is nothing in the type system that will catch this error. Even if the **FILE*** returned by **fopen()** points to a **FILE** that was allocated using **malloc()**, calling **free()** will not release OS resources (and maybe not even all memory, if there were additional allocations used in the representation of the file handle).

More than forty years of C and other languages prove that (given sufficient effort) these problems can often be handled "by hand" and through extensive testing. However, experience also shows that "being careful" and extensive testing is a time sink. Often, these efforts still leave serious errors in the code, causing crashes and security violations. We must do better.

Problems [2-4] are results of explicit memory release and the presence of multiple storage classes. They can be solved by using garbage collection, but problems [1] and [5] remain whether or not we use GC and it requires careful coding to avoid such problems. The problem is that GC reclaims memory only. Resources are not just memory; there are also file handles, thread handles, locks, sockets, and many other "non-memory resources." Adding finalizers to GC partially addresses this problem but even something as simple and well-known as calling **fclose()** at GC time is almost always unsuitable. However, finalizers:

- are executed at a nondeterministic time
- can be executed in the wrong context (e.g., on a system thread; this can inject deadlocks with program threads and possibly open security holes);
- are not guaranteed to run at all, so there is no guarantee the file will ever be closed as long as the program is running;
- can lead to excessive resource retention when the GC is not run frequently; and
- can make GC unnecessarily expensive if a finalizer can place a reference to an object it was supposed to destroy in an accessible location ("resurrection").

Consequently, the use of finalizers for resource cleanup is now actively discouraged in the major GC environments that support them, leaving the release of non-memory resources as a manual activity.

C++ has always had the language mechanisms to trivially cope with problems [1] and [5] ("resource safety"). Since 1988, techniques for doing so have been widely used. Since 1998, they have been an integral part of the ISO C++ standard library.

On the other hand, problems [2], [3], and [4] ("memory safety", "dangling pointers", or "lifetime safety") have been a constant worry and a major source of errors in real-world C++ code.

This article is a popular/informal description of a solution to these problems. A more formal and detailed description can be found in a companion paper [Sutter, 2015]. The techniques outlined here are made concrete by the Core Guidelines [Stroustrup, 2015] and their support tools.

2. Design constraints

From its inception, C++ was developed as a series of improving approximations to a set of ideals. The ideals included – and still include:

1. Perfect static type safety
2. Automatic release of general resources (perfect resource safety)
3. No run-time overhead compared to good hand-crafted code (the zero-overhead principle)
4. No restriction of the application domain compared to C and previous versions of C++
5. Compatibility with previous versions of C++ (long-term stability is a feature)

Perfectly matching these ideals (and more; see [Stroustrup,1994]) seemed impossible. However, this paper explains how we meet the first two ideals without damaging the next three using a combination of coding guidelines, static analysis, and simple library support.

Our solution has four parts:

1. *Eliminate dangling pointers*: We use simple, local, static analysis and a low-level **owner** annotation to detect dangling pointers.
2. *Eliminate resource leaks*: We use resource management abstractions (e.g., **std::vector** and **std::shared_ptr**) to handle resources anchored in a scope and a low-level owner annotation to statically prevent necessary **delete**s from being "forgotten."
3. *Eliminate type errors*: Severely limit the use of explicit type conversion (cast) and discourage the use of **union**s.
4. *Eliminate out-of-range access and access through the null pointer*: Use a range abstraction (**span**) and a "not null" abstraction (**not_null**) for pointer that must not hold the **nullptr**.

Each of these techniques have decades of background in C and/or C++. Our combination is novel and adds up to a general, scalable solution that offers guarantees. This paper discusses the first two parts. The last two parts are not particularly new and require little discussion.

3. Memory safety

Dangling pointers is the darkest nightmare of a language that relies on pointers and explicit release of resources. The C++ constructor/destructor mechanism was introduced to handle the resource management part of that problem (see §4), but pointers with their obvious opportunities for serious problems remain. Consider a few more examples:

```
int* glob;

void squirrel(int* q)
{
        // …
        glob = q;
}

void g()
{
        int* p = new int{7};
```

```
        squirrel(p);
        delete p;
        // …
        *glob = 9;        // write to deallocated memory
}
```

Here, the ***glob=9** writes to deallocated memory. The result is often corruption of apparently unrelated objects (later allocated in that space). It is an example of a large class or problems where some code squirrels away a pointer for later use, and then the pointed-to object is deleted. Such errors are hard to find and cause serious problems.

Consider also:

```
void g2()
{
        int* p = new int{7};
        squirrel(p);
        delete p;
        // …
        delete glob;      // double deletion
}
```

The result is often a crash or the corruption of the free store (heap) management mechanism. And finally:

```
void g3()
{
        int x = 7;
        int* p = &x;
        squirrel(p);
        // …
        delete glob;      // delete of local variable
}
```

Again, the result is often a crash or the corruption of the free store (heap) management mechanism.

A global variable is just one way a pointer can escape out of the scope where it is valid. Other ways include smart pointers, containers of pointers, objects holding pointers. We track them all.

These errors are not subtle, but in a large program they can be easily made, are hard to find, and can have catastrophic results. We have 40+ years of industrial practice deeming this a hard set of problems. Our general solution is conceptually simple, cheap to enforce using static analysis, and free of run-time overheads.

We say that a program is memory safe if every allocated object is deallocated (once only) and no access is done through a pointer (or reference, iterator, or other non-owning indirection) to an object that has been deleted or gone out of scope (and thus technically isn't an object any more – just a bag of bits).

To be type safe, we need memory safety so that an object cannot be accessed through a dangling pointer. Such access doesn't obey type rules. For example:

```
int* pi = new int{7};
int q = pi;
delete pi;
// …
double* pd = new double{7.7};
*q = 8;
double d = *pd;
```

In a type-safe program, **d** would obviously have the value **7.7**. However, ***pd** might have been allocated in the space used for ***pi**. Then, ***q** overwrites **7.7** with something that is not a properly normalized floating-point number and **d** will get a garbage value (in one simple test, **d** became **-7.84591e+298**). Thus, memory safety is a prerequisite for type safety.

Furthermore, to be perfectly type safe, a program must be free of range errors (access beyond the end of an array), free of access through the null pointer, etc. (§6).

3.1. Owners

Consider first the problem of deleting objects. An object constructed using **new** on the free store (dynamic memory, heap) must be destroyed using **delete**. On the other hand, statically allocated objects (e.g., global variables), objects on the stack (local objects), and objects that are part of an enclosing object (class members) should never be explicitly **delete**d because they will be implicitly destroyed.

Objects can also be allocated using **malloc()** and deallocated using **free()** (or similar functions), but the techniques described for **new** and **delete** apply to those also, so we will not discuss **malloc()**/**free()**.

- To avoid confusing pointers that must be **delete**d from pointer that must not, we introduce the concept of an owner. An **owner** is an object containing a pointer to an object allocated by **new** for which a **delete** is required.

Every object on the free store (heap, dynamic store) must have exactly one owner. If there are two pointers to an object on the free store, only one can be the owner. An object not on the free store does not have an owner.

Colloquially, we also refer to an object that directly or indirectly holds an owner as an owner (e.g., a **vector** of pointers, a **map**, and a **shared_ptr**); such objects are handled as resource managers (§4).

We focus on pointers. A reference is a restricted form of a pointer with some added syntactic sugar, so our techniques for pointers also deal with references.

3.2. A dynamic model

It is easy to implement the ownership model dynamically:

- attach an ownership bit to every pointer
- set the ownership bit when a value returned from **new** is assigned
- clear the ownership bit when a pointer not returned from **new** is assigned
- use **delete** whenever a pointer with the ownership bit is set goes out of scope

- use **delete** before overwriting a pointer with an ownership bit set

Assigning an owner to a non-owner has no effect on ownership; that simply creates an ordinary pointer and the original owner is still the unique owner.

Assigning an owner to another owner, **delete**s the target and makes it hold an ordinary pointer to the object owned by the source.

Such a design can handle assignment of a non-owner to an owner in a variety of ways that preserves the "exactly one owner" rule.

Unfortunately, this model is not practical in the context of C++: It doubles the size of pointers (or requires stealing bits out of machine addresses) and adds extra computation for pointer use. That violates the zero-overhead principle. It also breaks link compatibility with every existing C and C++ program.

This was as far as Bjarne got a few years ago. That work was never published because he considered the solution useless: it implies inefficiency and incompatibility. It is, however, useful for thinking about ownership.

3.3. A static model

Fortunately, we can implement a static variant of that dynamic model that implies no run-time overhead and no ABI breakage:

- Mark every **T*** that is an owner as **owner<T*>** in the source code.
- Let **new** return an **owner<T*>**
- Make sure that every **owner<T*>** is **delete**d or transferred to another **owner<T*>**.
- Never assign a plain **T*** to an **owner<T*>**

You can assign an **owner<T*>** to a plain **T***. The result is not two owners, but simply an owner and a non-owning pointer to the same object. Because we eliminate the possibility of dangling pointers, this is entirely safe and reasonable.

If you assign one **owner** to another, you must **delete** the target and make sure the source is neither used again nor **delete**d.

The **owner** template could be a real type, but to ensure ABI compatibility, it is not. It is simply an alias to help static analysis tools:

```
template<typename X> using owner = X;
```

This **owner** is part of the Guideline Support Library [Sutter, 2015], often called GSL. Given that, tools can ensure that an **owner<T*>** is used appropriately and that a plain **T*** is not treated as an owner. For example:

```
void f(owner<int*> p, int* q)
{
        delete q;       // bad!
        // no delete p; also bad
}
```

Importantly, an **owner** doesn't do anything. In particular, it does not have operations that execute **delete**. It simply allows static analysis to detect when the programmer fails to follow the model. For example:

```
void g(owner<int*> p, int* q, owner<int*> p2)
{
        p = q;          // bad: q is not an owner
        delete q;       // bad: q is not an owner
        q = p;          // OK: q points to the object owned by p
        p = p2;         // bad: no delete of owner p
        *q = 7;
        // bad: two owners of *p2
        // bad: no delete of p2
}
```

This could be fixed like this:

```
void g(owner<int*> p, int* q, owner<int*> p2)
{
        // p = q;       // bad: q is not an owner
        // delete q;    // bad: q is not an owner
        q = p;          // OK: q points to the object owned by p
        delete p;       // needed: we are about to overwrite p
        p = p2;         // OK: we just deleted p
        p2 = nullptr;   // so that there are not two owners of *p2
        // *q = 7;      // bad: assignment through now dangling pointer
        delete p;       // needed: we are about to leave g()
}
```

This kind of code is messy and tricky to get right, but easily enforced by static analysis. We use **owner** only where ABI compatibility with older code is essential and in the implementation of proper ownership abstractions (e.g., **vector** and **unique_ptr**). Where feasible, prefer proper ownership abstractions to simplify programming (§4).

3.4. Static model limitations

The dynamic model has the ownership attribute firmly attached to a pointer; in the static model, ownership is attached to a pointer's type. This implies that (using the static model), we cannot mix owners and non-owners in a homogenous container (e.g., a **vector** or an array). For example, consider an ordinary piece of (bad) C++ (not using **owner**):

```
vector<X*> glob;

X* mix(X* p1, int i)            // we don't know if p1 is an owner
{
        X x = 7;
        X* p2 = &x;             // p2 is not an owner
        X* p3 = new X{9};       // p3 is an owner
```

```
        glob = vector<X*>{p1,p2,p3};
        if (i%2)
                return p2;
        else
                return p3;
}
```

In plain ISO C++, we would be lost. Which elements of **glob** should be deleted? Should the result of a call of **mix()** be deleted?

Using the dynamic model, we can look at the elements of **glob** to see which ones should be **delete**d and we could examine the result of a call of **mix()** to see if a **delete** was needed. The **delete**s could be made implicit.

Using the static model, this code is first rejected because the **vector** elements disagree about ownership and because the return statements disagree about ownership. Next we find that this code is not easy to repair: There is no reasonable way of writing C++ that mixes ownership in containers or return statements and still avoid leaks and erroneous **delete**s. To repair this code we have to decide whether **glob** should be a **vector<owner<X*>>** or a plain **vector<X*>** and whether **mix()**'s return type should be **owner<X*>** or a plain **X***. Once we have decided, we can correct the program appropriately.

The basic rule is that we must statically (through the type system) know whether a pointer is an owner or not. Similarly for a reference:

```
int x = 7;
int& p = (i)? x : *new int{9};              // bad
owner<int&> q = (i)? x : *new int{9};       // bad
```

This is a limitation of current (unannotated) C++. There are a few real-world uses of "mixed ownership." To work, they emulate the dynamic model. For example:

```
vector<X*> glob;
vector<bool> glob_owner;

pair<X*,bool> mix(X* p1, bool own, int i)
{
        X* x = 7;
        X* p2 = &x;
        X* p3 = new X{9};
        glob = vector<X*>{p1,p2,p3};
        glob_owner = vector<bool>{own,false,true};
        If (i%2)
                return {p2,false};
        else
                return {p3,true};
}
```

Now the users of **mix()** can examine the hand-crafted ownership bits to ensure proper deletion. For example:

```
void cleanup()
{
        for (size_t i = 0; i<v.size(); ++i)
                if (glob_owner[i])
                        delete owner[i];
}
```

Such code is either brittle or encapsulated in suitable abstractions.

When we enforce the static ownership model, this code will not compile: Either **glob** holds **owner<X>**s and we fail to **delete** some or it holds plain **X***s and we "forgot" to keep **owner**s around for all objects. To deal with that, we provide a mechanism for locally suppressing checking. Such suppression is intended to be used sparingly in trusted implementations of ownership abstractions. They resemble casts (explicit type conversions) in that they manually override the type system.

As implemented, our ownership model considers a pointer as pointing to a full object (e.g., to the "outermost" object enclosing by-value data members) or to the beginning of an array. As writers of serious optimizers for C++ and garbage collectors know, there are cases where an object is kept alive by a pointer into the middle of an object or a pointer to one-beyond-the-end of an object. For example:

```
int* tricky(int n)
{
        int* p = new int[n];
        // …
        return &p[n/2];
        // …
        return &p[n];
}
```

We handle most such cases, but for a few tricky examples we have to be conservative. Our general strategy is "better safe than sorry."

3.5. Overuse of owner

If we used **owner** and static analysis for conventional C-style code littered with pointers and subtle uses of pointers the result would be unmanageable. We would simply need so many **owner** annotation that those annotations would themselves become a nuisance and a source of errors. This has been seen in languages depending on annotations (such as Cyclone and Microsoft's SAL annotations for C and C++ code) and our own experiments. To be manageable on an industrial scale, **owner** annotation must be rare. We ensure that by "burying" them in proper ownership abstractions, such as **vector**s and **unique_ptr**s. For example, the basic representation of **vector** may look like this:

```
template<SemiRegular T>
class vector {
        owner<T*> elem;      // the anchors the allocated memory
        T* space;            // just a position indicator
```

```
        T* end;             // just a position indicator
        // …
};
```

Systematic use of containers and similar "ownership abstractions" can limit the explicit use of **owner** in application code to examples where we need to maintain ABI compatibility with older code that use pointers explicitly in its interfaces.

We assert that in good code **owner** annotations are rare, so that we can assume that a plain pointer (a **T***) is not an owner. Think: 1000 plain pointers for each **owner**.

3.6. Pointer safety

To use a pointer safely, a pointer may not outlive the object it points to. More precisely, it may not be dereferenced after the object it pointed to was destroyed. Ignoring for the moment multithreading (§5), this can be achieved by ensuring that

- a pointer is not used in a function after it has been invalidated (it's owner has bee deleted)
- a pointer does not point into a scope that has been exited from.

The simplest example of this rule has been (incompletely) enforced by compilers for decades:

```
int* f()
{
        int x = 8;
        return &x;      // bad: pointer to local variable that's about to disappear
}
```

We use a simple set of rules, including:

- Don't transfer a pointer to a local to where it could be accessed by a caller
- A pointer passed as an argument can be passed back as a result
- A pointer obtained from **new** can be passed back from a function as a result

Basically, this means that if you visualize a stack growing up from caller to callee, pointers cannot be passed down the stack to below the object they point to. For example:

```
int*  f(int* p)
{
        int x = 4;
        return &x;              // No! would point to destroyed stack frame
        // …
        return new int{7};      // OK (sort of: doesn't dangle, but returns an owner as an int*)
        // …
        return p;               // OK: came from caller
}
```

As for the rules for owners, the rules for scope also apply to references and to objects containing pointers or references. For example:

```
vector<int*> glob;      // container of pointers

vector<int*> f()
{
        int x;
        glob.push_back(&x);    // Bad: pointer to local placed in global container
        vector<int*> v {&x};
        return v;              // Bad: v contains pointer to local
}
```

3.7. Scope limitations

As for ownership, the simple static model of scope imposes a few restrictions of use. For example:

```
X glob;

class Silly {
        X loc;
        X* p { &glob };
        X* mem(int x) { return (x<0)?&loc:p; }   // carefully obscure local and global
};

X* glob_p;

Silly make()
{
        Silly res;
        glob_p = res.mem(1);   // bad: could pass pointer to local to global
        return res;            // bad: could pass pointer to local out of function
}

void use()
{
        Silly r = make();
        *glob_p = 7;
}
```

The use of **glob_p** in ***glop_p** is safe because **mem()** was passed a **1**, so **glob_p** points to global variable. However, you need global analysis tracking the value of the argument to **Silly** to see that. Our rules just see a pointer coming from a local object (**res**) being passed out of scope and deems it bad. More realistic classes, such as **vector**, do not obscure the source of the pointers it returns (the way **Silly** does) so something like a pointer to a **vector** element is no problem – as long as the **vector** is alive, it points to an object owned by the **vector**. However, examples as subtle as **Silly** are also known to be error-prone.

A pointer validly derived from a valid pointer is also valid. Consider:

```
int* find(int* first, int* last, int x)
{
```

```
        for (; first!=last; ++first)
                if (*first==x)
                        return first;
        return last;
}
```

This is (obviously) essential for the STL algorithms.

3.8. Pointer invalidation

A pointer can become invalid because the object to which it refers is destroyed or reallocated. Consider:

```
vector<int> v = { 1,2,3 };
int* p = &v[2];
v.push_back(4);         // v's elements may move
*p = 5;                 // bad: p might have been invalidated
int* q = &v[2];
v.clear();              // all v's elements are deleted
*q = 7;                 // bad: p is invalidated
```

A tool that knows the semantics of **vector** can determine that **v.push_back(4)** invalidates **p** (according to the definition in the ISO standard) because the elements of **v** may be reallocated to make room for **4**. Similarly, **v.clear()** invalidates **q** because it deletes all of **v**'s elements.

A dynamic model of invalidation could track function call nesting and use back pointers to perfectly track which pointers are valid. As for ownership, such a model has only theoretical interest because of the cost in run-time and memory it would incur. Also, it would imply a pointer implementation that is incompatible with existing C++.

Instead, we again use a static model. Our pointer safety verification tool [Sutter,2015]:

- Employs symbolic execution to detect invalidation. Symbolic execution is necessary to avoid too many false positives for flow-sensitive examples.
- Data flow between functions rely on the tracking of ownership as described in §3.6.

The analysis is compile-time and local, so that it has no run-time impact and scales to large code bases.

Range errors, null pointers, and uninitialized pointers are handled by separate techniques (§6).

As for dangling pointers and for ownership, this model detects all possible errors. This means that we can guarantee that a program is free of uses of invalidated pointers. There are many control structures in C++, addresses of objects can appear in many guises (e.g., pointers, references, smart pointers, iterators), and objects can "live" in many places (e.g., local variables, global variables, standard containers, and arrays on the free store). Our tool systematically considers all combinations. Needless to say, that implies a lot of careful implementation work (described in detail in [Sutter,2015]), but it is in principle simple: all uses of invalid pointers are caught.

3.9. Static invalidation limitations

Checking for invalidation by a static tool implies a few limitations. Consider a tricky example:

```
int* tricky(vector<int>& vi, int x)
{
        if (x)
                vi.push_back(4);
        else
                vi[2] = 4;

}

void use(int x)
{
        vector<int> v = { 1,2,3 };
        int* p = &v[2];
        tricky(v,x);
        *p = 7;
}
```

Here, **tricky(v,x)** may or may not invalidate **p** depending on the value of **x**. The "trick" is that the tool deduces what the function does to its arguments and its return value from the function's argument and return types. Here, the tool knows (without looking at the implementation of **tricky()**) that **tricky()** takes a **vector<int>&** so it can invalidate pointers to its **vector<int>** argument. Consequently, the call **tricky(v,x)** must be assumed to invalidate pointers to **v**'s elements, so that ***p=7** is rejected as a (possible) access through an invalid pointer.

Note that this is still local analysis; we just look at a function declaration with respect to possible invalidation. When invalidation is possible, we must be conservative and assume the worst. We consider whole-program analysis incompatible with the needs of large programs, the need for a fast debug cycle, and the needs of a language supporting dynamic linking. Whole-program can be useful, and we will use it, but not in the usual debug cycle.

How do we know whether a function invalidates a pointer? In general we do not know, so we must be conservative. We assume that **const** member functions and functions taking **const** arguments are not invalidating. For example:

```
vector<int>& v = { 1, 2, 3, 4 };
int* p = &v[0];

if (v.size()<10) return;   // does not invalidate
int x = v[7];              // does not invalidate
v[7] = 99;                 // invalidates (if you are simple minded, relying on const)
v.push_back(42);           // invalidates
```

In general, **const** arguments and **const** member functions are very useful in reducing the number of false positives because they eliminate many possibilities for invalidation. Uses of **const_cast** are noted and trigger greater conservatism in the analysis. In general, uses of casts eliminates most of the benefits listed here, so with a few exceptions the guidelines ban them.

We can do better if we know the semantics of operations. The calls to **v[7]** don't invalidate pointers into **v**; they just read and write the **int** referred to. However, looking just at which functions are non-**const**, we cannot know that. Writing to **v[7]** mutates an element of the **vector**, but not the structure of the **vector**, so all pointers into the **vector** remain valid. Thus, a tool can do better if it either knows the semantics of the container or we use a **[[lifetime(const)]]** annotation on that function to help. We dislike annotations because they can be misused (causing errors) and because pervasive reliance on annotations makes code hard to understand. However, rare annotations on key operations can yield significant improvements to analysis (limiting excess conservatism and false positives).

3.10. Safe pointer use

As we are catching unsafe uses, we can safely use iterators, smart pointers, and containers of pointers. For example, we can safely use raw pointers (**T***s) and iterators where we earlier would have worried about lifetime issues and may have used smart pointers. For example:

```
void some_function(int*, int*);

void user()
{
        int a[10];
        // …
        some_function(a,&a[10]);
}
```

Here, the call of **some_function()** cannot affect the lifetime of the elements passed, so there is no need for manipulating ownership using **shared_ptr**. This is perfectly safe because **some_function()** cannot store an argument pointer away for later use (it did not get **owner**s) and (in the absence of concurrency, see §5) the targets for the pointers cannot go out of scope until after **some_function()** returns to **user()**. Importantly, this reduces the need to pass around smart pointers. To contrast, consider:

```
void some_other_function(shared_ptr<int>, int nelem);

void user2()      // needlessly complicated
{
        const max = 10;
        shared_ptr<int> p(new int[max]);
        // …
        some_other_function(p, max);
}
```

For **some_other_function()** we have to suffer the cost of free store allocation and deallocation in addition to the cost of **shared_ptr**'s manipulation of its use count. It's also uglier and offers more opportunities for errors.

In function signatures, limiting the use of ownership pointers (e.g., **unique_ptr** and **shared_ptr**) to cases where you actually want to manipulate ownership is important for generality and performance.

4. Resource safety

C++'s model of resource management is based on the use of constructors and destructors to specify the meaning of object construction (initialization) and destruction (cleanup). For scoped objects, destruction is implicit at scope exit, whereas for objects placed in the free store (heap, dynamic memory) using **new**, **delete** is required. This model has been part of C++ since the very earliest days (1979).

4.1. Constructors and destructors

Consider a simple, classical example:

```
template<typename T>
class Vector {
public:
        Vector(int s);    // allocate space for s elements of type T and default initialize them
        ~Vector();        // destroy the elements and deallocate their memory
        // …
};

void f(int n)
{
        Vector<string> vs(n);
        // …
}
```

For each call of **f()**, a **vector** of **n string**s is constructed and properly destroyed upon exit. This scheme works perfectly for scoped objects, members of other objects, and parts of a class hierarchy. It has a weakness, though, for objects placed on the free store using **new**. You have to remember to **delete** each object created once and only once:

```
Vector<string>* make();          // construct a Vector<string> using new and fill it (somehow)

void use()
{
        auto p = make();
        auto& r = *make();
        delete p;
        // …
        delete p;
}
```

We have two errors here: the object assigned to **p** is deleted twice and the object bound to **r** isn't deleted. This kind of problem has been a major problem over the years. The lifetime and ownership rules of §3 ensures that they are now caught.

A conventional and simple repair of such examples is to use a "smart pointer." For example:

```
unique_ptr<Vector<string>> make();    // construct a Vector<string> using new and fill it
```

```
void use()
{
        auto p = make();
        // auto& r = *make();   // bad: the unique_ptr is a temporary
        auto q = make();
        auto& r = *q.get();
        // …
}
```

We could, of course, handle the problem by adding **owner** and use Guideline [Stroustrup,2015] support tools to ensure that the deletions are done correctly, but that is tedious and low-level. It doesn't scale because the **owner** annotation themselves become a source of errors. Using a smart pointer abstraction (like the ISO C++ standard library's **unique_ptr**) is easier and more elegant.

4.2. Move semantics

Some form of pointer is necessary for classical object-oriented programming where a base class acts as the interface to a number of derived classes of potentially different sizes. However, in the case of **Vector** (and many other classes, including the ISO C++ standard library containers), we do not need run-time polymorphism, so we can (and should) eliminate explicit use of pointers – even "smart" pointers. In such cases, we resolve the problem like this:

```
Vector<string> make(); // construct a Vector<string> using new and fill it

void use()
{
        auto v = make();
        auto v2 = make();
        // …
}
```

To get the **Vector<string>** out of **make()** without potentially expensive copying of element, we rely on a move constructor:

```
template<typename T>
class Vector {
public:
        // … usual stuff …
        Vector(Vector&& r) :elem{r.elem}, sz{r.sz} { r.elem=nullptr; r.sz=0; }
private:
        owner<T*> elem;
        int sz;
}
```

That is, the constructor taking the **&&** (rvalue reference) argument doesn't copy. Instead it moves; that is, it "steals" the representation of its argument and makes the argument empty. Such a constructor is called a move constructor.

This technique of moving objects rather than copying them has been used for decades, but only infrequently and unsystematically. The notion of direct language support for move semantics was pioneered by Howard Hinnant and is part of C++11. The use of move semantics allows us to move a large object implemented as a handle from scope to scope without overhead and without resorting to error-prone explicit use of pointers and explicit memory management.

One implication of having move semantics is that we can completely encapsulate the management of non-scoped memory. The basic rule is that **new** and **delete** belong inside the implementation of abstractions: "no naked **new**s" and "no naked **delete**s."

4.3. RAII

A resource manager (also known as a resource handle) is typically structured like the **Vector** example. In fact, **Vector** is a simplified version of the standard library **vector**. It controls access to resources (here, an array of elements on the free store). Typically, some resources are acquired in the constructors and released in the destructor. The resource acquisition in the constructor led to the old and cumbersome name "Resource Acquisition Is Initialization", usually abbreviated to RAII.

For most resources (memory, locks, file handles, etc.), acquisition can fail so a conventional resource manager must be able to report an error. The only fully general mechanism for that is throwing an exception if resource acquisition fails, and that's what the standard containers do. This scheme handles nested objects, class hierarchies, and containers simply and efficiently. However, there are also schemes that rely on explicit error handling. Those are brittle and rarely generalize; the best basically hand-simulate RAII.

To be safe from exception throws, an owner pointer should not be the only handle to an object in a context where an exception can be thrown. For example:

```
void f(int x)
{
        auto p = new int{666};          // bad: will leak (we might not get to "delete p")
        auto p2 = make_unique(777);     // OK: unique_ptr's destructor will delete the int
        if (x<0) return;
        might_throw();
        delete p;
}
```

Our guideline checker tool will catch the problematic **p**. One major use of **unique_ptr** is to ensure proper deletion of objects allocated on the free store in a function.

4.4. Standard library support

The ISO C++ standard-library provides containers following the resource manager model, providing RAII and move semantics: **vector**, **map**, **set**, **unordered_map**, and more. It also provides "smart" pointers for representing shared ownership (**shared_ptr**) and unique/exclusive ownership (**unique_ptr**).

5. Other problems

The model presented so far relies on sequential execution in a language with lexical scoping. In a multi-threaded system, we need to handle more cases.

Our rule set for concurrency is not yet fully developed. In particular, detached threads with pointers to shared data can be tricky. However, threads that are joined at the end of their scopes can be analyzed much as called functions and **shared_ptr**s can be used to keep data alive for threads with less well-behaved lifetimes.

There are also obvious opportunities for tools that analyze for race conditions and deadlocks.

Here, we will only briefly mention other ways of breaking the C++ type system. These problems are well known and have well-known solutions, so we will not address them here. Follow the Guidelines [Stroustrup,2015], use the ISO C++ standard library, and the Guideline Support Library (GSL) [Sutter, 2015b] to prevent those problems. For example:

- Misuse of casts can lead to type and memory violations: eliminate almost all casts.
- Misuse of unions can lead to type and memory violations: use a **variant** class rather than a plain **union** in most cases.
- Out-of-range access can lead to type and memory errors: use **path** from the GSL (a non-owning range abstraction).
- Access through a null pointer can lead to type and memory errors: use **not_null** from the GSL.
- Access through an uninitialized pointer can lead to type and memory errors: don't use uninitialized variables except as targets for low-level input operations, where essential.
- To minimize range errors, we also recommend using a **make_array()** function that returns an **owner<path<T>>** to allocate an array on the free store, rather than using **new** or **malloc()** directly.

The aim of the Code Guidelines is to eliminate a large range of errors by mutually supportive rules. No one rule can by itself prevent a large class of errors: ban one misuse and others will become popular (this is often referred to as "playing whack-a-mole"). Thus, our ideal is a large set of mutually supportive rules that together deliver type and memory safety guarantees.

Note that we can and do statically check the rules related to range safety and **nullptr** dereferencing, etc. The only reasons we don't claim complete type safety is

- Our rules have not yet been completely and systematically enforced on large code bases; our tools needs to mature.
- We need a few (isolated and explicit) type violations to effectively deal with hardware (e.g., casts to turn machine addresses into typed pointers and the ability to input into uninitialized memory).

6. Historical Note

To solve the problems related to dangling pointers and resource management, we applied a "cocktail" of techniques

- A type system to allow flexible and efficient ownership abstractions
- A support library to simplify programming by minimizing explicit manipulation of ownership and lifetime
- Local static analysis (in places using symbolic execution to handle flow analysis) to extend the static type system to enforce ownership and lifetime rules

Each, by itself, has been tried repeatedly in C++ and other languages relying on manual resource release, but individually they don't scale. The combination allows for a simple solution with no overheads and no compatibility problems in the context of C++.

We note that the problems related to resource leaks and to dangling pointers stemming from manual resource management have been pervasive in mainstream programming languages for decades. They have been the source of much complexity and many serious errors. They are only partially (and expensively) addressed by garbage collection. Obviously, the problem was hard – even though the solution is simple, cheap, and complete. In particular, we solve the garbage problem by not littering; i.e., not generating any "garbage."

This solution has deep roots in C++. Resource management based on constructors and destructors was among the very first features added to C to make C++ [Stroustrup,1982]. This work was followed up by the integration of resource management and error handling (RAII) [Stroustrup,1994], and eventually with Howard Hinnant's work on **unique_ptr** and move semantics for C++11 [Hinnant,2002].

The use of libraries to complement language features when dealing with resources also goes back to the earliest days of C++ and is embodied in the current ISO C++ standard library's containers (e.g., **vector**), resource management pointers (e.g., **shared_ptr**), threads, file handle (e.g., **ifstream**), and more.

The idea of attaching information to a pointer to make it possible to validate its use is equally old. Ranges are of course attached to array abstractions in many languages and Dennis Ritchie proposed a "fat pointer" along those lines. The **path** from the GSL is our deliberately simple variant of that idea. For ranges, no static model is sufficient, so some run-time checking is necessary. However, static analysis can minimize such checks – often better than individual programmers can in large code bases. The dynamic models for ownership (and of immutability (**const**)), thinking of properties as part of the value of an object, have been part of Bjarne's mental model of C++ for decades. Unfortunately, as for "fat pointers," the time and space overheads and compatibility precluded their use. They were simply mental models. However, their static counterparts are applicable in real-world contexts.

The use of static analysis for lifetime and ownership is part of a trend going back to the early days of C where lint was used to compensate for weaknesses of the type system. There has always been a tradeoff between what we could afford for the compiler to check and what was better done by an external tool. The tradeoffs are sometimes hard to make and change over time as needs and techniques evolve. Some of what our checker tool does could obviously be done by a compiler.

The static analysis tool is part of a long tradition of static analysis from many academic and industrial uses. For example, we rely on a variant of symbolic execution to trace ownership and lifetime through a function [Sutter, 2015]. To make this scalable to industrial uses, using local analysis only is essential.

From now on, we expect serious systems programming languages to apply a similar cocktail of technique to achieve simple, safe, and efficient access to hardware.

7. Acknowledgements

This article is the result of discussions around the Code Guidelines, and especially about its rules for lifetime and resource safety. In addition to the authors here, the main participants in this work were Neil MacIntosh, the implementer of the GSL and the lifetime safety analysis tool, and Andrew Pardoe.

Also thanks to Sergey Zubkov and Marc Eaddy for constructive comments on drafts of this paper.

8. References

- [ISO,2014] *Standard for Programming Language C++* (late draft 2014). http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4296.pdf (aka C++14).
- [Hinnant,2002] H. Hinnant: *A Proposal to Add Move Semantics Support to the C++ Language*. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2002/n1377.htm
- [Stroustrup, 1982] B. Stroustrup: Classes: *An Abstract Data Type Facility for the C Language*. Sigplan Notices, January, 1982.
- [Stroustrup, 1994] B. Stroustrup: *The Design and Evolution of C++*. Addison Wesley. ISBN 0-201-54330-3. 1994.
- [Stroustrup,2015] B. Stroustrup and H. Sutter: *C++ Core Guidelines*. https://github.com/isocpp/CppCoreGuidelines .
- [Sutter, 2015] H. Sutter and N. MacIntosh: *Lifetime Safety: Preventing Leaks and Dangling*. https://github.com/isocpp/CppCoreGuidelines/blob/master/docs/Lifetimes%20I%20and%20II%20-%20v0.9.1.pdf .
- [Sutter, 2015b] *The Guideline Support Library*. https://github.com/microsoft/gsl .