Concepts of programming languages: Memory models

Gunnar Bergmann

Universität zu Lübeck

***Abstract*—Resource handling is an important aspect of programming languages and related errors are still a common problem in today's software. Managing it manually is hard and error prone, so safer alternatives are required. Garbage collection protects against most problems with memory handling but is not usable in programs with strict memory and time constraints nor can it handle other type of resources.**

This paper presents alternatives to manual resource management that are safer to use but do not impose severe performance degradation. At first it explores the concept of RAII and show how it can be implemented, how typical classes look and then presents and solves limitations of RAII. Afterwards Rust's ownership model will be explored. It helps preventing typical security issues currently present in C++ by applying additional checks at compile time.

I. Introduction

In some languages, most notably C, the programmer needs to manage the memory by explicitly calling `free` whenever it is no longer needed. This leads to many problems:

- memory leaks when memory is never released
- use after free when a reference to an already freed location is accessed
- double delete when memory is deallocated twice
- repeated code because the memory needs to be released on every code path. As usual copied code often leads to other problems, especially when it is changed later.
- unreadable code when many nested *if*-statements are used to prevent repeated code.
- separation of allocation and release
- Exceptions are nearly impossible to implement because they skip the code necessary for releasing.

Detecting and debugging memory related errors is hard. Leaked memory accumulates slowly and needs a certain amount before causing problems. *Use after free* and *double delete* lead to *undefined behavior*. It is possible that everything works as expected and just under rare circumstances cause crashes or security issues. There are programs for the detection of potential errors but they are rarely used and can not detect all issues while often alerting false positives.

Garbage collection solves memory management by tracing references and detecting which objects can be safely removed. It works under the assumption that there is enough memory until the next collection cycle. It is not usable in software with strict memory constraints and the unpredictable frequency of collection cycles limits the benefit for software with real time requirements. Garbage collectors also can not handle other resources than memory because these are often unique, lack additional reserves and require a deterministic order of resource release.

Except for *use after free* all these problems can be solved by different classes using RAII [7]. Still *use after free* remains an important issue that causes lots of problems that the ownership model introduced with Rust [4] solves.

The alternatives presented by this note will also be able to create file handles or network sockets, that automatically close, as well as mutex locks that can unlock when no longer needed [7].

II. Local variables

The easiest form of automated memory management are local variables. Local variables can not outlive their scope and the memory is automatically reclaimed by the program [1]. They can be returned from functions by copying their values but pointers to the variables become invalid. It is easy for compilers to save them on the stack or hold them in registers, making local variables the fastest form of memory allocation, because deallocation takes places by decreasing the stack size at the end of each function and the locality of reference avoids cache misses. It is not possible to increase the capacity of stack allocated memory once it is no longer on the top of the stack and a stack allocated variable can not outlive it's function call.

The values of local variables can be returned from a function by copying the memory into a specific register or dedicated memory location but the variable itself is no longer valid, so keeping references to it is harmful.

These limits make exclusive local variable impractical for most cases.

III. RAII

When C++ added object oriented programming to C it also provided a way to encapsulate initialization and release of member variables. Classes provide a better encapsulation of their data and thus may have a *constructor* for the initialization and a *destructor* for the resource release. [7]

A minimal string class may look like

```
class String {
private:
    char* data; // pointer to a character
public:
    // Constructor
    String(const char* s) {
        data = new char[strlen(s)+1];
        strcpy(data, s);
    }

    // disable copying
    String(const String&) = delete;

    // Destructor
    ~String() {
        delete[] data;
    }
};
```

This concept is known by the name *Resource Acquisition Is Initialization* abbreviated as *RAII*, because the resource used by the object is allocated on it's construction.

Instances of these classes can act as an extension of the normal local variables. They can also be stack allocated but unlike the primitive ones their destructor is run at the end of the scope.

This string class from above can be simply extended to allow dynamically growing strings. It just needs to allocate a new chunk of memory and copy the old data and the extension into a single array.

```
concat(const char* s) {
    char* old = data;
    int len = strlen(old)+strlen(s)+1;
    data = new char[len]; // more memory
    strcpy(data, old);
    strcat(data, s);
    delete[] old;         // free old memory
}
```

The memory can now grow dynamically in a completely safe way but is still bound to a scope and automatically deallocated by the destructor. Like other scoped variables the values can be passed out of the scope by creating a new object with the same content. Depending on programming language and implementation the content is either copied or transferred to the new object, leaving an empty one behind.

When an object is destroyed, members and base classes destructors are also called [1]. This allows safe composition of multiple classes, where each one uses RAII. Typically some or all of the operations are generated by the compiler, so that except for writing some basic data structures and wrappers around imported code from other languages it is not necessary to provide these for each class.

RAII can be used for many types of resources. Some examples from C++ are the classes `fstream` for files and `vector` for dynamic growable arrays. The prime example is the usage of RAII for automatically releasing mutex locks:

```
{
    lock_guard<mutex> guard(some_mutex);
    // ... code inside the mutex
} // some_mutex automatically unlocked
```

A. Destruction order

An object is destroyed at the end of it's lifetime, which can be automatically done for temporary objects and local ones when they leave the scope, in many languages visible as the closing brace `}`.

Member variables are destroyed by their parents destructor from the last to the first and afterwards the base classes' destructors are called if existent [1].

In all the automatic destructor calls the variables are destructed in reverse of their creation order and independent from the code path [1]. RAII's destruction order is deterministic. The reverse order ensures that no object is destroyed while others depending on it are still alive.

There are additional ways of creating and destroying objects manually, which is needed for implementing some basic types that itself use RAII but need to manage their internal memory manually. Forsimplified example the C++ type `vector` allocates a chunk of raw memory and constructs the objects internally. When elements are removed or the `vector` is destroyed, then all of the elements destructors are called before the object itself gets destroyed. You rarely have to do the management yourself unless you need some specialized and efficient datastructures that are not part of the standard library.

When an exception is thrown the runtime library performs *stack unwinding* by going back all function calls and destroying every object until the according catch-statement has been reached. Destructors should not throw exceptions because throwing in a stack rewind would lead to two simultaneous active exception which most languages can not handle and immediately abort. [1]

B. Solved Problems

Under the assumption that the implementation of classes is correct, RAII solves some of the problems of manual memory management. RAII improves the code structure by reducing code duplication. It pulls both acquisition and release from the code using it into one class, that keeps the related code close together. By removing the need for calling `new` and `delete` outside of the class it solves *double deletes* and memory leaks. It does not solve *use after free* because there may still be references to memory after the owning object has released it. Also data structures require strict hierarchies so that there is a order for the destructors. Any cases of cyclic references are not possible although the section about *smart pointers* shows ways to relax these constraint.

C. Containers

The most important building block for programs are *containers* [7]. These are generic classes that allocate memory for holding multiple objects and vary in their performance characteristics and usable operations. On destruction they call the destructors of all the elements in them and free the memory. The most important container is `vector` for storing a dynamically growable array and `map` or `unordered_map` for associative containers.

Some operations can remove, add or internally move elements, causing pointers to individual elements or *iterators* to dangle. In modern C++ this is a common error because it is often not visible from the outside that elements may move and under which circumstances, although some containers guarantee stable references.

D. smart pointer

Unlike using objects directly, a pointer can be moved but the memory it refers to is not invalidated, so all other pointers remain valid. We use the term *owning pointers* to refer to those types of pointers that needs to free the memory on release. Later we will explore the ownership in the Rust programming language that uses the same concept in a way the compiler can verify to prevent common sources of errors.

Additionally to `owning pointers` there are other types of pointers that just refer to a memory location without ever acquiring or freeing the memory themselves. The pointers present in C are called *raw pointers* here.

Since raw pointer need to call the appropriate function to release the memory it is safer to provide a wrapper class that uses RAII to release the memory automatically. These are called smart pointers. As a general rule owning raw pointers should be avoided and either replaced by using the object without indirection or with a smart pointer. There are typically three types of smart pointers [3]:

- `unique_ptr` (C++), `Box` (Rust) is the most simple type. It holds a reference and frees the memory on deletion. It can not be copied. It models exclusive ownership [3].
- `shared_ptr` (C++), `Rc` (Rust) provides shared ownership. It lets multiple smart pointers refer to the same memory location. When the last one is destroyed it frees up the memory. `shared_ptr` is implemented with reference counting. When creating a new object it allocates additional space for another integer and stores a counter there. When a `shared_ptr` is copied it increases the counter and the destructor decreases it. When the counter drops to zero the inner object is destroyed. This smart pointer allows multiple references but the programmer needs to be careful to not create a cyclic dependency because that prevents the counter from reaching zero and may leak the whole cycle [3].
- `weak_ptr`(C++), `Weak` (Rust) refers to an object managed by shared pointers but allows the pointer to dangle. It does not own the resource nor can be used directly but can grant temporary ownership by beeing upgraded to a `shared_ptr`, which may fail. Already freed objects are although that fails if the object has already been destroyed. The weak pointer can be used to break cycles [3].
  For example in a tree structure each node has a list of shared pointers to child nodes and a weak pointer to it's parent. It forms a hierarchy because when the last reference to the root node is dropped the whole tree will recursively be released. It is still possible for a node to safely access it's parent.
  Weak pointers are implemented by adding another counter to the one used by *shared pointer*. When the reference counter drops to zero the object is destroyed but the memory not released, so that the second counter still remains valid. The memory is released on destruction of the last *weak pointer* [3].

Both containers and smart pointers store memory but they are used differently. Containers organize multiple objects of the same type, whereas smart pointers contain a single element, but vary in the access to it, and when inheritance is used, they allow downcasting to a base class.

IV. RAII in garbage collected languages

A garbage collector is an easily usable protection against memory related issues but lacks support of other resources. Classes often have *finalizers* that can close leaked resources but many of them including file handles and network sockets need to be closed as soon as possible so that other programs can use them. Often the release order matters or it takes an unpredictable time span to the next collection cycle so it sometimes takes too long and may cause nearly undetectable errors. Relying on *finalizers* therefore is strongly discouraged.

Traditionally garbage collected languages use `finally` to run specific code for resource release at the end of the scope, although that requires the discipline to surround every resource usage with a finally block and as a consequence is rarely used. Also resource handling occures more othen than defining a class containing a closable resource, so RAII reduces code duplication in comparison to `finally` [6].

Some languages like *D* support a garbage collector for memory and RAII for other resources [2]. Other languages were extended with special keywords that enable RAII-like behavior. For example *Python 2.5* [5] introduced the methods `__enter__` and `__exit__` that can be used with the new keyword `with` to support automatic closing of objects:

```
with open("test.file") as f:
    content = f.read()
```

V. Rust

Rust is a new language that aims at memory safety, abstractions without overhead and multithreading. [4]

Traditional RAII can resolve some common errors but it does not prevent against dangling references. Modern programs heavily rely on iterators and modifying a container while iterating over it may cause the iterator to point to

now invalid memory similar to a dangling reference or skip elements or visit them twice.

Accessing the same memory location from to different threads may easily lead to data races. Even when a RAII ensures safety for a variable, another thread may access in an intermediate state that is not safe to use.

Rust uses the concepts of ownership, borrowing and lifetimes to eliminate these bugs without introducing additional overhead.

A. Ownership

Ownership means that there is always a single variable that can access the value and return it. The ownership can be transferred to other variables but not shared. [4]

The strict semantics of ownership ensure that RAII is used correctly and that no variable can be shared across threads.

Variables have ownership over the resources they access. You can create an heap-allocated array with three elements with

```
let a = vec![1, 2, 3];
```

The variable `a` has ownership over the content of the array. When `a` goes out of scope it automatically reclaims the memory by using RAII. You can transfer ownership to another value.

```
let b = a;
```

The array is moved to `b`. Accessing it afterwards is a compiler error.

```
a.push(4); // append 4
```

will not compile because the content of `a` has been moved to `b` and is no longer accessible.

When multiple control paths exist and a value has been moved in one, it is no longer accessible afterwards until a new value has been assigned. Functions can take a value by move, which makes the code more efficient than copying values and also prevents accidental modifications.

Some primitive types are an exception from the ownership rule and are copied instead of moved, because move and copy are equal for them.

B. Borrowing

Ownership alone is not useful because just a single source can access a value and you can not even pass it to a function without losing the ownership. [4] Instead you can temporarily borrow the value. It is technically just a reference like in C++ but with additional checks at compile time.

Rust employs a concept similar to a read-write-lock: Only one mutable reference at a time is allowed but multiple immutable ones. Borrowing makes it impossible to transfer ownership or modify a container while a reference or an iterator to it exists.

Having multiple read-only references is save because no one can change anything.

```
fn foo() -> &i32 {
    let a = 12;
    return &a;
}
```

will not compile because the local variable `a` goes out of scope at the end of `foo` and invalidates the returned reference.

```
let x = vec![1,2,3];
let reference = &mut x;
println!("{}", x);
```

will not compile because x can not be used by the print function while there is still a mutable reference to it.

This is not an actual lock and does not make the types thread-safe. Sending a reference to another thread is not allowed. It just prevents creation of dangling references. Still the same mechanism is also used for thread-safety to prevent unlocking a mutex while still holding a reference to it's content:

```
{
    let guard = mutex.lock().unwrap();
    modify_value(&mut guard);
}
```

C. Lifetimes

The compiler enforces and validates the borrowing and ownership rules with lifetimes. These are bound to the scope and every type has an implicit lifetime on it. [4] References are represented by a leading single quote: `'lifetime` There is a special lifetime `'static` for items that life as long as the process exists.

```
struct Foo {
    x: i32,
}

// the lifetime 'a is a generic parameter
// it returns a reference with the same
// livetime as the input
fn first_x<'a>(
        first: &'a Foo,
        second: &Foo)
        -> &'a i32 {
    &first.x
}
```

In this example the parameter `first` and the returned reference are annoteted with the generic livetime parameter `'a`. It is not possible for the caller to let the reference outlive the object or modify it while the reference still exists. Taking immutable references is still possible, but mutable ones are forbidden.

Although the compiler generates and checks the lifetime for every object, they can be automatically generated based on the context except for some ambiguous cases.

D. Anonymous functions

Like most modern languages rust supports the creation of anonymous functions, also called lambda functions. These are created locally and can access the local variables. To obey the safety rules without introducing additional overhead there are three basic types of anonymous functions.

The simplest type accesses the local variables by reference. It can not be returned from the function because then references could no longer access the data.

The second type takes the variables by move and forbids the access after the function definition. For taking copies of the value first store a copy and then use the copy inside of the function.

The last type can be called only once. This allows the function to consume the inner data instead of leaving it in a valid state. To build similar behaviors for own types these can take the `self`-reference by move. `self` accesses the object on which the method is called, similar to the `this` by other types. Rust's ownership ensures that the object is moved inside a method and then destroyed at the end of the call. [4]

```
fn do_something(self) {
    // ...
} // self is destroyed
```

E. Limits of ownership

It is not possible to protect against all kinds of bugs even with the strict ownership rules. Especially the value inside of a shared smart pointer has an unpredictable lifetime. While it can be guaranteed that the value has at least the pointers lifetime, aliasing mutable references can not be prevented. There are some types in Rusts standard library that allow safe access e.g. by falling back to runtime checks.

F. Ownership in C++

Ownership is nothing that was invented for Rust. As mentioned in the smart pointer section it is often useful to think about owning references for structuring a program [3]. Only the application of additional checks to enforce correct ownership semantics is new.

At CppCon 2015 a new programming guide and a tool, that can check for rules similar to the rust compiler, although less strict, were announced. Some of Rust's rules for example those that protect against data races in multithreaded programs are also not covered. [8]

VI. Conclusion

This paper has shown how RAII can create a safer and simpler alternative to manual memory management and how some internally unsafe classes can provide useful building blocks for creating quite safe programs. Afterwards it has presented the ownership concept of Rust that prevents common problems of RAII.

In comparison to garbage collection RAII can not handle cyclic references. The programmer always needs to use an adequate container or smart pointer, structure the code and break cyclic references manually. On the other side this forces a good structure for the program and allows a deterministic and immediate destruction which enables the use of RAII for other types of resources.

Whereas garbage collection is a general solution for all kinds of memory related issues, RAII is one for all kinds of resources except those ordered in cyclic dependencies.

Ownership can eliminate many issues at compile time and provides a fine grained control over the destruction of objects. Other languages have to detect problems at runtime or don't protect against them at all. This does not just prevent some sources of memory issues but also other classes of problems that need to by found by extensive testing otherwise.

On the downside it is a lot to write. Simply passing a reference around is not possible because you need to ensure that no invalidated variable can be used and sometimes extra lifetime annotations are required so that the borrow checker can verify the code. Even when you sacrifice performance and make extensive use of copying objects or just use reference counting, there is still a lot work involved in manually creating copies every time and you loose the ability to use mutable objects, which need other verbose workarounds.

References

[1] Working Draft, Standard for Programming Language C++. Technical report, 2015. N4296.

[2] Ali Cehreli. *Programming in D: Tutorial and Reference*. CreateSpace Independent Publishing Platform, 1 edition, 8 2015.

[3] Scott Meyers. *Effective modern C++ : 42 specific ways to improve your use of C++11 and C++14*. O'Reilly Media, Sebastopol, CA, 2014.

[4] The Rust Project. The Rust Programming Language. https://doc.rust-lang.org/stable/book/, 2015. [Online; accessed 12-November-2015].

[5] Python Software Foundation. Python 2.7.10 Documentation. docs.python.org/2/, 2015. [Online; accessed 12-November-2015].

[6] Bjarne Stroustrup. FAQ. https://isocpp.org/faq. [Online; accessed 15-November-2015].

[7] Bjarne Stroustrup. Foundations of c++. In Helmut Seidl, editor, *Programming Languages and Systems*, volume 7211 of *Lecture Notes in Computer Science*, pages 1–25. Springer Berlin Heidelberg, 2012.

[8] Herb Sutter. Writing good C++14 ... by Default. https://github.com/isocpp/CppCoreGuidelines/blob/master/talks/Sutter%20-%20CppCon%202015%20day%202%20plenary%20.pdf. [Online; accessed 12-November-2015].