Software Assurance Tips

A product of the Software Assurance Tips Team[1]

Jon Hood

Monday 12th April, 2021

1 Sticking with a RAII Standard

Updated Wednesday 11th February, 2026

Resource Acquisition Is Initialization (RAII). When working with legacy code, variables and memory management often do not use RAII concepts. When memory is created in an uninitialized state, developers risk the use of uninitialized memory further down in the application data flow.

To solve this, C++ implemented two ways of initializing new memory: *default* initialization and *value* initialization. While default initialization is typically faster, developers quickly realized that there were rarely reasons to keep memory uninitialized. Initializing memory as soon as it's allocated keeps developers from shooting themselves in the foot.

One of the best ways to prevent memory errors is to use RAII. Particularly in exception-based code, RAII concepts provide a memory-safe technique for resource management.[2] Several issues face developers when maintaining legacy code with new programming concepts, particularly in the DoD. Legacy code is generally void of RAII concepts and has not been built with the latest standards of safe programming practices.

1.1 Updating the Code

The first option for maintaining legacy code in a memory-safe way is to update the code. Wrapping old pointers in smart pointers can help make the code more maintainable.

```
bool *newBool = new bool();
if (*newBool)
        cout << "We're␣true!"␣<<␣endl;
else
␣␣␣␣␣␣␣␣cout␣<<␣"We're false!" << endl;
delete newBool;
```

Listing 1: Non-compliant listing

```
auto newBool = make_shared<bool>();
//Another way: shared_ptr<bool> newBool(new bool());
if (*newBool)
        cout << "We're␣true!"␣<<␣endl;
else
␣␣␣␣␣␣␣␣cout␣<<␣"We're false!" << endl;
```

Listing 2: RAII-compliant listing

There are several issues with updating the old code to the RAII-compliant code:

1. Forgetting to remove the manual memory management can cause double-free errors.
2. Updating the code can cause translation issues (eg: between default and value initialization)
3. Inconsistency can increase maintenance burdens for maintained code.
4. Deletion of arrays require special handling.

1.2 Double Frees

Developers can introduce double free conditions where a pointer is managed by both a smart pointer and in the code itself. Also, conditions arise when a single pointer is handed over to multiple smart pointers for management.

At the time of this writing, the latest versions of Fortify, Coverity, Checkmarx, Parasoft, clang-analyzer, and the GCC 10 `-fanalyzer` flag are all incapable of identifying the double free in Listings 3 and 4.

```
bool *newBool = new bool();
shared_ptr<bool> test1 (newBool);
shared_ptr<bool> test2 (newBool);
//...
```

Listing 3: Double Smart Pointer

```
bool *newBool = new bool();
shared_ptr<bool> test1 (newBool);
//...
delete newBool;
```

Listing 4: Smart and Dumb Pointer

1.3 Initialization Errors

We've also seen a large number of initialization errors when updating to smart pointers. Using the above examples, it's tempting for a developer to use shortcuts and commit the error demonstrated in Listing 5.

```
shared_ptr<bool> test1 (new bool);
cout << "Test1 is: " << *test1 << endl;
```

Listing 5: Uninitialized Boolean

The unpredictable nature of this issue can be demonstrated in the example in Listing 6 where the Boolean uses default initialization and obtains a random value from previously-freed memory.

```
#include <climits>
#include <iostream>
#include <memory>
#include <random>
using namespace std;
int main()
{
        random_device rd;
        mt19937 gen(rd());
        uniform_int_distribution<> distrib(INT_MIN, INT_MAX);
        vector<int*> deleteLater;
        for (int i = 0; i < 100; i++)
        {
                int *deleteMe = new int;
                *deleteMe = distrib(gen);
                //delete some memory now
                if (*deleteMe > 0)
                        delete deleteMe;
                else
                        deleteLater.push_back(deleteMe);
        }
        shared_ptr<bool> newBool(new bool);
        cout << "Value: " << *newBool << endl;
        //delete some memory later
        for (int *n : deleteLater)
                delete n;
        return 0;
}
```

Listing 6: Uninitialized Boolean

1.4 Inconsistency Can Increase Maintenance Costs

Consider updated code that uses a multitude of different memory handling methods. Some memory is managed manually using `malloc` and `free`. Some pass the previous pointer values to smart pointers for their management. Others have been updated to use the `make_shared` construct. Still others have been updated to custom classes and structs.

Consistency is more important than updating. Introducing multiple memory handling patterns to a section of code increases its complexity. If old memory handling methods cannot be replaced or are not planned to be replaced, it may make more sense to stick with confusing (but consistent) code rather than adding additional complexity.

1.5 Smart Pointers and Arrays

Finally, some special considerations are needed for allowing smart pointers to handle arrays. The `shared_ptr` construct permits the developer to define a custom deleter like the one for this array of 10 Booleans:

```
shared_ptr<bool> test1{
        new bool[10], [](const bool *ptr) { delete [] ptr; }
};
```

Listing 7: Custom Deleter

1.6 Conclusion

Smart pointers and RAII concepts help developers prevent memory errors that can plague software. Updating legacy code to RAII concepts can increase its maintainability, usefulness, and security. Nevertheless, updates to the code should not come at a price of inconsistency. When updates can only be applied partially or introduce additional complexity, consistency should be preferred.

References

[1] Jon Hood, ed. SwATips. https://www.SwATips.com/.

[2] Bjarne Stroustrup. “Exception Safety: Concepts and Techniques”. In: (2001). URL: https://www.stroustrup.com/except.pdf.