

Application-level countermeasures:
Safer C++ with smart pointers

2 weeks ago

Instead of C strings `char*` and C arrays `int[]`,
C++ has safer alternatives:

```
std::string s;  
std::array<int,N> a;  
std::vector<int> v;
```

Not 100% memory safe, but a lot memory-*safer*,
esp. against **spatial bugs**

To prevent **temporal bugs**, C++ provides smart pointers

Spot the bug!

```
1. void f(double* data, int N) {  
2.     double* tmp = new double[N];  
3.     setup(data, tmp, N);  
4.     if (not_needed(data, tmp, N))  
5.         return;  
6.     calculate(data, tmp, N);  
7.     delete[] tmp;  
8. }
```

This is not nice C++ code: we should use a C++ `std::array` instead of a `double*` pointer to a C-style array, but ignore that

Potential memory leak

```
1. void f(double* data, int N) {  
2.     double* tmp = new double[N];  
3.     setup(data, tmp, N);  
4.     if (not_needed(data, tmp, N))  
5.         return;  
6.     calculate(data, tmp, N);  
7.     delete[] tmp; // tmp is not freed if we return early  
8. }
```

Fixing this

```
1. void f(double* data, int N) {  
2.     double* tmp = new double[N];  
3.     setup(data, tmp, N);  
4.     if (not_needed(data, tmp, N)) {  
5.         delete[] tmp;  
6.         return;  
7.     }  
8.     calculate(data, tmp, N);  
9.     delete[] tmp;  
10. }
```

Spot the bug!

What if `setup(data, tmp, N)` throws an exception?

Fixing this

```
1. void f(double* data, int N) {  
2.     double* tmp = new double[N];  
3.     try { setup(data, tmp, N);  
4.         }  
5.         catch (int err) { delete[] tmp ; throw err; }  
6.     if (not_needed(data, tmp, N)) {  
7.         delete[] tmp;  
8.         return; }  
9.     calculate(data, tmp, N);  
10.    delete[] tmp;  
11. }
```

Spot the bug!

maybe calculate can throw an exception...

Better still – if `calculate()` can throw an exception

```
1. void f(double* data, int N) {
2.     double* tmp = new double[N];
3.     try { setup(data, tmp, N);
4.         if (not_needed(data, tmp, N)) {
5.             delete[] tmp;
6.             return;
7.         }
8.         calculate(data, tmp, N);
9.         delete[] tmp;
10.    }
11.    catch (int err) { delete[] tmp ; throw err; }
12. }
```

Much nicer: `unique_ptr`

```
1. void f(double* data, int N) {  
2.     unique_ptr <double[]> tmp  
3.         = make_unique <double[]> (N);  
4.     setup(data, tmp, N);  
5.     if (not_needed(data, tmp, N)) { return; }  
6.     calculate(data, tmp, N);  
7.     // delete tmp;  
8. }
```

Unique pointer: **automatic deletion when owner - here `tmp` - goes out-of-scope**

Note: this secure code is simpler than the original insecure code!

C++ smart pointers

- `unique_ptr`
- `share_ptr`
- `weak_ptr`

To prevent temporal memory corruption:
memory management is automated

More generally, it improves code quality

Safe?

```
1. void f(double* data, int N) {  
2.     double* tmp1 = new double[N];  
3.     double* tmp2 = new double[N];  
4.     calc(data, tmp1, tmp2, N);  
5.         // calc is guaranteed not to throw an exception  
6.     delete[] tmp1;  
7.     delete[] tmp2;  
8. }
```

Remember: memory allocation may fail

```
1. void f(double* data, int N) {  
2.     double* tmp1 = new double[N];  
3.     double* tmp2 = new double[N]; // may fail  
4.     calc(data, tmp1, tmp2, N);  
5.         // calc is guaranteed not to throw an exception  
6.     delete[] tmp1; // not reached if tmp2 allocation fails  
7.         // => memory leak  
8.     delete[] tmp2;  
9. }
```

Fix using unique pointers (and auto)

```
1. void f(double* data, int N) {  
2.     auto tmp1 = make_unique<double[]>(N);  
3.     auto tmp2 = make_unique<double[]>(N);  
4.     calc(data, tmp1, tmp2, N); // guaranteed not to throw  
5.     // delete[] tmp1;  clean-up is automatic  
6.     // delete[] tmp2;  
7. }
```

`auto` is simply convenient short-hand, that avoids having to write out the type

What's suspicious here?

```
1. float* suspicious(float* x, float* y, int N) {  
2.     float* tmp = new float[N];  
3.     process(3.1415, x, y, tmp, N);  
4.     delete[] x;  
5.     delete[] y;  
6.     return tmp;  
7. }
```

Again, we should be using C++-style `float` arrays,
not these `float*` C-style arrays

What's suspicious here ?

```
1. float* suspicious(float* x, float* y, int N) {
2.     float* tmp = new float[N];
3.     process(3.1415, x, y, tmp, N);
4.     delete[] x; // Does caller expect us to delete x
5.     delete[] y; //                               and y?
6.     return tmp; // Caller responsible for freeing tmp
7. }
```

Improved code using unique pointers

```
1.  unique_ptr<float[]> suspicious
2.                                     ( unique_ptr<float[]> x,
3.                                     unique_ptr<float[]> y, int N)
4.  {  auto tmp = make_unique<float[]>(N);
5.      process(3.1415, x, y, tmp, N);
6.      // delete[] x;      clean-up is automatic
7.      // delete[] y;
8.      return tmp;
9.  }
```

Note that

- It is clear to the caller that the function `suspicious` takes ownership of `x` and `y`, and hence responsibility for freeing them
- The caller of the function becomes owner of the return value `tmp`
- Freeing `tmp` is automatic, when it eventually goes out-of-scope

C++ smart pointers

- `unique_ptr`: only one pointer to an object, which *owns* the object
 - automatic de-allocation when owner dies
 - created using `make_unique`
 - move semantics to transfer ownership

move operation for unique pointers

```
namespace std; // so we don't have to write std:: everywhere
```

```
unique_ptr<int> a = make_unique<int>(42);
```

```
unique_ptr<int> b = a; // illegal
```

```
unique_ptr<int> b { move(a) }; // legal
```

```
    // b is now owner, a is set to null
```

C++ smart pointers

- `unique_ptr`: only one pointer to an object, which owns the object
 - automatic de-allocation when owner dies
 - created using `make_unique`
 - move semantics to transfer ownership
- `share_ptr`: possible many pointers to same object
 - **reference counting** to free object when all references are gone
 - intuition: shared pointers collectively own the object they point to
 - created using `make_shared`

Using shared_ptr

```
1. void g() {
2.     unique_ptr<int> u = make_unique<int>(42);
3.     shared_ptr<int> u2 = u; // illegal
4.
5.     shared_ptr<int> sp = make_shared<int>(43);
6.     shared_ptr<int> sp2 = sp; // reference count increased to 2
7.     shared_ptr<int> sp3 = sp; // reference count increased to 3
8.
9.     f(sp);
10.    // maybe f stores sp somewhere, increasing ref-count
11.
12. }    // Can sp be freed ?
13.    //      Yes, unless f stores ss somewhere
```

C++ smart pointers

- `unique_ptr`: only one pointer to an object, which owns the object
 - automatic de-allocation when owner dies
 - created using `make_unique`
 - move semantics to transfer ownership
- `share_ptr`: possible many pointers to same object
 - reference counting to free object when all references are gone
 - intuition: shared pointers collectively own the object they point to
 - created using `make_shared`
- `weak_ptr`
 - a weak pointer is ignored for reference counting,
 - so a weak pointer may point to an object that no longer exists
 - mandatory `lock` before using it

Using weak_ptr - with expired() with lock()

```
shared_ptr<int> sp = make_shared<int>(42);
shared_ptr<int> sp2 = sp; // reference count increased
weak_ptr<int> wp = sp; // allowed, but reference count not increased

int i = *sp2; // using shared pointer is always allowed
int j = *wp;  // compile error:
               //      using weak pointer without locking is NOT allowed

// We can check if object wp points to, i.e. wp*, no longer exists
if ( wp.expired() ) {
    ... // object wp points to has been de-allocated
}

// To use wp, we have to lock it to obtain a shared pointer
if ( shared_ptr<int> sp3 = wp.lock() ) {
    // w is alive, and we can access it via shared pointer sp
    int k = *sp3;
    ...
}
```

RAII (Resource Acquisition is Initialisation)

Smart pointers are example of RAII

RAII is programming idiom where ownership of some resource is given to reference and clean-up is automatic when unique owner disappears

Bit of a misnomer, because it's also – mainly? - about clean-up, not just about initialisation

The resource can be **heap-allocated object**, but also

- **file descriptor**
- **port**
- **lock** or **mutex**
- ...

Example: RAI for files

```
void proc() {  
    FILE* fp = fopen("readme.txt", "r");  
    // lots of complicated code using fp  
    ... ; fread(buffer, 1, N, fp); ...  
    // we are done using fp  
    fclose(fp);  
}
```

Spot the bug!

if an exception is thrown before we get to `fclose()`
then the file will not be closed

Example: RAII for files using smart pointers

```
void proc() {  
    unique_FILE fp = fopen("readme.txt", "r");  
    // lots of complicated code using fp  
    ..; fread(buffer, 1, N, fp); ...  
    // we are done using fp  
    // fclose(fp);   clean up automatic if fp goes out of scope  
}
```

if we define `unique_FILE` as unique pointer type with a custom clean-up operation

```
unique_FILE = unique_ptr<FILE, file_cleanup>;  
struct file_cleanup {  
    void operator()(FILE* fp) { fclose(fp); }  
};
```

Recap – memory corruption countermeasures

- To prevent spatial bugs in C,
we can use safer string libraries or `-fbounds-safety`
- To prevent spatial bugs in C++,
we can use C++-style strings, arrays, and vectors
- To prevent temporal bugs in C++,
we can use smart pointers
- To *detect* memory corruption bugs,
we can use SAST tools (e.g. cppcheck or compiler)
or use fuzzing as DAST technique
- We can also turn on platform level defenses
eg ASLR, NX, CFI, memory tagging, guard pages, ...

Of course, nicer to use a language where we don't have to worry about memory corruption at all...