Writing C++ The "C++ Way"

Secure Coding Using Modern C++

Adhokshaj Mishra

Sr. Technical Architect, Pyramid Cyber Security and Forensic Ltd.

August 20, 2017

Who am I?

- Security researcher (malware and applied cryptology)
- Spoilt by C++ beyond repair!
- Working as Sr. Technical Architect (Security), Pyramid Cyber Security and Forensic Ltd.
- Mostly dabbling into fairly low level stuff (OS, compiler, linker, loader yada yada)
- Reach me at me@adhokshajmishraonline.in

Agenda

- Why “modern” C++?
- What is wrong with traditional way of teaching C++?
- Hidden gifts of C++: dark corners and pitfalls
- Strings
- I/O
- Arrays
- Memory
- Pointers
- Function pointer, functor, lambdas and std::function

Why "modern" C++

- Modern C++: C++11 (ISO/IEC 14882:2011) and above
- Classic C++: C++03 (ISO/IEC 14882:2003) and below
- Modern C++ provides abstractions over many nitty gritty details which were most common sources of bugs.
- It makes C++ less frustrating (ever had fun with segmentation faults?)

What is wrong with traditional way of teaching C++?

- Teaching C++ as a set of "deviations" from C is a shortcut. C++ is not a superset of C.
- Collection of similar named functions instead of classes (strlen, strcmp, strcpy blah blah).
- Makes C++ seem harder than it really is. No wonders people tend to hate it (Ugh, C++ is too complicated, Java/.NET/Python is fun).

Hidden gifts of C++

- Buffer related bugs (buffer overflow, off by one errors)
- Memory related bugs (memory leakage, double free, use after free)
- Pointer magic (dangling pointer)
- Format string vulnerability (data leakage, arbitray write) (cruft from C)

Strings: The Classic Way

- Old C++ way: char*
- String operations are not intuitive. Want to append? strlen x 2 + malloc + strcpy + strcat. Forgot null terminator? Magic is waiting to happen.
- char* strings require you to understand pointers long before you can touch strings. It requires whole "name of array is pointer to start of array" conversation way too early.
- It forces you to understand, and pay attention to, null terminator in all string operations; otherwise somewhere magic is waiting to happen.

Strings: The Modern Way

```
#include <string>

std::string s1 = “hello ”;
std::string s2 = “world”;
```

Want to concatenate two strings?

```
std::string greeting = s1 + s2;
```

Want to compare strings?

```
if (s1 == “hello”)
```

or

```
std::string::compare
```

I/O

- Cruft from C: printf, scanf, fprintf, fscanf etc. These generally give better performance, but are a constant source of bugs. Also, these force you to understand and remember cryptic(?) format specifiers.
- Modern C++ way: Stream I/O (cin, cout, cerr). These will give slightly worse performance, but a beginner need not to worry about it.

```
#include <iostream>
std::cout << “Total number of apples ” <<
no_of_apples;
```

- Want string formatting? Use StringStream instead of sprintf()
- Readability rules.

Arrays: The Classic Way

Why is it bad?

- Brings pointers into story too soon
- Leads to off by one, bounds checking errors (overflows, magic, and segmentation fault)
- Needs hand-rolled loops way too often

Arrays: The Modern Way

The C++ way: <vector>. It is probably the only class a beginner needs.

```
#include <vector>
std::vector<int> buffer(size);
```

- Want to access the element at index i? Use `buffer[i]`, or `buffer.at(i)`.
- Want to add more elements? Use `buffer.push_back()`, and it will take care of it automatically.
- Want to create some more space in buffer? Use `buffer.resize()`
- Out of range access will result in an exception instead of some "magic". No more buffer overflows, and no more resulting exploits.

Memory and Pointers

Raw pointers are bad (read: tricky to get it right)

- Burden of deallocation is on programmer. Forgot to delete? Hello memory leakage.
- By extension of first point, how do you know whether a pointer is good or bad. Result? Use after free/double free bugs.
- No concept of ownership. A memory can be owned by none, one, or more “parties”.

Memory and Pointers

Solution: Smart pointers (unique_ptr, shared_ptr)

```
#include <memory>
 ...  // some code here
{
// dynamic allocation that cannot be copied to another
pointer.  Deallocation happens as soon as it goes out
of scope.
std::unique_ptr<int> ptr1 = std::make_unique<int[]>(5);

// deallocation happens when reference count becomes
0.
std::shared_ptr<int> ptr2 = std::make_shared<int[]>(5);
}
// both memory allocations are deallocated
automatically
```

Function Pointer

- Cruft from C
- A literal pointer to location in memory where function is stored
- Cryptic syntax
- Useful for callbacks

Example: `void (*(*f[])())()` declares `f` to be array of unspecified size, of pointers to functions, which return pointers to functions which return void.

By the way, I am uncomfortable with the fact that I am comfortable with this statement. Oh well...

Function Pointer

Why are function pointers bad?

- Hard to decode syntax
- Cannot be optimized easily. To be specific, there is no easy way to inline calls via pointer.

Is there a better way to do it?

Functor

- A functor (also known as function object) is a class with () operator overridden so that it can be called like a function.
- Syntax is much cleaner.
- It can be optimized easily.
- Functors are more flexible as they can have internal state

Example:

```
struct add_x
{
    add_x(int x) :  x(x) {}
    int operator()(int y) const { return x + y; }
private:
    int x;
};
```

Functor

Disadvantages of functor:

- Code becomes more complex, as you have to define a class, constructor, operator() etc.
- Every functor must have a name, even if it is used only once. Naming things correctly is a hard problem, and by naming single use functors, we are reducing our "name pool"

What if we could somehow kick the name out of equation, grab the functor by its ears, and drop it in the parameter where it belogs?

Lambda Expression

- Lambda expressions are basically syntactic sugar around functors, allowing one to define a functor without a name, right at the place where it is needed.
- Reduces the programmer work for using `std::for_each`, `std::transform` etc.

Example: `[](int &n){ n++; }`

Lambda Expression

Issues with Lambda Expression:

- Every lambda is of different type, even if signature is exactly same. This is because every lambda is different class behind the curtains.
- By extension of previous point, you cannot treat lambda expressions as objects. (wait, what?!?)
- You cannot apply a lambda (or a functor for that matter, without defining another functor) partially (is that a thing?)
- By extension of previous point, there is no currying with lambda and/or functor (yay, functional programming)

std::function

- `std::function` implements a type erasure mechanism that allows a uniform treatment of different functor types (but with same signature).
- `std::function` can be used as objects (it can be passed to, as well as returned from a function, can be put in a vector, map, array or some other data structure)
- It allows partiall application, and thereby currying.

std::bind

- `std::bind` allows partial application of function.
- Mostly used in combination of `std::function`. Together, these make pretty powerful tool.

You generally use it when you need to pass a functor to some algorithm. You have a function or functor that almost does the job you want, but is more configurable (i.e. has more parameters) than the algorithm uses. So you bind arguments to some of the parameters, and leave the rest for the algorithm to fill in:

```
// raise every value in vec to the power of 7
std::transform(vec.begin(), vec.end(), some_output,
std::bind(std::pow, _1, 7));
```

Got any questions?

Thank You