EDA with Modern C++

Jeff Trull

22 June 2017

About Me

Circuit Designer -> VLSI Designer -> CAD -> EDA -> contract software engineering

About Me

Circuit Designer -> VLSI Designer -> CAD -> EDA -> contract software engineering

Shameless Plug

I am available for consulting work. See contact info in this presentation.

Modern C++

To me

- Functional Programming
- Generic Programming
- Design Patterns
- The Standard Library
- Whatever is in C++11/14/17...

Example

What does this code do?

```
bool
foo(std::vector<int> const& values) {
    for ( int i = 0; i < values.size(); ++i ) {
        if ((values[i] % 2) == 0) {
            return true;
        }
    }
    return false;
}
```

Example

How about this?

```
int
bar(std::vector<int> const& values) {
    int x = 0;
    for ( int i = 0; i < values.size(); ++i ) {
        if ((values[i] % 2) == 0) {
            x++;
        }
    }
    return x;
}
```

Saying What You Mean

```
bool foo = any_of(values.begin(), values.end(),        // range
                  [](int i) { return i % 2 == 0; });   // criteria

int count = count_if(values.begin(), values.end(),
                     [](int i) { return i % 2 == 0; });
```

By using standard library algorithms we get:

- improved expressiveness
- separation of concerns (values vs. predicate)
- code reuse

CVE-2017-5689

Part of Intel's Active Management Technology code, as reverse engineered:

```
char * computed_response;    // expected auth response
char * user_response;        // actual response
char * response_length;      // length of actual response
if(strncmp(computed_response, user_response, response_length))
   deny_access();
```

By supplying an empty response attackers can ensure strncmp returns zero and access succeeds.

CVE-2017-5689

Part of Intel's Active Management Technology code, as reverse engineered:

```
char * computed_response;    // expected auth response
char * user_response;        // actual response
char * response_length;      // length of actual response
if(strncmp(computed_response, user_response, response_length))
   deny_access();
```

By supplying an empty response attackers can ensure strncmp returns zero and access succeeds.

CVE-2014-0160

a.k.a. "Heartbleed", because it's a bug in the TLS Heartbeat code. Paraphrasing slightly:

```
char * pl;                    // inbound message
char * bp;                    // outbound buffer
uint16_t payload;             // byte count supplied from network
memcpy(bp, pl, payload);
```

Attackers can claim a large byte count but supply a small message; the difference is read from the stack.

CVE-2014-0160

a.k.a. "Heartbleed", because it's a bug in the TLS Heartbeat code. Paraphrasing slightly:

```
char * pl;                   // inbound message
char * bp;                   // outbound buffer
uint16_t payload;            // byte count supplied from network
memcpy(bp, pl, payload);
```

Attackers can claim a large byte count but supply a small message; the difference is read from the stack.

Thinking on the right level

All programs have bugs. But we can make them less likely with the right structure, i.e. :

- Safely construct string
- Perform operation on string

is easier to show correctness than "perform byte operation using pointers X and Y, and count Z"

Generic Programming

Generic programming is a critical part of code reuse. It greatly expands the number of types we can use with a piece of code. We are already familiar with this in the standard containers:

```
vector<int> ints{7, 42, 1};
vector<string> strings{"abc", "123", "foo", "bar"};

// same code, different container
auto i_it = find(ints.begin(), ints.end(), 42);
assert(i_it != ints.end());
auto s_it = find(strings.begin(), strings.end(), "foo");
assert(s_it != strings.end());
```

But we can treat code generically as well!

Netlist Visitor

Have you had to write something like this?

```
void DoNothingC(Instance *) {}

void
traverse_netlist_c(Netlist * n,
                   void (*v)(Instance *)) {
    // ...
    // every time we encounter an instance, call the visitor
    Instance * inst;
    v(inst);          // always calls this pointer
}
```

This (along with void*) is the height of generality in C, but we can do better

Generic Visitor

```
struct DoNothingInstanceVisitor {
    void operator()(Instance *) const {}
};

template<typename InstanceVisitor = DoNothingInstanceVisitor>
void
traverse_netlist(Netlist * n,
                 InstanceVisitor v = InstanceVisitor()) {
    // ...
    // every time we encounter an instance, call the visitor
    Instance * inst;  // = whatever
    v(inst);          // optimized away for DoNothingInstanceVisitor
}
```

If a visitor is supplied, the compiler will inline the call and optimize it. Otherwise, the call is **completely** eliminated.

Concepts

Concepts represent requirements for a type to be used by other code. If the type meets those requirements, it is said to "model" the concept. For example:

- A **CopyConstructible** type can be copied
- A **Callable** type can be used as a function (via operator())
- A **RandomAccessIterator** has constant time indexing

And so on. Concepts are defined by what expressions are valid and what the semantics should be, not (as in most OOP schemes) by what base classes the type has.

Concept Checking

Meeting the Concepts

```
std::vector<int>  foo{4,3,2,1};
std::list<int>    bar{9,7,8,6};

std::sort(foo.begin(), foo.end());    // OK: RandomAccessIterator
std::sort(bar.begin(), bar.end());    // big compiler error
```

Concept Checking

Meeting the Concepts

```
std::vector<int>  foo{4,3,2,1};
std::list<int>    bar{9,7,8,6};

std::sort(foo.begin(), foo.end());     // OK: RandomAccessIterator
std::sort(bar.begin(), bar.end());     // big compiler error
```

Example (Errors)

```
error: no match for ‘operator-’
template argument deduction/substitution failed...
```

Concept Checking

Meeting the Concepts

```
std::vector<int>  foo{4,3,2,1};
std::list<int>    bar{9,7,8,6};

std::sort(foo.begin(), foo.end());     // OK: RandomAccessIterator
std::sort(bar.begin(), bar.end());     // big compiler error
```

Example (Errors)

```
error: no match for 'operator-'
template argument deduction/substitution failed ...
```

The Real Problem

`std::list` only supplies a **BidirectionalIterator**

Performance

Despite having a higher level of abstraction, generic code can produce better performance.
Let's quantify this.

Performance
Google Benchmark

Google Benchmark is a microbenchmarking tool, suitable for small to medium size functions where you want to compare implementation choices.
It works something like this:

```
static void BM_impl1(benchmark::State& state) {
  // do setup
  while (state.KeepRunning()) {
    // do work
    auto result = impl1();
    benchmark::DoNotOptimize( result );
  }
}

...
BENCHMARK(BM_impl1)
BENCHMARK(BM_impl2)
```

Performance
Google Benchmark

Google Benchmark is a microbenchmarking tool, suitable for small to medium size functions where you want to compare implementation choices.
It works something like this:

```
static void BM_impl1(benchmark::State& state) {
  // do setup
  while (state.KeepRunning()) {
    // do work
    auto result = impl1();
    benchmark::DoNotOptimize( result );
  }
}

...
BENCHMARK(BM_impl1)
BENCHMARK(BM_impl2)
```

Performance

Google Benchmark

Google Benchmark is a microbenchmarking tool, suitable for small to medium size functions where you want to compare implementation choices.
It works something like this:

```
static void BM_impl1(benchmark::State& state) {
  // do setup
  while (state.KeepRunning()) {
    // do work
    auto result = impl1();
    benchmark::DoNotOptimize( result );
  }
}

...
BENCHMARK(BM_impl1)
BENCHMARK(BM_impl2)
```

Performance
Google Benchmark

Google Benchmark is a microbenchmarking tool, suitable for small to medium size functions where you want to compare implementation choices.
It works something like this:

```
static void BM_impl1(benchmark::State& state) {
  // do setup
  while (state.KeepRunning()) {
    // do work
    auto result = impl1();
    benchmark::DoNotOptimize( result );
  }
}

...
BENCHMARK(BM_impl1)
BENCHMARK(BM_impl2)
```

Performance

Google Benchmark

Google Benchmark is a microbenchmarking tool, suitable for small to medium size functions where you want to compare implementation choices.
It works something like this:

```
static void BM_impl1(benchmark::State& state) {
  // do setup
  while (state.KeepRunning()) {
    // do work
    auto result = impl1();
    benchmark::DoNotOptimize( result );
  }
}

...
BENCHMARK(BM_impl1)
BENCHMARK(BM_impl2)
```

Performance

std::sort vs. qsort

qsort is the C way to sort... it takes a function pointer for the comparison operation, like this:

```
int
qs_comparator( void const * a, void const * b ) {
    // sort in *decreasing* order by reversing comparison
    return *reinterpret_cast<int const *>(b) - *reinterpret_cast<int
        const *>(a);
}
```

We can use it in a benchmark like this:

```
void qs_sorter( std::vector<int> v ) {
    qsort(v.data(), v.size(), sizeof(int), qs_comparator);
    benchmark::DoNotOptimize(v);
}
```

Performance
std::sort

std::sort is much the same but you can supply any function object that returns bool

```
auto std_comparator = [](int a, int b) { return b < a; };  //
    reversed

void std_sorter( std::vector<int> v ) {
    sort(v.begin(), v.end(), std_comparator);
    benchmark::DoNotOptimize(v);
}
```

Performance

std::sort

std::sort is much the same but you can supply any function object that returns bool

```
auto std_comparator = [](int a, int b) { return b < a; };  //
    reversed

void std_sorter( std::vector<int> v ) {
    sort(v.begin(), v.end(), std_comparator);
    benchmark::DoNotOptimize(v);
}
```

Performance

std::sort

std::sort is much the same but you can supply any function object that returns bool

```
auto std_comparator = [](int a, int b) { return b < a; };  //
    reversed

void std_sorter( std::vector<int> v ) {
    sort(v.begin(), v.end(), std_comparator);
    benchmark::DoNotOptimize(v);
}
```

Performance

Results

```
-------------------------------------------------------
Benchmark                  Time           CPU Iterations
-------------------------------------------------------
BM_qsort/10000        700347 ns     700353 ns        981
BM_stdsort/10000      453085 ns     453074 ns       1555
```

std::sort benefits from inlining in two ways:

- the indirect call is eliminated
- the comparison code can be integrated and optimized with the rest of the loop

Class Design

The class declaration reveals how you intend it to be used. There is powerful leverage in getting this right the first time.

- Smart pointers for indicating ownership
- Reference qualifiers allow efficient implementations
- Efficient and const-correct access to internal containers

unique_ptr

- low overhead
- appropriate for private heap variables
  - no need to remember to free it
  - but your class becomes move-only (not a big problem)
- appropriate for factory outputs
  - returning one indicates that ownership is transferred to the caller
  - you have to move it

unique_ptr


```
class Design;
class Instance;
class Cell;

std::vector<std::unique_ptr<Cell>>
readCells(std::string const & fileName);
```

shared_ptr

- shared ownership
- the pointer itself and its reference counts are thread safe
- somewhat slower due to extra state and synchronization
- useful for lifetime extension
- optimization: use make_shared if possible

shared_ptr

```
class Library {

    std::shared_ptr<Cell> findCell(std::string const& cellName);

private:
    std::vector<std::shared_ptr<Cell>> cells_;

};
```

The Library stores and hands out references to cells.

shared_from_this

- so you can supply shared references to yourself

```
class Design : public std::enable_shared_from_this<Design> {

public:
    std::shared_ptr<Instance>
    addInstance(std::shared_ptr<const Cell> cell, std::string name);

private:
    std::vector<std::shared_ptr<Instance>> instances_;
};
```

shared_from_this


```
std::shared_ptr<Instance>
Design::addInstance(std::shared_ptr<const Cell> cell,
                    std::string name) {
    instances_.emplace_back(
        std::make_shared<Instance>(std::move(name),
                                   cell,
                                   shared_from_this()));
    return instances_.back();
}
```

Instances are constructed by their parent Design, but they need to be able to supply a reference to their parent.

shared_from_this

```
std::shared_ptr<Instance>
Design::addInstance(std::shared_ptr<const Cell> cell,
                    std::string name) {
    instances_.emplace_back(
        std::make_shared<Instance>(std::move(name),
                                   cell,
                                   shared_from_this()));
    return instances_.back();
}
```

Instances are constructed by their parent Design, but they need to be able to supply a reference to their parent.

weak_ptr

- item may or may not be present
- caching
- breaking reference loops

```
class Instance {
public:
    Instance(std::string const & name,
             std::shared_ptr<const Cell> cell,
             std::weak_ptr<Design> parent);
    ...
```

Using a weak_ptr to refer to its parent solves the reference cycle and allows instances to be deleted.

Member Function Qualifiers

For constness

You can have nearly identical methods, one returning const ref (or iterator), one without. The compiler will choose based on context:

```
template<typename T>
struct Holder {
    using const_it     = typename container_t::const_iterator;
    using it           = typename container_t::iterator;

    const_it begin() const;
    const_it end() const;

    it begin();
    it end();
};
```

Member Function Qualifiers

For constness


```
Holder<int> const citems{1, 2, 3};
Holder<int>        items{1, 2, 3};

for(auto & i : items) {      // accesses non-const version of begin/
    end
    i = 5;                   // works
}
for(auto & i : citems) {     // accesses const version of begin and
    end
    // i = 5;                // fails to compile
}
```

Member Function Qualifiers

For memory efficiency

Let's add access to the entire set of items:

```
container_t const & contents() const &;
container_t & contents() &;
container_t && contents() &&;
```

The compiler will generate the right call depending on how the container_t is used.

Member Function Qualifiers

For memory efficiency


```
auto & itemsr = items.contents();
itemsr.push_back(4);

auto & itemscr = citems.contents();
// itemscr.push_back(4);    // will not compile

auto itemsmr = Holder<int>{1, 2, 3}.contents();    // move
    initialization
```

Do you avoid -O3?

Do you avoid -O3?

Optimization bugs are largely a thing of the past

If your code stops working after changing the optimization level, it is **usually** not a compiler bug. There are useful flags that can help you track it down.

Undefined Behavior

At higher optimization levels the compiler will assume that your code does not contain **undefined behavior**. This includes such things as:

- **signed** overflow or excessive shifting
- array access outside bounds
- pointers of different types referring to the same memory
- reading uninitialized values

John Regehr's blog has an excellent treatment of the subject, and the cppreference article on the subject has many good examples.

Flags

-**O3** activates a number of specific optimization flags depending on your compiler and version. You can find out which ones with:

```
g++ -c -Q -O3 --help=optimizers > /tmp/O3-opts
g++ -c -Q -O2 --help=optimizers > /tmp/O2-opts
diff /tmp/O2-opts /tmp/O3-opts | grep enabled
```

Understanding which one triggered the change can help you track down the problem.

Your compiler may also support disabling some of the assumptions directly, for example `-fno-strict-aliasing`, or adding special compile-time warnings, like `-Wstrict-overflow`, to let you know when it is using an assumption to optimize.

Sanitizers

Both gcc and clang now support **sanitizers**, which instrument your code to detect different kinds of problems. These have been very effective in finding bugs, particularly when coupled with fuzzing or a good unit test suite.
`-fsanitize=undefined` will enable a number of useful checks. Here is a famous (and wrong) example from the Apple Secure Coding Guide:

```
void apple(int m, int n) {
    size_t bytes = n * m;        // UB activated, potentially

    if (n > 0 && m > 0 && SIZE_MAX/n >= m) { // a check *after* UB
        may have occurred
        /* allocate "bytes" space */
        std::cout << "all good\n";
    } else {
        std::cerr << "rejecting allocation of excessive size\n";
    }
}
```

Sanitizers
-fsanitize=undefined


```
int main() {
    // trigger UB
    apple(1 << 16, 1 << 16);
}
```

With the **undefined** sanitizer enabled Clang (though not g++) will produce:

```
ub.cpp:9:22: runtime error: signed integer overflow: 65536 * 65536 cannot be represented in type
all good
```

Compiler Explorer

Let's take a closer look at that last piece of code

Summary

- Strive for *meaning* over *mechanics* in your code
- Let the compiler do its job

Resources

- John Regehr's blog
- From Mathematics to Generic Programming
- Modern C++ Design
- C++ Seasoning
- CppLang Slack Channel
- CppCon
- This presentation TODO
- Performance Resources
- Practical Performance Practices
- Herb Sutter on parameters