Systems Programming & Beyond using C++ and D

Andre Alexandrescu
andrei@erdani.com

Prepared for LASER Summer School 2012
Assumption: You are intelligent and talented
Won’t teach trivialities
Dwell on “diffs”
Focus on internalizing simple fundamentals with broad impact instead of rote memorization
  Only few errors are unforced in PL design
Rules of Engagement

- I ask question:
  - You raise hand
  - I acknowledge
  - You SHOUT

- You ask question:
  - You raise hand
  - I ignore, microphone runners acknowledge
  - You raise hand *with microphone in it*
  - I acknowledge
  - You talk

- No two consecutive questions
Why?
“OK, efficiency is the word. But then why wouldn’t I just use C or C + glue?”
C

- + Fast
- + Close to the machine

- – Weak abstraction mechanism
- – No safety
- – Verbose
C + glue

- + Flexibility
- + Separation of concerns
- + Get to use another language

- - Hybrid object model and memory layout
- - Impedance mismatch
- - Get to use another language
C++

- + Perfect integration
- + Better abstraction mechanisms than C
- + Expressiveness, range

- – Lack of flexibility
- – Size
- – Imperfections (e.g. safety)
C++’s Computational Model
- Part of many battles, most of which it won
- Retrofitted with armor and weaponry, sent back to battle
- Amazing combination of old technology and new human ingenuity

- Aimed niche: efficiency AND modeling power
  - All else is secondary
# Example

```cpp
#include <iostream>
int main() {
    std::cout << "Hello, world!" << std::endl;
}
```

- Incorrect
- Inefficient
- Relies on one exceedingly subtle language rule to work
- Surreptitiously uses inversion of control
- Works due to an exception disallowed everywhere else
Memory model

- Memory bedrock inherited largely from C
- You know where most everything is
- Exceptions:
  - Pointer to vtable
  - Pointer to parent class in virtual inheritance schemes
  - Address of constructor/destructor
  - Offsets of members in some classes
- Impacts:
  - Inherent efficiency
  - Approach to safety
Most code predictably translates into either:

- Simple machine code instructions
- Direct function calls
- Indirect function calls
- That’s pretty much it!

Exceptions:

- Constructor/destructor calls
- Exception throwing, transporting, catching
Constructor and destructor calls

- Ctor calls + copy ctor calls == dtor calls
- No ownership transfer semantics (C++11 fixes that)
- Invariant is difficult to maintain
  - Very often spurious calls are present
  - Benchmark: 1-7 ctor calls for the same code depending on compiler and optimization level
- Geared toward value semantics, not polymorphism
You don’t pay for what you don’t use

- RTTI/dynamic_cast adds per-class overhead even when never used.
- Exceptions have non-zero costs even when never used
  - Impossible to specify “nothrow” modularly
    - Even C functions are assumed to throw!
  - All exceptions are catchable
  - Complicates most function frames
Building Abstraction
What is abstraction?
What is abstraction?

Selective ignorance
Why is selective ignorance good?
Why is selective ignorance good?

Allows you to ignore *details*
Why is ignoring details good?
Why is ignoring details good?

MODULARITY
Most software engineering tools and techniques ultimately aim at improving modularity
Modularity

- Type systems
- Functions
- Information hiding
- Encapsulation
- Objects
- Design patterns
- Lazy evaluation
- Lambdas
- Monads
- Memcache!
- . . .
Modularity through abstraction

- Abstraction is a powerful modularity mechanism
- Power in nomenclature
- Stable: $\parallel$ abstractions $\parallel < \parallel$ details $\parallel$
- Allows keeping details in separation
- Allows improvement of details in separation
Abstraction liabilities

- Efficiency cost (“abstraction friction”)
- Cognitive cost
- Verbosity cost
- Informational cost
- Leakiness
C++ functions as abstraction mechanism

- Low barrier of entry, starting at $0
- Eager evaluation
- Genericity (templates)
- Specialization (template specialization)

- Ambiguous input and output parameters
- Uninformative signatures
  - No escaping information
  - No purity information
  - No safety information
  - No exception information
  - People use convention for all of the above
vector<int> v;
int e;
...
for (int i = 0; i < v.size(); ++i) {
    if (v[i] == e) {
        /* do work */ ...
        break;
    }
}

- Relies on vector
- Mixes searching details with work details
- Difficult to improve modularly
Example: find item in array, work with it

```cpp
vector<int> v;
int e;
...
auto i = find(v.begin(), v.end(), e);
if (i != v.end()) {
    ...
    /* do work */
    ...
}
```

- Works with many topologies
- Searching details separated from work details
- Can be improve modularly
find can be improved in isolation

template <class RIt, class V>
RIt find(RIt b, RIt e, const V& v) {
    for (; (e - b) & 3; ++b) {
        if (*b == v) return b;
    }
    for (; b != e; b += 4) {
        if (*b == v) return b;
        if (b[1] == v) return b + 1;
        if (b[2] == v) return b + 2;
        if (b[3] == v) return b + 3;
    }
}
Good functional abstractions

- Address frequent (**ENFORCE**) or specific (**multiwayUnion**) intent
- Sound-bite name (verb); easy to talk about
- Express inputs, outputs, lifetime, and ownership
- Hide operational details; easy to describe in one sentence
- Generally applicable; do not commit to irrelevant details (**find**)
- **Do not treat complexity as a detail**
Building abstractions with objects
C++ classes

- Class is the unit of encapsulation in C++
- Access control applies at class level
- Methods and friend functions have access to everything inside the class
  - Not the base class
- Key point: multiple instances of the class
- Key point: information hiding through data encapsulation
Class methods vs. free functions

- Conventional wisdom: methods are cool, free functions are so 1960s
- Yet:
  - Free functions *improve* encapsulation over methods
  - Free functions may be more general
  - Free functions decouple better
  - Free functions support conversions on their left-hand argument
Surprising fact #2

Making a function a method should be your last, not first, choice
Value vs. reference semantics

- Per discussion in Part 1:
  - Equal copies algebraically equivalent (think `int`)
  - Polymorphic values have “personality”—one instance, many referents
- Always decide early on value vs. reference
- Never define ambiguous-gender types
  - Exception: exceptions
class V {
public:
    V(); // nothrow
    V(const V & rhs); // deep copy
    ~V();
    V& operator=(const V& rhs);
    ... 
};
class R {
    R& operator=(const R& rhs);  // not defined
protected:
    R(const R & rhs);  // optional; deep copy
public:
    R();  // optional
    virtual ~R();  // essential
    virtual unique_ptr<R> clone() {
        CHECK(typeid(*this) == typeid(R));
        return new R(*this);
    }
    ...;
};
The C++ Standard Library
The C++ Standard Library

- “The reptilian brain:” The C Standard Library
- “Limbic system:” iostreams, string, complex, …
- “Neocortex:” The Standard Template Library
The STL

- Arguably the best software library ever conceived
- Focuses on asking the right question
- “What is the essence of the fundamental algorithms and containers?”
  - Arguably most everything else is aftermath
- Abstract: Leaving room for redesign is fail
- Frictionless: Leaving incentive for reimplementation is fail
**Bottleneck design**

- Many algorithms, no categorization
- Few iterators, strict categorization
  - input
  - output
  - forward
  - bidirectional
  - random access
- Many containers, loose categorization
STL algorithms

- Many fundamental algorithms
  - sorting
  - searching
  - accumulation
  - transformation
- Never operate directly on containers
- Never affect topology, only data
STL containers

- No hierarchy
- Federation of loosely categorized objects
- Linear: vector, list, deque
- Associative: set, map, unordered map
- Choice is by complexity/validity guarantees
The star of STL: iterators

- Mediate traffic between algorithms and containers
- Generalization of pointers

- Input: \(i == j, \ast i, \ast \ast i\)
- Forward: Input, \(\ast i = j\)
- Bidirectional: Forward, \(\ast \ast i\)
- Random access: Bidirectional, \(i[n], \ast \ast i\)
What do you need for...

- Linear search
- Subsequence brute force search
- Boyer-Moore search
- Quicksort
- Bring-to-front
- Stable remove
- Unstable remove
STL assets

- Abstraction power
- Efficiency
- Offers a model for other algorithms, containers, and iterators
- Plays perfectly into C++’s strengths
  - Most other languages cannot implement STL
  - This is remarkable
STL liabilities

- Lack of lambda functions makes many algorithms tenuous
  - Fixed in C++11
- Allocator design inadequate
- Syntax relatively verbose
- “C++ complete”
  - Ethos impossible to understand outside of C++
Practical advice

- Prefer STL to ad-hoc containers
  - Prefer our containers to the default ones
- Prefer iterators to indexed access
- Understand, use, and extend algorithms

- Avoid dogma beyond all else
Generic Programming
What is generic programming?
What is generic programming?

Generic Programming is the endeavor of finding the most abstract expression of a computation without losing its essence or efficiency.
Symptoms of generic programming deficit

- “Same drama, different stage”
  - Implement same algorithm several times
- *Design* with high-level notions but *implement* from first principles
- Can’t reuse a function without changing a few details
- Similar methods in unrelated hierarchies
C++ for generic programming?

- Template engine abstracts at no/low cost
- Post-hoc specialization possible
- Good efficiency to begin with
- No commitment to one paradigm

- Ad-hoc features, no unifying arc
- Uses vast array of odd features
Example: specializing on iterator

- Recall from earlier:

```cpp
template <class RIt, class V>
RIt find(RIt b, RIt e, const V& v) {
    for (; (e - b) & 3; ++b) {
        if (*b == v) return b;
    }
    for (; b != e; b += 4) {
        if (*b == v) return b;
        if (b[1] == v) return b + 1;
        if (b[2] == v) return b + 2;
        if (b[3] == v) return b + 3;
    }
}
```

- How do we specialize `find` only for random-access iterators?
Dispatching on category

- Classic technique: “tag dispatching”
- Not so classic: dispatch is static
- Now defining the base version working for all iterators

```cpp
template <class It, class V>
It find(It b, It e, const V& v,
        std::input_iterator_tag) {
    for (; e != b; ++b) {
        if (*b == v) return b;
    }
}
```
Now defining the specialized version

```cpp
template <class It, class V>
It find(It b, It e, const V& v,
        std::random_access_iterator_tag) {
    for (; (e - b) & 3; ++b) {
        if (*b == v) return b;
    }
    for (; b != e; b += 4) {
        if (*b == v) return b;
        if (b[1] == v) return b + 1;
        if (b[2] == v) return b + 2;
        if (b[3] == v) return b + 3;
    }
}
```
template <class It, class V>
It find(It b, It e, const V& v) {
    return find(b, e, v,
        typename iterator_traits<It>::iterator_category());
}
int main() {
  vector<int> vec;
  list<int> lst;
  find(vec.begin(), vec.end(), 42);
  find(lst.begin(), lst.end(), 42);
}
We’re about done!

C++
Prolegomena
Rules of Engagement

Why?

C++’s Computational Model

Building Abstraction

Building abstractions with objects

The C++ Standard Library

Generic Programming

We’re about done!