C++ best practices

Scott Snyder

BNL

Sep 24, 2019

Introduction

- This talk: Good C++ programming.
- It's a big topic and a big language, so have to pick just a few aspects to cover today.
  - Biased towards useful modern features, and by what typically gets used in ATLAS code.
- Additional references given at the end.

Outline:

RAII

- "Resource Acquisition Is Initialization"
- One of the most important C++ idioms.
- Often have code that falls into the pattern:
  1. Acquire some resource.
  2. Do something with it.
  3. Release the resource.
- One must ensure that the third step is always done, no matter what happens in the second step.
- RAII: The first and third steps can be encapsulated in a class, that acquires the resource in its constructor and frees it in its destructor.
- "Resource" here can be things like a dynamic memory allocation, an open file, or a lock.
- Or more abstract: timers, current working directory, etc.

RAII Example

- A concrete example: the Gaudi `ChronoSvc`. Used like this:

```
chronosvc->chronoStart ("someName");
ATH_CHECK( doSomething() );
chronosvc->chronoStop ("someName");
```

- Reminder: `ATH_CHECK(x)` is equivalent to (roughly)

```
if (x.isFailure()) {
  ATH_MSG_ERROR (#x " failed!");
  return StatusCode::FAILURE;
}
```

- But that's buggy! If `doSomething()` returns failure, then the `ATH_CHECK` macro will return the error up to the caller.
  - Without running `chonoStop`.
  - And what if `doSomething()` raises an exception?

RAII Example 2

- An attempt at coding this correctly might be something like this:

```
chronosvc->chronoStart ("someName");
StatusCode sc;
try {
  sc = doSomething();
}
except (...) {
  sc.ignore();
  chronosvc->chronoStop ("someName");
  throw;
}
chronosvc->chronoStop ("someName");
ATH_CHECK (sc);
```

- We have to do this *every time* we use `ChronoSvc`.
- This is crazy!

RAII Example 3

Solution with RAII:

```
class Chrono {
public:
  Chrono (const std::string& name, IChronoSvc* svc)
    : m_name(name), m_svc(svc)
  { m_svc->chronoStart (m_name); }
  ~Chrono() { m_svc->chronoStop (m_name); }
private:
  std::string m_name;          IChronoSvc* m_svc;
};
```

Use like:

```
  {  Chrono chrono ("someName", chronosvc);
     ATH_CHECK( doSomething() );  }
```

Much better!

Range-based for

The traditional way of iterating over a STL container looks like:

```
const Container* c = ...;
Container::const_iterator it  = c->begin();
Container::const_iterator end = c->end();
for (; it != end; ++it) {
  const Element* elt = *it;
  ...
```

Range-based equivalent:

```
for (const Element* elt : *c) {
  ...
```

Much easier to read!

Range-based for 2

- Can generally replace loops using iterators with range-based for unless you need to reference the iterators themselves within the loop.
- Can generalize further using range classes from boost. Example:

```
#include "boost/range/iterator_range.hpp"
class Foo { ...
  iterator foo_begin();
  iterator foo_end();
  typedef boost::iterator_range<iterator> range;
  range foo_range() { return range(foo_begin(), foo_end()); }
... }; ...
void func (Foo& foo)
{
  for (FooElt& elt : foo.foo_range()) {...
/// Also:
void func (iterator beg, iterator end) {
  for (Elt& elt : boost:make_iterator_range(beg, end)) {
```

C++20 should have ranges in the standard library.

auto

Sometimes C++ type names can get quite unwieldy:

```
std::map<int, std::string>& m = ...;
std::pair<std::map<int, std::string>::iterator, bool>
  ret = m.insert (std::make_pair (1, "x"));
if (ret.second) ...
```

We can use auto to let the compiler figure out the type itself:

```
auto ret =  m.insert (std::make_pair (1, "x"));
if (ret.second) ...
```

Can also use qualifiers:

```
// A reference to whatever's returned by something().
auto& ret =  something();
```

When to use auto

Some people advocate using auto "almost everywhere" (GOTW 94):

```
int x = 42;
unsigned long y = 42;
employee e (empid);
```

```
auto x = 42;
auto y = 42ul;
auto e = employee {empid};
```

But real concerns about readability, searchability, and robustness.

```
const Foo* doSomething();
...
auto foo = doSomething();
// What is the type of foo here?  You have to look up doSomething()
// in order to find out!  Makes it much harder to find all places
// where the type Foo gets used.

// If the return type of doSomething() changes, you'll get
// an error here, not at the doSomething() call.
foo->doSomethingElse();
```

Experience so far is that wide use of auto hurts readability.

When to use auto 2

A common error with auto:

```
std::vector<std::vector<int> > arr = ...;
for (auto v : arr) {
  for (auto elt : v) {
    doSomething (elt);
  }
}
```

- Will make copies of each std::vector<int>!
- Because auto v binds by value.
- Could fix with for (const auto& v : arr) {
- Still have to know the type, even with auto!

When to use auto 3

Current recommendations:

- Do not use `auto` in place of a (possibly qualified) simple type. Prefer 'constructor' syntax for class types.

```
int x = 42;
const Foo* foo = doSomething();
std::vector<int> v1 (10);
std::vector<int> v2 {1, 2, 3};
```

- But can use `auto` when initializing from a new-expression

```
auto p = new Foo; // Similar for unique_ptr
```

- Use `auto` for complex derived types generated by STL-like classes.
- Use `auto` in generic template implementations.

Inlining

- Inlining in C++ is a two-edged sword:
  - Judicious use of inlining in C++ is essential to good performance ("removing the abstraction penalty").
  - However, overuse of inlining can have disastrous effects on resources required to build the system, and can also degrade runtime performance. Can also complicate debugging.
- General rule: If at all in doubt, don't inline it. Then look at profiling data to decide what you should inline.
  - (Profiling data are generally useless in telling you what should *not* have been inlined!)
- Empty functions and single-line getters/setters *may* be good candidates for inlining.
- Anything more than a couple lines long should generally not be inlined before seeing profiling data.

Virtual functions

Virtual functions are very useful. But they are slower than non-virtual function calls. While the additional overhead to the call itself is small, more important is that virtual function calls disable many other optimizations that the compiler may be able to do based on knowledge of the called function.

Don't indiscriminately declare functions as virtual — only do it where it's actually needed. Simple functions such as accessors should not usually be declared virtual.

- Virtual functions should always be declared with `virtual`.
- When virtual functions are overridden in a derived class, always use `override`. That way, you get an error if the signature in the base class changes.

```
struct B { virtual ~B();  virtual void a();  virtual void b(int); }
struct D : public B {
  virtual void a() override;  // ok
  virtual void b(float) override;  // error; doesn't override
```

Virtual functions 2

Beware of hiding methods from a base class:

```
struct B { virtual ~B();  virtual void b(int); };
struct D : public B {
  virtual void b(float);
  // Not meant to override --- so no override keyword.
  // However, this hides b(int) from the base class.
```

To make the base class signatures visible in the derived class, add:

```
struct D : public B {
  ...
  using B::b;
```

Final

A method may be declared as 'final', meaning it may not be overridden in any derived class. Allows the compiler to optimize calls to that method. Consider:

```
struct IPoint {
  virtual ~IPoint() {}
  virtual float x() const = 0; virtual float r() const = 0;
  ... };

struct XYPoint : public IPoint {
  float m_x, m_y;
  virtual float x() const override { return m_x; }
  virtual float r() const override { return hypot (m_x, m_y); }
  ... };

struct PolarPoint : public IPoint {
  float m_r, m_phi;
  virtual float r() const override { return m_r; }
  virtual float x() const override { return m_r * cos(m_phi); }
  ... };
```

Final 2

Then in an example like this:

```
float sumx (const std::vector<const XYPoint*>& v)
{
  float sum = 0;
  for (const XYPoint* p : v)
    sum += p->x();
  return sum;
}
```

Even though the pointers are of the concrete `XYPoint` type, the `x()` call will always be virtual (because we may actually have a derived class of `XYPoint`).

If we declare the methods `final`:

```
struct XYPoint : public IPoint {
  virtual float x() const override final { return m_x; }
```

then the `x()` call can be inlined.

Memory allocation

- Dynamic memory allocation tends to be a problem area in C++ programming, both in terms of efficiency and correctness.
- `new`/`delete` are relatively expensive. A few allocations per event are probably negligible, but if you have many thousands, they start to add up.
- Be careful of memory leaks, use-after-free, etc.

Reducing calls to malloc

- Avoid dynamic memory allocation entirely when possible.
- Don't use `malloc`/`free`.
  - (Exceptions: external libraries, alignment.)
- Try to avoid using `new`/`delete`.
- Use `unique_ptr` where possible.

Do you need to allocate memory at all?

Often see an object allocated dynamically, used only locally, then deleted.

```
Foo* foo = new Foo;
doSomething (foo);
delete foo;
```

```
Foo foo;
doSomething (&foo);
```

- Application of RAII! Faster, shorter, easier to read, and less error-prone.
- Same for class members. Consider embedding an object by value in the class.

```
struct Foo {
  Bar* m_bar;
  Foo() { m_bar = new Bar; }
  ~Foo() { delete m_bar; }
```

```
struct Foo {
  Bar m_bar;
  Foo() { }
  ~Foo() { }
```

(Sometimes doesn't work due to dependency or initialization order considerations.)

Containers in loops

Beware containers (or anything that allocates memory) inside loops:

```
for (int i=0; i < BIG; i++) {
  std::vector<int> v (10);
  ... do something with v ...
}
```

This allocates/frees the container memory each time through the loop.
Consider moving the declaration outside the loop:

```
std::vector<int> v;
for (int i=0; i < BIG; i++) {
  v.clear();  v.resize(10);  // or fill with zeros
  ... do something with v ...
}
```

Now allocate/free happens only once.

C string conversions

Similarly, beware of implicit C string to C++ string conversions.

```
void somefunc (const std::string& s, int i);
...
for (int i=0; i < LOTS; i++)
  somefunc ("a string", i);
```

Each time through the loop, we construct the C++ string, allocating and freeing memory in the process. Move the conversion out of the loop:

```
std::string a_string ("a string");
for (int i=0; i < LOTS; i++)
  somefunc (a_string, i);
```

unique_ptr

- A 'smart pointer' type that formalizes exclusive ownership.
- Apply RAII to `new`/`delete`.
- `unique_ptr` takes ownership of a pointer.
- The pointer is deleted when `unique_ptr` is.
- `unique_ptr` can transfer ownership. Happens automatically for unnamed temporary objects or for function returns. Otherwise use `std::move`.
- Best to create using `make_unique` rather than `new`.

```
// Allocate a new Foo object and take ownership.
// (Could use auto for this declaration.)
std::unique_ptr<Foo> foo = std::make_unique<Foo>();
...
// The Foo object is deleted when foo goes out of scope.
```

unique_ptr examples

```
// Function takes ownership of its argument.
void use_foo (std::unique_ptr<Foo> ptr);

// Function transfers ownership to the caller.
std::unique_ptr<Foo> make_foo(int arg) {
  // Args passed to Foo constructor.
  auto ptr = std::make_unique<Foo> (arg);
  return ptr;  // No std::move here.
}

void test() {
  // Return value of make_foo moved to p1.
  std::unique_ptr<Foo> p1 = make_foo(1);

  // Pass ownership.   std::move is required here.
  use_foo (std::move(p1));
  // p1 is now null.
```

More on unique_ptr

- Using `unique_ptr` not only *implements* passing of ownership, but when used in function signatures, *documents* that as well.
- Old `auto_ptr` is similar, but had several problems fixed by `unique_ptr`.
  - `auto_ptr` is deprecated — don't use in new code.
- Other useful `unique_ptr` methods:
  - `T* get()`: Return the held pointer.
  - `T* release()`: Return the held pointer and give up ownership.
  - `void reset()`: Clear the pointer.
- StoreGate and DataVector methods can take `unique_ptr` where ownership is transferred.
- Use `unique_ptr` when an object holds a pointer to an object that it owns.
  - Ensures that you'll get a compilation error if you forget to copy the pointer in copy ctor / assignment!
- `unique_ptr` should be passed by value or (maybe) by non-const reference. Should not be passed by const reference (pass a pointer if ownership is not being transferred).

unique_ptr as member

```
struct C {
  std::unique_ptr<Foo> m_ptr;

  C(int arg) : m_ptr (CxxUtils::make_unique<Foo> (arg)) {}

  // You'll get an error if you forget to provide this
  // but try to use the copy constructor.
  C(const C& c)
    : m_ptr (CxxUtils::make_unique<Foo> (*c.m_ptr) {}
```

Special class members

Classes have a number of member functions that have special roles and can sometimes be automatically generated by the compiler. Here, we'll take a closer look at some of them. For concreteness, suppose we have class members:

```
class Foo { ...
  // Some ordinary data member.
  int m_x;

  // A pointer to an object that we own.
  Bar* m_bar;

  // That's the old-fashioned way.  For comparison,
  // we'll also see what happens if we have:
  //std::unique_ptr<Bar> m_bar;
```

Constructors

Each constructor should initialize *all* members. If you don't declare any constructors, a default constructor (taking no arguments) is automatically generated.

```
Foo() : m_x(0), m_bar(new Bar) {}
Foo(int x) : m_x(x), m_bar(new Bar) {}
```

A constructor can call another ("delegation"). This allows factoring out some otherwise repeated code. For example:

```
Foo() : Foo(0) {}
Foo(int x) : m_x(x), m_bar(new Bar) {}
```

If we used `unique_ptr`, then we would write instead

```
Foo(int x) : m_x(x), m_bar(CxxUtils::make_unique<Bar>()) {}
```

Destructors

- Should free all resources allocated in the constructor.
- Should be virtual if the class has any virtual functions.
- May be omitted if empty. Compiler will generate a default.

```
~Foo() { delete m_bar; }
```

If we used `unique_ptr`, then we don't have to do anything in the destructor.

Copy constructor

- Construct a new object as a copy of another.
- If omitted, compiler default will simply copy all members.
- If the compiler-generated default works, best to use it.
- If you do not want the default, disable with:

```
Foo (const Foo&) = delete;
```

- If the class manages resources, you generally need to implement the copy constructor.
- For our example, the default will be (silently!) wrong.

```
Foo (const Foo& o) : m_x (o.m_x), m_bar(new Bar(*o.m_bar)) {}
```

If we used `unique_ptr`, then if we leave out the copy constructor, then we'll get a compilation error instead of a silent failure. With `unique_ptr`:

```
Foo (const Foo& o) : m_x (o.m_x),
                     m_bar (CxxUtils::make_unique<Bar> (*o.m_bar)) {}
```

Assignment

- Implements operator=.
- Again, disable compiler-generated default if you don't want it.
- Should return reference to this object. Protect against self-assignment.

```
Foo& operator= (const Foo& o) {
  if (&o != this) {
    m_x = o.m_x;
    delete m_bar;    m_bar = new Bar (*o.m_bar);
  }
  return *this; }
```

With unique_ptr:

```
Foo& operator= (const Foo& o) {
  if (&o != this) {
    m_x = o.m_x;
    m_bar = CxxUtils::make_unique<Bar> (*o.m_bar);
  }  return *this; }
```

Move

'Move' is like a copy except that the source object may be destroyed in the process. If the object owns resources, they can be transferred, rather than copied.

```
// && is a special reference that can only bind to objects
// that will go away (such as temporaries).
Foo (Foo&& o) :
  m_x (o.m_x), m_bar (o.m_bar)  { o.m_bar = 0; }
```

If you use `unique_ptr`, the compiler-generated default should be sufficient. But if you have an explicit copy constructor, the default move constructor won't be generated unless you do:

```
Foo (Foo&&) = default;
```

There's also a move assignment operator:

```
Foo& operator= (Foo&&);
```

Summary for special member functions

- Define a default constructor if appropriate.
- Each constructor should initialize all members.
  - Use constructor delegation to factor out repeated code.
- Destructor should be virtual if any member functions are.
- Destructor should clean up any resources allocated by constructors.
  - Use RAII techniques like `unique_ptr` to avoid having to do this explicitly.
- Empty destructors may be omitted.
- Delete copy and assignment if it doesn't make sense to copy the objects.
- Otherwise, if the objects own resources, copy and assignment should be explicitly implemented (compiler default is very likely wrong).
  - But don't implement them if the compiler defaults will work.
- If the objects own resources, consider providing move constructor / assignment. If you manage resources using STL types, the compiler defaults may work, but if you also have a copy constructor / assignment, you'll have to explicitly request the defaults.

Function signatures

Different ways to pass parameters to a function:

```
void foo (int val, const Foo& ref, int& out);
```

- Pass by non-const reference for arguments to be modified. (And clearly document that the argument is modified.)
- Pass simple types by value.
  - Also `unique_ptr` and iterators.
- Pass other arguments by `const` reference.
  - Or by `const` pointer if they may be null.
- Reference/pointer arguments should always be declared as `const` if possible, as should the member function itself.
  - I.e, '`const int*`'.
  - Don't use `const_cast`! Outside of low-level code, having to use `const_cast` generally means that there's a design problem somewhere.
- Use `unique_ptr` to document transfers of ownership.
- Prefer returning objects by value.

Function signatures 2

- Sometimes a function argument is unused, but the signature can't be changed.
- Compilers will often generate warnings that the argument is unused.
- Best way to suppress that is to comment out the argument name.

```
void foo (int bar, int /*baz*/) { ...
```

- Can also use `[[maybe_unused]]` (for example, if the argument is used only in an `assert`).

```
void foo (int bar, [[maybe_unused]] int baz) {
  assert (baz < 10);
```

A few comments on style

With a project like ATLAS, *readability* is very important.

> *Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do. — DE Knuth*

Try to write code that can be read easily by someone else.

- Avoid overlong lines and long functions.
- Format code neatly and consistently.
- Follow ATLAS/local naming conventions. Try to be consistent with acronyms.
  - Private data members must start with '`m_`' (could also be '`s_`' for a `static` member). Don't use these prefixes in other contexts.
  - The ATLAS static checker will warn about violations of this.

Style: Comments

Each module should have a brief comment describing what it does. If it is documented in a note or wiki page, give a reference within the source code. Comment types:

- **What?** is the code doing? Not useful to just mirror the code:

```
++counter;  // Increment the counter.
```

  but higher-level descriptions are sometimes helpful.
- **How?** What strategies/methods are used? Give reference if appropriate.
- **Why?** were things done the way they were?
- **Why not?** There may be an 'obvious' way to solve a problem that doesn't actually work. Say why not to keep someone from wasting time in the future trying it.

Use Doxygen formatting for comments.

Style: Duplicated code

Tempting as it is to copy an existing piece of code and make a few changes, this is usually a bad thing to do.

- Results in much more code that must be read to understand the program.
- Later maintainers will end up having to compare the code line-by-line to understand any differences.
- Big risk that bugs will be fixed in only one copy and left in others.
- Increases amount of memory required and decreases locality.

Lots of duplicated code in ATLAS. It seems that every piece of code for say, MDT, has copies with MDT changed to RPC, TGC, ... Try to factor what's actually different from what's the same.

Style: Duplicated code

Prefer tables to repeated code. Bad:

```
 if((celltqavailable) && (cellen > 1000) &&
    (caloLyr == EMB1C)) hists[caloLyr]->Fill(celltime);
 if((celltqavailable) && (cellen > 3000) &&
     (caloLyr == EMB2A)) hists[caloLyr]->Fill(celltime);
...
```

Better:

```
  static const float ethresh[] = { ...,
        1000 /* EMB1C*/, 3000 /* EMB2A*/, ...
  if (celltqavailable && cellen > ethrsh[caloLyr])
    hists[caloLyr]->Fill(celltime);
...
```

A few references

- Many useful talks at previous tutorial sessions
- ATLAS coding guidelines.
  - `http://atlas-computing.web.cern.ch/atlas-computing/projects/qa/draft_guidelines.html`
- Scott Meyers, *Effective C++* and *More Effective C++*.
- Herb Sutter's "Guru of the week" blog:
  - `http://herbsutter.com/gotw/`
  - `http://www.gotw.ca/gotw/`
  - Parts published as *Exceptional C++* and *More Exceptional C++*.

More about move

Consider a simple class that manages memory:

```
struct Vector {
  double* m_x;  int m_n;
  Vector (int n = 0) : m_x(new double[n]), m_n(n) {}
  Vector (const Vector& v)
     : m_x(new double[v.m_n]), m_n(v.m_n) { <copy>; }
  ~Vector() { delete[] m_x; }
  Vector& operator= (const Vector& v) {
    if (this != &v) {
      delete[] m_x;
      m_n = v.m_n; m_x = new double[m_n];  <copy>
    }
    return *this;
  }
  void swap (Vector& v) {
    std::swap (m_x, v.m_x);  std::swap (m_n, v.m_n);
  }
  double& operator[] (size_t i) { return m_x[i]; }
};
```

Try to use it:

```
struct Foo {
  Vector m_v;
  void setVector (const Vector& v) { m_v = v; }
};

Vector makeVector();

void test (Foo& f)
{
  f.setVector (makeVector());
}
```

Q

- What is the efficiency problem here?
- How can it be improved?

Copy semantics

a ← b

a becomes a copy of b.

b is unchanged.

Move semantics

a ← b

a becomes a copy of b.

b is possibly destroyed.

A move operation *transfers* the internal state from one object to another, without making a copy.

```
void
Vector::move(Vector& other){
  m_x = other.m_x;
  m_n = other.m_n;
  other.m_x = 0;
}
```

```
void
Vector::move(Vector& other){
  // Often simpler
  this->swap (other);
}
```

Move is different than copy only for objects managing external resources.

An expression like '`v`' is called an *lvalue* — something that can appear on the left-hand side of an assignment. (Well, not exactly, but close enough for now.)

An expression like '`makeVector()`' is called an *rvalue* — something that can only appear on the right-hand side of an assignment. (Again, not exactly...)

Rvalues are empheral — temporary — while lvalues are named and stay until they go out of scope.

For an rvalue, it's safe to automatically change a copy into a move. This is not true for an lvalue.

Enter rvalue references...

```
  void setVector (Vector&& v);
```

The reference here will match an rvalue only. ('&&' is *not* a reference to a reference — that doesn't exist in C++.)

```
struct Foo {
  void setVector (const Vector& v) { m_v = v; }
  void setVector (Vector&& v) { m_v.swap (v); }
```

And use it:

```
Vector v = makeVector();
foo.setVector (v);                  // Copy
foo.setVector (makeVector());       // Move
```

What if you want to move from an lvalue? Use `std::move`.

```
Vector v = makeVector();
foo.setVector (v);                  // Copy
foo.setVector (std::move (v));      // Move
```

Despite its name, `std::move` does not move anything itself — it's effectively a cast-to-rvalue.

Careful — v below is a lvalue

```
struct Bar {
  Vector m_v;
  Bar(Vector&& v) : m_v(v) {} // Copy to m_v
```

Use std::move for a move

```
struct Bar {
  Vector m_v;
  Bar(Vector&& v) : m_v(std::move(v)) {} // Move to m_v
```

What about function returns?

```
Vector makeVector()
{
  Vector v(10);
  ...
  return v;
}
```

Q: `v` above is an lvalue. Should we use `std::move` to avoid a copy?

A: No. Function returns are a special case, and will always match a rvalue reference. Using `std::move` might inhibit return value optimization, so you shouldn't use it.

Summary

- If an object manages resources outside of itself (usually memory), consider providing move operations.
- A move constructor and assignment operator should be provided in addition to the copy constructor and standard assignment operator.

```
Vector::Vector (Vector&& v)
   : m_x(0), m_n(0) { this->swap (v); }
Vector& Vector::operator= (Vector&& v) {
  if (this != &v) this->swap (v);
  return *this;
}
```

- Use `std::move` when you want to move the contents of some named object (except for function returns).