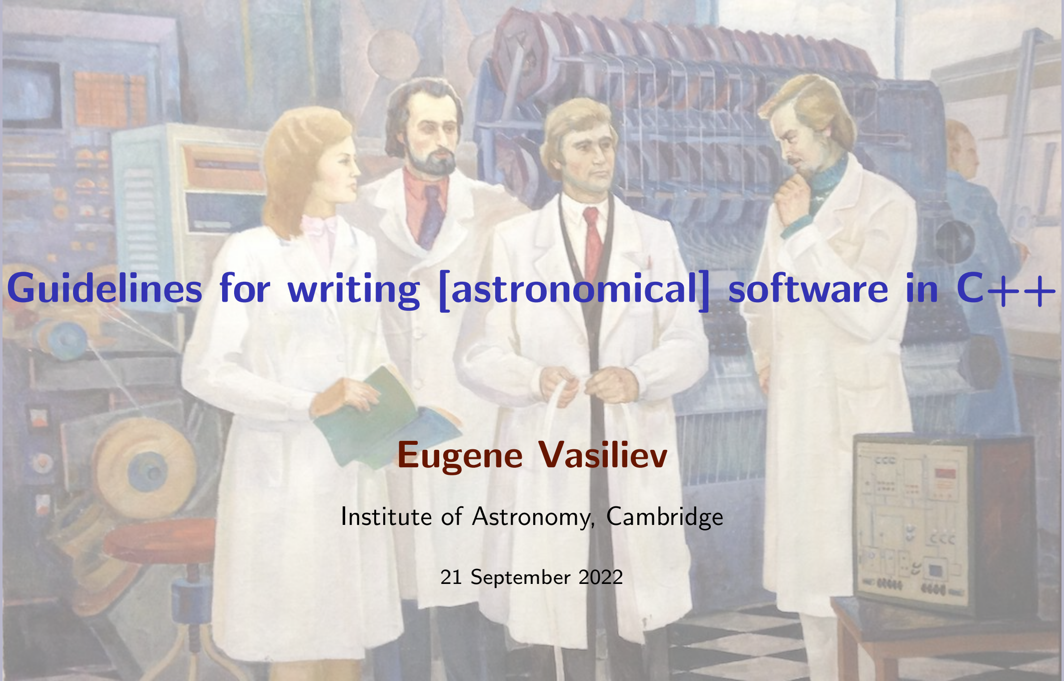

# Guidelines for writing [astronomical] software in C++

**Eugene Vasiliev**

Institute of Astronomy, Cambridge

21 September 2022

## Context (personal experience)

- I've been programming in C++ since high school, writing various kinds of science-oriented software;
- in 2015, I became a postdoctoral researcher at the University of Oxford, focusing on theoretical stellar dynamics in application to our Galaxy;
- at the time, the team had already written an assorted collection of programs and modules (with partly overlapping functionality) for performing specific tasks such as computation of orbits in galactic potentials, etc.;
- I examined the existing code base and found numerous architectural, stylistic and efficiency problems;
- after some initial planning, I decided to rewrite most of it largely from scratch;
- this is how the AGAMA framework for galactic dynamics was born.

In this presentation, I illustrate a few specific examples of code improvement (see Appendix A.1 in AGAMA reference for a more detailed discussion).

## [style] 0. self-documenting code in headers

When declaring a function in the .h file, it is not *necessary* to name the arguments (only their types are used). But of course it's *much better* to have meaningful names, which literally document the usage!

OLD

```
// Torus.h
class Torus {
  //....
  // Returns distance from given position to nearest point on torus
  double DistancetoPoint(const Position&, double&, double&) const;
};

// Torus.cc
double Torus::DistancetoPoint(const Position &pos,
    double &deltar, double &deltaz) const
```

NEW

Just write the full function signature in both .h and .cc files:)

## [style] 1. proper use of namespaces

Namespaces organize the code into individual "subject domains" and thus assist in readability (and also prevent name clashes).
An often used syntactic sugar is to write
`using namespace std;`
but it should be avoided (5 extra symbols don't justify possible confusion).

OLD

```
using namespace std;

class Matrix {
    //....
};

double norm(const vector &x);
double norm(const Matrix &x);
```

NEW

```
namespace math {
  class Matrix {
    //....
  };
}  // end namespace math

double norm(const std::vector &x);
double norm(const math::Matrix &x);
```

## [design] 2. Abstract classes and methods

In the object-oriented programming paradigm, we often start a class hierarchy with an *abstract* base class, which has no meaningful default implementation for [some of] its methods. These should be marked as abstract, triggering a compilation error if one tries to use them, rather than failing at runtime.

OLD

```
class PotentialBase{
  virtual double Phi(Position &x) const
  {
    // to be overridden by derived classes
    return 0;
  }
  virtual double dPhidx(Position &x) const
  {
    cout << "dPhidx(): Not implemented!";
    exit(-1);  return 0;
  }
};
```

NEW

```
class PotentialBase{
  virtual ~PotentialBase() {}
  /// potential at point x
  virtual double Phi
      (Position &x) const = 0;
  /// x-derivative of potential
  virtual double dPhidx
      (Position &x) const = 0;
};
```

## [design] 3. Immutability and const-correctness

If most or all objects in a computationally-oriented programs are *immutable*, this greatly simplifies parallelization (taken to the extreme, we arrive at the functional programming camp). This prohibits storage of intermediate calculations, so some redesign might be needed to retain efficiency.

OLD

```
struct Solver{
  Solver() {
    //some initial heavy setup
  }
  void init(double param) {
    //precompute some complicated
    //intermediate data from param
  }
  double compute(double x) const {
    //result depends on x and
    //precomputed intemediate data
  }
};
```

NEW

```
struct SolverStep1{
  SolverStep1() { /*initial setup*/ }
};
struct SolverStep2{
  SolverStep2(const SolverStep1& s1,
    double param) {
    //precompute intermediate data
  }
  double compute(double x) const {
    //use data precomputed in the
    //constructors of both classes
  }
};
```

## [design] 4. Structures with named members instead of arrays

When using a fixed-length sequence of values with different physical meaning, it may be better to represent it by a structure with appropriately named fields, rather than a plain array/vector.

For example, a 3d Cartesian point is clearly $x, y, z$, but it's not immediately obvious whether Cylindrical point is $R, z, \phi$ or $R, \phi, z$ or something else? Named fields lift the ambiguity and prevent a possible confusion between two length-3 arrays (one cannot accidentally pass an argument of a wrong type).

OLD

```
void convertCartesianToCylindrical(
  const double pointcar[3],
  double pointcyl[3]);
```

NEW

```
struct PointCar {
  double x, y, z;
};
struct PointCyl {
  double R, phi, z;
};
PointCyl convertCartesianToCylindrical
  (const PointCar& pointcar);
```

## [efficiency] 5. Exceptions or error codes?

Exceptions are a great mechanism for propagating errors (with meaningful messages!) up the execution stack, but they are not entirely cost-free.
If a heavily used piece of code has a high chance of firing an exception, consider replacing it with a special return value (NaN), which is also “contagious” and propagates to upper levels, where it could be caught (the downside being that it does not carry any detailed information about the source of error).

OLD

```
double findRoot(const IFunction& fnc,
  double x_lower, double x_upper)
{
  double fa = fnc(x_lower);
  double fb = fnc(x_upper);
  if(fa * fb > 0)
    throw std::runtime_error(
    "endpoints do not bracket root");
```

NEW

```
double findRoot(const IFunction& fnc,
  double x_lower, double x_upper)
{
  double fa = fnc(x_lower);
  double fb = fnc(x_upper);
  if(!( (fa <= 0 && fb >= 0) ||
        (fa >= 0 && fb <= 0) ) )
    // endpoints do not bracket root
    return NAN;
```

## [efficiency] 6. Use specialized algorithms when necessary

In some cases (in a "hot" piece of code), it may make sense to replace a general-purpose algorithm with a specialized and faster implementation.

OLD

```
/// solve the Kepler equation for eta:
/// phase = eta - ecc * sin(eta)
double solveKepler(
  double ecc, double phase)
{
  return findRoot(
    FncKepler(ecc,phase), 0, 2*M_PI);
}

struct FncKepler: public IFunction {
  const double Ecc, Phase;
  FncKepler(double ecc, double phase):
    Ecc(ecc), Phase(phase) {}
  double operator() (double eta) const
  { return eta-Ecc*sin(eta)-Phase; }
};
```

NEW

```
double solveKepler(double ecc, double phase)
{
  double eta, sineta, coseta, deltaeta = 0;
  sincos(phase, sineta, coseta);
  eta = phase + ecc * sineta / sqrt(1 -
    ecc * (2*coseta - ecc));  // initial guess
  do {  // Halley's method, converges cubically
    sincos(eta, sineta, coseta);
    double f  = eta - ecc * sineta - phase;
    double df = 1.  - ecc * coseta;
    double d2f= ecc * sineta;
    deltaeta  = -f*df / (df*df - 0.5*f*d2f);
    eta      += deltaeta;
  } while(fabs(deltaeta) > 1e-5);
  return eta;
}
```

## [efficiency] 7. Avoid dynamical memory allocation when possible

In "hot" code, dynamically allocating and destroying temporary workspace arrays on the heap may incur significant runtime costs, and when the size of this workspace is not too large, could be replaced with stack-based allocation.

OLD

```
void computeSomething1(int N) {
  double tmp[] = new double(N);
  // do stuff.. but if it throws
  // an exception, memory is lost
  delete[] tmp;
}

void computeSomething2(int N) {
  std::vector<double> tmp(N);
  // do stuff..
  // tmp is automatically freed
  // even if an exception arises
}
```

NEW

```
void computeSomething3(int N) {
  std::vector<double> tmpvec;
  double *tmp;
  if(N < 1000) {  // on stack
    tmp = static_cast<double*>(
      alloca(N*sizeof(double)));
  } else {  // on heap
    tmpvec.resize(N);
    tmp = &tmpvec.front();
  }
  // in any case, no need to
  // manually deallocate "tmp"
}
```

## [numerics] 8. Avoid catastrophic cancellation

Expressions involving subtraction of two very close floating-point values need to be recast into mathematically equivalent forms that avoid loss of precision (or Taylor-expanded when appropriate).

OLD

```
// get the smallest root of
// x^2 + 2*b*x + c
double solve_quadratic_x1(
  double b, double c)
{
  return -b - sqrt(b*b - c);
}
double lnx_over_x(double x)
{
  return log(1+x) / x;
}
```

NEW

```
double solve_quadratic_x1(double b, double c)
{
  return b > 0 ? -b - sqrt(b*b - c) :
    c / (-b + sqrt(b*b - c));
}

double lnx_over_x(double x) {
  return x > 7e-4 ? log(1+x) / x :
    // accurate asymptotic expansion at x->0
    (1 + x*(-1./2 + x*(1./3 + x*(-1./4))));
}
```

## [numerics] 9. Choose the right integration variable

When numerically integrating a function, it is crucial to ensure that the integrand is well-behaved, and implement an appropriate variable transformation if needed. In the integral $\int_0^1 \mathrm{d}x\, f(x)/\sqrt{x(1-x)}$, which has integrable endpoint singularities, we substitute $x = y^2\,(3-2y)$ to remove them from the denominator.

OLD

```
struct Integrand: public IFunction{
  double operator()(double x) const
  {
    return f(x) / sqrt((1-x)*x);
  }
};
```

NEW

```
struct Integrand: public IFunction{
  double operator()(double y) const
  {
    return f(y*y * (3 - 2*y)) /
      sqrt((3 - 2*y) * (1 + 2*y));
  }
};
```

## Further reading

### General

- Martin R., 2008, *Clean code*, Prentice Hall
- McConnell S., 2004, *Code complete*, Microsoft press

### C++-specific

- Meyers S., 2005, *Effective C++*, Addison–Wesley
- Sutter H., Alexandrescu A., 2004, *C++ coding standards*, Addison–Wesley

### Scientific computation-specific

- Press W., Teukolsky S., Vetterling W., Flannery B., *Numerical recipes*, 3rd ed., 2007, Cambridge University press