# Better coding (C++ grammar)

34th Collaboration Meeting

**Valérian Sibille**

MIT

21st February 2018

Massachusetts Institute of Technology

KATRIN
KARLSRUHE TRITIUM NEUTRINO EXPERIMENT

# Outline

# Why code better?

## Group

- Understand
- Maintain
- Re-use

## Yourself

- Trust
- Frustration

## Skills (long-term)

- Efficient
- Marketable

# Outline

# Don't copy paste

Orginal (awful) code

```
int main(){

  TH1D* hist2 = new TH1D("hist2", "", 100, 0, 10);
  TH1D* hist3 = new TH1D("hist3", "", 100, 0, 10);

  //fill histograms...

  for(unsigned k = 0; k < hist2->GetNbinsX() - 2; ++k)
    hist2->SetBinContent(k, hist2->GetBinContent(k+2));

  for(unsigned k = 0; k < hist3->GetNbinsX() - 3; ++k)
    hist3->SetBinContent(k, hist3->GetBinContent(k+3));

  //save hist2 and hist3 to a file...

  return 0;
}
```

# Don't copy paste

After first **git commit**

```
int main(){

  TH1D* hist2 = new TH1D("hist2", "", 100, 0, 10);
  TH1D* hist3 = new TH1D("hist3", "", 100, 0, 10);

  //fill histograms...

  for(unsigned k = 0; k < hist2->GetNbinsX() - 2; ++k)
    hist2->SetBinContent(k, hist2->GetBinContent(k+2));

  for(unsigned k = 1; k < hist3->GetNbinsX() - 2; ++k)
    hist3->SetBinContent(k, hist3->GetBinContent(k+3));

  //save hist2 and hist3 to a file...

  return 0;
}
```

# Don't copy paste

## Make functions

- Instant propogation
- The more you call it, the better it gets
- Makes the intent clear

```
void shift(TH1D* hist, int shift_value){

  for(unsigned k = 1; k < hist->GetNbinsX() +1 - shift_value; ++k)
    hist->SetBinContent(k, hist->GetBinContent(k + shift_value));

}
```

⇒ Call it

```
...
shift(hist2, 2);
shift(hist3, 2);
```

# Don't copy paste whole files

**Shift.cpp**

```
void shift(TH1D* hist, int shift_value){

  for(unsigned k = 1; k < hist->GetNbinsX() +1 - shift_value; ++k)
    hist->SetBinContent(k, hist->GetBinContent(k + shift_value));

}
```

**Shift10.cpp**

```
void shift(TH1D* hist, int shift_value){

  for(unsigned k = 1; k < hist->GetNbinsX() +1 - shift_value; ++k)
    hist->SetBinContent(k, 10 * hist->GetBinContent(k + shift_value));

}
```

# Don't use **new**

```
TH1D* add_background(const TH1D* h, unsigned background){

  TH1D* result = new TH1D
  {"", "", h->GetNbinsX(), h->GetBinLowEdge(1), h->GetBinLowEdge(h->GetNbinsX()+1)};

  for(unsigned k = 1; k <result->GetNbinsX() +1 ; ++k)
  result->SetBinContent(k, h->GetBinContent(k) + background);

  return result;

}
```

```
TFile ROOTfile("spectrum.root");
TH1D* spectrum = dynamic_cast<TH1D*>(ROOTfile.Get("spectrum"));

std::vector<TH1D*> spectra;
for(unsigned k = 0; k < 1e5; ++k)
  spectra.push_back(add_background(spectrum, k));
```

# Don't use **new**

**new** leads to

- Memory leaks
- Segmentation faults
- Implicit casts
- Ugly syntax

```
TH1D add_background(const TH1D& h, unsigned background){

  TH1D result
  {"", "", h.GetNbinsX(), h.GetBinLowEdge(1), h.GetBinLowEdge(h.GetNbinsX()+1)};

  for(unsigned k = 1; k <result.GetNbinsX() +1 ; ++k)
    result.SetBinContent(k, h.GetBinContent(k) + background);

  return result;
}
```

# Don't use **new**

**Use references and implicit move / copy-elision**

- Simpler syntax
- No runtime errors
- No memory management
- Fast

```
TFile ROOTfile("spectrum.root");
TH1D spectrum = *dynamic_cast<TH1D*>(ROOTfile.Get("spectrum"));

std::vector<TH1D> spectra;
for(unsigned k = 0; k < 1e5; ++k)
  spectra.push_back(add_background(spectrum, k));
```

$\Rightarrow$ Probability $P(\text{need\_new}) < 10^{-3}$

# Don't (ab)use pointers

## Pointers are a pain

- Raw pointers are tied to **new**
- Unique (smart) pointers are verbose
- Shared (smart) pointers are costly

```
struct event{
  double energy;
  double pitch_angle;
};

struct cut{
  virtual bool reject(const event& event) const = 0;
  virtual ~cut(){};
};
```

# Don't (ab)use pointers

- Unless you really *need dynamic* polymorphism
- Analysis cuts with different parameters

```
struct low_energy_cut : cut{

  double low_cut;
  low_energy_cut(double low_cut):low_cut(low_cut){}
  bool reject(const event& event) const{
    return event.energy > low_cut;
  }

};
```

# Don't (ab)use pointers

- Pitch angle selections uses two parameters

```
struct pitch_angle_cuts : cut{

  double lower_bound, upper_bound;
  pitch_angle_cuts(double lower_bound, double upper_bound)
  :lower_bound(lower_bound),upper_bound(upper_bound){}
  bool reject(const event& event) const{
    return event.pitch_angle > lower_bound && event.pitch_angle < upper_bound;
  }

};
```

# Don't (ab)use pointers

- Combine cuts of different types into a `std::vector`

```
bool pass_default_cuts(const event& event){

  std::vector<std::unique_ptr<cut>> cuts;
  cuts.push_back(std::make_unique<low_energy_cut>(18.5));
  cuts.push_back(std::make_unique<pitch_angle_cuts>(0, std::atan(1)));

  for(const auto& cut : cuts)
    if(cut->reject(event)) return false;
  return true;

}
```

# Don't copy-paste (Slight return)

## Build systems

- No temporary files for the git server
- Strip irrelevant build instructions when copying files from other projects

# Outline

# Use a short scope

```
bool large_scope(){

  double temperature;
  double retarding_potential;
  double FDP_rate;
  bool drunk_operator;
  std::vector<double> pixel_counts;
  TH1D* spectrum;
  int n, m;
  //3 loops + 5 nested if then else

  std::string line;
  std::ifstream input("data.txt");
  TCanvas* my_beautiful_canvas;
  TGraph* graphCountsFinal;
  //2 nested loops + 2 if then else

  bool rubbish_analysis = true;
  return rubbish_analysis;

}
```

## Use a short scope (< 5 variables)

```
std::vector<double> read_data(std::ifstream& input){

        std::vector<double> pixel_counts;
        //read data
        return pixel_counts;
}

void process_data(std::vector<double>& pixel_counts){

// perform analysis;
}

TGraph make_plot(const std::vector<double>& processed_data){

        TGraph graph(processed_data.size());
        //cosmetics and assignement
        return graph;
}
```

# Use a short scope ($< 5$ variables)

## Advantages

- Human scope $\sim$ 5 variables
- Fewer name collisions
- Clear intentions
- Easier reading
- Easier debugging

# Have a reasonable number of arguments ($< 5$ variables)

**Too many arguments**

- Harder to parse for humans
- Needless details
- Must change `beta_spectrum` declaration for each addition

**beta_spectrum.hpp**

```
beta_spectrum(double temperature, double purity, double DTfraction,
double BA, double BS, double Bmax,
TH1* eloss_distribiution,
const char* fsd_T2_filename, const char* fsd_DT_filename,
const char* finalstatedistributionhtfilename);
```

## Have a reasonable number of arguments ($< 5$ variables)

**beta_spectrum.hpp**

```
beta_spectrum(gas_characteristics, magnetic_fields,
energy_loss, final_states);
```

**elsewhere.hpp**

```
struct gas_characteristics{
  double temperature;
  double trititum_purity;
  double DT_fraction;
};

struct magnetic_fields{
  double analysing_plane;
  double source;
  double pinch;
};
```

$\Rightarrow$ Make aggregates / structures / classes

# Make sure that a class instance is always valid

## Make class invariants

- More faithful representation
- Avoid pollution in processing methods

```
class gas_characteristics{
  double temperature;
  double T_purity;
  double DT_fraction;
public:
  gas_characteristics(double temperature, double T_purity, double DT_fraction)
  :temperature(temperature),T_purity(T_purity),DT_fraction(DT_fraction){
  if(temperature < 0 || tritium_purity < 0 || DT_fraction  < 0
     || tritium_purity > 1 || DT_fraction > 1)
    throw std::invalid_argument("Invalid gas characteristics.");
  }
};
```

# Write short methods

```
double SSCIntegratedSpectrum::CalculateSliceRate(double qU, size_t iSlice){
Initialize();
SSCDifferentialSpectrum* diffSpec = GetDifferentialSpectrum();
KMathKahanSum<double> sliceRate = 0.0;
IntegratorType integrator;
integrator.SetMethod(fIntegrationMethod);
integrator.SetPrecision(fIntegrationPrecision);
integrator.ThrowExceptions(false);
integrator.FallbackOnLowPrecision(true);
SSCSlice& rbslice = fSource->GetSlice(iSlice);
for (size_t jRing = 0; jRing < rbslice.GetNRings(); jRing++) {
// ring filter
if (fRingFilter >= 0 && jRing != (size_t) fRingFilter)
continue;
SSCRing& ring = rbslice.GetRing(jRing);
for (size_t kSegment = 0; kSegment < ring.GetNSegments(); kSegment++) {
// segment filter
if (fSegmentFilter >= 0 && kSegment != (size_t) fSegmentFilter)
continue;
const SSCSegment& segment = ring.GetSegment(kSegment);
int32_t voxelNumber = segment.GetVoxelNumber();
SSCResponseBase* thisSegmentsResponse = nullptr;
if (fResponse) {
if (voxelNumber > -1) {
if (fResponseCache.size() <= (size_t) voxelNumber)
fResponseCache.resize( voxelNumber+1, nullptr );
// create a new response in the cache:
if (!fResponseCache[voxelNumber]) {
fResponseCache[voxelNumber] = fResponse->Clone();
fResponseCache[voxelNumber]->SetParameter(SSCEParameter::RingFilter, jRing);
fResponseCache[voxelNumber]->SetParameter(SSCEParameter::SegmentFilter, kSegment);
}
thisSegmentsResponse = fResponseCache[voxelNumber];
```

# Write short methods $\ll 20$ lines

```
else {
thisSegmentsResponse = fResponse;
}
// trigger initialization of scattering probabilites
thisSegmentsResponse->SetVoxel(&segment);
thisSegmentsResponse->Initialize();
}
const double localPotential = segment.GetPotential();
const double effectiveQU = qU - localPotential;
const double localTemperature  = segment.GetTemperature();
double thetaMax = asin( sqrt( segment.GetB() / kDefaultBMax ) );
if (thisSegmentsResponse)
thetaMax = thisSegmentsResponse->GetThetaMax();
// set doppler properties
fCompDiffSpectra.SetParameters({
{ SSCEParameter::Temperature, localTemperature },
{ SSCEParameter::ThetaMax, thetaMax }
});
// set diff. spectra concentrations
for (size_t i = 0; i < fCompDiffSpectra.size(); ++i) {
const SSCEGasSpecies gasSpec = fCompDiffSpectra.GetGasSpecies(i);
fCompDiffSpectra.SetBulkVelocity(gasSpec, segment.GetBulkVelocity(gasSpec));
//number of molecules in pixel (only inside magnetic flux (A_S) )
const double nMolecules = segment.GetA() * segment.GetColumnDensity(gasSpec);
double nNuclei = (double) SSCGasDynamicsBase::GetNumberOfNuclei(gasSpec) * nMolecules;
if (gasSpec == SSCEGasSpecies::MolecularTritium)
nNuclei *= GetTritiumPurity();
fCompDiffSpectra.SetConcentration(i, nNuclei);
}
// configure integration limits
set<double> integrationLimits;
/* Upper integration limit:
* Endpoint energy minus (plus) positive (negative) neutrino mass + safety margin (e.g. doppler effect)
```

# Write short methods $\ll 20$ lines

```
const set<double> globalLimits = fCompDiffSpectra.GetIntegrationLimits();
//                    KDEBUG("Integration limits from child spectra in slice " << iSlice << ": " << globalLimits);
const set<double> diffSpecLimits = (diffSpec) ? diffSpec->GetIntegrationLimits() : set<double>();
const double globalLowerLimit   = (globalLimits.size() <= 1) ? 0.0 : *globalLimits.begin();
const double globalUpperLimit   = (globalLimits.empty())     ? 0.0 : *globalLimits.rbegin();
const double diffSpecUpperLimit = (diffSpecLimits.empty())   ? 0.0 : *diffSpecLimits.rbegin();
/* Lower integration limit:
* Retarding potential (minus HV fluctuations).
*/
double responseLowerLimit = effectiveQU;
/* First intermediate integration limit:
* Upper end of transmission function slope.
*/
double transmissionEdge = responseLowerLimit;
if (thisSegmentsResponse) {
const pair<double, double> transmissionLimits = thisSegmentsResponse->GetTransmissionLimits(qU, localPotential);
responseLowerLimit = transmissionLimits.first;
transmissionEdge = transmissionLimits.second;
}
responseLowerLimit = max(responseLowerLimit, globalLowerLimit);
transmissionEdge = max(transmissionEdge, globalLowerLimit);
if (globalUpperLimit - responseLowerLimit < 1E-12)
continue;
// prepare final states
//                    if (fsd /*&& std::is_same<IntegratorType, KMathIntegratorThreaded>::value*/)
//                        fsd->PrepareUpToEnergy(endpoint-lowerLimit);
if (globalUpperLimit - transmissionEdge > 1.5 )
integrationLimits.insert(transmissionEdge);
//                    // insert another integration limit at the second transmission edge
//                    if (diffSpec && diffSpecUpperLimit - transmissionEdge > 24.0) {
//                        const double scatteringEdge = transmissionEdge + 12.0;
//                        integrationLimits.insert(scatteringEdge);
//                    }
```

# Write short methods $\ll 20$ lines

```
if (diffSpecUpperLimit > 0.0) {
if (fKeepEOIntegrationLimitStable) {
/**
* THIS FIX IS REALLY IMPORTANT!
* Trying to keep the upper diff. spec. integration limit "fixed" for
* numerical stability at lower retarding potentials:
*/
double stableEndpointLimit = diffSpecUpperLimit;
if (diffSpec && diffSpecUpperLimit - transmissionEdge > 8.5) {
constexpr double jumpInterval = 2.0;
//                  stableEndpointLimit = (floor(endpoint/jumpInterval)+0.5)*jumpInterval;
stableEndpointLimit = 18575.0 + jumpInterval/2.0;
while (stableEndpointLimit > diffSpecUpperLimit)
stableEndpointLimit -= jumpInterval;
while (stableEndpointLimit < diffSpecUpperLimit)
stableEndpointLimit += jumpInterval;
}
if (stableEndpointLimit > *integrationLimits.rbegin())
integrationLimits.insert(stableEndpointLimit);
}
else  {
if (diffSpecUpperLimit > *integrationLimits.rbegin())
integrationLimits.insert(diffSpecUpperLimit);
}
}
if (globalUpperLimit > *integrationLimits.rbegin())
integrationLimits.insert(globalUpperLimit);
// now add dedicated integration intervals for difficult "peak" like regions
for (double limit : globalLimits) {
if (limit <= responseLowerLimit || limit == globalUpperLimit || limit == diffSpecUpperLimit)
continue;
integrationLimits.insert(limit);
}
KMathKahanSum<double> integrationSum = 0.0;
uint32_t cIntSteps = 0;
```

# Write short methods $\ll 20$ lines

```
//                cout << setprecision(12) << "qU: " << qU << "; pot: " << localPotential << endl;
//                cout << "int-limits: " << integrationLimits << endl;
auto it1 = integrationLimits.begin(), it2 = it1;
if (it2 != integrationLimits.end())
std::advance(it2, 1);
// now that we know our integration limits/segments, start integrating each of those
for (uint16_t iInterval = 0; it2 != integrationLimits.end(); ++it1, ++it2, ++iInterval) {
const uint32_t minIntSteps = (nIntervals != fIntegrationMinSteps.size())
? *max_element(fIntegrationMinSteps.begin(), fIntegrationMinSteps.end())
: fIntegrationMinSteps[iInterval];
const uint32_t maxIntSteps = (nIntervals != fIntegrationMaxSteps.size())
? *max_element(fIntegrationMaxSteps.begin(), fIntegrationMaxSteps.end())
: fIntegrationMaxSteps[iInterval];
integrator.SetMinSteps( minIntSteps );
integrator.SetMaxSteps( maxIntSteps );
auto integrand = [&] (double E) -> double {
return Integrand(E, qU, localPotential, voxelNumber);
};
//                auto integrand = bind(&SSCIntegratedSpectrum::Integrand, this, placeholders::_1, qU, localPotential);
integrator.SetRange( *it1, *it2 );
double integResult = integrator.Integrate( integrand );
//                        if (fIntegrator.NumberOfSteps() > 4097) {
//                            KWARN("Numerical integration required " << fIntegrator.NumberOfSteps() << " steps:\n"
//                                  "Interval: " << *it1 << " - " << *it2 << " ("<< (*it2-*it1) << "); E0: " << endpoint << "; mNu2: " << mNu2);
//                        }
if (std::isnan(integResult)) {
//                            throw SSCException() << "Numerical integration failed in interval " << *it1 << " - " << *it2
//                                                 << " at qU = " << fEffectiveQU << ".";
KERROR("Numerical integration failed in interval " << *it1 << " - " << *it2
<< " at qU-potential = " << effectiveQU << ".");
}
else {
//                            KDEBUG("  [" << *it1 << " - " << *it2 << "] " << integResult << " (" << fIntegrator.NumberOfSteps() << ")");
```

# Tell me a story!

```
template <class FunctionConvolver>
void SmearFinalStateFile(std::istream& input, std::ostream& output,
const char* delimiter, const FunctionConvolver& functionConvolver){

  auto data = read<Point<double>>(input);
  auto result = functionConvolver.convolution(std::cbegin(data), std::cend(data));
  for(const auto& point : result) output<<point<<delimiter;

}
```

# Outline

# Take home

## Don't

- Copy-paste code
- Copy-paste files
- Use **new**
- Overuse pointers
- Push rubbish


## Do

- Few variables at scope
- Use functions
- Few number of arguments
- Use data structures
- Write short methods / functions

*Code should read as a story!*

# Thank you

Thank you for your attention