ATLAS C++ coding guidelines*

Version 2.1

Scott Snyder (BNL) Shaun Roe (CERN)
and the former ATLAS Quality Control group

January 1, 2026

Table of Contents

*Correspondence to snyder@bnl.gov.
Generated from https://gitlab.cern.ch/ssnyder/coding-rules/-/blob/master/rules.md by pandoc version 3.1.11.1 on January 9, 2026
The current version of this document is avalailable at https://atlassoftwaredocs.web.cern.ch/coding-guidelines.

1 Introduction

This note gives a set of guidelines and recommendations for coding in C++ for the ATLAS experiment.

There are several reasons for maintaining and following a set of programming guidelines. First, by following some rules, one can avoid some common errors and pitfalls in C++ programming, and thus have more reliable code. But even more important: a computer program should not only tell the machine what to do, but it should also tell *other people* what you want the machine to do. (For much more elaboration on this idea, look up references on "literate programming," such as [1].) This is obviously important any time when you have many people working on a given piece of software, and such considerations would naturally lead to code that is easy to read and understand. Think of writing ATLAS code as another form of publication, and take the same care as you would writing up an analysis for colleagues.

This document is derived from the original ATLAS C++ coding standard, ATL-SOFT-2002-001 [2], which was last revised in 2003. This itself derived from work done by the CERN "Project support team" and SPIDER project, as documented in CERN-UCO/1999/207 [3]. These previous guidelines have been significantly revised to take into account the evolution of the C++ language [4], current practices in ATLAS, and the experience gained over the past decade.

Some additional useful information on C++ programming may be found in the references [5–13].

This note is not intended to be a fixed set of rigid rules. Rather, it should evolve as experience warrants.

2 Naming

This section contains guidelines on how to name objects in a program.

2.1 Naming of files

- **Each class should have one header file, ending with "`.h`", and one implementation file, ending with "`.cxx`".** [source-naming]

Some exceptions: Small classes used as helpers for another class should generally not go in their own file, but should instead be placed with the larger class. Sometimes several very closely related classes may be grouped together in a single file; in that case, the files should be named after whichever is the "primary" class. A number of related small helper classes (not associated with a particular larger class) may be grouped together in a single file, which should be given a descriptive name. An example of the latter could be a set of classes used as exceptions for a package.

For classes in a namespace, the namespace should not be included in the file name. For example, the header for `Trk::Track` should be called `Track.h`.

Implementation ("`.cxx`") files that would be empty may be omitted.

The use of the "`.h`" suffix for headers is long-standing ATLAS practice; however, it is unfortunate since language-sensitive editors may then default to using "C" rather than "C++" mode for these files. For Emacs, it can help to put a line like this at the start of the file:

```
// This file is really -*- C++ -*-.
```

2.2 Meaningful names

- **Choose names based on pronounceable English words, common abbreviations, or acronyms widely used in the experiment, except for loop iteration variables.** [use-meaningful-names]

  For example, `nameLength` is better than `nLn`.

  Use names that are English and self-descriptive. Abbreviations and/or acronyms used should be of common use within the community.

- **Do not create very similar names.** [no-similar-names]

  In particular, avoid names that differ only in case. For example, `track` / `Track`; `c1` / `cl`; `XO` / `X0`.

2.3 Required naming conventions:

Generally speaking, you should try to match the conventions used by whatever package you're working on. But please try to always follow these rules:

- **Use only ASCII characters in identifier names** [ascii-identifiers]

  This is what C++ calls the basic character set. Specifically, identifiers should use only the characters a-z, A-Z, 0-9, and underscore.

  Handling of non-ASCII characters is implementation-defined. While many compilers can indeed handle extended (unicode) characters, not all tools may process them correctly. Some characters may not display correctly, depending on a user's local installation. Further, it is often not obvious how to type an arbitrary unicode character that one sees displayed, especially since there exist distinct characters that look very similar or identical.

- **Use prefix `m_` for private/protected data members of classes.** [data-member-naming]

  Use a lowercase letter after the prefix `m_`.

  An exception for this is xAOD data classes, where the member names are exposed via ROOT for analysis.

- **Do not start any other names with `m_`.** [m-prefix-reserved]

- **Do not start names with an underscore. Do not use names that contain anywhere a double underscore.** [system-reserved-names]

  Such names are reserved for the use of the compiler and system libraries.

  The precise rule is that names that contain a double underscore or which start with an underscore followed by an uppercase letter are reserved anywhere, and all other names starting with an underscore are reserved in the global namespace. However, it's good practice to just avoid all names starting with an underscore. An exception is the use of a single underscore to indicate something that's structurally required but ignored.

2.4 Recommended naming conventions

If there is no already-established naming convention for the package you're working on, the following guidelines are recommended as being generally consistent with ATLAS usage.

- **Use prefix `s_` for private/protected static data members of classes.** [static-members]

Use a lowercase letter after the prefix `s_`.

- **The choice of namespace names should be agreed to by the communities concerned.** [namespace-naming]

  Don't proliferate namespaces. If the community developing the code has a namespace defined already, use it rather than defining a new one. Examples include `Trk::` for tracking and `InDet::` for inner detector.

- **Use namespaces to avoid name conflicts between classes.** [use-namespaces]

  A name clash occurs when a name is defined in more than one place. For example, two different class libraries could give two different classes the same name. If you try to use many class libraries at the same time, there is a fair chance that you will be unable to compile and link the program because of name clashes. To solve the problem you can use a namespace.

  New code should preferably be put in a namespace, unless typical ATLAS usage is otherwise. For example, ATLAS classes related to the calorimeter conventionally start with "Calo" rather than being in a C++ namespace.

- **Start class and enumeration types with an uppercase letter.** [class-naming]

```
class Track;
enum State { green, yellow, red };
```

- **Type alias (`typedef`) names should start with an uppercase letter if they are public and treated as classes.** [typedef-naming]

```
using TrackVector =
  std::vector<MCParticleKinematics*>;
```

- **Alternatively, a type alias (`typedef`) name may start with a lowercase letter and end with `_t`.** [typedef-naming-2]

  This form should be reserved for type names which are not treated as classes (e.g., a name for a fundamental type) or names which are private to a class.

```
using mycounter_t = unsigned int;
```

- **Start names of variables, members, and functions with a lowercase letter.** [variable-and-function-naming]

```
double energy;
void extrapolate();
```

Names starting with s_ and m_ should have a lowercase letter following the underscore.

Exceptions may be made for the case where the name is following standard physics or mathematical notation that would require an uppercase letter; for example, uppercase E for energy.

- **In names that consist of more than one word, write the words together, and start each word that follows the first one with an uppercase letter.** [compound-names]

```
class OuterTrackerDigit;
double depositedEnergy;
void findTrack();
```

Some ATLAS packages also use the convention that names are entirely lowercase and separated by underscores. When modifying an existing package, you should try to be consistent with the existing naming convention.

- **All package names in the release must be unique, independent of the package's location in the hierarchy.** [unique-package-names]

If there is a package, say "A/B/C", already existing, another package may not have the name "D/E/C" because that "C" has already been used. This is required for proper functioning of the build system.

- **Underscores should be avoided in package names.** [no-underscores-in-package-names]

The old ATLAS rule was that a _ should be used in package names when they are composites of one or more acronyms, e.g. TRT_Tracker, AtlasDB_*. Underscores should be avoided unless they really help with readability and help in avoiding spelling mistakes. TRTTracker looks odd because of the double "T". Using underscores in package names will also add to confusion in the multiple-inclusion protection lines.

- **Acronyms should be written as all uppercase.** [uppercase-acronyms]

```
METReconstruction, not MetReconstruction
MuonCSCValidation, not MuonCscValidation
```

Unfortunately, existing code widely uses both forms.

3 Coding

This section contains a set of items regarding the "content" of the code. Organization of the code, control flow, object life cycle, conversions, object-oriented programming, error handling, parts of C++ to avoid, portability, are all examples of issues that are covered here.

The purpose of the following items is to highlight some useful ways to exploit the features of the programming language, and to identify some common or potential errors to avoid.

3.1 Organizing the code

- **Header files must begin and end with multiple-inclusion protection.** [header-guards]

```
#ifndef PACKAGE_CLASS_H
#define PACKAGE_CLASS_H
// The text of the header goes in here ...
#endif // PACKAGE_CLASS_H
```

Header files are often included many times in a program. Because C++ does not allow multiple definitions of a class, it is necessary to prevent the compiler from reading the definitions more than once.

The include guard should include both the package name and class name, to ensure that is unique.

Don't start the include guard name with an underscore; such names are reserved to the compiler.

Be careful to use the same string in the `ifndef` and the `define`. It's useful to get in the habit of using copy/paste here rather than retyping the string.

Some compilers support an extension `#pragma once` that has similar functionality. A long time ago, this was sometimes faster, as it allowed the compiler to skip reading headers that have already been read. However, modern compilers will automatically do this optimization based on recognizing header guards. As `#pragma once` is nonstandard and has no compelling advantage, it is best avoided.

In some rare cases, a file may be intended to be included multiple times, and thus not have an include guard. Such files should be explicitly commented as such, and should usually have an extension other than "`.h`"; "`.def`" is sometimes used for this purpose.

- **Use forward declaration instead of including a header file, if this is sufficient.** [forward-declarations]

```
class Line;
class Point
{
public:
  // Distance from a line
  Number distance(const Line& line) const;
};
```

Here it is sufficient to say that `Line` is a class, without giving details which are inside its header. This saves time in compilation and avoids an apparent dependency upon the `Line` header file.

Be careful, however: this does not work if `Line` is actually an alias (as is the case, for example, with many of the xAOD classes).

- **Each header file must contain the declaration for one class only, except for embedded or very tightly coupled classes or collections of small helper classes.** [one-class-per-source]

This makes your source code files easier to read. This also improves the version control of the files; for example the file containing a stable class declaration can be committed and not changed any more.

Some exceptions: Small classes used as helpers for another class should generally not go in their own file, but should instead be placed with the larger class. Sometimes several very closely related classes may be grouped together in a single file; in that case, the files should be named after whichever is the "primary" class. A number of related small helper classes (not associated with a particular larger class) may be grouped together in a single file, which should be given a descriptive name. An example of the latter could be a set of classes used as exceptions for a package.

- **Implementation files must hold the member function definitions for the class(es) declared in the corresponding header file.** [implementation-file]

  This is for the same reason as for the previous item.

- **Ordering of `#include` statements.** [include-ordering]

  `#include` directives should generally be listed according to dependency ordering, with the files that have the most dependencies coming first. This implies that the first `#include` in a "`.cxx`" file should be the corresponding "`.h`" file, followed by other `#include` directives from the same package. These would then be followed by `#include` directives for other packages, again ordered from most to least dependent. Finally, system `#include` directives should come last.

```
// Example for CaloCell.cxx
// First the corresponding header.
#include "CaloEvent/CaloCell.h"
// The headers from other ATLAS packages,
// from most to least dependent.
#include "CaloDetDescr/CaloDetDescrElement.h"
#include "SGTools/BaseInfo.h"
// Headers from external packages.
#include "CLHEP/Geometry/Vector3D.h"
#include "CLHEP/Geometry/Point3D.h"
// System headers.
#include <cmath>
```

  Ordering the `#include` directives in this way gives the best chance of catching problems where headers fail to include other headers that they depend on.

  Some old guides recommended testing on the C++ header guard around the `#include` directive. This advice is now obsolete and should be avoided.

```
// Obsolete --- don't do this anymore.
#ifndef MYPACKAGE_MYHEADER_H
# include "MyPackage/MyHeader.h"
#endif
```

The rationale for this was to avoid having the preprocessor do redundant reads of the header file. However, current C++ compilers do this optimization on their own, so this serves only to clutter the source.

- **Do not use “`using`” directives or declarations in headers or prior to an `#include`.** [no-using-in-headers]

  A `using` directive or declaration imports names from one namespace into another, often the global namespace.

  This does, however, lead to pollution of the global namespace. This can be manageable if it’s for a single source file; however, if the directive is in a header file, it can affect many different source files. In most cases, the author of these sources won’t be expecting this.

  Having `using` in a header can also hide errors. For example:

```
// In first header A.h:
using namespace std;

// In second header B.h:
#include "A.h"

// In source file B.cxx
#include "B.h"
...
vector<int> x;  // Missing std!
```

  Here, a reference to `std::vector` in `B.cxx` is mistakenly written without the `std::` qualifier. However, it works anyway because of the `using` directive in `A.h`. But imagine that later `B.h` is revised so that it no longer uses anything from `A.h`, so the `#include` of A.h is removed. Suddenly, the reference to `vector` in `B.cxx` no longer compiles. Now imagine there are several more layers of `#include` and potentially hundreds of affected source files. To try to prevent problems like this, headers should not use `using` outside of classes. (Within a class definition, `using` can have a different meaning that is not covered by this rule.)

For similar reasons, if you have a `using` directive or declaration in a "`.cxx`" file, it should come after all `#include` directives. Otherwise, the `using` may serve to hide problems with missing namespace qualifications in the headers.

This rule does not apply when `using` is used to define a type alias (similarly to `typedef`).

3.2 Control flow

- **Do not change a loop variable inside a `for` loop block.** [do-not-modify-for-variable]

  When you write a for loop, it is highly confusing and error-prone to change the loop variable within the loop body rather than inside the expression executed after each iteration. It may also inhibit many of the loop optimizations that the compiler can perform.

- **Prefer range-based `for` loops.** [prefer-range-based-for]

  Prefer a range-based for to a loop with explicit iterators. That is, prefer:

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

to

```
std::vector<int> v = ...;
for (std::vector<int>::const_iterator it = v.begin();
     it != v.end();
    ++it)
{
  doSomething (*it);
}
```

In some cases you can't make this replacement; for example, if you need to call methods on the iterator itself, or you need to manage multiple iterators within the loop. But most simple loops over STL ranges are more simply written with a range-based for.

As of C++20, you can initialize additional variables in a range-based for:

```
void foo (const std::vector<float>& v) {
  for (int i = 0; float f : v) {
    std::cout << i++ << " " << f << "\n";
  }
}
```

- **Switch statements should have a `default` clause.** [switch-default]

  A `switch` statement should have a `default` clause, rather than just falling off the bottom, as a cue to the reader that this case was expected.

  In some cases, a `switch` statement may be on a `enum` and includes `case` clauses for all possible values of the `enum`. In such cases, a `default` cause is not required. Recent compilers will generate warnings if some elements of an `enum` are not handled in a `switch`. This mitigates the risk that a `switch` does not get updated after a new `enum` value is added.

- **Each clause of a `switch` statement must end with `break`.** [switch-break]

  If you must "fall through" from one switch clause to another (excluding the trivial case of a clause with no statements), this should be explicitly indicated using the `fallthrough` attribute. This should, however, be a rare case.

```
switch (case) {
case 1:
  doSomething();
  [[fallthrough]];
case 2:
  doSomethingMore();
  break;
...
```

Recent compilers will warn about such constructs unless you use the attribute or a special comment. For new code, using the attribute is preferred.

- **An `if`-statement which does not fit in one line must have braces around the conditional statement.** [if-bracing]

  This makes code much more readable and reliable, by clearly showing the flow paths.

  The addition of a final else is particularly important in the case where you have if/else-if. To be safe, even single statements should be explicitly blocked by `{}`.

```
if (val == thresholdMin) {
  statement;
}
else if (val == thresholdMax) {
  statement;
}
else {
  statement;  // handles all other (unforeseen) cases
}
```

- **Do not use `goto`.** [no-goto]

  Use `break` or `continue` instead.

  This statement remains valid also in the case of nested loops, where the use of control variables can easily allow to break the loop, without using `goto`.

  `goto` statements decrease readability and maintainability and make testing difficult by increasing the complexity of the code.

  If `goto` statements must be used, it's better to use them for forward branching than backwards, and the functions involved should be kept short.

3.3 Object life cycle

3.3.1 Initialization of variables and constants

- **Declare each variable with the smallest possible scope and initialize it at the same time.** [variable-initialization]

It is best to declare variables close to where they are used. Otherwise you may have trouble finding out the type of a particular variable.

It is also very important to initialize the variable immediately, so that its value is well defined.

```
int value = -1;   // initial value clearly defined
int maxValue;     // initial value undefined ...
                  //   NOT recommended
```

- **Avoid use of “magic literals” in the code.** [no-magic-literals]

If some number or string has a particular meaning, it’s best to declare a symbol for it, rather than using it directly. This is especially true if the same value must be used consistently in multiple places.

Bad example:

```
class A
{
  ...
  TH1* m_array[10];
};

void A::foo()
{
  for (int i = 0; i < 10; i++) {
    m_array[i] = dynamic_cast<TH1*>
      (gDirectory()->Get (TString ("hist_") +
       TString::Itoa(i,10)));
}
```

Better example:

```
class A
{
  ...

  static const s_numberOfHistograms = 10;
  static TString s_histPrefix;
```

```
  TH1* m_array[s_numberOfHistograms];
};

TString s_histPrefix = "hist_";

void A::foo()
{
  for (int i = 0; i < s_numberOfHistograms; i++) {
    TString istr = TString::Itoa (i, 10); // base 10
    m_array[i] = dynamic_cast<TH1*>
     (gDirectory()->Get (s_histPrefix + istr);
}
```

It is not necessary to turn *every* literal into a symbol. For example, the '10' in the example above in the `Itoa` call, which gives the base for the conversion, would probably not benefit from being made a symbol, though a comment might be helpful. Similarly, sometimes `reserve()` is called on a `std::vector` before it is filled with a value that is essentially arbitrary. It probably also doesn't help to make this a symbol, but again, a comment would be helpful. Strings containing text to be written as part of a log message are also best written literally.

In general, though, if you write a literal value other than '0', '1', `true`, `false`, or a string used in a log message, you should consider defining a symbol for it.

- **Use the <numbers> header for mathematical constants.** [math-constants]

  Basic mathematical constants are available in the header <numbers>. Use these in preference to the M_ constants from math.h or explicit definitions:

```
#include <numbers>
#include <cmath>
double f (double x) {
  return std::sin (x * std::numbers::pi);
}
```

- **Declare each type of variable in a separate declaration statement, and**

do not declare different types (e.g. `int` and `int*`) in one declaration statement. [separate-declarations]

Declaring multiple variables on the same line is not recommended. The code will be difficult to read and understand.

Some common mistakes are also avoided. Remember that when you declare a pointer, a unary pointer is bound only to the variable that immediately follows.

```
int i, *ip, ia[100], (*ifp)();   // Not recommended

// recommended way:
LoadModule* oldLm = 0;  // pointer to the old object
LoadModule* newLm = 0;  // pointer to the new object
```

Bad example: both `ip` and `jp` were intended to be pointers to integers, but only `ip` is — `jp` is just an integer!

```
int* ip, jp;
```

- **Do not use the same variable name in outer and inner scope.** [no-variable-shadowing]

  Otherwise the code would be very hard to understand; and it would certainly be very error prone.

  Some compilers will warn about this.

- **Be conservative in using `auto`.** [using-auto]

  The `auto` keyword allows one to omit explicitly writing types that the compile can deduce. Examples:

```
auto x = 10;  // Type int deduced
auto y = 42ul;  // Type unsigned long deduced.
auto it = vec.begin();  // Iterator type deduced
```

Some authorities have recommended using `auto` pretty much everywhere you can (calling it “auto almost always”). However, our experience has been that this adversely affects the readability and robustness of the code. It generally helps a reader to understand what the code is doing if the type is apparent, but with `auto`, it often isn't. Using `auto` also makes it more difficult to find places where a particular type is used when searching the code with tools like LXR. It can also make it more difficult to track errors back to their source:

```
const Foo* doSomething();
...  a lot of code here ...
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();
```

`auto` has also been observed to be a frequent source of errors leading to unwanted copies of objects. For example, in this code:

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

each element of the outermost vector will be copied, as the assignment to v will be done by value. One would probably want:

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

but having to be aware of the type like this kind of obviates the motivation for using `auto` in the first place. Using the type explicitly makes this sort of error much more difficult.

The current recommendation is to generally not use `auto` in place of a (possibly-qualified) simple type:

```
// Use these
int x = 42;
```

```
const Foo* foo = doSomething();
for (const CaloCell* cell : caloCellContainer) ...
Foo foo (x);

// Rather than these
auto x = 42;
auto foo = doSomething();
for (auto cell : caloCellContainer) ...
auto foo = Foo {x};
```

There are a few sorts of places where it generally makes sense to use `auto`.

– When the type is already evident in the expression and the declaration would be redundant. This is usually the case for expressions with `new` or `make_unique`.

```
// auto is fine here.
auto foo = new Foo;
auto ufoo = std::make_unique<Foo>();
```

– When you need a declaration for a complicated derived type, where the type itself isn't of much interest.

```
// Fine to use auto here; the full name of the
// type is too cumbersome to be useful.
std::map<int, std::string> m = ..;
auto ret = m.insert (std::make_pair (1, "x"));
if (ret.second) ....
```

– In the case where a class method returns a type defined within the class, using the `auto` syntax to write the return type at the end of the signature can make things more readable when the method is defined out-of-line:

```
template <class T> class C {
public:
  using ret_t = int;
  ret_t foo();
};
```

```

// Verbose: the return type is interpreted at the
// global scope, so it needs to be qualified with
// the class name.
template <class T>
typename C<T>::ret_t C<T>::foo() ...

// With this syntax, the return type is
// interpreted within the class scope.
template <class T>
auto C<T>::foo() -> ret_t ...
```

  – `auto` may also be useful in writing generic template code.

In some cases, C++20 allows declaring a template function without the `template` keyword when the argument is declared as `auto`:

```
auto fn (auto x) { return x + 1; }
```

It is recommended to avoid this syntax for public interfaces.

In general, the decision as to whether or not to use `auto` should be made on the basis of what makes the code easier to *read*. It is bad practice to use it simply to save a few characters of typing.

3.3.2 Constructor initializer lists

- **Let the order in the initializer list be the same as the order of the declarations in the header file: first base classes, then data members.** [member-initializer-ordering]

  It is legal in C++ to list initializers in any order you wish, but you should list them in the same order as they will be called.

  The order in the initializer list is irrelevant to the execution order of the initializers. Putting initializers for data members and base classes in any order other than their actual initialization order is therefore highly confusing and can lead to errors.

  Class members are initialized in the order of their declaration in the class; the order in which they are listed in a member initialization list makes no difference whatsoever! So if you hope to understand what is really going on when your objects are being initialized, list the members in the initialization list in the order in which those members are declared in the class.

Here, in the bad example, `m_data` is initialized first (as it appears in the class) *before* `m_size`, even though `m_size` is listed first. Thus, the `m_data` initialization will read uninitialized data from `m_size`.

Bad example:

```
class Array
{
public:
  Array(int lower, int upper);
private:
  int* m_data;
  unsigned m_size;
  int m_lowerBound;
  int m_upperBound;
};
Array::Array(int lower, int upper) :
  m_size(upper-lower+1),
  m_lowerBound(lower),
  m_upperBound(upper),
  m_data(new int[m_size])
{ ...
```

Correct example:

```
class Array
{
public:
  Array(int lower, int upper);
private:
  unsigned m_size;
  int m_lowerBound;
  int m_upperBound;
  int* m_data;
};
```

```
Array::Array(int lower, int upper) :
  m_size(upper-lower+1),
  m_lowerBound(lower),
  m_upperBound(upper),
  m_data(new int[m_size]) { ...
```

Virtual base classes are always initialized first, then base classes, data members, and finally the constructor body for the derived class is run.

```
class Derived : public Base   // Base is number 1
{
public:
  explicit Derived(int i);
  // The keyword explicit prevents the constructor
  // from being called implicitly.
  //   int x = 1;
  //   Derived dNew = x;
  // will not work

  Derived();

private:
  int m_jM;   // m_jM is number 2
  Base m_bM;  // m_bM is number 3
};

Derived::Derived(int i) : Base(i), m_jM(i), m_bM(i) {
// Recommended order       1        2         3
...
}
```

3.3.3 Copying of objects

- **A function must never return, or in any other way give access to, references or pointers to local variables outside the scope in which they are declared.** [no-refs-to-locals]

Returning a pointer or reference to a local variable is always wrong because it gives the user a pointer or reference to an object that no longer exists.

Bad example:

You are using a complex number class, `Complex`, and you write a method that looks like this:

```
Complex&
calculateC1 (const Complex& n1, const Complex& n2)
{
  double a = n1.getReal()-2*n2.getReal();
  double b = n1.getImaginary()*n2.getImaginary();

  // Create local object.
  Complex C1(a,b);

  // Return reference to local object.
  // The object is destroyed on exit from this
  // function: trouble ahead!
  return C1;
}
```

In fact, most compilers will spot this and issue a warning.

This particular function would be better written to return the result by value:

```
Complex calculateC1 (const Complex& n1,
                     const Complex& n2)
{
  double a = n1.getReal()-2*n2.getReal();
  double b = n1.getImaginary()*n2.getImaginary();

  return Complex(a,b);
}
```

- **If objects of a class should never be copied, then the copy constructor and the copy assignment operator should be deleted.** [copy-protection]

Ideally the question whether the class has a reasonable copy semantic will naturally be a result of the design process. Do not define a copy method for a class that should not have it.

By deleting the copy constructor and copy assignment operator, you can make a class non-copyable.

```
// There is only one ATLASExperimentalHall,
// and that should not be copied
class ATLASExperimentalHall
{
public:
  ATLASExperimentalHall();
  ~ATLASExperimentalHall();

  // Delete copy constructor to disallow copying.
  ATLASExperimentalHall(const ATLASExperimentalHall& )
     = delete;

  // Delete assignment operator to disallow assignment.
  ATLASExperimentalHall&
  operator=(const ATLAS_ExperimentalHall&) = delete;
};
```

In older versions of the language, this was achieved by declaring the deleted methods as `private` (and not implementing them). For new code, prefer explicitly deleting the functions.

```
// There is only one ATLASExperimentalHall,
// and that should not be copied
class ATLASExperimentalHall
{
public:
  ATLASExperimentalHall();
  ~ATLASExperimentalHall();

private:
  // Disallow copy constructor and assignment.
  ATLASExperimentalHall(const ATLASExperimentalHall&);
  ATLASExperimentalHall& operator=
```

```
    (const ATLAS_ExperimentalHall&);
};
```

- **If a class owns memory via a pointer data member, then the copy constructor, the assignment operator, and the destructor should all be implemented.** [define-copy-and-assignment]

  The compiler will generate a copy constructor, an assignment operator, and a destructor if these member functions have not been declared. A compiler-generated copy constructor does memberwise initialization and a compiler-generated copy assignment operator does memberwise assignment of data members and base classes. For classes that manage resources (examples: memory (new), files, sockets) the generated member functions probably have the wrong behavior and must be implemented by the developer. You have to decide if the resources pointed to must be copied as well (deep copy), and implement the correct behavior in the operators. Of course, the constructor and destructor must be implemented as well.

  Bad Example:

```
class String
{
public:
  String(const char *value=0);
  ~String(); // Destructor but no copy constructor
             // or assignment operator.
private:
  char *m_data;
};

String::String(const char *value)
{ // Correct behavior implemented in constructor.
  m_data = new char[strlen(value)]; // Fill m_data
}
String::~String()
{ // Correct behavior implemented in destructor
  delete m_data;
}

```

```
...


// Declare and construct a.  m_data points to "Hello"
String a("Hello");

// Open new scope
{ // Declare and construct b.
  // m_data points to "World"
  String b("World");

  b=a;
  // Execute default op= as synthesized by the compiler.
  // This is simply memberwise assignment.
  // For pointers like m_data, this is a bitwise copy
  // ==> m_data of "a" and "b" now point to the
  //     same string "Hello"
  // ==> 1) Memory b used to point to never deleted:
  //        a possible memory leak!
  //     2) When either a or b goes out of scope,
  //        its destructor  will delete the memory
  //        still pointed to by the other.
}

// Close scope: b's destructor called;
// memory still pointed to by `a' deleted!
String c=a;
// But m_data of a is undefined!!
```

- **Assignment member functions must work correctly when the left and right operands are the same object.** [self-assign]

This requires some care when writing assignment code, as this case (when left and right operands are the same) may require that most of the code is bypassed.

```
A& A::operator=(const A& a)
{
```

```
  if (this != &a) {
    // ... implementation of operator=
  }
}
```

3.4 Conversions

- **Use explicit rather than implicit type conversion.** [avoid-implicit-conversions]

  Most conversions are bad in some way. They can make the code less portable, less robust, and less readable. It is therefore important to use only explicit conversions. Implicit conversions are almost always bad.

- **Use the C++ cast operators (`dynamic_cast` and `static_cast`) instead of the C-style casts.** [use-c++-casts]

  In general, casts should be strongly discouraged, especially the old style C casts.

  The new cast operators give the user a way to distinguish between different types of casts, and to ensure that casts only do what is intended and nothing else.

  The C++ `static_cast` operator allows explicitly requesting allowed implicit conversions and between integers and enumerations. It also allows casting pointers up and down a class hierarchy (as long as there's no virtual inheritance), but no checking is done when casting from a less- to a more-derived type.

  The C++ `dynamic_cast` operator is used to perform safe casts down or across an inheritance hierarchy. One can actually determine whether the cast succeeded because failed casts are indicated either by a `bad_cast` exception or a null pointer. The use of this type of information at run time is called Run-Time Type Identification (RTTI).

```
int n = 3;
double r = static_cast<double>(n) * a;

class Base { };
```

```
class Derived : Base { };
void f(Derived* d_ptr)
{
  // if the following cast is inappropriate
  //  a null pointer will be returned!
  Base* b_ptr = dynamic_cast<Base*>(d_ptr);
  // ...
}
```

- **Do not convert `const` objects to non-`const`.** [no-const-cast]

  In general you should never cast away the constness of objects.

  If you have to use a `const_cast` to remove `const`, either you're writing some low-level code that that's deliberately subverting the C++ type system, or you have some problem in your design or implementation that the `const_cast` is papering over.

  Sometimes you're forced to use a `const_cast` due to problems with external libraries. But if the library in question is maintained by ATLAS, then try to get it fixed in the original library before resorting to `const_cast`.

  The keyword `mutable` allows data members of an object that have been declared const to remain modifiable, thus reducing the need to cast away constness. The `mutable` keyword should only be used for variables which are used for caching information. In other words, the object appears not to have changed but it has stored something to save time on subsequent use.

- **Do not use `reinterpret_cast`.** [no-reinterpret-cast]

  `reinterpret_cast` is machine-, compiler- and compile-options-dependent. It is a way of forcing a compiler to accept a type conversion which is dependent on implementation. It blows away type-safety, violates encapsulation and more importantly, can lead to unpredictable results.

  `reinterpret_cast` has legitimate uses, such as low-level code which deliberately goes around the C++ type system. Such code should usually be found only in the core and framework packages.

  Exception: `reinterpret_cast` is required in some cases if one is not using old-style casts. It is required for example if you wish to convert a callback function signature (X11, expat, Unix signal handlers are common causes). Some external libraries (X11 in particular) depend on casting function pointers. If you absolutely have to work around limitations in external libraries, you may of course use it.

One particularly nasty case to be aware of and to avoid is *pointer aliasing*. If two pointers have different types, the compiler may assume that they cannot point at the same object. For example, in this function:

```
int convertAndBuffer (int* buf, float x)
{
  float* fbuf = reinterpret_cast<float*>(buf);
  *fbuf = x;
  return *buf;
}
```

the compiler is entitled to rewrite it as

```
int convertAndBuffer (int* buf, float x)
{
  int ret = *buf;
  float* fbuf = reinterpret_cast<float*>(buf);
  *fbuf = x;
  return ret;
}
```

(As a special case, you can safely convert any pointer type to or from a `char*`.) The proper way to do such a conversion is with a `std::bit_cast`:

```
#include <bit>
int convertAndBuffer (int* buf, float x)
{
  *buf = std::bit_cast<int> (x);
  return *buf;
}
```

Prior to C++20, the recommended way to do this was with a union, but that should not be used for new code.

3.5 The class interface

3.5.1 Inline functions

- **Header files must contain no implementation except for small functions to be inlined. These inlined functions must appear at the end of the header after the class definition.** [inline-functions-impls]

  If you have many inline functions, it is usually better to split them out into a separate file, with extension "`.icc`", that is included at the end of the header.

  Inline functions can improve the performance of your program; but they also can increase the overall size of the program and thus, in some cases, have the opposite result. It can be hard to know exactly when inlining is appropriate. As a rule of thumb, inline only very simple functions to start with (one or two lines). You can use profiling information to identify other functions that would benefit from inlining.

  Use of inlining makes debugging hard and, even worse, can force a complete release rebuild or large scale recompilation if the inline definition needs to be changed.

3.5.2 Argument passing and return values

- **Pass an unmodifiable argument by value only if it is of built-in type or small; otherwise, pass the argument by `const` reference (or by `const` pointer if it may be null).** [large-argument-passing]

  An object is considered small if it is a built-in type or if it contains at most one small object. Built-in types such as `char`, `int`, and `double` can be passed by value because it is cheap to copy such variables. If an object is larger than the size of its reference (typically 64 bits), it is not efficient to pass it by value. Of course, a built-in type can be passed by reference when appropriate.

```
void func(char c);   // OK
void func(int i);    // OK
void func(double d); // OK
void func(complex<float> c); // OK

```

```
void func(Track t); // not good, since Track is large,
                    // so there is an overhead in
                    // copying t
```

Arguments of class type are often costly to copy, so it is preferable to pass a const reference to such objects; in this way the argument is not copied. Const access guarantees that the function will not change the argument.

In terms of efficiency, passing by pointer is the same as passing by reference. However, passing by reference is preferred, unless it is possible to the object to be missing from the call.

```
void func(const LongString& s);  // const reference
```

- **If an argument may be modified, pass it by non-`const` reference and clearly document the intent.** [modifiable-arguments]

  For example:

```
// Track @c t is updated by the addition of hit @c h.
void updateTrack(const Hit& h, Track& t);
```

  Again, passing by references is preferred, but a pointer may be used if the object can be null.

- **Use `unique_ptr` to pass ownership of an object to a function.** [pass-ownership]

  To pass ownership of an object into a function, use `unique_ptr` (by value):

```
void foo (std::unique_ptr<Object> obj);

...

  auto obj = std::make_unique<Object>();
  ...
  foo (std::move (obj));
```

  In most cases, `unique_ptr` should be passed by *value*. There are however a few possible use cases for passing `unique_ptr` by reference:

- The called function may replace the object passed in with another one. In this case, however, consider returning the new object as the value of the function.
- The called function may only conditionally take ownership of the passed object. This is likely to be confusing and error-prone and should probably be avoided. Consider if a `shared_ptr` would be better in this case.

There is basically no good case for passing `unique_ptr` as a const reference.

If you need to interoperate with existing code, object ownership may be passed by pointer. The fact that ownership is transferred should be clearly documented.

Do not pass ownership using references.

Here are a some additional examples to illustrate this. Assume that class `C` contains a member `Foo* m_owning_pointer` which the class deletes. (In modern C++, it would of course usually be better for this to be a `unique_ptr`.)

```
// --- Best
void C::takesOwnership (std::unique_ptr<Foo> foo)
{
   delete m_owning_pointer;
   m_owning_pointer = foo.release();
}

// --- OK if documented.
// Takes ownership of the @c foo pointer.
void C::takesOwnership (Foo* foo)
{
   delete m_owning_pointer;
   m_owning_pointer = foo;
}

// --- Don't do this!
void C::takesOwnership (Foo& foo)
```

```
{
  delete m_owning_pointer;
  m_owning_pointer = &foo;
}
```

- **Return basic types or new instances of a class type by value**. [return-by-value]

  Returning a class instance by value is generally preferred to passing an argument by non-const reference:

```
// Bad
void getVector (std::vector<int>& v)
{
  v.clear();
  for (int i=0; i < 10; i++) v.push_back(v);
}

// Better
std::vector<int> getVector()
{
  std::vector<int> v;
  for (int i=0; i < 10; i++) v.push_back(v);
  return v;
}
```

  The return-value optimization plus move semantics will generally mean that there won't be a significant efficiency difference between the two.

- **Use `unique_ptr` to return ownership.** [returning-ownership]

  If a function is returning a pointer to something that is allocated off the heap which the caller is responsible for deleting, then return a `unique_ptr`.

  If compatibility with existing code is an issue, then a plain pointer may be used, but the caller takes ownership should be clearly documented.

  Do not return ownership via a reference.

```
// Best
std::unique_ptr<Foo> makeFoo()
```

```
{
  return std::make_unique<Foo> (...);
}

// OK if documented
// makeFoo() returns a newly-allocated Foo;
// caller must delete it.
Foo* makeFoo()
{
  return new Foo (...);
}

// NO!
Foo& makeFoo()
{
  Foo* foo = new Foo (...);
  return *foo;
}
```

- **Have `operator=` return a reference to `*this`.** [assignment-return-value]

  This ensures that

  ```
  a = b = c;
  ```

  will assign `c` to `b` and then `b` to `a` as is the case with built-in objects.

- **Use `std::span` to represent and pass a bounded region of memory.** [span]

  In particular, use `std::span` instead of passing a pointer with a separate element count (or even worse, a pointer to an array with no bounds information).

  So you can use this:

  ```
  #include <span>
  int sum (const std::span<int>& s)
  {
  ```

```
  int ret = 0;
  for (int i : s) ret += i;
  return ret;
}
```

instead of

```
int sum (const int* p, size_t n)
{
  int ret = 0;
  for (size_t i = 0; i < n; i++) ret += p[i];
  return ret;
}
```

One might expect that `std::span` would include an `at()` method, to allow indexing with bounds checking, but that is only available in C++23. In the meantime, `CxxUtils::span` is very similar to `std::span` but does implement `at()`.

3.5.3 `const` correctness

- **Declare a pointer or reference argument, passed to a function, as `const` if the function does not change the object bound to it.** [const-arguments]

  An advantage of `const`-declared parameters is that the compiler will actually give you an error if you modify such a parameter by mistake, thus helping you to avoid bugs in the implementation.

  ```
  // operator<< does not modify the String parameter
  ostream& operator<<(ostream& out, const String& s);
  ```

- **The argument to a copy constructor and to an assignment operator must be a `const` reference.** [copy-ctor-arg]

  This ensures that the object being copied is not altered by the copy or assign.

- **In a class method, do not return pointers or non-`const` references to private data members.** [no-non-const-refs-returned]

Otherwise you break the principle of encapsulation.

If necessary, you can return a pointer to a `const` or `const` reference.

This does not mean that you cannot have methods returning an `iterator` if your class acts as a container.

An allowed exception to this rule if the use of the singleton pattern. In that case, be sure to add a clear explanation in a comment so that other developers will understand what you are doing.

- **Declare as `const` a member function that does not affect the state of the object.** [const-members]

  Declaring a member function as `const` has two important implications. First, only `const` member functions can be called for `const` objects; and second, a `const` member function will not change data members

  It is a common mistake to forget to `const` declare member functions that should be `const`.

  This rule does not apply to the case where a member function which does not affect the state of the object overrides a non-const member function inherited from some super class.

- **Do not let `const` member functions change the state of the program.** [really-const]

  A `const` member function promises not to change any of the data members of the object. Usually this is not enough. It should be possible to call a `const` member function any number of times without affecting the state of the complete program. It is therefore important that a `const` member function refrains from changing static data members or other objects to which the object has a pointer or reference.

3.5.4 Overloading and default arguments

- **Use function overloading only when methods differ in their argument list, but the task performed is the same.** [function-overloading]

  Using function name overloading for any other purpose than to group closely related member functions is very confusing and is not recommended.

3.5.5 Comparisons

- **Define comparisons for custom types using `operator==` and `operator<=>`.** [comparison-operators]

  Comparisons of for a custom class should be written using `operator==` (for equality/inequality) and `operator<=>` (for ordering). The compiler will supply the other comparison operators (`operator!=`, `operator<`, etc.) automatically. Where possible, `operator<=>` is best defined using the same operator on the members involved. Examples:

```
#include <compare>
#include <tuple>

class S
{
public:
  bool operator== (const S& other)
  {
    return m_key == other.m_key;
  }
  std::strong_ordering operator<=> (const S& other)
  {
    return m_key <=> other.m_key;
  }
private:
  int m_key;
};


class Version
{
public:
  bool operator== (const Version& other)
  {
    return m_major == other.m_major &&
           m_minor == other.m_minor;
  }

```

```
  std::strong_ordering
  operator<=> (const Version& other)
  {
    return
      std::make_tuple (m_major, m_minor) <=>
      std::make_tuple (other.m_major, other.m_minor);
  }
private:
  int m_major;
  int m_minor;
};
```

3.6 new and delete

- **Do not use new and delete where automatic allocation will work.** [auto-allocation-not-new-delete]

  Suppose you have a function that takes as an argument a pointer to an object, but the function does not take ownership of the object. Then suppose you need to create a temporary object to pass to this function. In this case, it's better to create an automatically-allocated object on the stack than it is to use new / delete. The former will be faster, and you won't have the chance to make a mistake by omitting the delete.

```
// Not good:
Foo* foo = new Foo;
doSomethingWithFoo (foo);
delete foo;

// Better:
Foo foo;
doSomethingWithFoo (&foo);
```

- **Match every invocation of new with one invocation of delete in all possible control flows from new.** [match-new-delete]

  A missing delete would cause a memory leak.

  However, in the Gaudi/Athena framework, an object created with new and registered in StoreGate is under control of StoreGate and must not be deleted.

In new code, you should generally use `make_unique` for this.

```
#include <memory>

  ...
  DataVector<C>* dv = ...;
  auto c = std::make_unique<C>("argument");
  ...
  if (test) {
    dv->push_back (std::move (c));
  }
```

`auto_ptr` was an attempt to do something similar to `unique_ptr` in older versions of the language. However, it has some serious deficiencies and should not be used in new code.

- **A function should explicitly document if it takes ownership of a pointer passed to it as an argument.** [explicit-ownership]

  The default expectation for a function should be that it does *not* take ownership of pointers passed to it as arguments. In that case, the function must *not* invoke `delete` on the pointer, nor pass it to any other function that takes ownership.

  However, if the function is clearly documented as taking ownership of the pointer, then it *must* either delete the pointer or pass it to another function which will ensure that it is eventually deleted.

  Rather than simply documenting that a function takes ownership of a pointer, it is recommended that you use `std::unique_ptr` to explicitly show the transfer of ownership.

```
void foo (std::unique_ptr<C> ptr);

  ...
  std::unique_ptr<C> p (new C);
  ...
  foo (std::move (p));
```

```
  // The argument of foo() is initialized by move.
  // p is left as a null pointer.
```

- **Do not access a pointer or reference to a deleted object.** [deleted-objects]

  A pointer that has been used as argument to a `delete` expression should not be used again unless you have given it a new value, because the language does not define what should happen if you access a deleted object. This includes trying to delete an already deleted object. You should assign the pointer to `nullptr` or a new valid object after the `delete` is called; otherwise you get a "dangling" pointer.

- **After deleting a pointer, assign it to `nullptr`.** [deleted-objects-2]

  C++ guarantees that deletion of null pointers is safe, so this gives some safety against double deletes.

```
X* myX = makeAnX();
delete myX;
myX = nullptr;
```

  This is of course not needed if the pointer is about to go out of scope, or when objects are deleted in a destructor (unless it's particularly complicated). But this is a good practice if the pointer persists beyond the block of code containing the `delete` (especially if it's a member variable).

3.7 Static and global objects

- **Do not declare variables in the global namespace.** [no-global-variables]

  If necessary, encapsulate those variables in a class or in a namespace. Global variables violate encapsulation and can cause global scope name clashes. Global variables make classes that use them context-dependent, hard to manage, and difficult to reuse.

  For variables that are used only within one "`.cxx`" file, put them in an anonymous namespace.

```
namespace {
  // This variable is visible only in the file
```

```
  // containing this declaration, and is guaranteed
  // not to conflict with any declarations from
  // other files.
  int counter;
}
```

- **Do not put functions into the global namespace.** [no-global-functions]

  Similarly to variables, functions declarations should be put in a namespace. If they are used only within one “.cxx” file, then they should be put in an anonymous namespace.

  In a few cases, it might be necessary to declare a function in the global namespace to have overloading work properly, but this should be an exception.

3.8 Object-oriented programming

- **Do not declare data members to be public.** [no-public-data-members]

  This ensures that data members are only accessed from within member functions. Hiding data makes it easier to change implementation and provides a uniform interface to the object.

```
class Point
{
public:
  Number x() const; // Return the x coordinate
private:
  Number m_x;
  // The x coordinate (safely hidden)
};
```

  The fact that the class `Point` has a data member `m_x` which holds the x coordinate is hidden.

  An exception is objects that are intended to be more like C-style structures than classes. Such classes should usually not have any methods, except possibly a constructor to make initialization easier.

- **If a class has at least one virtual method then it must have a public virtual destructor or (exceptionally) a protected destructor.** [virtual-destructor]

  The destructor of a base class is a member function that in most cases should be declared virtual. It is necessary to declare it virtual in a base class if derived class objects are deleted through a base class pointer. If the destructor is not declared virtual, only the base class destructor will be called when an object is deleted that way.

  There is one case where it is not appropriate to use a virtual destructor: a mix-in class. Such a class is used to define a small part of an interface, which is inherited (mixed in) by subclasses. In these cases the destructor, and hence the possibility of a user deleting a pointer to such a mix-in base class, should normally not be part of the interface offered by the base class. It is best in these cases to have a nonvirtual, nonpublic destructor because that will prevent a user of a pointer to such a base class from claiming ownership of the object and deciding to simply delete it. In such cases it is appropriate to make the destructor protected. This will stop users from accidentally deleting an object through a pointer to the mix-in base-class, so it is no longer necessary to require the destructor to be virtual.

- **Always re-declare virtual functions as virtual in derived classes.** [redeclare-virtual]

  This is just for clarity of code. The compiler will know it is virtual, but the human reader may not. This, of course, also includes the destructor, as stated in item [virtual-destructor, page 42]. Virtual functions in derived classes which override methods from the base class should also be declared with the `override` keyword. If the signature of the method is changed in the base class, so that the declaration in the derived class is no longer overriding it, this will cause the compiler to flag an error. (As an exception, `override` is not required for destructors. Since there is only one possible signature for a destructor, `override` doesn't add anything.)

```
class B
{
public:
  virtual void foo(int);
};
```

```

class D : public B
{
public:
  // Declare foo as a virtual method that overrides
  // a method from the base class.
  virtual void foo(int) override;
};
```

- **Avoid multiple inheritance, except for abstract interfaces.** [no-multiple-inheritance]

  Multiple inheritance is seldom necessary, and it is rather complex and error prone. The only valid exception is for inheriting interfaces or when the inherited behavior is completely decoupled from the class's responsibility.

  For a detailed example of a reasonable application of multiple inheritance, see [12], item 43.

- **Avoid the use of friend declarations.** [no-friend]

  Friend declarations are almost always symptoms of bad design and they break encapsulation. When you can avoid them, you should.

  Possible exceptions are the streaming operators and binary operators on classes. Other possible exceptions include very tightly coupled classes and unit tests.

- **Avoid the use of protected data members.** [no-protected-data]

  Protected data members are similar to friend declarations in that they allow a controlled violation of encapsulation. However, it is even less well-controlled in the case of protected data, since any class may derive from the base class and access the protected data.

  The use of protected data results in one class depending on the internals of another, which is a maintenance issue should the base class need to change. Like friend declarations, the use of protected member data should be avoided except for very closely coupled classes (that should generally be part of the same package). Rather, you should define a proper interface for what needs to be done (parts of which may be protected).

3.9 Notes on the use of library functions.

- **Use `std::abs` to calculate an absolute value.** [std-abs]

  The return type of `std::abs` will conform to the argument type; other variants of `abs` may not do this.

  In particular, beware of this:

```
#include <cstdlib>
float foo (float x)
{
  return abs(x);
}
```

  which will truncate `x` to an integer. (Clang will warn about this.)

  Conversely, in this example:

```
#include <cmath>
int (int x)
{
  return fabs(x);
}
```

  the argument will first be converted to a float, then the result converted back to an integer.

  Using `std::abs` uniformly should do the right thing in almost all cases and avoid such surprises.

- **Use C++20 ranges with caution.** [ranges]

  C++20 adds *ranges*, an abstraction an abstraction of something that can be iterated over. Essentially, a range is something that can return `begin()` and `end()` iterators. The `ranges` library allows composing and transforming ranges. For example:

```
#include <ranges>
...
auto even = [](int i) { return (i%2) == 0; };
auto sq = [](int i) { return i*i; };
```

```cpp
using namespace std::views;
auto r = iota(0, 6) | filter(even) | transform(sq);
for (int i : r) std::cout << i << " ";
```

Ranges can be very useful. However, they need to be used with caution.

– Do not reimplement missing functionality yourself.

  Much of that C++20 ranges library originated from an external library, range-v3 [14]. However, many useful operations from that library did not make it into the C++20 standard (some are added in later versions of the standard). For example, in C++20 ranges, there is no straightforward way to initialize a `std::vector` from a range. If such additional functionality is needed, it should be added centrally in `CxxUtils` rather than being reimplemented where it is needed. In that way, it can be shared with other parts of Athena. This also makes it easier to replace any such reimplemented functionality with versions from the standard library when they become available.

– Functions used to define ranges should not have side effects.

  One can define a range in terms of functions that filter and transform the range, as in the example above. However, it may be difficult to predict under exactly what circumstances these functions may be called, as this depends on the implementation of the range components. Therefore, functions used with ranges should not have side-effects (and should generally execute quickly).

– Beware of dangling ranges.

  Ranges are often references to other objects. Like any references, they must not outlive the object that they reference.

```cpp
auto squares()
{
  auto sq = [](int i) { return i*i; };
  std::vector<int> v {1, 2, 3, 4};
  return v | std::views::transform(sq);
   // BAD: returns a range with a dangling
   //      reference to a deleted vector.
```

```
}
```

- Do not modify containers referenced by ranges.

  Similarly, do not modify a container referenced by a range. Some of the range components may cache results internally; changing the underlying container may cause these to return incorrect results.

```
std::vector<int> v {1, 2, 3, 4};
auto sq = [](int i) { return i*i; };
auto r = v | std::views::transform(sq);
v.insert (v.begin(), 5); // BAD: may invalidate
                         // the range r.
```

In general, C++20 view objects should be used directly after they are defined, and not saved in, say, member variables.

3.10 Thread friendliness and thread safety

Code that is to be run in AthenaMT as part of an `AthAlgorithm` must be "thread-friendly." While the framework will ensure that no more than one thread is executing a given `AthAlgorithm` instance at one time, the code must ensure that it doesn't interfere with *other* threads. Some guidelines for this are outlined below; but in brief: don't use static data, don't use `mutable`, and don't cast away `const`. Following these rules will keep you out of most potential trouble.

Code that runs as part of an `AthService`, an `AthReentrantAlgorithm`, a data object implementation, or other common code may need to be fully "thread-safe;" that is, allow for multiple threads to operate simultaneously on the *same* object. The easiest way to ensure this is for the object to have no mutable internal state, and only `const` methods. If, however, some threads may be modifying the state of the object, then some sort of locking or other means of synchronization will likely be required. A full discussion of this is beyond the scope of these guidelines.

To run successfully in a multithreaded environment, algorithmic code must also respect the rules imposed by the framework on event and conditions data access. This is also beyond the scope of these guidelines.

- **Follow C++ thread-safety conventions for data objects.** [mt-follow-c++-conventions]

The standard C++ container objects follow the rule that methods declared as `const` are safe to call simultaneously from multiple threads, while no non-const method can be called simultaneously with any other method (`const` or non-const) on the same object.

Classes meant to be data objects should generally follow the same rules, unless there is a good reason to the contrary. This will generally happen automatically if the rules outlined below are followed: briefly, don't use static data, don't use `mutable`, and don't cast away `const`.

Sometimes it may be useful to have data classes for which non-const methods may be called safely from multiple threads. If so, this should be indicated in the documentation of the class, and perhaps hinted from its name (maybe like `ConcurrentFoo`).

- **Do not use non-`const` static variables** [mt-no-nonconst-static]

  Do not use non-const static variables in thread-friendly code, either global or local.

```
int a;
int foo() {
  if (a > 0) {  // Bad use of global static.
    static int count = 0;
    return ++count;    // Bad use of local static.
  }
  return 0;
}

struct Bar
{
  static int s_x;
  int x() { return s_x; } // Bad use of static
                          // class member.
};
```

  A `const static` is, however, perfectly fine:

```
static const std::string s = "a string";  // OK, const
```

It's generally OK to have `static` mutex or thread-local variables:

```
static std::mutex m;  // OK. It's a mutex,
                      // so it's meant to be accessed
                      // from multiple threads.
static thread_local int a; // OK, it's thread-local.
```

(Be aware, though, that thread-local variables can be quite slow.) A static `std::atomic<T>` variable may be OK, but only if it doesn't need to be updated consistently with other variables.

- **Do not cast away `const`** [mt-no-const-cast]

  This rule was already mentioned above. However, it deserves particular emphasis in the context of thread safety. The usual convention for C++ is that a `const` method is safe to call simultaneously from multiple threads, while if you call a non-const method, no other threads can be simultaneously accessing the same object. If you cast away `const`, you are subverting these guarantees. Any use of `const_cast` needs to be analyzed for its effects on thread-safety and possibly protected with locking.

  For example, consider this function:

```
void foo (const std::vector<int>& v)
{
  ...
  // Sneak this in.
  const_cast<std::vector<int>&>(v).push_back(10);
}
```

  Someone looking at the signature of this function would see that it takes only a `const` argument, and therefore conclude that that it is safe to call this simultaneously with other code that is also reading the same vector instance. But it is not, and the `const_cast` is what causes that reasoning to fail.

- **Avoid mutable members.** [mt-no-mutable]

  The use of `mutable` members has many of the same problems as `const_cast` (as indeed, `mutable` is really just a restricted version of `const_cast`). A `mutable` member can generally not be changed from a non-const method without some sort of explicit locking or other synchronization. It is best avoided in code that should be used with threading.

`mutable` can, however, be used with objects that are explicitly intended to be accessed from multiple threads. These include mutexes and thread-local pointers. In some cases, members of `atomic` type may also be safely made `mutable`, but only if they do not need to be updated consistently with other members.

- **Do not return non-`const` member pointers/references from `const` methods** [mt-const-consistency]

  Consider the following fragment:

```
class C
{
public:
  Impl* impl() const { return m_impl; }
private:
  Impl* m_impl;
};
```

  This is perfectly valid according to the C++ `const` rules. However, it allows modifying the `Impl` object following a call to the `const` method `impl()`. Whether this is actually a problem depends on the context. If `m_impl` is pointing at some unrelated object, then this might be OK; however, if it is pointing at something which should be considered part of `C`, then this could be a way around the `const` guarantees.

  To maintain safety, and to make the code easier to reason about, do not return a non-const pointer (or reference) member from a `const` member function.

- **Be careful returning `const` references to class members.** [mt-const-references]

  Consider the following example:

```
class C
{
```

```
public:
  const std::vector<int>& v() const { return m_v; }
  void append (int x) { m_v.push_back (x); }
private:
  std::vector<int> m_v;
};

int getSize (const C& c)
{
  return c.v().size();
}

int push (C& c)
{
  c.append (1);
}
```

This is a fairly typical example of a class that has a large object as a member, with an accessor the returns the member by const reference to avoid having to do a copy.

But suppose now that one thread calls `getSize()` while another thread calls `push()` at the same time on the same object. It can happen that first `getSize()` gets the reference and starts the call to `size()`. At that point, the `push_back()` can run in the other thread. If `push_back()` runs at the same time as `size()`, then the results are unpredictable — the `size()` call could very well return garbage.

Note that it doesn't help to add locking within the class `C`:

```
class C
{
public:
  const std::vector<int>& v() const
  {
    std::lock_guard<std::mutex> lock (m_mutex);
    return m_v;
  }
  void append (int x)
```

```
  {
    std::lock_guard<std::mutex> lock (m_mutex);
    m_v.push_back (x);
  }
private:
  mutable std::mutex m_mutex;
  std::vector<int> m_v;
};
```

This is because the lock is released once `v()` returns — and at that point, the caller can call (`const`) methods on the `vector` instance unprotected by the lock.

Here are a few ways in which this could possibly be solved. Which is preferable would depend on the full context in which the class is used.

- Change the `v()` accessor to return the member by value instead of by reference.
- Remove the `v()` accessor and instead add the needed operations to the `C` class, with appropriate locking. For the above example, we could add something like:

```
size_t C::vSize() const
{
  std::lock_guard<std::mutex> lock (m_mutex);
  return m_v.size();
}
```

- Change the type of the `m_v` member to something that is inherently thread-safe. This could mean replacing it with a wrapper around `std::vector` that does locking internally, or using something like `concurrent_vector` from TBB.
- Do locking externally to class `C`. For example, introduce a mutex that must be acquired in both `getSize()` and `push()` in the above example.

3.11 Formatted output

- **Prefer `std::format` to `printf` or `iostream` formatting.** [use-format]

  For new code, use the C++20 formatting library to format values to a string rather than using `printf`-style formatting or using `iostream` manipulators.

  Example:

```
#include <format>
...
const char* typ = "ele";
float energy = 14.2345;
int mask = 323;

std::cout << std::format
  ("A {1:.2f} GeV {0} mask {2:#06x}.\n",
   typ, energy, mask);
// prints: A 14.23 GeV ele mask 0x0143.
```

  Compare using `printf`-style formatting:

```
#include "CxxUtils/StrFormat.h"
...
std::cout << CxxUtils::strformat
  ("A %.2f GeV %s mask %#06x.\n",
   energy, typ, mask);
```

  or `iostream`:

```
#include <iomanip>
...
const int default_precision = std::cout.precision();
const std::ios_base::fmtflags default_flags =
  std::cout.flags();
const char default_fill = std::cout.fill();
std::cout << "A " << std::fixed << std::setprecision(2)
          << energy << std::defaultfloat
```

```
            << std::setprecision(default_precision)
            << " GeV " << typ << " mask "
            << std::hex << "0x" << std::setfill('0')
            << std::setw(4) << mask
            << std::setfill(default_fill)
            << ".\n";
std::cout.flags(default_flags);
```

Like the streaming operator, `std::format` has a way of customizing how a given type is formatted. However, it is somewhat more involved than for `operator<<`; in addition, `std::format` will not use existing custom streaming operators. Therefore, for generating printable representations of class instances, it is probably better in most cases to use the `iostream` mechanism.

3.12 Assertions and error conditions

- **Pre-conditions and post-conditions should be checked for validity.** [pre-post-conditions]

  You should validate your input and output data whenever an invalid input can cause an invalid output.

- **Don't use assertions in place of exceptions.** [assertion-usage]

  Assertions should only be used to check for conditions which should be logically impossible to occur. Do not use them to check for validity of input data. For such cases, you should raise an exception (or return a Gaudi error code) instead.

  Assertions may be removed from production code, so they should not be used for any checks which must always be done.

3.13 Error handling

- **Use the standard error printing facility for informational messages. Do not use `cerr` and `cout`.** [no-cerr-cout]

  The "standard error printing facility" in Athena/Gaudi is `MsgStream`. No production code should use `cout`. Classes which are not Athena-aware could use `cerr` before throwing an exception, but all Athena-aware classes should use `MSG::FATAL` and/or throw an exception. In addition, it is acceptable to use writes to `cout` in unit tests.

When using `MsgStream`, note that a call to, e.g., `msg() << MSG::VERBOSE` that is suppressed by the output level has a higher runtime cost than a call suppressed by `if (msgLvl <= MSG::VERBOSE)`. The `ATH_MSG` macros (`ATH_MSG_INFO` and `ATH_MSG_DEBUG` etc) wrap `msg()` calls in appropriate if statements and are preferred in general for two reasons: they take up less space in the source code and indicate immediately that the message is correctly handled.

- **Check for all errors reported from functions.** [check-return-status]

  It is important to always check error conditions, regardless of how they are reported.

- **Use exceptions to report fatal errors from non-Gaudi components.** [exceptions]

  Exceptions in C++ are a means of separating error reporting from error handling. They should be used for reporting errors that the calling code should not be expected to handle. An exception is “thrown” to an error handler, so the treatment becomes non-local.

  If you are writing a Gaudi component, or something that returns a Gaudi `StatusCode`, then you should usually report an error by posting a message to the message service and returning a status code of `ERROR`.

  However, if you are writing a non-Gaudi component and you need to report an error that should stop event processing, you should raise an exception.

  If your code is throwing exceptions, it is helpful to define a separate class for each exception that you throw. That way, it is easy to stop in the debugger when a particular exception is thrown by putting a breakpoint in the constructor for that class.

```
#include <stdexcept>

class ExcMyException
  : public std::runtime_error
{
```

```
public:
  // Constructor can take arguments to pass through
  // additional information.
  ExcMyException (const std::string& what)
    : std::runtime_error ("My exception: " : what)
  {}
};

...

  throw MyException ("You screwed up.");
```

- **Do not throw exceptions as a way of reporting uncommon values from a function.** [exception-usage]

  If an error *can* be handled locally, then it should be. Exceptions should not be used to signal events which can be expected to occur in a regular program execution. It is up to programmers to decide what it means to be exceptional in each context.

  Take for example the case of a function `find()`. It is quite common that the object looked for is not found, and it is certainly not a failure; it is therefore not reasonable in this case to throw an exception. It is clearer if you return a well-defined value.

- **Do not use exception specifications.** [no-exception-specifications]

  Exception specifications were a way to declare that a function could throw one of only a restricted set of exceptions. Or rather, that's what most people wanted it to do; what it actually did was require the compiler to check, at runtime, that a function did not throw any but a restricted set of exceptions.

  Experience has shown that exception specifications are generally not useful and non-empty exception specifications are now an error [15]. They should not be used in new code, and are not allowed in C++20.

  There is also the keyword `noexcept`. The motivation for this was really to address a specific problem with move constructors and exception-safety, and it is not clear that it is generally useful [16]. For now, it is not recommended to use `noexcept`, unless you have a specific situation where you know it would help.

- **Do not catch a broad range of exceptions outside of framework code.** [no-broad-exception-catch]

  The C++ exception mechanism allows catching a thrown exception, giving the program the chance to continue execution from the point where the exception was caught. This can be used some specific cases where you know that some specific exception isn't really a problem. However, you should catch only the particular exception involved here. If you use an overly-broad catch specification, you risk hiding other problems. Example:

```
try {
  return getObject ("foo");
  // getObject may throw ExcNotFound if the "foo"
  // object is not found.  In that case we can just
  // return 0.
}
catch (ExcNotFound&) {
  return 0;
}

// But one would not want to do this, since that would
// hide other errors:
catch (...) {
  return 0;
}
```

- **Prefer to catch exceptions as `const` reference, rather than as value.** [catch-const-reference]

  Classes used for exceptions can be polymorphic just like data classes, and this is in fact the case for the standard C++ exceptions. However, if you catch an exception and name the base class by value, then the object thrown is copied to an instance of the base class.

  For example, consider this program:

```
#include <stdexcept>
#include <iostream>
```

```

class myex : public std::exception {
public:
  virtual const char* what() const noexcept
  { return "Mine!"; }
};

void foo()
{
  throw myex();
}

int main()
{
  try {
    foo();
  }
  catch (std::exception ex) {
    std::cout << "Exception: " << ex.what() << "\n";
  }
  return 0;
}
```

It looks like the intention here is to have a custom message printed when the exception is caught. But that's not what happens — this program actually prints:

```
Exception: std::exception
```

That's because in the `catch` clause, the `myex` instance is copied to a `std::exception` instance, so any information about the derived `myex` class is lost. If we change the catch to use a reference instead:

```
catch (const std::exception ex&) {
```

then the program prints what was probably intended.

```
Exception: Mine!
```

Recent versions of gcc will warn about this.

3.14 Parts of C++ to avoid

Here a set of different items are collected. They highlight parts of the language which should be avoided, either because there are better ways to achieve the desired results or because the language features are still immature. In particular, programmers should avoid using the old standard C functions, where C++ has introduced new and safer possibilities.

- **Do not use C++ modules.** [no-modules]

  Modules were introduced in C++20 as a better alternative to `#include`. If a module is referenced via `import`, it avoids repeatedly parsing the code as well as avoiding issues that arise due to interference between headers. However, building modules requires significant support from the build system, and the support in compilers and associated tools is still very immature. Even using the standard library as a module is not fully functional with C++20.

  For now, avoid any use of modules. With C++23, it may be possible to use standard libraries as modules, but building ATLAS code as modules will require significant additional development.

- **Do not use C++ coroutines.** [no-coroutines]

  Coroutines allow for a non-linear style of control flow, where one can return from the middle of a function and then resume execution from that point at a later time. However, the coroutine interfaces available in C++20 are quite low-level: they are intended to be used as building blocks for other library components rather than for direct use by user code. Further, uncontrolled use of the type of control flow made possible by coroutines has the potential to be terribly confusing.

  For now, avoid use of coroutines. If you have a use case that would greatly benefit from using coroutines, please consult with software coordination. This recommendation will be revisited for new versions of C++ which may include easier mechanisms for using coroutines.

- **Do not use `malloc`, `calloc`, `realloc`, and `free`. Use `new` and `delete` instead.** [no-malloc]

You should avoid all memory-handling functions from the standard C-library (`malloc`, `calloc`, `realloc`, and `free`) because they do not call constructors for new objects or destructors for deleted objects.

Exceptions may include aligned memory allocations, but this should generally not be done outside of low-level code in core packages.

- **Do not use functions defined in `stdio`. Use the `iostream` functions in their place.** [no-stdio]

  `scanf` and `printf` are not type-safe and they are not extensible. Use `operator>>` and `operator<<` associated with C++ streams instead, along with `std::format` to handle formatting (see use-format, page 52). `iostream` and stdio functions should never be mixed.

  Example:

```
// type safety
char* aString("Hello Atlas");
printf("This works: %s \n", aString);
cout <<"This also works:"<<aString<<endl;
char aChar('!');
printf("This does not %s \n", aChar);
   // and you get a core dump
cout <<"But this is still OK :"<<aChar<<endl;

//extensibility
std::string aCPPString("Hello Atlas");
printf("This does not work: %s \n", aCPPString);
   //Core dump again
```

  It is of course acceptable to use stdio functions if you're calling an external library that requires them.

  If you need to use `printf` style formatting, see "`CxxUtils/StrFormat.h`." However, `std::format` is preferred for new code.

- **Do not use the ellipsis notation for function arguments.** [no-ellipsis]

  Prior to C++ 11, functions with an unspecified number of arguments had to be declared and used in a type-unsafe manner:

```
// avoid to define functions like:
void error(int severity, ...) // "severity" followed
                              // by a zero-terminated
                              // list of char*s
```

This method should be avoided.

As of C++11, one can accomplish something similar using variadic templates:

```
template<typename ...ARGS>
void error(int severity, ARGS...)
```

This is fine, but should be used judiciously. It's appropriate for forwarding arguments through a template function. For other cases, it's worth thinking if there might be a simpler way of doing things.

An ellipsis can also occur in a catch clause to catch any exception: `catch(...)`. This is acceptable, but should generally be restricted to framework-like code.

- **Do not use preprocessor macros to take the place of functions, or for defining constants.** [no-macro-functions]

Use templates or inline functions rather than the pre-processor macros.

```
// NOT recommended to have function-like macro
#define SQUARE(x) x*x

// Better to define an inline function:
inline int square(int x) {
  return x*x;
};
```

- **Do not declare related numerical values as `const`. Use `enum` declarations.** [use-enum]

The `enum` construct allows a new type to be defined and hides the numerical values of the enumeration constants.

```
enum State {halted, starting, running, paused};
```

- **Do not use `NULL` to indicate a null pointer; use the `nullptr` keyword instead.** [nullptr]

  Older code often used the constant `0`. `NULL` is appropriate for C, but not C++.

- **Do not use `const char*` or built-in arrays "`[]`"; use `std::string` instead.** [use-std-string]

  One thing to be aware of, though. C++ will implicitly convert a `const char*` to a `std::string`; however, this may add significant overhead if used in a loop. For example:

```
void do_something (const std::string& s);
...
for (int i=0; i < lots; i++) {
  ...
  do_something ("hi there!");
```

  Each time through the loop, this will make a new `std::string` copy of the literal. Better to move the conversion to `std::string` outside of the loop:

```
std::string myarg = "hi there!";
for (int i=0; i < lots; i++) {
  ...
  do_something (myarg);
```

- **Avoid using union types.** [avoid-union-types]

  Unions can be an indication of a non-object-oriented design that is hard to extend. The usual alternative to unions is inheritance and dynamic binding. The advantage of having a derived class representing each type of value stored is that the set of derived class can be extended without rewriting any code. Because code with unions is only slightly more efficient, but much more difficult to maintain, you should avoid it.

  Unions may be used in some low-level code and in places where efficiency is particularly important. Unions may also be used in low-level code to avoid pointer aliasing (see no-reinterpret-cast, page 28).

- **Avoid using bit fields.** [avoid-bitfields]

  Bit fields are a feature that C++ inherited from C that allow one to specify that a member variable should occupy only a specified number of bits, and that it can be packed together with other such members.

```
class C
{
public:
  unsigned int a : 2;   // Allocated two bits
  unsigned int b : 3;   // Allocated three bits
};
```

  It may be tempting to use bit fields to save space in data written to disk, or in packing and unpacking raw data. However, this usage is not portable. The C++ standard has this to say:

  > Allocation of bit-fields within a class object is implementation-defined. Alignment of bit-fields is implementation-defined. Bit-fields are packed into some addressable allocation unit. [Note: Bit-fields straddle allocation units on some machines and not on others. Bit-fields are assigned right-to-left on some machines, left-to-right on others. – end note]

  Besides portability issues, there are other other potential issues with bit fields that could be confusing: bit fields look like class members but obey subtly different rules. For example, one cannot form a reference to a bit field or take its address. There is also an issue of data races when writing multithreaded code. It is safe to access two ordinary class members simultaneously from different threads, but not two adjacent bit fields. (Though it is safe to access simultaneously two bit field members separated by an ordinary member. This leads to be possibility that thread-safety of bit field access could be compromised by the removal of an unrelated member.) Access to bit fields also incurs a CPU penalty.

  In light of this, it is best to avoid bit fields in most cases. Exceptions would be cases where saving memory is very important and the internal structure of the class is not exposed.

For some cases, `std::bitset` can be a useful, portable replacement for bit fields.

- **Do not use `asm` (the assembler macro facility of C++).** [no-asm]

  Many special-purpose instructions are available through the use of compiler intrinsic functions. For those rare use cases where an `asm` might be needed, the use of the `asm` should be encapsulated and made available in a low-level package (such as `CxxUtils`).

- **Do not use the keyword `struct` for types used as classes.** [no-struct]

  The `class` keyword is identical to `struct` except that by default its contents are private rather than public. `struct` may be allowed for writing non-object-oriented PODs (plain old data, i.e. C structs) on purpose. It is a good indication that the code is on purpose not object-oriented.

- **Do not use static objects at file scope. Use an anonymous namespace instead.** [anonymous-not-static]

  The use of `static` to signify that something is private to a source file is obsolete; further it cannot be used for types. Use an anonymous namespace instead.

  For entities which are not public but are also not really part of a class, prefer putting them in an anonymous namespace to putting them in a class. That way, they won't clutter up the header file.

- **Do not declare your own alias for booleans. Use the `bool` type of C++ for booleans.** [use-bool]

  The `bool` type was not implemented in C. Programmers usually got around the problem by typedefs and/or const declarations. This is no longer needed, and must not be used in ATLAS code.

- **Avoid pointer arithmetic.** [no-pointer-arithmetic]

  Pointer arithmetic reduces readability, and is extremely error prone. It should be avoid outside of low-level code.

- **Do not declare variables with `register`.** [no-register]

The `register` keyword was originally intended as a hint to the compiler that a variable will be used frequently, and therefore it would be good to assign a dedicated register to that variable. However, compilers have long been able to do a good job of assigning values to registers; this is anyway highly-machine dependent.

Use of the `register` keyword now an error.

3.15 Readability and maintainability

- **Code should compile with no warnings.** [no-warnings]

  Many compiler warnings can indicate potentially serious problems with your code. But even if a particular warning is benign, it should be fixed, if only to prevent other people from having to spend time examining it in the future.

  Warnings coming from external libraries should be reported to whomever is maintaining the ATLAS wrapper package for the library. Even if the library itself can't reasonably be fixed, it may be possible to put a workaround in the wrapper package to suppress the warning.

  See [17] for help on how to get rid of many common types of warning. If it is really impossible to get rid of a warning, that fact should be documented in the code.

- **Keep functions short.** [short-functions]

  Short functions are easier to read and reason about. Ideally, a single function should not be bigger than can fit on one screen (i.e., not more than 30–40 lines).

- **Avoid excessive nesting of indentation.** [excessive-nesting]

  It becomes difficult to follow the control flow in a function when it becomes deeply nested. If you have more than 4–5 indentation levels, consider splitting off some of the inner code into a separate function.

- **Avoid duplicated code.** [avoid-duplicate]

  This statement has a twofold meaning.

The first and most evident is that one must avoid simply cutting and pasting pieces of code. When similar functionalities are necessary in different places, they should be collected in methods, and reused.

The second meaning is at the design level, and is the concept of code reuse.

Reuse of code has the benefit of making a program easier to understand and to maintain. An additional benefit is better quality because code that is reused gets tested much better.

Code reuse, however, is not the end-all goal, and in particular, it is less important than encapsulation. One should not use inheritance to reuse a bit of code from another class.

- **Document in the code any cases where clarity has been sacrificed for performance.** [document-changes-for-performance]

  Optimize code only when you know you have a performance problem. This means that during the implementation phase you should write code that is easy to read, understand, and maintain. Do not write cryptic code, just to improve its performance.

  Very often bad performance is due to bad design. Unnecessary copying of objects, creation of large numbers of temporary objects, improper inheritance, and a poor choice of algorithms, for example, can be rather costly and are best addressed at the architecture and design level.

- **Avoid creating type aliases for classes.** [avoid-typedef]

  Type aliases (typedefs) are a serious impediment in large systems. While they simplify code for the original author, a system filled with aliases can be difficult to understand. If the reader encounters a class `A`, he or she can find an `#include` with "`A.h`" in it to locate a description of `A`; but aliases carry no context that tell a reader where to find a definition. Moreover, most of the generic characteristics obtained with aliases are better handled by object oriented techniques, like polymorphism.

  Aliases are acceptable where they provide part of the expected interface for a class, for example `value_type`, etc. in classes used with STL algorithms. They are often indispensable in template programming and metaprogramming, and are also part of how xAOD classes and POOL converters are typically defined.

In other contexts, they should be used with care, and should generally be accompanied with a comment giving the rationale for the alias.

Aliases may be used as a "customization point;" that is, to allow the possibility of changing a type in the future. For example, the auxiliary store code uses integers to identify particular auxiliary data items. But rather than declaring these as an integer type directly, an alias `auxid_t` is used. This allows for the possibility of changing the type in the future without having to make changes throughout the code base. It also makes explicit that variables of that type are meant to identify auxiliary data items, rather than being random integers.

An alias may also be used inside a function body to shorten a cumbersome type name; however, this should be used sparingly.

- **Code should use the standard ATLAS units for time, distance, energy, etc.** [atlas-units]

As a reminder, energies are represented as MeV and lengths as mm. Please use the symbols defined in `GaudiKernel/SystemOfUnits.h`.

```
#include "GaudiKernel/SystemOfUnits.h"

float pt_thresh = 20 * Gaudi::Units::GeV;
float ip_cut = 0.1 * Gaudi::Units::cm;
```

3.16 Portability

- **All code must comply with the 2020 version of the ISO C++ standard (C++20).** [standard-cxx]

A draft of the standard which is essentially identical to the final version may be found at [4]. However, the standards documents are not very readable. A better reference for most questions about what is in the standard is the cppreference.com website [5].

At some point, compatibility with C++23 will also be required.

- **Make non-portable code easy to find and replace.** [limit-non-portable-code]

Non-portable code should preferably be factored out into a low-level package in Control, such as `CxxUtils`. If that is not possible, an `#ifdef` may be used.

However, `#ifdefs` can make a program completely unreadable. In addition, if the problems being solved by the `#ifdef` are not solved centrally by the release tool, then you resolve the problem over and over. Therefore. the using of `#ifdef` should be limited.

- **Headers supplied by the implementation (system or standard libraries header files) must go in <> brackets; all other headers must go in "" quotes.** [system-headers]

```
// Include only standard header with <>
#include <iostream>  // OK: standard header
#include <MyFyle.hh> // NO: nonstandard header

// Include any header with ""
#include "stdlib.h"           // NO: better to use <>
#include "MyPackage/MyFyle.h" // OK
```

- **Do not specify absolute directory names in include directives. Instead, specify only the terminal package name and the file name.** [include-path]

  Absolute paths are specific to a particular machine and will likely fail elsewhere.

  The ATLAS convention is to include the package name followed by the file name. Watch out: listing the package name twice is wrong, but some build systems don't catch it.

```
#include "/atlas/sw/dist/1.2/Foo/Bar/Qux.h"
  // Wrong
#include "Foo/Bar/Qux.h"  // Wrong
#include "Bar/Bar/Qux.h"  // Wrong
#include "Bar/Qux.h"      // Right
```

- **Always treat include file names as case-sensitive.** [include-case-sensitive]

Some operating systems, e.g. Windows NT, do not have case-sensitive file names. You should always include a file as if it were case-sensitive. Otherwise your code could be difficult to port to an environment with case-sensitive file names.

```
// Includes the same file on Windows NT,
// but not on UNIX
#include <Iostream>   //not correct
#include <iostream>   //OK
```

- **Do not make assumptions about the size or layout in memory of an object.** [no-memory-layout-assumptions]

The sizes of built-in types are different in different environment. For example, an int may be 16, 32, or even 64 bits long. The layout of objects is also different in different environments, so it is unwise to make any kind of assumption about the layout in memory of objects.

If you need integers of a specific size, you can use the definitions from <cstdint>:

```
#include <cstdint>

int16_t  a;       // A 16-bit signed int
uint8_t  b;       // A 8-bit unsigned int
int_fast_16_t c;  // Fastest available signed int type
                  // at least 16 bits wide.
```

The C++ standard requires that class members declared with no intervening access control keywords (public, protected, private) be laid out in memory in the order in which they are declared in the class. However, if there is an access control keyword between two member declarations, their relative ordering in memory is unspecified. In any case, the compiler is free to insert arbitrary padding between members.

- **Take machine precision into account in your conditional statements. Do not compare floats or doubles for equality.** [float-precision]

Have a look at the std::numeric_limits<T> class, and make sure your code is not platform-dependent. In particular, take care when testing floating point values for equality. For example, it is better to use:

```
const double tolerance = 0.001;

...

#include <cmath>

if (std::abs(value1 - value2) < tolerance ) ...
```

than

```
if ( value1 == value2 ) ...
```

Also be aware that on 32-bit platforms, the result of inequality operations can change depending on compiler optimizations if the two values are very close. This can lead to problems if an STL sorting operation is based on this. A fix is to use the operations defined in `CxxUtils/fpcompare.h`.

- **Do not depend on the order of evaluation of arguments to a function; in particular, never use the increment and decrement operators in function call arguments.** [order-of-evaluation]

  The order of evaluation of function arguments is not specified by the C++ standard, so the result of an expression like `foo(a++, vec(a))` is platform-dependent.

  ```
  func(f1(), f2(), f3());
  // f1 may be evaluated before f2 and f3,
  // but don't depend on it!
  ```

  Beware in particular if you're using random numbers. The result of something like

  ```
  atan2 (static_cast<double>(rand()),
         static_cast<double>(rand()));
  ```

  can change depending on how it's compiled.

- **Do not use system calls if there is another possibility (e.g. the C++ run time library).** [avoid-system-calls]

For example, do not forget about non-Unix platforms.

- **Prefer `int` / `unsigned int` and `double` types.** [preferred-types]

  The default type used for an integer value should be either `int` or `unsigned int`. Use other integer types (`short`, `long`, etc.) only if they are actually needed.

  For floating-point values, prefer using `double`, unless there is a need to save space and the additional precision of a `double` vs. `float` is not important.

- **Do not call any code that is not in the release or is not in the list of allowed external software.** [no-new-externals]

4 Style

This section concerns the style, as opposed to the functionality, of the code.

4.1 General aspects of style

- **The `public`, `protected`, and `private` sections of a class must be declared in that order. Within each section, nested types (e.g. `enum` or `class`) must appear at the top.** [class-section-ordering]

  The public part should be most interesting to the user of the class, and should therefore come first. The private part should be of no interest to the user and should therefore be listed last in the class declaration.

```
class Path
{
public:
  Path();
  ~Path();

protected:
  void draw();

private:
  class Internal {
    // Path::Internal declarations go here ...
```

```
  };
};
```

- **Keep the ordering of methods in the header file and in the source files identical.** [method-ordering]

  This makes it easier to go back and forth between the declarations and the definitions.

- **Statements should not exceed 100 characters (excluding leading spaces). If possible, break long statements up into multiple ones.** [long-statements]

- **Limit line length to 120 character positions (including white space and expanded tabs).** [long-lines]

- **Include meaningful dummy argument names in function declarations. Any dummy argument names used in function declarations must be the same as in the definition.** [dummy-argument-names]

  Although they are not compulsory, dummy arguments make the class interface much easier to read and understand.

  For example, the constructor below takes two `Number` arguments, but what are they?

```
class Point
{
public:
  Point (Number, Number);
};
```

  The following is clearer because the meaning of the parameters is given explicitly.

```
class Point
{
public:
  Point (Number x, Number y);
};
```

- **The code should be properly indented for readability reasons.** [indenting]

  The amount of indentation is hard to regulate. If a recommendation were to be given then two to four spaces seem reasonable since it guides the eye well, without running out of space in a line too soon. The important thing is that if one is modifying someone else's code, the indentation style of the original code should be adopted.

  It is strongly recommended to use an editor that automatically indents code for you.

  Whatever style is used, if the structure of a function is not immediately visually apparent, that should be a cue that that function is too complicated and should probably broken up into smaller functions.

- **Do not use spaces in front of `[]` and to either side of `.` and `->`.** [spaces]

```
a->foo()   // Good
x[1]       // Good
b . bar()  // Bad
```

  Spacing in function calls is more a matter of taste. Several styles can be distinguished. First, not using spaces around the parentheses (K&R, Linux kernel):

```
foo()
foo(1)
foo(1, 2, 3)
```

  Second, always putting a space before the opening parenthesis (GNU):

```
foo ()
foo (1)
foo (1, 2, 3)
```

  Third, putting a space before the opening parenthesis unless there are no arguments.

```
foo()
```

```
foo (1)
foo (1, 2, 3)
```

Fourth, putting spaces around the argument list:

```
foo()
foo( 1 )
foo( 1, 2, 3 )
```

In any case, if there are multiple arguments, they should have a space between them, as above. A parenthesis following a C++ control keyword with as `if`, `for`, `while`, and `switch` should always have a space before it.

- **Keep the style of each file consistent within itself.** [style-consistency]

  Although standard appearance among ATLAS source files is desirable, when you modify a file, code in the style that already exists in that file. This means, leave things as you find them. Do not take a non-compliant file and adjust a portion of it that you work on. Either fix the whole thing, or code to match.

- **Prefer `using` to `typedef`.** [prefer-using]

  To declare a type alias, prefer the newer `using` syntax:

  ```
  using Int_t = int;
  ```

  to the `typedef` syntax:

  ```
  typedef int Int_t;
  ```

  The `using` syntax makes it clearer what is being defined; it can also be used to declare templated aliases.

4.2 Comments

- **Use Doxygen style comments before class/method/data member declarations. Use "//" for comments in method bodies.** [doxygen-comments]

  ATLAS has adopted the Doxygen code documentation tool, which requires a specific format for comments. Doxygen comments either be in a block delimited by `/** */` or in lines starting with `///`. We recommend using the first form for files, classes, and functions/methods, and the second for data members.

```
/**
 * @file MyPackage/MyClusterer.h
 * @author J. R. Programmer
 * @date April 2014
 * @brief Tool to cluster particles.
 */

#ifndef MYPACKAGE_MYCLUSTERER_H
#define MYPACKAGE_MYCLUSTERER_H


#include "MyPackage/ClusterContainer.h"
#include "xAODBase/IParticleContainer.h"
#include "AthenaBaseComps/AthAlgTool.h"


namespace MyStuff {


/**
 * @brief Tool to cluster particles.
 *
 * This tool forms clusters using the method
 * described in ...
 */
class MyClusterer
{
public:
  ...

  /**
   * @brief Cluster particles.
   * @param particles List of particles to cluster.
   * @param[out] clusters Resulting cluster list.
```

```
   *
   * Some additional description can go here.
   */
  StatusCode
  cluster (const xAOD::IParticleContainer& particles,
           ClusterContainer& clusters) const;

  ...

private:
  /// Property: Cluster size.
  float m_clusterSize;

  ...
};


} // namespace MyStuff


#endif // MYPACKAGE_MYCLUSTERER_H
```

See the ATLAS Doxygen page [18].

Remember that the `/* */` style of comment does not nest. If you want to comment out a block of code, using `#if 0` / `#endif` is safer than using comments.

- **All comments should be written in complete (short and expressive) English sentences.** [english-comments]

  The quality of the comments is an important factor for the understanding of the code. Please do fix typos, misspellings, grammar errors, and the like in comments when you see them.

- **In the header file, provide a comment describing the use of a declared function and attributes, if this is not completely obvious from its name.** [comment-functions]

```
class Point
```

```
{
public:
  /**
   * @brief Return the perpendicular distance
   *        of the point from Line @c l.
   */
  Number distance (Line l);
};
```

The comment includes the fact that it is the perpendicular distance.

5 Changes

5.1 Version 2.1 (Jan 1, 2026)

- Migrated source to pandoc-markdown. Produce mkdocs-compatible output. Minor edits.

5.2 Version 2.0 (March 6, 2024)

- Updated for C++20.
  - Don't use modules or coroutines.
  - Add recommendation to use `<numbers>`.
  - Suggest using `auto` to move the return type to the end of a method signature when returning types defined within the class.
  - Suggest not defining template functions without the `template` keyword.
  - Recommend `std::format` for formatted output.
  - Note that range-for can have init-statements.
  - Mention `std::bit_cast`.
  - Recommend `using` instead of `typedef`. Rephrase previous references to `typedef`.
  - Comparisons should be defined in terms of `operator==` and `operator<=>`.
  - Mention `std::span`.
- Some additional references.
- Clarify that non-ASCII characters should not be used in identifier names.

- Clarify that variable-length argument lists of variadic template functions are OK.

5.3 Version 0.7 (Sep 18, 2019)

- Minor cleanups and updates to take into account that we now require C++17.
- Use the `fallthrough` attribute, not a comment.
- Allow omitting the `default` clause in a `switch` statement on an `enum` that handles all possible values. Recent compilers will warn if some values are not handled, and it's better to get such a diagnostic at compile-time rather than at runtime.
- Clarify avoid-typedef section.
- Mention preference for `ATH_MSG_` macros.
- Don't require `override` for destructors.
- Avoid using `#pragma once`.

5.4 Version 0.6 (Dec 20, 2017)

- The `register` keyword is an error in C++17.
- Dynamic exception specifications are errors in C++17.
- Exceptions should be caught using const references, not by value.
- Discourage using protected data.

5.5 Version 0.5 (Nov 21, 2017)

- Add an initial set of guidelines for AthenaMT.
- Add recommendation to prefer range-based for.

5.6 Version 0.4 (Nov 16, 2017)

- Minor updates: we're now using c++14. Add note about implicit fallthrough warnings with gcc7. Add rule to use std::abs().

5.7 Version 0.3 (Aug 23, 2017)

- Add recommendation to avoid bit fields.

1486 5.8 Version 0.2 (Aug 9, 2017)

1487 - Small typo fixes.
1488 - Add a brief description of pointer aliasing.
1489 - Add more details about argument passing to functions.
1490 - Add recommendation on `auto`.

1491 References

1492 [1] D. Knuth, Literate programming, The Computer Journal **27**, 97 (1984).

1493 [2] ATLAS Quality Control Group, ATLAS C++ Coding Standard, ATL-SOFT-2002-001, 2001.

1494 [3] CERN Project Support Team, C++ Coding Standard, CERN-UCO/1999/207, 2000.

1495 [4] Standard for the Programming Language C++, n4868, 2020.

1496 [5] C++ reference, (n.d.).

1497 [6] News, Status & Discussion about Standard C++, (n.d.).

1498 [7] C++ Stories, (n.d.).

1499 [8] R. Grimm, Modernes C++, (n.d.).

1500 [9] H. Sutter, Guru of the week archive, (2008).

1501 [10] H. Sutter, Guru of the week archive, (2013).

1502 [11] S. Meyers, *Effective C++, 3rd Edition* (Addison-Wesley, 2005).

1503 [12] S. Meyers, *Effective STL* (Addison-Wesley, 2001).

1504 [13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns, Elements of Reusable Object-Oriented Software* (Addison-Wesley, 1994).

1505 [14] E. Niebler, Range library for C++, (n.d.).

1506 [15] H. Sutter, A Pragmatic Look at Exception Specifications, C++ Users Journal **20**, (2002).

1507 [16] A. Krzemieński, noexcept — what for?, (2014).

1508 [17] FaqCompileTimeWarnings ATLAS wiki page, (n.d.).

1509 [18] DoxygenDocumentation ATLAS wiki page, (n.d.).