C++ 11 Guidelines

With ATS 7.0 the compiler requirements will be changed so that the minimum compiler support will be C++ 11 compliant. This represents a significant change in language capabilities and this document is about how the ATS team should use those capabilities.

⚠ THIS IS A PRELIMINARY DOCUMENT PROVIDED FOR REVIEW

Specific features and use guidelines

Required

These features must be used. These are not optional.

- `nullptr` - The use of a literal `0` or `NULL` will no longer be allowed. `nullptr` is in all ways superior to either of these.
- `override` - when overriding a method, the `override` qualifier must be used. If the method is virtual then the `virtual` qualifier must also be used. Consistent use of these features will prevent subtle and hard to diagnose errors.

Strongly Recommended

These features should be use when possible. These are features with good value in almost all circumstances. Exceptions are permitted but must have some justification.

- Container style for loops.
- `constexpr` - use this instead of complex `#define` values.
- `static_assert` - in preference to any `#define` or run time check if possible.
- member initialization - Use for members initialized to the same value in all constructors.
- method `delete` and `default` - rather than the current hack for preventing copying, these should be used.
- unique_ptr - single allocated resources that need to be cleaned up (in particular strings) should use this rather than depending on explicit clean up code. (or use ats_scoped_str? Need to consider this)

Recommended

These are mechanisms that are usually useful but can be harmful if over used.

- `auto` - Use when the type isn't obvious or the variable is used only in a limited single scope (e.g. for loop variables). Remember that using auto means no one reviewing the code will know the type either. The most command and best use is as the type for iterators in a for loop, but use of the container style for loop is even better.
- `lambda` - Useful for one shot functions that are only used locally. This can make std::for_each
- Initializer lists and uniform initialization - handy for disambiguating initializations and improving over all code consistency.
- Cross construction invocation - This can clean up code if a class has a variety of constructors and all of them initialize the same part of the class in the same way. Rather than cutting and pasting or introducing an artificial init() method that initialization can be delegated to a base constructor. In many cases, however, direct member initialization can do this as well and that is the preferred mechanism.
- Enum raw type control - it is now possible to control the underlying type of an enum. This is useful in cases where the size matters (e.g., byte vs. 4 bytes).

Use With Caution

These mechanisms are useful in more limited circumstances and if used improperly can cause problems. They require a deeper understanding of C++ mechanics to use effectively. If you are unsure about whether to use any of these, don't.

- `std::move` and move constructors - These can greatly increase efficiency when used correctly. When used inappropriately these can use the sort of problems that caused auto_ptr to be removed from the language. The classic use case is for objects that contain a single or small set of pointers which can be cheaply moved in comparison to allocating and copying the pointed to data.
- Enum classes - this prevents enum names from "leaking" in to the surrounding scope which can be good or bad depending on circumstances.
- `std::function` - this captures function objects so that methods, free functions, lambdas, and functors can be handled uniformly. Interfaces that accept callbacks can use this to simplify both the interface and the internal implementation while providing more choice to the caller.

Hardcore

These mechanisms are deep C++ and just the use cases can be hard to explain. Use with much caution. If you need an explanation of what these terms mean, you shouldn't be using them. All of them, however, solve serious problems in the language and are therefore appropriate to use in such circumstances.

- `decltype`
- alternate function declaration
- `extern` template
- variadic template declaration
- perfect forwarding
- `final`

Avoid

tbd

Related articles

- C++ 11 Guidelines