OpenJDK
Installing
Contributing
Sponsoring
Developers' Guide
Vulnerabilities
JDK GA/EA Builds
Mailing lists
Wiki · IRC
Bylaws · Census
Legal
Workshop
JEP Process
Source code
Mercurial
GitHub
Tools
Git
jtreg harness
Groups
(overview)
Adoption
Build
Client Libraries
Compatibility & Specification Review
Compiler
Conformance
Core Libraries
Governing Board
HotSpot
IDE Tooling & Support
Internationalization
JMX
Members
Networking
Porters
Quality
Security
Serviceability
Vulnerability
Web
Projects
(overview, archive)
Amber
Audio Engine
CRaC
Caciocavallo
Closures
Code Tools
Coin
Common VM Interface
Compiler Grammar
Detroit
Developers' Guide
Device I/O
Duke
Font Scaler
Galahad
Graal
Graphics Rasterizer
IcedTea
JDK 7
JDK 8
JDK 8 Updates
JDK 9
JDK (..., 21, 22)
JDK Updates
JavaDoc.Next
Jigsaw
Kona
Kulla
Lambda
Lanai
Leyden
Lilliput
Locale Enhancement
Loom
Memory Model Update
Metropolis
Mission Control
Modules
Multi-Language VM
Nashorn
New I/O
OpenJFX
Panama
Penrose
Port: AArch32
Port: AArch64
Port: BSD
Port: Haiku
Port: Mac OS X
Port: MIPS
Port: Mobile
Port: PowerPC/AIX
Port: RISC-V
Port: s390x
Portola
SCTP
Shenandoah
Skara
Sumatra
Tiered Attribution
Tsan
Type Annotations
Valhalla
Verona
VisualVM
Wakefield
Zero
ZGC
ORACLE

JEP 347: Enable C++14 Language Features

Owner	Kim Barrett
Type	Infrastructure
Scope	Implementation
Status	Closed / Delivered
Release	16
Component	hotspot / other
Discussion	hotspot dash dev at openjdk dot java dot net
Reviewed by	David Holmes, Mikael Vidstedt
Endorsed by	Mikael Vidstedt
Created	2018/07/23 14:30
Updated	2023/06/17 05:24
Issue	8208089

Summary

Allow the use of C++14 language features in JDK C++ source code, and give specific guidance about which of those features may be used in HotSpot code.

Goals

Through JDK 15, the language features used by C++ code in the JDK have been limited to the C++98/03 language standards. With JDK 11, the code was updated to support building with newer versions of the C++ standard, although it does not yet use any new features. This includes being able to build with recent versions of various compilers that support C++11/14 language features.

The purpose of this JEP is to formally allow C++ source code changes within the JDK to take advantage of C++14 language features, and to give specific guidance about which of those features may be used in HotSpot code.

Non-Goals

This JEP does not propose any usage or style changes for C++ code in the JDK that is outside of HotSpot. The rules are already different for HotSpot and non-HotSpot code. For example, C++ exceptions are used in some non-HotSpot code, but are disallowed in HotSpot by build-time options. However, build consistency requirements will make the newer language features available for use in all C++ code in the JDK.

Description

Changes to the Build System

To take advantage of C++14 language features some build-time changes are required, with specifics depending on the platform compiler. Minimum acceptable versions of the various platform compilers also need to be specified. The desired language standard should be specified explicitly; later compiler versions might default to a later, and possibly incompatible, language standard.

- Windows: Visual Studio 2017 is required for JDK 11. (Earlier versions will generate a configure-time warning and may or may not work.) For Visual Studio 2017 the default C++ standard is C++14. The `/std:c++14` option should be added. Support for older versions will be dropped entirely.
- Linux: Replace the `-std=gnu++98` compiler option with `-std=c++14`. The minimum supported version of gcc is 5.0.
- macOS: Replace the `-std=gnu++98` compiler option with `-std=c++14`. The minimum supported version of clang is 3.5.
- AIX/PowerPC: Replace the `-std=gnu++98` compiler option with `-std=c++14` and require the use of xlclang++ as the compiler. The minimum supported version of xlclang++ is 16.1.

Changes to C++ Usage in HotSpot code

The existing restrictions and best-practice recommendations for C++ usage in HotSpot code are based on the C++98/03 language standard, and described in the HotSpot Style Guide.

We will add similar restrictions and guidelines for features of more recent language standards to that document. They will be described by a table of permitted features and another of excluded features. Use of permitted features may be unconditional or may have some restrictions or additional guidance. Use of excluded features is forbidden in HotSpot code.

There is a third category, undecided features, about which HotSpot developers have not reached a consensus, or possibly discussed at all. Use of these features is also forbidden.

Note, however, that the use of some language features may not be immediately obvious and may accidentally slip in anyway, since the compiler will accept them. As always, the code review process is the main defense against this.

Proposed changes to a feature's categorization are approved by rough consensus of the HotSpot Group members, as determined by the Group Lead. Such changes must be documented in updates to the Style Guide.

Lists of new features for C++11 and C++14, along with links to their descriptions, can be found in the online documentation for some of the compilers and libraries:

- C++ Standards Support in GCC
- C++ Support in Clang
- Visual C++ Language Conformance
- libstdc++ Status
- libc++ Status

As a rule of thumb, permitting features which simplify writing code and, especially, reading code, are encouraged.

HotSpot has largely avoided using the C++ Standard Library. Some of the reasons for that may be obsolete (in particular, bugs encountered in early versions), while others may still be applicable (minimizing dependencies). Categorizing pieces of the Standard Library should go through the same process as language features.

An initial set of feature categorizations for HotSpot follows.

Permitted

- `constexpr`
  - Relaxing requirements on `constexpr` functions (n3652)
  - Generalized constant expressions (n2235)

  Relaxed `constexpr` (n3652) is perhaps the key feature distinguishing C++14 from C++11. The `constexpr` feature will permit the elimination of some clumsy macro code in favor of `constexpr` functions. This feature is also the foundation for simplified metaprogramming idioms. See mpl11 and mpl11_2.

- Sized deallocation (n3778) — Syntax is already in use, because of Visual Studio.

- Variadic templates
  - Variadic templates (n2242)
  - Extending variadic template template parameters (n2555)

  Probably only occasionally useful directly, but foundational for simplified metaprogramming idioms such as described in mpl11 and mpl11_2.

- Static assertions (n1720) — Replaces HotSpot STATIC_ASSERT macro, providing better error messages.

- `decltype`
  - Declared type of an expression (n2343)
  - `decltype` and call expressions (n3276)

  Important metaprogramming tool. Needed to implement a simplified function template SFINAE utility.

- Right angle brackets (n1757) — Eliminates an annoying syntactic wart.

- Default template arguments for function templates (CWG D226) — In addition to being useful in its own right, this is needed to implement a simplified function template SFINAE utility.

- Template aliases (n2258) — Typedefs with template parameters. Provides syntax for partial specializations of a class template, rather than using inheritance (in sometimes inappropriate ways). Also used as metafunctions in simplified approach to metaprogramming mpl11 and mpl11_2.

- Strongly-typed enums (n2347) — Allows explicit control of the underlying type for an enum, rather than leaving it potentially implementation defined (and varying between implementations). Also allows strong typing for enum classes, eliminating implicit conversions. It is recommended that *scoped-enums* be used when the enumerators are indeed a logical set of values. Use of *unscoped-enums* is permitted, though ordinary constants should be preferred when the automatic initializer feature isn't used. The *enum-base* should always be specified explicitly, rather than leaving it dependent on the range of the enumerator values and the platform.

- Delegating constructors (n1986) — Eliminate some code duplication and simplification by allowing specialized constructors to chain to more general constructors.

- Explicit conversion operators (n2437) — Use to make some existing conversion operators safe.

- Standard Layout Types (n2342)

- Defaulted and deleted functions (n2346)

- Dynamic initialization and destruction with concurrency (n2660) — Thread-safe function-local statics.

- `<type_traits>` is a core metaprogramming library. It eliminates the need for many of the HotSpot metaprogramming utilities, which were modeled on corresponding parts of this library.

- `final` virtual specifiers for classes and virtual functions (n2928), (n3206), (n3272) — The `final` specifier permits devirtualization of virtual function calls. This can provide better performance than relying on the compiler's use of techniques such as points-to analysis or speculative devirtualization. The `overrides` specifier for virtual functions, which is also described in the referenced papers, may be considered at a later time.

- Local and unnamed types as template parameters (n2657) — Allows local definition of used-once helper classes to be placed near the point of use when the use is a template, rather than requiring such helpers be defined at namespace scope.

- `nullptr` and `std::nullptr_t` (n2431)
- `auto` variable declarations (n1984) — Use only when it makes the code clearer or safer. Do not use it merely to avoid the inconvenience of writing an explicit type, unless that type is itself difficult to write. For local variables, this can be used to make the code clearer by eliminating type information that is obvious or irrelevant. Excessive use can make code much harder to understand.
- Function return type deduction (n3638) — Only use if the function body has a very small number of `return` statements, and generally relatively little other code.
- Expression SFINAE (n2634)

Excluded

- New string and character literals
  - New character types (n2249)
  - Unicode string literals (n2442)
  - Raw string literals (n2442)
  - Universal character name literals (n2170)

  HotSpot doesn't need any of the new character and string literal types.
- User-defined literals (n2765) — User-defined literals should not be added casually, but only through a proposal to add a specific UDL.
- Inline namespaces (n2535) — HotSpot makes very limited use of namespaces.
- `using namespace` directives. In particular, don't use `using namespace std;` to avoid needing to qualify Standard Library names.
- Propagating exceptions (n2179) — HotSpot does not permit the use of exceptions, so this feature isn't useful.
- `thread_local` (n2659) — Use HotSpot `THREAD_LOCAL` macro; see discussion for JDK-8230877.
- `<atomic>` (n2427), (n2752); instead, use the HotSpot `Atomic` class and related facilities.
- `[[deprecated]]` attribute (n3760) — Not relevant in HotSpot code.

Similar lists for some other projects

- Google C++ Style Guide — Currently targeting C++11 and should not use C++14
- C++11 and C++14 use in Chromium — Categorizes features as allowed, banned, or to be discussed.
- llvm Coding Standards — Currently targeting C++11 and should not use C++14.
- Using C++ in Mozilla code — C++14 support is required.

Risk and Assumptions

There may be other platforms with toolchains that do not yet support the C++14 language standard.

There may be bugs in the support for some new features by some compilers.