## C++17 Core Language Features

# Deprecated & Removed Features

A screenshot of a computer

Description automatically generated

A computer screen shot of a keyword

Description automatically generated

**A screenshot of a computer error

Description automatically generated**

A screenshot of a computer

Description automatically generated

A screenshot of a computer code

Description automatically generated

A screenshot of a computer program

Description automatically generated

A screenshot of a computer function

Description automatically generated

Summary of the features that were deprecated in C++11 and C++14 and then removed in C++17:

1. **Register Storage Class**:
   * In C++98, **register** was a storage class specifier hinting the compiler to store the variable in a register for optimization.
   * Removed in C++17 as modern compilers no longer need this optimization hint.
   * Even if used, it has no effect on program semantics, and compilers may issue warnings about its use.
2. **Increment Operator (++) on bool Type**:
   * In pre-C++17 versions, the increment operator **++** could be applied to a **bool** type.
   * Removed in C++17, and decrement operators (**--**) were never allowed for **bool** types.
   * Attempting to compile code that uses **++** with a **bool** type in C++17 results in a compiler error.
3. **Trigraphs**:
   * In pre-C++17, trigraphs were used to represent special characters that some keyboards couldn't input directly.
   * Removed in C++17 as modern keyboards support special characters, making trigraphs unnecessary.
   * Their removal can speed up the compilation process as the compiler no longer needs to translate trigraphs into different characters.
4. **Exception Specifications**:
   * In C++98, exception specifications allowed functions to advertise the exceptions they may throw.
   * Deprecated in C++11 and replaced by the **noexcept** specifier, which provides stronger guarantees.
   * Using exception specifications with modern compilers may result in warnings.
5. **std::auto\_ptr**:
   * Introduced in C++98, **std::auto\_ptr** was a smart pointer with problematic ownership semantics.
   * Removed in C++17 and replaced by safer alternatives such as **std::unique\_ptr** and **std::shared\_ptr**.
   * Legacy code using **std::auto\_ptr** should be updated to use modern smart pointers.
6. **Deprecated and Removed Library Functions**:
   * Several library functions and algorithms were deprecated and removed:
     + **std::random\_shuffle** was replaced with **std::shuffle**.
     + Unary function and **std::ptr\_fun** were replaced by other functions.
     + **std::bind1st** and **std::bind2nd** were replaced with **std::bind**.

These changes reflect the evolution of the C++ language and standard library, aiming for safer, more efficient, and more modern programming practices.

Top of Form

# Changes

A screenshot of a computer program

Description automatically generated

A white background with text and black text

Description automatically generated with medium confidence

A white background with black text

Description automatically generated

A computer screen shot of a code

Description automatically generated

A computer screen shot of a computer code

Description automatically generated

Here are some of the changes introduced in C++17 related to features from C++11 and C++14:

1. **Auto Type Deduction with Direct List Initialization**:
   * In C++11, using **auto** with direct list initialization always created an **initializer\_list**.
   * In C++17, two rules were introduced for direct list initialization:
     + If the list contains only one element, **auto** deduces the type from that element.
     + If there are multiple elements, it results in an ill-formed code.
   * Copy list initialization always deduces the type as **initializer\_list**.
2. **Range-based For Loop**:
   * Until C++14, the begin and end iterators in a range-based for loop had to be of the same type.
   * In C++17, this restriction is removed, allowing begin and end iterators to be of different types, as long as they support the **!=** operator for comparison.
3. **Static Assert**:
   * Introduced in C++11 for compile-time assertions, **static\_assert** accepts a condition and an optional message.
   * In C++17, the message parameter became optional. If the condition fails, the compiler issues an error, and the optional message is displayed as part of the error.

These changes in C++17 enhance the language's usability, flexibility, and expressiveness, allowing for more concise and safer code.

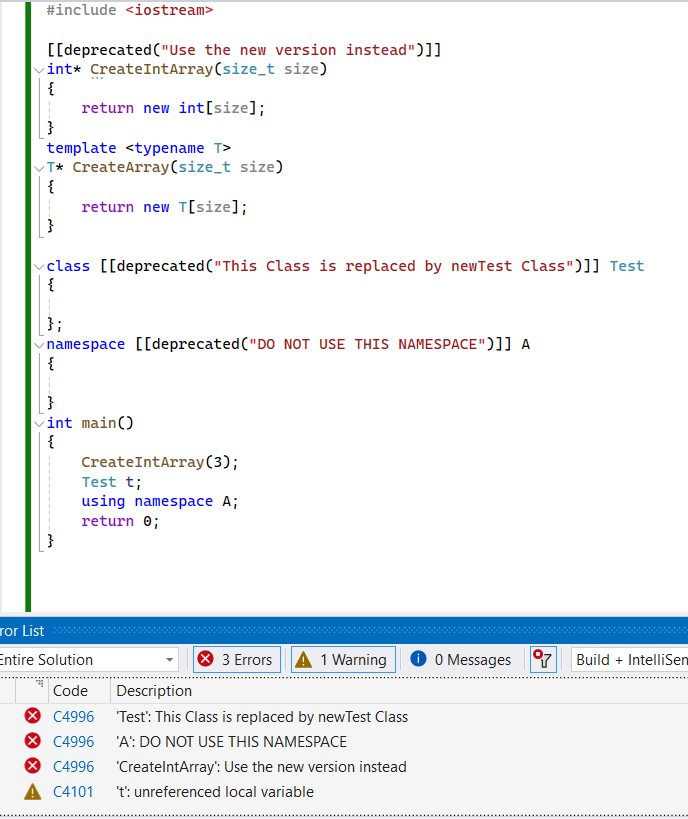
# AttributesTop of Form

A screenshot of a computer program

Description automatically generated

A green and white text with black text

Description automatically generated



A screenshot of a computer program

Description automatically generated

In C++, attributes are used to provide additional information to the compiler, which it may use for optimization or checks. Before C++11, different compiler vendors used different keywords for attributes, such as **\_\_declspec** for Visual Studio and **\_\_attribute** for GCC.

C++11 introduced some standard attributes, making code more portable across different compilers. However, custom attributes are still not supported in C++, and only built-in attributes can be used. C++14 and C++17 introduced additional attributes that can be used in code.

**Here are a few examples of attributes introduced in C++17:**

1. **[[deprecated]] Attribute**:
   * This attribute indicates that a function, class, or namespace is deprecated and should not be used.
   * It can accept an optional message, providing information about why it is deprecated.
   * Example: **[[deprecated("This class is replaced by NewTestClass")]] class OldTestClass {};**
2. **[[nodiscard]] Attribute**:
   * This attribute indicates that the return value of a function should not be ignored.
   * When applied to a function, it warns if the return value is not used.
   * Example: **[[nodiscard]] int getValue();**

These attributes help improve code quality by providing additional information to developers and allowing compilers to perform more checks and optimizations. However, it's important to note that attribute support may vary among different compilers, and some attributes may only generate warnings instead of errors, depending on the compiler used.

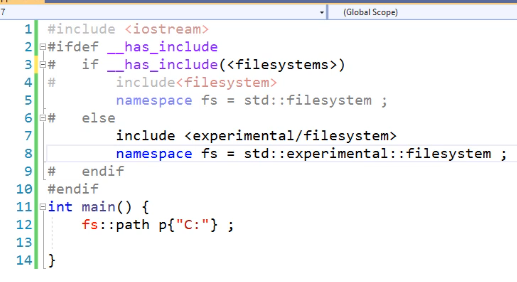
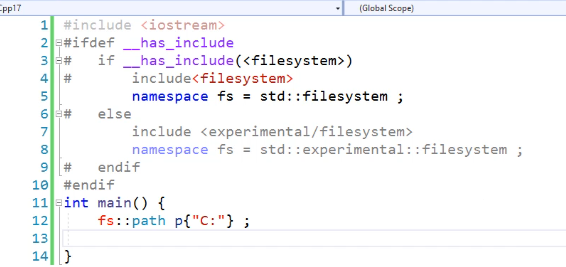
# Feature Test Macros

A screenshot of a test

Description automatically generated

The **\_\_has\_include** macro introduced in C++17 is a feature test macro used to check if a particular header file is available for inclusion. This macro is particularly useful for tracking the availability of header files, especially in cases where different compiler vendors may provide different implementations or experimental versions of certain features.

Here's how you can use the **\_\_has\_include** macro:



In this example:

* If the **<filesystem>** header is available, it is included, and the **filesystem** namespace alias is created.
* If **<filesystem>** is not available but **<experimental/filesystem>** is available, the experimental version is included, and the **filesystem** namespace alias is created under the **experimental** namespace.
* If neither header is available, a compilation error is issued.

The **\_\_has\_include** macro can only be used within preprocessor directives like **#if** and **#elif**, and it helps in writing more portable code by automatically including the correct header file based on its availability.

In addition to **\_\_has\_include**, C++20 introduced more feature test macros like **\_\_cpp\_capture\_star\_this**, which indicate the month and year when a particular feature was added to the working draft of the C++ standard. These feature test macros are useful for checking the availability of specific C++ language features and can be used with preprocessor directives as well. However, it's essential to note that these macros are part of the C++20 standard.

Top of Form

# If & switch With Initialization

* In C and C++, it's common to initialize variables within the scope of a loop or conditional statement to limit their visibility and avoid polluting the outer scope. This practice helps manage variables more efficiently and keeps the code clean.
* In C++17, the initialization capability of conditional statements (such as **if** and **switch**) was enhanced, allowing for variable initialization directly within the statement itself. This enhancement helps further reduce the scope of variables and makes the code more concise.

**Enhanced if Statement:**

A computer code with many lines

Description automatically generated with medium confidence

**Enhanced switch Statement:**

A screen shot of a computer code

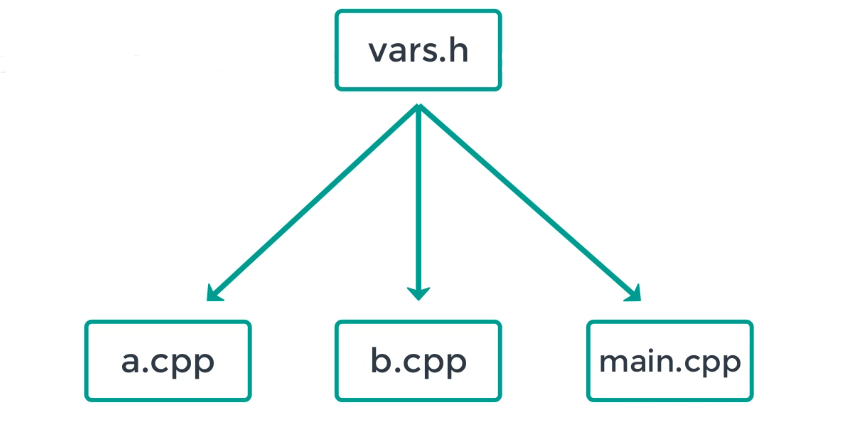
Description automatically generated

A screen shot of a computer code

Description automatically generated**Example Usage:**

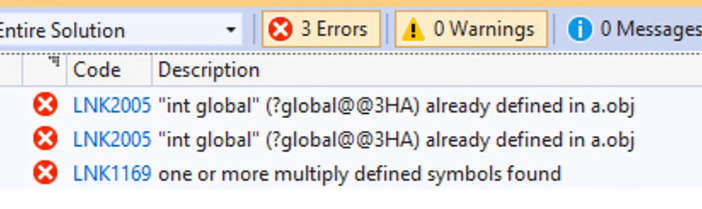
In both cases, the scope of the initialized variable (**x** in the **if** statement and **errorCode** in the **switch** statement) is limited to the respective block, making the code more readable and reducing the risk of variable misuse outside of its intended scope.

# inline Variables



**the challenges and solutions related to using global variables across multiple source files in C++. Here's a summary:**

1. **Global Variables and Header Files:** Global variables declared and initialized in header files can lead to linker errors because the definition of the variable appears in multiple source files. This violates the one definition rule in C++.



1. **Using extern Keyword:** To resolve linker errors, global variables should not be defined in header files. Instead, they should be declared as **extern** in header files and defined in one of the source files. This allows them to be accessed in multiple source files without violating the one definition rule.
2. **inline Keyword:** In C++17, the **inline** keyword can be used with global variables. This allows the variables to have multiple definitions across different source files, but the linker will pick only one definition. This can simplify the process of using global variables, but it's important to ensure that wherever the variable is used, it's declared as **inline** or **extern**.

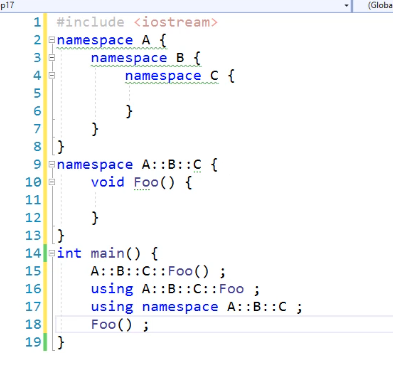


1. **inline with Static Member Variables:** While the **inline** keyword can be applied to static member variables inside a class, it doesn't change their behavior significantly. It doesn't make them implicitly inline, and defining them inside the class itself doesn't change the need for initialization in a source file. However, in C++17, **inline** with static member variables can simplify header-only libraries, especially when dealing with classes containing static member variables.

A computer code with text

Description automatically generated with medium confidence

# Nested Namespaces.



1. **Purpose of Namespaces:** Namespaces are used to logically group classes and functions together, providing a way to avoid naming conflicts and organize code more effectively.
2. **Nested Namespaces:** C++ allows the creation of nested namespaces, where one namespace is defined within another. This allows for further organization and structuring of code.
3. **C++17 Improvement:** In C++17, the ability to create nested namespaces is enhanced through the scope resolution operator (::). This means that there's no need to explicitly use the **namespace** keyword before every namespace declaration, simplifying the syntax for working with nested namespaces.
4. **Accessing Nested Namespaces:** Functions and classes defined within nested namespaces can be accessed from other parts of the code using various methods:
   * Full qualified name: specifying the complete namespace path to the function or class.
   * Using declaration: introducing specific names from a namespace into the current scope.
   * Global using directive: importing all names from a namespace into the current scope.
5. **Syntactic Convenience:** Using the scope resolution operator for creating nested namespaces is described as a syntactic convenience for programmers, making the code more readable and manageable.

Overall, the discussion highlights the importance of namespaces in organizing code and the improvements introduced in C++17 for working with nested namespaces.

# noexcept

A screenshot of a computer program

Description automatically generated

In C++ 98, there was a feature called exception specifications, which allowed functions to declare what types of exceptions they could throw. For instance, a function could specify that it throws only integer or float type exceptions. However, this feature was not widely adopted in the industry, and in C++11, it was deprecated and replaced by the **noexcept** keyword.

C++17 completely removes the exception specifications feature from the language. This removal is not considered a significant loss because even under C++98, many compilers did not fully support this feature, especially when used alongside modern C++ features.

Instead of exception specifications, C++17 relies solely on the **noexcept** keyword for exception handling. This keyword is used to specify that a function does not throw any exceptions. In C++11 and C++14, **noexcept** was not part of the function state, which could lead to potential issues. However, in C++17, **noexcept** is integrated into the function state, providing better clarity and preventing potential problems.

An example illustrating the change in behavior between C++11/14 and C++17 involves function pointers with exception specifications. In earlier versions, if a function pointer with an exception specification was initialized with the address of a function that could potentially throw a different type of exception, it could lead to runtime errors. However, in C++17, this scenario is flagged as an error at compile-time, ensuring type safety and adherence to the **noexcept** contract.

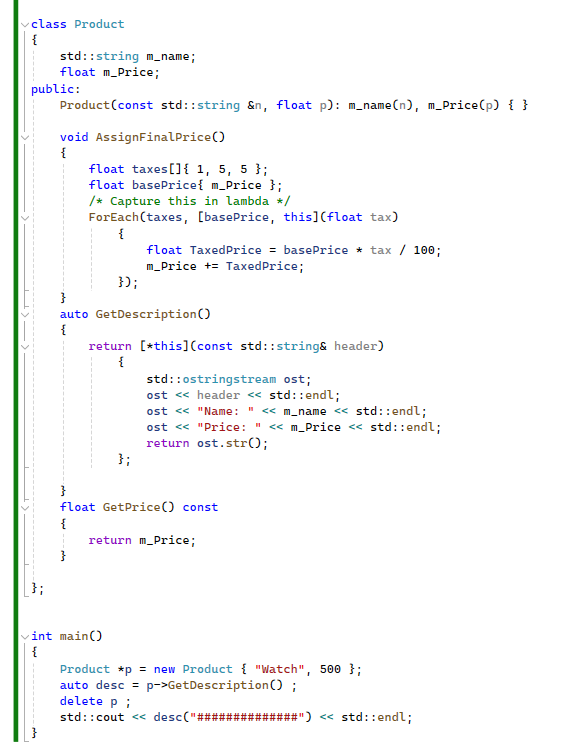
# constexpr Lambda

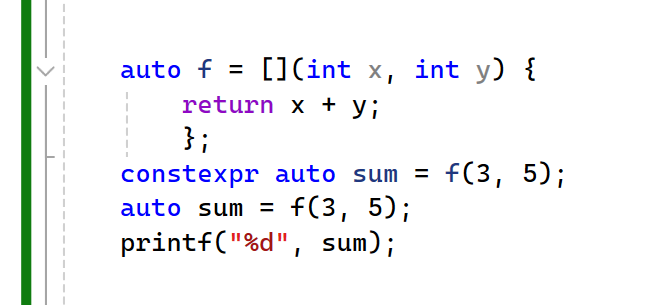
In C++17, lambda expressions received two significant enhancements:

1. **Capture \*this for Improved Safety**:
   * In previous versions of C++, capturing **this** inside a lambda expression would result in a pointer to the enclosing class instance being captured. However, this approach posed a risk of dangling pointers if the lambda outlived the object it captured.
   * C++17 introduced the ability to capture **\*this**, which captures a copy of the enclosing class instance by value. This ensures that the lambda holds a valid copy of the object, even if the original object goes out of scope.
   * By capturing **\*this**, a copy of the class instance is created inside the lambda expression, preventing potential issues with dangling pointers.
   * This enhancement provides greater safety and reliability when using lambda expressions in C++.
2. **Automatic constexpr Evaluation**:
   * In C++17, lambda expressions that meet the criteria to be evaluated at compile-time as **constexpr** functions automatically become constant expressions.
   * This means that if a lambda expression can be evaluated at compile-time, it will be treated as a constant expression without the need for explicit use of the **constexpr** keyword.
   * The compiler evaluates such lambda expressions at compile-time whenever possible, resulting in more efficient code execution.
   * If a lambda expression is used in a context requiring a constant expression, C++17 automatically evaluates it as such, simplifying code and improving performance.
   * However, if the lambda expression is used in a non-constant expression context, it is evaluated at runtime as usual.
   * This enhancement streamlines the usage of **constexpr** functionality, making it easier to create efficient and optimized code.

These enhancements to lambda expressions in C++17 significantly improve their functionality, safety, and performance, making them even more powerful tools for modern C++ development.

Top of Form





# Structured Bindings

A white paper with black text

Description automatically generated

A white paper with black text

Description automatically generated

Structured bindings offer a concise and efficient way to initialize variables with the values of member variables of a class or elements of an array. This feature significantly reduces the verbosity of code that involves such initializations. Here's an overview of how structured bindings work and their benefits:

1. **Direct Initialization**:
   * Instead of creating variables and initializing them one by one with the values from members of a class or elements of an array, structured bindings allow you to initialize variables directly with the elements or members of an object.
   * This streamlined syntax reduces the amount of code required for initialization tasks, leading to more readable and maintainable code.
2. **Compatibility and Requirements**:
   * Structured bindings can be used to initialize variables from objects of classes, structures, or arrays.
   * When initializing from objects of classes, the members being accessed should be public.
   * The number of variables being initialized should match the number of elements in the object being accessed.

A computer screen shot of a program

Description automatically generated

A computer code with text

Description automatically generated

1. **Syntax and Usage**:

* The syntax for structured bindings involves declaring variables and initializing them with each member of the object in a single statement.
* References and const volatile qualifiers can also be applied during initialization.
* Structured bindings automatically create an anonymous entity, which is a copy of the object, and allow access to members through the bindings.

1. **Benefits**:

* Enhanced readability: Structured bindings improve code readability by directly associating values with meaningful variable names.
* Efficient initialization: By reducing the need for manual initialization steps, structured bindings make code more efficient and less error-prone.

1. **Examples**:

* Structured bindings can be used with user-defined classes, standard library classes like **pair**, and arrays.
* They facilitate the initialization of multiple variables in a single statement, enhancing code clarity and conciseness.

A screen shot of a computer code

Description automatically generated

1. **Array Handling**:

* When used with arrays, structured bindings preserve the array type and prevent automatic decay to a pointer, maintaining the integrity of the original array structure.
* A screen shot of a computer

  Description automatically generatedHowever, when an array is decayed to a pointer, as in some contexts, structured bindings will reflect this behavior.

A screenshot of a computer screen

Description automatically generated

Overall, structured bindings offer a powerful tool for simplifying variable initialization tasks in C++, leading to cleaner and more expressive code.