cppcoreguidelines for noexcept

The Power of `noexcept` in Modern C++: A Deep Dive into cppcoreguidelines

cppcoreguidelines for noexcept offers a crucial lens through which to understand and implement exception safety in modern C++ development. This powerful keyword, `noexcept`, has become an indispensable tool for writing more robust, performant, and predictable code. By clearly declaring that a function will not throw exceptions, you unlock significant optimizations for the compiler and provide invaluable guarantees to users of your code. This article will delve deeply into the rationale behind `noexcept`, explore its implications across various C++ constructs, and illuminate how the cppcoreguidelines steer us toward its judicious application, ensuring we leverage its benefits effectively while avoiding common pitfalls. We'll cover everything from the fundamental definition of `noexcept` to its intricate interactions with move semantics, destructors, and the overall exception safety strategy of your projects.

Table of Contents

- When Not to Use `noexcept`
- Practical Application and Best Practices

Understanding the `noexcept` Specifier in C++

The `noexcept` specifier, introduced in C++11, is a declarative keyword that asserts a function will not throw any exceptions. This declaration isn't merely a suggestion to the compiler; it's a contract. When you mark a function as `noexcept`, you're fundamentally promising that any execution path within that function will complete without invoking `throw` or propagating an exception. This promise has profound implications for how the compiler can optimize the generated code. It signals that the runtime overhead associated with exception handling – such as unwinding the stack and invoking destructors for objects in scope – can be bypassed for calls to such functions. Think of it like a highly guarded express lane; because you've guaranteed no unexpected detours (exceptions), the journey can be made much faster.

The `noexcept` specifier can also take a boolean constant expression, allowing for conditional exception guarantees. For instance, `noexcept(expression)` evaluates the `expression`. If it's true, the function is treated as `noexcept`; otherwise, it's not. This provides a nuanced way to express that a function's exception-free nature depends on certain conditions, often related to the properties of the types it operates on. This flexibility is vital for writing generic code that can adapt its exception behavior based on the context.

The `noexcept` Specifier: A Syntactic and Semantic Distinction

From a purely syntactic standpoint, `noexcept` is appended to a function signature, much like `const` or a return type. However, its semantic impact is far more substantial. It influences not only the

compiler's optimization strategies but also the compiler's ability to generate specialized versions of functions, particularly within template metaprogramming. When the compiler knows a function is `noexcept`, it can make aggressive optimizations that would be unsafe if exceptions were a possibility. For example, it can eliminate the need for stack unwinding code, which is a significant part of exception handling overhead.

Furthermore, the `noexcept` specifier plays a critical role in how functions are invoked and how control flow is managed. If a `noexcept` function does throw an exception (which should, by definition, never happen), the program will immediately terminate by calling `std::terminate()`. This is a drastic measure, but it underscores the severity of breaking the `noexcept` contract. It's a last resort to prevent undefined behavior that could arise from attempting to handle an exception where none was expected.

Why Use `noexcept`? The Compelling Benefits

The decision to use `noexcept` is driven by a desire for enhanced performance and improved code reliability. One of the most significant advantages is the potential for substantial performance gains. Compilers can perform aggressive optimizations on `noexcept` functions because they don't need to generate the fallback code required for exception handling. This includes eliminating stack unwinding mechanisms, simplifying control flow, and potentially inlining functions more readily. For performance-critical sections of code, these optimizations can lead to noticeable improvements in execution speed.

Beyond raw speed, `noexcept` significantly enhances code predictability and simplifies reasoning about program behavior. When you mark a function as `noexcept`, you are providing a strong guarantee to anyone using that function. They can rely on the fact that calling it won't disrupt their control flow with an unexpected exception. This simplifies error handling strategies, allowing developers to focus on handling anticipated error conditions rather than being constantly vigilant about potential exceptions from seemingly innocuous function calls.

Performance Optimizations Enabled by `noexcept`

The compiler's ability to optimize `noexcept` functions is a game-changer. Without the need to prepare for stack unwinding, the generated machine code can be leaner and more efficient. This means that for frequently called functions, or functions operating in tight loops, marking them as `noexcept` can yield tangible performance benefits. Consider a scenario where you have a simple getter function. If this getter were to throw, the compiler would have to generate code to manage potential exceptions. However, if it's marked `noexcept`, the compiler can generate a direct return, skipping all the exception-handling machinery.

The impact of these optimizations is particularly pronounced in environments where performance is paramount, such as embedded systems, high-frequency trading platforms, or game development. By judiciously applying `noexcept` to functions that are guaranteed not to throw, developers can fine-tune their applications for maximum efficiency without sacrificing correctness.

Improved Exception Safety and Predictability

One of the core tenets of writing robust software is exception safety. The `noexcept` specifier directly contributes to this by providing a clear and actionable way to enforce exception-free behavior. When a function is `noexcept`, it simplifies the exception safety guarantees you need to provide. For example, if a function is marked `noexcept`, it inherently offers the strongest exception safety guarantee: strong exception safety (i.e., if an exception occurs, the program state is not corrupted). This is because, by definition, no exception will occur.

This predictability is invaluable for maintaining complex codebases. Developers can quickly identify which parts of the code are guaranteed to execute without interruption, allowing them to reason about control flow and resource management more effectively. It helps delineate areas of the code where exception handling is necessary versus areas where it's a non-concern.

`noexcept` and Exception Safety Guarantees

Understanding the interplay between `noexcept` and exception safety guarantees is fundamental to writing robust C++ code. In C++, we often discuss different levels of exception safety: basic, strong, and nothrow (or no-failure). The `noexcept` specifier directly aligns with the "nothrow" guarantee, representing the highest level of exception safety. A function declared `noexcept` promises that it will never throw an exception. If, by some error in implementation, it does throw, the program will terminate via `std::terminate()`, preventing potentially catastrophic undefined behavior that could arise from trying to handle an unexpected exception.

This strong guarantee simplifies the design of other functions. If a function `f()` calls another function `g()` and `f()` is marked `noexcept`, it implies that `g()` must also be `noexcept` or that any exceptions thrown by `g()` must be caught and handled internally within `f()` without propagating. This helps propagate the `noexcept` guarantee through call chains, allowing for more predictable and optimizable code throughout the system.

The Nothrow Guarantee and Its Implications

The "nothrow" guarantee, embodied by `noexcept`, is the most stringent form of exception safety. It means that the operation is guaranteed to succeed without any exceptions being thrown. This is particularly important for operations that are fundamental to the state of objects, such as destructors and move operations. If a destructor were to throw an exception, it could lead to a cascade of problems, including resource leaks and program termination in unexpected places, especially during stack unwinding.

When you can confidently declare a function as `noexcept`, you are providing a solid foundation for the overall exception safety of your program. It means that the operations within that function are well-defined and will complete their intended task without the need for runtime exception handling. This

clarity is a significant advantage in managing the complexity of modern software.

How `noexcept` Simplifies Exception Safety Reasoning

The declarative nature of `noexcept` dramatically simplifies how developers reason about exception safety. When a function is marked `noexcept`, it's like a signpost indicating that you don't need to worry about exceptions originating from this particular path. This allows for a more structured approach to exception handling. You can identify critical sections of your code that must remain exception-free, such as those involved in resource management or essential state transitions.

This clarity also aids in debugging. If a `std::terminate()` call occurs, you know it likely originates from a violation of a `noexcept` contract. This significantly narrows down the search space for bugs, making it easier to pinpoint the source of the problem. Instead of tracing exception propagation, you're looking for places where a `noexcept` function might have been incorrectly implemented or called in a context where it's not safe.

`noexcept` in Constructors and Destructors

The application of `noexcept` to constructors and, especially, destructors is a critical aspect of writing safe and efficient C++ code. For destructors, the cppcoreguidelines strongly recommend making them `noexcept` by default. This is because destructors are automatically invoked during stack unwinding when an exception is thrown. If a destructor itself throws an exception, the program will call `std::terminate()`. This is a severe issue because it means that even if the initial exception was manageable, the program crashes anyway due to the destructor's failure.

Constructors, on the other hand, have a more nuanced story. While it's often desirable to make them `noexcept`, it's not always possible or safe. If a constructor's operations can fail in a way that requires throwing an exception to signal an error (e.g., failed memory allocation or I/O operations), then

marking it `noexcept` would be incorrect and dangerous. However, constructors that perform simple initialization or involve types that are themselves `noexcept` can often and should be marked `noexcept`.

The `noexcept` Guarantee for Destructors

As mentioned, destructors are special. Their automatic invocation during exception unwinding makes them prime candidates for the `noexcept` guarantee. Imagine an object managing a file handle. If an exception occurs in the function where this object was created, the destructor will be called to close the file. If, during the process of closing the file, an error occurs that causes the destructor to throw an exception, the program is in a dire state. It's trying to handle one exception and then encounters another, leading to `std::terminate()`. Therefore, destructors should be `noexcept` unless there is an extremely compelling and well-understood reason otherwise.

The cppcoreguidelines emphasize this point heavily. For destructors, unless you have a very specific, thoroughly analyzed reason for them to throw, assume they should be `noexcept`. This contributes to a more stable program by preventing cascading terminations during exception handling.

`noexcept` in Constructors: A Case-by-Case Analysis

Constructors present a more complex scenario. A constructor's primary job is to initialize an object. If that initialization can fail in a way that needs to be communicated to the caller via an exception, then the constructor should not be marked `noexcept`. For example, a constructor that attempts to allocate a large amount of memory might fail and throw `std::bad_alloc`. In such cases, `noexcept` would be inappropriate.

However, consider a constructor for a simple data structure that only initializes members with default values or from arguments that are themselves guaranteed not to throw. In these situations, marking

the constructor `noexcept` is beneficial. It allows for optimizations and provides a stronger guarantee to users of the class. The key is to carefully consider the operations within the constructor. If any of those operations can throw an exception, and that exception is the intended way to signal failure, then `noexcept` should not be applied. Otherwise, it should be considered.

`noexcept` with Move Operations

The interaction between `noexcept` and move constructors and move assignment operators is a cornerstone of efficient resource management in C++. Move operations (move constructor and move assignment operator) are designed to transfer ownership of resources from one object to another, often avoiding expensive deep copies. For these operations to be truly effective and predictable, especially in contexts like exception handling, they are often expected to be `noexcept`.

If a move operation is not `noexcept`, and an exception is thrown during a move, the original object might be left in an indeterminate state. This is problematic because the original object might still be in use or its resources could be partially transferred. Conversely, if move operations are `noexcept`, they provide a strong guarantee that the move will either succeed entirely or leave the source object in a well-defined state (often empty or unchanged), preventing disruptions during critical operations like container reallocations or exception handling.

The Importance of `noexcept` for Move Constructors

A move constructor's role is to efficiently initialize a new object from a temporary or expiring object. If this process can be guaranteed not to throw exceptions, marking it `noexcept` is highly advantageous. This is especially true within standard library containers. When a container like `std::vector` needs to reallocate its memory (e.g., when it grows), it uses move semantics if available to transfer the existing elements to the new memory location. If the move constructor is `noexcept`, the reallocation is guaranteed to succeed. If it throws, the container might be left in an inconsistent state, and the

program could terminate.

The cppcoreguidelines recommend that move constructors should be `noexcept` whenever possible. This means that the operations within the move constructor, such as transferring pointers or other resources, must be exception-safe. If a move constructor must potentially throw (e.g., if it involves an operation that itself might throw and cannot be caught internally), then it should not be marked `noexcept`.

`noexcept` for Move Assignment Operators

Similar to move constructors, move assignment operators benefit greatly from the `noexcept` specifier. A move assignment operator transfers resources from a source object to an existing target object. If this operation can be performed without throwing exceptions, marking it `noexcept` provides important guarantees. This is crucial for operations that might involve replacing the contents of an object, such as in `std::vector::operator=`. If the move assignment throws, the target object could be partially updated, and the source object's state might also be compromised.

When implementing move assignment operators, aim to make them `noexcept`. This usually involves ensuring that any underlying operations for transferring ownership are exception-safe. If a move assignment operator needs to perform an operation that can throw (and this exception is the intended way to signal failure), then it should not be marked `noexcept`. The default for move assignment operators, as with move constructors, should lean towards `noexcept` to ensure stability and performance.

`noexcept` and Standard Library Containers

The standard C++ library, particularly its container components like `std::vector`, `std::list`, and `std::map`, relies heavily on the `noexcept` specifier to ensure predictable and efficient behavior,

especially during operations that might involve reallocation or modification. The library's design often leverages `noexcept` to provide strong guarantees about the success of its operations.

For instance, when `std::vector` needs to grow, it typically performs a reallocation. It first allocates new memory, then moves the existing elements to the new location using move constructors (if available and `noexcept`), and finally deallocates the old memory. If the move constructors are `noexcept`, this entire reallocation process is guaranteed to succeed (barring memory allocation failures themselves, which typically throw `std::bad_alloc`). If the move constructors were not `noexcept` and threw an exception, the vector would be left in an inconsistent state, potentially corrupting the data it holds.

Container Reallocations and the `noexcept` Guarantee

Container reallocations are prime examples of where `noexcept` shines. Operations that trigger reallocation, such as `push_back` on a `std::vector` when its capacity is reached, must be exception-safe. The standard library achieves this by requiring that element move operations (move constructor and move assignment) be `noexcept`. If they are, the container can safely move elements to a new memory block. If an exception were to occur during a move, the `std::terminate()` handler would be invoked, preventing further corruption.

This dependency means that when you define your own types that are intended to be stored in standard containers, you should strive to make their move constructors and move assignment operators `noexcept` if they can safely do so. This not only benefits your custom types but also ensures they integrate seamlessly and predictably with the powerful algorithms and containers provided by the C++ standard library.

Impact on Container Efficiency and Safety

The pervasive use of `noexcept` within the standard library containers significantly contributes to their

efficiency and safety. By guaranteeing that operations like move semantics won't throw, the library can perform optimizations that would be unsafe otherwise. For example, if `std::vector` knows that moving elements is a no-throw operation, it can perform the reallocation with greater confidence and fewer error-checking mechanisms compared to a scenario where exceptions are a possibility.

This translates to real-world performance benefits. When your program frequently adds or removes elements from containers, the efficiency gained from `noexcept` move operations can be substantial. More importantly, it ensures that these operations are robust, preventing the dreaded situation where a container becomes corrupted due to an uncaught exception during a critical modification.

Common Pitfalls and cppcoreguidelines Recommendations

While `noexcept` offers significant advantages, its misuse or misunderstanding can lead to subtle bugs and unexpected program terminations. The cppcoreguidelines provide clear directives to help developers navigate these complexities and leverage `noexcept` effectively. One of the most common pitfalls is marking a function `noexcept` when it can actually throw exceptions. This is a direct violation of the `noexcept` contract and will result in `std::terminate()` if the exception is propagated.

Another area of concern is with templates. When writing generic code, it's easy to unintentionally make a template function `noexcept` even when the underlying types used in a specific instantiation do throw. This is where the conditional `noexcept(expression)` form becomes invaluable. The cppcoreguidelines often advocate for using this conditional form to ensure that template code correctly adapts its exception-free nature based on the types it operates on.

The Dangers of Incorrectly Marked `noexcept` Functions

The most dangerous pitfall is marking a function `noexcept` when it can, in fact, throw. This breaks the contract the specifier is meant to uphold. The consequences are severe: if an exception is thrown and

propagates out of such a function, `std::terminate()` will be called. This means your program will crash, and you'll have to debug the issue. Often, the root cause might be an exception from a library function that the `noexcept` function indirectly called, or a resource allocation failure.

The cppcoreguidelines strongly advise against this. Instead, if a function can throw, it should not be marked `noexcept`. If it's crucial for certain operations within the function to be exception-free, then those specific operations should be wrapped in `try-catch` blocks, and the exceptions handled internally without propagating. The function signature should reflect the actual exception-throwing behavior.

Leveraging Conditional `noexcept` for Generic Code

Templates offer immense power, but they also introduce complexity when dealing with exception specifications. A function template might be perfectly `noexcept` when used with certain types, but throw when used with others. The `noexcept(expression)` syntax is the solution. For example, you might write a generic swap function that is `noexcept` only if the underlying type's swap operation is itself `noexcept`.

The cppcoreguidelines recommend using conditional `noexcept` judiciously in template code. This ensures that your generic functions behave correctly and safely regardless of the specific types they are instantiated with. It allows for maximum optimization when possible, while still respecting the exception-throwing capabilities of the types involved.

Exception Specification Exceptions and std::terminate

It's essential to understand that if a `noexcept` function does throw, the behavior is not a caught exception; it's an immediate call to `std::terminate()`. This is not a bug in the exception handling mechanism; it's the defined behavior for violating a `noexcept` guarantee. This means that `noexcept`

functions cannot be used in contexts where catching exceptions is necessary for recovery.

This behavior is intentional. The `noexcept` specifier is a promise of non-throwing behavior. When that promise is broken, the system effectively gives up, as it cannot safely proceed in a state where an exception was not expected. Therefore, any function marked `noexcept` must be rigorously tested to ensure it lives up to its promise, or any exceptions it might encounter must be caught and handled within the function itself.

When Not to Use `noexcept`

While the benefits of `noexcept` are substantial, it's not a panacea. There are definite scenarios where applying `noexcept` is inappropriate, and doing so can lead to more problems than it solves. The most significant reason not to use `noexcept` is when a function's fundamental purpose involves signaling failure through exceptions. If a function is designed to throw specific exceptions to indicate that an operation could not be completed (e.g., failing to parse invalid input, or an out-of-bounds access that throws `std::out_of_range`), then marking it `noexcept` would be a severe misrepresentation of its behavior and a direct path to `std::terminate()`.

Furthermore, if a function relies on other functions or operations that can throw exceptions, and those exceptions must be propagated to the caller for handling, then the calling function cannot be `noexcept`. This is a chain reaction: if any part of a `noexcept` function can throw and propagate that exception, the `noexcept` guarantee is broken.

Functions Designed to Throw Exceptions

Some functions are explicitly designed to throw exceptions as their primary mechanism for error reporting. Consider a function that parses a configuration file. If the file is malformed, it's natural for this function to throw a custom exception detailing the parsing error. Marking such a function

`noexcept` would be fundamentally incorrect. The exception is the intended outcome in error cases, allowing the caller to gracefully handle the malformed input.

The cppcoreguidelines encourage a clear distinction: if your function's contract involves throwing exceptions to signal errors, do not mark it `noexcept`. Attempting to force exception-throwing functions into a `noexcept` signature will lead to program termination and a breakdown of intended error handling logic.

Dependencies on Throwing Operations

Even if a function itself appears simple, it might internally call other functions or use operations that can throw. If these thrown exceptions are not caught and handled within the `noexcept` function, then the function cannot be `noexcept`. For example, a function that performs a complex calculation might indirectly use a library function that throws on error. If this error needs to be communicated up the call stack, the function containing the call to the throwing library function cannot be `noexcept`.

This is why careful analysis of all function dependencies is crucial when considering the `noexcept` specifier. If any reachable code path within a function can result in an uncaught exception propagating outward, the `noexcept` specifier is not appropriate.

The Cost of `noexcept` When Not Needed

While the performance benefits of `noexcept` are real, it's also important to consider the overhead of checking and ensuring that a function is indeed `noexcept`. If a function is very simple and rarely called, or if its operations are inherently expensive, the marginal gain from `noexcept` might be negligible. In such cases, the focus should remain on correctness and clarity rather than aggressively applying `noexcept` where it doesn't significantly impact performance or safety.

The cppcoreguidelines advocate for thoughtful application of `noexcept`. Don't mark functions `noexcept` just for the sake of it. Understand the implications, analyze the dependencies, and only apply the specifier when it genuinely contributes to the performance, safety, and predictability of your code. Sometimes, the cost of rigorously ensuring `noexcept` compliance might outweigh the benefits for less critical functions.

Practical Application and Best Practices

Putting the `noexcept` specifier into practice effectively requires a strategic approach. The cppcoreguidelines offer a robust framework for making informed decisions. A good starting point is to analyze your codebase and identify functions that are prime candidates for `noexcept`. These often include simple accessors, destructors, move operations, and functions that perform computations without any inherent failure modes that require exceptions.

When in doubt, err on the side of not using `noexcept` and then specifically examine if making it `noexcept` provides a tangible benefit without compromising correctness. This iterative approach, guided by the principles of the cppcoreguidelines, will help you build more robust and performant C++ applications.

Identifying `noexcept` Candidates

Start by looking at your destructors; they are almost always candidates for `noexcept`. Then, examine your move constructors and move assignment operators. If they safely transfer resources without any potential for throwing, mark them `noexcept`. Simple getter functions that only return member values and don't perform any operations that could fail are also excellent candidates. Functions that perform mathematical calculations with well-defined inputs and outputs, or functions that manipulate data structures in ways that are guaranteed not to fail (e.g., internal helper functions), should also be considered.

Use static analysis tools. These tools can often flag potential issues with `noexcept` declarations, such as functions marked `noexcept` that still contain `throw` statements or calls to non-`noexcept` functions that are not properly handled. These tools can be invaluable in identifying violations of the `noexcept` contract.

The Role of Static Analysis Tools

Static analysis tools are your best friend when working with `noexcept`. They can automatically scan your code and flag potential problems, such as functions marked `noexcept` that contain code paths that could lead to an exception. This includes detecting calls to functions that are not marked `noexcept`, or detecting explicit `throw` statements within `noexcept` functions. These tools can significantly reduce the manual effort required to ensure that your `noexcept` declarations are accurate.

Beyond simple detection, some static analyzers can also provide suggestions for how to make functions `noexcept` by identifying specific operations that prevent the `noexcept` guarantee. This proactive approach helps in both identifying existing issues and in designing code with `noexcept` in mind from the outset.

Gradual Adoption and Refactoring

For existing codebases, adopting `noexcept` might not be an overnight process. It's often best approached gradually. Start with the most critical areas: destructors, move operations, and performance-sensitive functions. As you refactor, ensure that any changes made to support `noexcept` (e.g., adding `try-catch` blocks internally or using `noexcept(expression)`) are well-tested. The cppcoreguidelines support this incremental approach, emphasizing that clarity and correctness should not be sacrificed for the sake of aggressively applying `noexcept`.

When refactoring, pay close attention to the exception safety guarantees you are providing. The goal is to make your code more predictable and performant. If a refactoring effort to make a function `noexcept` introduces more complexity or makes the code harder to understand, it might be better to reconsider. The aim is to simplify, not to complicate.

FAQ: cppcoreguidelines for noexcept

Q: What is the primary benefit of using `noexcept` according to cppcoreguidelines?

A: The primary benefits are enhanced performance due to compiler optimizations and improved predictability and reliability of code by providing a strong guarantee that a function will not throw exceptions.

Q: Should destructors always be marked `noexcept`?

A: Yes, generally speaking, destructors should be `noexcept` by default. If a destructor throws an exception, it will cause `std::terminate()` to be called, which is highly undesirable during exception unwinding.

Q: When is it appropriate to not use `noexcept`?

A: You should not use `noexcept` for functions whose core purpose is to signal errors by throwing exceptions, or for functions that call other functions that might throw exceptions and those exceptions need to be propagated to the caller for handling.

Q: How does `noexcept` affect standard library containers?

A: Standard library containers, like `std::vector`, rely on `noexcept` move operations (move constructor and move assignment) to perform reallocations safely and efficiently. If these move operations were not `noexcept`, container operations could leave the container in an inconsistent state.

Q: What happens if a function marked `noexcept` does throw an exception?

A: If a function marked `noexcept` throws an exception, the program will immediately call `std::terminate()`. This is a hard crash, not a caught exception, and is the mechanism for enforcing the `noexcept` contract.

Q: Can `noexcept` be used with templates?

A: Yes, and it's highly recommended for generic code. The conditional `noexcept(expression)` form allows templates to be `noexcept` only when the types they operate on support `noexcept` operations, ensuring safe and optimal behavior.

Q: What is the relationship between `noexcept` and strong exception safety?

A: A function declared `noexcept` inherently provides the strongest exception safety guarantee: the "nothrow" guarantee. This means the operation will succeed without throwing, thus preventing any disruption or state corruption that exceptions might cause.

Q: Are there performance implications when a function is not

`noexcept`?

A: Yes, functions that are not `noexcept` incur some runtime overhead because the compiler must generate code to handle potential stack unwinding and exception dispatching. This overhead is avoided for `noexcept` functions.

Q: How can I check if my code adheres to `noexcept` best practices?

A: Utilize static analysis tools. They can scan your codebase and identify potential violations, such as `noexcept` functions that still contain explicit `throw` statements or call non-`noexcept` functions without proper internal handling.

Q: Is it always beneficial to make move operations `noexcept`?

A: It is generally highly beneficial to make move constructors and move assignment operators `noexcept` if they can safely do so without throwing exceptions. This is crucial for the efficient and safe operation of standard library containers and for predictable resource management.

Cppcoreguidelines For Noexcept

Cppcoreguidelines For Noexcept

Related Articles

- courage and the development of moral fortitude aristotle
- counseling psychology treatment
- counseling psychology approaches

Back to Home