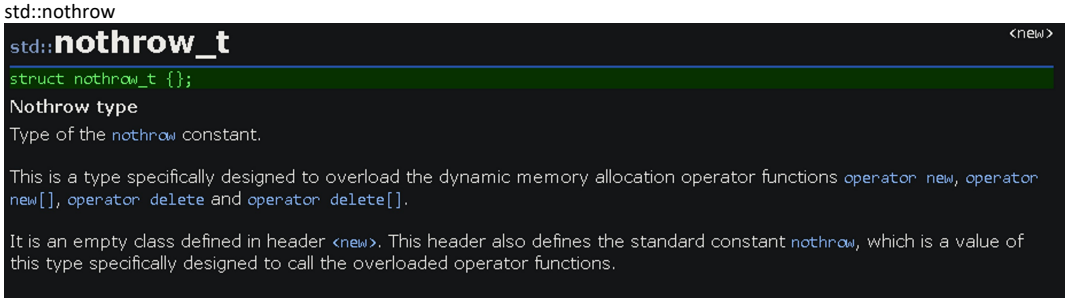
Exceptions

08 June 2021 20:30

std::nothrow



std::nothrow_t <new>

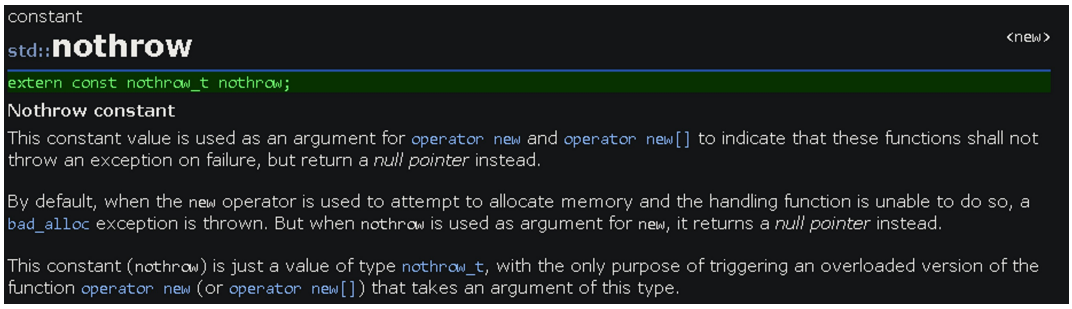
```
struct nothrow_t {};
```

Nothrow type

Type of the nothrow constant.

This is a type specifically designed to overload the dynamic memory allocation operator functions operator new, operator new[], operator delete and operator delete[].

It is an empty class defined in header <new>. This header also defines the standard constant nothrow, which is a value of this type specifically designed to call the overloaded operator functions.



constant

std::nothrow <new>

```
extern const nothrow_t nothrow;
```

Nothrow constant

This constant value is used as an argument for operator new and operator new[] to indicate that these functions shall not throw an exception on failure, but return a *null pointer* instead.

By default, when the new operator is used to attempt to allocate memory and the handling function is unable to do so, a bad_alloc exception is thrown. But when nothrow is used as argument for new, it returns a *null pointer* instead.

This constant (nothrow) is just a value of type nothrow_t, with the only purpose of triggering an overloaded version of the function operator new (or operator new[]) that takes an argument of this type.

Exception handling provides a mechanism to **decouple** handling of errors or other exceptional circumstances from the typical control flow of your code

- Try blocks and catch blocks work together -- A try block detects any exceptions that are thrown by statements within the try block, and routes them to the appropriate catch block for handling.
- A try block must have at least one catch block immediately following it, but may have **multiple catch blocks listed & executed in sequence**.
- Once routed to a catch block for handling, the exception is considered handled, and execution will resume as normal after the catch block.

```
try // Look for exceptions that occur within try block and route to attached catch block(s)
{
    // If the user entered a negative number, this is an error condition
    if (x < 0.0)
        throw "Can not take sqrt of negative number"; // throw exception of type const char*

    // Otherwise, print the answer
    std::cout << "The sqrt of " << x << " is " << sqrt(x) << '\n';
}
catch (const char* exception) // catch exceptions of type const char*
{
    std::cerr << "Error: " << exception << '\n';
}
```

Different data types from throw

```
throw -1; // throw a literal integer value
throw ENUM_INVALID_INDEX; // throw an enum value
throw "Can not take square root of negative number"; // throw a literal C-style (const char*) string
throw dX; // throw a double variable that was previously defined
throw MyException("Fatal Error"); // Throw an object of class MyException
```

```
try{} -- try:
throw -- raise
catch (const Exception& ex) -- except Exception as ex:
```

Procedure:

1. When an exception is raised (using throw), execution of the program immediately **jumps to the nearest enclosing try block** (propagating up the stack if necessary to find an enclosing try block).
2. If any of the catch handlers attached to the try block handle that type of exception, that handler is executed and the exception is considered handled.
3. If no appropriate catch handlers exist, refer stack unwinding.
4. **TYPE of exception**:
   For example, a char exception will not match with an int catch block.
   An int exception will not match a float catch block.
   However, casts from a derived class to one of its parent classes will be performed.

   Note that the compiler will not perform implicit conversions or promotions when matching exceptions with catch blocks!

Catch parameters work just like function parameters

- Exceptions of fundamental types can be caught by value
- Exceptions of non-fundamental types should be caught by const reference to avoid making an unnecessary copy.

Unwinding the stack

1. First, the program looks to see if the exception can be handled immediately (which means it was thrown inside a try block).
2. If not, the current function is terminated, and the program checks to see if the function's caller will handle the exception.
3. If not, it terminates the caller and checks the caller's caller.
4. Each function is terminated in sequence until a handler for the exception is found

Uncaught exceptions - every throw needs a catch

When main() terminates with an unhandled exception, the operating system will generally notify you that an unhandled exception error has occurred.
Possibilities:

1. include printing an error message
2. popping up an error dialog
3. simply crashing.

Catch-all handler

uses the ellipses operator (…) as the type to catch.

```
catch (...) // catch-all handler
{
    std::cout << "We caught an exception of an undetermined type\n";
}
```

One interesting use for the catch-all handler is to wrap the contents of main():

```
#include <iostream>

int main()
{

    try
    {
        runGame();
    }
    catch(...)
    {
        std::cerr << "Abnormal termination\n";
    }

    saveState(); // Save user's game
    return 1;
}
```

To prevent stack unwinding

Often, the catch-all handler block is left empty:

```
catch(...) {} // ignore any unanticipated exceptions
```

This will catch any unanticipated exceptions and prevent them from stack unwinding to the top of your program, but does no specific error handling.

Use of exceptions

Overloaded functions

Unfortunately, because overloaded operators have specific requirements as to the number and type of parameter(s) they can take and return, there is no flexibility for passing back error codes or boolean values to the caller. However, since exceptions do not change the signature of a function, they can be put to great use here

Constructors

Constructor must fail for some reason, simply throw an exception to indicate the object failed to create. In such a case,

- object's construction is aborted, and all class members (which have already been created and initialized prior to the body of the constructor executing) are destructed as per usual.
- However, the class's destructor (class where exception is thrown) is never called (because the object never finished construction).

Exception Class

Exception class is just a normal class that is designed specifically to be thrown as an exception. Create and throw exception class anonymous object and catch by const reference.. And use get member to get error string

```cpp
#include <iostream>
#include <string>

class ArrayException
{
private:
    std::string m_error;

public:
    ArrayException(std::string error)
        : m_error(error)
    {
    }

     const char* getError() const { return m_error.c_str(); }
};

class IntArray
{
private:

    int m_data[3]; // assume array is length 3 for simplicity
public:
    IntArray() {}

    int getLength() const { return 3; }

    int& operator[](const int index)
    {
        if (index < 0 || index >= getLength())
            throw ArrayException("Invalid index");

        return m_data[index];
    }

};

int main()
{
    IntArray array;

    try
    {
        int value{ array[5] };
    }
    catch (const ArrayException &exception)
    {
        std::cerr << "An array exception occurred (" << exception.getError() << ")\n";
    }
}
```

Derived exception classes

```cpp
class Base
{
public:
    Base() []
};

class Derived: public Base
{
public:
    Derived() []
};

int main()
{
    try
    {
        throw Derived();
    }
    catch (const Base &base)
    {
        std::cerr << "caught Base";
    }
    catch (const Derived &derived)
    {
        std::cerr << "caught Derived";
    }

    return 0;
}
```

Catch is executed in the sequential order in which it is listed.. So if

1. If derived is thrown and derived is first->derived catch is executed (Objects of type derived will MATCH the base handler - derived is base)
2. If derived is thrown and base is first->base catch is executed
3. If base is thrown and only base catch is executed (Objects of type Base will NOT match the Derived handler)

So, Handlers for derived exception classes should be listed before those for base classes.

Example of standard exception classes

Eg: operator new can throw std::bad_alloc if it is unable to allocate enough memory. A failed dynamic_cast will throw std::bad_cast.

All of these exception classes are derived from a single class called **std::exception**. std::exception is a small **interface** class designed to serve as a base class to any exception thrown by the C++ standard library.

https://en.cppreference.com/w/cpp/error/exception

https://www.cplusplus.com/reference/exception/exception/

Instead of get method you have a virtual what() .. Using which what of sent exception class is called.

```cpp
#include <cstddef> // for std::size_t
#include <iostream>
#include <exception> // for std::exception
#include <limits>
#include <string> // for this example

int main()
{
    try
    {
        // Your code using standard library goes here
        // We'll trigger one of these exceptions intentionally for the sake of the example
        std::string s;
        s.resize(std::numeric_limits<std::size_t>::max()); // will trigger a std::length_error or allocation exception
    }
    // This handler will catch std::exception and all the derived exceptions too
    catch (const std::exception& exception)
    {
        std::cerr << "Standard exception: " << exception.what() << '\n';
    }

    return 0;
}
```

exception.what() -- Returns a null terminated character sequence that may be used to identify the exception.

Function try blocks

Function try blocks are designed to allow you to establish an exception handler around the body of an entire function, rather than around a block of code.

```cpp
class B : public A
{
public:
    B(int x) try : A{x} // note addition of try keyword here
    {
    }
    catch (...) // note this is at same level of indentation as the function itself
    {
                // Exceptions from member initializer list or constructor body are caught here

                std::cerr << "Exception caught\n";

                // If an exception isn't explicitly thrown here, the current exception will be implicitly rethrown
    }
};
```

Exception specs

Noexcept() operator - **compile time** check to see if expression is declared to not throw any exceptions
https://en.cppreference./com/w/cpp/language/noexcept

noexcept operator (since C++11)

The noexcept operator performs a compile-time check that returns `true` if an expression is declared to not throw any exceptions.

It can be used within a function template's noexcept specifier to declare that the function will throw exceptions for some types but not others.

Syntax

noexcept(*expression* **)**

Returns a prvalue of type `bool`.

noexcept specifier (since C++11)

Specifies whether a function could throw exceptions.

Syntax

noexcept	(1)	
noexcept(*expression***)**	(2)	
throw()	(3)	(since C++17) (deprecated in C++17) (removed in C++20)

1) Same as **noexcept(true)**
2) If *expression* evaluates to `true`, the function is declared not to throw any exceptions. A **(** following **noexcept** is always a part of this form (it can never start an initializer).
3) Same as **noexcept(true)** (see dynamic exception specification for its semantics before C++17)

expression - contextually converted constant expression of type `bool`

noexcept specifier specifies/informs to compiler if a function as non-throwing.
noexcept operator checks in compile-time if a particular expression is declared to throw any exception

Note that a **noexcept** specification on a function is not a compile-time check; it is merely a method for a programmer to inform the compiler whether or not a function should throw exceptions. The compiler can use this information to enable certain optimizations on non-throwing functions as well as enable the noexcept operator, which can check at compile time if a particular expression is declared to throw any exceptions. For example, containers such as `std::vector` will move their elements if the elements' move constructor is **noexcept**, and copy otherwise (unless the copy constructor is not accessible, but a potentially throwing move constructor is, in which case the strong exception guarantee is waived).

https://en.cppreference.com/w/cpp/language/noexcept

In C++, all functions are classified as either **non-throwing** (do not throw exceptions) or **potentially throwing** (may throw an exception).

Exception specifications are a language mechanism that was originally designed to document what kind of exceptions a function might throw as part of a function specification

- noexcept(false) means the function is potentially throwing
- noexcept, meaning the function is non-throwing

```
1 | void doSomething() noexcept; // this function is non-throwing
```

if an exception exits a noexcept function,

- std::terminate will be called.
- If std::terminate is called from inside a noexcept function, stack unwinding may or may not occur (depending on implementation and optimizations), which means your objects may or may not be destructed properly prior to termination.