C++ Exceptions

## Object lifetime

What is an object’s lifetime? Technically speaking, when does an object’s lifetime begin and end. Its lifetime begins when the constructor has successfully completed and returned and its lifetime ends as the destructor begins to run. **[[1]](#footnote-1)** This implies that while a constructor is running the object does not ‘exist’ yet and won’t until it has completed successfully. Similarly, once control reaches the destructor of an object then that object now ceases to exist.

## Exceptions in constructors

Since an object whose constructor has failed never actually existed in the first place then the only means of notifying the caller that this has happened is by throwing an exception. The destructor is not called (because the object doesn’t technically exist yet) and so care and attention must be paid to reversing any allocation and resource acquisition that the constructor has performed up to the point of the exception being thrown.

In a slightly more complex scenario where the object being created has a base class and/or it owns a member object through composition then all objects need to be constructed successfully in order for the initial object to be considered successfully created. A real world equivalence is to say that when a car is created on the factory line it is not truly a car if some of its internal components are corrupted or don’t even exist i.e. no steering wheel or pedals made of jelly. So if any part of the base class or member object fails then the whole object has failed.

## Exceptions and operator new

There are currently three versions of the operator new function, two of which attempt to allocate new memory and the other makes use of existing memory. Only one will throw an exception if the allocation fails whilst the other two have different strategies.

|  |  |
| --- | --- |
| **(1) (throwing)** | void\* operator new (std::size\_t size); |
| **(2) (nothrow)** | void\* operator new (std::size\_t, const std::nothrow\_t& val) noexcept; |
| **(3) (placement)** | void\* operator new (std::size\_t, void\* ptr) noexcept; |

1. The first version of operator new takes the number of bytes required as an argument. If the allocation fails then the std::bad\_alloc exception is thrown. Otherwise a pointer to the start of the new allocated block is returned.
2. The second version also takes the number of bytes required and a special nothrow value (typically the constant std::nothrow) as arguments. In the event of a failure then a null pointer is returned instead of an exception being thrown. Otherwise a pointer to the start of the new allocated block is returned.
3. The third version doesn’t actually allocate any memory at all. It takes a pointer as an argument that is assumed to be pointing to a valid block of memory previously allocated and constructs the new object at that memory location. Since no memory is allocated by this version of operator new then it is safe to assume no exception will be thrown (however, the constructor of the new object might still throw…). Finally the pointer itself is returned.

**Invariants [[2]](#footnote-2)**

*An invariant is some condition assumed to be true about a piece of data at some point in its life. Operations may affect the invariant turning it to false but is then rectified so that it becomes true again. For example, an invariant decided upon may be that a vector should always be sorted and other operations are written to assume that is the case as a precondition. By calling push\_back(n) on the vector, it is quite likely that the vector is not strictly sorted anymore and therefore the invariant is broken. So a quick call to sort(v.begin(), v.end()) will sort the vector and satisfy the invariant that the vector is sorted.*

## Exception safety guarantees

There are three levels of exception safety guarantee (formulated by David Abrahams). These are (from weakest to strongest): **[[3]](#footnote-3)**

* The basic guarantee
* The strong guarantee
* The No-Throw guarantee

There is a fourth alternative which is to provide no exception safety guarantee whatsoever but this is bad practice, makes the code less robust and other users will have little faith in the code. In practice it is best to provide at least the basic guarantee for operations and the strong guarantee when it makes sense to do so.

The strong guarantee and no-throw guarantee can often be attainable but impractical due to an expensive “all-around-the-houses” route involving copying which hits the performance of the code.

## Basic guarantee

The basic guarantee means that no resource or memory leaks will occur and that all invariants are maintained (*see side panel*). Any stored data will remain valid but may have been modified from its state before the exception was thrown.

## Strong guarantee

The strong guarantee provides the basic guarantee and in addition states that in the event of an exception being thrown there will be no side effects and all data will retain their initial values from before the exception was thrown. This can be implemented by using rollback semantics. Basically, the state is preserved and is as if the operation that threw was never called in the first place. The operation either succeeds or fails. If it fails the state is exactly as it was before the operation started.

An example is where an operation is called on a vector to insert a new element somewhere in the middle of the existing elements. Traditionally the elements after the insertion point are moved up one place one at a time (assuming no reallocation is necessary) and the new element is dropped into the newly vacated slot. The copy constructor could throw on any one of those elements as they are being copied to the new location. If it did and wasn’t handled internally then there could possibly be a memory leak (both the new storage and existing storage exist but only a reference to the existing storage exists). Also the vector is in an invalid state as some of the elements have shifted up but others haven’t.

The strong guarantee says that even if this situation occurs the internal implementation will fix it and roll back so that the new storage is deallocated and the vector’s elements are still valid and in the same locations as they were before the insert operation was called. It would be as if the insert operation was never called in the first place.

## No-Throw guarantee

The no-throw guarantee states that all operations will succeed definitely and an exception will not be thrown in any circumstance. Internally the operation may throw exceptions but those would be caught and handled there and so not exposed to the caller.

An example of a no-throw guarantee is the classic swap idiom that swaps two pointers. There is nothing that can go wrong to cause an exception to be thrown so is guaranteed to succeed.

## Techniques

Various techniques exist to assist in handling exceptions in common operations in an attempt to not leak memory and preserve data. Consider the following example:

Example 1

**void** **processNewObject**() {

Object \* p = **new** Object;

p->process();

**delete** p;

}

In this simple example if the allocation of Object fails then there is no dynamic memory to clean up and no data has been modified. But what if the call to process() on the second line causes an exception to be thrown? The last statement that deletes the pointer is never reached and the result is a memory leak.

A simple way to get around this problem is an idiom called Resource Acquisition Is Initialization.

## Resource Acquisition Is Initialization (RAII)

This is a rather simple technique but very powerful. Instead of assigning new allocated memory to a raw pointer and hoping that the delete statement is reached the pointer is handed to a special object that will manage the lifetime of the pointer instead.

Example 2

**void** **processNewObject**() {

UniquePointer<Object> p(**new** Object);

p->process();

}

A smart pointer such as UniquePointer is just one solution. Any object that takes ownership of a pointer can be used in RAII. Upon creation it is handed a raw pointer to the heap allocated Object. The smart pointer is now responsible for that pointer. Since the smart pointer is stack allocated it is guaranteed to be deleted once it goes out of scope.

If all goes well and no exception is thrown then the smart pointer deletes the raw pointer when it goes out of scope (at the end of the function body). However if an exception is thrown when process() is called, through stack unwinding, the smart pointer is still destroyed and deallocates the raw pointer with it. In either case the handling of the allocated memory is accounted for and no leak will occur.

The idiom gets it name from the idea that some resource is acquired and is typically used to initialize a resource-managing object all in one statement:

UniquePointer<Object> p(**new** Object);

## Handling constructor exceptions

Most constructors are tasked with initializing its members but remembering that any one of those operations can result in an exception being thrown it is important for the constructor to roll back any initialization that has occurred pre-exception. A simple example to demonstrate:

Example 3

**class** StringArray {

**private**:

String \* p;

**public**:

**StringArray**(**int** size, **const** String& s) {

p = **new** String[size];

**fill**(p, p+size, s);

}

};

At first glance this seemingly innocuous code doesn’t look too troubling. It is an overloaded constructor that takes as arguments the size of the string array and a string to initialize each element with. But it provides no exception safety and, in the worst case where an exception is thrown, is a leak waiting to happen.

If the first line of the constructor fails during the allocation of the string array then we have no problems. The allocation failed so we have nothing to clean up and therefore the constructor failed meaning that the StringArray object never existed.

The second line that calls the **fill()** algorithm, however, could be problematic. It loops through the array and places a copy of **s** in each location. So the copy constructer of String is called size times and any one of those could potentially fail.

Since our StringArray constructor has no exception handling the exception is allowed to leave the constructor and control is handed back to the caller of the constructor but notice that the array allocation has now leaked. Remember, the constructor has failed so the object never truly existed which means the destructor is not called which is where the cleanup code would have been.

So the constructor has to roll back the pre-exception initialization. This is done with a try-catch block.

Example 4

**class** StringArray {

**private**:

String \* p;

**public**:

**StringArray**(**int** size, **const** String& s) {

**try** {

p = **new** String[size];

**fill**(p, p+size, s);

}

**catch**(...) {

**delete** [] p;

}

}

};

The same two lines from the first constructor now live inside the **try** block. If either of these operations fail and an exception is thrown the exception will be caught by the **catch** block and gives us the ability to clean up. Here we simply delete the array.

**Handling exceptions in operations that modify data**

Imagine a scenario where you have a vector that contains ten strings and you want to erase the last three strings. A naive implementation of that function may look like this:

Example 5

**void**

**Vector<string>::**

**erase**(**int** from, **int** num) {

**for** (**int** i=from; i<from+num; i++) {

eraseElement(i);

--numElements;

}

}

|  |  |
| --- | --- |
| index |  |
| 0 | String |
| 1 | String |
| 2 | String |
| 3 | String |
| 4 | String |
| 5 | String |
| 6 | String |
| 7 | … |
| 8 | … |
| 9 | String |

The example features a simple loop that starts at the index of the first element to be erased and continues for num iterations. The eraseElement() call inside the **for** loop represents the application logic that erases an element. There are many ways in which this could be implemented depending on project requirements. Then numElements is decremented to reflect that there is now one less element in the vector.

But assume that eraseElement() throws on the third and final iteration of the loop. What are we left with? A vector with holes in it! More specifically, a vector containing strings from index 0 to index 6, nothing at index 7 and 8 and the last string at index 9 (it’s still there as the erase operation failed).

If the vector was handed back to the caller of **erase()** in this state it would be pretty difficult, if not impossible, to use in any way that makes sense. Not even the basic guarantee is provided here. There are no leaks but some class invariants are not maintained such as the size of the vector. What result would a call to vector.size() return, for instance? And how would it be meaningful in any way?

So how can it be improved? Well, if the decision to provide the basic guarantee is chosen then the change is minimal:

Example 6 – basic guarantee

**void**

**Vector<string>::**

**erase**(**int** from, **int** num) {

**try** {

**for** (**int** i=from+num-1; i>=from; i--) {

eraseElement(i);

--numElements;

}

}

**catch**(...) {

**throw**;

}

}

In example 6’s version of **erase()** the **for** loop is placed inside a **try**-**catch** block and the elements are erased in reverse order (from the last element to be erased to the first element to be erased). Same as before, with each successful erased element numElements is decremented. The significant difference between example 5 and example 6 is that if a call to eraseElement() fails in example 6 then all invariants, such as the size, are maintained.

The **erase()** operation overall has failed since it only erased some of what we asked it to but the vector is still in a valid condition, can recover and can continued to be used.

By re-throwing the caught exception in the **catch** block we give the caller the information that something went wrong which gives them the opportunity to try and recover.

## Guidelines

* Every function and method should provide at least the basic safety guarantee and, where it makes sense, the strong and possibly no-throw guarantee.
* Never ever ever let a destructor throw an exception. **Bad Things Will Happen! ™** It’s fine, though, for a destructor to throw an exception that is also immediately caught and handled within the destructor’s body. Just don’t let the exception leave the destructor.
* Throw exceptions by value and catch by reference. From the compiler’s point of view there is nothing wrong in throwing and catching a pointer but it is difficult to work with. Who is responsible for the pointer? Should the caller delete the pointer to the exception after handling it or does the library handle it for you? Due to this ambiguity it is best practice to throw by value and catch by reference making it a non-issue.

## The critical line

1. www.gotw.ca/gotw/066.htm [↑](#footnote-ref-1)
2. Andrew Koenig - http://www.drdobbs.com/cpp/a-simple-invariant/240009670 [↑](#footnote-ref-2)
3. https://en.wikipedia.org/wiki/Exception\_safety [↑](#footnote-ref-3)