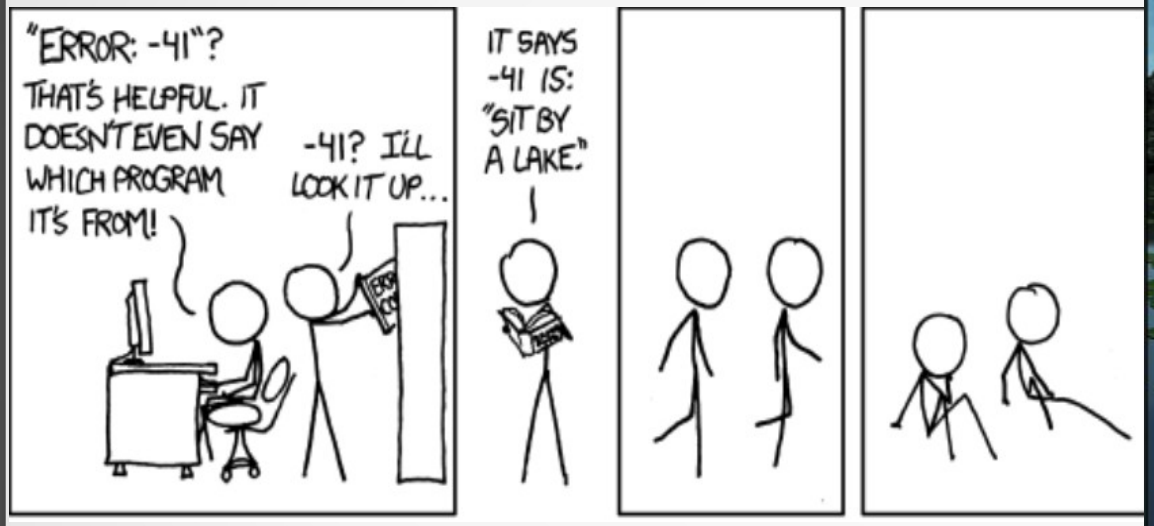
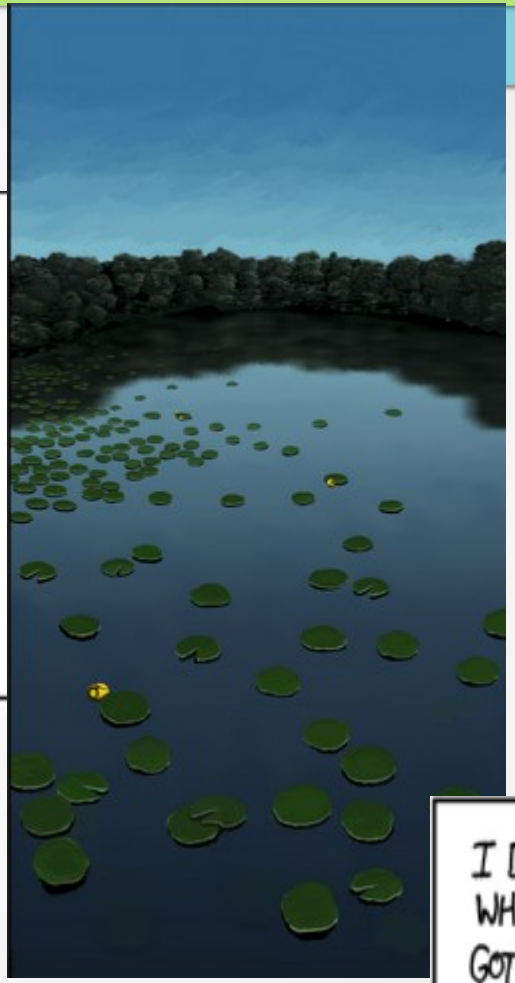
C++ Exception Handling (Part II)





http://xkcd.com/1024/

Krishna Kumar

Exceptions: They are not errors!

"some part of the system couldn't do what it was asked to do"

- Catching Exception:
- void f() {

```
    try {
        throw E {};
    }
    catch(H) {
        // when do we get here?
    }
}
```

Catching Multiple Exceptions

```
try {
 // Code to Try
}
catch(arg_1) {
  // One Exception
}
catch(arg_2) {
  // Another Exception
}
```

Object lifetimes

```
// Example code
{
    Parrot& perch = Parrot();
}
```

- When does an object's lifetime begin?
  - When its constructor completes successfully and returns normally. That is, when control reaches the end of the constructor body or completes an earlier *return* statement
- When does an object's lifetime end?
  - When its destructor begins. That is, when control reaches the beginning of the destructor body

Constructor exception

{ Parrot& perch = Parrot(); }

- The state of the object before its lifetime begins is exactly the same as after its lifetime ends: There is no object.
- What does emitting an exception from a constructor mean?
  - It means that construction has failed, the object never existed, its lifetime never began. Indeed, the only way to report the failure of construction—namely, the inability to correctly build a functioning object of the given type—is to throw an exception. Incidentally, this is why a destructor will never be called if the constructor didn't succeed—there's nothing to destroy

What exactly happens when a constructor emits an exception?

```
class C : private A
{
    B b_;
};
```

In the **C** constructor, how can you catch an exception thrown from the constructor of a base subobject (such as **A**) or a member object (such as **b_**)?

Function try blocks

```
C::C()
try
    : A ( /*...*/ ), b_( /*...*/ ) {}
catch( ... )
{
// We get here if either A::A() or B::B() throws.
}
```

- If A::A() succeeds and then B::B() throws, the language guarantees that A::~A() will be called to destroy the already-created A base subobject before control reaches this catch block
- **Is this needed at all? In C++, if construction of any base or member subobject fails, the whole object's construction must fail.**

Morals about function try blocks

- **Moral #1:** Constructor function try block handlers are only good for translating an exception thrown from a base or member subobject constructor. They are not useful for any other purpose.
- **Moral #2:** Destructor function try blocks have little or no practical use. Destructor should NEVER emit an exception.
- **Moral #3:** All other function try blocks have no practical use.
- **Moral #4:** Always perform unmanaged resource acquisition in the constructor body, never in initializer lists, i.e., Resource Allocation Is Initialisation (RAII).

Exception-safe function calls

// Example 1(a)

f(expr1, expr2);

// Example 1(b)

f(**g**(expr1), **h**(expr2));

- What is the order of evaluation of the functions f, g, and h and the expressions expr1 and expr2? Assume that expr1 and expr2 do not contain more function calls.

Function call – rules:

- **All of a function's arguments must be fully evaluated before the function is called.** This includes the completion of any side effects of expressions used as function arguments.
- Once the execution of a function begins**, no expressions from the calling function may begin or continue to be evaluated until execution of the called function has completed.** Function executions never interleave with each other.
- **Expressions used as function arguments may generally be evaluated in any order,** including interleaved, except as otherwise restricted by the other rules.

Function call – rules:

f(expr1, expr2);

- We can say is that both **expr1 and expr2 must be evaluated before f is called.** The compiler may choose to perform the evaluation of expr1 before, after, or interleaved with the evaluation of expr2.

f(**g**(expr1), **h**(expr2));

- expr1 must be evaluated before **g** is called.
- expr2 must be evaluated before **h** is called.
- Both **g** and **h** must complete before **f** is called.
- The evaluations of expr1 and expr2 may be interleaved with each other, but nothing may be interleaved with any of the function calls.
- For example, no part of the evaluation of expr2 or any of the execution of h may occur from the time g begins until it ends; however, it’s possible for h to be called either before or after g.

Function call & exception safety

```
// Example 2


// In some header file:

void f( T1*, T2* );


// At some call site:

f( new T1, new T2 );
```

- What does New-expression does?
  - it allocates memory;
  - it constructs a new object in that memory; and
  - if the construction fails because of an exception the allocated memory is freed.
- new-expression is essentially a series of two function calls: one call to operator new, and then a call to the constructor.

Exception safety problems (I)

- What happens if the compiler decides to generate code that performs the following steps in order:
  1. allocate memory for the T1
  2. construct the T1
  3. allocate memory for the T2
  4. construct the T2
  5. call f
- If either step 3 or step 4 fails because of an exception, the C++ standard does not require that the T1 object be destroyed and its memory deallocated.
- This is a classic memory leak, and clearly Not a Good Thing.

Exception safety problem (II)

- Another possible sequence of events is the following:
  1. allocate memory for the T1
  2. allocate memory for the T2
  3. construct the T1
  4. construct the T2
  5. call f

- If step 3 fails because of an exception, then the memory allocated for the T1 object is automatically deallocated (step 1 is undone), but the standard does not require that the memory allocated for the T2 object be deallocated. The memory is leaked.
- If step 4 fails because of an exception, then the T1 object has been allocated and fully constructed, but the standard does not require that it be destroyed and its memory deallocated. The T1 object is leaked.

Help from unique_ptr?

```
// Example 3


// In some header file:

void f( std::unique_ptr<T1>,
std::unique_ptr<T2> );


// At some call site:

f( std::unique_ptr<T1>{ new
T1 }, std::unique_ptr<T2>{ new
T2 } );
```

- This code attempts to “throw unique_ptr at the problem.”
- **Nothing has changed**. Example 3 is still **not exception-safe,** for exactly the same reasons as before.
- The resources are **safe only if they really make it into a managing unique_ptr**, but the same problems already noted can still occur before either unique_ptr constructor is ever reached.

Exception safety (unique_ptr)

1. allocate memory for the T1
2. construct the T1
3. allocate memory for the T2
4. construct the T2
5. construct the unique_ptr<T1>
6. construct the unique_ptr<T2>
7. call f

1. allocate memory for the T1
2. allocate memory for the T2
3. construct the T1
4. construct the T2
5. construct the unique_ptr<T1>
6. construct the unique_ptr<T2>
7. call f

Again, the same problems are present if either of steps 3 or 4 throws.

Fortunately, though, this is not a problem with unique_ptr; it's just being used the wrong way, that's all. Let's see how to use it better.

Enter make_unique (C++14)

```
// Example 4


// In some header file:
void f( std::unique_ptr<T1>, std::unique_ptr<T2> );


// At some call site:
f( make_unique<T1>(), make_unique<T2>() );
```

Basic idea of make_unique

- **Functions called from the same thread won't interleave**, so we want to provide a function that does the work of allocation and construction of the object and construction of the unique_ptr.
- Because the function should be able to work with any type, we want to express it as a **function template.**
- Because the caller will want to pass constructor parameters from outside make_unique, we'll use the **C++11 perfect forwarding style** to pass those along to the new-expression inside make_unique.
- Because shared_ptr already has an analogous std::make_shared, for consistency we'll call this one **make_unique**.