

C++ Exceptions

Jim Fawcett

CSE687-OnLine – Object Oriented Design

Summer 2017

How to Deal with Exceptional States?

- Ignore them:
 - Wrong thing to do for all but demo programs.
- Abort processing – detect but don't try to recover:
 - Not appropriate for programs with safety issues or critical missions.
- Have functions return error codes:
 - Program is constantly spending CPU cycles looking for rare events.
 - Easy to miss a check.
- Use C++ Exceptions

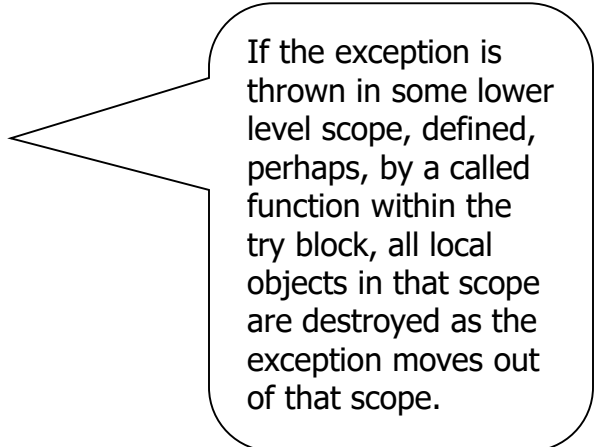
C++ Exceptions

- Exceptions use three keywords:
throw, try, and catch
 - `throw e`:
constructs an exception object, `e`, and takes it out of an enclosing context defined by a `try` block
 - `try {...}`:
defines, for thrown exceptions, an enclosing context with specified catch handlers
 - `catch(E e) {...}`:
exception handler `catch(E e)` responds to an exception object of type “E”

Example

If no exception is thrown, the code in the try block is executed, the catch clause is skipped, and computation resumes after the catch clause.

```
try
{
    // some code that may throw an exception
}
Catch(exception &e)
{
    // some processing to attempt to recover from error
    // based on information carried by the exception
}
```



If the exception is thrown in some lower level scope, defined, perhaps, by a called function within the try block, all local objects in that scope are destroyed as the exception moves out of that scope.

If an exception is thrown somewhere in the try block, the remaining code in the try block is skipped, and a matching catch clause is entered, if found. Computation resumes after the last statement in matching catch clause. Matching is based on the type of the exception.

Chained Handlers

- Exception handlers are often chained at the end of a try block, e.g.:

```
try {  
    // some code that may throw an exception  
}  
Catch(T1 t1) {  
    // processing for type T1  
}  
Catch(T2 t2) {  
    // processing for type T2  
}
```

- Matching attempts are based on the order of declaration of the handlers.

Cleaning Up

- C++ exception handling guarantees that as an exception leaves a scope all objects in that scope that have been successfully constructed will have their destructors called.
- Only destructors are called, so resources that are allocated in that scope but not deallocated by destructors will be lost.
- So, in an exception environment, you must make all allocations within objects and deallocate in their destructors.
- If a second exception is thrown while a first has not yet been handled, the special function `terminate()` is called (more on that later).

Matching

- A catch handler matches a thrown exception based on its type.
 - If you throw a literal string, say: “big trouble in River City” then it can be caught with the catch handler: `catch(char *msg) { ... }`.
 - An exception handler that accepts a reference to a base class object will match a derived class object, as well as the base type specified.
 - If a derived class object is passed to a handler by value it will be sliced to a base class object.
 - If, however, a derived object is passed by reference, no slicing occurs, and polymorphic calls within the handler are honored.
 - A catch handler with an ellipsis, `catch(...) { ... }`, will catch any exception thrown in its context, not caught earlier.

Uncaught Exceptions

- If none of the catch handlers for a try block matches a thrown exception the exception moves to the next enclosing try block.
- If there is no match in any enclosing try block the exception is uncaught. An uncaught exception also occurs if a new exception is thrown before an existing one is handled. Cleanups may fail to occur with an uncaught exception, so this is an error.
- If an exception is uncaught the special function `terminate()` is called.
- Uncaught exceptions can always be avoided by enclosing the contents of `main` in a try block with an ellipsis handler.

terminate() Function

- Terminate is a function pointer with default value the C library function abort().
- You can define your own terminate handler using
 `set_terminate(void(*)());`
- Example:

```
void Arnold() { std::cout << "I'll be back" }
int main() {
    set_terminate(Arnold);
    :
}
```

Rethrowing Exceptions

- If your catch handler does not completely handle an exception you may re-throw it to the next enclosing context.

```
catch(E e)
{
    // processing to handle e is incomplete
    throw;
}
```

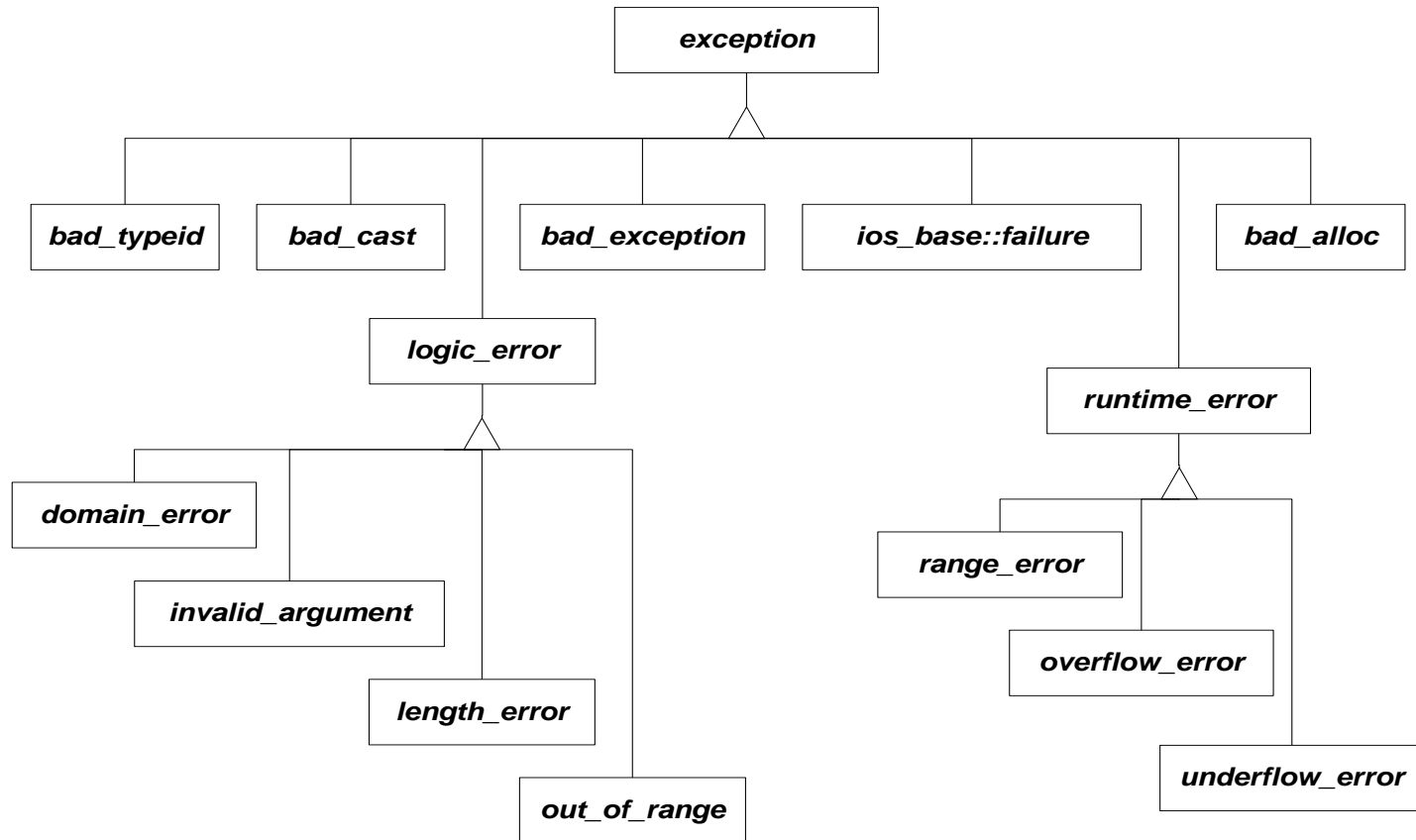
- This allows processing an exception in several passes as it travels up through a series of try-contexts.

Standard Exceptions

- Standard Exception class:

```
namespace std {  
    class exception {  
        public:  
  
        virtual const char* what() const throw();  
  
        // create, copy, assign, and destroy  
        // exception objects  
    };  
}
```

Standard Exceptions



Exception Specifications

- All exception specifications have been removed from C++11 except for `throw()` and `nothrow()`.
- A function can declare exception specifications:
 - `void f() throw (E1, E2, E3);`
declares that `f` may throw any of `E1`, `E2`, or `E3`.
 - `void f() throw()`
declares that no exceptions are thrown in `f`.
 - `void f()`
declares that any type exception may be thrown in `f`.

Specification Violations

- If an exception specification is violated, the special function `unexpected()` is called when the exception is thrown.
- By default `unexpected()` terminates execution. However, you may change that behavior by defining your own:

```
void FreddyKrueger() { ... }
int main()
{
    set_unexpected(FreddyKrueger);
    :
}
```

Exception Safety (Sutter, 2000)

- Basic guarantee:
 - In the presence of exceptions thrown by called global functions, object messages, template parameters, or library calls, the code:
 - will not leak resources.
 - will remain in a consistent, if unpredictable, state.

Exception Safety

- Strong guarantee:
 - If an operation terminates because of an exception, program state will remain unchanged.
 - This implies commit-or-rollback semantics, including that no references or iterators will become invalid if an operation fails.

Exception Safety

- Nothrow guarantee:
 - A function will not emit an exception under any circumstances.
 - Strong exception safety isn't possible unless certain functions are guaranteed not to throw.

Exception Safety

- To implement strong exception safety:
 - In each function, take all the code that might emit an exception and do all its work safely off-to-the-side.
 - Only when you know that work has succeeded should you modify program state, by swapping current state with the off-to-the-side state, using only non-throwing operations like pointer swaps.
 - Destructors must always provide the nothrow guarantee, since destructors are called in the scope of an exception and a second active exception will always immediately call `terminate()` without further cleanup.

References

- The C++ Programming Language, 4th Edition, Stroustrup, Addison-Wesley
- Exceptional C++, Sutter, Addison- Wesley, 2000
- There is a very nice summary in our text:
The C++ Standard Library, 2nd Edition, Nicolai Josuttis, Addison Wesley