Exception Handling

RMIT UNIVERSITY

Objectives

- Explain the basic concepts for Exception Handling in C++
- Implement Exception Handling in C++

Exceptions

- An exception is anything **unexpected**, or **unusual**, about a program's execution. It is a situation that is **not normal**.
- An exception could be an error condition, or caused by unexpected input from the user.
- For example, a user could enter a negative value for their age, or a file could not exist.

Exceptions

- **Exception handling** is the process of dealing with an exceptional circumstance.
- Exception handling code must decide how to either return the program’s execution to normality, or terminate the program.
- An exception handler **catches** an exception. Exceptions are **thrown**.

Throwing an Exception

- In C++, anything can be thrown, such as a string or an int, or a specially written class.
- This is different to Java, where an object must either inherit from Exception, or implement Throwable.
- To throw an exception, simply use the `throw` keyword:

```
if (!connected) {
   throw NotConnectedException();
}
```

Catching Exceptions

- Exceptions are caught using a `try..catch` block.
- A section of code is **tried**, and any exceptions that are thrown are **caught** by catch blocks.

```
try {
   connectToDatabase()
}
catch(NotConnectedException& e) {
   cerr << “Not connected!\n”;
}
```

Catch-all blocks

- A catch-all block is used when you wish to catch any other exception.
- Catch-all blocks should be the last in a sequence of catch blocks, as catch blocks located after one will never be reached.
- It is specified as follows:

```
try {
   //
}
catch(...) {
   // catch all
}
```

Uncaught exceptions

- Exceptions are rethrown automatically if no catch block exists that will handle them.
- Alternatively, if you want to handle an exception, but then rethrow it, you can use the `throw` keyword without a parameter.

```
catch(...) {
    throw; // rethrow
}
```

terminate

- If no part of the program handles an exception, it will eventually be thrown by `main`.
- When `main` throws the exception, `std::terminate()` is called.
- This by default calls the `abort()` function, which stops the program and sends an abort signal to the operating system.

Throw lists

- A **throw list** is a list of exceptions that a function may throw. Throw lists are sometimes called **exception specifications**.
- Throw lists are specified after a function declaration. This provides a guarantee that the function will only throw exceptions of that type.

```
void doSomething() throw(int, string);
```

Empty throw lists

- If a throw list is empty, the function is guaranteeing that it will, under no circumstances, throw an exception.
- This is a powerful specification – it means that the function must handle all exceptions it generates itself.

unexpected

- If a function attempts to throw an exception that is not specified in its throw list, the program will call `std::unexpected()`.
- By default, `std::unexpected()` will in turn call `std::terminate()`, and the program will abort.

Exceptions

- Exceptions are useful for error handling across independent modules.
- These modules might not be developed together, but can use a standard way of indicating an error.
- Exceptions that are thrown by a module or library become part of the API for that library.

Exceptions

- More robust than returning an error code.
- Error codes are usually integers – this limits the usefulness. In C++, we could throw a class that provides meaningful debug information.
- What if the function needs to return an integer – how would it return an error if all integer values were legal?

Exceptions

- Out of interest, who knows what `printf` returns, and who has ever checked the return value from `printf`?
- I know I don't, and never have. Error codes may be ignored.
- Trying to ignore an exception will usually result in program termination. Much more robust!

Exceptions

- Exceptions are cleaner than printing an error to `stderr`.
- External libraries shouldn’t assume that `stderr` is going to be read by a developer.
- It may be that `stderr` *can’t* be read! Don’t assume you have a screen or terminal to write to.

Standard Exceptions

- C++ defines a number of standard exceptions.
- These all inherit from `std::exception`, although it is unclear why.
- All are located in the `std` namespace.
- The most common one you might encounter is `std::bad_alloc`, thrown when `new` cannot allocate memory.

Standard Exceptions

- Others that may occur:
  - `bad_cast`: when a `dynamic_cast` fails.
  - `out_of_range`: when a range-checked access to a vector (via `at()` member function) accesses invalid location.
  - `ios_base::failure`: When attempting to `clear()` an `iostream`'s error state, and this fails.

nothrow new

- For those times when you don't want `new` to throw an exception, a special form of `new` exists. This is called **nothrow new.**
- In this instance, 0 is returned if an error occurs, and no exception is thrown.

```
int* tmp = new(nothrow) int[10];

Game* g = new(nothrow) Game();
```

Exceptions in constructors

- Exceptions are often used to indicate an error in a constructor.
- This was their main driving force – how do you return an error from a function that has no return type?
- It is important to realise that an object is not constructed until the constructor is finished.
- Objects that throw exceptions during construction are not valid!

Exceptions in Constructors

```
int main() {
    try {
        A* aPtr = new A();
    }
    catch(...) {
        cerr << "Error constructing "
            << aPtr->getName(); << endl;
    }
}
```

Exceptions in destructors

- Likewise, destructors that throw exceptions are a problem.
- The object may be partly destroyed (especially if an exception occurs in a base class), and therefore this leaves the object in an undefined state.
- Destructors must never throw! Assume all destructors have `throw()` as their throw list.

Exceptions in Destructors

- We’ve looked at a few standard exceptions – in particular, `std::bad_alloc` is thrown when new cannot allocate any memory.
- But, if new cannot allocate memory, how can `std::bad_alloc` (an object that must be created) be thrown?
- Answer: Every C++ implementation *must* leave sufficient memory to throw `std::bad_alloc`

Exceptions in destructors

- So, if `std::bad_alloc` has been thrown, and caught, the program must attempt to free up memory.
- If a destructor throws an exception, is there enough memory to throw it?
- Answer: **probably not**. At this point, the operating system is likely to start going crazy, and everything is undefined.

Exceptions in destructors

- The moral of the story is to not have destructors that could throw. This is a Bad Thing™
- Related to this is `operator delete` and `operator delete[]`.
- If you have defined your own versions of these, they should also be specified to not throw.

Exceptions in delete

- In a similar manner to exceptions in destructors, the first thing that is likely to be called after a `std::bad_alloc` exception is thrown is `delete`.
- If this can throw, it may not have enough memory to allocate an exception and Bad Things will probably occur.

Exception Safety

- The proper behaviour of a program when an exception is thrown is called **exception safety**.
- Exception safety concerns what state the program is left in (can it be recovered?) and whether resources are freed and not leaked.

Basic Exception Safety

- The Basic Guarantee, as defined by Dave Abrahams, is that if an exception is thrown, objects do not leak resources.
- Note that program can have basic exception safety, but still be left in an undefined state, making recovery impossible.

Strong Exception Safety

- The strong guarantee is that if an exception is thrown, the program remains unchanged.
- This is similar to database transactions – if they fail, nothing happened.
- Often, by implementing the basic guarantee, we have strong exception safety anyway.

Nothrow guarantee

- In the Nothrow guarantee, an operation or a function will not throw an exception.
- This is required to ensure we can meet strong and basic guarantees.
- If some functions are allowed to throw anything, we have a real problem – we cannot ensure some operations will occur without exceptions being thrown.

Writing Exception Safe Code

- A key point when thinking about using exceptions is that exception safety is and must be a part of your design.
- It’s a surprisingly common theme to see exceptions tacked on to code as an afterthought.

Writing Exception Safe Code

- The basic principle of writing exception safe code is to do anything that could throw an exception first.
- By doing this, we can then proceed knowing that all throwing operations have succeeded
- We can then change the object or program’s state using non-throwing functions.

Writing Exception Safe Code

- Exception safety is a lot easier to achieve if an object or a function has only one responsibility.
- Attempting to do two or more things in a single function makes it harder to perform throwing operations first, before modifying program state.

Writing Exception Safe Code

- The finest example of writing exception safe code can be found in Herb Sutter’s “Exceptional C++”, pages 25-60.
- He takes a non-exception safe Stack object, and converts it to an exception safe version.
- Along the way, he shows many elegant solutions to problems that can be achieved if functions follow the guarantees.