18 - Exception Handling

December 1, 2022

COMP2404

Darryl Hill

Contents

No Program is Perfect

Your program can fail or act unexpectedly in many ways:

- Every program of reasonable size has bugs.
  - Bug - an implementation error.
- It is very difficult to anticipate every combination of inputs.
- Your code might interact with faulty hardware, other faulty code.
- etc.

Code ***Robustness***: Ability of your program to handle **faults** or **exceptions**.

Fault:

- Defect, bug, something causing your program to crash.

Exception:

- An event outside of the expected.

Dealing with Faults

Types of fault:

- Bad input
  - Wrong data type
  - Unexpected format
  - Network failure
- Software bugs
  - Array out of bounds
  - Segmentation faults
  - Null pointers, dereferencing bad pointers

Stage 1: Fault Prevention

Write software in a way that minimizes faults.

- Follow good OO design
  - Easy to isolate and test a class with a well-defined, simple purpose
  - Difficult to test an entire program with multiple sub-purposes
- Code reviews
  - Your brain can trick you into not seeing your mistakes
  - Other people can see them
- Testing
  - Unit-testing
  - Integration testing
  - System testing
- You should unit test new code, then integrate it and test your program as a whole, then test it in real world

Stage 2: Fault Detection

Aim: discover faults before the user does.

- Testing
  - Part of the software development life cycle.
  - Usually a planned (systematic) approach.
- Debugging
  - Systematic approach has failed, time to use the debugger / print statements
  - Not planned
- Beta-testing
  - People using your code is a good way to discover faults.
  - When a fault is discovered use debugging to find it.

Testing

Test success paths:

- Assume things go right
  - Data is sanitized (error checked beforehand)
  - Users use your software properly

Test failure paths:

- Assume things go wrong
  - Bad input, null pointers
  - Real users

Debugging

Crashing?

- Use print statements or debugger to find ***exactly*** where it is crashing.

Bad data?

- Print out the variable every step of the way
- Find ***exactly*** where it is getting corrupted
  - Often different from where crash occurs

Debugging is not guess-work

Finding ***exactly*** where the problem is occurring is half the battle

Sometimes works, sometimes crashes?

- Are you accessing bad memory? Did you initialize your variables?
  - Sometimes in C++ bad memory will still work ¯_(ツ)_/¯

Once you know ***exactly*** where the problem is, if you cannot figure it out, ask someone else.

Code Robustness

Code Robustness: the ability of a program to tolerate faults and keep running in a useful way.

Keeping our code robust:

- Error checking:
  - Assertions or contracts.
  - Inline error checking.
  - Goal is to detect a potential error state before it becomes part of your program.
- Exception handling:
  - When an unexpected (possibly faulty) state is reached.
  - Handled using an alternate flow of control.
    - Typically this alternate flow is programmatically expensive.
  - We may try and rejoin normal control flow or simply crash gracefully.

Errors

Difference between errors and exceptions:

- Errors ways that we expect things could go wrong.
  - Someone tries to enter a value out of range.
  - Someone makes a spelling mistake on their username.

- Simple, anticipated problems
- Handled during regular program flow using inline error handling.

- Exceptions are more rare and unexpected:
  - File system crashed
  - A system call did not work
- Handled outside of regular flow by an alternate control flow.

Error Handling

Inline error handling.

- Intermixing of program and error-handling logic.

```
do thing
if thing doesn't work
    process error
else
    proceed as normal
```

Good for simple error checking.

- Too much or too complex and code is difficult to read.
- Basic data sanitation is ok.

Exception Handling

Exception Handling

- Used to resolve errors that are
  - less frequent
  - harder to check
  - harder to predict

Possible courses of action:

- Allow execution to continue.
- Notify the user there is a problem and continue.
  - We may not be sure of the state.
    - Did that file save or load properly?
  - An inconsistent state may cause problems later.
- Terminate the program and exit in a controlled manner

Exception Handling

Exception handling (EH) is elegant is some ways, because it separates

- Error reporting - finding the problem
- Error handling - handling the problem

These things can occur in different parts of the program.

- EH provides an alternate return structure.
- Normal program stack return method is bypassed.

These things come at a price.

- Slow and expensive compared to inline error checking.
- Alternate control flow can can cause important code to be skipped.

Exception Handling

C++ library has an `exception` class:

- Used as a base class of user-defined `exception` classes.
- Constructor takes a `string message` argument
- Member function `what()` returns that message
- `exception` is "thrown" when there is an error.

We don't have to use `exception`

- We can throw anything, but
- Users of your class will likely try and catch an `exception` rather than your custom class.

Exception Handling Example


```
void func(){
  float x, num, den
  //initialize num and den
  try{
    if (den==0){
      throw "Divided by 0";
    }
    x = num / den;
  }
  catch (char* error){
    cout<<error;
  }
}
```

Error checking

Error reporting

Error handling

if we `throw` something the code execution exits the `try` block

Code after is not executed:
`x = num / den;`
is skipped

Enters the `catch` block

coding examples <p1> and <p2>

Coding Examples p1 and p2 Notes

- Stack of function calls with a `try` block at the top
- `cin.good()` tells if there was an input error
- We can `throw` whatever we like
  - not just `exceptions`
- Observe that we can throw local objects.
  - The temporary object is an `lvalue` that is copied into the catch parameter, however
  - Sometimes the compiler optimizes the two copies into a single object.
    - You might notice there is no call to a copy or move constructor.
- Processing continues after the `catch` block.

try/throw/catch

`try`:

- Block of potentially dangerous code
- If something goes wrong here, we would like to handle it
- What goes wrong may not be in the try block, but perhaps within a function within the try block

try/throw/catch

`throw`:

- We've detected some inconsistencies that we were not prepared for, so we ”pass the buck” by `throwing` an exception
- May be in a `try` block, or within a function call within a `try` block, or within a function within a function within...
- If we throw something, and there is no `try/catch` block somewhere up the call stack, the program will terminate.

try/throw/catch

`catch`:

- Block that deals with the problem (by catching the exception)
- Try and figure out what went wrong and handle it gracefully.
- This block must immediately follow the `try` block
  - think `if-else` blocks

Throw in Called Function Example


```
void func(){
float x, num, den;
//initialize num and den
try{
    x = divide(num, den);
} catch (char* error){
    cout<<error;}
}
```

```
float divide(float a, float b){
    if (b == 0) throw "Divided by 0";
    return a/b;
}
```

We `throw` within a function within the `try` block.

try/throw/catch

- When an exception is thrown, control is transferred to the `catch` block
  - If the `catch` block is down the stack, then all stack frames between the `throw` and `catch` are popped off.
- Separation of error reporting and error handling is good software engineering.
- You might write Classes used by others
  - You don't know the circumstances when the error occurs.
  - There might be no console to print to!
  - Your code might be a library for some other app
  - Whoever made the app should decide how to handle errors.

Throw in Called Function Example


```
float divide(float a, float b){
    if (b == 0) throw "Divided by 0";
        return a/b;
}
```

This is the call stack. This stack frame is popped.

```
float middle(float a, float b){
    return divides(a,b);
}
```

No `try` block here, so this stack frame is popped.

```
void func(){
    float x, num, den;
    //initialize num and den
   try{
       x = middle(num, den);
   } catch (char* error){
       cout<<error;}
}
```

Control is returned to this function, within the `catch` block.

Coding Examples p3 and p4, Notes

coding examples <p3> and <p4>

- Note how control is transferred, and how this affects (potentially) return values
- Putting the `try/catch` block inside or outside of a `for`-loop can affect behaviour
  - In one case the for-loop terminates
  - In the other the for-loop continues
- When making `try/catch` blocks should take note of how the control flow is interrupted and possible repercussions.

Multiple catch blocks

A `throw` statement has one parameter

- Compiler looks for the `catch` block with a matching parameter
- If we want to catch everything, then use `catch(...)`
  - the `throw` parameter cannot be used here
  - compiler does not know the types of the arguments

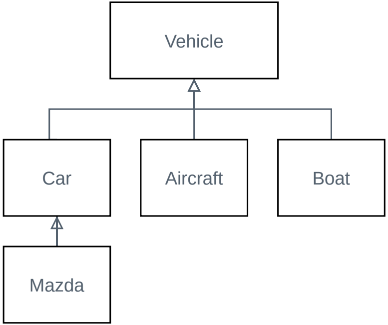
coding example <p5>

Example p5 Notes:

- We see examples of different catch blocks.
- The order of the catch blocks matter when using inheritance.

Multiple Catch Blocks - Order Matters


```
void func(){
  try{
    if (/*condition*/)
      throw Car();
  }
  catch (Vehicle& v){ }
  catch (Aircraft& a){ }
  catch (Boat& b){ }
  catch (Mazda& m){ }
  catch (Car& c){ }
  catch (...) { }
}
```



The order of `catch` blocks matters!

- derived class
- base class
- catch all

Re-Thrown Exceptions


```
float divide(float a, float b){
  if (b == 0) throw "Divided by 0";
  return a/b;
}
```

Throw the initial exception.

```
float middle(float a, float b){
  try{ return divides(a,b); }
  catch (char*){
    cout<<"Caught! Throw again!";
    throw; // no parameter ok here} }
```

We need to do some cleanup here, throw again. We may throw the same exception or a different one.

```
void func(){
  try{
    x = middle(num, den);
  } catch (char* error){
    cout<<error;}
}
```

Catch the re-thrown exception.

coding example <p6>

Exception Specifications

In C++, throw declarations are optional

```
int func(int x) throw (int, exception) { <body> }
```

- May throw `int` or `exception`

```
int func(int x) { <body> }
```

- May throw anything

```
int func( int x ) throw(){ <body> }
```

- May NOT throw anything

Throws that are not caught call `terminate()`

Stack Unwinding

Stack unwinding is when an exception is thrown and the call stack is popped down to the catch block

- throw and catch bypass normal return structure
- May cause problems
  - local variables are destroyed
  - return values never returned
  - memory not deallocated
- Can cause an inconsistent state
  - `cout<<"This code should never execute";`
  - (I hate when this code executes)

Graceful Stack Unwinding

Put all pointers in the catch parameter

- Cleanup occurs in the catch block
- Violates OO design
  - Are you sure someone will clean it for you?

Put a catch block in every called function that might potentially need it

- Each function does its own cleanup
- Good encapsulation

Graceful Stack Unwinding

Make everything an object.

- Destructors are invoked on scope exit
  - Even when exiting via a thrown exception
- Cleanup is automatic

coding example <p7>

Coding Example p7 Notes

- 1st example - memory to be deallocated is placed in the Error object
- 2nd - destructor deletes the array
- 3rd - array is an object which destroys itself

Using Catch Parameter


```
class Error_message{
    public:
        Error_message(char* str, int* p):
            message(str), arrayPtr(p){}
        char* message;
        int* arrayPtr;
}
```

```
void g(){ try { f();}
    catch (Error_message& m){
        delete [ ] m.arrayPtr;
        cout<<m.message;
    }
}
```

```
void f(){
    int* a = new int[10];
    if (/*condition*/)
        throw Error_message("error",a);
}
```

Better: Unwinding with Re-throw


```
void f(){
    int* a = new int[10];
    try { d(); }
    catch (char*){
        delete [ ] a;
        throw;
    }
}
```

```
void g(){ try { f();}
    catch (char* error){
        cout<<error;
    }
}
```

```
void d(){
    if (/*condition*/)
        throw "error";
}
```

Best: Unwinding with Destructors


```
class MyArray{
public:
    MyArray(int size){arr = new int[size];}
    ~MyArray(){delete [ ] arr;}
private:
    int* arr;
}
```

```
void g(){ try { f();}
    catch (char* error){
        cout<<error;
    }
}
```

```
void f(){
    MyArray a(10);
    d();
}
```

```
void d(){
    if (/*condition*/)
        throw "error";
}
```