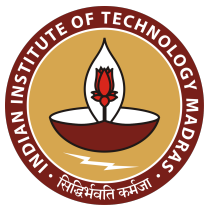
Errors, Exceptions and Testing

Saurav Samantaray

Department of Mathematics

Indian Institute of Technology Madras

November 6, 2024



Assertion

- Assertion (`assert`) is a way of forcing the program to terminate execution, should something unexpected happen.
- For example, to calculate the square root of a number entered at the command-line

```
#include <iostream>
#include <cassert>
#include <cmath>
using namespace std;

int main()
{
   double a;
   cout << "Enter a non−negative number\n ";
   cin >> a;
   //Run without assertion: assert(a >= 0.0);
   cout << " The square root of  "<< a;
   cout << " is " << sqrt(a) << "\n";
   return 0;
}
```

Assertion

- What happens when a user ignores the request and enters a negative number at the command line?
- Without the assert statement it is likely that the program will complete without error.
- This is because the computer's floating point unit renders the result of some calculations such as `sqrt(-1.0)` as "not a number" or `nan` for short.
- Other examples of floating point operations which produce the answer `nan` include $0.0/0.0$ and $\mathbf{log}(0.0)$.
- Some calculations such as $1.0/0.0$ will resolve to a floating point representation of infinity (`inf`).
- In a program, once one variable has been set to `nan` or `inf` then this value is likely to propagate to later parts of the calculation.
- It is normally best to check for this sort of error at the earliest possible stage so that computation is not wasted.

Preconditions

- Every section of a program (where a "section" could be a function, method, block, for-loop iteration body etc.) can be thought of as having the task to produce a *postcondition* when given a valid *precondition*.
- For example, the postcondition of the previous program (the thing which it is tasked to do) is that it prints the square root of a given number.
- It does this subject to the precondition that the number is nonnegative.
- Consider a method which finds all the roots of a function $f(x)$ in the half-open range $x_{min} \leq x \leq x_{max}$.
- This method might need to assume as a precondition that the function f is continuous and differentiable over the same range $x_{min} \leq x \leq x_{max}$.
- More trivially, it might also need to assume that $x_{min} < x_{max}$.

Preconditions

- What should happen if $x_{min} > x_{max}$ or $x_{min} = x_{max}$?
- If the precondition for correct functionality is not met then what should happen?
- Some of the most important decisions that a programmer has to make are about how errors should be treated.
- What should happen if the user misreads a prompt and enters some invalid input?
- What should happen if the application writer accidentally permutes the input arguments of a library function?
- What should happen if some numerical scheme has generated `inf` or `nan`?
- The answer to all these questions is the same: "It depends".
- It's good to treat errors differently depending on their severity, both in terms of how likely they are to happen and in terms of how easy it might be to fix the problem and carry on.

Handling errors

We propose a strategy for handling errors which is built on a framework of three levels of errors.

1. If the error can be fixed safely, then *fix* it. If need be, *warn* the user.
2. If the error could be caused by some reasonable user input then throw an *exception* up to the calling code, since the calling code should have enough context to fix the problem.
3. If the error should not happen under normal circumstances then trip an *assertion*.

- These three basic levels could be further refined.
- One may distinguish between errors that trip assertions (which are normally removed in optimised code) and errors that should halt the program under all circumstances.
- At the other end of the scale, you might distinguish between error fixes which are silent and,
- those which should warn the user that something has been changed.

Handling errors

- The *exception* level of error is a compromise between patching the problem to carry on, and stopping completely.
- It is used in circumstances where the caller of a function may have enough information to be able to deal with the error.
- For example, a non-linear Newton root finder may diverge and hence signal an error,
- but the programmer may know that the original task in question can still be solved by calling the same function with a different initial guess,
- or by calling it with a damping factor, or by calling a bisection root finder.
- The logic would be to first try the Newton solver,
- but if that function signalled an error then to find the root using a more expensive bisection routine.

Introducing the Exception

- An exception in C++ is a way of interrupting the normal flow of control of a program and throwing a bundle of information back to the calling code.
- This bundle of information is encapsulated inside an object.
- We define in this section a class called `Exception`, but objects of any class may be thrown between functions to signal an error.

The use of exceptions requires the keywords `try`, `throw` and `catch`

`try` is used in the calling code and tells the program to execute some statements in the knowledge that some error might happen.

`throw` is used when the error is identified. The function called will encapsulate information about the error into an `Exception` object and throw it back to the caller.

`catch` is used in the calling code to show how to attempt to fix the error. Every block of code that has the `try` keyword must be matched by a `catch` block.

Implementing Exception Safety via try and catch

- `try` and `catch` are the most important keywords in C++ as far as implementing exception safety goes.
- To make statements exception safe, we enclose them within a `try` block and handle the exceptions that emerge out of the `try` block in the `catch` block:

```
void SomeFunc()
{
   try
   {
      int* numPtr = new int;
      *numPtr = 999;
      delete numPtr;
   }
   catch(...)  // ... catches all exceptions
   {
      cout << "Exception in SomeFunc(), quitting" << endl;
   }
}
```

Using catch(...) to Handle All Exceptions

- See ”mem_all.cpp”
- Using `try` and `catch` in Ensuring Exception Safety in Memory Allocations
- ”mem_all.cpp” demonstrates the usage of `try` and `catch` blocks.
- `catch()` takes parameters, just like a function does, and `...` means that this `catch` block accepts all kinds of exceptions.
- In this case, however, we might want to specifically isolate exceptions of type `std::bad_alloc` as these are thrown when `new` fails.
- Catching a specific type will help us handle that type of problem in particular, for instance, show the user a message telling what exactly went wrong.

Results

- For this example, we used -1 as the number of integers that we wanted to reserve.
- This input is ridiculous, but users do ridiculous things all the time.
- In the absence of the exception handler, the program would encounter a very ugly end.
- But thanks to the exception handler, you see that the output displays a decent message: `Got to end, sorry!`

Catching Exception of a Type

- The exception in ”mem_all.cpp” was thrown from the C++ Standard Library.
- Such exceptions are of a known type, and
- catching a particular type is better for us, as we can pinpoint the reason for the exception,
- do better cleanup, or at least show a precise message to the user.
- See ”mem_all_1.cpp”.
- Compare the output of ”mem_all.cpp” and ”mem_all_1.cpp”.
- We see that we are now able to supply a more precise reason for the abrupt ending of the application, namely, “bad array new length.
- This is because we have an additional catch block (yes, two catch blocks), one that traps exceptions of the type catch(bad_alloc &), which is thrown by `new`.

Throwing Exception of a Type Using `throw`

- When we caught `std::bad_alloc`, we actually caught an object of class `std::bad_alloc` thrown by `new`.
- It is possible for us to throw an exception of our own choosing.
- All we need is the keyword `throw`

```
void DoSomething()
{
   if(something_unwanted)
   throw object;
}
```

Exception

```
try {
  int age = 15;
  if (age >= 18) {
    cout << "Access granted – you are old enough.";
  } else {
    throw (age);
  }
}
catch (int myNum) {
  cout << "Access denied – You must be at least 18 years old.\n";
  cout << "Age is: " << myNum;
}
```

- We use the try block to test some code: If the age variable is less than 18, we will throw an exception, and handle it in our catch block.
- In the catch block, we catch the error and do something about it. The catch statement takes a parameter: in our example we use an int variable `(myNum)` (because we are throwing an exception of `int` type in the try block `(age)`, to output the value of `age`.
- If no error occurs (e.g. if age is 20 instead of 15, meaning it will be be greater than 18), the catch block is skipped:

Throwing a Custom Exception at an Attempt to Divide by Zero

- See ”div_zero.cpp”.
- The code not only demonstrates that we are also able to catch exceptions of type `char*`.
- but also that you caught an exception thrown in a called function `Divide()`
- Also note that we did not include all of `main()` within `try {}`,
- we only include the part of it that we expect to `throw`.
- This is generally a good practice, as exception handling can also reduce the execution performance of our code.

How Exception Handling Work

- In "div_zero.cpp", we threw an exception of type `char*` in function `Divide()` that was caught in the `catch(char*)` handler in calling function `main()`.
- Where an exception is thrown, using `throw`, the compiler inserts a dynamic lookup for a compatible `catch(Type)` that can handle this exception.
- The exception handling logic first checks if the line throwing the exception is within a `try` block.
- If so, it seeks the `catch(Type)` that can handle the exception of this Type.
- If the throw statement is not within a `try` block or if there is no compatible `catch()` for the exception type, the exception handling logic seeks the same in the calling function.
- So, the exception handling logic climbs the stack, one calling function after another, seeking a suitable `catch(Type)` that can handle the exception.
- At each step in the stack unwinding procedure, the variables local to that function are destroyed in reverse sequence of their construction.

Class std::exception

- In catching `std::bad_alloc`, we actually caught an object of class `std::bad_alloc` thrown by `new`.
- `std::bad_alloc` is a class that inherits from C++ standard class `std::exception`, declared in header < exception >.
- `std::exception` is the base class for the following important exceptions
  - `bad_alloc`—Thrown when a request for memory using new fails.
  - `bad_cast`—Thrown by `dynamic_cast` when you try to cast a wrong type (a type that has no inheritance relation)
  - `ios_base::failure`—Thrown by the functions and methods in the `iostream` library
- Class `std::exception` that is the base class supports a very useful and important virtual method `what()` that gives a more descriptive reason on the nature of the problem causing the exception.

Exception Class

- When an error occurs we want the code to “throw” two pieces of information: a one-word summary of the problem type and a more lengthy description of the error.
- We write a class Exception (shown below) to store these two pieces of information, and with the ability to print this information when required.