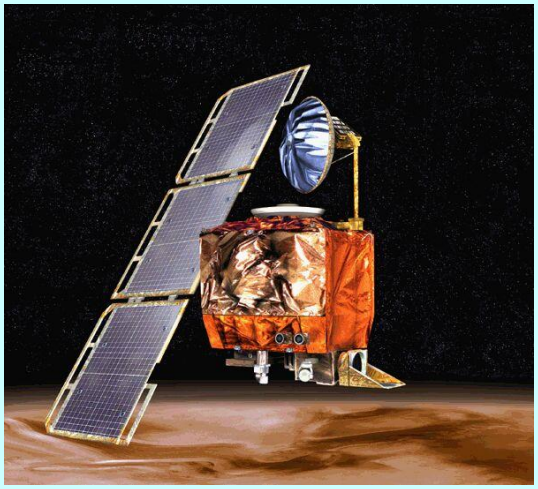
Chapter 5
Errors



Bjarne Stroustrup and from Internet

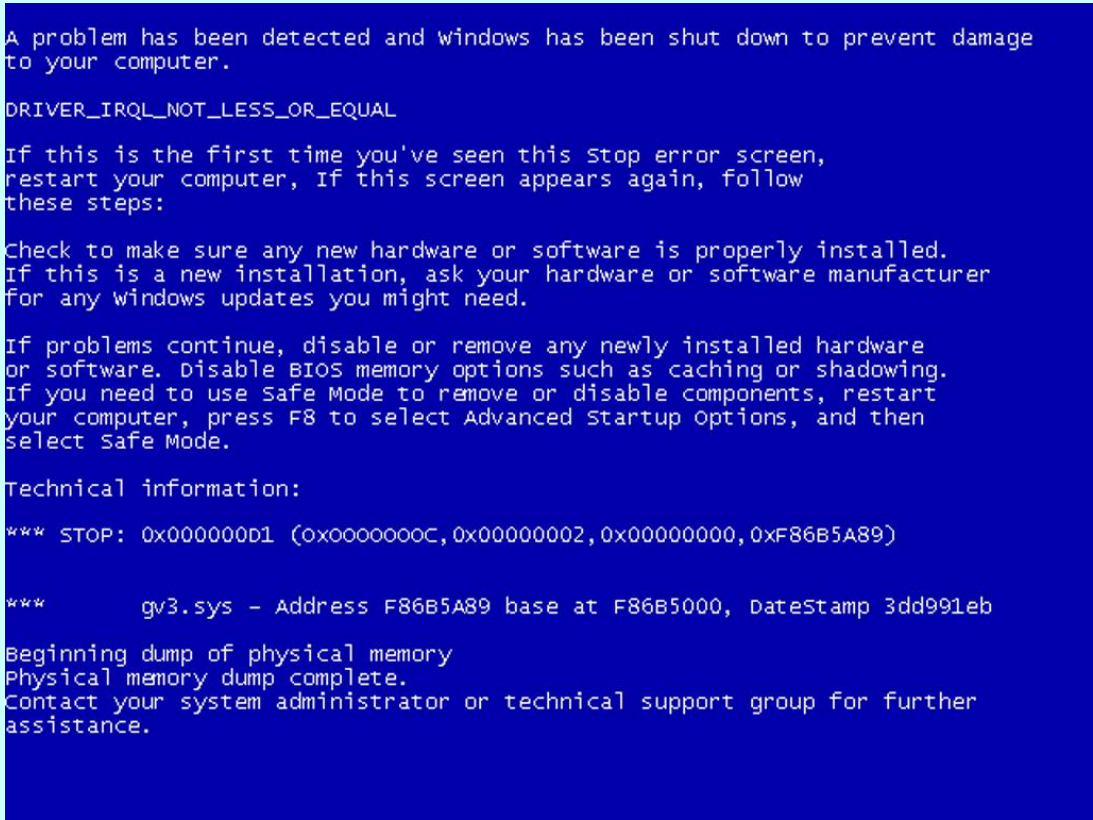
www.stroustrup.com/Programming

Abstract

- When we program, we have to deal with errors. Our most basic aim is correctness, but we must deal with incomplete problem specifications, incomplete programs, and our own errors. Here, we'll concentrate on a key area: how to deal with unexpected function arguments. We'll also discuss techniques for finding errors in programs: debugging and testing.

Overview

- Kinds of errors
- Argument checking
  - Error reporting
  - Error detection
  - Exceptions
- Debugging
- Testing



Errors

- " … I realized that from now on a large part of my life would be spent finding and correcting my own mistakes."
  - Maurice Wilkes, 1949
- When we write programs, errors are natural and unavoidable; the question is, how do we deal with them?
  - Organize software to minimize errors.
  - Eliminate most of the errors we made anyway.
    - Debugging
    - Testing
  - Make sure the remaining errors are not serious.
- My guess is that avoiding, finding, and correcting errors is 95% or more of the effort for serious software development.
  - You can do much better for small programs.
    - or worse, if you're sloppy

Your Program

1. Should produce the desired results for all legal inputs
2. Should give reasonable error messages for illegal inputs
3. Need not worry about misbehaving hardware
4. Need not worry about misbehaving system software
5. Is allowed to terminate after finding an error

3, 4, and 5 are true for beginner's code; often, we have to worry about those in real software.

Sources of errors

- Poor specification
  - "What's this supposed to do?"
- Incomplete programs
  - "but I'll not get around to doing that until tomorrow"
- Unexpected arguments
  - "but **sqrt()** isn't supposed to be called with **-1** as its argument"
- Unexpected input
  - "but the user was supposed to input an integer"
- Code that simply doesn't do what it was supposed to do
  - "so fix it!"

Kinds of Errors

- Compile-time errors
  - Syntax errors
  - Type errors
- Link-time errors
- Run-time errors
  - Detected by computer (crash)
  - Detected by library (exceptions)
  - Detected by user code
- Logic errors
  - Detected by programmer (code runs, but produces incorrect output)

Check your inputs

- Before trying to use an input value, check that it meets your expectations/requirements
  - Function arguments
  - Data from input (istream)

Bad function arguments

- The compiler helps:
  - Number and types of arguments must match

```
int area(int length, int width)
{
    return length*width;
}

int x1 = area(7);                 // error: wrong number of arguments
int x2 = area("seven", 2);        // error: 1st argument has a wrong type
int x3 = area(7, 10);             // ok
int x5 = area(7.5, 10);           // ok, but dangerous: 7.5 truncated to 7;
                                  //       most compilers will warn you
int x = area(10, -7);             // this is a difficult case:
                                  //        the types are correct,
                                  // but the values make no sense
```

Bad Function Arguments

- So, how about `int x = area(10, -7);` ?
- Alternatives
  - Just don't do that
    - Rarely a satisfactory answer
  - The caller should check
    - Hard to do systematically
  - The function should check
    - Return an "error value" (not general, problematic)
    - Set an error status indicator (not general, problematic – don't do this)
    - Throw an exception
- Note: sometimes we can't change a function that handles errors in a way we do not like
  - Someone else wrote it and we can't or don't want to change their code

Bad function arguments

- Why worry?
  - You want your programs to be correct
  - Typically the writer of a function has no control over how it is called
    - Writing "do it this way" in the manual (or in comments) is no solution – many people don't read manuals
  - The beginning of a function is often a good place to check
    - Before the computation gets complicated
- When to worry?
  - If it doesn't make sense to test every function, test some

How to report an error

- Return an "error value" (not general, problematic)

```
int area(int length, int width)    // return a negative value for bad input
{
   if(length <=0 || width <= 0) return -1;
   return length*width;
}
```

- So, "let the caller beware"

```
int z = area(x,y);
if (z<0) error("bad area computation");
// …
```

- Problems
  - What if I forget to check that return value?
  - For some functions there isn't a "bad value" to return (e.g., max())

How to report an error

- Set an error status indicator (not general, problematic, don't!)

```
int errno = 0;    // used to indicate errors
int area(int length, int width)
{
   if (length<=0 || width<=0) errno = 7;  // // means or
   return length*width;
}
```

- So, "let the caller check"

```
int z = area(x,y);
if (errno==7) error("bad area computation");
// ...
```

- Problems
  - What if I forget to check **errno**?
  - How do I pick a value for **errno** that's different from all others?
  - How do I deal with that error?

How to report an error

- Report an error by throwing an exception

```
class Bad_area { };  // a class is a user defined type
                     // Bad_area is a type to be used as an exception

int area(int length, int width)
{
   if (length<=0 || width<=0) throw Bad_area{};   // note the {} – a value
   return length*width;
}
```

- Catch and deal with the error (e.g., in **main()**)

```
try {
   int z = area(x,y);        //  if area() doesn't throw an exception
}                            // make the assignment and proceed
catch(Bad_area) {            // if area() throws Bad_area{}, respond
   cerr << "oops! Bad area calculation – fix program\n";
}
```

Exceptions

- Exception handling is general
  - You can't forget about an exception: the program will terminate if someone doesn't handle it (using a **try … catch**)
  - Just about every kind of error can be reported using exceptions
- You still have to figure out what to do about an exception (every exception thrown in your program)
  - Error handling is **never** really simple

Out of range

- Try this

```
vector<int> v(10);    // a vector of 10 ints,
                      // each initialized to the default value, 0,
                      // referred to as v[0] .. v[9]
for (int i = 0; i<v.size(); ++i) v[i] = i;  // set values
for (int i = 0; i<=10; ++i)                 // print 10 values (???)
    cout << "v[" << i << "] == " << v[i] << endl;
```

- vector's operator[ ] (subscript operator) reports a bad index (its argument) by throwing a Range_error if you use #include "std_lib_facilities.h"
  - The default behavior can differ
  - You can't make this mistake with a range-for

Exceptions – for now

- For now, just use exceptions to terminate programs gracefully, like this

```
int main()
try
{
    // …
}
catch (out_of_range&) {   // out_of_range exceptions
    cerr << "oops – some vector index out of range\n";
}
catch (…) {                    // all other exceptions
    cerr << "oops – some exception\n";
}
```

A function error()

- Here is a simple error() function as provided in std_lib_facilities.h
- This allows you to print an error message by calling error()
- It works by disguising throws, like this:

```
void error(string s)        // one error string
{
        throw runtime_error(s);
}

void error(string s1,  string s2)    // two error strings
{
        error(s1 + s2);     // concatenates
}
```

Using **error( )**

- Example

```
cout << "please enter integer in range [1..10]\n";
int x = -1;      // initialize with unacceptable value (if possible)
cin >> x;
if (!cin)        // check that cin read an integer
      error("didn't get a value");
if (x < 1 || 10 < x)       // check if value is out of range
      error("x is out of range");
// if we get this far, we can use x with confidence
```

Exception safe

- **none**: Your code should never offer that. This code will leak everything, and break down at the very first exception thrown.
- **basic**: This is the guarantee you must at the very least offer, that is, if an exception is thrown, no resources are leaked, and all objects are still whole
- **strong**: The processing will either succeed, or throw an exception, but if it throws, then the data will be in the same state as if the processing had not started at all (this gives a transactional power to C++)
- **nothrow/nofail**: The processing will succeed.

Example of code

```
void doSomething(T & t)
{
   if(std::numeric_limits<int>::max() > t.integer)  // 1.   nothrow/nofail
      t.integer += 1 ;                              // 1'.  nothrow/nofail
   X * x = new X() ;                // 2. basic : can throw with new and X constructor
   t.list.push_back(x) ;            // 3. strong : can throw
   x->doSomethingThatCanThrow() ;   // 4. basic : can throw
}
```

https://stackoverflow.com/questions/1853243/do-you-really-write-exception-safe-code

Basic

```
void doSomething(T & t)
{
   if(std::numeric_limits<int>::max() > t.integer)  // 1.   nothrow/nofail
      t.integer += 1 ;                              // 1'.  nothrow/nofail
   std::auto_ptr<X> x(new X()) ;    // 2.  basic : can throw with new and X constructor
   X * px = x.get() ;               // 2'. nothrow/nofail
   t.list.push_back(px) ;           // 3.  strong : can throw
   x.release() ;                    // 3'. nothrow/nofail
   px->doSomethingThatCanThrow() ;  // 4.  basic : can throw
}
```

Strong: costly

```
void doSomething(T & t)
{
   // we create "x"
   std::auto_ptr<X> x(new X()) ;    // 1. basic : can throw with new and X constructor
   X * px = x.get() ;               // 2. nothrow/nofail
   px->doSomethingThatCanThrow() ;  // 3. basic : can throw

   // we copy the original container to avoid changing it
   T t2(t) ;                        // 4. strong : can throw with T copy-constructor

   // we put "x" in the copied container
   t2.list.push_back(px) ;          // 5. strong : can throw
   x.release() ;                    // 6. nothrow/nofail
   if(std::numeric_limits<int>::max() > t2.integer)  // 7.   nothrow/nofail
      t2.integer += 1 ;                              // 7'.  nothrow/nofail

   // we swap both containers
   t.swap(t2) ;                     // 8. nothrow/nofail
}
```

Google style

Exceptions

We do not use C++ exceptions.

Pros:

- Exceptions allow higher levels of an application to decide how to handle "can't happen" failures in deeply nested functions, without the obscuring and error-prone bookkeeping of error codes.
- Exceptions are used by most other modern languages. Using them in C++ would make it more consistent with Python, Java, and the C++ that others are familiar with.
- Some third-party C++ libraries use exceptions, and turning them off internally makes it harder to integrate with those libraries.
- Exceptions are the only way for a constructor to fail. We can simulate this with a factory function or an `Init()` method, but these require heap allocation or a new "invalid" state, respectively.
- Exceptions are really handy in testing frameworks.

Cons:

- When you add a `throw` statement to an existing function, you must examine all of its transitive callers. Either they must make at least the basic exception safety guarantee, or they must never catch the exception and be happy with the program terminating as a result. For instance, if `f()` calls `g()` calls `h()`, and `h` throws an exception that `f` catches, `g` has to be careful or it may not clean up properly.
- More generally, exceptions make the control flow of programs difficult to evaluate by looking at code: functions may return in places you don't expect. This causes maintainability and debugging difficulties. You can minimize this cost via some rules on how and where exceptions can be used, but at the cost of more that a developer needs to know and understand.
- Exception safety requires both RAII and different coding practices. Lots of supporting machinery is needed to make writing correct exception-safe code easy. Further, to avoid requiring readers to understand the entire call graph, exception-safe code must isolate logic that writes to persistent state into a "commit" phase. This will have both benefits and costs (perhaps where you're forced to obfuscate code to isolate the commit). Allowing exceptions would force us to always pay those costs even when they're not worth it.
- Turning on exceptions adds data to each binary produced, increasing compile time (probably slightly) and possibly increasing address space pressure.
- The availability of exceptions may encourage developers to throw them when they are not appropriate or recover from them when it's not safe to do so. For example, invalid user input should not cause exceptions to be thrown. We would need to make the style guide even longer to document these restrictions!

https://google.github.io/styleguide/cppguide.html#Exceptions

Google style

noexcept

Specify `noexcept` when it is useful and correct.

Definition:

The `noexcept` specifier is used to specify whether a function will throw exceptions or not. If an exception escapes from a function marked `noexcept`, the program crashes via `std::terminate`.

The `noexcept` operator performs a compile-time check that returns true if an expression is declared to not throw any exceptions.

Pros:

- Specifying move constructors as `noexcept` improves performance in some cases, e.g. `std::vector<T>::resize()` moves rather than copies the objects if T's move constructor is `noexcept`.
- Specifying `noexcept` on a function can trigger compiler optimizations in environments where exceptions are enabled, e.g. compiler does not have to generate extra code for stack-unwinding, if it knows that no exceptions can be thrown due to a `noexcept` specifier.

Cons:

- In projects following this guide that have exceptions disabled it is hard to ensure that `noexcept` specifiers are correct, and hard to define what correctness even means.
- It's hard, if not impossible, to undo `noexcept` because it eliminates a guarantee that callers may be relying on, in ways that are hard to detect.

How to look for errors

- When you have written (drafted?) a program, it'll have errors (commonly called "bugs")
  - It'll do something, but not what you expected
  - How do you find out what it actually does?
  - How do you correct it?
  - This process is usually called "debugging"

Debugging

- How ***not*** to do it

```
while (program doesn't appear to work) { // pseudo code
    Randomly look at the program for something that "looks odd"
    Change it to "look better"
}
```

- Key question

How would I know if the program actually worked correctly?

Program structure

- Make the program easy to read so that you have a chance of spotting the bugs
  - Comment
    - Explain design ideas
  - Use meaningful names
  - Indent
    - Use a consistent layout
    - Your IDE tries to help (but it can't do everything)
      - You are the one responsible
  - Break code into small functions
    - Try to avoid functions longer than a page
  - Avoid complicated code sequences
    - Try to avoid nested loops, nested if-statements, etc.
      (But, obviously, you sometimes need those)
  - Use library facilities

First get the program to compile

- Is every string literal terminated?

```
cout << "Hello, << name << '\n';          // oops!
```

- Is every character literal terminated?

```
cout << "Hello, "  << name << '\n;         // oops!
```

- Is every block terminated?

```
if (a>0) { /* do something */
else { /* do something else */ }  // oops!
```

- Is every set of parentheses matched?

```
if (a                    // oops!
    x = f(y);
```

- The compiler generally reports this kind of error "late"
  - It doesn't know you didn't mean to close "it" later

First get the program to compile

- Is every name declared?
  - Did you include needed headers? (e.g., std_lib_facilities.h)
- Is every name declared before it's used?
  - Did you spell all names correctly?

```
int count;        /* … */ ++Count;        // oops!
char ch;          /* … */ Cin>>c;         // double oops!
```

- Did you terminate each expression statement with a semicolon?

```
x = sqrt(y)+2     // oops!
z = x+3;
```

Debugging

- Carefully follow the program through the specified sequence of steps
  - Pretend you're the computer executing the program
  - Does the output match your expectations?
  - If there isn't enough output to help, add a few debug output statements

```
cerr << "x == " << x << ", y == " << y << '\n';
```

- Be very careful
  - See what the program specifies, not what you think it should say
    - That's much harder to do than it sounds
    - `for (int i=0; 0<month.size(); ++i) {` // *oops!*
    - `for( int i = 0; i<=max; ++j) {` // *oops! (twice)*

Debugging

- When you write the program, insert some checks ("sanity checks") that variables have "reasonable values"
  - Function argument checks are prominent examples of this

```
if (number_of_elements<0)
        error("impossible: negative number of elements");

if (largest_reasonable<number_of_elements)
        error("unexpectedly large number of elements");

if (x<y) error("impossible: x<y");
```

- Design these checks so that some can be left in the program even after you believe it to be correct
  - It's almost always better for a program to stop than to give wrong results

Debugging

- Pay special attention to "end cases" (beginnings and ends)
  - Did you initialize every variable?
    - To a reasonable value
  - Did the function get the right arguments?
    - Did the function return the right value?
  - Did you handle the first element correctly?
    - The last element?
  - Did you handle the empty case correctly?
    - No elements
    - No input
  - Did you open your files correctly?
    - more on this in chapter 11
  - Did you actually read that input?
    - Write that output?

Debugging

- "If you can't see the bug, you're looking in the wrong place"
  - It's easy to be convinced that you know what the problem is and stubbornly keep looking in the wrong place
  - Don't just guess, be guided by output
    - Work forward through the code from a place you know is right
      - so what happens next? Why?
    - Work backwards from some bad output
      - how could that possibly happen?
- Once you have found "the bug" carefully consider if fixing it solves the whole problem
  - It's common to introduce new bugs with a "quick fix"
- "I found the last bug"
  - is a programmer's joke

Note

- Error handling is fundamentally more difficult and messy than "ordinary code"
  - There is basically just one way things can work right
  - There are many ways that things can go wrong
- The more people use a program, the better the error handling must be
  - If you break your own code, that's your own problem
    - And you'll learn the hard way
  - If your code is used by your friends, uncaught errors can cause you to lose friends
  - If your code is used by strangers, uncaught errors can cause serious grief
    - And they may not have a way of recovering

Pre-conditions

- What does a function require of its arguments?
  - Such a requirement is called a pre-condition
  - Sometimes, it's a good idea to check it

```
int area(int length, int width)  // calculate area of a rectangle
   // length and width must be positive
{
        if (length<=0 || width <=0) throw Bad_area{};
        return length*width;
}
```

Post-conditions

- What must be true when a function returns?
  - Such a requirement is called a post-condition

```
int area(int length, int width)  // calculate area of a rectangle
   // length and width must be positive
{
        if (length<=0 || width <=0) throw Bad_area{};
        // the result must be a positive int that is the area
        // no variables had their values changed
        return length*width;
}
```

Pre- and post-conditions

- Always think about them
- If nothing else write them as comments
- Check them “where reasonable”
- Check a lot when you are looking for a bug
- This can be tricky
  - How could the post-condition for area() fail after the pre-condition succeeded (held)?

Testing

- How do we test a program?
  - Be systematic
    - "pecking at the keyboard" is okay for very small programs and for very initial tests, but is insufficient for real systems
  - Think of testing and correctness from the very start
    - When possible, test parts of a program in isolation
      - E.g., when you write a complicated function write a little program that simply calls it with a lot of arguments to see how it behaves in isolation before putting it into the real program (this is typically called "unit testing")
  - More on this question in Chapter 26

Design for Testing

- Use well-defined interfaces
  - so that you can write tests for the use of these interfaces
  - Define invariants, pre- and post conditions
- Have a way of representing operations as text
  - so that they can be stored, analyzed and replayed
- Embed tests of unchecked assumptions (assertions) in the calling and/or called code
  - to catch bad arguments before system testing
- Minimize dependencies and keep dependencies explicit
  - to make it easier to reason about the code
- Have a clear resource management strategy

This will also minimize debugging!

Performance

- Is it efficient enough?
  - Note: *Not* "Is it as efficient as possible?"
  - Computers are fast: You'll have to do millions of operations to even notice (without using tools)
  - Accessing permanent data (on disc) repeatedly can be noticed
  - Accessing the web repeatedly can be noticed
- Time "interesting" test cases
  - *e.g.*, using **time** or **clock()**
  - Repeat ≥3 times; should be ± 10% to be believable

Performance

- What's wrong with this?

```
for (int i=0; i<strlen(s); ++i)  {
        // do something with s[i]
}
```

- It was part of an internet message log analyzer
  - Used for files with many thousands of long log lines

Using clock()

```
int n = 10000000;                   // repeat do_somenting() n times
clock_t t1 = clock();
if (t1 = = clock_t(-1)) {           // clock_t(-1) means "clock() didn't work"
    cerr << "sorry, no clock\n";
    exit(1);
}
for (int i = 0; i<n; i++) do_something();  // timing loop
clock_t t2 = clock();
if (t2 = = clock_t(-1)) {
    cerr << "sorry, clock overflow\n";
    exit(2);
}
cout << "do_something() " << n << " times took "
    << double(t2-t1)/CLOCKS_PER_SEC << " seconds "    // scale result
    << " (measurement granularity: 1/"
    << CLOCKS_PER_SEC << " of a second)\n";
```

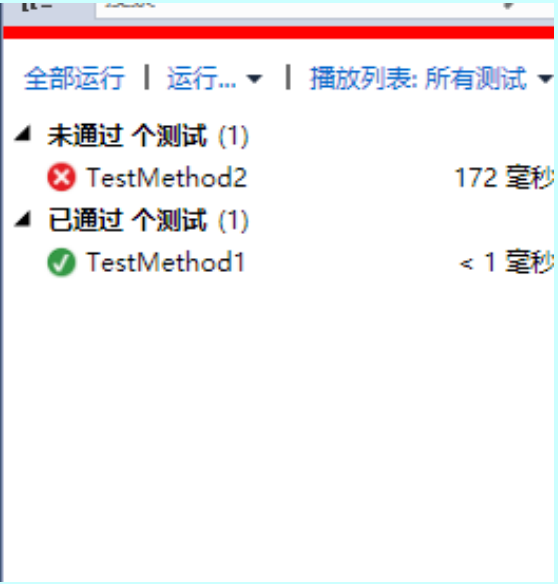
Unit Test

unittest.cpp

```
public:

    TEST_METHOD(TestMethod1)
    {
        // TODO: 在此输入测试代码
        Assert::AreEqual(3,add(1,2));
    }


    TEST_METHOD(TestMethod2)
    {
        // TODO: 在此输入测试代码
        Assert::AreEqual(5, add(1, 2));
    }
};
```



TEST_METHOD(TestMethod1)
{
// TODO: **在此输入测试代码**
Assert::AreEqual(3,add(1,2));
}

TEST_METHOD(TestMethod2)
{
// TODO: **在此输入测试代码**
Assert::AreEqual(5, add(1, 2));
}

Review-Error in Practice

```
    r = kvm_init_debug();
    if (r) {
        printk(KERN_ERR "kvm: create debugfs files failed\n");
        goto ↓out_undebugfs;
    }

    return 0;

out_undebugfs:
    unregister_syscore_ops(&kvm_syscore_ops);
    misc_deregister(&kvm_dev);
out_unreg:
    kvm_async_pf_deinit();
out_free:
    kmem_cache_destroy(kvm_vcpu_cache);
out_free_3:
    unregister_reboot_notifier(&kvm_reboot_notifier);
    unregister_cpu_notifier(&kvm_cpu_notifier);
out_free_2:
out_free_1:
    kvm_arch_hardware_unsetup();
out_free_0a:
    free_cpumask_var(cpus_hardware_enabled);
out_free_0:
    kvm_irqfd_exit();
out_irqfd:
    kvm_arch_exit();
out_fail:
    return r;
} « end kvm_init »
```

Review-Error in Practice 2

```
/* Definition needed to prevent unresolved external in unistd.h */
static int errno;
```

```
/* register adapter */
result = dvb_register_adapter(&dvb->adapter, DRIVER_NAME, THIS_MODULE,
                &dev->usbdev->dev, adapter_nr);
if (result < 0) {
    printk(KERN_ERR "%s: dvb_register_adapter failed "
        "(errno = %d)\n", DRIVER_NAME, result);
    goto ↓fail_adapter;
}
```

Review-Error in Practice 3

```
void nonono(const char* file, int line, const char* msg) {
    fprintf(stderr, "Nonono! %s:%d %s\n", file, line, msg);
    exit(-5);
}
#undef assert
#define assert(a) do {if (!(a)) nonono(__FILE__, __LINE__, #a);}while(0)
```

```
static void
build_append_nameseg(GArray *array, const char *seg)
{
    int len;

    len = strlen(seg);
    assert(len <= ACPI_NAMESEG_LEN);

    g_array_append_vals(array, seg, len);
    /* Pad up to ACPI_NAMESEG_LEN characters if necessary. */
    g_array_append_vals(array, "____", ACPI_NAMESEG_LEN - len);
}
```

Review-Error in Practice 4

```
STDAPI DllGetClassObject(REFCLSID rclsid, REFIID riid, LPVOID *ppv)
{
    CQGAVssProviderFactory *factory;
    try {
        factory = new CQGAVssProviderFactory;
    } catch (...) {
        return E_OUTOFMEMORY;
    }
    factory->AddRef();
    HRESULT hr = factory->QueryInterface(riid, ppv);
    factory->Release();
    return hr;
}
```

Readings

- PPP: Chapter 5, 26

Next

- PPP Chapter 6、7: Desk Calculator Review