Name: _____ Date: _____

Read the following article and answer the questions on page 3.

Don't Ignore That Error!

Pete Goodliffe

I was walking down the street one evening to meet some friends in a bar. We hadn't shared a beer in some time, and I was looking forward to seeing them again. In my haste, I wasn't looking where I was going. I tripped over the edge of a curb and ended up flat on my face. Well, it serves me right for not paying attention, I guess.
It hurt my leg, but I was in a hurry to meet my friends. So, I pulled myself up and carried on. As I walked farther, the pain was getting worse. Although I'd initially dismissed it as shock, I rapidly realized there was something wrong.
But I hurried on to the bar regardless. I was in agony by the time I arrived. I didn't have a great night out, because I was terribly distracted. In the morning, I went to the doctor and found out I'd fractured my shin bone. Had I stopped when I felt the pain, I would've prevented a lot of extra damage that I caused by walking on it. Probably the worst morning after of my life.

TOO MANY PROGRAMMERS write code like my disastrous night out.

Error, what error? It won't be serious. Honestly. I can ignore it. This is not a win- ning strategy for solid code. In fact, it's just plain laziness. (The wrong sort.) No matter how unlikely you think an error is in your code, you should always check for it, and always handle it. Every time. You're not saving time if you don't; you're storing up potential problems for the future.

We report errors in our code in a number of ways, including:

- **Return codes**
  can be used as the resulting value of a function to mean "it didn't work." Error return codes are far too easy to ignore. You won't see anything in the code to highlight the problem. Indeed, it's become normal practice to ignore some standard C functions' return values. How often do you check the return value from `printf`?

- **errno**
  is a curious C aberration, a separate global variable set to sig- nal error. It's easy to ignore, hard to use, and leads to all sorts of nasty problems—for example, what happens when you have multiple threads calling the same function? Some platforms insulate you from pain here; others do not.
- **Exceptions**
  are a more structured language-supported way of signaling and handling errors. And you can't possibly ignore them. Or can you? I've seen lots of code like this:

```
try {
    // ...do something...
}
catch (...) {} // ignore errors
```

  The saving grace of this awful construct is that it highlights the fact that you're doing something morally dubious.

If you ignore an error, turn a blind eye, and pretend that nothing has gone wrong, you run great risks. Just as my leg ended up in a worse state than if I'd stopped walking on it immediately, plowing on regardless of the red flags can lead to very complex failures. Deal with problems at the earliest opportunity. Keep a short account.

Not handling errors leads to:

- **Brittle code**. Code that's filled with exciting, hard-to-find bugs.
- **Insecure code.** Crackers often exploit poor error handling to break into software systems.
- **Poor structure.** If there are errors from your code that are tedious to deal with continually, you probably have a poor interface. Express it so that the errors are less intrusive and their handling is less onerous.

Just as you should check all potential errors in your code, you need to expose all potentially erroneous conditions in your interfaces. Do not hide them, pre- tending that your services will always work.

Why don't we check for errors? There are a number of common excuses. Which of these do you agree with? How would you counter each one?

- Error handling clutters up the flow of the code, making it harder to read, and harder to spot the "normal" flow of execution.
- It's extra work, and I have a deadline looming.
- I know that this function call will never return an error (`printf` always works, `malloc` always returns new memory—if it fails, we have bigger problems...).
- It's only a toy program, and needn't be written to a production-worthy level.

Exercises:

1. What program errors that you usually encounter with?

2. From your programming experiences, how did you prevent errors?