Example Class continued
Software Engineering Concepts: Testing
Thoughts on Assignment 1

CS 311 Data Structures and Algorithms

Lecture Slides

Wednesday, September 4, 2024

Glenn G. Chappell
Department of Computer Science
University of Alaska Fairbanks
ggchappell@alaska.edu
© 2005–2024 Glenn G. Chappell
Some material contributed by Chris Hartman

Unit Overview
Advanced C++ & Software Engineering Concepts

Topics: Advanced C++

- ✓ Expressions
- ✓ Parameter passing I
- ✓ Operator overloading
- (part) Example class
- Parameter passing II
- Invisible functions I
- Managing resources in a class
- Containers & iterators
- Invisible functions II
- Error handling
- Using exceptions

Topics: S.E. Concepts

- ✓ Abstraction
- ✓ Assertions
- Testing
- Invariants

Review

Software Engineering Concepts: Assertions

An **assertion** is a statement made in code that something must be true—or else the code is not working properly.

C++ supports assertions through `assert`, a function-like preprocessor macro defined in header `<cassert>`. It takes a **Boolean expression** (something true or false).

```
assert(n >= 0);
```

If preprocessor symbol `NDEBUG` is not defined, then `assert` evaluates the given expression. If it is true, then `assert` does nothing further; otherwise, it crashes with a message—an application of the **fail-fast** concept.

If `NDEBUG` is defined—typically done by IDEs in release builds—then `assert` does nothing at all.

See `assertion.cpp`.

TO DO

- Write a C++ package containing a class whose objects store and handle a time of day, in hours, minutes, and seconds.
  - Name the class `TimeOfDay`.
  - Give it reasonable constructors.
    - Default constructor.
    - Constructor from hrs/mins/secs.
  - Give it reasonable operators.
    - Pre & post `++`, `--` to make the time go forward & back by 1 second.
    - Stream insertion (`<<`) to print the time like "`3:21:05`", 24-hr time.
    - It might be nice to add other operators. We will not, due to time constraints.
  - Give it other reasonable member functions.
    - `getTime`: get hours/minutes/seconds from an object.
    - `setTime`: set an object's time, from given hrs/mins/secs.
  - For each function, write assertions so that if all pass, then the function will work.

Partially done. See `timeofday.hpp` & `timeofday.cpp`.

For a program that uses class `TimeOfDay`, *see* `timeofday_main.cpp`.

Example Class

continued

Example Class
More CODE

TO DO

- Finish coding class `TimeOfDay` as specified.

Done. See `timeofday.hpp` & `timeofday.cpp`.

Example Class
Notes [1/6]

Note 1. In a C++ constructor, use of **member initializers** is generally preferred to **assignment** of data members.

```
class Skink {
private:
    int _feet;
public:
    Skink()
    {
        _feet = g(3) + 1;
    }
```

Data member

Assignment

```
class Skink {
private:
    int _feet;
public:
    Skink()
        :_feet(g(3) + 1)
    {}
```

Member initializer

Every data member is constructed (default constructed if there is no initializer). On the left, `_feet` is default constructed and then set. On the right, it is constructed with the value we want.

Example Class
Notes [2/6]

Note 2. It is a good idea to have a special convention for naming C++ data members.

Common conventions include beginning the name with "`m_`" or beginning or ending the name with an underscore (`_`).

```
class Skink {
private:
    int _feet;
```

I do not care which convention you use, but mark these names *somehow*—and be consistent about it.

Example Class
Notes [3/6]

Note 3. Avoid duplication of code & other information.

Look at the two `operator++` functions. We could have put the incrementing code into both of them, but we did not.

Also, the constructors call `setTime`.

Why is this a good thing?

It is common for some operators to be based on other operators. For example, postincrement nearly always calls preincrement, as it does in `TimeOfDay`. Thus, we can nearly always write postincrement for a class without knowing anything about how incrementing works in that class.

Example Class
Notes [4/6]

Note 3 (cont'd). Avoiding code duplication is a special case of the **DRY Principle**, formulated by Andy Hunt & Dave Thomas in *The Pragmatic Programmer*, 1999 (DRY = Don't Repeat Yourself):

> Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.

Consider what this means.

Advantages of being DRY:

- Code is more maintainable, easier to change.
- Changes are less likely to introduce bugs. (Suppose some knowledge exists in two places. Then someone changes one of them.)

Example Class
Notes [5/6]

Note 4. Let your compiler help you! All C++ compilers have many **warnings** that they can give. I turn lots of warnings on, including the unused-variable warning.

Q. The dummy `int` parameter to post-increment is never used. How do we avoid an unused-variable warning?

A. Tell your compiler the variable might not be used, using an **attribute**.

```
public:
    // Post-increment
    Foo operator++([[maybe_unused]] int dummy)
    {
        …
```

Attribute

Example Class
Notes [6/6]

Note 5. There are three ways to deal with the possibility of invalid parameter values.

Responsibility for handling the problem lies with the code executed …

1. Insist that given parameters are valid. ← … **before** the function.
2. Allow invalid parameter values, but fix them in the function. ← … **in** the function.
3. If invalid parameter values are passed, then signal the client code that there is a problem. ← … **after** the function.
   - We will discuss this further when we cover *Error Handling*.

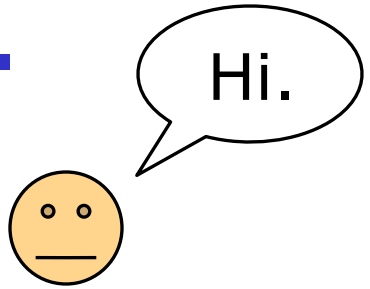
Method #1 is generally the easiest.

Look at the three-parameter constructor. Which method is used?

Software Engineering Concepts: Testing

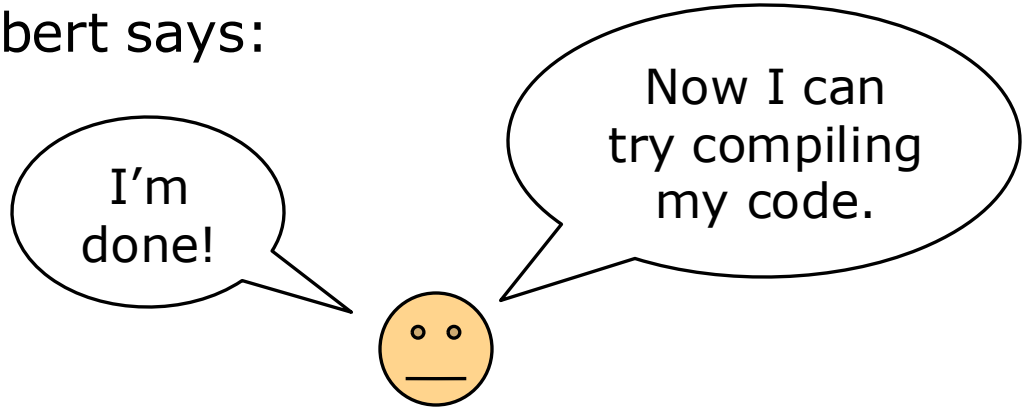
Software Engineering Concepts: Testing Problem [1/2]

Meet Egbert. Egbert is a software developer.



Like many of us, when Egbert writes software, his natural tendency is to start at the beginning, and write until he gets to the end. He writes the first function, and then then second, etc.

When he gets to the end, Egbert says:



But Egbert is making a *terrible mistake!* <Cue ominous music>

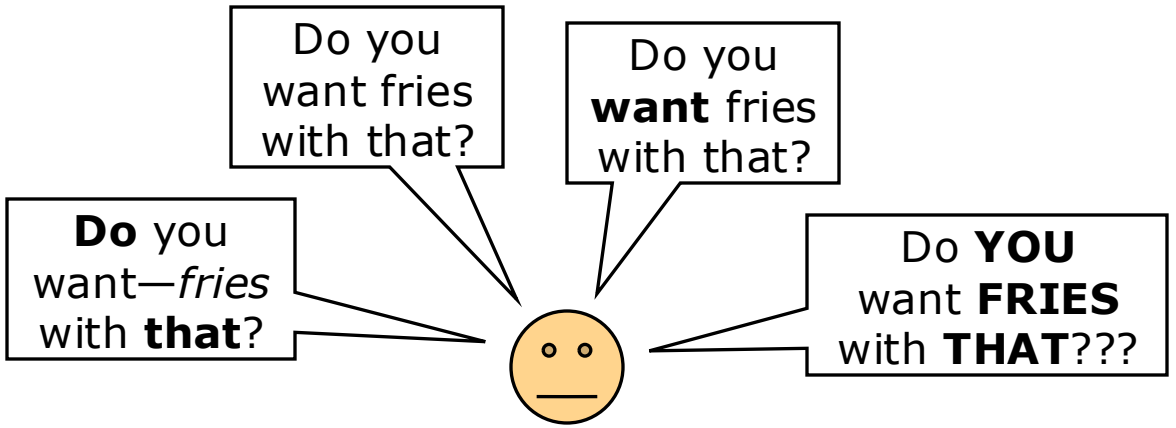
Software Engineering Concepts: Testing Problem [2/2]

Egbert's thinking has two problems. First, code is *done* when:

- It works.
- It is in a readable, maintainable, deliverable form.

Second, while Egbert works, he has no idea if he is doing the right thing. When necessary functions are not present, code cannot be executed—or tested. And *testing is how we find most bugs*.

Perhaps Egbert should prepare himself for a different career.



Revised Process

OR—Egbert could learn to work in a different way.

A Revised Development Process

- Step 1. Write dummy versions of all required components.
  - Make sure **the code compiles**. ← First priority
- Step 2. Fix every bug you can find.
  - "Not having any code in the function body" is often a bug.
  - Write notes to yourself in the code.
  - Make sure **the code works**.
  - In this step, the code should always compile.
- Step 3. Put the code into final, deliverable form.
  - The code needs to be pretty, well commented/documented, and in line with coding standards.
  - Many comments can be based on notes to yourself.
  - Make sure **the code is finished**.
  - In this step, the code should always work.

Unit Testing [1/2]

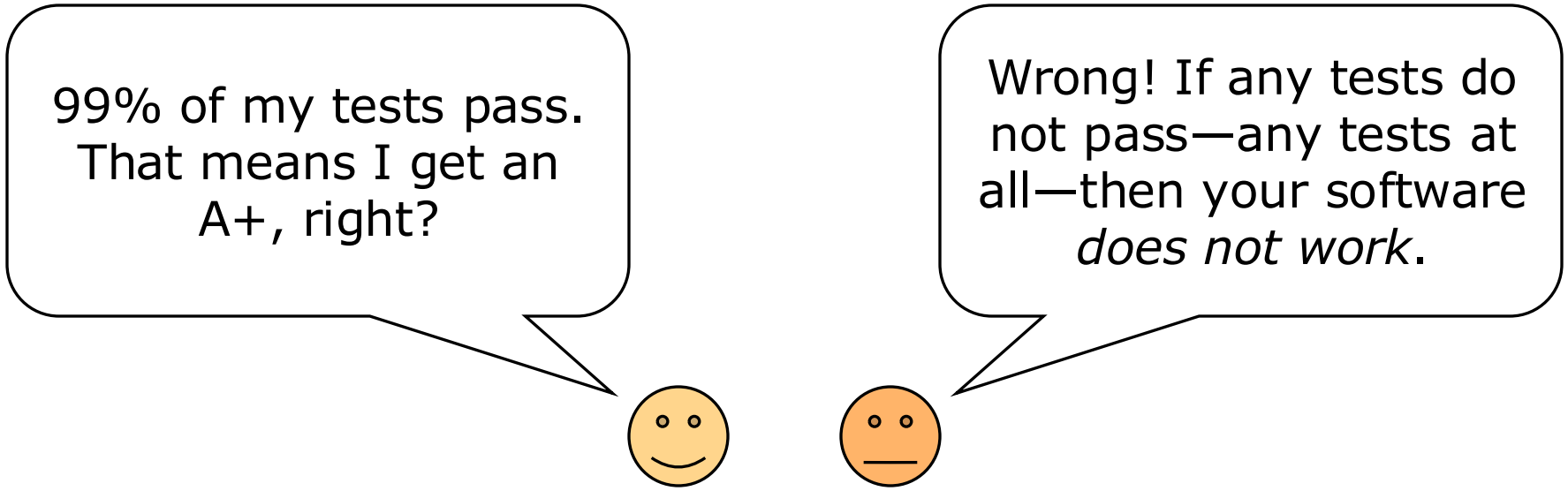
There are many kinds of software testing. One important kind is **unit testing**—tests for the various **units** in the code (functions, classes, etc.), individually.

Unit testing is common enough that high-quality unit-testing **frameworks** (also called **harnesses**, for some reason) are available for most/all major programming languages.

This semester, I will provide test programs for most assignments. These will do unit testing. My test programs use a C++ unit-testing framework called **doctest**.

Unit Testing [2/2]

Avoid thinking about unit tests in terms of something like a 90-80-70-60 scale.



There is a lot more to say about testing and software development processes. *See CS 372.*

Software Engineering Concepts: Testing
Compiler Warnings

Testing is not the only way to find bugs.

Modern compilers have bug-checking capabilities. When a compiler thinks it might have found a bug, it issues a **warning**. Unlike errors, warnings do not stop compilation; they simply cause a message to be issued.

However, these warnings are generally turned off by default.

Strong suggestions:

- Figure out how to turn on compiler warnings. Turn lots of them on.
- Make it your goal to get your code to compile with no warnings.
- If you find a warning unhelpful, then turn it off.

I have posted a link to a site with suggested compiler-warning settings.

Thoughts on Assignment 1

Thoughts on Assignment 1 [1/3]

We have now covered all the Advanced C++ material needed for Assignment 1.

Here are a few quick thoughts on Assignment 1, before we move to the Assignment 2 material.

Thoughts on Assignment 1 [2/3]

In Assignment 1, you write a C++ class of the kind you have probably already written in CS 202 (Computer Science II).

But Assignment 1 differs somewhat from CS 202 work:

- Do a *really good job*; standards are high now.
- Pass a thorough test suite.
- Apply ideas we have covered.
  - Pass parameters using the appropriate methods.
  - Make overloaded operators member or global functions, as appropriate.
  - Avoid duplicating code or other information.
  - You will be required to write assertions as we did in the example class: at the beginning of each function body, write assertions so that, if the assertions all pass, then the function will work properly.
- Follow the *Coding Standards*.

Thoughts on Assignment 1 [3/3]

I have provided a test program, as I will for most assignments in this class. These test programs require the `doctest.h` header. If you use an IDE, then this header should be one of your project files.

To download the header, see the posted link.

In general, there is a huge difference between passing all tests, and passing (say) all but one of the tests. In the former case, your code *works*; in the latter case, it does not.

In this class:

- Being on time is important.
- Working code is more important.

For this first assignment, passing all tests is *absolutely required*. Submitted code that does not pass all tests will not be graded.