Software Engineering Concepts: Invariants
Simple Class Example continued
Thoughts on Project 1

CS 311 Data Structures and Algorithms

Lecture Slides

Monday, August 31, 2020

Glenn G. Chappell

Department of Computer Science

University of Alaska Fairbanks

ggchappell@alaska.edu

Some material contributed by Chris Hartman

Unit Overview
Advanced C++ & Software Engineering Concepts

Major Topics: Advanced C++

- ✓ Expressions
- ✓ Parameter passing I
- ✓ Operator overloading
- ✓ Parameter passing II
- ✓ Invisible functions I
- Integer types
- Managing resources in a class
- Containers & iterators
- Invisible functions II
- Error handling
- Using exceptions
- A little about Linked Lists

We are currently here, working on an example class illustrating some of the ideas we have discussed.

Major Topics: S.E. Concepts

- Invariants
- Testing
- Abstraction

Review

Part 1

Review
Invisible Functions I [1/2]

A C++ compiler may automatically write a number of member functions. Here are six important ones:

```
Dog();                                // Default ctor
~Dog();                               // Dctor
Dog(const Dog & other);               // Copy ctor
Dog & operator=(const Dog & rhs);     // Copy assignment
Dog(Dog && other);                    // Move ctor*
Dog & operator=(Dog && rhs);          // Move assignment*
```

The **Big Five**

***Move** operations were added in C++11.

For each function, the automatically generated version calls the corresponding member function for each data member.

Review
Invisible Functions I [2/2]

The default ctor is automatically generated when we declare no ctors.

For the Big Five, we covered the rules for when they are automatically generated. But you do not need to know these; just follow the Rule of Five.

Two special options. Use these!

- Automatically generate. `Dog(Dog && other) = default;`
- Eliminate the function. `Dog(Dog && other) = delete;`

The **Rule of Five**: If you define one of the Big Five, then define or =delete all of them.

Typically, this happens when an object directly manages some resource—like dynamically allocated memory—that will need to be cleaned up.

Software Engineering Concepts: Invariants

Software Engineering Concepts: Invariants Basics [1/2]

An **invariant** is a condition that is always true at a particular point in a computation. Typically, it says something about the *value* of a variable.

```
if (ix < 0)
{
    flagError("Index too small");
    return;
}
// Invariant: ???
if (ix >= myVec.size())
{
    flagError("Index too large");
    return;
}
// Invariant: ???
myItem = myVec[ix];
```

Suppose `myVec` is a `vector<int>`.

We wish to set (non-const) `int` variable `myItem` equal to `myVec[ix]`, if possible.

Q. When would it be impossible?

A. ???

Software Engineering Concepts: Invariants Basics [1/2]

An **invariant** is a condition that is always true at a particular point in a computation. Typically, it says something about the *value* of a variable.

```
if (ix < 0)
{
    flagError("Index too small");
    return;
}
// Invariant: ???
if (ix >= myVec.size())
{
    flagError("Index too large");
    return;
}
// Invariant: ???
myItem = myVec[ix];
```

Suppose `myVec` is a `vector<int>`.

We wish to set (non-const) `int` variable `myItem` equal to `myVec[ix]`, if possible.

Q. When would it be impossible?

A. When `ix` is out of range, that is, when it is not a valid index for `myVec`.

Software Engineering Concepts: Invariants Basics [1/2]

An **invariant** is a condition that is always true at a particular point in a computation. Typically, it says something about the *value* of a variable.

```
if (ix < 0)
{
    flagError("Index too small");
    return;
}
// Invariant: ix >= 0
if (ix >= myVec.size())
{
    flagError("Index too large");
    return;
}
// Invariant: (ix >= 0) && (ix < myVec.size())
myItem = myVec[ix];
```

Suppose `myVec` is a `vector<int>`.

We wish to set (non-const) `int` variable `myItem` equal to `myVec[ix]`, if possible.

Q. When would it be impossible?

A. When `ix` is out of range, that is, when it is not a valid index for `myVec`.

Software Engineering Concepts: Invariants Basics [2/2]

We use invariants:

- To ensure that we are allowed to perform various operations.
- To remind ourselves—and others who may read our code later—of information that is implicitly known in a program. This can make code more maintainable.
- To document ways in which code can be used.
- To help us verify that our programs are correct.

It is a good idea to comment invariants that are *not obvious*.

(The invariants on the previous slide were obvious, I think.)

In this class, we are explicit about *some* of our invariants, and we use them in our documentation.

Software Engineering Concepts: Invariants Pre & Post [1/4]

We are particularly interested in two special kinds of invariants: *preconditions* and *postconditions*.

A **precondition** is an invariant at the beginning of a function.

- The responsibility for making sure the precondition is true rests with the calling code (that is, the client).
- In practice, a precondition **states what must be true for the function to execute properly**.

A **postcondition** is an invariant at the end of a function.

- It tells what services the function has performed for the client code.
- The responsibility for making sure the postcondition is true rests with the function itself.
- In practice, a postcondition **describes the function’s effect using statements about values**.

Software Engineering Concepts: Invariants Pre & Post [2/4]

Preconditions and postconditions are the basis of **operation contracts**.

- We think of a function call as the carrying out of a contract. The function says to the caller, “If you do *this* [preconditions], then I will do *this* [postconditions].”
- If the preconditions are met, then the function is required to make the postconditions true upon its *normal* termination.
  - We consider abnormal termination (throwing an exception) later.
- If the preconditions are not met, then the function may be considered to have no responsibilities.

Software Engineering Concepts: Invariants
Pre & Post [3/4]

Example 1

- Write reasonable pre- and postconditions for the following function, which is supposed to compute the average speed of an object, given the distance it travels and the time elapsed.

```
// avgSpeed
// Pre:  ???
// Post: ???

double avgSpeed(int dist,
                int time)
{
    return double(dist) / double(time);
}
```

Preconditions:
What **must be true** for the function to execute properly?

Postconditions:
Describe the function's effect using statements about values.

Software Engineering Concepts: Invariants Pre & Post [3/4]

Example 1

- Write reasonable pre- and postconditions for the following function, which is supposed to compute the average speed of an object, given the distance it travels and the time elapsed.

Preconditions:
What **must be true** for the function to execute properly?

```
// avgSpeed
// Pre:  time != 0.
// Post: return == dist/time, where the computation is
//        done using floating-point division.
double avgSpeed(int dist,
                int time)
{
   return double(dist) / double(time);
}
```

Postconditions:
Describe the function's effect using statements about values.

Software Engineering Concepts: Invariants Pre & Post [4/4]

Example 2

- Write reasonable pre- and postconditions for the following function, which is supposed to store the number 7 in the provided memory.

```
// store7
// Pre:  ???


// Post: ???
void store7(int * ptr)
{
    *ptr = 7;
}
```

Preconditions:
What **must be true** for the function to execute properly?

Postconditions:
Describe the function's effect using statements about values.

Software Engineering Concepts: Invariants
Pre & Post [4/4]

Example 2

- Write reasonable pre- and postconditions for the following function, which is supposed to store the number 7 in the provided memory.

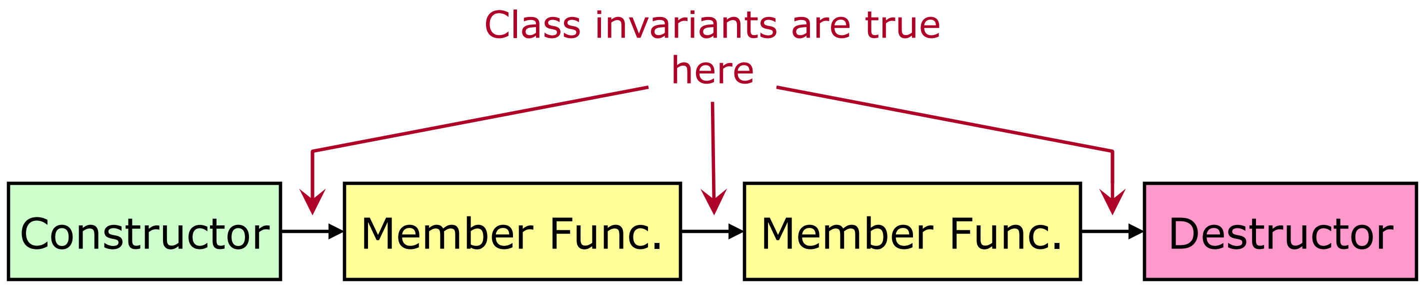
```
// store7
// Pre:  ptr points at writable memory sufficient
//         to hold an int.
// Post: *ptr == 7.
void store7(int * ptr)
{
    *ptr = 7;
}
```

Preconditions:
What **must be true** for the function to execute properly?

Postconditions:
Describe the function's effect using statements about values.

Software Engineering Concepts: Invariants
Class Invariants [1/4]

For a given class, a **class invariant** is an invariant that holds for an object of the class, whenever execution is not inside a member function.



- Class invariants are preconditions for every public member function, except constructors.
- Class invariants are postconditions for every public member function, except the destructor.
- Since we know this, you do not need to list class invariants in the pre- and postcondition lists of public member functions.
- In practice, class invariants are **statements about data members** that indicate what it means for an object to be **valid** or **usable**.

Software Engineering Concepts: Invariants
Class Invariants [2/4]

Example

- Write reasonable class invariants for the following class.

Class invariants:
statements about data members that indicate what it means for an object to be **valid** or **usable**.

```
// class Date
// Invariants:
//     ???

class Date {
public:
    [Lots of code goes here]
private:
    int _mo;    // Month 1..12
    int _day;   // Day 1..#days in month given by _mo
};  // End class Date
```

Software Engineering Concepts: Invariants
Class Invariants [2/4]

Example

- Write reasonable class invariants for the following class.

Class invariants:
statements about data members that indicate what it means for an object to be **valid** or **usable**.

```
// class Date
// Invariants:
//     1 <= _mo <= 12.
//     1 <= _day <= #days in month given by _mo.
class Date {
public:
    [Lots of code goes here]
private:
    int _mo;    // Month 1..12
    int _day;   // Day 1..#days in month given by _mo
};  // End class Date
```

Software Engineering Concepts: Invariants
Class Invariants [3/4]

Think about the creation of dynamic objects. In C++, we do:

```
Foo * p = new Foo[100];
```

But we could, instead, simply allocate some memory and point a `(Foo *)` at it. This is what is usually done in C—and is still legal in C++:

```
Foo * p = (Foo *)malloc(100 * sizeof(Foo));
```

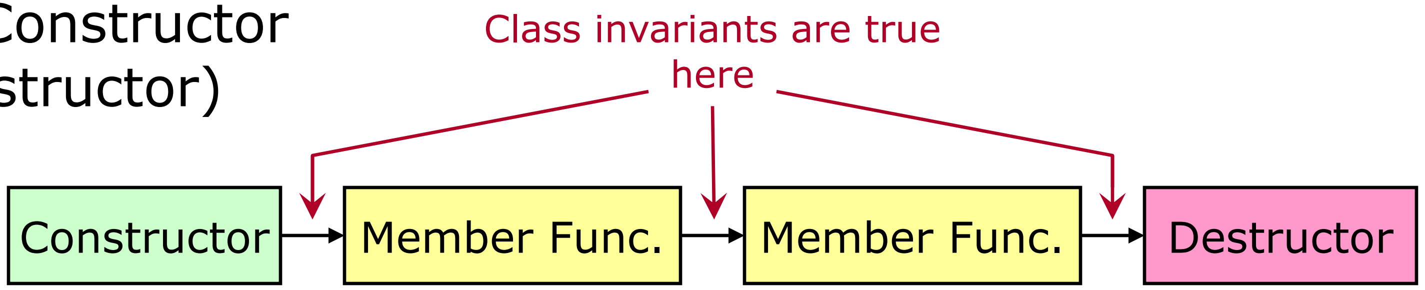
I claim that the first way of doing it is better. Why?

- Yes, it is simpler and cleaner. What other reasons are there?
- Hint. The two lines of code above do not do the same thing.
- Another hint. We are discussing invariants.
- *See the next slide.*

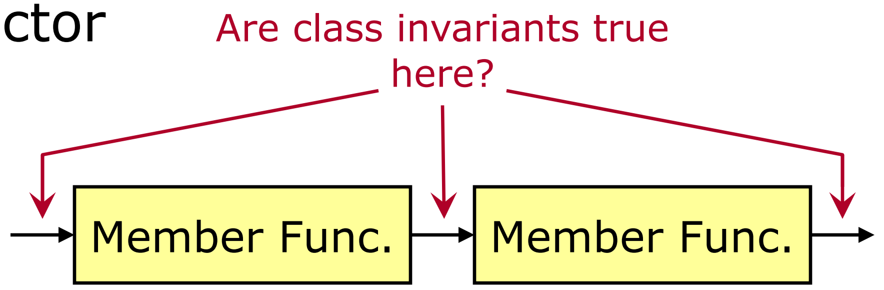
Software Engineering Concepts: Invariants
Class Invariants [4/4]

In C++, using `new` calls a constructor for each allocated item, thus ensuring that class invariants are true for that item.

With Constructor
(& Destructor)



Without Constructor
(& Destructor)



The job of a constructor is to make the class invariants true.

Review

Part 2

Review
Simple Class Example

TO DO

- Write a C++ package consisting primarily of a class whose objects store and handle a **time of day**, in hours, minutes, and seconds.
  - Name the class `TimeOfDay`.
  - Give it reasonable ctors, etc.
    - Default ctor, ctor from hours/minutes/seconds, Big Five.
    - Can we use automatically generated functions?
      - We can =default the Big Five.
      - We will need to write our own default ctor and 3-parameter ctor.
  - 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.
    - Assignment (=) is included in the Big Five, above.
    - It might be nice to add more. 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, giving hours/minutes/seconds.
    - Again, time constraints prevent us from adding more.

Partially done. See `timeofday.h/.cpp`*.*
For a program that uses class `TimeOfDay`*, see* `timeofday_main.cpp`*.*

Simple Class Example

continued

Simple Class Example
More CODE

TO DO

- Finish writing class `TimeOfDay`.

Done. See `timeofday.h/.cpp`*.*

Simple Class Example
Notes [1/3]

Note 1. External interface does not dictate internal implementation (although it certainly influences it).

Class `TimeOfDay` deals with the outside world in terms of hours, minutes, and seconds. However, it has only one data member, which counts seconds.

Simple Class Example
Notes [2/3]

Note 2. Avoid duplication of code.

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 without knowing anything about how incrementing works in any particular class.

Simple Class Example
Notes [3/3]

Note 3. 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.
   - Use preconditions.
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 ctor. Which method is used?

Thoughts on Project 1

Thoughts on Project 1 [1/6]

We have covered all the Advanced C++ material needed for Project 1. In that project, you write a C++ class of the kind you have probably already written in CS 202 (Computer Science II).

But Project 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 correctly.
  - Make overloaded operators member or global functions, as appropriate.
  - Handle invisible functions appropriately (Define? =delete? =default?)
  - Document preconditions where these exist, along with class invariants.
  - Etc.
- Follow the *Coding Standards*.

Thoughts on Project 1 [2/6]

I have provided a test program for this project.

There is a huge difference between passing all the 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.

Thoughts on Project 1 [3/6]

Do not write unnecessary code.

If the various automatically generated member functions do the right thing, then use them. Just because a class needs to have (say) a copy ctor, does not mean you need to write it.

If a function can call some other function to do most or all of its work, then it should do so.

Thoughts on Project 1 [4/6]

What happens if the client code does something bad?

```
ProductOrder po;
po.setNumber(-2);
```

The number of items in the inventory is not allowed to be negative. How should the above be handled?

See Note 3 in the *Simple Class Example* topic, and observe that simply forbidding incorrect parameter values—by writing a precondition—is (1) a correct way of handling this situation, and (2) very easy.

Thoughts on Project 1 [5/6]

Let your compiler help you.

All C++ compilers have many **warnings** that they can give. By default, most of these are turned off. Strong suggestion: turn them *all* on. (If you find a warning unhelpful, then turn that warning off.) Get your code to compile with *no warnings at all*.

One tricky issue: the dummy `int` parameter to post-increment is never used. How do we avoid an unused-variable warning?

Answer. Tell your compiler it may not be used, using an **attribute**.

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

Thoughts on Project 1 [6/6]

Some important things to note about Project 1 are the things you do *not* need to do:

- Anything related to passing by Rvalue reference—other than =default'ing the move operations.
- Explicitly defining any of the Big Five (but *do* =default them).
- Anything covered in forthcoming *Advanced C++* topics.