Invariants
Simple Class Example

CS 311 Data Structures and Algorithms
Lecture Slides
Monday, January 28, 2013

Chris Hartman
Department of Computer Science
University of Alaska Fairbanks
cmhartman@alaska.edu
Based on material by Glenn G. Chappell

Unit Overview
Advanced C++ & Software Engineering Concepts

Major Topics: Advanced C++

- ✓ The structure of a package
- ✓ Parameter passing
- ✓ Operator overloading
- ✓ Silently written & called functions
- Pointers & dynamic allocation
- Managing resources in a class
- Templates
- Containers & iterators
- Error handling
- Introduction to exceptions
- Introduction to Linked Lists

Major Topics: S.E. Concepts

- ✓ Abstraction
- ✓ Invariants
- Testing
- Some principles

Review
Operator Overloading

Operators can be implemented using global or member functions.

- Global: the parameters are the operands.
- Member: first operand is **`*this`**, the rest are parameters.
- Postfix increment & decrement (**`n++`**, **`n--`**) get a dummy **`int`** parameter, to distinguish them from the prefix versions (**`++n`**, **`--n`**).

Implement an operator using a member function, unless you have a good reason not to.

- Good Reason #1: To allow for implicit type conversions on the first argument. Applies to: non-modifying arithmetic, comparison, and bitwise operators.
  - For example: **`+ - * / % == != < <= > >=`**
- Good Reason #2: When you cannot make it a member, because it would have to be a member of a class you cannot modify.
  - Quintessential examples: stream insertion (**`<<`**) and extraction (**`>>`**).

We usually use operators only for operations that happen **quickly**.

- One exception: Assignment for container types.

Review
Silently Written & Called Functions [1/3]

C++ will **silently write** four important member functions:

- Default ctor.
- Copy ctor.
- Copy assignment.
- Dctor.

"The Big Three"

When

- The default ctor is silently written when you declare no ctors.
- The other three are silently written when you do not declare them.

The silently written versions:

- Are **public**.
- Call the **corresponding functions** for all data members.

Review
Silently Written & Called Functions [2/3]

Some of these can be **silently called** as well.

- The default ctor is called when you declare a variable with no ctor parameters, and when you declare an array (or, generally, any container holding already initialized objects).
- The copy ctor is called when you pass by value and *maybe* when you return by value.
- The dctor is called:
  - On an automatic (local, non-static) object when it goes out of scope.
  - On a static object when the program ends.
  - On a non-static member object when the object it is a member of is destroyed.
  - On a dynamic object when you delete a pointer to it.

Review
Silently Written & Called Functions [3/3]

Silently written functions are **good**.

- Do not waste effort. If the compiler will write a perfectly good function for you, then do not write it yourself.

So, use them often. And when you do, indicate this in a comment.

- This is a reminder that these functions exist and are part of the class design.

The Law of the Big Three

- **If you need to declare one of the Big Three** (copy ctor, copy assignment, dctor), **then you probably need to declare all of them.**
- This tends to happen when the class manages a resource (for example, dynamically allocated memory, an open file, etc.). More on this soon.

How do we eliminate the copy ctor and copy assignment?

- **Declare** the copy ctor and copy assignment **`private`**.
- Do not **define** them.

Now **no one** can call these functions.

- You (the class author) cannot accidentally call them, because you did not define them.
- Client code *can* define them, but that does not matter; they cannot call them, because they are private.

Software Engineering Concepts: Invariants
Basics [1/2]

An **invariant** is a condition about the value of a variable that is always true at a particular point in an algorithm.

Example

- Suppose that **`myArray`** is an array of **`int`**'s with size **`myArraySize`**.
- We wish to set the variable **`myItem`** equal to **`myArray[i]`**, if possible.

```
if (i < 0)
{
    errorMessage("Error: i is too small");
    return;
}
// Invariant: i >= 0
if (i >= myArraySize)
{
    errorMessage("Error: i is too large");
    return;
}
// Invariant: (i >= 0) && (i < myArraySize)
myItem = myArray[i];
```

Software Engineering Concepts: Invariants Basics [2/2]

We use invariants:

- To ensure that we are allowed to perform various operations.
- To remind ourselves of the information that is implicitly known in a program.
- To document ways in which code can be used.
- To help us verify that our programs are correct.

Software Engineering Concepts: Invariants
Pre & Post [1/3]

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 (ie. 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, postconditions **describe the function's effect using statements about objects & values**.

Software Engineering Concepts: Invariants Pre & Post [2/3]

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 (exceptions) later.
- If the preconditions are not met, then the function can be considered to have no responsibilities.

Punch Line

- In this class, we write preconditions and postconditions for **every** function you write (except, possibly, **`main`**).
  - See the “Coding Standards”.

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

Example

- 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 objects & values.

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

Example

- 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 enough writable memory
//        to hold an int
// Post: *ptr == 7
void store7(int * ptr)
{
   *ptr = 7;
}
```

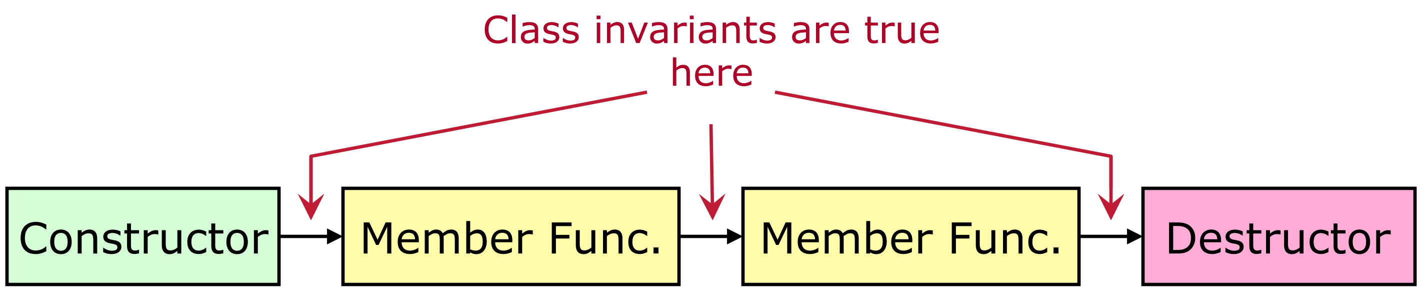
Preconditions:
What **must be true** for the function to execute properly?

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

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

Another important kind of invariant is a **class invariant**.

- A *class invariant* is an invariant that holds whenever an object of the class exists, and execution is not in the middle of a public member function call.



- Class invariants are preconditions of 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]

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 {
// Date: public functions
public:
    [Lots of code goes here]
// Date: data members
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]

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 {
// Date: public functions
public:
    [Lots of code goes here]
// Date: data members
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 dynamic allocation. In “C”, we do:

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

In C++, we prefer:

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

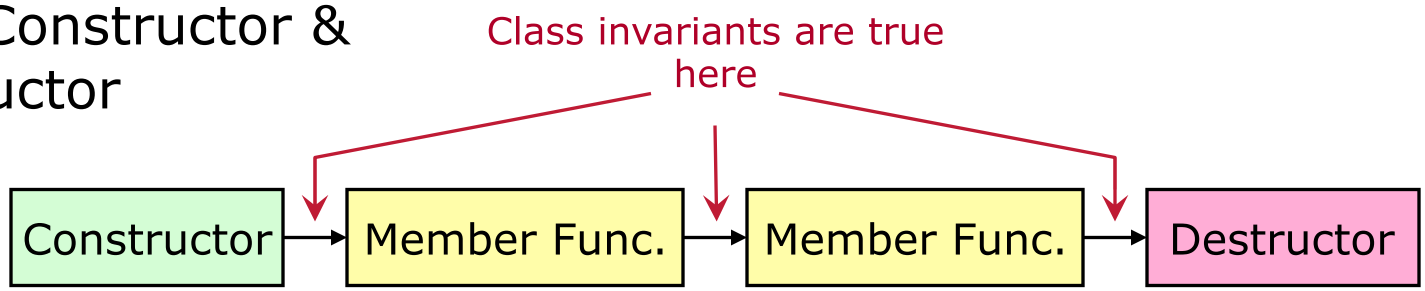
Why?

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

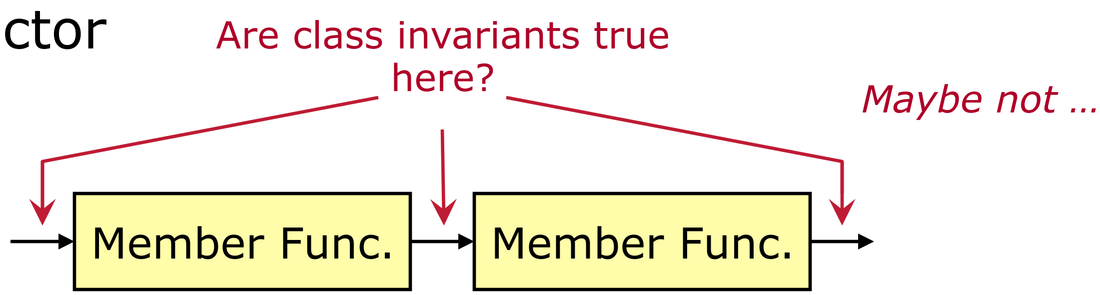
Software Engineering Concepts: Invariants
Class Invariants [4/4]

In C++, using "**`new`**" calls a constructor, thus ensuring that class invariants are true.

With Constructor & Destructor



Without Constructor & Destructor



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

Simple Class Example
Write It!

TO DO

- Write a simple class that stores and handles a **time of day**, in seconds.
  - Call it "**`TimeSec`**".
  - Give it reasonable ctors, etc.
    - Can we use silently written functions?
  - Give it reasonable operators.
    - Like what?

Simple Class Example
Write It!

TO DO

- Write a simple class that stores and handles a **time of day**, in seconds.
  - Call it “**`TimeSec`**”.
  - Give it reasonable ctors, etc.
    - Can we use silently written functions?
      - *Yes, for the Big Three.*
      - *We will write our own default ctor.*
  - Give it reasonable operators.
    - Like what?
      - *Pre & post* `++`, `--`.
      - *Equality & inequality:* `==`, `!=`.
      - *Stream insertion:* `<<`.
      - *Note: It would be reasonable to add more. We will not, but only due to time constraints.*
  - Give it other member functions.
    - *getTime*
    - *setTime*
    - *toString*

Simple Class Example
Notes [1/2]

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

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

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 constructor calls setTime, etc.)
- Why is this a good thing?

Simple Class Example
Notes [2/2]

Note 3: There are three ways to deal with the possibility of invalid parameters.

- Insist that the parameters be valid.
  - Use a precondition.
- Allow invalid parameter values, but then fix them.
- If invalid parameter values are passed, signal the client code that there is a problem.
  - We will discuss this further when we get to “Error Handling”.

Responsibility for dealing with the problem lies with the code executed …

← … **before** the function.

← … **in** the function.

← … **after** the function.

Look at the three-parameter constructor. Which solution was used there?

Software Engineering Concepts: Testing
A Tragic Story [1/4]

Suppose you are writing a software package for a customer.

- The project requires the writing of four functions.

```
double foo(int n);    // gives ipsillic tormorosity of n
void foofoo(int n);   // like foo, only different
int bar(int n);       // like foofoo, only more different
char barbar(int n);   // like bar; much differenter
```

So, you get to work. You start by writing function **foo** …

Software Engineering Concepts: Testing
A Tragic Story [2/4]

… after a huge amount of effort, the deadline arrives. But you are not done. However, you do have three of the four functions written. Here is what you have.

```
double foo(int n)
{
    [amazingly clever code here]
}

void foofoo(int n)
{
    [stunningly brilliant code here]
}

int bar(int n)
{
    [heart-breakingly high-quality code here]
}

// Note to self: write function barbar.
```

Software Engineering Concepts: Testing
A Tragic Story [3/4]

You meet with the customer. You explain that you are not done. The customer is a bit annoyed, of course, but he knows that schedule overruns happen in every business.

So, he asks, "Well, what *have* you finished? What can it do?"

Unfortunately, you do not have all the function prototypes in place. Thus your unfinished package, when combined with the code that is supposed to use it, *does not even compile*, much less actually do anything.

You tell the customer, "Um, actually, it can't do anything at all."

"Do you want to see my beautiful code?" you ask.

"No," replies the customer, through clenched teeth.

The customer storms off and screams at your boss, who confronts you and says you had better have something good in a week. You solemnly assure them that this will happen.

You go back to work …

Software Engineering Concepts: Testing
A Tragic Story [4/4]

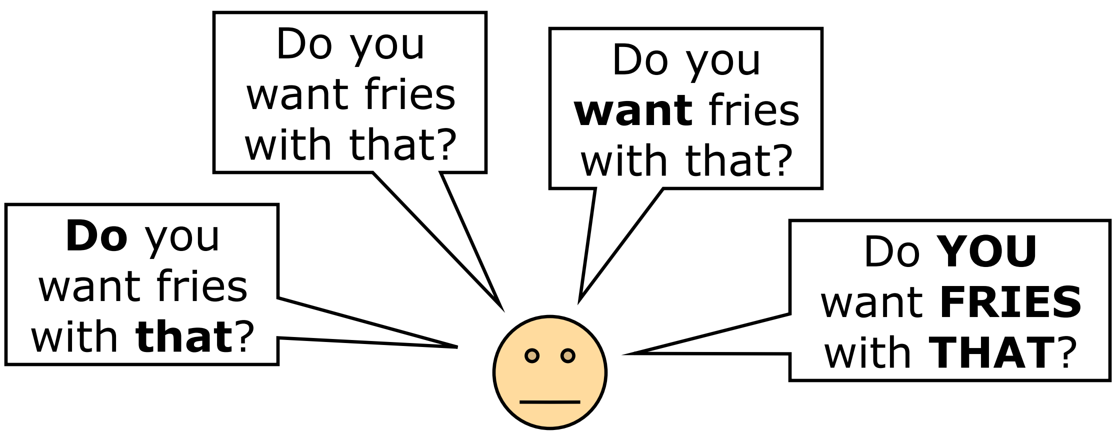
… and you write a do-nothing function **`barbar`**, just to get things to compile.

However, when you do this, you realize that, since you have never done a proper compile of the full package, you have never really done a proper test of the first three functions.

Now that you *can* test them, you find that they are full of bugs.

Alas, you now know that you have been far too optimistic; nothing worthwhile is going to get written in the required week.

You begin practicing your lines for your exciting new career:



Software Engineering Concepts: Testing Lessons

Observations

- Code that does not compile is worthless *to a customer*, even if it is "nearly done".
- It *might* not be worth anything *to anyone*; **you can't tell**, because …
- Code that does not compile cannot be tested, and so it *might* be much farther from being done than you suspect.
- Testing is what uncovers bugs.

Conclusion

- First priority: **Get your code to compile**, so that it can be tested.

A Revised Development Process

- Step 1. Write dummy versions of all required modules.
  - Make sure the code **compiles**.
- Step 2. Fix every bug you can find.
  - "Not having any code in the function body" is a bug.
  - Write notes to yourself in the code.
  - Make sure the code **works**.
- Step 3. Put the code into finished form.
  - Make it 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**.

Software Engineering Concepts: Testing
Try Again [1/3]

Suppose you had used this revised development process earlier.

Step 1. Write dummy versions of all required modules.

```
double foo(int n)    // gives ipsillic tormorosity of n
{}  // WRITE THIS FUNCTION!!!
void foofoo(int n)  // like foo, only different
{}  // WRITE THIS FUNCTION!!!
int bar(int n)      // like foofoo, only more different
{}  // WRITE THIS FUNCTION!!!
char barbar(int n)  // like bar; much differenter
{}  // WRITE THIS FUNCTION!!!
```

Does it compile?

- No. My compiler says **foo**, **bar**, **barbar** must each return a value.

Software Engineering Concepts: Testing
Try Again [2/3]

Continuing Step 1.

Add dummy **`return`** statements.

```
double foo(int n)   // gives ipsillic tormorosity of n
{ return 1.; }   // WRITE THIS FUNCTION!!!
void foofoo(int n)  // like foo, only different
{}               // WRITE THIS FUNCTION!!!
int bar(int n)      // like foofoo, only more different
{ return 1; }    // WRITE THIS FUNCTION!!!
char barbar(int n)  // like bar; much differenter
{ return 'A'; }  // WRITE THIS FUNCTION!!!
```

Does it compile?

- Yes. Step 1 is finished.

Software Engineering Concepts: Testing
Try Again [3/3]

Step 2. Fix every bug you can find.

You begin testing the code. Obviously, it performs very poorly. But you begin writing and fixing. And running the code. So when something does not work, *you know it*. When you figure something out, you make a note to yourself about it.

As before, the deadline arrives, but the code is not finished yet.

You meet with the customer. “The project is not finished,” you say, “but **here is what it can do**.”

You estimate how long it will take to finish the code.

You can make this estimate with confidence, because you have a list of tests that do not pass; you know exactly what needs to be done.

Software Engineering Concepts: Testing
Development Methodologies

Software-development methodologies often include standards for how code should be tested.

- In particular, see, “Test-Driven Development”.

Many people recommend *writing your tests first*.

- Each time you add new feature, you first write tests (which should fail), then you make the tests pass.
- When the finished test program runs without flagging problems, Step 2 is done. Pretty up the code, and it is finished.

We will use a variation on this in the assignments in this class.

- I will provide the (finished) test program.
- However, when you turn in your assignment, I act as the customer; I do not want to see code that does not compile.