Software Engineering Concepts: Testing
Simple Class Example continued
Pointers & Dynamic Allocation

CS 311 Data Structures and Algorithms

Lecture Slides

Monday, September 14, 2009

Glenn G. Chappell
Department of Computer Science
University of Alaska Fairbanks
CHAPPELLG@member.ams.org

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
Software Engineering Concepts: Invariants [1/2]

An **invariant** is a condition that is always true at a particular point in an algorithm.

Special kinds

- **Precondition**. An invariant at the beginning of a function. The responsibility for making sure the preconditions are true rests with the calling code.
  - What must be true for the function to execute properly.
- **Postcondition**. An invariant at the end of a function. Tells what services the function has performed for the caller.
  - Describe the function's effect using statements about objects & values.
- **Class invariant**. An invariant that holds whenever an object of the class exists, and execution is not in the middle of a public member function call.
  - Statements about data members that indicate what it means for an object to be valid or usable.

Notes

- The above form the basis for **operation contracts**.
- The job of a constructor is to make the class invariants true.

Review
Software Engineering Concepts: Invariants [2/2]

Write reasonable class invariants for the following class.

```
// class Foo
// Invariants:
//     v.size() >= 6.
class Foo {
public:
    …
    // bar
    // Pre: None.
    bar()
    { v[5] = 0; }
    …
private:
    std::vector<int> v;
};
```

Statements about data members that indicate what it means for an object to be valid or usable.

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]

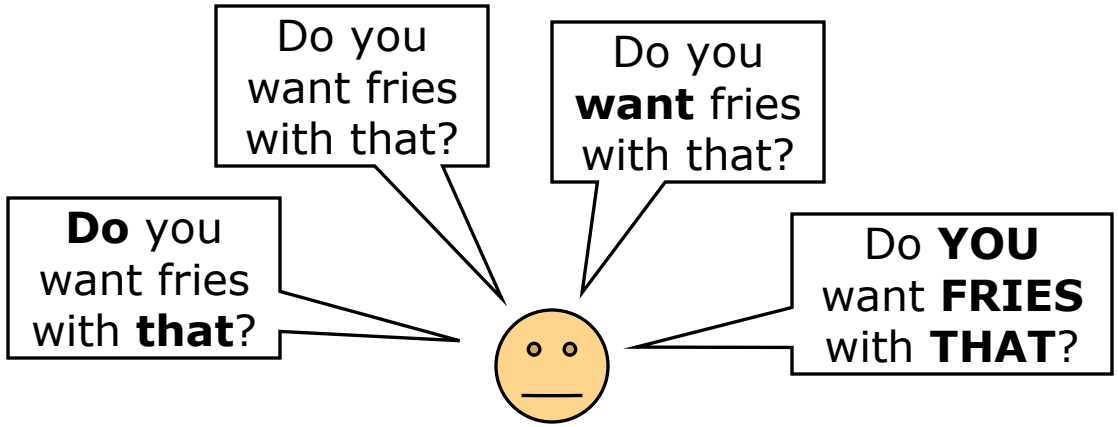
… and you write a do-nothing function **`barbar`**, just to get things to compile.

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.

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.

Simple Class Example continued
Write Some More

Last time, we began writing a simple class whose objects store a time of day, in seconds: class **`TimeSec`**.

- In class, we put together the general structure of the package.
- I have added getter & setter functions and another ctor.
- Now we continue …

TO DO

- Write the member functions and associated global operators for class **`TimeSec`**.
- Apply the software engineering concepts just discussed.

Done. See **`timesec.h`**, **`timesec.cpp`**, *on the web page.*

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.
- 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?

Pointers & Dynamic Allocation
Introduction

A **pointer** is a variable that contains the memory address of another variable.

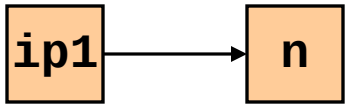
- The **type** of a pointer is the type of the value it points to, followed by "*****".

```
int n;  // int
int * ip1; // pointer to int
```

Get the address of a variable using the address-of operator ("**&**").

- This operator returns a pointer to the variable.

```
ip1 = &n;
```



Get the value pointed to using the **dereference** operator ("*****").

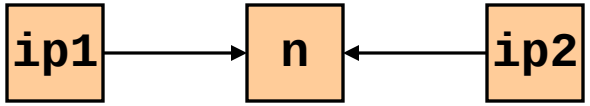
- This operator returns a reference to the value the pointer points to.

```
cout << *ip1;  // outputs n
*ip1 = 3;      // sets n to 3
```

Pointers & Dynamic Allocation
Copying and Null

We can copy and assign pointers. This does not copy the value pointed to.

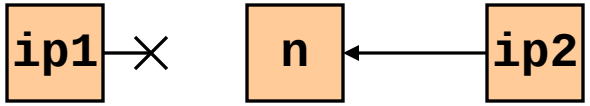
```
int * ip2 = ip1;
```



A special value: the **null pointer**, represented by "**0**" used as a pointer.

- Why? This is a value we can test for.
- Alternative: "**NULL**", defined in **<iostream>**.

```
ip1 = 0;
if (ip1 == 0) …
ip1 = NULL;
```



Do not dereference a null pointer or an uninitialized pointer.

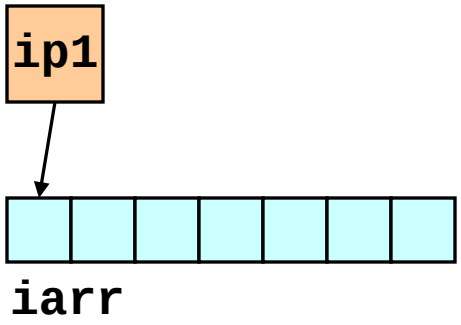
```
cout << *ip1;  // BAD!! ip1 is null

int * ip3;
*ip3 = 2;      // BAD!! ip3 is uninitialized
```

Pointers & Dynamic Allocation
Pointer Arithmetic [1/2]

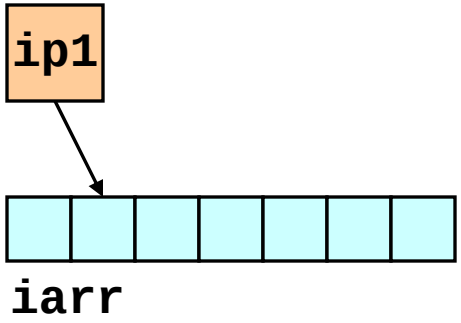
The name of an array can be used as ("decays to") a pointer to the first item in the array.

```
int iarr[7];
ip1 = iarr;
ip1 = &(iarr[0]);  // Same as above
```



When a pointer points to an array item, we can increment and decrement the pointer to move it to the next and previous items in the array.

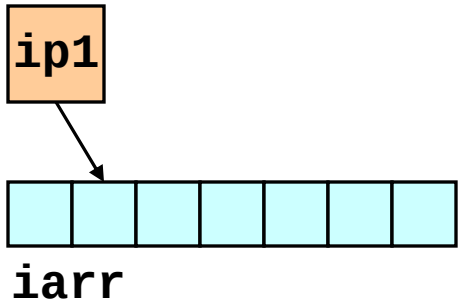
```
++ip1;
```



Do not increment or decrement a pointer that does not point to an array.

Pointer Arithmetic [2/2]

Once you understand that, other arithmetic operators act as expected.




```
ip1 += 3;  // advances ip1 by 3
ip1 -= 2;  // moves ip1 back 2
ip2 = ip1 + 4;
   // points ip2 to the item 4 past ip1's item
   // does not change ip1
std::ptrdiff_t d = ip2 - ip1;  // Sets d to 4
```

Only subtract pointers that point to the same array.

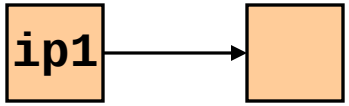
You can also use a pointer as if it **is** an array.

```
cout << ip1[2];  // same as *(ip1+2)
```

Dynamic Allocation [1/2]

The **new** operator allocates and constructs a value of a given type and returns a pointer to it.

```
ip1 = new int;
```



We can add constructor parameters.

```
ip1 = new int(2);
```

When we do dynamic allocation, we must deallocate using **delete** on a pointer (any pointer) to the dynamic value.

```
delete ip1;  // destroys the int; does not affect ip1
```

Do not depend on the destructor of the pointer to do this. The destructor of a pointer does **nothing**.

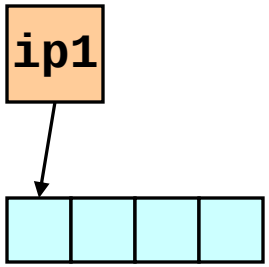
- When dynamic memory is never deallocated, we have a **memory leak**.

Dynamic Allocation [2/2]

The **new** and **delete** operators also have array forms.

```
ip1 = new int[4];

delete [] ip1;
```



Notes

- You cannot specify constructor parameters in the array version of **new**. Array items are always default constructed.
- Do not mix array & non-array versions. Use the proper form of **delete** for each **new**.