CS3157: Advanced Programming

Lecture #8

June 19

Shlomo Hershkop

shlomo@cs.columbia.edu

Outline

- More CPP
  - basics in depth
  - pointers and references differences
  - class abstractions
  - class inheritance
  - creating classes
  - long example
  - cgi and c/cpp
- Reading:
  - chapters 16, 17, 18 (c++, classes, overloading)

Announcements

- We wrap up the course next week
  - homework 2 released
  - lab due this week
  - will get last lab this week
  - will post sample final
  - need to run makeup class on friday

3

pass by references

- c: manipulates variables by reference through the use of pointers

- c++ introduces another way of manipulating references variables
  - reference parameters
  - &
  - easier to use than pointers

4

functions

```
void addTo(int a, int b)  {
  a = a+b;
}


void addToRef(int &a, int b){
  a = a+b;
}
```

5

careful

- in theory could return referenced variables
- but if not declared static, will return dangling pointers
- also like pointers, need to assign to make sense

```
int sum = 129;
int &refint = sum;
refint = 2;
```

6

Variable scope

- CPP allows you to specify scope through unary scope operator (::)
- So can differentiate between local and global variables

code

```
int count = 10;

int main(){
 int count = 5;

 // count is local
 // ::count is global
 // std::count is the same as 2
```

if then else

- ---- ? ----- : -------

- condition ? then : else

- (a ==5) ? a++ : a =0;

9

OOP

- we will talk later today about why everything has been shifting to OOPD

- when you use OOD you end up working with objects and states

- Lets say you need to manipulate fractions, how would you do this in basic c ?
  - do you have a more elegant solution ?

- Lets talk about it in C++

10

Functions organization

- You've programmed classes in Java
- What kind of functions exist with well designed classes

11

Functions

- Accessor
  - get some state information from the object
- Mutator
  - change information
- Helper
  - internal functions to accomplish tasks cleanly
- Predicate
  - help answer simple yes/no questions

12

CPP classes

- A class if a collection of functions and variables
- In CPP we have constructors and destructors

- Anyone remember how to define a constructor ?
- destructor ?
- when are they invoked ?

13

Example

```
class Counter {
public:
  int x;
  void print() { cout << x
  <<endl;
  }
}
```

14

accessing variables

- `Count mycounter;`
- `mycounter.x = 7;`
- `mycounter.print();`

- `Counter *countPTR;`
- `counterPTR->print();`

15

abstraction

- important when defining a class to separate how to use the class and how we are representing the information

- how can we fix the count class ??

16

Example

```
class Counter {
private:
  int x;
public:
  void setCount(int newcnt) {
     x = newcnt;
  }
  void print() { cout << x <<endl;
  }
}
```


issues

- being careful not return private references

Practice

- code the counter class

- add a static member ID (you need myid)

- int Foo::ID =0;
  - in global scope

Hands on Coding

- Please code a Fraction Class
- main should look like:

```
int main(void) {
cout<<“start”<<endl;
Fraction f1;
cout<<“End”<<endl;
  return 0;
}
```

- add constructor/destructor
- add print to them and see what it outputs
- add a global fraction
- now add a pointer to a fraction
  - what happened to the destructor ?

21

- what if we wanted to keep roman numerals as a counter ?

22

Example II

```
class Counter {
private:
  char * x;
  char * convertInt(int number);
public:
  Counter() { … }
  ~Counter { … }
  void setCount(int newcnt) {
      x = convertInt(x);
  }
  void print() { cout << x <<endl;
  }
}
```

23

ok so far….

- when we have an int as a counter, its easy to say lets add one to the counter

- `Counter c;`
- `c.x = 4;`
- `c.x++;`

24

assignment

- by default = will assign all variables (simple) one by one

```
Count a;
Counter b;
a.setCount(19);
b = a;
b.print();
```

25

question

- so how do you do this with a roman numeral counter ?

26

background

- to answer that easily need to cover some background material

27

- `const` keyword
  - there are times when we want to ensure that our program will not change a specific variables value.
  - variables can be declared const
  - const int x =3;
  - const Count c(13);
  - functions need to be declared const when dealing with const variable members

28

const class members

- const class members are assigned at construction time using the : notation

```
class Worker {
public:
   Worker(int id,int job);
   int getID () const;
private:
   const int _ID;
   int _job;
}
```

29

constructor

```
Worker(int id, int job) : _ID(id) {
   _job = job;
}
```

30

Classes within classes

- class member variables can be other classes
- important: member constructors are actually called before main class constructors
  - does this make sense ?

31

this

- this is a keyword
- represents a pointer to the class itself
- this->x
- or (*this).x

32

static

- static members have instance wide scope and livability
- great for shared variable
- have to be careful how used

33

assert

- special macro runs a test
- if true continues
- if false
  - dies without calling destructors

34

friends

- can declare a function to be a friend
- allows access to private member of the class
- not scoped during definition

35

Order of running program

- In c we saw that the program always starts from main
- As mentioned in class this is different in cpp

36

What can go wrong

- The good thing about cpp is that your program can now crash many times even before reaching main ☺

37

Ordering and where to look for problems

- Global variables
  - Assignments and constructors
  - What else ??
- Main
- Local variables
- End local variables
- End main
- Global destructors

38

code

- I’d like to cover a bunch of code examples now illustrating the power of classes
- Will start from simple array and work out a complex class
- Then start on a simple data structure

Class friends

- allows two or more classes to share private members
- e.g., container and iterator classes
- friendship is not transitive

Operator overloading

- Most operators can be overloaded in cpp
- Treated as functions
- But its important to understand how they really work

41

- +
- ~
- -
- !
- =
- *
- /=
- +=
- <<

- >>
- &&
- ++
- []
- ()
- new
- delete
- new[]
- ->
- >>=

Look up list

42

Operators which can not be overloaded

- .
- .*
- ::
- ?:
- sizeof

43

- X = X + Y
- Need to overload
  +
  =
- But this doesn't overload +=

44

- Functions can be member or non-member
- Non-member as friends
- If its member, can use this
- (), [], -> or any assignments must be class members

- When overloading need to follow set function signature

45

- Code from fig18_03 (c book)

- Will cover next class in depth

46

unary

- Y += Z
- Y.operator+=(Z)

- ++D
- member
  – D.operator++()
- Non member
  – operator++(D)

47

Next

– Software engineering
  - Will cover most in class, you are responsible for understanding high level overview

48

What is Software Engineering?

- Stephen Schach: “Software engineering is a discipline whose aim is the production of fault-free software, delivered on time and within budget, that satisfies the user’s needs.”
- includes:
  - requirements analysis
  - human factors
  - functional specification
  - software architecture
  - design methods
  - programming for reliability
  - programming for maintainability
  - team programming methods
  - testing methods
  - configuration management

49

People

- you can’t do everything yourself
- e.g., your assignment: “write an operating system”
- where do you start?
- what do you need to write?
- do you know how to write a device driver?
- do you know what a device driver is?
- should you integrate a browser into your operating system?
- how do you know if it’s working?

50

Why

- in school, you learn the mechanics of programming
- you are given the specifications
- you know that it is possible to write the specified program in the time allotted
- but not so in the real world...
  - what if the specifications are not possible?
  - what if the time frame is not realistic?
  - what if you had to write a program that would last for 10 years?
- in the real world:
  - software is usually late, over budget and broken
  - software usually lasts longer than employees or hardware
- the real world is cruel and software is fundamentally brittle

51

Who

- the average manager has no idea how software needs to be implemented
- the average customer says: “build me a system to do X”
- the average layperson thinks software can do anything (or nothing)
- most software ends up being used in very different ways than how it was designed to be used

52

Time

- you never have enough time
- software is often under budgeted
- the marketing department always wants it tomorrow
- even though they don't know how long it will take to write it and test it
- "Why can't you add feature X? It seems so simple..."
- "I thought it would take a week..."
- "We've got to get it out next week. Hire 5 more programmers..."

53

Complexity

- software is complex!
- or it becomes that way
  - feature bloat
  - patching
- e.g., the evolution of Windows NT
  - NT 3.1 had 6,000,000 lines of code
  - NT 3.5 had 9,000,000
  - NT 4.0 had 16,000,000
  - Windows 2000 has 30-60 million
  - Windows XP has at least 45 million...

54

Necessity

- you will need these skills!

- risks of faulty software include
  - loss of money
  - loss of job
  - loss of equipment
  - loss of life

55

Therac-25

- http://sunnyday.mit.edu/papers/therac.pdf

- therac-25 was a linear accelerator released in 1982 for cancer treatment by releasing limited doses of radiation

- it was software-controlled as opposed to hardware-controlled (previous versions of the equipment were hardward-controlled)

- it was controlled by a PDP-11; software controlled safety

- in case of error, software was designed to prevent harmful effects

56

- BUT
- in case of software error, cryptic codes were displayed to the operator, such as:
- “MALFUNCTION xx”
- Where 1 < xx < 64

- operators became insensitive to these cryptic codes
- they thought it was impossible to overdose a patient
- however, from 1985-1987, six patients received massive overdoses of radiation and several died

57

- main cause:
- a race condition often happened when operators entered data quickly, then hit the up-arrow key to correct the data and the values were not reset properly

- the manufacturing company never tested quick data entry— their testers weren’t that fast since they didn’t do data entry on a daily basis

- apparently the problem had existed on earlier models, but a hardware interlock mechanism prevented the software race condition from occurring

- in this version, they took out the hardware interlock mechanism because they trusted the software

58

Example2: Ariane 501

- next-generation launch vehicle, after ariane 4
- presigious project for ESA
- maiden flight: june 4, 1996
- inertial reference system (IRS), written in ada
  - computed position, velocity, acceleration
  - dual redundancy
  - calibrated on launch pad
  - relibration routine runs after launch (active but not used)
- one step in recalibration converted floating point value of horizontal velocity to integer
- ada automatically throws out of bounds exception if data conversion is out of bounds
- if exception isn't handled... IRS returns diagnostic data instead of position, velocity, acceleration

59

- perfect launch
- ariane 501 flies much faster than ariane 4
- horizontal velocity component goes out of bounds
- IRS in both main and redundant systems go into diagnostic mode
- control system receives diagnotic data but interprets it as wierd position data
- attempts to correct it...
- ka-boom!
- failure at altitiude of 2.5 miles
- 25 tons of hydrogen, 130 tons of liquid oxygen, 500 tons of solid propellant

60

- expensive failure:
  - ten years
  - $7 billion
- horizontal velocity conversion was deliberately left unchecked
- who is to blame?
- “mistakes were made”
- software had never been tested with actual flight parameters
- problem was easily reproduced in simulation, after the fact

61

Mythical man-month

- Fred Brooks (1975)
- book written after his experiences in the OS/360 design
- major themes:
  - Brooks’ Law: “Adding manpower to a late software project makes it later.”
  - the “black hole” of large project design: getting stuck and getting out
  - organizing large team projects and communication
  - documentation!!!
  - when to keep code; when to throw code away
  - dealing with limited machine resources
- most are supplemented with practical experience

62

No silver bullet

- paper written in 1986 (Brooks)
- “There is no single development, in either technology or management technique, which by itself promises even one order-of magnitude improvement within a decade of productivity, in reliability, in simplicity.”
- why? software is inherently complex
- lots of people disagreed, but there is no proof of a counter-argument
- Brooks’ point: there is no revolution, but there is evolution when it comes to software development

63

SE Mechanics

- well-established techniques and methodologies:
  - team structures
  - software lifecycle / waterfall model
  - cost and complexity planning / estimation
  - reusability, portability, interoperability, scalability
  - UML, design patterns

64

Team Structures

- why Brooks' Law?
  - training time
  - increased communications: pairs grow by
- while people/work grows by
  - how to divide software? this is not task sharing
- types of teams
  - democratic
  - "chief programmer"
  - synchronize-and-stabilize teams
  - eXtreme Programming teams

65

Lifecycles

- software is not a build-one-and-throw-away process
- that's far too expensive
- so software has a lifecycle
- we need to implement a process so that software is maintained correctly
- examples:
  - build-and-fix
  - waterfall

66

Software lifestyle cycle

- 7 basic phases (Schach):
  - requirements (2%)
  - specification/analysis (5%)
  - design (6%)
  - implementation (module coding and testing) (12%)
  - integration (8%)
  - maintenance (67%)
  - retirement
- percentages in ()'s are average cost of each task during 1976-1981
- testing and documention should occur throughout each phase
- note which is the most expensive!

67

Requirements

- what are we doing, and why?
- need to determine what the client needs, not what the client wants or thinks they need
- worse— requirements are a moving target!
- common ways of building requirements include:
  - prototyping
  - natural-language requirements document
- use interviews to get information (not easy!)
- example: your online store

68

Specifications

- the “contract”— frequently a legal document
- what the product will do, not how to do it
- should NOT be:
  - ambiguous, e.g., “optimal”
  - incomplete, e.g., omitting modules
  - contradictory
- detailed, to allow cost and duration estimation
- classical vs object-oriented (OO) specification
  - classical: flow chart, data-flow diagram
  - object-oriented: UML
- example: your online store

69

Design Phase

- the “how” of the project
- fills in the underlying aspects of the specification
- design decisions last a long time!
- even after the finished product
  - maintenance documentation
  - try to leave it open-ended
- architectural design: decompose project into modules
- detailed design: each module (data structures, algorithms)
- UML can also be useful for design
- example: your online store

70

Implementation

- implement the design in programming language(s)
- observe standardized programming mechanisms
- testing: code review, unit testing
- documentation: commented code, test cases
- integration considerations
  - combine modules and check the whole product
  - top-down vs bottom-up ?
  - testing: product and acceptance testing; code review
  - documentation: commented code, test cases
  - done continually with implementation (can't wait until the last minute!)
- example: your online store

71

Maintenance Phase

- defined by Schach as any change
- by far the most expensive phase
- poor (or lost) documentation often makes the situation even worse
- programmers hate it
- several types:
  - corrective (bugs)
  - perfective (additions to improve)
  - adaptive (system or other underlying changes)
- testing maintenance: regression testing (will it still work now that I've fixed it?)
- documentation: record all the changes made and why, as well as new test cases
- example: your on-line store— how might the system change once it's been implemented?

72

Retirement phase

- the last phase, of course

- why retire?
  - changes too drastic (e.g., redesign)
  - too many dependencies (“house of cards”)
  - no documentation
  - hardware obsolete

- true retirement rate: product no longer useful

73

Planning and Estimation

- we still need to deal with the bottom line
  - how much will it cost?
  - can you stick to your estimate?
  - how long will it take?
  - can you stick to your estimate?

- how do you measure the product (size, complexity)?

74

Reusability

- impediments:
  - lack of trust
  - logistics of reuse
  - loss of knowledge base
  - mismatch of features
- how to:
  - libraries
  - APIs
  - system calls
  - objects (OOP)
  - frameworks (a generic body into which you add your particular code)

75

Portability

- Java and C#
- Java: uses a JVM
  - write once, run anywhere (sorta, kinda)
- C#: also uses a JVM
  - emphasizes mobile data rather than code
- winner?
  - betting against Microsoft is historically a losing proposition...

76

interoperability

- e.g., CORBA
- define abstract services
- allow programs in any language to access services in any language in any location
- object-ish

77

Scalability

- something to keep in mind
- don't worry about scaling beyond the abilities of the machine
- avoid unnecessary barriers
- from single connection to forking processes to threads...

78

homework

- homework 2 is out
- larger project
- not enough time ☺
- Software engineering solution ??

79

PANIC!!

80

- just kidding
- sit down and plan out the classes
- start to code them out
- run small tests (can even code this separately and rerun every once in a while)

Next class

- wrap up lab
- reading
- reading
- coding hw2