Announcement

- No Lab this Week
- Will be meeting in the class (Nov 29)
- Groups have been emailed
  - Feel free to start early
C++

- class inheritance
- Code Examples

Software Engineering

Reading:
- chapters 16, 17, 18 (C++, classes, overloading)

Constructors

In C++ a class has a default constructor only if you don’t define any

class Example1 {
    public:
        int a, b, c;
        void multiply (int n, int m) {
            a=n; b=m; c=a*b;
        }
};
Example

- So can say `Example1 ex;`
- Which will create a constructor for you

- As soon as you define any constructor (say taking one arg) the above line stops working

- Blame the compiler

surprise

- By default
- Compiler will create
  - copy constructor
  - copy assignment operator
  - default destructor.
Differences

- Should understand the difference between
  - X
  - *X
  - &X
  - X.Y
  - X->Y
  - (*X).Y

friendship

- Functions (which you’ve seen) and classes which can access private class members
- So you’ve friend functions for overloading’
- Here is an example of friend class
// friend class
#include <iostream>
using namespace std;

class CSquare;

class CRectangle { 
  int width, height;
  public:
    int area ()
    {return (width * height);}
    void convert (CSquare a);
  
  class CSquare { 
    private:
      int side;
    public:
      void set_side (int a) 
    {side=a;} 
    friend class CRectangle;
  
    void CRectangle::convert (CSquare a) 
    { 
      width = a.side;
      height = a.side;
    }
  
  int main () { 
    CSquare sqr;
    CRectangle rect;
    sqr.set_side(4);
    rect.convert(sqr);
    cout << rect.area();
    return 0;
  }
C++ hierarchy

- composition:
  - creating objects with other objects as members

- derivation:
  - defining classes by expanding other classes
  - like "extends" in java

```cpp
class SortedIntArray : public IntArray {
public:
  void sort();
private:
  int *sortBuf;
}; // end of class SortedIntArray
```

- "base class" (IntArray) and "derived class" (SortIntArray)
- derived class can only access public or protected members of base class

- public derivation means that users of the derived class can access the public portions of the base class

- private derivation means that all of the base class is inaccessible to anything outside the derived class

- private is the default
// derived classes
#include <iostream>
using namespace std;

class CPolygon {
protected:
    int width, height;
public:
    void set_values (int a, int b)
        { width=a; height=b; }
    
    class CRectangle: public CPolygon {
        public:
            int area ()
                { return (width * height); } }

    class CTriangle: public CPolygon {
        public:
            int area ()
                { return (width * height / 2); } }

    int main () { 
        CRectangle rect;
        CTriangle trgl;
        rect.set_values (4,5);
        trgl.set_values (4,5);
        cout << rect.area() << endl;
        cout << trgl.area() << endl;
        return 0;
    }
Access changes downwards

- Can change the access of the parent class

```c++
class B : protected A {
}
```

- This means that if A has a public method `print`, it is now protected in B

Multiple inheritance

- Can use comma separated list to show multiple inheritance
Class derivation

- encapsulation
  - derivation maintains encapsulation
  - i.e., it is better to expand IntArray and add sort() than to modify your own version of IntArray

- friendship
  - not the same as derivation!!
  - example:
    - B2 is a friend of B1
    - D1 is derived from B1
    - D2 is derived from B2
    - B2 has special access to private members of B1 as a friend
    - But D2 does not inherit this special access
    - nor does B2 get special access to D1 (derived from friend B1)

Derivation and pointer conversion

- derived-class instance is treated like a base-class instance
- but you can't go the other way
- example:
  ```
  main() {
  IntArray ia, *pia; // base-class object and pointer
  StatsIntArray sia, *psia; // derived-class object and pointer
  
pia = &sia; // okay: base pointer -> derived object
  psia = pia; // no: derived pointer = base pointer
  psia = (StatsIntArray *)&ia; // sort of okay now since:

  // 1. there's a cast
  // 2. pia is really pointing to sia,
  // but if it were pointing to ia, then
  // this wouldn't work (as below)
  psia = (StatsIntArray *)&ia; // no: because ia isn't a StatsIntArray
  ```
Streams

- I/O from the system point of view, is simply a collection of bytes
- C++ allows you to access I/O on two levels
  - low level: very quick but it's your job to make sense of the bytes
  - high level: slower since it's converting the bytes to specific types

libraries

- `<iostream>`
  - cin
  - cout
  - cerr (unbuf)
  - clog (buf)
- `<istream>`
- `<ostream>`
- `<iomanip>`
fstream

- for file processing C++ uses ifstream and ofstream objects to read/write from files

- since formatting is automatically figured out for you, how do you print out the pointer address (i.e. %p) using cout ??
casting

- casting is your friend

```
<< static_cast<void*>(str_ptr) << endl;
```

```
cout.put('a');
  // will print single character to the output stream

can cascade

cout.put('a').put('b').put('c');
```
Examples

- 21_10
- 21_11

get

- one character at a time
- even whitespaces
- EOF
- different version of get
- 21_12
- 21_13
getline

- read till delim
- throws it out
- replaces it with \0

- 21_14

low level

- low level read/write

- 21_15
stream formatting

- In C it’s a headache and half to format the output
- C++ has lots of help in stream formatters

21_16
21_17

Templates

- The idea is to generalize your code as much as possible to deal with any class you will be handling
- Can allow your classes to store any type correctly (think void pointer)
- Allows a function to adopt to specific constraints
Templates

template<typename X>
void foo(X &first, X second){
  first += second;
}

- 22_1
templates

- can also take non general arguments
- `template<class T, int x> ...`
- integrates into inheritance
- allows friendly relationships

STL

- standard template library
- tons of useful stuff here

- Once do any serious C++ programming you should consider STL
  - they’ve worked out all the bugs 😊
  - very efficient
  - make sure you understand what you are doing
static members

- array<float> X;
- array<int> Y;

- if there are static variable members each gets separate copy while
- array<float> x2
  - will share with above X

responding to errors

- part of good object oriented programming is anticipating problems

- what can we do if something goes wrong??
- try
- catch

- similar to java
  - except won’t escalate if not caught unless specifically enclose in try->catch blocks
  - general exception class (<exception> and std::exception)
Software engineering

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

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
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?

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
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

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…”
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...

Necessity

- you will need these skills!

- risks of faulty software include
  - loss of money
  - loss of job
  - loss of equipment
  - loss of life
Therac-25


- 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 hardware-controlled)

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

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

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
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.

Example 2: Ariane 501

- next-generation launch vehicle, after Ariane 4
- prestigious 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
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 diagnostic data but interprets it as weird position data
attempts to correct it...
ka-boom!
failure at altitude of 2.5 miles
25 tons of hydrogen, 130 tons of liquid oxygen, 500 tons of solid propellant

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
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

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
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

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
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

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 documentation should occur throughout each phase
- note which is the most expensive!
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

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
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

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
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?

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
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)?

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)
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...

interoperability

- e.g., CORBA

- define abstract services

- allow programs in any language to access services in any language in any location

- object-ish
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...

homework

- homework 3 is now out
- larger project
- not enough time 😊
- Software engineering solution ??
PANIC!!

- just kidding
- sit down and plan out the classes
  - Your own code
  - Group wide view if any interaction
- start to code them out
- run small tests (can even code this separately and rerun every once in a while)
- coordinate