CS102 Unit 0a – Course Intro

Mark Redekopp
Introduction

• This is how we often see and interact with software
  – In truth we interact with it far more than we think
  – We are interacting with software when we drive, fly, turn on the lights, watch TV, go to the bank, or buy something with our credit card

• So what is it really?
Introduction

• This is how the movies think computers see software
  – The far right picture is reasonably accurate

• While all programs eventually end up as 1s and 0s, we generally program using some form of "high-level" or scripting language
Computer Abstractions

- Computer systems can be viewed as a layered stack of abstractions from basic HW to complex SW
- Assembly and machine code are the fundamental instructions a computer processor can execute
  - Too low level
- Enter high level languages
  - More powerful and succinct descriptive abilities
- Because of how the hardware works, our software must be written using certain structures
  - This class is intended to teach you those programming structures.
This Class

• The goal of this class is two-fold
  – Teach you the basics of programming
  – Develop mathematical and algorithmic thinking skills needed to excel in future courses
Course Structure

- The course is broken into 4 units each consisting of:

  1. **Lectures**
  2. **Lab** (Tools + Practice + Teamwork)
  3. **Portfolio** (Self-selected programming problems)
  4. **Homework** (Individual practice and programming assignments)

### Units

1. **Scalar Processing (Expressions & Conditionals)**
2. **Linear (1D) Processing (Loops)**
3. **Multidimensional Processing (Nested Loops)**
4. **Divide & Conquer (Functions and Abstraction)**
Exams and Grading

- The course will utilize 3 exams during our Quiz section
  - **Midterm 1** – Oct. 6
  - **Midterm 2** – Nov. 3
  - **Final** – Dec. 9

- Grading will be as follows:

<table>
<thead>
<tr>
<th>Component</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Labs</td>
<td>6%</td>
</tr>
<tr>
<td>Portfolio</td>
<td>6%</td>
</tr>
<tr>
<td>Homework</td>
<td>32%</td>
</tr>
<tr>
<td>Lowest Midterm</td>
<td>12%</td>
</tr>
<tr>
<td>Highest Midterm</td>
<td>22%</td>
</tr>
<tr>
<td>Final Exam</td>
<td>22%</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>100%</strong></td>
</tr>
</tbody>
</table>
Syllabus
Expectations

• Attend lectures & be engaged
  – Ask questions
  – We're a team...I need you!
  – I'll give you my best. Try to give me yours!

• Catch the wave!
  – Start assignments early, schedule weekly practice time, read and review other sources of input
20-Second Timeout

• Who Am I?
  – Teaching faculty in EE and CS
  – Undergrad at USC in CECS
  – Grad at USC in EE
  – Work(ed) at Raytheon
  – Learning Spanish (and Chinese?)
  – Sports enthusiast!
    • Basketball
    • Baseball
    • Ultimate Frisbee?
Programming Languages 1

• Declarative Languages
  – Describe the what but not the how
  – Examples: HTML, CSS
Programming Languages 2

- Imperative/Structured Languages
  - Describe the what (data) and how (instructions/algorithm)
  - Examples: C/C++, Java, Javascript, Python (which I'll use today)
  - The focus of most programming courses
  - Programs are like a recipe for how to operate on data

```python
import math

a = int(input("Enter a: ")
b = int(input("Enter b: "))
c = int(input("Enter c: "))

det = b*b - 4*a*c

if(det >= 0):
    r1 = (-b - math.sqrt(det)) / (2*a)
    r2 = (-b + math.sqrt(det)) / (2*a)
    print(f'Roots are {r1} and {r2}')
else:
    print("Imaginary roots")
```
High Level Languages

Just like half of the world's spoken tongues, most of the 2,300-plus computer programming languages are either endangered or extinct. As powerhouse C/C++, Visual Basic, Cobol, Java and other modern source codes dominate our systems, hundreds of older languages are running out of life.

As a dead collection of engineers-electronic lexicographers, if you will, all the moment to save, or at least document the lingoes of classic software. They're combing the globe's 9 million developers in search of coders still fluent in these nearly forgotten tongues. Among the most endangered are Ada, APL, B (the predecessor of C), Lisp, Oberon, Smalltalk, and Simula.

Code-caker Grady Booch, Rational Software's chief scientist, is working with the Computer History Museum in Silicon Valley to record and, in some cases, maintain languages by writing new compilers so our ever-changing hardware can grok the code. Why bother? "They tell us about the state of software practice, the minds of their inventors, and the technical, social, and economic forces that shaped history at the time," Booch explains. "It will provide the raw material for software archaeologists, historians, and developers to learn what worked, what was brilliant, and what was an utter failure." Here's a peek at the strongest branches of programming's family tree. For a nearly exhaustive rundown, check out the Language List at [HTTP://WWW.INFORMATIK.UMI-BREMIG.DE/JAVAMISC/LANG_LIST.HTML]. - Michael Mendes

![Computer Languages Chart](http://www.digibarn.com/collections/posters/tongues/ComputerLanguagesChart-med.png)

Sources: Paul Budkin; Brent Hoberman, associate director of computer science at IBM Research; The Retrocomputing Museum; Todd Proebsting, senior researcher at Microsoft; Gino Wiederhold, computer scientist, Stanford University

© 2023 by Mark Redekopp. This content is protected and may not be shared, uploaded, or distributed.

http://www.digibarn.com/collections/posters/tongues/ComputerLanguagesChart-med.png
Why C++

• C++ is used widely
• C++ is "close" to the hardware (HW)
  – Makes it fast
  – Makes it flexible (Near direct control of the HW)
  – Makes it dangerous (Near direct control of the HW)
  – In fact, many other languages are themselves written in C/C++
• Because if you learn C++ you can likely learn MOST languages very quickly
• Because that's what we use in CS 103
What Language Aspects Will We Learn?

• Programming skills in C/C++
  – Overlaps with the first 20% of CS 103
  – Data Representation
  – Basics of discrete mathematics
  – Expressions
  – Conditional Statements
  – Iterative Statements (Loops)
  – Functions
  – Arrays

• Problem solving using common programming 'idioms'
More than just "Coding"...

<table>
<thead>
<tr>
<th>Level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specification</td>
<td>• A precise problem statement to capture what the application requires (often requires the designer to make choices)</td>
</tr>
<tr>
<td>Problem Solving</td>
<td>• Understanding specification</td>
</tr>
<tr>
<td></td>
<td>• Planning, especially partitioning into sub-problems</td>
</tr>
<tr>
<td></td>
<td>• Identifying and using appropriate idioms</td>
</tr>
<tr>
<td></td>
<td>• Solving difficult sub-problems</td>
</tr>
<tr>
<td></td>
<td>• Writing &quot;glue code&quot; to tie everything together</td>
</tr>
<tr>
<td>Idioms</td>
<td>• Simple programming patterns/templates for solving specific tasks that can be used to connect your problem solving approach to actual code</td>
</tr>
<tr>
<td>Semantics</td>
<td>• Meaning of a program or any of its parts</td>
</tr>
<tr>
<td>Syntax</td>
<td>• Rules/grammar of the language</td>
</tr>
</tbody>
</table>
Problem Solving Idioms

- An idiom is a colloquial or common mode of expression
  - Example: "raining cats and dogs"
- Programming has common modes of expression that are used quite often to solve problems algorithmically
- We have developed a repository of these common programming idioms. We STRONGLY suggest you...
  - Reference them when attempting to solve programming problems
  - Familiarize yourself with them and their structure as we cite them until you feel comfortable identifying them

**Rule / Exception Idiom**

- **Name**: Rule/Exception
- **Description**: Perform a default action and then use an `if` to correct an exceptional case
- **Structure**: Code for some default action (i.e. the rule) is followed by code for the exceptional case

```c
// Default action
if( /* Exceptional Case */ )
{
    // Code for exceptional case
}
```

- **Example(s):**
  - Base pay plus bonus for certain exceptional employees

```c
bool earnedBonus = /* set somehow */;
int bonus = /* set somehow */;

int basePay = 100;
if( earnedBonus == true )
{
    basePay += bonus;
}
```

- **Notes**: This can be implemented with an `if/else` where an `else` implements the other.
STARTING TO THINK LIKE A COMPUTER
It’s A Numbers Game

• **Fact 1**: Everything in a computer is a number
  – Sure. Things like 102 and 3.9 are numbers
  – But what about text and images and sound?
  – Everything!

• **Fact 2**: Computers can only work with or "see" 1 or 2 numbers at a time (i.e. they can only do 1 thing at a time)

• Humans process information differently
  – Therein lies some of the difficulty of learning programming

```plaintext
import math

a = int(input("Enter a: "))
b = int(input("Enter b: "))
c = int(input("Enter c: "))
det = b*b - 4*a*c
if(det >= 0):
    r1 = (-b - math.sqrt(det)) / (2*a)
r2 = (-b + math.sqrt(det)) / (2*a)
print(f'Roots are {r1} and {r2}')
else:
    print("Imaginary roots")
```

Combine 2c. Flour Mix in 3 eggs
Example (1)

• What do you see?
  – The letter 'a'!

• What does the computer see?
  – A number; each text character is coded to a number
    • Example: Character map / Insert symbol
Text Representation

- Most common character code is ASCII (UTF-8)
- Every character, even non-printing, characters have a corresponding numbers
  - Decimal (base 10) / Hexadecimal (base 16)

https://www.commfront.com/pages/ascii-chart
Example (2)

• What do you see?
  – A circle!

• What does the computer see?
  – Coordinate pairs of each "pixel"
  – ...or...
  – \( r = 120 \); origin = (10, 14)
  – Computer has to enumerate and visit each location and color it black
Example (3)

• What do you see?
  – A man's face!

• What does the computer see?
  – Many numbers (aka pixels)
  – Value corresponds to color

Image taken from the photo "Robin Jeffers at Ton House" (1927) by Edward Weston
The Connection with Mathematics

• Brightness
  – Each pixel value is increased/decreased by a constant amount
  – \( P_{\text{new}} = P_{\text{old}} + B \)
    • \( B > 0 \) = brighter
    • \( B < 0 \) = less bright

• Contrast
  – Each pixel value is multiplied by a constant amount
  – \( P_{\text{new}} = C \times P_{\text{old}} + k \)
    • \( C > 1 \) = more contrast
    • \( 0 < C < 1 \) = less contrast

• Same operations performed on all pixels
"Enough" is NOT enough

• As we program we must be explicit
  – Example: drawing the circle on the screen

• Being general is not sufficient; we must be explicit!
  – Imagine a recipe for cinnamon rolls that simply read:
    • Mix and bake the following: butter, that white powdery baking substance, eggs, just enough sugar, and cinnamon. Enjoy!
    – How much of each, how much is "enough", how long, in what order?

• We will try to work on some of discrete math skills that help us explicitly define and analyze our programs
Integers and floating-point types; Division and modulus operations

WORKING WITH NUMBERS IN C++
Data Types

• How should the numbers (actually the bits: 1s and 0s) the computer is storing be interpreted: as a letter, an integer (aka an 'int'), a number with decimals (aka 'floating point' or 'double')

• C/C++ types help tell the computer hardware how to interpret the bits/numbers being stored in computer memory and what circuits to use to process them

• Let's learn the first two C++ data types:
  – int – integers only; no decimals (e.g. 4750, -18, 1908734)
  – double – very large numbers all the way down to very small fractions (e.g. 6.02E23, 1.5, -0.000248)
Division

• Computers perform division differently based on the types used as inputs

  • **Integer** Division:
    – When dividing two integer values, the result will also be an integer (any remainder/fraction will be dropped)
      - 10 / 4 = 2      52 / 10 = 5      6 / 7 = 0

  • **Floating-point** (Double) & Mixed Division
    – 10.0 / 4.0 = 2.5      52.0 / 10 = 5.2      6 / 7.0 = 0.8571
    – Note: If one input is a double, the other will be promoted temporarily (aka *implicitly "casted"*) to compute the result as a double
Modulus

• Dividing two integers yields an integer quotient
• Using the modulus operator (%) will divide two integers but yield the remainder!
• Examples:

\[
\begin{align*}
7 \div 3 &= 2 \quad &\text{but} \quad 7 \mod 3 &= 1 \\
75 \div 10 &= 7 \quad &\text{but} \quad 75 \mod 10 &= 5 \\
27 \div 4 &= \_\_ \quad &\text{but} \quad 27 \mod 4 &= \_\_ \\
59 \div 12 &= \_\_ \quad &\text{but} \quad 59 \mod 12 &= \_\_
\end{align*}
\]
HOMEWORK AND SURVEY
Modulus

- Dividing two integers yields an integer quotient
- Using the modulus operator (\%) will divide two numbers but yield the remainder!
- Examples:
  - $7 \div 3 = 2$ but $7 \% 3 = 1$
  - $75 \div 10 = 7$ but $75 \% 10 = 5$
  - $27 \div 4 = 6$ but $27 \% 4 = 3$
  - $59 \div 12 = 4$ but $59 \% 12 = 11$