Introduction to Systems Programming
מבעת لتוכנות מערכות

234122

Slides by: Gill Barequet, Gershon Elber, Omer Strulovich and Ron Rubinstein, Chaim Gotsman, Yechiel Kimchi, Yossi Gil, Eliezer Kantorowitz, Ayal Itzkoviz, Dani Kotlar and Itai Sharon.
Course Objectives

- Learn to develop large scale software
- Learn to work in UNIX environment
- Extend knowledge of C
- Meet C++
- Learn about object oriented programming
13 lectures, 13 tutorials

Grading Policy
- 70% - final exam (must get at least 55 in the final exam),
- 30%- HW assignments.

4 HW assignments
- Unix (1%),
- C (12%),
- Python (5%) and
- C++ (12%).
<table>
<thead>
<tr>
<th>Lectures</th>
<th>Tutorials</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
</tr>
<tr>
<td></td>
<td>Project management</td>
</tr>
<tr>
<td>2</td>
<td>Advanced C programming</td>
</tr>
<tr>
<td>3</td>
<td>Advanced C programming</td>
</tr>
<tr>
<td>4</td>
<td>Abstract Data Types (ADTs)</td>
</tr>
<tr>
<td>5</td>
<td>Intro to Python</td>
</tr>
<tr>
<td>6</td>
<td>C++: Basics</td>
</tr>
<tr>
<td>7</td>
<td>C++: Classes</td>
</tr>
<tr>
<td>8</td>
<td>C++: Overloading/static members</td>
</tr>
<tr>
<td>9</td>
<td>C++: Inheritance</td>
</tr>
<tr>
<td>10</td>
<td>C++: Templates</td>
</tr>
<tr>
<td>11</td>
<td>C++: STL</td>
</tr>
<tr>
<td>12</td>
<td>C++: Exceptions</td>
</tr>
<tr>
<td>13</td>
<td>Advance topics: STL, C++ casts, Finding memory leaks</td>
</tr>
</tbody>
</table>
Programs

- Typically 10-1,000 lines.
- Developed by an individual.
- Typically used by the programmer.
- Might run first time.
- Can be coded immediately.
- Requires debugging.
- No standards required.
- **Relatively Easy** !!

**Example: A sorting program**

Software

- Typically at least 10,000 lines.
- Developed by a team.
- Used by “customers”.
- *Never* runs first time.
- Requires analysis/design.
- Requires long-term maintenance.
- Standard compliant.
- **Difficult** !!

**Example: A WEB browser**

**Consequences**
What makes software **good**?

- **Fulfilling requirements**
  - Determining software requirements is not the subject of this course

- **Easy to maintain:**
  - Fix bugs
  - Add new features
  - Not breaking old code!
  - Portable

- **code is correct**
- **code is good**
What makes code good/bad?

- **Simple** to read/manage
- No **code duplication**
- Can be **tested** easily
- Errors are **found early** and simple to isolate
- **Portable**
- **Efficient**, but will not be discussed here
- **Modularity**
```c
void printBestStudent(char* names[], int n, double* grades[], int gradeCounts[])
{
    int i, j;
    char* bestStudent;
    double gradeSum, bestAvg, currAvg;

    if (n <= 0)
        return;

    gradeSum = 0.0;
    for (i = 0; i < gradeCounts[0]; i++) {
        gradeSum += grades[0][i];
    }
    bestAvg = gradeSum / gradeCounts[0];
    bestStudent = names[0];

    for (i = 1; i < n; i++) {
        gradeSum = 0.0;
        for (j = 0; j < gradeCounts[i]; j++) {
            gradeSum += grades[i][j];
        }
        currAvg = gradeSum / gradeCounts[i];
        if (currAvg > bestAvg) {
            bestStudent = names[i];
            bestAvg = currAvg;
        }
    }

    printf("The student with the highest average is %s", bestStudent);
}
```
void printBestStudent(Student* students[], int size)
{
    int i, j;
    double gradeSum, bestAvg, currAvg;
    Student* bestStudent;

    if (size <= 0)
        return;

    gradeSum = 0.0;
    for (i = 0; i < students[0]->numGrades; i++) {
        gradeSum += students[0]->grades[i];
    }
    bestAvg = gradeSum / students[0]->numGrades;
    bestStudent = students[0];

    for (i = 1; i < size; i++) {
        gradeSum = 0.0;
        for (j = 0; j < students[i]->numGrades; j++) {
            gradeSum += students[i]->grades[j];
        }
        currAvg = gradeSum / students[i]->numGrades;
        if (currAvg > bestAvg) {
            bestStudent = students[i];
            bestAvg = currAvg;
        }
    }

    printf("The student with the highest average is %s", bestStudent->name);
}
```c
void printBestStudent(Student* students[], int size)
{
    int i, j;
    double gradeSum, bestAvg, currAvg;

    if (size <= 0)
        return;

    gradeSum = 0.0;
    for (i = 0; i < students[0]->numGrades; i++) {
        gradeSum += students[0]->grades[i];
    }
    bestAvg = gradeSum / students[0]->numGrades;
    Student* bestStudent = students[0];

    for (i = 1; i < size; i++) {
        gradeSum = 0.0;
        for (j = 0; j < students[i]->numGrades; j++) {
            gradeSum += students[i]->grades[j];
        }
        currAvg = gradeSum / students[i]->numGrades;
        if (currAvg > bestAvg) {
            bestStudent = students[i];
            bestAvg = currAvg;
        }
    }

    printf("The student with the highest average is %s", bestStudent->name);
}
```

Déjà vu
void printBestStudent(Student* students[], int size)
{
    int i;
    double bestAverage, currentAverage;
    if (size <= 0)
        return;

    bestAverage = getStudentAverage(students[0]);
    Student* bestStudent = students[0];

    for (int i = 1; i < size; i++) {
        currentAverage = getStudentAverage(students[i]);
        if (currentAverage > bestAverage) {
            bestStudent = students[i];
            bestAverage = currentAverage;
        }
    }

    printf("The student with the highest average is %s", bestStudent->name);
}

double getStudentAverage(Student* student) {
    int i;
    double sum = 0.0;
    for (i = 0; i < student->numGrades; ++i) {
        sum += student->grades[i];
    }
    return sum / student->numGrades;
}
A (better) code example

```c
Student* getBestStudent(Student* students[], int size) {
    int i;
    if (size <= 0)
        return NULL;

    Student* best = students[0];
    for (i = 1; i < size; i++) {
        if (getStudentAverage(students[i]) > getStudentAverage(best)) {
            best = students[i];
        }
    }

    return best;
}

void printBestStudent(Student* students[], int size) {
    Student* best = getBestStudent(students, size);
    if (best != NULL) {
        printf("The student with the highest average is %s", best->name);
    }
}
```
Why **C/C++?**

- De-facto industrial standard.
- “Mother tongue” of Unix systems.
- Structured language.
- C++ is an object-oriented syntactic extension of C.
Why OOP?

- An **intuitive** concept.
- Enables **modularity** and **information hiding** (encapsulation).
- Makes maintenance **easier**.
Why UNIX/Linux?

- Multi-user/multi-tasking operating system.
- De-facto standard across multiple platforms.
- Powerful procedural user-interface.
- Stable.
Why UNIX/Linux?

- Can run on a variety of platforms.
- Good programming tools (e.g. text manipulation programs).
- The open systems philosophy.
Unix - History

• Originally developed during the late 1960s as a side project at AT&T Bell Labs (in PDP-7 assembly!)
  – by Ken Thompson and Dennis Ritchie et al.
• Re-written in C in 1973
• Adopted by universities and commercial purposes during the 1970s and 1980s
• Several flavors exist, including BSD, System V and the Unix-like Linux
• Originally a PUI (Programmable User Interface) based.
Unix – the very basics

• The Kernel of the Unix OS is operated via a special programmable user interface called a **shell**.

• Over the time, quite a few shells were developed: sh, ksh, csh, tcsh, bash.

• The shell commands offers a PUI and constitutes a scripting language.

• Shell commands are divided into **internal** and **external** commands. Examples of **internal** commands:
  – cd - move around directories.
  – set - defines env. variables, affecting shell behavior.
  – alias - allows one to define shortcuts to commands.
Unix – the very basics II

• If the shell does not recognize a command, it tries to find it in a directory in the path env. variable.

• Examples of such external commands:
  • ls - view directory contents.
  • gcc - a compilers for C programs.
  • man - view a manual for a command.
  • make - a tool to manage a project consisting of multiple files.
  • diff – Find differences between two text files.

Try “man gcc” and read and find what ‘–E’ does.
Basic project management I

• A project consisting of a single file can be simply compiled using an (external) shell command like:
  
gcc –o test test.c
  
creating, if compilation successful, ‘test’, which is a new external executable command.

• But what if my project consists of multiple C files?

• An external command called ‘make’ was developed to ease the management of large projects.

• make checks dependencies and take action only when a dependent was modified.

  – A .o (.obj under windows) depends on .c or .cpp (C++) file
    » The compile will be invoked to recreate a .o if a .c was modified.

  – An executable (.exe under windows) depends on .o files.
    » The executable will be recreated if at least one .o was changed.

Why is this useful?
Here is a very simple use of the **make** command:

- Consider a project with the following files:
  - file1.c, file2.c, project.c, project.h.
- The make dependency script (typically called ‘makefile’) can look like:

```bash
file1.o: file1.c project.h
    gcc -c -o file1.o file1.c # Compilation

file2.o: file2.o project.h
    gcc -c -o file2.o file2.c # Compilation

project.o: project.c project.h
    gcc -c -o project.o project.c # Compilation

project: file1.o file2.o project.o
    gcc -o project file1.o file2.o project.o # Link
```
Basic project management III

• But even make depends on:
  – The type of sources (C, C++, assembler, etc.) it manages.
  – The operating system it operates on.
  – The specific compiler(s) to be used, etc.

• Hence, **cmake** was invented to support make.
  – cmake automatically synthesizes makefiles for the **make** utility that confirm to the environment (i.e. O.S., found compilers, etc.) it detects.

• And also issues of **managing version of sources**:
  – Ever wanted to revert on some code changes you did?
  – Multiple files means multiple authors so we will want to track who has been doing what change.

• More on project management as the course progresses (mostly during tutorials).
The C Programming Language (2nd Edition, ANSI-C)
B. Kernighan and D. Ritchie, Prentice-Hall, 1988

A. Kelley and I. Pohl, Addison-Wesley Professional, 1998

Unix in a Nutshell (4th Edition)
A. Robbins, O'Reilly Media, Inc, 2005
The C++ Programming Language (3rd Edition)
B. Stroustrup, Addison-Wesley Longman, 1997

Effective C++: 55 Specific Ways to Improve Your Programs and Designs (3rd Edition)
S. Meyers, Addison-Wesley Professional, 2005

C++ Primer (4th Edition)
S. Lippman, J. Lajoie, B. Moo, Addison-Wesley Professional, 2005

Code Complete: A Practical Handbook of Software Construction 2nd edition
Steve McConnell, Microsoft Press, 2004
"Programs must be written for people to read, and only incidentally for machines to execute."

- Abelson and Sussman