

COSC345

Software Engineering

Instructors and Support People

- Andrew Trotman
 - Office: 123A, Owheo
 - Email: andrew.trotman@otago.ac.nz
 - No office hours, just drop in
- Department Kaiāwhina (Māori and Pacific Support)
 - Steven Mills
- Department Disability Support
 - Contact the main office.

Learning Outcomes

- This paper will give students practice in:
 - Designing a software system
 - Planning the development
 - Carrying out the development using appropriate tools
 - Evaluating their work
- Students will learn to use "good practices" including:
 - Version control
 - Static checking
 - Testing
 - Following industry or platform standards

Academic Integrity and Academic Misconduct

- Academic integrity means being honest in your studying and assessments. It is the basis for ethical decision-making and behaviour in an academic context. Academic integrity is informed by the values of honesty, trust, responsibility, fairness, respect and courage. Students are expected to be aware of, and act in accordance with, the University's Academic Integrity Policy.
- Academic Misconduct, such as plagiarism or cheating, is a breach of Academic Integrity and is taken very seriously by the University. Types of misconduct include plagiarism, copying, unauthorised collaboration, submitting work written by someone else (including from a file sharing website, text generation software, or purchased work) taking unauthorised material into a test or exam, impersonation, and assisting someone else's misconduct. A more extensive list of the types of academic misconduct and associated processes and penalties is available in the University's Student Academic Misconduct Procedures.

Academic Integrity and Academic Misconduct

- It is your responsibility to be aware of and use acceptable academic practices when completing your assessments. To access the information in the Academic Integrity Policy and learn more, please visit the University's Academic Integrity website at www.otago.ac.nz/study/academicintegrity, or ask at the Student Learning Centre (HEDC) or the Library, or seek advice from your paper co-ordinator.
- For further information:
- **Academic Integrity Policy**
 - <http://www.otago.ac.nz/administration/policies/otago116838.html>
- **Student Academic Misconduct Procedures**
 - <http://www.otago.ac.nz/administration/policies/otago116850.html>

What is Software Engineering?

- Building and maintaining large scale software projects
 - A branch of
 - Computer Science
 - Engineering
 - Management
- How to manage
 - People, processes, products
- Origins
 - Successes and (too often) failures

Course Outline

- 13 weeks
 - Lectures (Monday Noon, Wednesday 1:00pm)
 - Tutorials (Tuesday 9:00am, Thursday 9:00am)
 - Lab (Friday 2:00pm – 4:00pm)
 - So you can work on your project
- 18 point paper
 - So 180 hours throughout the semester
 - 6 hours per week for 13 weeks are class time.
 - So how many hours of your time should you spend on the paper?

Assessment

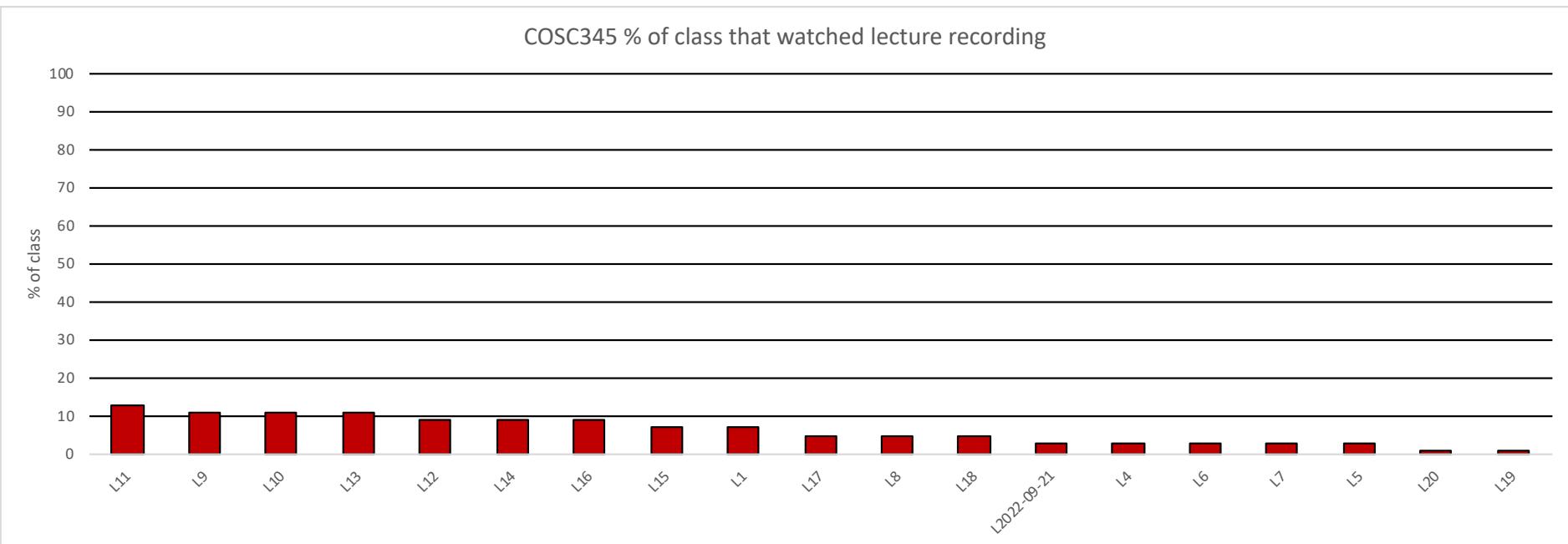
- Assessment
 - 50% internal assessment
 - 50% exam
- 4 assignments (10% each)
 - Group engineering project (teams of 3 or 4)
 - Due dates are:
 - 9 August, 30 August, 20 September, 11 October
 - Presentations (0% each):
 - 15 & 17 October
- Terms for attending tutorials (20 of 26 tutorials)
- Marks for group-work in lectures (20 of 24 lectures)
- Talk to me if you have extenuating circumstances

Resources

- Books
 - Recommended
 - S. McConnell, *Code Complete*, Microsoft Press, 1991
 - Additional recommended reading:
 - F. Brooks, *Mythical Man Month*
 - J. Bentley, *Programming Pearls*
 - J. Bentley, *More Programming Pearls*
 - S. Maguire, *Writing Solid Code*
 - I. Sommerville, *Software Engineering*

Resources

- Electronic
 - COSC345 Blackboard Website
 - Lecture notes and other material
 - Email (check this regularly)
 - Lecture recordings (no, you won't watch them, go to class)



Course Outline

- Little Languages (today)
- Team Organization
- Project Planning
- Requirements
- Waterfall
- Agile
- Communications
- Make
- Version Control
- Continuous Integration
- Literate Programming
- Static Analysis
- Code Review
- Debugging
- Profilers
- The Stack
- The Heap
- Memory Managers
- Memory Checkers
- Libraries
- Components
- Portability
- Ethics
- Project Presentations

Little Languages

What are Little Languages?

- Elegant interfaces to control programs
- Simple problem
 - Complex to code in a ‘regular’ language (e.g. Java)
 - Easily described in a ‘limited’ language
- Interactive problem
 - Texturally control a program
- Typical problems
 - Repetitive descriptions
 - Tricky manipulations
 - Control of hardware
- Just like methods and routines they move repetitive or complex tasks to one place

Why Little Languages?

- Simplify the problem
 - Reduce the chance of making errors
- Reduce repetition
 - Reduce the chance of making errors
- Easy to write
 - Compilers and interpreters are easy to write
 - Programs in little languages are easy to write
- Often line-based languages
- Easy to extend
- Reusable in many related projects

Example Little Languages

- *Make*
 - *Unix shell* / DOS command prompt / PowerShell
 - Lex / Yacc
 - *Python*
 - Perl
 - *LaTeX*
 - Awk
-
- You have probably used those marked in *italics*

Example: Multi-choice Program

- You have been approached by a lecturer about writing a program that will interactively conduct a multi-choice exam for the fictitious COMP103. The questions have just been written and delivered to you. Your task now is to write a program that conducts the exam.

Multi-choice Exam

1. In the context of COMP103, which of the below is an example of a run time error:
 - (A) a false start at a track and field event
 - (B) an overflowing bathtub
 - (C) leaving the cricket crease early
 - (D) stalling at the traffic lights
 - (E) an error in coding which compiles correctly but doesn't run

2. Which of the following statements about local variables are true:
 - (A) Local variables may have a specified visibility.
 - (B) Local variables can be declared as static.
 - (C) Local variables are defined within methods.
 - (D) Local variables are automatically initialised.
 - (E) None of them.

C Program

```
#include <stdio.h>
#include <ctype.h>
int main(int argc, char *argv[])
{
char buffer[10];
int correct = 0;
printf("1. In the context of COMP103, which of the below is an example of a run time
error:\n");
printf("\n");
printf("(A) a false start at a track and field event\n");
printf("(B) an overflowing bathtub\n");
printf("(C) leaving the cricket crease early\n");
printf("(D) stalling at the traffic lights\n");
printf("(E) an error in coding which compiles correctly but doesn't run\n");
printf("Answer\n");
fgets(buffer, sizeof(buffer), stdin);
if (toupper(*buffer) == 'E')
    correct++;
printf("\n");
printf("2. Which of the following statements about local variables are true:\n");
printf("(A) Local variables may have a specified visibility\n");
printf("(B) Local variables can be declared static\n");
printf("(C) Local variables are defined within methods\n");
printf("(C) Local variables are automatically initialised\n");
printf("(E) None of them\n");
fgets(buffer, sizeof(buffer), stdin);
if (toupper(*buffer) == 'C')
    correct++;
printf("Your score:%d", correct);
return correct;
}
```

Needs

- What does this program *need* to do?
 - Print text
 - Read answers
 - Mark (match) answers
 - Compute scores
- Ideal use of a little language! Perhaps use:
 - T: text output
 - A: answer input
 - M: match answers
 - C: compute values

Input / Output / Matching

T:

- : 1. In the context of COMP103, which of the
- : below is an example of a run time error:
- : (A) a false start at a track and field event
- : (B) an overflowing bathtub
- : (C) leaving the cricket crease early
- : (D) stalling at the traffic lights
- : (E) an error in coding which compiles correctly
- : but doesn't run

:

A:

M: E

Computing scores

C : #correct = 0

M :E

CY: #correct = #correct + 1

CN: #wrong = #wrong + 1

Whole Program

C: #correct = 0

T:

- : 1. In the context of COMP103, which of the below is an example of a run time error:
 - : (A) a false start at a track and field event
 - : (B) an overflowing bathtub
 - : (C) leaving the cricket crease early
 - : (D) stalling at the traffic lights
 - : (E) an error in coding which compiles correctly but doesn't run
- :

A:

M: E

CY:#correct = #correct + 1

T:

- : 2. Which of the following statements about local variables are true:
 - : (A) Local variables may have a specified visibility
 - : (B) Local variables can be declared static
 - : (C) Local variables are defined within methods
 - : (D) Local variables are automatically initialised
 - : (E) None of them
- :

A:

M: C

CY:#correct = #correct + 1

T: Your score:#correct

PILOT

- Programmed Inquiry, Learning, Or Teaching language
 - First developed in 1962
 - Standardized by IEEE 1991
- A little language for teaching
 - Multi choice questions
 - Short answer questions
 - Interactive learning sessions
- Versions released by Apple (and others)

Designing Little Languages

- Study the whole problem (regardless of the problem)
 - Think about how a little language might help
- Think about:
 - Abstraction
 - Identify the key components
 - Identify the key manipulations
 - Simplicity
 - Keep any language as simple as possible
 - Linguistic structure
 - What is the natural form of expression
 - Always allow comments

Language Design

- Completeness
 - Ensure all the necessary operations exist
 - Write several programs on paper first
 - Then write a compiler or interpreter
 - Ensure all the necessary restrictions exist
 - Make sure you can't write a program that does them
 - Add features dictated by use
- Orthogonality
 - Keep unrelated features unrelated
- Parsimony
 - Delete unneeded features

Language Design

- Generality
 - Use an operation for many purposes
- Similarity
 - Make the language suggestive
- Extensibility
 - Make it extendable
 - Allow the language to evolve
- Openness
 - Ensure related tools can be used with it
 - Use pre-existing tools to build it (Lex / Yacc / AWK)
 - Build on experience of using languages (Java, etc.)

References

- J. Bentley, *More Programming Pearls*, Chapter 9

COSC345

Software Engineering

Team Organization

Assignment

- Are you in a group?
- When is assignment-1 due?
- Do you have a schedule between now and then?
- When does your team meet?
- Do you have a github archive yet?

Outline

- Teams
 - Effective teams
 - Ineffective teams
 - Team composition
 - Task oriented
 - Talent oriented
 - Cohesiveness
 - Communications
 - Motivation
- Individuals
 - Skills to have
 - Getting staff

Teams

- A team is:
 - At least two people
 - Each with specific tasks
 - Working together for a common goal
 - And who have dependencies on each other
- A good team is:
 - Greater than the sum of its parts
- A bad team is:
 - Less than the weakest member
- Most significant software projects built by a team

Team Size

- Keep teams small (10 or fewer)
- Small teams often informal and emergent
 - The leader often emerges
 - Designs done by those with most skill
 - Tasks allocated by ability
 - Make decisions by consensus
- Large teams often appointed
 - Require more time in communication
 - Can be incohesive
 - Hard to manage

Effective Teams

- To be effective a team needs:
 - Direction
 - Agreed common goals
 - A supportive environment
 - Good communications
 - Mutual respect
 - Commitment
 - To the team
 - To the project
 - Individual roles

Effective Team Managers

- Ensure:
 - Realism
 - All deadlines must be realistic
 - Consistency
 - Everyone must feel valued for their contribution
 - Respect
 - Take into consideration people's strengths and weaknesses
 - All team members must be given the chance to contribute
 - Inclusion
 - People contribute when they feel heard
 - All people (junior and senior) must be comfortable speaking
 - Honesty
 - Tell the truth about what's going well and what's not
 - Defer to those of greater skill or knowledge

Ineffective Teams

- Bad leadership
- Failure to compromise or cooperate
- Lack of participation
- Procrastination
- Missed milestones
- Late deliverables
- Failed projects
- Bad management is one of the most significant contributors to project failure

Team Composition

- Teams must have complementary:
 - Personalities
 - Experience
 - Technical skills
- Leadership need not mean management
- No simple single answer
 - Task oriented
 - Team software process (Humphrey, 2000)
 - Talent oriented
 - The surgical team (Mills, 1971)
 - Mesh Management

Task Oriented

- Team leader
- Development manager
- Planning manager
- Quality / process manager
- Support manager

Relationship or Talent Oriented

- The surgeon (chief programmer)
- Co-pilot
- Administrator
- Editor
- Secretaries
- Program clerk
- Tool-smith
- Tester
- Language expert

Manage or Lead

- The job of a manager is to plan, organise, and coordinate
- The job of a leader is to inspire and motivate

Team Cohesiveness

- Team is more important than the members
- If the team is cohesive:
 - Team standards will emerge
 - Members will learn from each other
 - Members will be familiar with each other's work
 - Programs will become team property
 - Team goals become individual goals
- But...
 - Preference of democracy over leadership
 - Resistant to changes in leadership
- Try to increase cohesiveness
 - Take ownership of project
 - Run team building events (socials)

Team Communication

- Vital for effective team work
- But...
 - As size increases communications takes longer
 - For complete communications
 - $n*(n-1)$ messages
 - Communications must become “partial”
 - Managers often act as buffers
- Look out for
 - Personality clashes
 - Workspace inhibitors
 - Senior member domination
- Flat teams work better than structured teams

Team Motivation

- Your staff have basic human needs
 - Social
 - Provide places to meet and time to communicate
 - Esteem
 - Show people that they are valued
 - Self realization
 - Give responsibility of tasks to individuals
 - Provide mechanisms for people to develop their skills
 - Mental stimulation
 - Don't make people do things they don't want to (for too long)
 - Eating, drinking, sleeping
 - You are even responsible for this

Getting Staff

- Often you won't have a choice
- Some skills are hard to find (and so cost more)
 - DBA, GUI designers
- The “best” person may not have the “best” skills
 - Staff must interact with
 - Other staff
 - Managers
 - Clients
 - Suppliers and third parties

Individual's Skills to Consider

- Application domain experience
- Platform experience
- Programming experience
- Problem solving experience
- Educational background
- Communication skills
- Adaptability
- Attitude
- Personality
- Nature of the task and the individual

Negative Productivity

- “The best (top 5% or so) are ten times better than average, not better than bad. So an average ‘best’ programmer is worth 10 average programmers (at least). An average ‘best’ programmer is worth 100 average ‘bad’ programmers (or infinitely more if the bad ones have negative productivity which is the usual case.)“ (Scott)
- “It has been widely demonstrated that productivity of software developers may vary radically from person to person, and on average about one in five programmers has a negative productivity, i.e. she [*sic*] slows down others’ work, reducing the quantity/quality of the final outcome” (Pianon 2004 citing Feller & Fitzgerald 2002).

Pre-Hire Knowledge Building

- Standard methods
 - CVs are factual
 - Interviews are personable
 - Recommendations are often unreliable
- Testing
 - Such as:
 - Programming tests
 - Psychometric profiling
 - Aptitude tests
 - Usefulness often debated
- It's easier to spot bad than to spot good

References

- F. Brooks, *Mythical Man Month*, Chapter 3
- W. S. Humphrey, *Introduction to the team software process*.
- I. Sommerville, *Software Engineering*, Chapter 25

COSC345

Software Engineering

Requirements

Outline

- User requirements
- System requirements
- Functional requirements
- Non-functional requirements
- Problems
- Suggestions

Requirements

- A description of
 - The services to be provided
 - The constraints on the system
- Express
 - The needs of the customer for a system that helps in some way
- *Requirements engineering:*
 - Finding out requirements
 - Analyzing requirements
 - Documenting requirements
 - Checking the constraints on the system

Definition

- Requirements could be:
 - A high level description of roughly what's needed
 - A detailed formal definition of:
 - What the system will do
 - How it will do it
 - When it will do it
- If a company puts out a bid then:
 - The requirements could be the call
 - A successful contractor will then specify what they will build to match the requirements
- But, the project will change over time and the requirements cannot be “too rigid”

User Requirements

- What the user thinks they want
 - Natural language statements (and diagrams) of what services the user expects and any constraints that are present (like time)
- Should:
 - Include functional *and* non-functional requirements
 - Be understandable by the user base
 - Without technical (implementation) knowledge
 - Avoid design characteristics
- Potential Problems:
 - Lack of clarity
 - Lack of functional / non-functional separation
 - Incorrect amalgamation of several requirements into one
 - Requirements should be written to the audience
 - For example: An executive overview should be no longer than 1 page

System Requirements

- Also known as a *Functional Specification*
- Specify
 - How the user requirements are to be fulfilled
 - The external behavior of the system
 - What is going to be built
 - A precise and detailed description of the system's functions, services and constraints
 - May form part of the contract between builder and buyer
- It may be necessary to build a prototype to formalize the requirements
- Be careful with natural language
 - It is inconsistent and ambiguous

Functional Requirements

- Can be either user or system requirements
- Statements of services provided
 - What the system should do
 - What the system should not do
 - How the system will respond to input
- Problems lead to cost overruns and delays:
 - Imprecise specification
 - Leads to short-cuts
 - Ambiguous specification
 - Leads to easiest implementation
- Functional requirements should be:
 - Complete, precise, and consistent (... but they won't be)

Non-Functional Requirements

- Constraints on the software
 - Reliability
 - Security
 - Timing constraints (e.g. response times)
 - Resource constraints
 - CPU / disk usage / Internet bandwidth usage / etc.
 - Development process constraints
 - Including standards
- Apply to the whole system not a system function
- Imagine a 111 response call system
 - What kinds of functional and non-functional requirements do you expect?

Non-Functional Requirements

- Product requirements
 - Performance constraints
 - Memory / CPU usage
- Organizational requirements
 - Standard followed (ISO 9000)
 - Programming Language
 - Design methodologies
- External requirements
 - Interfaces (e.g. Z39.50)
 - APIs, data structures, and byte-ordering conventions
 - Legal / ethical requirements
- Non-functional requirements are difficult to verify

Quantitative Requirements

- If possible, all requirements should be verifiable
 - Functional requirements are already verifiable
 - “The system must [insert operation here]”
 - Non-functional requirements need careful writing
 - The system should be “fast”
 - How can this be verified?
 - The system must have a mean performance of 10 ops/sec
 - This can be verified
- Reword non-functional requirements so they can be verified – but not all can be
 - How can “must be maintainable” be quantified?

Domain Specific Requirements

- Those requirements specific to the domain
 - Language
 - Equations
 - Definitions
 - Protocols
 - Standards
- Acts as a reference to understand functional and non-functional requirements
- E.g. a location-based system needs GPS “stuff”

Structured Languages

- Requirements can be less ambiguous if a controlled vocabulary is used
- And / or a form might be used

| | |
|-------------|--|
| Function | Compute Insulin Dose. Safe sugar level |
| Description | Computed the dose of insulin to be delivered when the current measured sugar level is in the safe zone between 3 and 7 units |
| Inputs | Current sugar reading (r2), the previous two readings (r0 and r1) |
| Outputs | CompDose – the dose in insulin to be delivered |
| Description | Main control loop |
| Action | CompDose is 0 if the sugar level is stable or falling or if the level is increasing but the rate of increase is decreasing. If the level is increasing and the rate of increase is increasing then CompDose is computed by ... |
| ... | ... |

From Sommerville Figure 6.12

Form Based Requirements

- Include a description of:
 - Functionality
 - Inputs and outputs
 - Where the output goes
 - What is required for this component
 - Action to be taken
 - The procedural contract (pre and post conditions)
 - Side effects (if there are any)
 - Etc.

Software Requirements Document

- Official statement of what should be implemented
- Diverse reader base including:
 - Developers – build the system
 - Management – plan and manage the project
 - Test team – verify the software
 - Board members – pay for it
- Includes
 - User requirements
 - System requirements
- International standards
 - US Department of Defense
 - IEEE / ANSI 830-1998

IEEE / ANSI 830-1998

- Introduction
 - Purpose of the requirements document
 - Scope of the product
 - Definitions, acronyms and abbreviations
 - References
 - Overview of the remainder of the document
- General Description
 - Product perspective
 - Product functions
 - User characteristics
 - General constraints
 - Assumptions and dependencies
- Specific requirements
- Appendices
- Index

Problems

- Requirements that are “too rigid”
 - Constrain the project
 - Stifle innovation
 - Requirements without reason will be ignored!
- Requirements documents must be flexible
 - They may be self-contradictory
 - They may be functionally impossible
- Projects change over time
 - Requirements can change so rapidly that the documents are out of date before they are finished
- Over-detailed specification may take a long time to write
 - Longer than the system takes to build!
- Programming languages are formal specification languages

Suggestions

- Settle on a *standard* format
 - Omissions are unlikely if they become “habit”
 - Requirements are easier to check
- Include details of *whose* requirement it was
 - That individual can verify compliance
 - Consult that individual if changes are needed
- Separate *mandatory* from *desirable* requirements
 - Use consistent language (shall / should)
- Requirements should be written to the *audience*
 - E.g. executive overview

References

- I. Sommerville, *Software Engineering*, Chapter 6

Project Planning

COSC345
Software Engineering

Overview

- Success and Failure
- Software management activities
- Project plans
- Scheduling
- Plotting progress
 - Gantt and Pert charts
- Risks and risk management

Success and Failure

- Good management does not guarantee success
- Bad management nearly guarantees failure
 - Late deliverables
 - Cost overruns
 - Requirements failures
- Every project needs a “champion”
 - Someone who will fight for the project

Software Management is Hard

- The product is intangible – so monitor the project
 - Progress on a bridge is visible (you can *see* that)
 - Progress on software is not (you cannot *see* it)
 - What does 80% finished mean for software?
 - Can a product ever be finished?
- With software there are no standard processes
 - Building a bridge is a standard process
 - Building software is different each time
 - Some standard tools that can help
 - Some “rules of thumb” that can help
 - Software projects are usually unique
 - Prior experience may not be helpful
 - Technology changes make knowledge obsolete
 - Symbian / iOS / Android / Harmony OS?

Management Activities

- Requirements and proposal writing
 - Objectives, cost, and schedule estimates
- Project planning and scheduling
 - *Activities*
 - *Milestones*
 - Tangible achievements, e.g.:
 - Objects or libraries completed
 - Hardware needed for the project received
 - Fully staffed
 - *Deliverables*
 - Finished (deliverable) pieces, e.g.:
 - Documents
 - Pieces of software

Management Activities

- Project costing
 - Hardware
 - Software (compilers, libraries, debuggers, etc)
 - Staff (including yourself)
 - Consumables
- Project monitoring and reviews
 - Used in problem prediction
- Staff turnover management
 - Staff quality usually determined by price
 - Quality staff many not be available!
 - Training
- Report writing
 - For clients and upper management

Project Planning

- Software development plan
 - How and when the software will be developed
 - Foreseeable problems and solutions (*risks*)
 - Constituent parts of the software
 - Prerequisites
 - Hardware dependencies (e.g. embedded systems)
 - Software dependencies (e.g. third party libraries)
- Quality plan
 - Readability
 - Maintainability
 - Efficiency
- Validation plan
 - How the software will be shown to be valid

Project Planning

- Maintenance plan
 - Predicts maintenance requirements and costs
 - Changing software after delivery
 - Repair of faults (bug-fixing)
 - Increasing functionality
 - Adaptation to new environments (configurations)
- Configuration management plan
 - Different versions
 - Different operating systems
 - Different pricing schemes (Windows 10 has ten editions!)
 - Home / pro / enterprise / education / pro education / enterprise LTSC / IoT Enterprise / Team / Pro for Workstations
 - Discontinued editions: Mobile / Mobile Enterprise / IoT Mobile / S / 10X
- Staff development plan
 - How staff skills will be used and developed

Project Tracking

- Establish project constraints
 - Delivery date, budget, hardware, software, staff levels
- Assessment of parameters
 - Software design, size, interdependencies
- Define milestones and deliverables
- While (project continues)
 - Initiate new activities
 - Review progress (typically weekly or daily)
 - If (problem)
 - Review problem, initiate solution
 - Revise project constraints, parameters, and schedule
 - Renegotiate constraints and deliverables

Deliverable: The Project Plan

- Executive Overview
- Introduction
- Project description
- Resource requirements
 - Prices, schedules
- Organization
 - People and roles
- Project breakdown
 - Identifiable activities, milestones, and deliverables
- Risk analysis
 - Possible risks, and solutions
- Project schedule
 - Dependencies between activities
 - Time to milestones
 - Allocation of people to tasks
- Monitoring and reporting
 - How the project will be monitored
 - When reports are to be delivered
- Conclusion

The Project Plan

- Must look good, read well, and be accurate
 - Presentation makes a big difference
 - Accurate project planning is vital
- Managers need information to manage
 - Software is intangible
 - Reports and deliverables are the only way to manage
 - Cost estimates and schedules must be kept up-to-date
- Milestones and deliverables
 - Must be concrete (not virtual or unverifiable)
 - Deliverables many consist of many milestones

Scheduling

- Necessary time and resources
 - Previous estimates are uncertain because
 - This project is unique
 - Different languages / OS / design methods may be used
- Usually optimistic
 - Even if not then the slack gets wasted
- Use management tools
 - Microsoft Project, Google Sheets for Project Management, etc.
 - Keep it up-to-date

Scheduling

- Divide the project into pieces and estimate each
 - Don't make tasks too small (less than a week)
 - Don't make tasks too large (8-10 weeks)
 - Many tasks might be done in parallel
 - Identify dependencies between tasks
 - Assume problems will occur
 - Mechanical failure, staff turnover, bad estimates, resource unavailability
 - Estimate as if no errors will occur, add contingency (50-100%)
 - Allow for staff issues (holidays, illness, personal problems)
 - Allow for dependencies on others (delivery of goods)
 - Include all schedulable resources (disk, CPU, people)

Estimation Rules of Thumb

- 1/3 (33%) Project planning
- 1/6 (16%) Coding
- 1/4 (25%) Component testing
- 1/4 (25%) System testing
- Estimate program size
 - Lines Of Code (LOC) or thousands of LOC (KLOC)
 - Industry output about 1000 LOC per developer per year
 - Working, documented, and in production
 - About 240 working days per year
 - “garage developers” don’t write commercial software

LOCs

- Estimates based on:
 - Whole program size
 - Sum of the functional unit sizes
- Useful for:
 - Error rate estimation
 - Productivity rate estimation
- But dependant on:
 - Programmer style
 - Programming language
- Biggest problem:
 - Can only be known once the program is finished!

Basic COCOMO

- Constructive Cost Model (1981)
- Project Types
 - Simple projects
 - Well understood applications, small teams
 - Moderate projects
 - More complex, limited experience
 - Embedded projects
 - Complex, strongly coupled to hardware, software, regulations, etc.
- Metrics based on statistics drawn from a large number of software projects

Basic COCOMO

- Estimates:
 - Effort = $a(\text{KLOC})^b$
 - Time = $c(\text{Effort})^d$ months
 - People = Effort / Time
- Good for quick estimation, but does not consider hardware constraints, programmer skill, or modern tools
 - Intermediate COCOMO (81) and COCOMO II (1997)
 - Not enough time to discuss these in detail in class
 - See INFO310 or the text book for more details

| | a | b | c | d |
|----------|-----|------|-----|------|
| Simple | 2.4 | 1.05 | 2.5 | 0.38 |
| Moderate | 3.0 | 1.12 | 2.5 | 0.35 |
| Embedded | 3.6 | 1.20 | 2.5 | 0.32 |

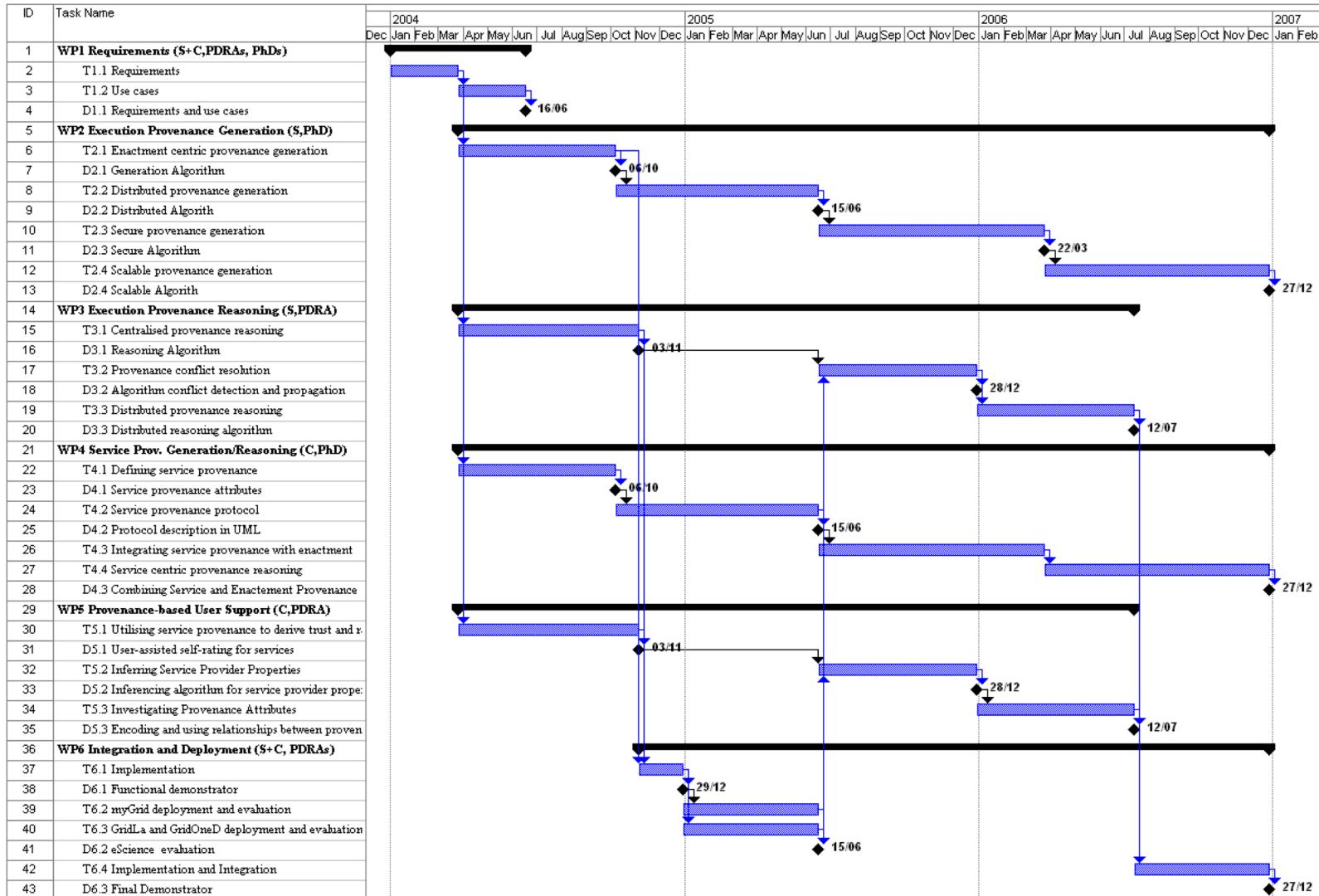
Functional Point Analysis (FPA)

- Size based on *user perceived functionality*
 - Language and style independent
- Define the functional requirements
 - Categorise:
 - Outputs, inquiries, inputs, internal files, and external interfaces
 - Define complexity and assign some functional points
 - Productivity is measured in points implemented per month
 - We now have a cost based on user's requirements
 - And so can modify the “spec” based on cost
- International Standards include:
 - Common Software Measurement International Convention (COSMIC)
- Problems
 - Does not deal with algorithmic complexity or effort
 - Complexity estimates are estimator dependant
 - Biased towards data processing systems (because of the categories)

Gantt Charts

- Invented in 1917 by Charles Gantt
- Focus on tasks needed to complete a project
- Each task represented by horizontal (time) bar
 - Length of the bar represents length of task
- Arrows connecting tasks represent dependencies
- Diamonds are milestones and deliverables
- Come in many different forms
 - Often tool dependent
- Software will often identify critical paths

Gantt Chart

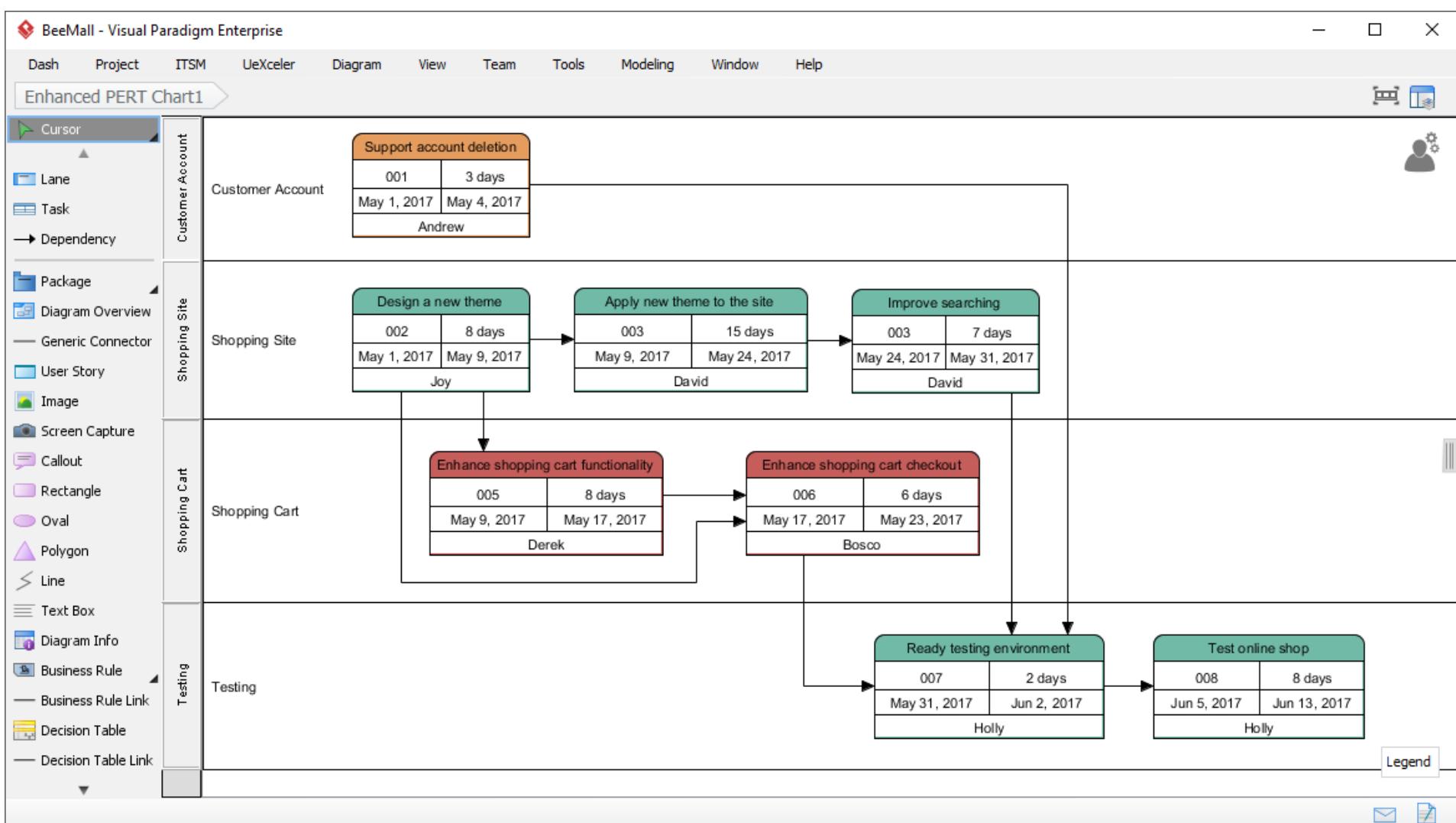


From: http://www.pasoa.org/case_files/pasoa-workplan.gif

PERT Charts

- Invented in the 1950s by the US Navy
- Completed tasks crossed out
- Partially completed tasks slashed out
- Details of task shown in box
- The critical path is highlighted

Pert Chart



From: <https://www.visual-paradigm.com/features/project-management-diagrams/enhanced-pert-chart/>

Risks

- Project risks
 - Staff and management turnover
 - Hardware and other dependency (including spec) availability
 - Requirements change
- Product risks
 - Failure of third party tools (bad libraries, etc.)
 - Project size underestimation
 - Requirements change
- Business risks
 - Technology changes (e.g. introduction of smart watches)
 - Competitors introducing similar product
 - Requirements change

Risk Management

- Project managers
 - Anticipate risks
 - Take actions to avoid risks
 - Develop solutions to risks
- Project plan
 - Includes a risk analysis
 - Consequence of risks occurring
 - Cost of avoiding / fixing consequence
 - Contingency plans
- This is a continuous process

Risk Management

- Risk identification
 - Team exercise often through brainstorming
- Look for
 - Technology risks
 - Will technology change (e.g. popularity of Windows vs. MacOS)
 - People risks
 - Staff leaving / holidays / parental leave
 - Organization risks
 - Takeovers / mergers / management changes / corporate focus
 - Tools risks
 - Will the tools live up to requirements (software and hardware)
 - Requirements risks
 - Estimation risks
 - Society risks
 - Global pandemic

Risk Management

- Risk analysis
 - For each risk
 - Judge probability of occurring (low, middle, high)
 - Judge cost of recovery (catastrophic, serious, tolerable, insignificant)
- Risk planning
 - Plan for avoidance
 - Plan for impact minimization
 - Have contingency plans
- Risk monitoring
 - Regularly re-assess each risk

References

- F. Brooks, *Mythical Man Month*, Chapter 2
- I. Sommerville, *Software Engineering*, Chapter 5

COSC345

Software Engineering

The Waterfall Model

Outline

- Why have software development models?
- Manufacturing models
- Defining waterfall
- Refining waterfall
- Top down versus bottom up design
- Versioning

Software Development Models

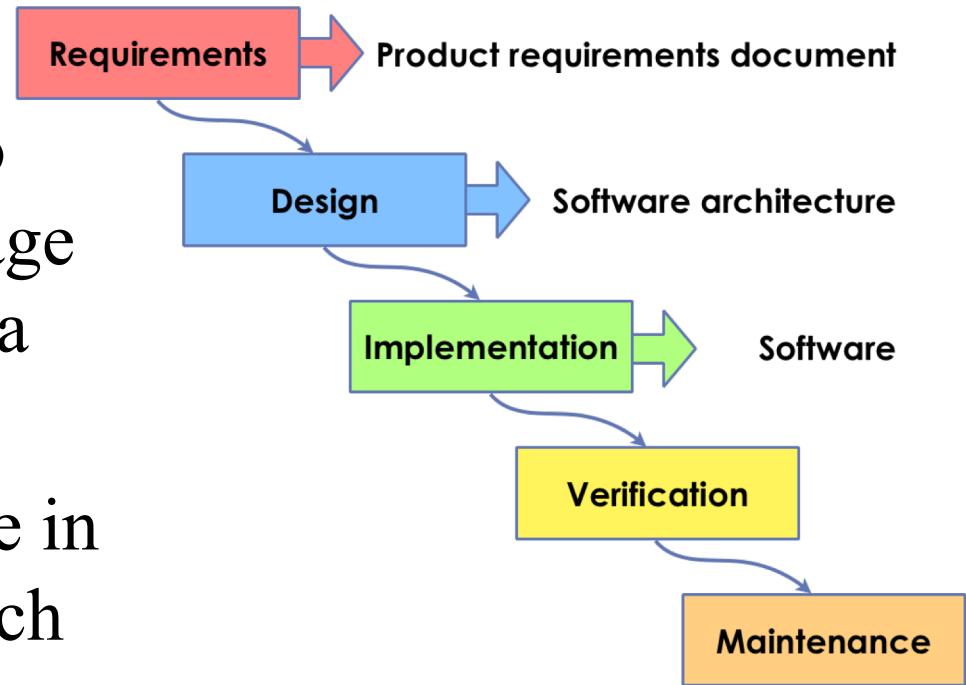
- Software developed in teams needs management
 - Discussed in the earlier project planning lecture
 - Initial plans will not be perfect
 - They will have missing or incorrect information
 - Resource allocation
 - People
 - Infrastructure
 - Timeline estimation
 - For resource allocation
 - Risk management
- Separation of concerns between coding and management
 - Skill sets are different

Manufacturing Models

- Most forms of engineering consume physical resources
 - Civil engineers build bridges
 - ... that don't collapse (very often)
 - Staff of different types are scheduled into construction plans
- When industrial-scale software development began it made sense to try to use manufacturing models
 - Better than having no idea what to do
 - Seeking homogeneity for project plans
 - But software is very heterogeneous by nature
- Change is expensive
 - A design change once a bridge is being built is prohibitively expensive. Is this the case for software?

Waterfall Model

- Five stages progress in order within project
- Named “waterfall” due to progress from stage to stage appearing to operate like a waterfall
- The stages are not to scale in terms of time taken on each
- May be worth considering that waterfalls can be rather destructive, in practice!



Stage 1: Requirements Capture

- We discussed requirements capture in a recent lecture
 - Crucially important for any engineering project
 - Almost certainly the first stage of any project
 - Ideally involves planning and discussion with clients
- Dilbert cartoons critique software development models
 - Teams refusing to progress from requirements stage!
 - A system with requirements too complex for users to be able to understand it?
 - Just add the “easy to use” requirement!

Stage 2: Design

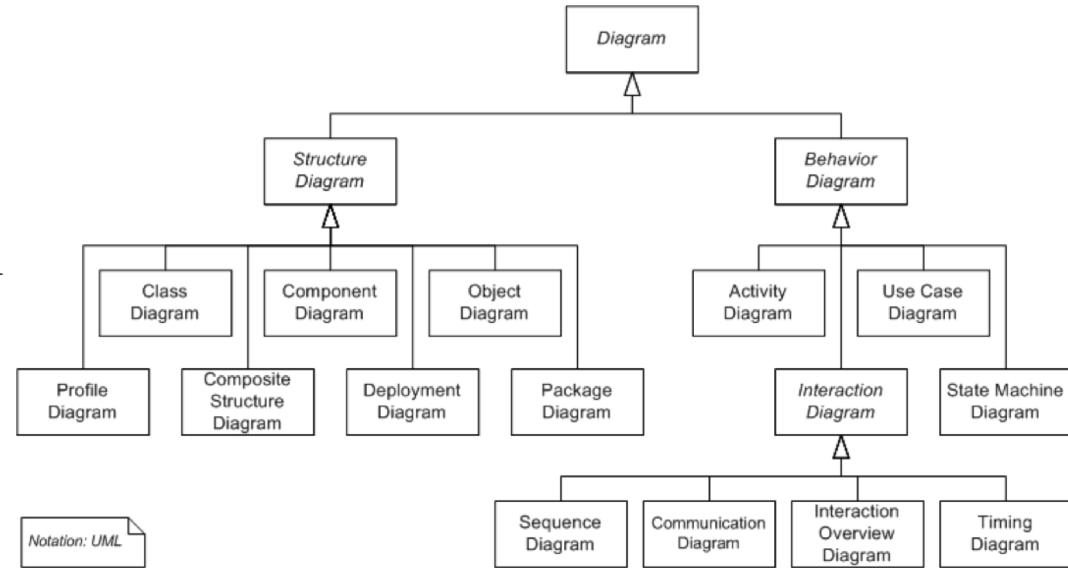
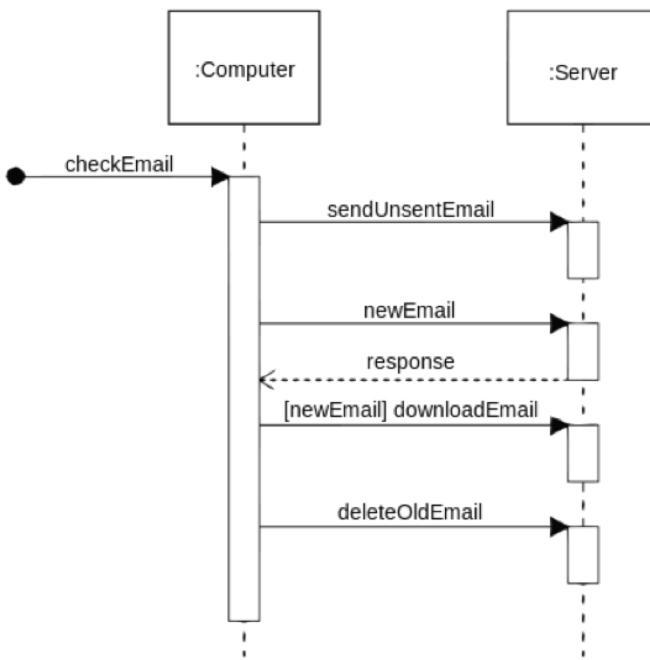
- Take requirements relating to software behavior and frame them as relevant to software engineering effort
 - **Models** of software functionality
 - Components, OO, etc.
 - **Schemas** of data
 - Database Entity/Relations, or OO, etc.
 - **Business logic**
 - Database triggers, business rules, etc.
- UML diagrams model many aspects of software, e.g.
 - **Class diagrams**
 - Methods, fields, inheritance, composition
 - **Sequence diagrams**
 - How do components interact?

Aside: Meta-software

- Software that helps you build software
- Typically:
 - Programming integrated development environments
 - Compilers, etc.
- But also, web development increasingly web-based:
 - Wikis facilitate sites that can be dynamically updated
 - Content Management Systems
- Database examples
 - FileMaker Pro, Oracle Forms, HyperCard (!)
- Programs you write that write software
- Little languages

Aside: UML2 Diagrams

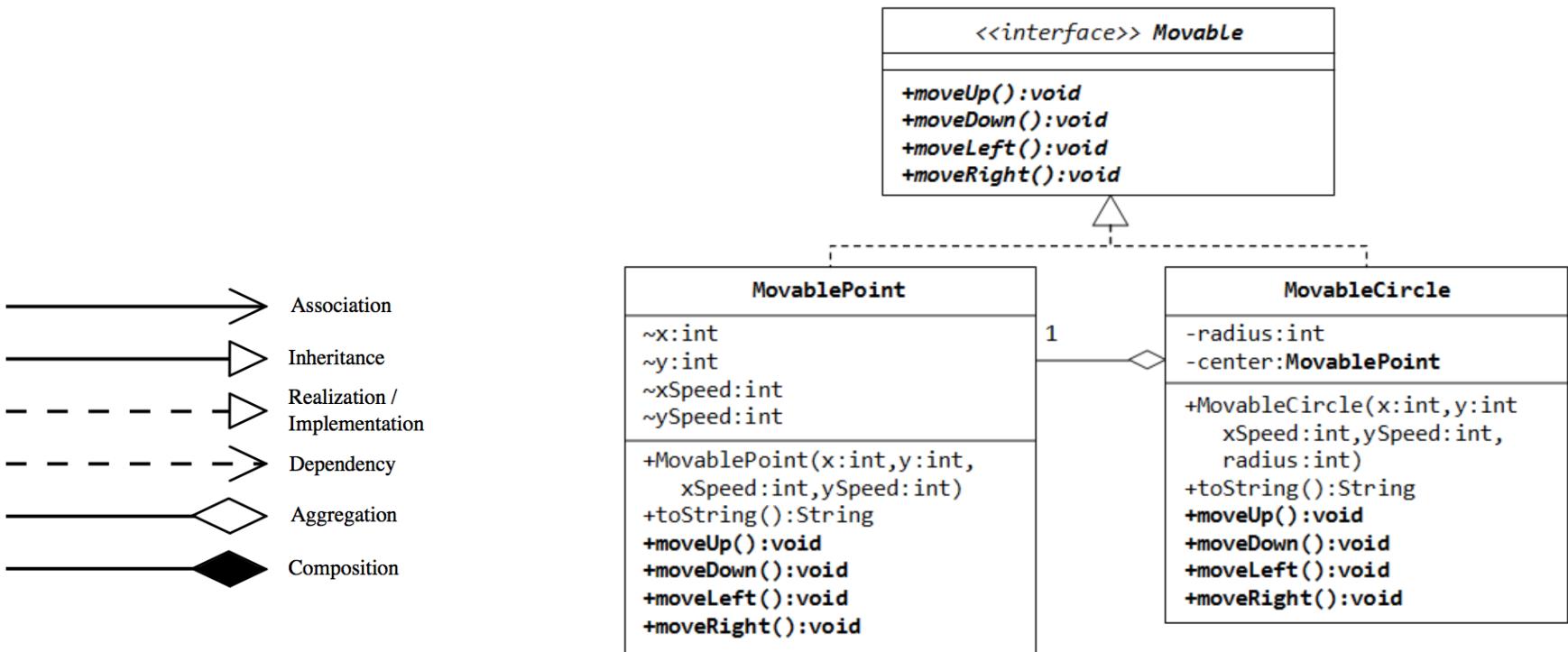
- Structural
 - Class diagram
 - Component diagram



- Behavioural
 - Sequence diagram
 - Activity diagram
 - Use case diagram

Aside: UML2 Class Diagrams

- Class diagrams are useful even if you don't design your software using formal methods
- There are tools that will generate these from source code



Stage 3: Implementation

- Development teams are tasked with writing various software components based on specifications provided to them
- Software components then need to be integrated
 - Often through Continuous Integration (CI)
- Documentation is created
 - Ideally this is integrated into the code itself
 - With tools like Doxygen or Javadoc
- Coding is expected to be one third of project time
 - Unlike CS papers, where it appears to be most of the time

Stage 4: Verification

- Removing defects from the system
 - i.e. debugging
- At first this may be done by the project teams
 - May be problems with components
 - May be problems with component integration
- Eventually bug reports will come from end users
 - Very likely that these will be requirements mismatches
 - Users will test software in previously unforeseen ways
- What about test-driven development?
 - Unit test engineering allows debugging during implementation

Stage 5: Maintenance

- Why? Almost all code projects **evolve features**
 - This is one reason that software has a version number
 - What version is Google Docs at?
 - What does this imply?
 - Repeat small-scale version of software development model
 - ... but can probably have a shorter requirements gathering stage
- Software and hardware **dependencies** can change
 - Language, library, or operating system upgrades

Waterfall for software development

- Waterfall has become more of an anti-model
 - A model you are likely to be assumed to know about
 - A model against which improved models are compared
- The forward-only flow causes inflexibility
 - Can make sense when reworking earlier stages is expensive
 - such as building bridges
 - But that's not the case for software engineering
 - where changes can be inexpensive
- Risks make all stages expensive and slow

Building on Existing Software

- Earlier software development models from past era
 - More likely that a software project would be a sizeable and self-contained effort
- Reliance on frameworks and other components can significantly speed up development time
 - However, problems may emerge at integration time
 - e.g., design phase may have misunderstood aspects of an API

Augmented Waterfall Models

- Address obvious failings of the Waterfall Model
 - Overlap the five phases of the original model
 - Apply phases to sub-projects of the main project
- Winston W. Royce often “credited” with inventing the Waterfall Model, but
 - His model understood the value of feedback
 - Suggested that the phases should be passed over **twice**
 - Was explicit in the value of prototypes to effective engineering
 - Prototyping tools can be extremely rich and powerful

Top-down versus Bottom-up

- Top-down design considers abstract components first
 - Refine design into subcomponents progressively
 - Eventually reach the level of code components
 - Classes, methods, stubs, implementations
- Bottom-up design writes lower-level components first
 - Progressively creates overall system
 - Build methods into classes, join classes, etc.
- But neither works entirely, on its own
 - Waterfall is essentially top-down
 - Agile is essentially bottom-up (see upcoming lecture)

Top-down versus Bottom-up

- Applying either design process rigidly has risks
 - Top-down design can cause need for reimplementation when decomposition is not steered toward existing components
 - Bottom-up design can lead to components that do not interface cleanly with other (unforeseen) parts of a system
- GUI programming often requires a hybrid approach
 - Interface implementation is often top-down in GUI “builder”
 - Application’s persistence of state through bottom-up objects

Aside: Version Numbers

- Versioning is worth some thought
 - Tags in version control system?
 - Are digits in a sequence a number or a string?
 - How to indicate sequence from alpha to production?
- Semantic Versioning
 - Major.Minor.Patch[.tag]
 - Specifies practices, version comparison rules
 - Indicates API compatibility
 - Patch increments don't break APIs
- Alternative: irrational convergence?
 - TeX is 3.141592653...

References

- Waterfall model
 - https://en.wikipedia.org/wiki/Waterfall_model
- Unified Modeling Language
 - https://en.wikipedia.org/wiki/Unified_Modeling_Language
- Semantic Versioning
 - <https://semver.org>

COSC345

Software Engineering

The Agile Model

Outline

- 12 Principles of the Agile Manifesto
 - We will explore each in turn
- Kanban
 - Visualising the progress of team projects
 - Helps track tasks, and avoid bottlenecks, in project work
- Scrum
 - Project management involving:
 - Controlled group sizes
 - Controlled durations of tasks within project
 - Daily stand-up meetings
 - Particular team roles

Agile Manifesto Principles

- Customer satisfaction by early and continuous delivery of valuable software
- Welcome changing requirements, even in late development
- Deliver working software frequently (weeks rather than months)
- Close, daily cooperation between business people and developers
- Projects are built around motivated individuals, who should be trusted
- Face-to-face conversation is the best form of communication (co-location)

Agile Manifesto Principles

- Working software is the primary measure of progress
- Sustainable development, able to maintain a constant pace
- Continuous attention to technical excellence and good design
- Simplicity—the art of maximizing the amount of work not done—is essential
- Best architectures, requirements, and designs emerge from self-organizing teams
- Regularly, the team reflects on how to become more effective, and adjusts accordingly

Principle 1

- “Customer satisfaction by early and continuous delivery of valuable software”
- Some points to note:
 - “**Early**”
 - Being successful means you beat your own plans
 - “**Continuous delivery**”
 - Continuous Integration & Continuous Delivery (CI/CD) see upcoming lecture
 - “**Valuable**”
 - Software the customer wants

Principle 2

- “Welcome changing requirements, even late in development”
- Agile processes harness change for the customer's competitive advantage
- Some points to note:
 - To “**welcome**” is to encourage changing requirements
 - If using CI/CD, software will have been delivered even if there are changes **late in development**
 - **Competitive advantage** from adopting change effectively

Principle 3

- “Deliver working software frequently (weeks rather than months)”, with a preference to the shorter timescale
- Some points to note:
 - **“Couple of weeks”**
 - Continuous delivery of working software on scale of a couple of weeks is rapid!
 - See: rapid application development (RAD)
 - **“Preference”**
 - Indicates a strong push-back against software development staying hidden, in-house

Principle 4

- “Close, daily cooperation between business people and developers”
- Some points to note:
 - This is very different from many companies’ past approaches:
 - *Was* common to have divisions with skills and experience in silos
 - Explicit about **daily** interactions:
 - There is a daily cycle in many agile methodologies

Principle 5

- “Projects are built around motivated individuals, who should be trusted”
- Give them the environment and support they need, and trust them to get the job done
- Some points to note:
 - If programmers aren’t **motivated**, they won’t deliver efficiently
 - Recognition that **environment** is important to consider
 - The **trust** involves avoiding micromanagement
- We’ve already talked about the Talent Oriented Model for Team Organisation

Principle 6

- “Face-to-face conversation is the best form of communication (co-location)”
- Some points to note:
 - Requiring *lots* of **face-to-face** time needs meetings to be easy
 - Typically schedule daily meetings
 - Suggesting clients talk to developers, *and* developers talk to developers
 - Online collaborative tools are used in remote groups

Principle 7

- “Working software is the primary measure of progress”
- Some points to note:
 - This principle has caused some push-back:
 - Has been interpreted to indicate code over documentation
 - ... which leads to technical debt
 - Technical debt is work that must be paid back in future
 - e.g. bad solution that must be refactored later
 - (causes major problems if key developers become unavailable)
 - The **primary** measure is not the only measure!

Principle 8

- “Sustainable development, able to maintain a constant pace”
- The sponsors, developers, and users should be able to maintain a constant pace indefinitely
- Some points to note:
 - Sprinting can be very effective but may not be **sustainable!**
 - Maintaining a **constant pace** implies a lack of pressure
 - Covers *all* of the key stakeholders
 - Because all key stakeholders are involved throughout

Principle 9

- “Continuous attention to technical excellence and good design”
- Some points to note:
 - Attention to **good design** likely involves making improvements
 - Not involving **technical excellence** may cause technical debt
 - **Continuous attention** differs from waterfall-like schemes, since a linear progression through phases would not work
- Many agile teams tend to ignore this principle as it is not visible and cannot be measured
 - This (in Andrew’s experience) can lead to an out-of-control build-up of technical debt. Principle 9 is important!

Principle 10

- “Simplicity—the art of maximizing the amount of work not done—is essential”
- Some points to note:
 - Maximising **work not done**, but must still achieve excellence
 - **Simplicity** may involve changing component boundaries
 - Refactoring code has been known to reduce its length, simplify it, and cause it to run more quickly

Principle 11

- “Best architectures, requirements, and designs emerge from self-organizing teams”
- Some points to note:
 - Teams that **self-organise** involve balanced participation
 - The self-organisation *can* be done within an overall framework
 - The three output types mentioned covers much of software engineering other than the implementation and testing

Principle 12

- “Regularly, the team reflects on how to become more effective, and adjusts accordingly”
- Some points to note:
 - **Regular** use of management processes such as self-reflection
 - Environment must support the ability to **adjust** behaviour
 - There are many potential ways to measure being **effective**

Kanban

- From manufacturing (Toyota)
 - Each process issues requests (kanban) to its suppliers when it consumes its supplies
 - Each process produces according to the quantity and sequence of incoming requests
 - No items are made or transported without a request
 - The request associated with an item is always attached to it
 - Processes must not send out defective items, to ensure that the finished products will be defect-free
 - Limiting the number of pending requests makes the process more sensitive and reveals inefficiencies

Kanban

- Adaptation to software development as way of managing team members' cognitive resources
- Favours continuous flow and continuous delivery
 - Kanban Board:
 - A board representing all work items and their stage of completion, sometimes including the requestor's details
 - Allows all team members to see where all pieces are at any time
 - Avoids accumulation of work-in-progress tasks
 - Because of column limits
 - Obvious when running low on work
 - No work is done without it being on the board
 - Ensures a collective effort to progress all tasks to completion
- The Toyota principles all apply

TIMELINE

Zoom: Auto Today

| Oct 2019 | | Nov 2019 | | | | | | | | | |
|---|--------|---|-------|---|-----------------------------|--------------|---------------|--|--|------------|--|
| (0) | Wed 30 | Thu 31 | Fri 1 | Sat 2 | Sun 3 | Mon 4 | Tue 5 | | | | |
| TIMELINE (0/5 IN PROGRESS) | | | | | | | | | | | |
| <div style="display: flex; justify-content: space-between;"> <div style="flex: 1;"> (0) BACKLOG </div> <div style="flex: 1;"> 100% New Customer Portal - Responsive Implementation for the home page  2d </div> <div style="flex: 1;"> 0% Feature A Improvements  + 0d </div> <div style="flex: 1;"> 20% Develop New feature /Predicting the future/  0d </div> </div> | | | | | | | | | | | |
| REQUESTED (2/0) | | READY TO START (1/0) | | IN PROGRESS (4/0) | | | | | | DONE (1/0) | |
| | | | | TECH DESIGN (1/0) | READY FOR TECH REVIEW (1/0) | CODING (1/0) | TESTING (1/0) | | | | |
| BUGS | | | | | | | | | | | |
| <div style="display: flex; align-items: center;"> 1715 Alex <div style="border: 1px solid #ccc; padding: 5px; margin-left: 10px;"> External link textbox is longer than it should be on new card details A </div> </div> | | | | | | | | | | | |
| FRONT-END | | | | | | | | | | | |
| 1577 None | | 1572 None | | 1574 Gabriel | | | | 1575 Alex | | | |
| Develop reusable components  New subtask... | | Front page - Develop Front End  New subtask... | | Front page - Add Google Analytics Tracking Code  New subtask... | | | | Customize the admin panel according to the customer's needs  New subtask... | | | |
| BACK-END | | | | | | | | | | | |
| 1579 None | | 1573 Jason | | 1576 Jason | | | | | | | |
| Extend the database schema  New subtask... | | Front page - Develop Backend  New subtask... | | Integrate the customer's billing system with the CMS  New subtask... | | | | | | | |

Scrum

- Approach to agile management of software projects
- Development work is divided into short **sprints**
 - Likely to involve no more than a few weeks' work
- Daily **scrums**: 15-minute stand-up meetings
 - Meeting has to finish before people become uncomfortable
 - Do planning not problem-solving: subgroups can do that
 - **Scrum master**
 - Like a project manager but with no HR work
 - **Product owner**
 - Non-technical
 - keeps customers in the loop
 - **Development team**
 - Rest of the group (preferably fewer than 10)

Kanban or Scrum?

- Some agile groups use Kanban
- Some use Scrum
- Some use both
 - You'll often see a Kanban board used with Scrum and sprints

Waterfall or Agile?

- Agile favours:
 - Bottom-up design and build
 - Continuous integration and continuous delivery
 - Obviously useful for a web-based business
- Waterfall favours:
 - Top-down design focusing on components
 - Obviously useful for formal contracts such as military work
- In practice, **neither works entirely** on its own
 - In Andrew's experience, you need to know what you are building before you start, but must build for Continuous Integration and Continuous Delivery
 - Plan to start at both ends and meet in the middle

References

- Agile software development
 - https://en.wikipedia.org/wiki/Agile_software_development
 - The twelve principles are lifted from this Wikipedia article
- Atlassian agile coach
 - <https://www.atlassian.com/agile>
- Kanban board
 - https://en.wikipedia.org/wiki/Kanban_board
- Scrum (software development)
 - [https://en.wikipedia.org/wiki/Scrum_\(software_development\)](https://en.wikipedia.org/wiki/Scrum_(software_development))
- Rapid application development
 - https://en.wikipedia.org/wiki/Rapid_application_development

COSC345

Software Engineering

Communications

Outline

- Brooks Law
- Late Projects
- Divisible tasks
- Indivisible tasks
- Communications tasks
- The work book
- Source code communications

Brooks Law

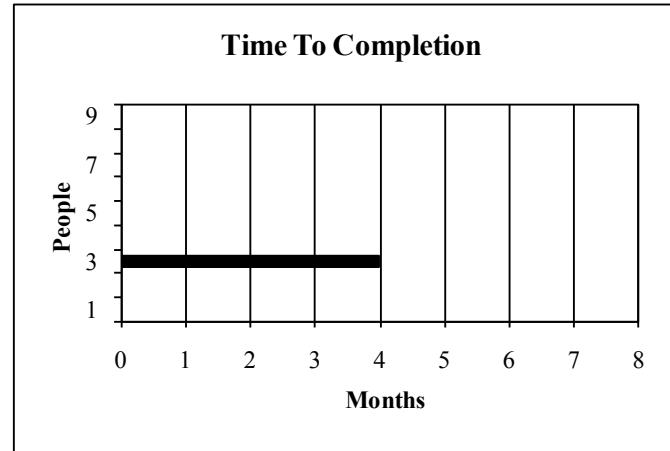
- “When schedule slippage is recognized, the (natural and traditional) response is to add manpower. Like dousing a fire with gasoline, this makes matters worse, much worse. More fire requires more gasoline, and thus begins a regenerative cycle which ends in disaster.”
(Brooks)
- Brooks Law:
Adding manpower to a late software project will make it later

The Man-Month

- Definition:
 - The productivity of one person in one month
- Software projects are often measured in
 - Man-Months
 - Lines of Code (LOCs)
 - Functional Points (FPA)
 - What's the exchange rate between these?

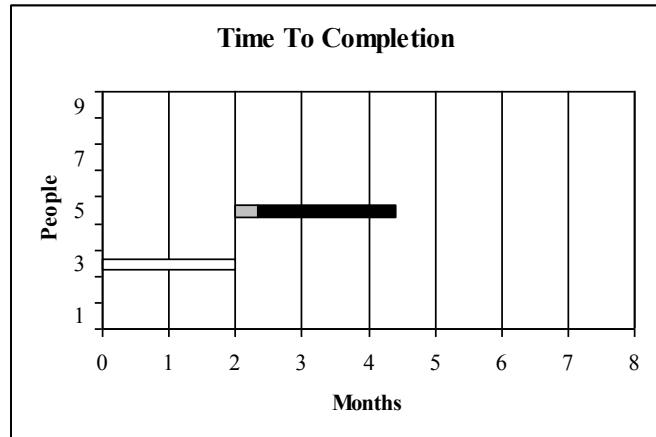
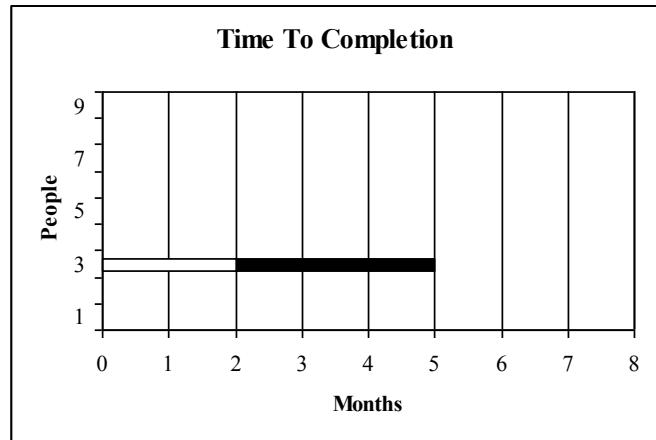
Late Projects

- Add manpower?
- Example
 - A task is estimated at 12 man-months and split between 3 people (over 4 calendar months)
 - Milestones are set at the end of each calendar month
 - The first milestone is reached at the end of the 2nd calendar month
- Now what?



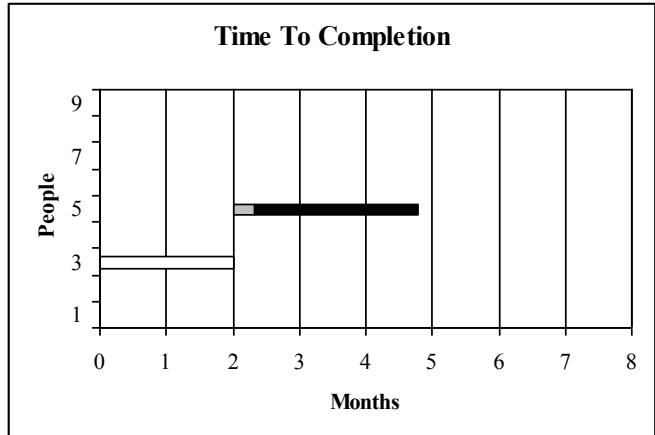
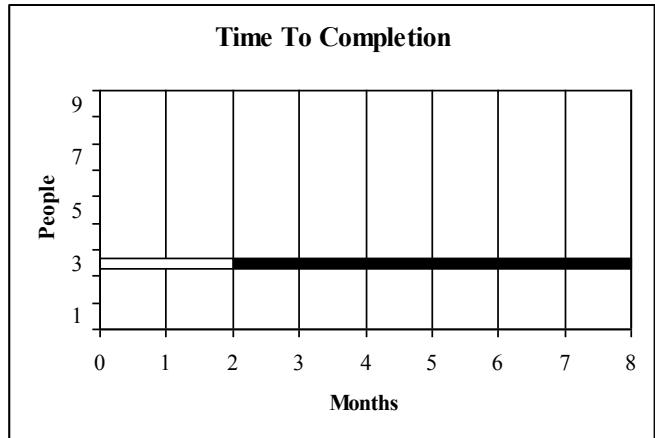
Catch Up

- Analysis
 - First part of schedule misestimated
 - 9 man-months remain
 - 2 calendar months remain
 - Project requires 4.5 people ($9 \div 2$)
 - Add 2 new people
- Problem
 - Training takes 1 calendar month
 - 2 man-months done that month
 - At end of calendar month 3
 - 5 Trained developers
 - 7 Man-months work remains
 - Completion date
 - After calendar month 4!
- How many new staff are needed?
 - Did we just double the group size?



Catch Up

- Analysis
 - Whole schedule misestimated
 - 18 man-months remain
 - 2 calendar months remain
 - Project requires 9 people ($18 \div 2$)
 - Add 6 new people
- Problem
 - Training takes 1 calendar month
 - 2 man-months done that month
 - At end of calendar month 3
 - 9 Trained developers
 - 16 Man-months work remains
 - Completion date
 - After calendar month 4!
- How many new staff are needed?
 - Did we just triple the group size?



Solutions

- Reschedule
 - If the planning is accurate then this is unnecessary
 - Allow enough time
 - Make sure rescheduling isn't needed again
- Trim the task
 - Trim non-essential tasks (carefully)
 - Recall: “shall” and “should” tasks
 - Then reschedule
 - In practice this happens anyway

Software

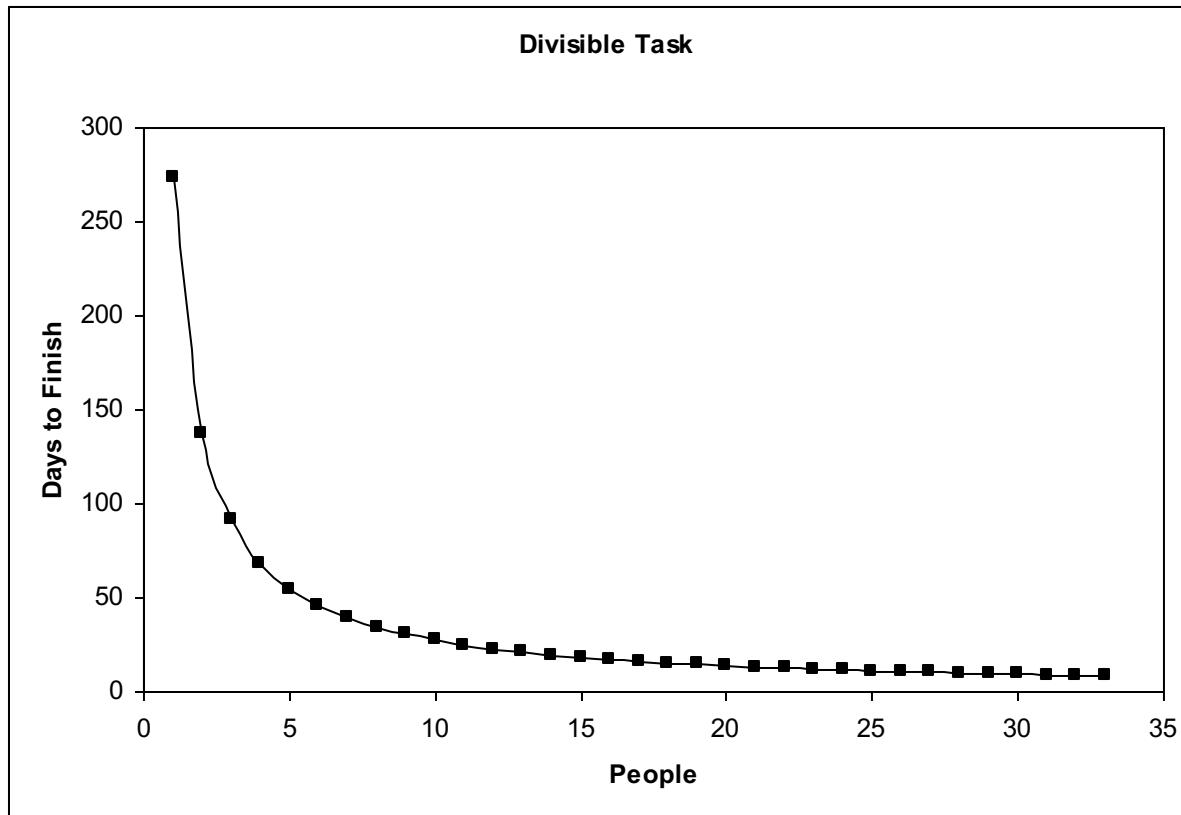
- Often indivisible
 - Interdependencies make division difficult
 - Debugging is often sequential
 - Testing and fixing is an iterative process
- Requires substantial amounts of communications
 - Training
 - Linear cost (can't train one person in parallel)
 - Intercommunications
 - Can be controlled
 - Group meetings
 - Person to person
 - Emails

Digging Holes

- If it takes
 - 1 person one day to dig a hole 1m by 1m by 1m deep
- How long does it take
 - 2 people to dig 2 holes 1m by 1m by 1m deep?
 - 2 people to dig a hole 1m by 2m by 1m deep?
 - 2 people to dig a hole 1m by 1m by 2m deep?
- What's the difference between the tasks?

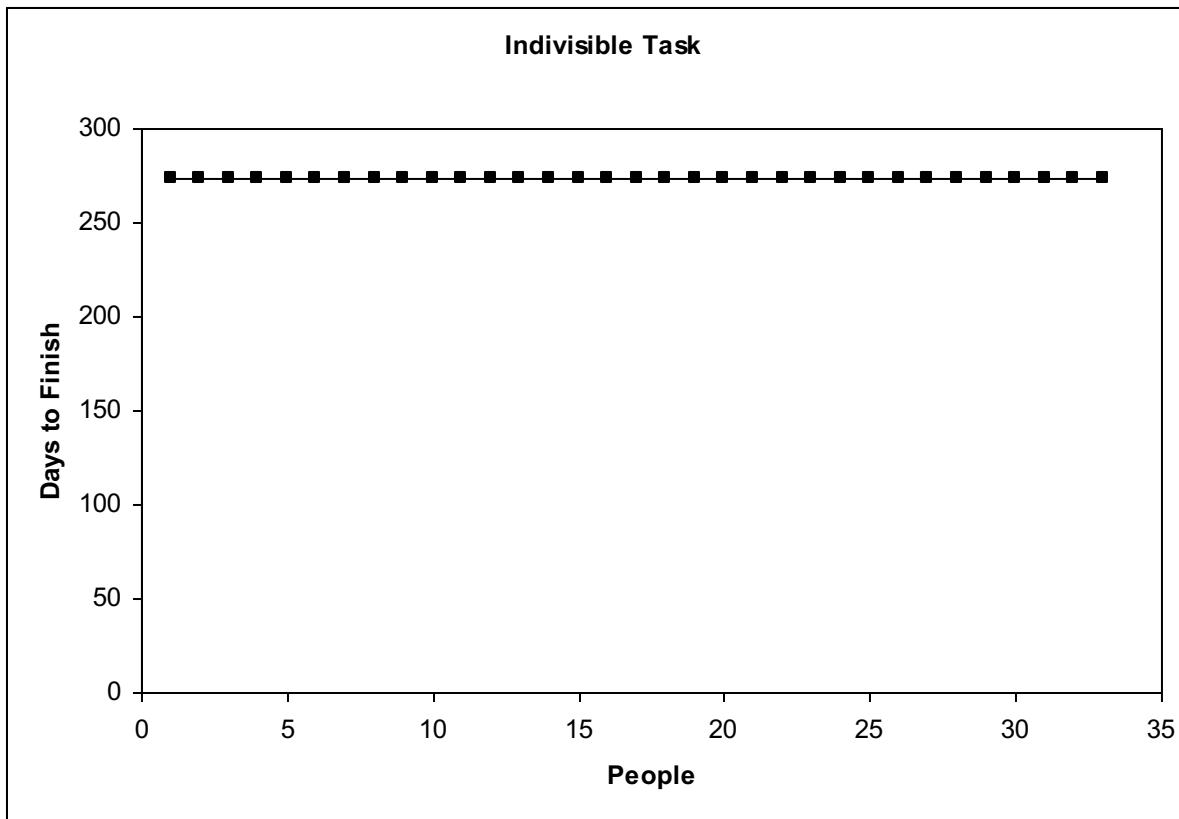
Divisible Tasks

- If it takes 1 person 9 months to hand-plough a 1000m^2 field, how long would it take with 7 people?



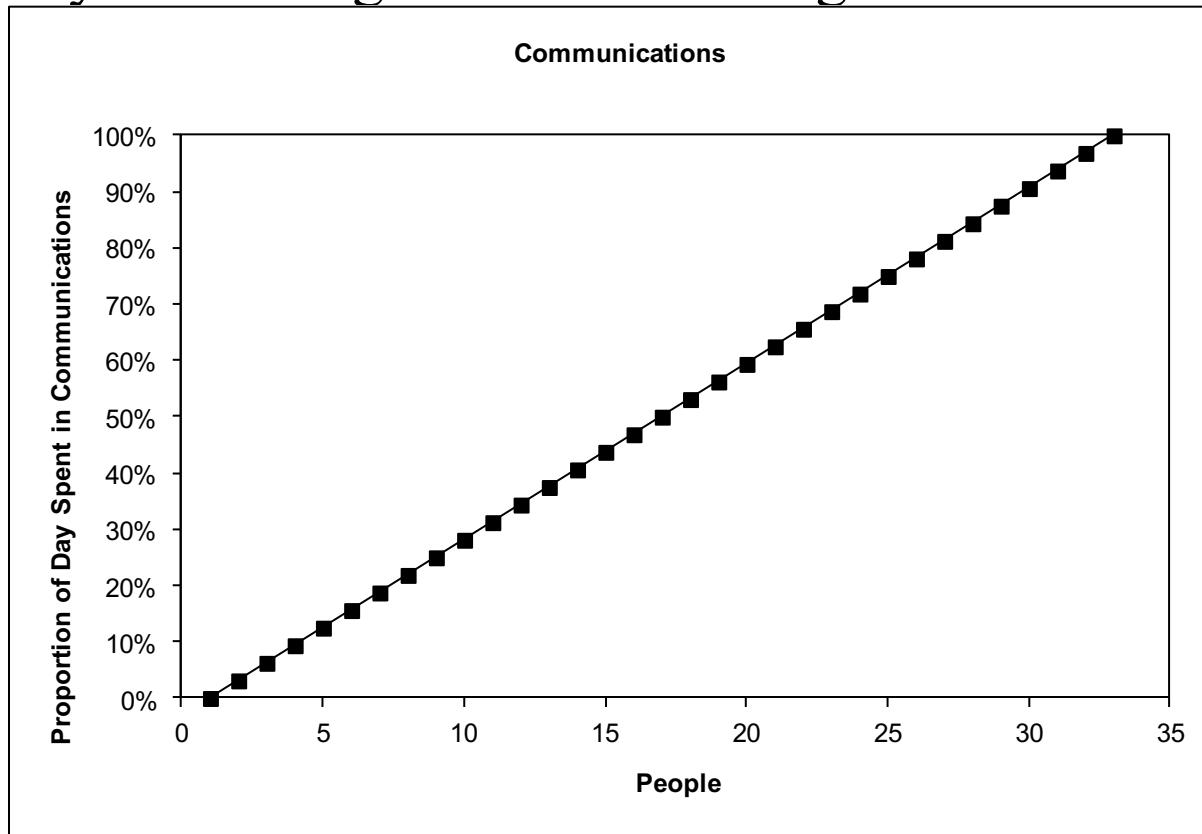
Indivisible Tasks

- If it takes 1 mother 9 months to grow a baby, how long would it take if 7 family members help?



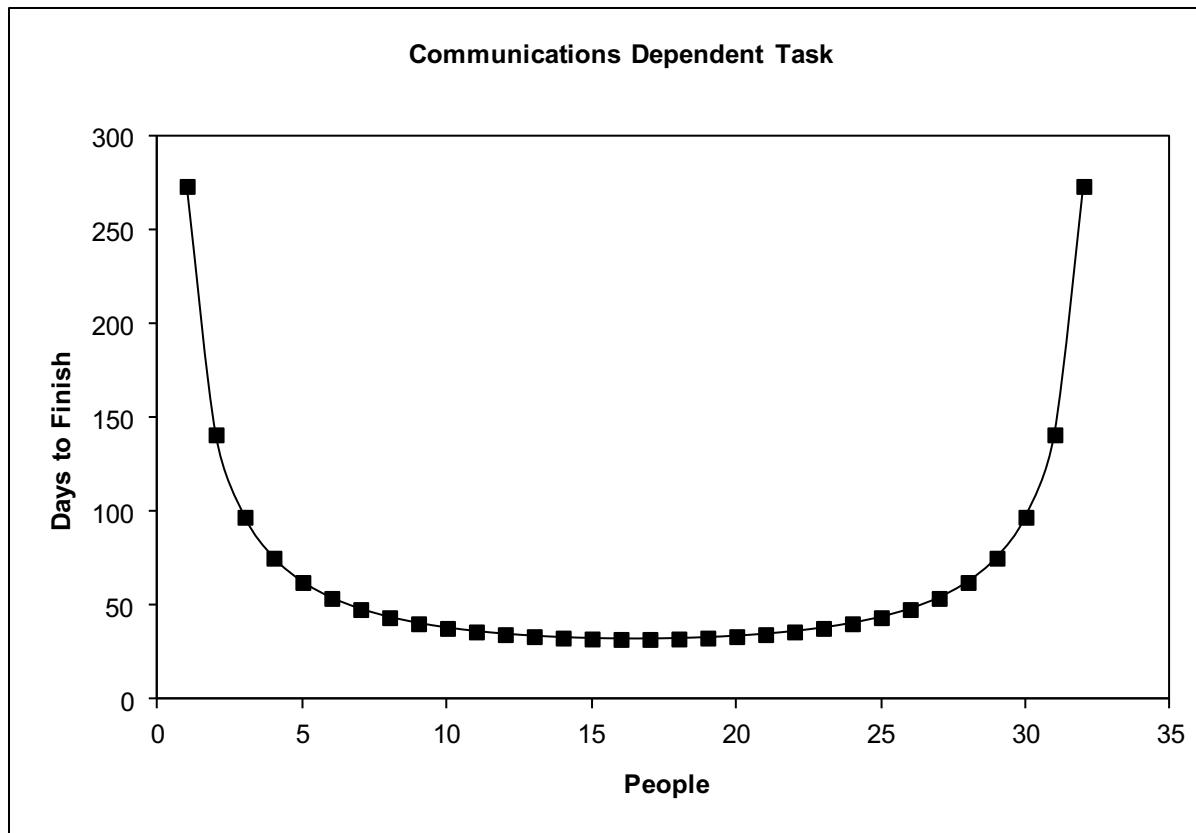
Communications

- Communication cost increases with people
 - $n(n-1)$ messages or $(n(n-1))/2$ exchanges
- Assuming 15 mins per exchange per day:
 - Only 32 messages can be exchanged in an 8 hour day!



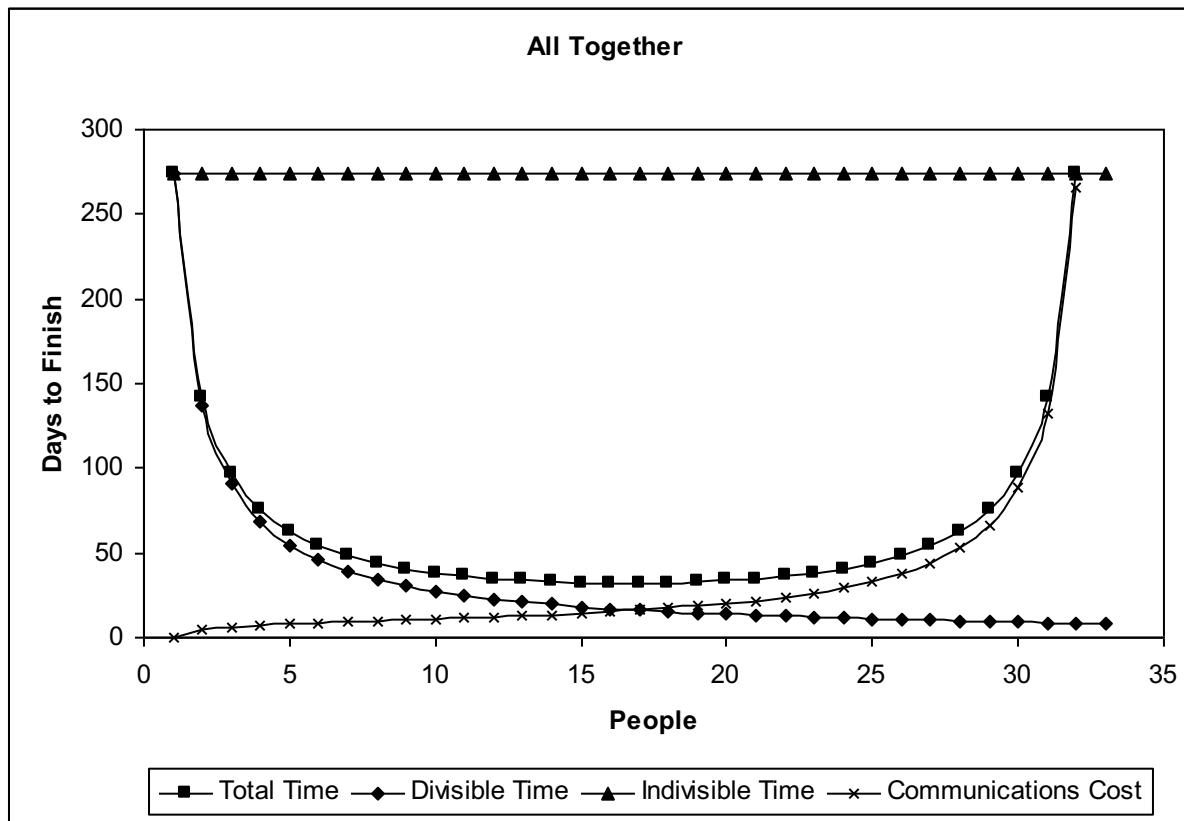
Communication Dependent Tasks

- A divisible task requiring significant communication



What's Happening Here?

- The total time is
 - Time for each person, plus
 - Time for communications (15 mins per daily exchange)



Communications

- Good communications is vital for a project
- Types of communications
 - Informal discussion
 - A clear project structure will increase informal discussion and reduce project-wide discussion
 - In a chaotic environment $n(n-1)/2$ exchanges needed
 - Meetings
 - Formal meetings reduce misunderstandings
 - Increase overall understanding
 - Give everyone a chance to speak
 - All exchanges are heard by everyone
 - Project Workbook
 - Log of all communications in the project

Project Workbook

- All documents in the project
 - Requirements, design, technical standards, etc.
- Controls the distribution of information
 - All relevant information gets to all relevant people
- Mechanics
 - Small group (10), number the documents
 - Middle group (100), threads are needed
 - Large group (1000), rigid structure needed
- Keep it up to date

The OS/360 Workbook

- Every programmer saw the whole book
- Updated daily
- Loose leaf folder approach
 - Only changes were distributed (amended pages)
 - Change sheets discussing changes
- After six months
 - Workbook was 5 feet thick!
 - 100 copies distributed
 - Daily change log about 2 inches (150 pages)
 - Maintenance took a significant part of the day!
- How would it be done today?

Modern Tools

- Github comes with:
 - Wiki
 - Store design documents
 - Store brainstorming ideas
 - Store notes on releases
 - Use it
 - Bug-tracker
 - Use it
- Slack and Discord have become popular
 - A discussion forum
 - Use it
- Trello or other Kanban tools

References

- F. Brooks, *Mythical Man Month*, Chapter 2 & 7

COSC345

Software Engineering

Version Control

Outline

- ***BRING YOUR LAPTOP TO THE TUTORIAL***
 - *We will connect GitHub to Visual Studio*
- Some Problems
 - Communications
 - File system problems
- Version control
 - Basic principles and use
- When to use version control
- Examples
 - Subversion, git
- Best practice

Programmer Communications

- Programmers need to know
 - Who is working on which files
 - To avoid collisions
 - Who made which changes to which files
 - Who broke my code!
 - How to get
 - The latest version of a source code file
 - How to rollback to the most recent working version
 - Any given version in which a bug is reported
- Programmers need to manage
 - Different configurations (different OS / versions)
 - Concurrent debugging sessions

The File System

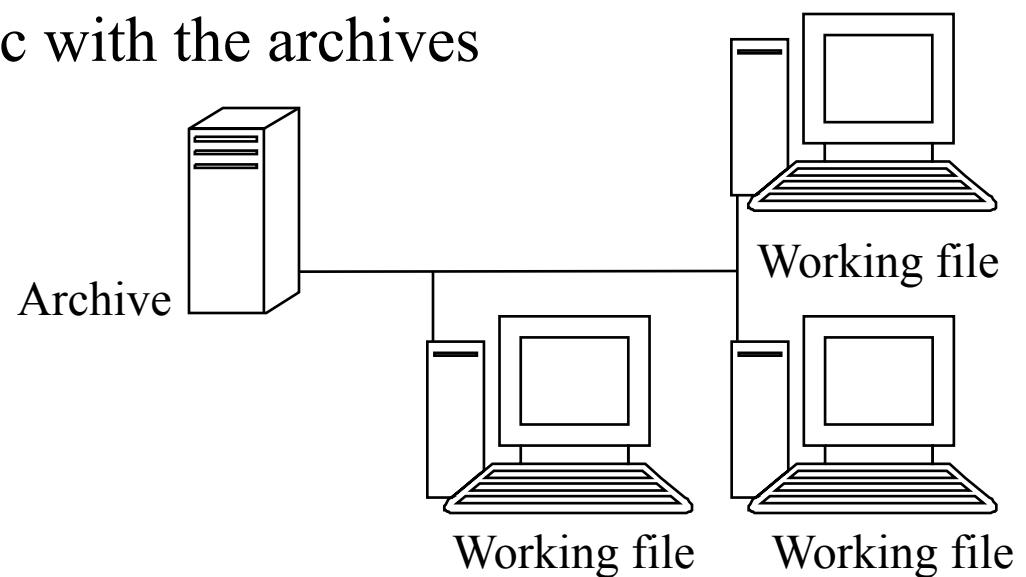
- The problem lies in the file system
 - Most file systems are absent of version numbers
 - Most file systems don't keep logs of file modifications
 - Backup files
 - Get lost
 - Cumbersome to create
 - Are hard to manage
 - Reasons for creating backups become forgotten
 - Today its possible to keep all old versions
 - How will we manage just the ones worth keeping?
 - Just the versions of source code that compile?
- Which old edits are worth keeping?

Version Control

- Storage and organization mechanism for backups
 - A database of old (and new) versions, with notes
- Allows users (developers) to
 - Retrieve old versions of a file
 - Track changes (who, what, and when)
 - Edit a file while another user is editing it
 - Use good housekeeping practices (documentation)
- Version control systems store
 - Who made a change
 - What they changed
 - When they changed it
 - Why they changed it (supplied by the user)

Basic (Generic) Structure

- Centralized archive
 - Held on file server (or in the cloud)
- Decentralized development
 - Copies held in local directories
 - Different developers, different instances
 - Developers can sync with the archives



Terminology

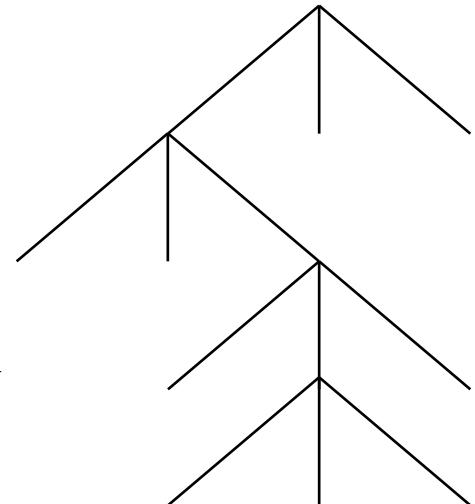
- Repo (or archive)
 - The version control system database
- Working File
 - A (source code) file ready for use
- Version
 - A file at some point in time
- Revisions
 - Changes of, or corrections to, a version
- Delta (diffs)
 - Representation of the differences between two versions
 - Allow the reconstruction of versions from each other

Basic Operations

- Check-out or Get or Pull
 - Get an (editable) version onto the file system
 - Allows user to say which version
- Check-in or Commit
 - Update a file in the archive
 - Allows the user to describe the nature of the changes
 - Auto-increments a version number (or creates a new hash)
- Some have Push (to a remote repository)
- Administration
 - View logs and view differences
 - Locking mechanisms / Access control

Branches

- Multiple divergent development paths
 - Debug one release while working on another
 - Make substantial changes nondestructively
- Trunk
 - A linear procession of versions
- Branch
 - A path of versions away from a trunk
- Version number
 - A trunk, branch, and version
 - E.g. 1.3.2.2



Keywords Embedding

- Some allow substitution markers in files
 - Most useful of which is: the version number
 - Which can be used in “about” boxes
 - Not all version control systems support this

Lock Management

- Some version control systems:
 - Distinguish get from check-out
 - Get: read only access
 - Check-out: read/write access
 - Prevents multiple users editing the same file at the same time

When to use Version Control

- Always
- When an audit trail is needed
 - Always
 - Who changes what, when and why
- For group development
 - Always
 - Prevents file breakages
 - Manages multiple concurrent edits
 - Manages change merging (multi-way)

When to use Version Control

- When multiple configurations exist
 - Always
 - Different operating systems
 - Requires multiple concurrent changes
 - Manage file system transport to different OS (cr/lf, etc.)
- For non-programming tasks
 - Always
 - Writing and publishing (including HTML)
 - E.g. GitHub does version control on a project's wiki!
 - Engineering (drawings)
- Always

Rewards

- Prevents accidental breakages
- Prevents accidental losses
- Base-lining before experimentation
- Centralized auto-builds (Continuous Integration)
- Centralized backups
- Becomes a developers “safety-net”

Examples

- SCCS – Source Code Control System
 - 1972, Marc Rochkind at AT&T Bell Labs
 - GNU version
 - Compatibly Stupid Source Control (CSSC)
 - Stores diff and rebuilds each file each time
 - No multi-way merging
- RCS - Revision Control System
 - 1982, Walter F. Tichy at Purdue University
 - 1991, RCS was licensed under GPL
 - Designed to be used by multiple developers
 - Stores latest version and reverse diffs

Examples

- Subversion
 - Open source replacement for CVS
 - Subversion is still quite popular!
- Tons of others
 - PVCS
 - SourceSafe
 - TeamSuite
- Distributed version control
 - Git
 - Mercurial

Distributed Version Control

- Typified by: Git and Mercurial
- Designed for large or distributed development groups
- Each developer
 - Has a snapshot (clone) of a repo (typically from a central archive)
 - Protects against loss (because it's also a backup)
 - Check in and out of their local archive (commit)
 - Synchronise with the origin (push)
- Advantages:
 - Allows developers to work off-line
 - Faster because syncs are usually local
 - Allows private experiments without branching the central repository
- Disadvantages
 - Merging is difficult and time consuming
 - Initial setup (cloning the archive) takes time

Git

- To create a repo
 - Easiest done at GitHub (or Bitbucket, etc.)
- To create a local copy of a repo
 - `git clone`
 - E.g.: `git clone https://github.com/andrewtrotman/JASSv2.git`
- To update your local copy of the repo (and files) to that on GitHub
 - `git pull`
- To commit *all* your local changes
 - `git commit -a -m "message"`
- To put your changes onto GitHub
 - `git push`
- To add a file to the repo
 - `git add <filename>`
- To see what changes you're made (and not yet checked in)
 - `git status`

Git

- If you want to make changes without affecting others, *fork* their repo. You can do this directly on GitHub. Go to their repo and click fork!
- Clone your fork, make your changes, commit to your forked repo

Keep Your Fork Up-to-Date

The screenshot shows a GitHub repository page for `andrewtrotman / sarcasm-detection`. The repository is a fork of `prasy/sarcasm-detection`. The 'Code' tab is selected. A red circle highlights the status message: "This branch is 1 commit behind prasy:master". Another red circle highlights the "Sync fork" button. A third red circle highlights the "Update branch" button.

This branch is 1 commit behind prasy:master.

Sync fork

Update branch

Code master 1 branch 0 tags

Contribute Compare Sync fork

This branch is out-of-date

Update branch to keep this branch up-to-date by syncing 1 commit from the upstream repository.

Learn more

Compare

Update branch

Readme

0 stars

0 watching

1 fork

Releases

No releases published

Create a new release

Packages

No packages published

Publish your first package

Languages

Python 100.0%

Detecting Target of Sarcasm Using Ensemble Methods

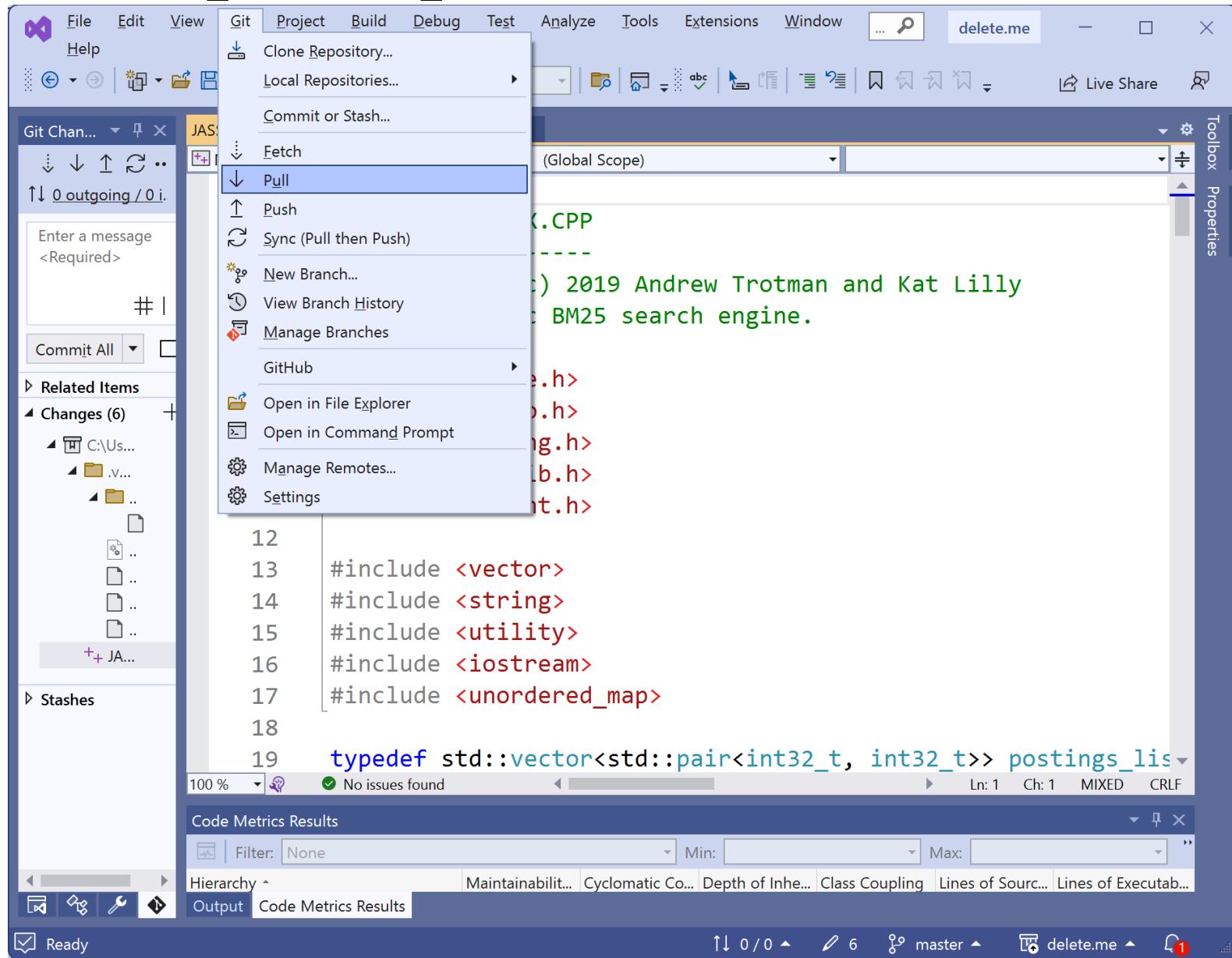
Code related to the ALTA's 2019 Shared Task - "Detecting Target of Sarcasm using Ensemble Methods". Our work was presented in ALTA 2019 The 17th Annual Workshop of the Australasian Language Technology Association

The link to the paper can be found -> <http://bit.do/pradalta>

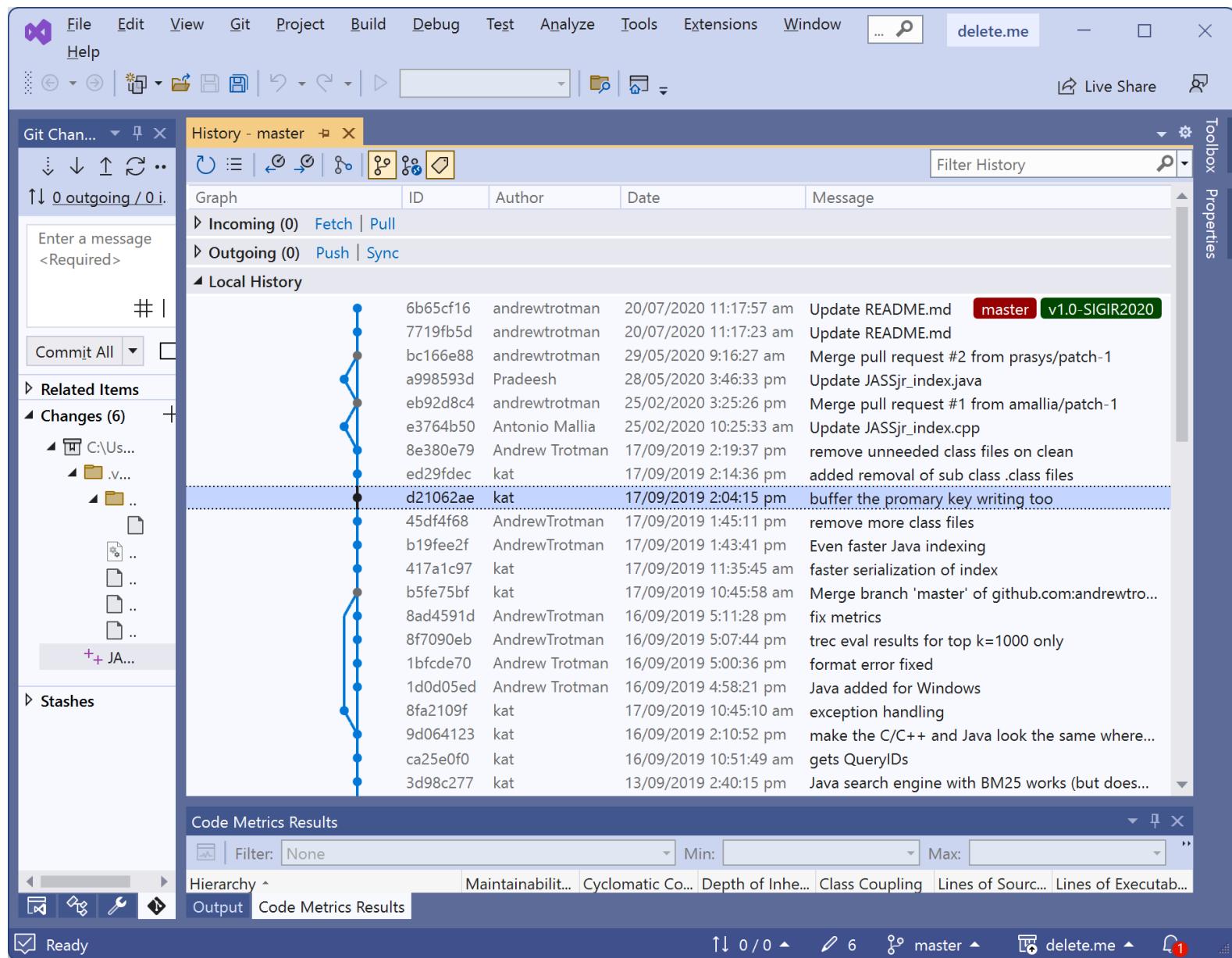
Git and Visual Studio (VS)

- By connecting git / GitHub to your IDE you can do all the “command line stuff” directly from the IDE including:
 - Commit
 - Push
 - Pull
 - View histories
 - Branch
- Visual Studio (and Xcode) have this built-in

Git pull / push / commit in VS



Git history in VS



Git diff in VS

- Differences are shown side-by-side with the differences highlighted

The screenshot shows the Visual Studio interface with the following details:

- Git Changes - delete.me** window:
 - Branch: master
 - Changes: 1 change (2 additions, 1 deletion)
 - Message: Enter a message <Required>
 - Commit All, Amend buttons
 - Related Items: Changes (6) and Stashes
 - Changes list:
 - C:\Users\andrew\programming\delete.me (A)
 - .vs\delete.me\vt1 (A)
 - ipch\AutoPCH\4a1b8ecb6e37e4fc (A)
 - JASSJR_SEARCH.ipch (A)
 - Browse.VC.db (A)
 - Browse.VC.db-shm (A)
 - Browse.VC.db-wal (A)
 - Browse.VC.opendb (A)
 - + JASSjr_search.cpp (M)
- Diff - JASSjr_search.cpp* vs JASSjr.search.cpp*** window:
 - Shows 1 change (2 additions, 1 deletion).
 - Staging controls: Stage, Show staging controls.
 - Left pane: JASSjr_search.cpp (Index) - Miscellaneous Files, Global Scope.
 - Right pane: JASSjr_search.cpp (Working tree) - Miscellaneous Files, Global Scope.
 - Code differences:

```
1  /*
2   * JASSJR_SEARCH.CPP
3   * -----
4   * Copyright (c) 2019 Andrew Trotman and Kat Li
5   * Minimalistic BM25 search engine.
6  */
7  #include <math.h>
8  #include <stdio.h>
9  #include <stdlib.h>
10 #include <string.h>
11 #include <stdint.h>
12 #include <sys/stat.h>
13 #include <sys/types.h>
14 // #include <string>
15 -#include <string>
16 #include <vector>
17 #include <iomanip>
18 #include <iostream>
19 #include <unordered_map>
20
21 /*
22  * CONSTANTS
```

The line `-#include <string>` is highlighted in red, and the line `+// #include <string>` is highlighted in green.
- Code Metrics Results** window at the bottom.

Pull Requests

- When you’re “finished” with the changes to your local repo (commit / push), issue a *pull request* on GitHub
- The original repo owner will receive a request to merge and can choose to do so (or to ignore you)
- They can review the code and choose whether to accept or reject your changes

Example GitHub Pull Request

The screenshot shows a GitHub pull request page for the repository `andrewtrotman/JASSv2`. The pull request is titled "Update SWIG File to Handle Python Strings #15". It is an open pull request with 7 commits from the branch `prasys:master` into the `andrewtrotman:master` branch.

Conversation: prasys commented 1 minute ago: The current `anytime()` method does not handle pythons string objects. This is to fix the C++ to python string object conversion tested on : python 3.7

Commits:

- prasys added 7 commits 6 months ago
 - Update PyJASS.swg (Verified) 81ed021
 - Update PyJASS.swg (Verified) da3b66e
 - Update PyJASS.swg (Verified) e4e85d2
 - Merge branch 'andrewtrotman:master' into master (Verified) c4f165d
 - Update PyJASS.swg (Verified) 85df398
 - Merge branch 'andrewtrotman:master' into master (Verified) eee9393
 - Merge branch 'andrewtrotman:master' into master (Verified) 412d81f

Reviewers: No reviews. Still in progress? Convert to draft.

Assignees: No one—assign yourself.

Labels: None yet.

Projects: None yet.

Milestone: No milestone.

Development: Successfully merging this pull request may close these issues. None yet.

Notifications: You're receiving notifications because you're watching this repository. Unsubscribe. Customize.

Branch Status: Some checks haven't completed yet (1 in progress and 1 pending). Details:

- Codacy Static Code Analysis: In progress — Codacy Static Code Analysis
- continuous-integration/appveyor/pr: Pending — Waiting for AppVeyor build to complete

Comment Section: Write, Preview, Close pull request, Comment.

Remember, contributions to this repository should follow our [GitHub Community Guidelines](#).

ProTip! Add .patch or .diff to the end of URLs for Git's plaintext views.

Best Practice

- Always use version control
- Integrate it with your IDE
 - The next tutorial
- Do not share the sandbox
- Do not work outside the sandbox
- Do not bring contaminants into the sandbox
- Keep the sandbox in sync with the repository
- Check-in little and often (at least daily)
 - Keep the for up-to-date
- Build early and often

Best Practice

- Label each release
- Branch after each release
- Branch before every bug fixing period
- Assign ownership to trunks and branches

COSC345

Software Engineering

Continuous Integration

and

Continuous Delivery

Outline

- What is CI and CD
 - Often CI is used to refer to CD too
- What you might use CI to achieve
- Integration between CI and git
- Contrasting some web-based CI offerings
 - GitHub, Bitbucket, and GitLab
- Virtual machines and Docker containers

Continuous integration & delivery

- Continuous integration (CI) was originally about avoiding branches becoming difficult to merge
 - Merging small code chunks into your main code base
 - “merge little and often” was the mantra
- Now CI is about having automated processes that assist with the quality control of your product
 - Ensuring that fixing a bug doesn’t introduce a new bug
 - By running regression tests
 - Providing immediate feedback on the health of the code
 - By running unit and integration tests
 - Deliver the artefacts produced by your build process
 - Ready for deployment

What might you do with CI?

- **Compile** your code
 - Find errors quickly
- **Link and build**
 - Find packaging problems
- Extract **documentation**
 - Doxygen, etc. (see CodeDocs.xyx)
- Perform **tests**
 - You do have unit tests, right?
- Perform **Static analysis** of your code
 - e.g. Coverity Scan, Codacy, etc
- Update **status indicators**
 - Github badges, send email, post on Slack etc.
- **Deploy** your code (i.e. Continuous Delivery)
 - To a website or an online store
- Pretty much anything
 - It can be **app-specific**

Good Practice

- Lets imagine a common scenario: I have a GitHub project and I'm working on my source code and I make a change then commit then push
- How do I know I haven't broken anything?
 - How do I know what is dependant on my code
 - And therefore might now also be broken
- I solve this by running unit tests, integration tests, regression tests, and so on. I also want to update the documentation. This takes time and I'm busy
- So I set up continuous integration system that does all the work on every push (connected to GitHub by “hooks”)

CI uses hooks

- Almost all version control systems include ‘hooks’
 - Running user-defined code that is run at certain points in the commit or push process
- CI in git often uses the **post-commit** hook
 - Run some code after each and every commit
- Many other hooks available:
 - pre-commit hook can make commits fail based on scripts
 - post-checkout, pre-push, post-merge, commit-msg,
 - **post-receive**
 - Happens on a server after a git push

CI systems

- GitHub does CI with **GitHub Actions**
 - Compile, build, and run tests on GitHub
- But can trigger other CI tools such as
 - Coverity Scan, Codacy,
 - Static source code analysis systems
 - CodeCov
 - Code coverage system
 - CodeDocs
 - Automatically run Doxygen (etc.) and host the documentation

Good Practice on GitHub

- Set a hook on GitHub so that every push results in a call to a CI system
- For example, to
 - Build your code
 - Send an email notification on each failure to build
 - Post to a Slack channel on every push

Bitbucket & GitLab

- Bitbucket has “pipelines” directly integrated:
 - Each **step** of your bitbucket-pipelines.yml definition starts a new Docker **container** with a clone of your repository and runs the contents of the respective **script** section in that container
 - Templates available: PHP, Java, Python, Ruby, C#, C++, etc.
- GitLab also has “pipelines”:
 - A pipeline is a group of **jobs** that are executed in parallel within **stages** defined within your .gitlab-ci.yml file.
 - Jobs are executed on **runners** that you register with GitLab
 - GitLab Runner can use Docker containers

Cross platform development

- Building software that runs on multiple OSs or different versions of the same OS?
 - The “it works on my computer” bug
- For **reproducibility** we need to control (at least):
 - Operating system
 - Compiler version
 - Libraries used
 - Etc.
- But we don’t want dozens of build machines in our CI
 - But we’re not (too) worried about build times
- A perfect use for **virtual servers** (virtualisation)

Websites relying on CI / CD

- If your code compiles and passes its tests then you want to “push it” to your “provider”, especially if you’re a web site.
- Example:
 - GitHub Pages provide a means for git repositories to appear as static websites through Continuous Delivery
 - GitHub Pages hosts HTML (etc.) content straight from git repositories, but CI/CD actually involves Jekyll
 - Jekyll is a static site generator with templates
 - We could host our documentation on GitHub Pages(or wikis) after it is automatically extracted after each successful push

Virtual machines (VMs)

- ‘**Host**’ runs ‘**guest**’ virtual machines as OS processes
 - Guests (typically) do not know that they are VMs
 - Can facilitate emulation of non-native hardware
 - New CPUs and software (paravirtualisation)
 - VMs now run at near-native speeds

Virtualisation hosts

- Full OS virtualisation
 - **VirtualBox**
 - Free, mostly open source, provides GUI
 - **QEMU**
 - Both emulator (dynamic translation) and virtualiser
 - **Vmware**
 - An industry leader with many products
 - **Xen**
 - Open source hypervisor
 - **KVM**
 - Virtualisation within Linux main-line kernel
- Non-GUI link to a guest VM
 - **Vagrant** system above VirtualBox, etc. – full virtualisation
 - **Docker** is a host for “containers” – not full virtualisation

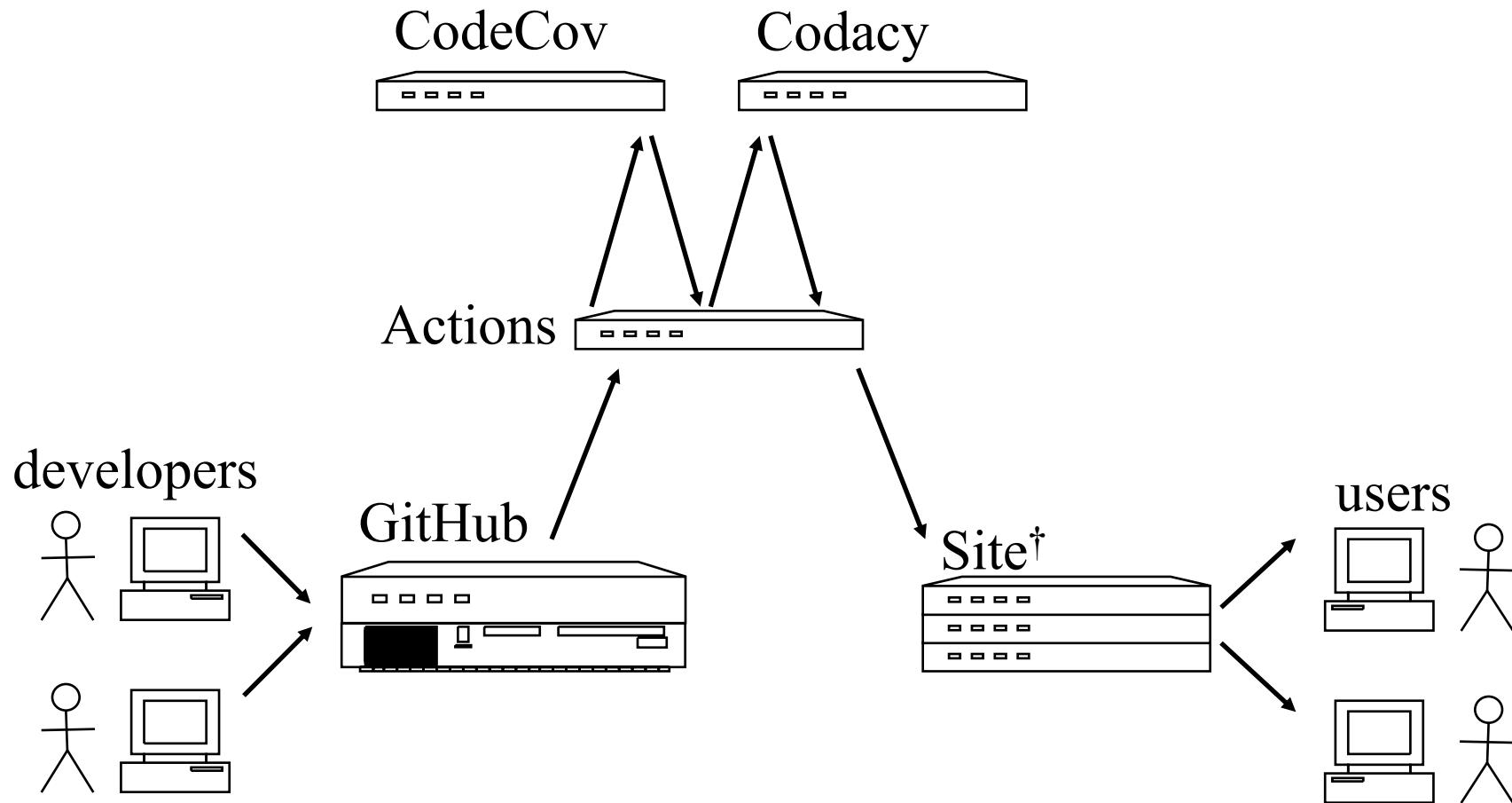
Containers

- Containers are like VMs, but **share the Linux kernel**
 - This makes them memory efficient and very quick to start
 - Can link parts of their storage and network to the host OS
 - Also easy to link containers to each other as components
 - Filesystems used by the container are distributed efficiently
 - Common ‘layers’ can be factored out, such as a base Linux layer
 - Over the base layer, just store the delta. e.g. a DB container

VMs, Containers, and CI/CD

- With containers you can use one host to run all the different OSs (and versions of OSs) you want to build on
- With VMs you can test on different versions of different OSs without having a different machine running each

CI/CD workflow



† Site might be an app store or a live web site

Conclusions

- Hook GitHub to GitHub Actions
 - As part of your project
- Common approach approach
 - Developers work on forks
 - Forks merged to origin by a robot
 - After all the tests are run
- Others
 - Push to live site each time a push to master succeeds
 - Can easily wind back if a bug is found

COSC345

Software Engineering

Code Review

Outline

- Formal versus informal code review
- Formal process: parts and considerations
- Tools to help with code review
 - GitHub pull requests
- Human considerations
- Peer review and open science
- Example (Apple's SSL bug)

Informal Code Review

- All programmers spend time *informally* reviewing code
 - You are usually working with libraries and APIs
 - Need to understand what the libraries do
 - Documentation often includes examples
 - » That have been reviewed by many people before being published
 - You review your own code while it evolves
 - Refactoring, refining, extending, etc.
 - Your unit tests demonstrate the expected way to use your units (classes) and others will look to your tests as best practice

Informal Review Processes

- Processes that don't produce explicit reports
 - Testing
 - You review your code while building unit tests
 - If it is hard to write the tests the code is hard to use
 - Pair programming
 - Protect against *your* typical errors
 - Code inspections
 - Another developer eyeballs your code
 - Code walk-throughs
 - Describe your code to another developer
- These methods:
 - Support best practice in programming
 - Modern tools can help with much of this

Formal Code Review

- *Formal* code review involves an agreed procedure
 - Involves multiple people
 - The developer
 - Other developers in the team
 - Usually a senior and or long-standing member of the team
 - Might be
 - Synchronous (involve formal meetings)
 - Asynchronous (done via email or slack or other mechanisms)
 - Almost certainly requires preparation work to be done
 - By the programmer whose code is being reviewed
 - By the reviewers to understand the context of the code
 - Usually formally documented (generates an audit trail)
 - Who were the reviewers
 - What did they say about the code being reviewed

Formal Review Process

(Fagan Inspection)

- Fagan's 1976 paper describes rates in LOCs/hour
 - **Overview** (500)
 - Review team quickly surveys the code and assigns roles
 - **Preparation** (100)
 - Participants review their assignments
 - **Inspection** (130)
 - Team steps through code under review
 - **Rework** (20)
 - Inspection report's issues are resolved
 - **Follow-up** (N/A)
 - Check on status of fixes
- Focus is finding major and minor errors (bugs)
- Also looks for absent code (missing cases)

Preparation for Formal Review

- Provide an overview for reviewer to follow
 - Which files contain modifications
 - What are the justifications for each of the changes made
 - This can come from the Version Control system
- Can use phases of development as formal structure
 - Design changes or additions
 - If the change is large enough
 - Implementation changes or additions
 - Does the code do what the design requires (and is it complete)
 - Testing
 - Is the code thoroughly tested and do the tests exemplify use
 - Can help other team members gain project knowledge

Coverage by Inspection

- Code review uses human resource
 - This must be prioritised or the review won't happen
- Informal code review can be used as a random-check
 - If programmers know their code might be reviewed then it can increase quality (as the programmers know their code might get looked at by others)
- Modern tools help facilitate 100% review coverage
 - Many organisations require *all* production code be reviewed
 - Some projects required multiple levels of review: it pays off
- The cost of code review is lower when inspecting changes than when inspecting new code

SmartBear Advice

- Guidelines from the SmartBear blog
 - Review fewer than 400 lines of code at a time
 - Inspection rates should involve fewer than 500 LOC per hour
 - Do not perform code review for more than an hour at a time
 - Set goals and capture metrics
 - Authors should annotate source code before the review
 - Use checklists
 - Establish a process for fixing defects found
 - Foster a positive code review culture
 - Embrace the subconscious implications of peer review
 - Practice lightweight code reviews
 - Lightweight (informal) review takes less than 20% the time of formal reviews and finds just as many bugs!

What to Look For

- Different things code review can look for:
 - **Deviations from house style**
 - Insufficient documentation
 - Indentation errors (c.f., “goto fail;”)
 - **Difficult to understand code**
 - Likely to have bugs
 - **Efficiency problems**
 - **Logic errors**
 - **Security vulnerabilities**
 - Inspection can be more specific
- Know your team: people may repeat errors
 - A good reason to keep and re-read your review records

Documentation From Code Review

- Process:
 - What was the purpose of the review
 - Who did the review and when
 - What were the results of the review
 - Reviews need to sign-off when finished
 - And takes some responsibility for the code at that point
 - Be careful to avoid a blame culture
- Content:
 - Comments (discussion) will be made in the review
 - Ideally related directly to the code
 - Important to document and prioritize changes
- Follow up:
 - Review may initiate code re-write or change

Tools

- Code review may be connected to version control
 - Commits can carry comments (so use them effectively)
 - Comment on the change, and why
- Code review may be connected to bug tracker
 - Which bug, fixed when, by who, and how
- Version control may be connected to bug tracker
- Some parts of code review can be mechanised through continuous integration
 - Unit, integration, and regression tests
 - Coverage
 - Static analysis
 - Etc.

GitHub Pull Requests

- Other sites have similar technology
- GitHub pull requests
 - Highlight all of the source code changes made
 - Give easy access to history
 - Allow comments near code
 - Leave questions
 - Give reactions
 - Help resolve conflicts before pull request is submitted
- Other tools plug into GitHub
 - Codacy checks for
 - Style violations
 - Duplicate code
 - Etc.

Human Considerations

- Code review can be highly stressful at first
 - It can be *uncomfortable* to expose your creative work
 - May have asymmetry in expertise of developer and reviewer
 - What might happen if reviewers are all less experienced?
 - What might happen if reviewers are all more experienced?
- Code review should focus on *problems*
 - Whilst ensuring that team members learn and evolve
- It is essential to establish an environment of *positive* feedback

Peer Review and Open Science

- Peer review is key for quality assurance in research
 - Other researchers reproduce and cross-check research results
- Growing push and enthusiasm for **open science**
 - Taxpayers fund university/government research
 - And so should see the results and the tools that gave those results
 - Open data not very useful without knowing how it was made
 - Increasing amounts of software in research
 - It should all be checked
- Code review is peer review
 - may it become ubiquitous

Apple's SSL Bug (Feb 2014)

```
if ((err = SSLFreeBuffer(&hashCtx)) != 0)
    goto fail;

if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
    goto fail;
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;
```

Apple's SSL Bug

- Was blamed on missing curly braces

```
if ((err = SSLFreeBuffer(&hashCtx)) != 0) {
    goto fail;
}
if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0) {
    goto fail;
}
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0) {
    goto fail;
}
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0) {
    goto fail;
}
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0) {
    goto fail;
}
    goto fail;
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0) {
    goto fail;
}
```

Apple's SSL Bug

- What is the real problem?
- According to Tobias Goeschel:
 - Just try the damn thing and see if it works
 - Write an automated test
 - Have someone else review your code
 - Pair program
 - Conditionals should express what you actually want to check
 - Don't copy and paste code
 - Make your methods do only one thing
 - Use readable, descriptive variable names

Apple's SSL Bug

- Goeschel re-writes that code as:

```
if ((err = SSLFreeBuffer(&hashCtx)) == 0)
    if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) == 0)
        if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) == 0)
            if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) == 0)
                if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) == 0)
                    err = SSLHashSHA1.final(&hashCtx, &hashOut);

if (err)
    goto fail;
```

- Then observes that this can be simplified to:

```
if ((err = SSLFreeBuffer(&hashCtx)) == 0 &&
    (err = ReadyHash(&SSLHashSHA1, &hashCtx)) == 0 &&
    (err = SSLHashSHA1.update(&hashCtx, &clientRandom)) == 0 &&
    (err = SSLHashSHA1.update(&hashCtx, &serverRandom)) == 0 &&
    (err = SSLHashSHA1.update(&hashCtx, &signedParams)) == 0 )
    err = SSLHashSHA1.final(&hashCtx, &hashOut);

if (err)
    goto fail;
```

Apple's SSL Bug

- What is the overall purpose of code review:
 - Take a piece of code
 - Make it easier to read
 - Reduce the complexity
 - Identify bugs
 - Replace the old code with better code
 - And, of course, make sure it still works
- But don't forget the other tools at your disposal
 - Continuous Integration, testing, static analysis, etc.
- Oh, and yes, static analysis should have identified the SSL bug because the extra `goto` results in lines of code that are not executed – something a static analysis tools should catch (and many compilers do catch)

References

- The Apple SSL bug (and relevant to the whole course)
 - T. Goeschel. 2014. Reflections on Curly Braces – Apple’s SSL bug and what we should learn from it.
<https://blog.codecentric.de/en/2014/02/curly-braces/>
- Code review (short and relevant)
 - SmartBear. (visited 15/9/2020). Best practices for code review.
<https://smartbear.com/learn/code-review/best-practices-for-peer-code-review/>
- Historic Context
 - M. E. Fagan. 1976. Design and code inspections to reduce errors in program development. IBM Syst. J. 15(3):182–211

COSC345

Software Engineering

Testing
Code Coverage
Static Analysis

Outline

- Various testing processes
 - *ad hoc* / formal / automatic testing
- Unit tests
- Code coverage metrics
- Design by contract
- Integration testing
- Regression tests
- Static analysis
- Test driven development

Ad hoc / Formal / Automated Testing

- All developers apply *ad hoc* testing of some sort
 - Necessitated by running and using the code
- **Formal testing** approaches offers other benefits
 - Make it easier to protect against regressions in code
 - Losses in functionality from bugs in new code
 - Provides an informal method of documenting use
- **Automated tools** can help with testing
 - Increasingly complex simulations accelerate bug discovery

Unit Tests

- Divide code into **units** and built **tests** for each unit
 - Tests are independent of each other, where possible
 - The tests should test **all** functionality of the unit
 - Test coverage of the unit's functionality should be complete
- Tested code should behave as if in production, even if it is interacting with **external resources**, e.g., databases
 - If necessary, tests **can** usefully be run in some sort of framework that links to stubs or proxies for external interactions
 - Although this means that another model is being included that captures the behaviour of those external resources
 - And this is, itself, prone to errors

Unit Tests in OO languages

- Classes are the obvious unit in OO
 - Other possible units:
 - Components, abstract data types, source code files, etc.
- Objects provide a boundary for state and behaviour
 - Low coupling, high cohesion, etc.
- Language facilities can help support testing needs
 - Polymorphism allows hiding test / production differences
- Well-designed objects often have low code length
 - OO *refactoring* should lead to methods with fewer LoC

Structure of a Unit Test

- Ideal structure is:
 - Setup
 - Execution
 - Validation
 - Clean up
- Common setup and clean up logic shared between tests
- Clean up allows another test to run right away
 - Don't forget to clean up otherwise the test has global effect
- There are dozens of test harnesses
 - They are sometimes integrated into the language and IDE

Cost / Benefit Of Unit Testing

- Cost
 - Testing can involve many lines of code
 - Can be 50% or more of the code base
 - And those lines of code **are not immune** from errors
 - Tests can take time to run
 - But Continuous Integration alleviates much of that time
 - But how long were you planning to be debugging code?
- Many benefits
 - Find bugs prior to **system integration**
 - Baseline against which to **check refactoring** safety
 - Baseline against which to **detect regression errors**
 - **API documentation** by indicating key concerns

Test Coverage

- What proportion of your code has been tested?
 - Do you want to release untested code?
- Many coverage metrics
 - **Functions**
 - Has every method / function / routine been called?
 - **Branches**
 - Have all paths from conditional statements been run?
 - **Conditions**
 - Have all expressions evaluated to both true and to false?
 - **Statements**
 - Has each statement been run?

Target Code Coverage Values

- Increasing coverage gets progressively harder
 - How much harder depends on the code being tested
- Safety critical systems may require 100% coverage
 - With caveats on what ‘coverage’ means
- Heuristics to use to prioritise testing
 - Have **loops** been skipped; run once; or run many times
 - Has **data flow** of variables been explored
 - Have all possible **exit points** of a function been tested
- Some lines of code cannot easily be tested
 - What to do when there is a fault writing to disk?
 - You can simulate this, but it’s diminishing returns

Code Coverage

- codecov.io will house your coverage reports and allow you to interact with them using a web browser
- You can connect codecov.io to your CI (GitHub Actions)

codecov.io

The screenshot shows the codecov.io interface for a GitHub repository named 'andrewtrotman/cosc345CI'. The repository has a green status badge indicating 'CI Passed' and a commit hash 'b6d6eb5' associated with 'master' and 'd69a989'. The coverage summary at the top right shows 57.89% coverage. Below the summary, there are tabs for 'Diff', 'Files' (selected), 'Build', and 'Graphs'. The main area displays the source code of 'AppDelegate.swift' with line numbers and annotations. Lines 16, 17, and 18 are highlighted in green, while lines 29, 30, 31, and 32 are highlighted in pink. A legend indicates that green means 'Covered' and pink means 'Partially covered'. The code itself is as follows:

```
// AppDelegate.swift
// cosc345CI
//
// Created by Andrew Trotman on 13/07/20.
// Copyright © 2020 Andrew Trotman. All rights reserved.

import UIKit

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {

    func application(_ application: UIApplication, didFinishLaunchingWithOptions launchOptions: [UIApplication.LaunchOptionsKey: Any]?) {
        // Override point for customization after application launch.
        return true
    }

    // MARK: UISceneSession Lifecycle

    func application(_ application: UIApplication, configurationForConnecting connectingSceneSession: UISceneSession, options: UIScene.ConnectionOptions) {
        // Called when a new scene session is being created.
        // Use this method to select a configuration to create the new scene with.
        return UISceneConfiguration(name: "Default Configuration", sessionRole: connectingSceneSession.role)
    }

    func application(_ application: UIApplication, didDiscardSceneSessions sceneSessions: Set<UISceneSession>) {
        // Called when the user discards a scene session.
        // If any sessions were discarded while the application was not running, this will be called shortly after application
        // Use this method to release any resources that were specific to the discarded scenes, as they will not return.
    }
}
```

At the bottom, a link reads 'Read our documentation on [viewing source code](#)'.

Language Support

- The D programming language

```
class Sum
{
    int add(int x, int y)
    {
        return x + y;
    }

unittest
{
    Sum sum = new Sum;
    assert(sum.add(3,4) == 7);
    assert(sum.add(-2,0) == -2);
}
```

- Compile and run for unit testing

See <https://dlang.org/spec/unittest.html>

Design by Contract™

- Bertrand Meyer (1986)
- Components should have a formal, precise and **verifiable** interface
 - Specifications are known as **contracts**
 - Similar to a business contract
 - If you provide this **then** I'll do that (and nothing else)
- Component, class, and method documentation could state the contract
- Unit tests should check the contract
- If the **caller** breaks the contract the **called** can too
 - e.g. passing NULL to `strlen()` has undefined behavior

Design by Contract™

- Contracts normally state
 - **Acceptable** and **unacceptable** input values or types, and their meanings
 - Return values or types, and their meanings
 - Error and exception condition values or types that can occur, and their meanings
 - Side effects
 - Preconditions
 - Postconditions
 - Invariants
 - Rarely, performance guarantees, (time or space used)

See: https://en.wikipedia.org/wiki/Design_by_contract

Design by Contract™

- Side effects
 - What global, component, or class changes occur
 - E.g. file pointer changes after a call to `fread()`
- Preconditions
 - What must happen before the contract
 - E.g. file must be open for read before `fread()` is called
- Postconditions
 - What is the result of the call
 - E.g. the file pointer does not decrease in a call to `fread()`
- Invariants
 - What does not change
 - E.g. the file does not change during a call to `fread()`

Language Support

- The D programming language

```
long square_root(long x)
    in
    {
        assert(x >= 0);
    }
    out (result)
    {
        assert((result * result) <= x && (result+1) * (result+1) > x);
    }
    do
    {
        return cast(long)std.math.sqrt(cast(real)x);
    }
```

See <https://tour.dlang.org/tour/en/gems/contract-programming>

Language Support

- The D programming language
 - `invariant()` is a special member function
 - It's called
 - **After** the constructor
 - **Before** entering a member function
 - **After** exiting a member function
 - **Before** the destructor is called
- This is a **crosscutting** technique
 - Other crosscutting behavior includes: logging
 - See Aspects
 - [https://en.wikipedia.org/wiki/Aspect_\(computer_programming\)](https://en.wikipedia.org/wiki/Aspect_(computer_programming))

Language Support

- The D programming language
 - For a class that manages dates:

```
class Date
{
    private
    {
        int year, month, day;
    }
    this(int year, int month, int day)
    {
        this.year = year;
        this.month = month;
        this.day = day;
    }
    invariant()
    {
        assert(year >= 1900);
        assert(month >= 1 && month <= 12);
        assert(day >= 1 && day <= 31);
    }
}
```

...

See: <https://tour.dlang.org/tour/en/gems/contract-programming>

Integration Testing

- **Integration testing** involves tackling the complexity of the *interacting* parts of the code
 - Unit tests are intentionally insulated
 - Integration tests are intentionally not!
- Integration tests fail due to
 - Side effects of one unit on another
 - Changes in the way a unit is used
 - Changes in the environment (Windows / Mac / etc)
- Business logic should be tested within the units
- Integration tests often require substantial run time
 - May need to set up & tear down multiple resources: DBs

Testing & Continuous Integration

- Most forms of automated testing are good candidates for use within the CI framework
 - Its good to know the tests are passing on each commit
 - Build badges on project websites, etc
 - For production code, it almost always makes sense to require all tests in the test suite to pass
 - This is easily enforced in the CI before a push to master
- CI can run both unit tests and integration tests
 - Many CI systems allow tests to be collected into phases
 - E.g. only run integration tests if unit tests pass

Regression Tests

- Rerunning previous tests to ensure the latest changes have not **regressed** the software
- The current version should pass all the old tests
- As the software is developed further (including maintenance) more tests are developed and the testing is more thorough
- An effective CI will run these tests in the background and over time – no need to wait!

Static Analysis

- Static analysis examines code without running it
 - We will see codacy.com in the tutorial
 - Coverity scan “can follow all the possible paths of execution through source code (including interprocedurally) and find defects and vulnerabilities caused by the conjunction of statements that are not errors independent of each other”
- Vast range of complexity levels is seen in the tools:
 - **Simple:** apply **linting** approaches to highlight bad code
 - e.g., usual C assignment / equality confusion

```
if (myvar = blah)
```
 - **Complex:** formal methods that simplify and model code
 - Data-flow analysis, Abstract interpretation, Symbolic execution

Symbolic Execution

- Symbolic execution tracks constraints on variables
 - Does not require any instrumentation of the source code
 - Can quickly be overcome by code path structure
- Consider: `if (x < 20) {A} else {B}`
 - Forks analysis down both branches
 - A : if $x < 20$
 - B : if $x \geq 20$
 - A might then have an `if`, and so on
- A loop over a string will gather array constraints
 - e.g., a 2-char C-string, `s`, will include
 - $s[0] \neq 0; s[1] \neq 0; s[2] = 0$
 - And so on

codacy.com

The screenshot shows the Codacy application interface for a repository named "cosc345CI". The left sidebar contains navigation links: Dashboard, Commits, Issues (selected), Pull Requests, Security, Code patterns, and Settings. The main content area displays "Current Issues" for the "master" branch of the file "AppDelegate.swift". It lists four issues:

- Line 15: Limit vertical whitespace to a single empty line. Currently 3.
- Line 16: Line should be 120 characters or less: currently 142 characters.
Code snippet:

```
func application(_ application: UIApplication, didFinishLaunchingWithOptions launchOptions: [UIApplication.LaunchOptionsKey: Any]?) -> Bool {
```
- Line 23: Line should be 120 characters or less: currently 176 characters.
Code snippet:

```
func application(_ application: UIApplication, configurationForConnecting connectingSceneSession: UISceneSession, options: UIScene.ConnectionOptions) -> UISceneConfiguration {
```
- Line 31: Line should be 120 characters or less: currently 151 characters.
Code snippet:

```
// If any sessions were discarded while the application was not running, this will be called shortly after application:didFinishLaunchingWithOptions.
```

Below this, another section for "SceneDelegate.swift" shows two issues:

- Line 15: Limit vertical whitespace to a single empty line. Currently 2.
- Line 16: Line should be 120 characters or less: currently 124 characters.
Code snippet:

```
func scene(_ scene: UIScene, willConnectTo session: UISceneSession, options connectionOptions: UIScene.ConnectionOptions) {
```
- Line 19: Line should be 120 characters or less: currently 141 characters.
Code snippet:

```
// This delegate does not imply the connecting scene or session are new (see `application:configurationForConnectingSceneSession` instead).
```

A circular icon with a smiley face is located in the bottom right corner of the interface.

Test-Driven Development

- Write your tests before your implementation
 - You should explicitly test that these **new tests fail!**
- A useful goal is to turn requirements into tests
 - Provides a **contract** against which to develop code
 - Contract is minimal: no need for more than passing tests
- Most TDD approaches also enshrine refactoring
 - Of course tests are run during the refactoring process

Can You Rebuild From Clean?

- CI systems will typically do this
 - Some can support caching various resources, for speed
 - Docker-based CI tools may cache container images
- Important that you are confident your project can be rebuilt using your build process from a clean start
- Build systems often do this to avoid side-effects
 - Environment variables, library versions, etc. present on someone's computer

References

- Wikipedia articles on
 - **Unit testing** https://en.wikipedia.org/wiki/Unit_testing
 - **Integration testing** https://en.wikipedia.org/wiki/Integration_testing
 - **Regression testing** https://en.wikipedia.org/wiki/Regression_testing
 - **Aspects** [https://en.wikipedia.org/wiki/Aspect_\(computer_programming\)](https://en.wikipedia.org/wiki/Aspect_(computer_programming))
 - **Test driven development** https://en.wikipedia.org/wiki/Test-driven_development

Reminder

- Hook codacy.com up to your CI
- Hook codecov.io up to your CI
- Use GitHub Issues (or similar) in your project

COSC345

Software Engineering

Debugging

Outline

- Sources of errors
- Stats on errors
- What is debugging?
- Debugging time
- Places to look
- Using test cases
- Fixing errors
- Debugging tools
 - Msdev, Xcode, etc.
 - Bug-tracking tools

Sources of Errors

- Analysis Errors
 - Misunderstanding of the process
 - Leads to specification errors
 - Misunderstanding of the specification
 - Leads to implementation errors
- Architecture Errors
 - A design of the software that doesn't fit the needs
 - Inappropriate algorithms
 - E.g. linear search where binary search needed
- Construction Errors
 - Logic errors in the program
 - Need debugging
 - Integration errors
 - Re-implement some interfaces

Effect of Size on Errors

- Errors per KLOC increases with project size
- Approximately consistent for each developer
 - Regardless of programming language
 - So higher level languages result in fewer errors per program
 - Because there are fewer lines of code per program

| Project Size (KLOC) | Density (Err / KLOC) |
|---------------------|----------------------|
| <2 | 0-25 |
| 2-16 | 0-40 |
| 16-64 | 0.5-50 |
| 64-512 | 2-70 |
| >512 | 4-100 |

Effect of Size on Productivity

- Boehm, Gray, Seewaldt
 - Compared productivity of small and large groups
 - Small groups 39% more productive than larger groups
 - Small: 2 developers
 - Large: 3 developers

| Project Size (KLOC) | LOC / month |
|---------------------|-------------|
| <2 | 333-2000 |
| 2-16 | 200-1250 |
| 16-64 | 125-1000 |
| 64-512 | 67-500 |
| >512 | 36-250 |

How Large is Typical?

- Most teams are small
- Most developers work on large projects

| Project Size (KLOC) | % of programmers |
|---------------------|------------------|
| <2 | 5-10% |
| 2-16 | 5-10% |
| 16-64 | 10-20% |
| 64-512 | 30-40% |
| >512 | 30-40% |

| Team Size | % of projects |
|-----------|---------------|
| 1-3 | 50% |
| 4-8 | 33% |
| 8-12 | 6% |
| 12-20 | 5% |
| 20-50 | 3% |
| 50+ | 2% |

Problem and Solution?

- As a project becomes larger
 - The productivity decreases
 - The error rate increases
 - Communication takes longer
- Developers should work in small groups
 - Directed by a “chief programmer”
 - See lecture notes for “Team Organization”
- Developers should build small components
 - Divide a large project into small parts

What are bugs?

- Bug
 - A place where the program departs from the program model
 - Bugs are your fault, not the computer's or compiler's
 - Assume it's your fault because if it is you'll get the blame
- Debugging
 - The process of identifying the cause of a defect and fixing it
 - Not a *methodology* for building reliable software
 - But if you debug you'll get more reliable software
 - Removing defects reduces development / maintenance costs
 - Debugging is the hardest part of programming

Debugging is a Science

- Scientific Method
 - Formulation of a question
 - Why does it not produce the expected result?
 - Hypothesis
 - What explains the difference between expected and observed result?
 - Prediction
 - Where in the program explains the difference?
 - Testing
 - Does observation (in the debugger) match prediction?
 - Analysis
 - What has testing told us (have we found the bug)?
 - Change
 - Does changing the source code fix the bug?

Debugging

- Be suspicious of
 - New and recently changed code
 - Old code
 - That was previously shown to be buggy
 - Was awkward to write
 - Comments in comments
 - Strings in strings
 - Your own debugging tests
- Check for the “usual” errors
- Talk to other developers about the bug
 - Explaining the problem is often enough
 - A second set of eyes often helps
 - They may have seen similar bugs before
- Take a break, relax, think, go back to debugging

Using Test Cases

- Reproduce the error several different ways
 - Reproducing an error reliably is often difficult
 - Generate a variety of different test cases
 - Multiple cases that should cause the error
 - Multiple cases that should not cause the error
- Don't jump to conclusions
 - Allow multiple hypotheses to surface
- Eliminate possible causes of the error
 - The result of negative tests
- Narrow the test case as far as possible
 - This helps identify the exact cause of the error
 - Reduce the *quantity* of code being run
- Use all available data in your hypothesis

Fixing Errors

- Save the original code (version control helps)
 - You might introduce new bugs
- Fix one bug at a time
- Trust the code not the comments
- Understand the cause before you fix it
 - Don't fix the symptoms, fix the cause
 - Then the symptoms will go away
- Change the code only for good reason
- Make sure the fix is a fix
 - Reproduce the error before you start
 - Try and reproduce the error after you finish

Debugging Time

- Brooks
 - $\frac{1}{4}$ Component and early system test
 - $\frac{1}{4}$ System test (all components in hand)
- McConnell
 - Up to $\frac{1}{2}$ of the development time
- McConnell
 - Finding the bug takes 90% of debugging time
 - Fixing the bug takes 10%
- Some studies have shown that good developers can find a set of bugs 20 times the rate of other developers (for the same set of bugs)

Quality

- Debugging is where program quality matters
 - Design
 - Can the design be made less complex
 - How could it be made better?
 - Readability
 - Is your code easy to read?
 - How could it be made better?
 - Code quality
 - Is the code simple?
 - Is the code easy to follow?
 - How could it be made better?
- If a bug is hard to find or fix
 - The program is probably badly written

Dos and Don'ts

- Do
 - Avoid bugs while first writing code
 - If you're unsure use the debugger to make sure
 - Find the *cause* of the problem
 - Understand the error before fixing it
 - Learn about the kinds of mistakes you make
- Don't
 - Guess where the error is
 - Program by trial and error
 - If the error goes away it doesn't mean the bug is fixed
 - Patch buggy results
 - Debugging is about fixing errors not patching patches

The Compiler

- Flags
 - Set warning level to maximum (or near maximum)
 - Treat warnings as errors and fix them all
 - This will pay off long term
 - Agree on project wide compiler flags
 - Where possible, use multiple compilers / OS / hardware
 - They can interpret your code differently
- Warnings and errors
 - Don't trust the line number
 - Don't trust the error message
 - Don't trust the compiler's second error message
 - In difficult cases, remove code to simplify

Use An Interactive Debuggers

The screenshot shows the Microsoft Visual Studio IDE interface for a C++ project named "poly". The main window displays the source code for the file `poly_cpu.c`. A red dot marks the current line of execution at the start of the `POLY_CPU::READ()` function. The code implements memory translation and handling for a floppy disk drive controller.

```
/*
 * POLY_CPU::READ()
 */
Byte poly_cpu::read(Word addr)
{
    long address, now;
    Byte answer;

    if (prot)
    {
        if (addr < 0xE000)
        {
            address = address_translate(addr);
            answer = true_memory[address];
        }
        else
            answer = memory[addr];

        /*
         * Floppy disk drive controller (WD1771 at E010)
        */
        if (addr >= 0xE014 && addr <= 0xE01B)
        {
            switch (addr)
            {
                case 0xE014: ... // Drive register ($40 if on size 1 of disk of $00 if on side 2
    }
```

The bottom of the screen features two debugger windows:

- Locals Window:** Shows the current values of local variables. The variable `prot` has a value of `0x00000001`.
- Registers Window:** Shows the current context as `poly_cpu::read(unsigned short)`. It lists variables `addr`, `answer`, `memory[addr]`, and `this` with their corresponding memory addresses.

At the bottom right, status information includes "Ln 277, Col 33" and buttons for `REC`, `COL`, `OVR`, and `READ`.

Bug Tracking

- Vital for large projects
- Allow reporting, management, discussion of bugs
 - Allocation of bugs to
 - Developers / testers
 - Given release of the software
- Design
 - Often based on a RDBMS
 - Often have web front end
- Examples
 - Bugzilla, FogBugz, in-house solutions
 - ***GitHub Issues***
 - So use it in your COSC345 project

Good Advice for Debugging

- Mark Jason Dominus's Good Advice
 - #11915: Only Sherlock Holmes can debug the program by pure deduction from the output. You are not Sherlock Holmes. Run the ***** debugger already.
 - #11921: It could be anything. Too bad you didn't bother to diagnose the error, huh?
 - #11938: If you have ‘some weird error’, the problem is probably with your frobnitzer.

References

- A. Koenig, *C Traps and Pitfalls*
- S. McConnell, *Code Complete*, Chapter 26
- S. Maguire, *Writing Solid Code*, Chapter 4

Reminder

- Use GitHub Issues (or similar) in your project

COSC345

Software Engineering

Literate Programming

Outline

- Documentation
- Self-documenting code
- Comments
- Doxygen (Javadoc is similar)

Documentation And Comments

- McConnell
 - Most developers like writing documentation
 - If the standards aren't unreasonable
 - It is a sign of pride (just like good layout)
- Documentation
 - Outside the program
 - Functional specification
 - Product manuals
 - First version written by the chief architect
 - Revisions done by professional authors
 - Can be as much as 2/3 of the project effort (Boehm, 1984)
 - Inside the program
 - We call these comments

Comments

- Source Code Documentation
 - Detailed description of the code
 - Most likely to be correct as they change with the code
 - The best form is the code itself!
 - Aim for total legibility, completely self-documenting code
 - Use a straightforward approach to coding
 - Choose meaningful variable names
 - Use appropriate routine names
 - Use named constants (not magic-numbers)
 - Ensure a clear and consistent layout
 - Reduce the use of break & continue: use a clean code flow
 - Use simple data structures

What Does This Do?

```
for i := 1 to Num do
MeetsCriteria[ i ] := True;
for i := 1 to Num / 2 do begin
j := i + i;
while ( j <= Num ) do begin
MeetsCriteria[ j ] := False;
j := j + i;
end;
end;
for i := 1 to Num do
if MeetsCriteria[ i ] then
writeln( i, ' meets criteria ' );
```

- Taken from McConnell

What Does This Do?

```
for PrimeCandidate:= 1 to Num do
    IsPrime[ PrimeCandidate ] := True;

for Factor:= 1 to Num / 2 do begin
    FactorableNumber := Factor + Factor;
    while ( FactorableNumber <= Num ) do begin
        IsPrime[ FactorableNumber ] := False;
        FactorableNumber := FactorableNumber + Factor;
    end;
end;

for PrimeCandidate := 1 to Num do
    if IsPrime[ PrimeCandidate] then
        writeln( PrimeCandidate, ' is Prime ' );
```

- Taken from McConnell

McConnell's Checklist (adapted)

- Routines
 - Does the name describe exactly what the routine does?
 - Does it perform one well-defined task?
 - Are all separate parts separated (into different routines)?
 - Is the interface obvious and clear?
- Data
 - Do type names document the declaration?
 - Are variables well named?
 - Are variables only used for the purpose they are named?
 - Are loop variables given informative names (not: i, j, k, x)?
 - Are enumerated types used instead of flags?
 - Are constants used instead of magic values (and strings)?
 - Are constants distinguished from types from variables?

McConnell's Checklist (adapted)

- Data organization
 - Are extra variables used for clarity when needed?
 - Are all references to a variable close to each other?
 - Are data structures simple (do they reduce complexity)?
 - Is complex data accessed through access routines (get and put)?
- Control
 - Is the normal path of the code clear?
 - Are related statements grouped together (and separated by spaces)?
 - Are independent groups of statements in a separate routine?
 - Does the normal case follow the *if* (and not the *else*)?
 - Do control structures minimize complexity?
 - Does each loop perform one and only one operation?
 - Is nesting minimized (should you use a routine for that)?
 - Are Boolean operations minimized (are they readable)?

McConnell's Checklist (adapted)

- Layout
 - Does the program's layout show the structure?
- Design
 - Is the code straightforward?
 - Does it avoid cleverness?
 - Are the implementation details hidden?
 - Are domain structures used (rather than computer structures)?

Comments Are Bad!

- Comments make the code harder to read
 - Worse: you have to read both the comments and the code!
- Natural languages are imprecise
 - Programming languages are precise
- Comments are long and prone to inconsistencies
 - Programming statements are short and to the point
- How can a comment be clear if the code isn't clear?
- Out of date comments are dangerous
 - Very few people keep comments up-to-date
- Highly commented parts of code have the highest error rate
 - (Lind & Vairavan 1989)
- Don't comment tricky code, re-write it!

Comments Are Good!

- All the arguments against comments are true
 - But those weren't good comments
- Comments should be used at a high level of abstraction
 - High level languages for programming
 - Higher level of abstraction for the comments
- Comments *clarify* the *intent* of the code
 - Comments don't repeat or explain the code
- They should read like headings in a textbook
 - One short sentence in natural language
- Disagreement between code and comments?
 - Both are probably wrong!
- Comments make you think about the code
 - If its hard to explain its probably wrong

What Does This Do?

```
// write out the sums 1..n for all n from 1 to num
crnt = 1;
prev = 0;
sum = 1;
for (i = 1; i < num; i++)
{
    System.out.println( sum );
    sum = crnt + prev;
    prev = crnt;
    crnt = sum;
}
```

- Taken From (adapted from McConnell):
 - jeroen.a-eskwadraat.nl/projects/vakken/prtki/website/collegelex1.pdf

Kinds of Comments

- Repeat of the code
 - Pointless: read the code
- Explanation of the code
 - Pointless: the code probably needs a re-write
- Marker in the code
 - Used during development to mark what has to be done
 - Remove when the coding session is over?
- Summary of the code
 - Designed to help readers of the code
- Description of intent
 - Explains the purpose of the code to a reader
- All comments (except summary & intent) are redundant
 - Navigational markers are sometimes useful too

Comment Placement

- Individual lines
 - Put on the end of the line of code
 - What do these (real world) comments mean?
 - long time /* no c */
 - mov ax, 723h /* R.I.P.L.V.B. */
 - Only comment when lines are complicated (better: break it up)
 - Don't individual line comment a loop or block start / end
 - Does it apply to the `for` or the contents of the `for`?
 - Good for structure member declarations
 - Good to mark a previous error (to stop someone “fixing” it)
- Blocks of code
 - Indent with the block of code to maintain readability
 - Explain what to expect (comment before, not after)
 - Explain surprises
 - Discuss *why* not *how*
 - Avoid abbreviations

Comment Places

- Variables and types
 - Comment units and allowable ranges
 - Comment encodings (better: use an enum)
 - Comment limits on input data
 - Document each bit in a flag
 - Document global variables (why *must* they be global)
 - Relationships between variables
- Special cases
 - Exhaustively comment a compiler work around
 - How else can someone know why the code is peculiar?
 - Exhaustively comment style violations
 - Why was it necessary to break convention
- Don't comment bad code – rewrite it
 - (Kernighan and Plauger, 1978)

Routines

```
'*****  
' Name: CopyString  
'  
' Purpose: This routine copies a string from the source  
'   string (source) to the target string (target).  
'  
' Algorithm: It gets the length of "source" and then copies each  
'   character, one at a time, into "target". It uses  
'   the loop index as an array index into both "source"  
'   and "target" and increments the loop/array index  
'   after each character is copied.  
'  
' Inputs: input The string to be copied  
'  
' Outputs: output The string to receive the copy of "input"  
'  
' Interface Assumptions: None  
'  
' Modification History: None  
'  
' Author: Dwight K. Coder  
' Date Created: 10/1/04  
' Phone: (555) 222-2255  
'*****
```

Routines and Modules

- Keep comments with the code they describe
 - Better: keep comments out of the routines
- Document the source of an algorithm
 - So you can go and look it up (to fix any bugs)
- Mark routines with comments
 - Helps break-up the code and helps with navigation
- Document modules
 - Document the purpose (at the top)
 - Remember: the routines are already documented

Recommendations

- Treat comments as code
 - Keep comments indented with the blocks of code
 - Make comments look like indented blocks of code

```
/*  
     Comment  
 */
```
- Comment as you go along
 - You’re unlikely to go back and do it later
 - You need the comments while you are writing the code
- Don’t write comments that are “too long”
- Never have a quota
 - N lines of code per comment
 - Because you’ll write empty comments!

Doxxygen (etc.)

- What if we use a (documentation) programming language to write the documentation and embed that in the source code (a little language)?
 - The code and documentation are together
 - A tool could extract the documentation
 - The C (Java, etc.) compiler could ignore the documentation
- There are many many tools for doing this, but we (as a community) have tended towards the syntax used by Doxygen and Javadoc

Doxxygen (etc.)

- Doxygen generates HTML and Latex source by running over your code base and extracting the documentation
- Commands are valid C/C++/Java comments so no need to extract the source code (the compiler will ignore Doxygen commands)
- Xcode parses Doxygen syntax and makes your documentation available on the right “pane”
 - Presumably other IDEs can do this too

Basic Syntax

- Doxygen commands are in “special” comments

```
/*!  
 */
```

- Commands start with an @

```
@file  
@brief  
@details  
@param  
@return
```

- There are alternative syntaxes, and many commands

Doxxygen Example

```
/*
 * @file
 * @brief Doxygen example
 * @author Andrew Trotman
 * @copyright 2023 Andrew Trotman
 */
/*
 * @brief Core methods.
 */
class code
{
public:
    /*
     * @brief Return the maximum of the two parameters.
     * @details This method is order-preserving: if a == b then a is returned.
     * @param first [in] First of the two.
     * @param second [in] Second of the two.
     * @return The largest of the two (compared using >=)
    */
    const TYPE &maximum(const TYPE &first, const TYPE &second)
    {
        return first >= second ? first : second;
    }
};
```

Using Doxygen

- Generate Doxygen config file

`doxygen -g`

- Run over the source code

`doxygen`

- View documentation

`open html/index.html`

file:///Users/andrew/teaching/cosc345/htm

My Project

Main Page Classes ▾ Files ▾ Search

Public Member Functions | List of all members

code Class Reference

Core methods. [More...](#)

```
#include <example.h>
```

Public Member Functions

const TYPE & **maximum** (const TYPE &first, const TYPE &second)
Return the maximum of the two parameters. [More...](#)

Detailed Description

Core methods.

Member Function Documentation

◆ **maximum()**

```
const TYPE& code::maximum ( const TYPE & first,
                            const TYPE & second
                           )
```

inline

Return the maximum of the two parameters.

This method is order-preserving: if $a == b$ then a is returned.

Parameters

first [in] First of the two.

second [in] Second of the two.

Returns

The largest of the two (compated using \geq)

Doxygen commands

- There are dozens of Doxygen commands including commands for
- Marking methods as overloads
@overloaded
- Copying documentation from another method (useful for overrides)
@copydoc
- Embedding source code
@code
- Creating a main page
@mainpage

Integrate with CI

- By integrating a site like codedocs.xyz with a site like GitHub on an Action its possible to automatically generate documentation from the source code on every push
 - So write documentation and keep it up to date
 - *So do this with your project*

References

- S. McConnell, *Code Complete*, Chapter 19
- Doxygen
 - <https://embeddedinventor.com/guide-to-configure-doxygento-document-c-source-code-for-beginners/>
 - <http://www.doxygen.nl>

COSC345

Software Engineering

Plan To Throw One (or Two) Away

Outline

- First systems
- Change
- Maintenance
- Bugs
- The second system

First Systems Are Bad

- In most systems, the first implementation is barely usable
- It may be
 - Too slow
 - Too big
 - Too awkward to use
 - All three
- There is no alternative other than to start again
 - The system must be redesigned with all the problems solved
 - This can be done piece by piece, or as a total re-write
- Management is faced with a decision
 - Throw away the throw-away, or
 - Sell the throw-away

There's No Choice

- Selling the throw-away
 - Buys time, but
 - Agonizes the user
 - Is a distraction to the re-design team
 - Carries a customer support cost
 - Carries high maintenance costs
 - Creates a bad reputation for the development team
- Remember, a programmer's job is to satisfy the user
 - Delivering a throw-away is philosophically bad
 - Delivering a good product is the only way
- Plan to throw one away – you will anyway! (Brooks, 1975)

Change

- Change is a way of life in software development
- Through the life-cycle of a project
 - Specifications change
 - Customer needs change
 - The need being fulfilled changes
- This is also true of hardware development
 - But hardware *appears to be* more rigid to the user
- As we better understand a problem we change the design
- This change must be managed in order to finish a product

Managing Change

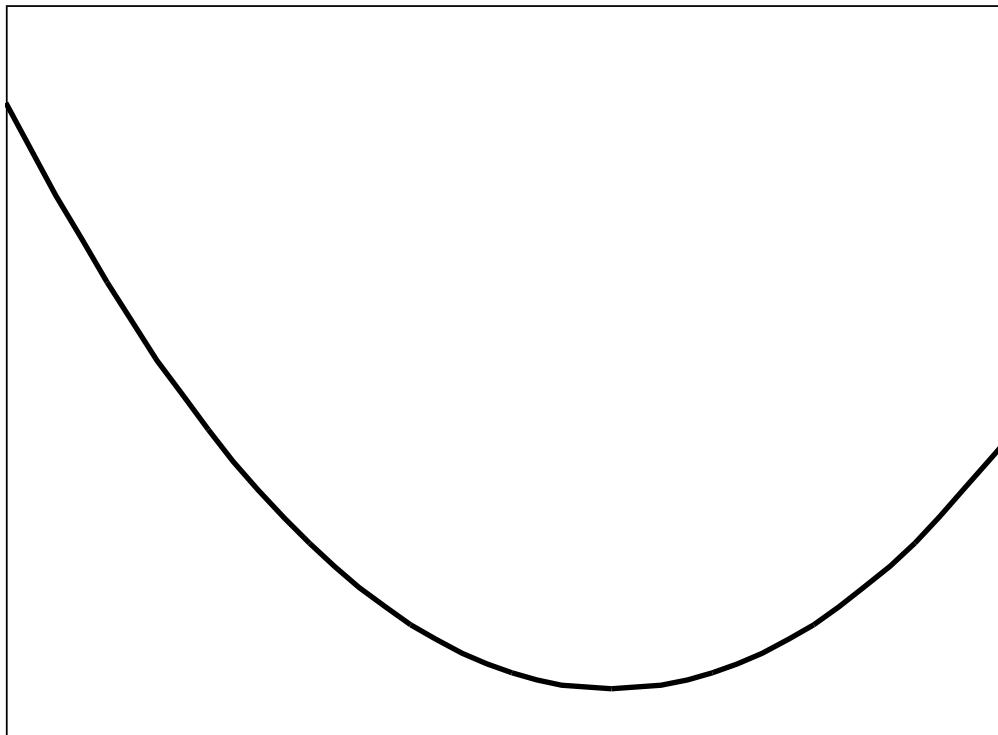
- Careful modularization
 - Objects / inheritance
 - Subroutines / methods / procedures / functions
 - Precise specification of internal interfaces
 - Standard calling sequences and parameter lists
- Use of standard:
 - Libraries
 - Objects
 - Data formats
- Literate programming
 - Reduces the error rate, makes debugging easier, but
 - Means the developers are “exposed” to criticism
 - If the organization is threatening developers won’t document

Maintenance

- Post-delivery change is called maintenance
 - Old “fixed” bugs reappear
 - New features demonstrate old defects
 - Users become familiar with the product
 - New (more subtle) bugs appear
- Fixing bugs
 - There’s a 20% – 50% chance of introducing a new bug!
 - Obvious and local behavior is easily fixed
 - Global and subtle ramifications are often ignored
 - Not often done by the original author
 - Often a trainee or new employee
 - Only way to thoroughly check a fix is regression testing
 - Which is expensive and time consuming – unless automated
- Maintenance cost is 40+% of development cost
 - More users will find more bugs

Bugs

- After release there's an initial period of identification
- Then there's a quiet period
 - This is not a stabilizing of the code-base
- Then new and subtle bugs are discovered



Successive Releases

- Building a system is entropy decreasing
 - Order is introduced to the project
- Maintenance is entropy increasing
 - Fixing bugs often results in architectural deconstruction
- Lehman & Belady discovered
 - Total number of modules increases linearly with release
 - Modules affected increases exponentially with release
 - All repairs tend to destroy the structure
 - Increasing time is spent fixing the fixes
 - Eventually all time is spent fixing fixes
 - The software has “worn out” (suffered bit-decay)

The Second System Effect

- An architect's first work is sparse and clean
 - It is done carefully and with restraint
- Piece by piece embellishments become obvious
 - These are stored away for “next time”
- Eventually the architect is ready for a new design
 - This is the most *dangerous* design
 - This design emphasizes *features*
 - Third and subsequent designs emphasize *need*
- The second system tendency is to
 - Over-design using all the ideas from the first
 - Feature pack
 - Perfect obsolete features
 - Ignore innovation

Avoid The Effect?

- You can't!
 - A second system must be built
- You can manage it
 - Be aware that you are building a second system
 - Avoid feature embellishment
 - Avoid perfecting obsolete features
 - Examine the current needs
- Appoint a third-system team leader
 - Find someone who has done it twice before
- It is the third and subsequent systems you want

References

- F. Brooks, *Mythical Man Month*, Chapter 5 & 11

COSC345

Software Engineering

CMake and Build Tools

Outline

- What are build tools
- The build process
- Make
- Advanced make
- Makedepend
- Autotools
- CMake
- Warning:
 - Make is different everywhere you go!
 - Autotools works across Unices (and MinGW)
 - CMake works across OSes (including Windows)

What is a Build Tool for?

- Scripts and automates building of the software
- Provides a consistent interface
- Manages dependencies
- Runs tests
- Packages or installs build results
- Provides parallelism and incremental building
 - cd musl && time make - 34.33s
 - cd musl && time make -j8 - 4.62s
 - touch src/malloc/calloc.c && time make - 0.44s
- Is shell a build tool?

Build Tools

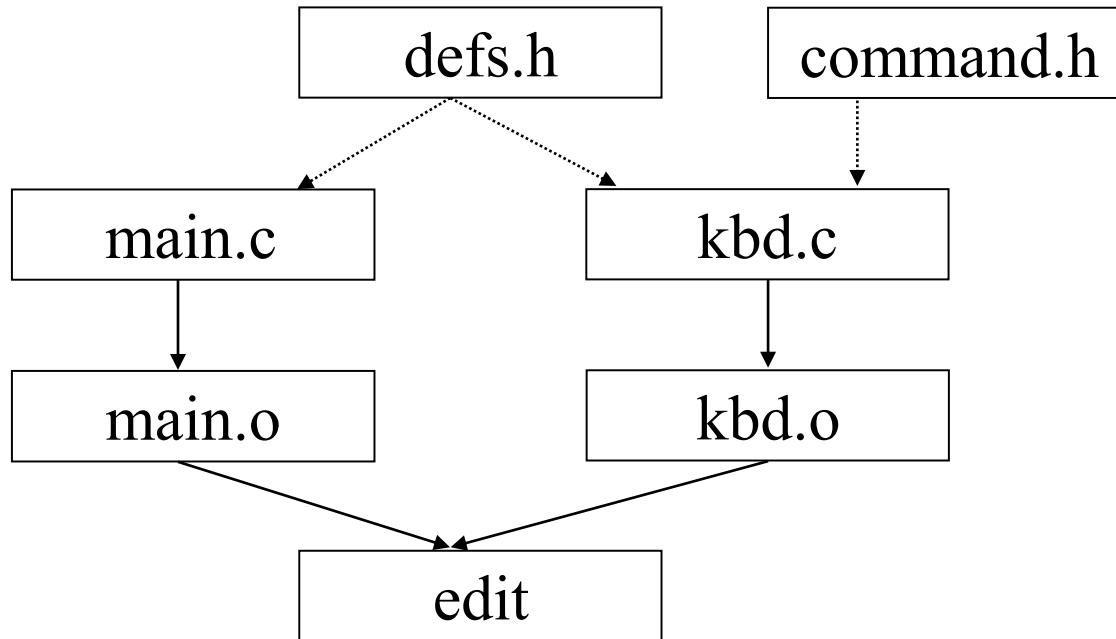
- Many languages use build tools
- C – Make, Autotools, Meson
- C++ – CMake, Build2
- D – Dub
- Java – Ant, Maven, Gradle
- JavaScript – Webpack, Vite, Browserify, Parcel, Rollup, Brunch
- Haskell – Cabal
- OCaml – Dune
- Erlang – Rebar3
- Zig – Zig Build System
- More... Bazel, Scala, Clojure, Python, etc.

Building C

- 1969 B Language (cut down BCPL)
- 1970 Unix – PDP11 16-bit address space 64kB
- 1972 C
- 1973 Unix rewritten in C
- 1976 Make
- C as smallest high-level language to write an OS
- Compile in one pass¹
- Multiple tools to limit memory usage, cpp, cc, ld
- Division of tools makes each easier to write

1) <https://bellard.org/tcc/tcc-doc.html#Code-generation>

Build Process



- `cc -c main.c`
- `cc -c kbd.c`
- `ld -dynamic-linker /lib64/ld-linux-x86-64.so.2 /lib64/libc.so /lib64/crt1.o main.o kbd.o -o edit`

Dependencies

- If a header changes then the source must be recompiled
 - If `command.h` changes then
 - recompile `kbd.c`
 - relink `edit`
- Actually, we must rebuild:
 - All files that include the header
 - All executables dependant on the object code
 - All libraries dependant on the object code
 - All executables dependant on the libraries
- And all in the right order!

What Make Does

- We need some way of scripting the build process
 - When changes are made:
 - Only re-build the parts that are necessary
 - Automatically work out which parts those are
- Make
 - Manages the creation of output files from source files
 - Uses a configuration file (makefile) to specify how
 - Uses file time-stamps to determine what's out-of-date
- Make also:
 - Enables those new to a project to build it without understanding the build process
 - Enables the developers to build:
 - Clean-up scripts
 - Install scripts

Make Rules

- A makefile is made from a series of rules

```
<target> : <dependency> ... <dependency>
           <command>
...
           <command>
```

- **<target>** is the name of the file to be built
- **<dependency>** is the name of a file needed to build it
- **<command>** is each command needed to create the target
 - Commands are indented by one tab (not spaces)
- A rule describes how and when to perform an action
- Unless specified otherwise make builds only the first target

Sample Makefile

```
edit : main.o kbd.o
      cc -o edit main.o kbd.o

main.o : main.c defs.h
       cc -c main.c

kbd.o : kbd.c defs.h command.h
       cc -c kbd.c

clean :
      rm edit main.o kbd.o
```

- **In the rule:**

```
edit : main.o kbd.o
      cc -o edit main.o kbd.o
```

- **The target is** edit
- **The dependents are** main.o and kbd.o
- **The command is** cc -o edit main.o kbd.o
 - Which will result in edit being built

What Make Does

- Read the makefile
- Generate a dependency graph for the *first* target
- Starting at the target
 - For every dependent
 - Target the dependent
 - If any dependent is more recent than the target
 - Execute the commands
 - If any target doesn't exist
 - Execute the commands
- To run make on Linux / MacOS: make
- To run make on Windows: nmake

Dependency Graph

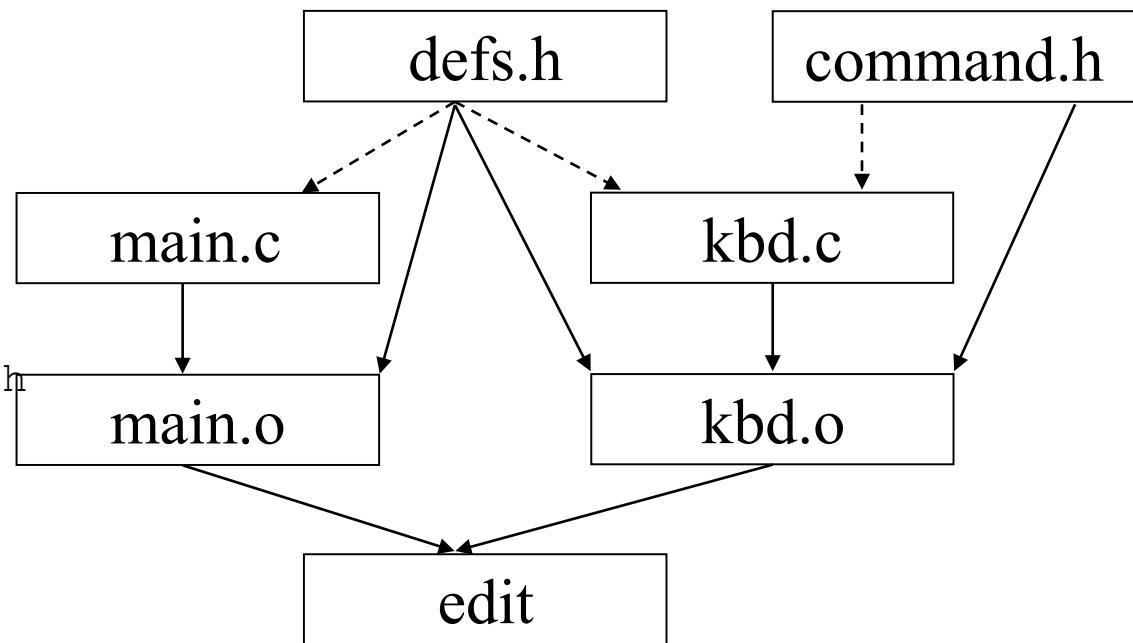
- In the example:
 - If any of the .h or .c files is “touched” then `edit` is rebuilt

```
edit : main.o kbd.o
      cc -o edit main.o kbd.o
```

```
main.o : main.c defs.h
      cc -c main.c
```

```
kbd.o : kbd.c defs.h command.h
      cc -c kbd.c
```

```
clean :
      rm edit main.o kbd.o
```



Commands

- Make will execute one or more commands specified:

- Each on separate line
 - Tab indented (not spaces)
 - Without blank lines between them

```
clean :  
    rm edit  
    rm main.o kbd.o
```

- Make does not execute commands directly, it calls a (configurable) shell to execute them
- If a command returns an error (non-zero) then make exits
 - This stops the build process at the first error
- Commands starting
 - ‘@’ are not echoed to the console (silent)
 - ‘-’ the return code is ignored (can’t produce an error)

Macros / Inference Rules

- Macros can be defined

```
objects = main.o kbd.o
```

- And used

```
edit : $(objects)  
cc -o edit $(objects)
```

- Then changes only need to be made in one place

- Some rules are inferred

```
main.o : defs.h  
kbd.o : defs.h command.h
```

- Make “knows” how to make .c files into .o files

Inference Rules

- `main.o` is made from `main.c` & `defs.h` by an inference rule:

```
$ (CC) -c $(CFLAGS) main.c
```

- Inference rules can be specified in the makefile

```
<.from><.to>:
```

```
  <command>
```

```
  ...
```

```
  <command>
```

- Example

```
.c.o:
```

```
  $ (CC) -c $(CFLAGS) $<
```

- Make **must** be told about inference rules

```
.SUFFIXES:                      #Clear the suffix list
```

```
.SUFFIXES: .c .o                  #Append to the suffix list
```

Predefined Macros

- Make has many built in default macros:
 - For compilers it includes (there are many)

| <i>Compiler</i> | <i>Flags</i> |
|-----------------|--------------|
| CC | CFLAGS |
| CXX | CXXFLAGS |

- For inference rules it includes:

| <i>Macro</i> | <i>Definition</i> |
|--------------|---|
| \$@ | Current target's full name |
| \$< | Dependent file with a later timestamp than the current target |
| \$? | All dependents with a later timestamp than the current target |
| \$* | Current target's path and base name minus file extension |
| \$+ | All dependents of the current target |

- As well as the environment variables

Other Rules

- Phony targets don't check the filesystem

```
.PHONY: clean
```

```
clean:
```

```
    rm *.o temp
```

- Multiple targets for a dependency

```
main.o kdb.o : defs.h
```

```
kbd.o : command.h
```

- Targets built by multiple commands (::)

```
libaspt.a :: kbd.o
```

```
    ar -rv libaspt.a kbd.o
```

```
libaspt.a :: another.o
```

```
    ar -rv libaspt.a another.o
```

- Makefiles can include other makefiles

```
include another.mak
```

- # at the beginning of a line is a comment

Gnuisms etc.

- Makefiles are often not portable containing GNU, BSD, etc. extensions
- Pattern Rules (common GNU extension)

```
% .o : %.c  
      $(CC) -c $(CFLAGS) $< -o $@
```

- Useful for directories in input or output

- For loops (common BSD extension)

```
.for variable [variable ...] in expression  
    <make-lines>  
.endfor
```

- Useful to apply a set of rules to a list of files

Dependencies

- The dependencies must be kept up-to-date
 - Every new header file included must be accounted for
 - This is time consuming and prone to error
- Tools exist to find the dependencies
 - gcc -M
 - makedepend
 - mkdep
 - And others too

Autotools

- Most makefiles have the same basic structure
- Generate this structure from config, a make

configure.ac

- AC_INIT([amhello], [1.0], [bug-automake@gnu.org])
- AM_INIT_AUTOMAKE([-Wall -Werror foreign])
- AC_PROG_CC
- AC_CONFIG_FILES([Makefile src/Makefile])
- AC_OUTPUT

Makefile.am

- SUBDIRS = src

src/Makefile.am

- bin_PROGRAMS = edit
- edit_SOURCES = main.c kbd.c defs.h command.h

Autotools

- Automatic dependency tracking
- Standard interface

make

make clean

make check

make install

make uninstall

make dist

- Configure installation prefix
 - ./configure --prefix ~usr
- Install to a different directory (for creating system packages)
 - make DESTDIR=\$HOME/inst install

Autotools Drawbacks

- Bad support for Windows
- Hard to debug
 - Configured using a general purpose macro language m4 – non-specific error messages
 - Developed as a set of interdependent tools aclocal, autoconf, autoheader, automake
 - Written in arcane languages Perl4/Perl5.6, posix shell, m4
 - Designed for porting between Unix platforms which no longer exist (only Linux, BSD, Mac still relevant)
 - Modern C is more consistent than 90s C

CMake

- Designed to solve the flaws in Autotools
- While keeping the good bits
- First class Windows support
- Specialised config language
- Support native build tools e.g. Visual Studio
- Target more than just make e.g. ninja
- Many builtin features e.g. remote dependency fetching
- Default to good practices e.g. only out of tree builds

CMakeLists.txt

- Minimal CMake config (with warnings)

```
add_executable(edit main.c kbd.c defs.h  
command.h)
```

```
$ mkdir build  
$ cd build  
$ cmake ..  
$ make
```

CMakeLists.txt

- Minimal correct CMake config

```
cmake_minimum_required(VERSION 3.8)
```

```
project(edit)
```

```
add_executable(${PROJECT_NAME} main.c kbd.c  
defs.h command.h)
```

```
$ cmake -S . -B build
```

```
$ cmake --build build
```

CMake Subdirectories

- Almost everyone uses a src dir

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.8)
```

```
project(edit)
```

```
add_subdirectory(src)
```

src/CMakeLists.txt

```
add_executable(edit main.c kbd.c defs.h  
command.h)
```

CMake Tests

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.8)
project(cmakeci)
```

```
include(CTest)
add_subdirectory(tests)
```

tests/CMakeLists.txt

```
add_executable(test-kbd test.c kbd.c)
add_test(test-kbd)
```

```
$ ctest --test-dir build
```

CMake Targets

- CMake is built around targets (things to build)
- Two types of targets
- Executables

```
add_executable(test-kbd test.c kbd.c)
```

- Libraries

```
add_library(kbd STATIC kbd.c kbd.h)
```

- Library types
 - STATIC – archive of objects, typically built to be linked into a program
 - SHARED – typically installed for runtime linking
 - MODULE – optional runtime extension i.e. dll

CMake Library (bad)

src/CMakeLists.txt

```
add_library(kbd STATIC kbd.c kbd.h)
```

test/CMakeLists.txt

```
add_executable(test-kbd test.c)
target_include_directories(test-kbd PRIVATE
..../src)
target_link_libraries(test-kbd kbd)
add_test(test-kbd)
```

CMake Library (good)

- Add header to lib instead, propagate it downstream

src/CMakeLists.txt

```
add_library(kbd STATIC kbd.c kbd.h)
target_include_directories(kbd PUBLIC
${CMAKE_CURRENT_LIST_DIR})
```

test/CMakeLists.txt

```
add_executable(test-kbd test.c kbd.c)
target_link_libraries(test-kbd kbd)
```

add_test(test-kbd)

CMake Scope

- Certain function calls must specify scope

PUBLIC

- Populates both properties for building and properties for using a target i.e. dependents receive the properties

PRIVATE

- Populates only properties for building a target i.e. nothing is shared with downstream

INTERFACE

- Populates only properties for using a target i.e. nothing is built

CMake Useful Functions

```
target_compile_options(first-test PRIVATE -fexceptions)
```

- Options passed to the compiler

```
target_link_options(first-test PRIVATE -lmath)
```

- Options passed to the linker

```
set_property(TARGET edit PROPERTY CXX_STANDARD 20)
```

- Special properties CMake is aware of. In this case it will trigger a compiler option. Many other properties available

```
file(COPY <files>... DESTINATION <dir>)
```

- Copy files during configure step

CMake Useful Functions Continued

`add_custom_target()`

- Custom steps during build

`find_package()`

- Find system installed libraries

`if()`

- Conditional compilation/linking

`foreach() / while()`

- Execute sequence of commands for each item in list

CMake FetchContent

- Like `find_package()` but downloads from the web

```
include(FetchContent)
```

```
set(RAYLIB_VERSION 4.5.0)
FetchContent_Declare(
    raylib
    URL
https://github.com/raysan5/raylib/archive/refs/tags/\${RAYLIB\_VERSION}.tar.gz
    FIND_PACKAGE_ARGS ${RAYLIB_VERSION}
)
FetchContent_MakeAvailable(raylib)
add_executable(${PROJECT_NAME} ${SOURCE_FILES})
target_link_libraries(${PROJECT_NAME} raylib)
```

CMake Drawbacks?

- Weird syntax from originating as a macro processor
- Functions cannot return values (assign globals)
- It's not extensible without touching C++
- But it's universal in the C++ world

References

- GNU make
 - <http://www.gnu.org/software/make/manual/make.html>
- NMAKE reference
 - <http://msdn2.microsoft.com/en-us/library/dd9y37ha.aspx>
- Cmake
 - <https://cmake.org/>

COSC345

Software Engineering

Optimisation

Outline

- Why optimise?
- Some places where software inefficiency arises
- Compiler, library and manual optimisation
- Making good use of heterogenous computing
- The 90/10 principle
- Measuring software behaviour
 - Instrumentation vs sampling
 - CPU performance counters
- Tool support
 - Profilers

Why Optimise?

- Slow software may lead to dissatisfied customers
 - It may be the user interface
 - Sluggish or unresponsive
 - This can be workflow-changing
 - While we wait for the system response, do something else
- Unoptimised software may lead to resource overuse
 - CPU utilisation is a concern for battery-powered devices
 - Memory use is relevant for resource-constrained devices
 - In a distributed system resource use is extra computers
 - And these cost
- Optimised software *may* be easier to maintain
 - Or not

Where Can Inefficiency Arise?

- Inefficiency at **programming level**:
 - Software design, *e.g.*, algorithm selection
 - Problematic programming language effects
 - Interpreted vs compiled languages
 - Difficulty expressing the need clearly
 - E.g. pointers vs array indexes
- Inefficiency at the **hardware level**:
 - Software mismatches aspects of CPU architecture
 - E.g. not using SIMD instructions in tight loops
 - Problematic memory and cache access patterns
 - Access to slow devices
- Access to **distributed resources**:
 - Network latency / bottlenecks

Inefficiency: Algorithms

- Consider the fundamental blocks in your design
 - Easy targets: searching and sorting
 - Hashing: $O(1)$
 - Linear search: $O(n)$
 - Binary search: $O(\log_2(n))$
 - Often a trade-off between space and time complexity
 - Well-built code should facilitate testing different algorithms
 - Remember: “separation of concerns”
 - Could caching or memoisation help?
 - That is: use a lookup table of answers rather than computing an answer
- Also consider performance in service dependencies
 - Indexes in databases (need to add or change and index?)
 - Can add external, scale-out caching layers

Inefficiency: Programming Language

- Object Oriented language's potential costs
 - Java will incur space and execution cost for new objects
 - Also, garbage collection may occur at unexpected times
 - Be aware of object creation within inner loops
 - Maybe reuse an existing object instance
- Copy-by-value versus copy-by-reference
 - Shallow copy vs deep copy vs using pointers
- Using an interpreted language?
 - Aim to spend time in libraries
 - Don't spend interpreter time managing loop variables

Inefficiency: CPU Effects

- Software emulation of missing hardware support
 - Unaligned integer reads
 - Emulation of missing instructions
 - Intel® Software Development Emulator
 - Emulate instructions that don't exist yet
 - » E.g emulation of AVX512 using AVX2
 - Virtualisation speed may depend on available CPU features
- Problems with code structure and CPU pipelines
 - Deep CPU pipelines may handle loops inefficiently
 - Because of branch mispredictions
 - CPUs have different pipeline structure
 - Boost performance by interleaving non-dependant instructions
- Not worth spending human time on: trust the compiler?

Inefficiency: Memory Hierarchy

- CPU registers
 - Usually best managed by compiler
 - In the past, we could use the `register` keyword in C
- Memory hierarchy includes:
 - Caches (of different sizes)
 - Streaming data may flush cache undesirably: use cache hints
 - Local RAM
 - Remote RAM
 - NUMA
 - non-uniform memory access
 - Hard Disk vs Solid State Disk
- Think about where your data is and how you access it

Inefficiency: Slow Hardware

- Get time-of-day has all sorts of optimisations
 - Finding out the time is a kernel function!
 - Most OSs apply tricks to avoid a kernel context switch
 - Accessing the real clock hardware can be slow
 - How does the computer use time to measure fast operations when getting the time is a slow operation, and the granularity of the clock may not be very high?
- GPUs and other external hardware accelerators
 - Very fast when data is loaded into them, but there may be a large cost to actually get the data there
 - Will it necessarily be faster to use the GPU?
 - May need to develop parametrised cost model (or heuristics)

Inefficiency: Distributed Resources

- Increasingly, components may be remotely hosted
 - Round-trip-time can be disruptive to the user experience
 - What is the round trip time to a server in the UK or USA?
- May benefit from asynchronous communications
 - But this will make handling failures more complex
 - Think about email: it appears to be very fast, but is actually quite slow
- May benefit from local cache
 - DNS

Types Of Optimisation You Can Effect

- What you can cause to be done automatically
 - Compiler options
 - Remember to check for changes in behaviour
 - Test with maximum optimisations *and* in debug mode
 - Use of good libraries
 - Which is best?
 - Evaluate and profile
- What you can do within your own code
 - Profiling and other measurement
 - Rewriting or reintegrating algorithms
 - Reworking your user interface

Auto-optimisation: Compiler Flags

- Almost always want maximum optimisations
 - Often makes debugging harder
 - Some optimisers will:
 - Reorder lines of code
 - Turn variables into registers (so they don't have addresses)
 - Remove “dead” code
 - Remove methods that don't get called
- But
 - May need to provide “don't touch!” hints to optimiser
 - C **volatile** keyword assumes variables change between use
 - e.g. in an interrupt handler
 - May need to signal when instruction reordering can't happen
 - Using fences or barriers (`std::atomic_thread_fence()`)

Compiler Flags

- GCC (and others) use `-O` to optimise
 - Not all compilers are equally good (GCC, Clang, Intel)
- But beware! Some optimisers make assumptions
 - Such as the absence of *pointer aliasing*
 - Which is common in C

| Option | Level | Run Time | Code Size | Memory Use | Compile Time |
|-------------------------------------|-----------------------------------|----------|-----------|------------|--------------|
| <code>-O0</code> | Compile time | + | + | - | - |
| <code>-O1</code> or <code>-O</code> | Run fast | - | - | + | + |
| <code>-O2</code> | Run faster | -- | | + | ++ |
| <code>-O3</code> | Run fastest | --- | | + | +++ |
| <code>-Os</code> | Program size | | -- | | ++ |
| <code>-Ofast</code> | Run fastest, non-accurate math | --- | | + | +++ |

Heterogeneous Computing

- Heterogeneous computing: using more than the CPU
 - **GPUs**: increasingly likely to have multi-GPU systems
 - **Vector processing units**: SIMD
 - **FPGAs**: field programmable gate arrays
 - **Neural Units**: for neural network evaluation
- Very fiddly to optimise code for each platform
 - Use libraries that automatically detect the environment
 - Intel mathematics libraries for optimised linear algebra, etc
 - Languages like OpenCL target heterogeneous systems

90/10 rule

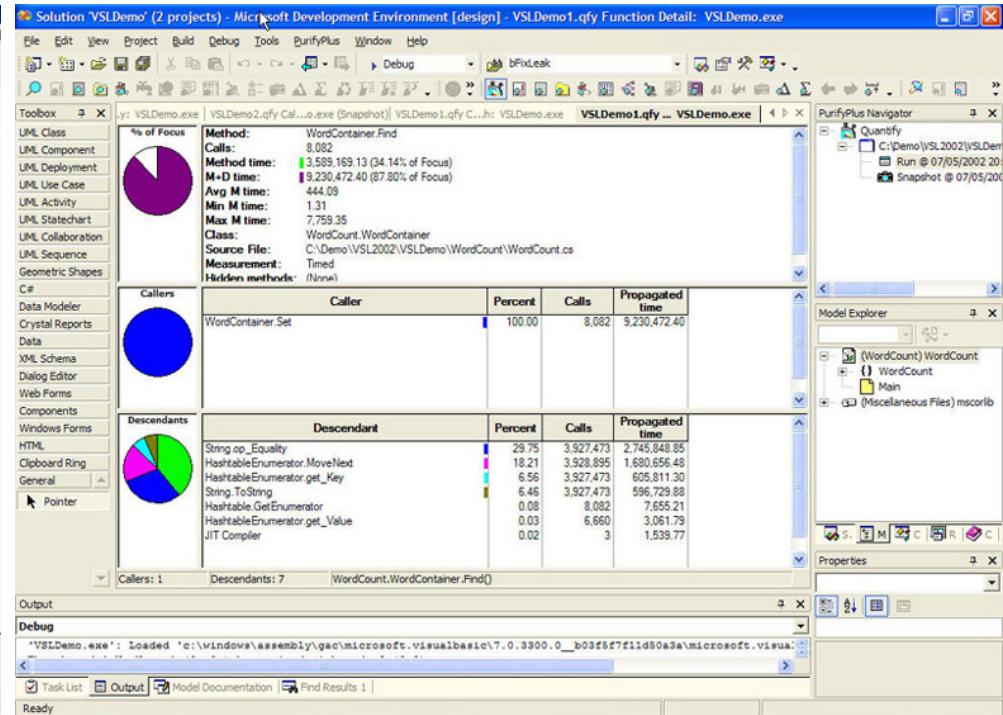
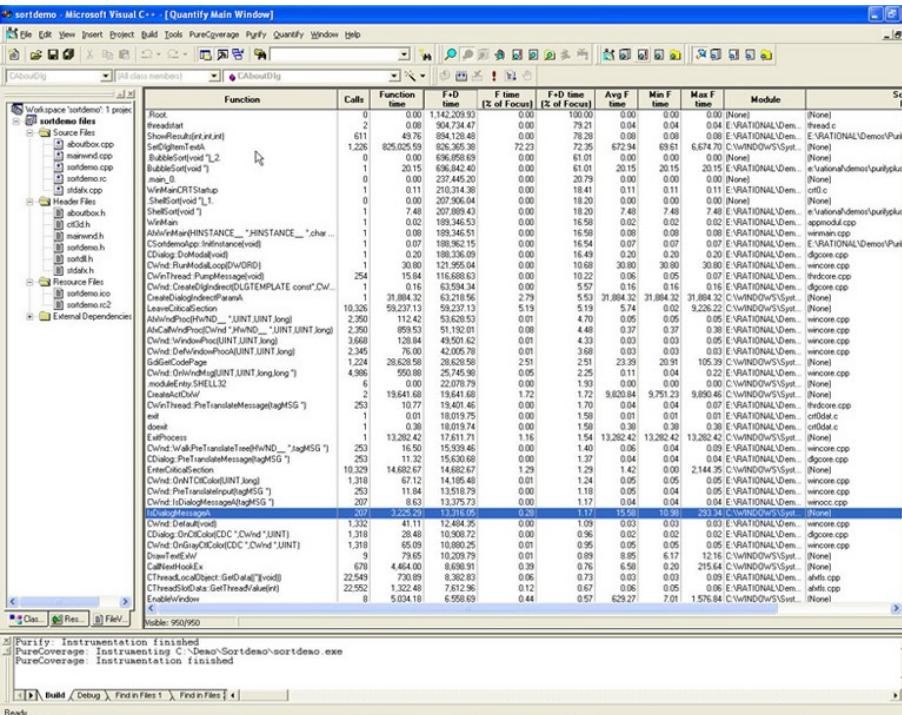
- Your program will spend the **majority** of its time in a **minority** of your source code
 - Focusing your optimisation on those
- Once you have done that it is rarely obvious where the time is spent

Time Profilers

- Where is your program spending its time?
- How do you find out?
- A **time profiler** will run your program and measure where the time is spent
- Three kinds
 - Instrumenting (e.g. Quantify)
 - Sampling (e.g. GPROF)
 - CPU performance counters (e.g. Intel PCM)

Quantify (Part of PurifyPlus)

- Instrumenting profiler
 - Counts function calls, and execution time, for each LoC
 - Draws nice graphs
 - GUI for exploring results



GPROF

- Sampling aided with instrumentation
 - Every few milliseconds the CPU is interrupted and the profiler captures the current PC location and call path
- Turned into reports at the end

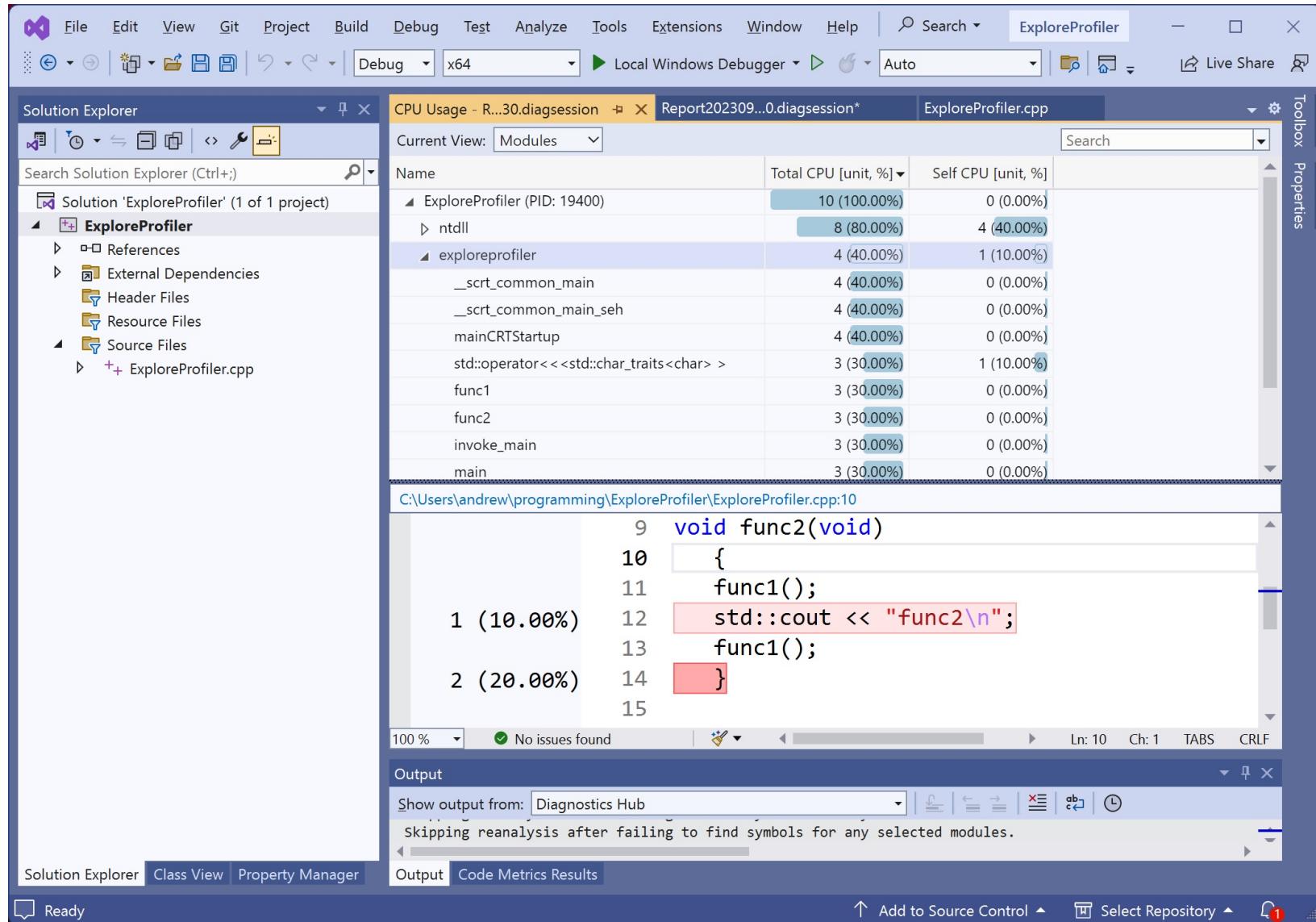
| Each sample counts as 0.01 seconds. | | | | | | | |
|-------------------------------------|------------|---------|-------|--------|--------|-------|-----------|
| % | cumulative | self | | self | | total | |
| time | seconds | seconds | calls | s/call | s/call | | name |
| 35.34 | 5.37 | 5.37 | 1 | 5.37 | 10.22 | | func1 |
| 33.03 | 10.39 | 5.02 | 1 | 5.02 | 5.02 | | func2 |
| 31.97 | 15.24 | 4.86 | 1 | 4.86 | 4.86 | | new_func1 |
| 0.20 | 15.27 | 0.03 | | | | | main |

GPROF

- Including call graph reports
 - And more

| Call graph (explanation follows) | | | | | |
|----------------------------------|--------|------|----------|--------|------------------------|
| index | % time | self | children | called | name |
| [1] | 100.0 | 0.03 | 15.24 | | <spontaneous> |
| | | 5.37 | 4.86 | 1/1 | main [1] |
| | | 5.02 | 0.00 | 1/1 | func1 [2] func2 [3] |
| ----- | | | | | |
| [2] | 67.0 | 5.37 | 4.86 | 1/1 | main [1] |
| | | 5.37 | 4.86 | 1 | func1 [2] |
| | | 4.86 | 0.00 | 1/1 | new_func1 [4] |
| ----- | | | | | |
| [3] | 32.9 | 5.02 | 0.00 | 1/1 | main [1] |
| | | 5.02 | 0.00 | 1 | func2 [3] |
| ----- | | | | | |
| [4] | 31.8 | 4.86 | 0.00 | 1/1 | func1 [2] |
| | | 4.86 | 0.00 | 1 | new_func1 [4] |
| ----- | | | | | |

Visual Studio



Intel PCM

- Performance Counter Monitor
 - Either text output or plug-in to OS tools

```

[CA] - Far 2.0.1420 x86 Administrator

IPC : instructions per CPU cycle <0..4 on Nehalem and Westmere>
BSEQ : relative to nominal CPU frequency/current CPU frequency/nominal frequency (includes Intel®
Turbo Boost Technology)
L3MISS: L3 cache misses
L2MISS: L2 cache misses (including other core's L2 cache *hits*)
L3HIT : L3 cache hit ratio <0.00-1.00>
L2HIT : L2 cache hit ratio <0.00-1.00>
L3CLM : ratio of CPU cycles lost due to L3 cache misses <0.00-1.00>, in some cases could be >1.0
due to a higher memory latency
L2CLM : ratio of CPU cycles lost due to missing L2 cache but still hitting L3 cache <0.00-1.00>
READ : bytes read from memory controller (in GBytes)
WRITE : bytes written to memory controller (in GBytes)

Core <SKT> : IPC : FREQ : L3MISS : L2MISS : L3HIT : L2HIT : L3CLM : L2CLM : READ : WRITE

  0   0   0.09  0.68   551 K  569 K  0.03  0.04  2.18  0.82  N/R  N/R
  1   0   0.09  0.68    6   463   0.96  0.43  0.99  0.99  N/R  N/R
  2   0   0.34  0.68   691   42 K  0.98  0.10  0.91  0.12  N/R  N/R
  3   0   0.01  0.68   10   6134   1.00  0.01  0.99  0.02  N/R  N/R
  4   0   0.15  0.68   97 K  120 K  0.19  0.04  0.84  0.04  N/R  N/R
  5   0   0.01  0.68   23   6435   1.00  0.03  0.99  0.01  N/R  N/R
  6   0   0.06  0.69   4056   37 K  0.87  0.00  0.95  0.07  N/R  N/R
  7   0   0.01  0.68   55   39 K  0.99  0.00  0.99  0.02  N/R  N/R
  8   0   0.19  0.68   89 K  104 K  0.24  0.01  0.93  0.11  N/R  N/R
  9   0   0.01  0.69   462   1725   0.74  0.00  0.90  0.00  N/R  N/R
 10  0   0.01  0.68   12   953   0.98  0.00  0.98  0.00  N/R  N/R
 11  0   0.29  0.68   12 K  64 K  0.81  0.07  0.15  0.14  N/R  N/R
 12  1   0.05  0.68   1812   27 K  0.93  0.01  0.91  0.04  N/R  N/R
 13  1   0.01  0.68   55   11 K  1.00  0.00  0.99  0.02  N/R  N/R
 14  1   0.55  0.69   4491   48 K  0.91  0.25  0.02  0.04  N/R  N/R
 15  1   0.01  0.68   46   15 K  1.00  0.00  0.99  0.01  N/R  N/R
 16  1   0.02  0.68   271   22 K  0.97  0.00  0.99  0.02  N/R  N/R
 17  1   0.01  0.68   32   23 K  1.00  0.00  0.99  0.02  N/R  N/R
 18  1   0.03  0.68   889   35 K  0.98  0.02  0.99  0.03  N/R  N/R
 19  1   0.02  0.68   392   25 K  0.98  0.01  0.99  0.02  N/R  N/R
 20  1   0.03  0.68   2899   42 K  0.93  0.00  0.91  0.03  N/R  N/R
 21  1   0.01  0.68   87   20 K  1.00  0.00  0.99  0.02  N/R  N/R
 22  1   0.12  0.68   12 K  99 K  0.88  0.13  0.04  0.07  N/R  N/R
 23  1   0.01  0.68   142   23 K  0.99  0.00  0.99  0.02  N/R  N/R

SKT  0   0.00  0.68   758 K  984 K  0.24  0.04  0.51  0.04  0.01  0.00
SKT  1   0.00  0.68   23 K  396 K  0.94  0.00  0.01  0.03  0.00  0.00

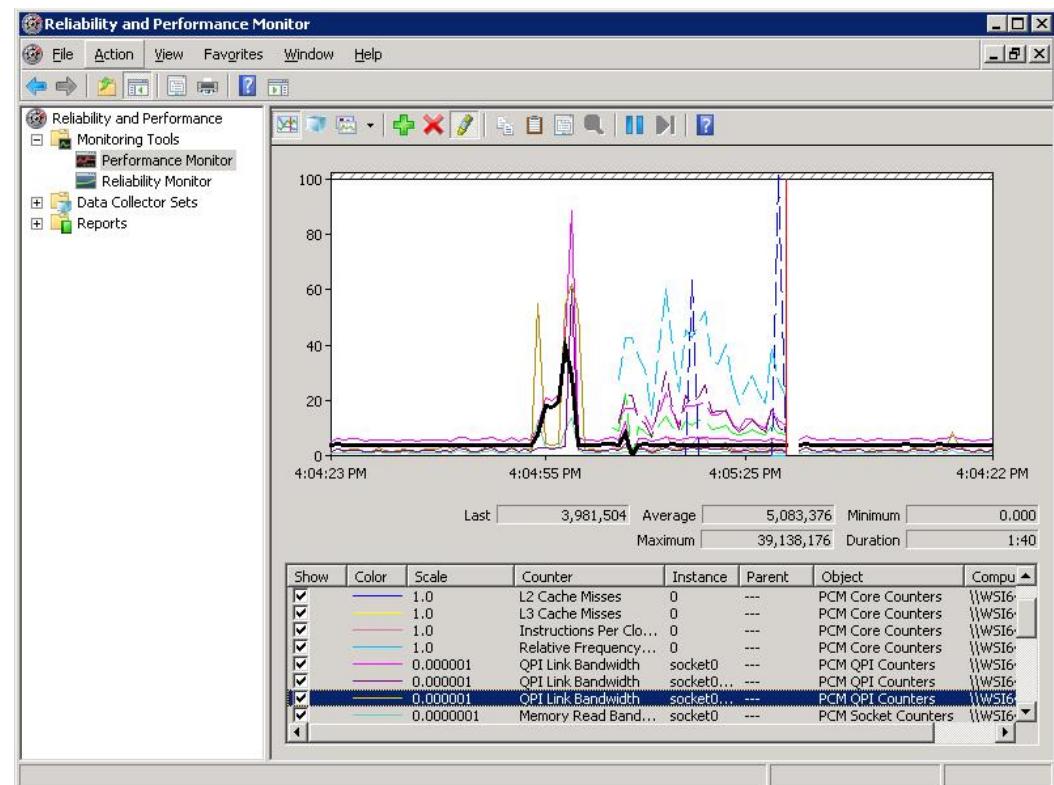
TOTAL *  0.00  0.68   773 K  1385 K  0.44  0.05  0.19  0.03  0.01  0.00

PHYSICAL CORE IPC: 0.16 => corresponds to 4.00 x core utilization

Intel® QPI traffic estimation in bytes (traffic coming to CPU/socket through QPI links):
          QPI0      QPI1
SKT  0   1997 K  1989 K
SKT  1   1760 K  13 K

Total QPI traffic: 5761 K   QPI traffic/Memory controller traffic: 0.34
C:\>
1 Left 2 Right 3 View... 4 Edit... 5 Print... 6 Minim... 7 Bind... 8 Integ... 9 Video... 10 Tree... 11 Viewits 12 Fields...

```



Other Profilers

- Cache Profilers
 - E.g. cachegrind (part of valgrind)
 - Does cache simulations, annotates code with cache misses
 - L1 instruction cache reads and misses
 - L1 data cache reads and read misses, writes and write misses
 - L2 unified cache reads and read misses, writes and writes misses.
- Memory Profilers
 - Tell you line-by-line how much memory is in use
 - E.g.: <https://pypi.org/project/memory-profiler/>
- Data-race checkers
 - Intel Inspector

Conclusions

- Optimisation is important for
 - User experience
 - Hardware cost (including running costs)
 - Green Computing
- 90/10 rule
- Use a profiler

COSC345

Software Engineering

Basic Computer Architecture

and

The Stack

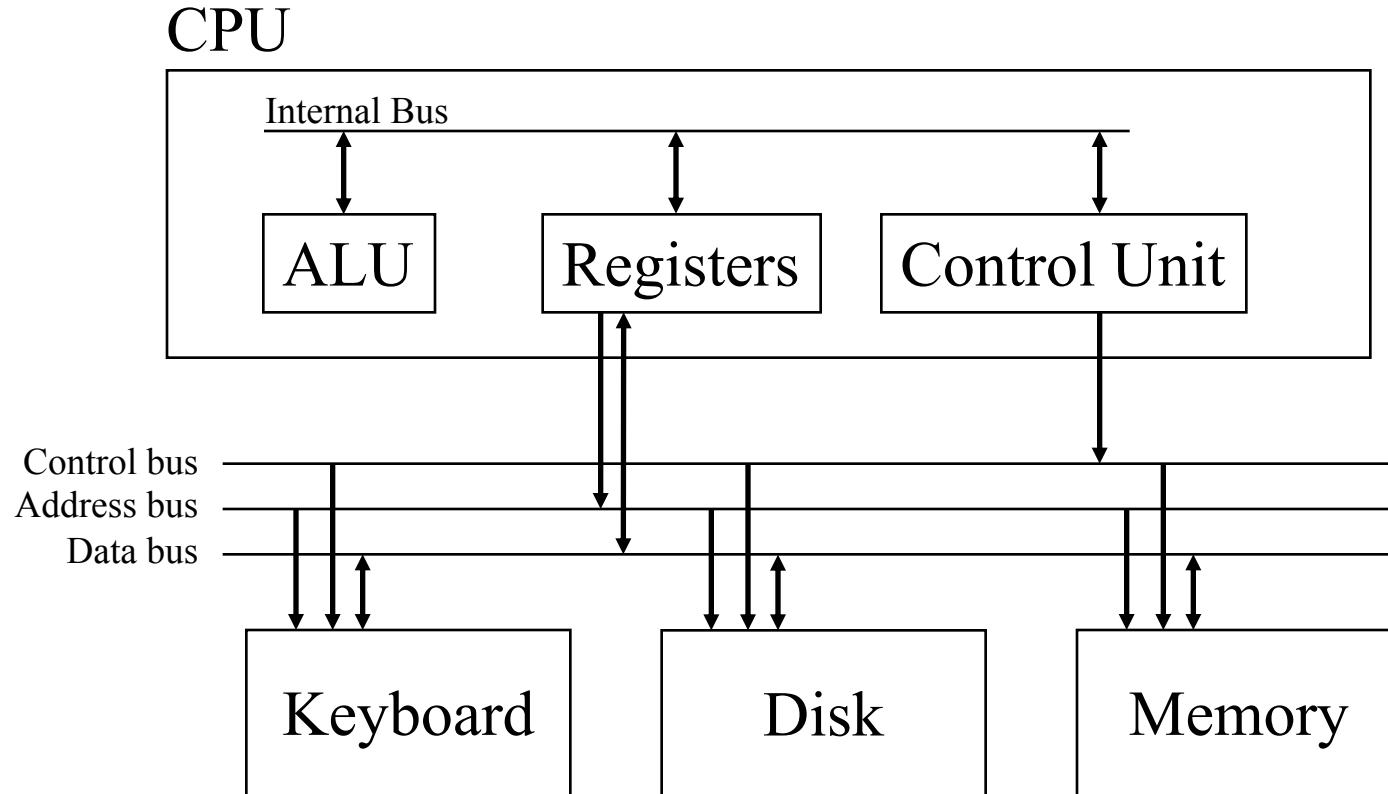
Outline

- A little about the 68HC11 CPU
 - Memory map
 - Registers
 - A little bit of assembly (never did us any harm)
- The stack
 - Procedure calls
 - Local variables
- For and while loops
- Stack problems

68HC11 CPU

- Motorola 68HC11
 - Very similar to the 6809 (used in COSC204)
 - Shares a common ancestor (the 6800)
- 16-bit von Neumann architecture
 - Shared data and program space
 - Contents addressable by location
 - Execution occurs in a sequential fashion
 - Unless told to do otherwise
- Full 64K of address space
 - All address locations exist
 - Input / Output devices

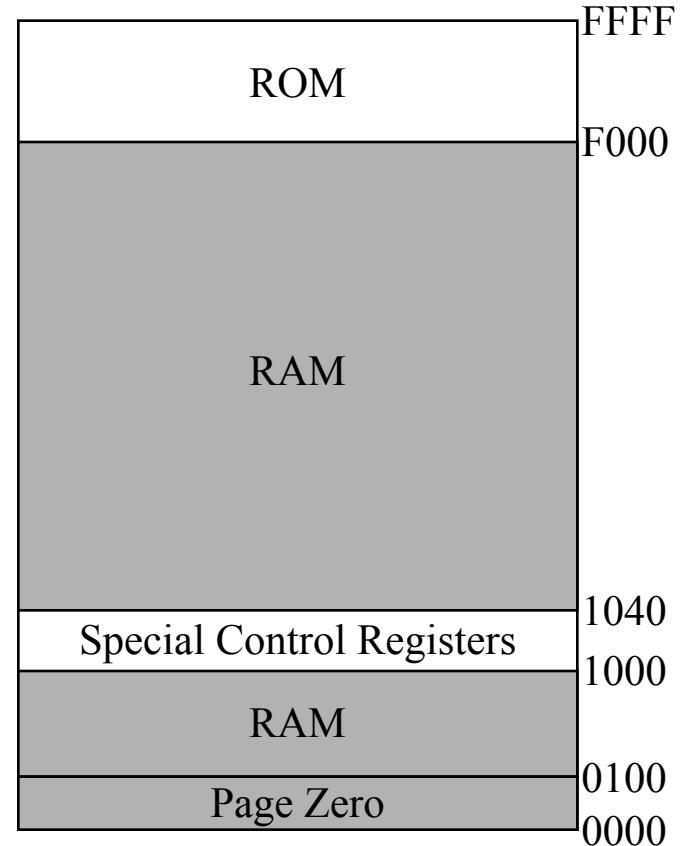
Architecture (Revision)



68HC11 Memory Map

| <i>Start</i> | <i>End</i> | <i>Function</i> |
|--------------|------------|---------------------------|
| 0000 | 00FF | Page Zero RAM (fast) |
| 0100 | 0FFF | RAM |
| 1000 | 103F | Special Control Registers |
| 1040 | EFFF | RAM |
| F000 | FFFF | ROM |

- Special Control registers (serial ports, etc.) are movable and the ROM / RAM boundary is implementation dependent



Instruction Syntax (Revision)

- Syntax

<label> <opcode> <parameters>

- Parameters

immediate

\$ hex value

% binary

@ octal

‘C’ ASCII character

- Example

DOTHIS

LDAA #\$44

LDAB #%01000100

Instruction Types (Revision)

- Instruction types
 - Arithmetic
 - ABA
 - Logic
 - ANDA #\$F6
 - Load and Store
 - LDAA \$EC0B
 - Transfer control
 - JSR \$6000
 - Special
 - WAI

Addressing Modes (Revision)

- Inherent
 - No arguments needed
 - ABA
- Immediate
 - Value specified immediately after opcode
 - LDAA #\$08
- Direct
 - Supplied memory location on “zero” (demand) page
 - LDAA \$59
- Extended
 - Supplied memory location
 - LDAA \$0600

Addressing Modes (Revision)

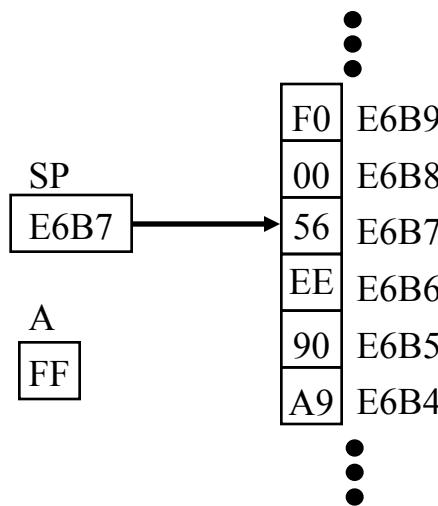
- Indexed (INDX and INXY)
 - Relative to the X or Y registers
 - LDAA \$05,X
- Relative
 - Used in branch instructions
 - BNE \$06
- Mixed mode
 - BRCLR 03,X #\$A8 \$FF00
 - If $(!(X+3) \& 0xA8)$ then PC=\$FF00

68HC11 Registers

- 68HC11 is typical and has few
 - 8 bit
 - A, B
 - Flags (Condition Codes)
 - 16 bit
 - D (A (high byte) and B (low byte) combined)
 - X, Y (index registers)
 - SP (the Stack Pointer)
 - PC (the Program Counter)

The Stack

- Just a location (high) in memory
- Special assembly instructions manipulate it
 - PSHA, PULB, LDS, INS, DES, TSX, etc.
- 68HC11
 - SP points to the “top” of the stack
 - PSHA: store A where SP points then $SP=SP-1$
 - PULA: $SP=SP+1$, load A with where SP points



- What is the effect of pushing A?
- What is the effect of pulling A?
- What happens when the stack gets “big”?

Subroutines

- Sample 68HC11 code

MAIN:

JSR HERE ; 6 cycles

RTS

HERE:

RTS ; 5 cycles

| Cycle | RTS (INH) | | |
|-------|-----------|--------|-----|
| | Addr | Data | R/W |
| 1 | OP | 39 | 1 |
| 2 | OP + 1 | - | 1 |
| 3 | SP | - | 1 |
| 4 | SP + 1 | Rtn hi | 1 |
| 5 | SP + 2 | Rtn lo | 1 |

| Cycle | JSR (DIR) | | | JSR (EXT) | | | JSR (IND,X) | | | JSR (IND,Y) | | |
|-------|-----------|--------|-----|-----------|--------|-----|-------------|----------|-----|-------------|----------|-----|
| | Addr | Data | R/W | Addr | Data | R/W | Addr | Data | R/W | Addr | Data | R/W |
| 1 | OP | 9D | 1 | OP | BD | 1 | OP | AD | 1 | OP | 18 | 1 |
| 2 | OP + 1 | dd | 1 | OP + 1 | hh | 1 | OP + 1 | ff | 1 | OP + 1 | AD | 1 |
| 3 | 00dd | (00dd) | 1 | OP + 2 | ll | 1 | FFFF | - | 1 | OP + 2 | ff | 1 |
| 4 | SP | Rtn lo | 0 | hhll | (hhll) | 1 | X + ff | (X + ff) | 1 | FFFF | - | 1 |
| 5 | SP - 1 | Rtn hi | 0 | SP | Rtn lo | 0 | SP | Rtn lo | 0 | Y + ff | (Y + ff) | 1 |
| 6 | | | | SP - 1 | Rtn hi | 0 | SP - 1 | Rtn hi | 0 | SP | Rtn lo | 0 |
| 7 | | | | | | | | | | SP - 1 | Rtn hi | 0 |

High Level Language Calls

- Is this procedure call free?

```
int getval(int p1, int p2, int p3, int p4)
{
    return p2;
}
```

```
int main(int argc, char *argv[])
{
    return getval(6, 7, 8, 9);
}
```

- How long does it take?

Procedure Calls

- C version

```
int getval(void)
{
    return 0;
}
```

```
int main(int argc, char *argv[])
{
    return getval();
}
```

- Hand assembly version

```
_GETVAL:
    LDD #0
    RTS

_MAIN:
    JSR _GETVAL
    RTS
```

- Compiler assembly version

```
_GETVAL:
    LDD #0
    RTS

_MAIN:
    TSX
    JSR _GETVAL
    TSX
    RTS
```

Calling Conventions

- The C calling convention (on 68HC11)
 - Place parameters onto the stack (reverse order)
 - Jump to the subroutine
 - Fix the stack
- The C returning convention (on 68HC11)
 - Load result into register D
 - Return

High Level Procedure Calls

- C Version

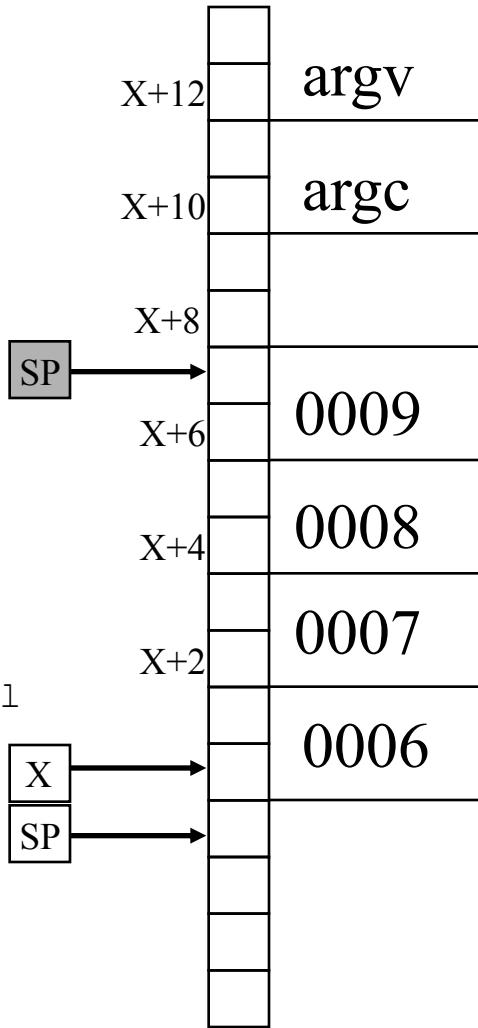
```
int getval(int p1, int p2, int p3, int p4)
{
    return p2;
}

int main(int argc, char *argv[])
{
    return getval(6, 7, 8, 9);
}
```

Assembly Version

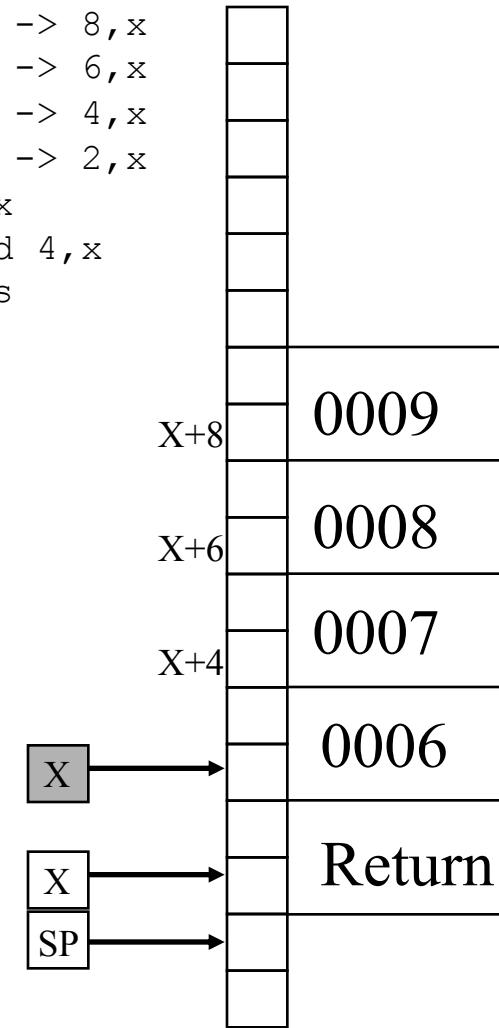
_main:

```
;     argv -> 12,x
;     argc -> 10,x
pshx
pshx
pshx
pshx
tsx
ldd #6
std 0,x
ldd #7
std 2,x
ldd #8
std 4,x
ldd #9
std 6,x
jsr _getval
tsx
pulx
pulx
pulx
pulx
rts
```



_getval:

```
;     p4 -> 8,x
;     p3 -> 6,x
;     p2 -> 4,x
;     p1 -> 2,x
tsx
ldd 4,x
rts
```



Local Variables

- Local variables go on the stack

- We have already seen this

```
int main(int argc, char *argv[])
{
    char byte;
    int integer;

    byte = 1;
    integer = 2;

    return byte + integer;
}
```

- What happens if there are many?
- What if they are large?
- What is at 4,x?
- How big is char byte?

```
_main:
;   integer -> 0,x
;   byte -> 2,x
;   argc -> 6,x
;   argv -> 8,x
        pshx
        pshx
        tsx
        ldab #1
        stab 2,x
        ldd #2
        std 0,x
        ldab 2,x
        clra
        tstb
        bpl x0.var
        coma
x0.var:
        addd 0,x
        pulx
        pulx
        rts
```

Loops

- for loops are really while loops

```
for (count = 0; count < 0xFF; count++)  
    x++;
```

- Is equivalent to

```
count = 0;  
while (count < 0xFF)  
{  
    x++;  
    count++;  
}
```

- How might this behave when “stepping” in an interactive debugger?

```
;           x -> 0,x  
;           count -> 2,x  
;           count = 0  
           ldd #0  
           std 2,x  
;           while()  
           jmp L5.loop  
  
L2.loop:  
;           x++  
           ldd 0,x  
           addd #1  
           std 0,x  
L3.loop:  
;           count++  
           ldd 2,x  
           addd #1  
           std 2,x  
L5.loop:  
;           while (count < 0xFF)  
           ldd 2,x  
           cpd 0,x  
           blt L2.loop
```

Loops and Variables

- What is the output of this program?

```
int main(int argc, char *argv[])
{
    int pos, y[1], result;

    result = 2;
    for (pos = 0; pos <= 1; pos++)
        y[pos] = 100;

    printf("pos:%d result:%d\n", pos, result);
}
```

Simplified Version

- C Version

```
int main(int argc, char *argv[])
{
int pos, y[1], result;

pos = 0;
result = 2;
y[1] = 100;

printf("pos:%d result:%d\n", pos, result);
}
```

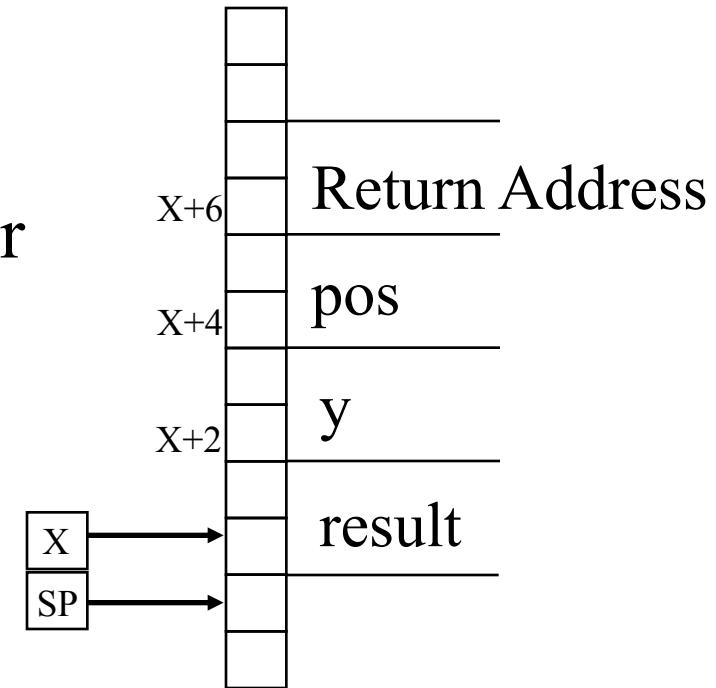
- Assembly Version

```
;      result -> 0,x
;          y -> 2,x
;      pos -> 4,x

; pos = 0
        ldd #0
        std 4,x
; result = 2
        ldd #2
        std 0,x
; y[1] = 100
        ldd #100
        std 4,x
```

Buffer-Overrun Attack

- Y is just a pointer into the stack
- Indices are just arithmetic expressions
- $y[n] = ((\text{char } *)y) + (n * \text{sizeof}(*y))$
- What is the result of $y[2]=100$?
- How is this exploited in a buffer overrun attack on a program?



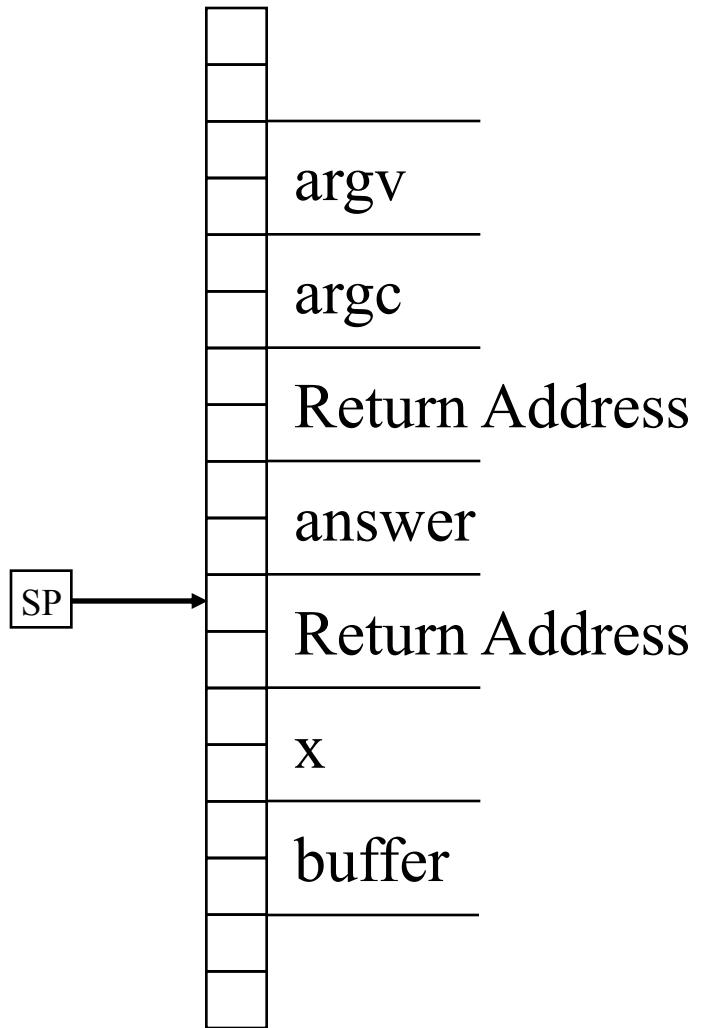
Out of Scope (Dangling Pointers)

- What happens to the stack here?

```
char *doit(void)
{
    char *x, buffer[2];

    x = buffer;
    return x;
}

int main(int argc, char *argv[])
{
    char *answer = doit();
    answer[2] = 'G';
    return 0;
}
```



Stack Problems

- What happens if the stack gets too large?
- How does this exhibit itself in your program?
- What happens if you overwrite the stack?
- How does this exhibit itself in your program?
- What happens when a pointer goes out of scope?
- How does this exhibit itself in your program?
- What happens when there are multiple threads?

Behaviors To Look For

- Crash when a function returns
 - The return address on the stack might be corrupt
- A passed parameter is different from that passed
 - The parameter might have been overwritten
- Variables changing values without reason
 - A dangling pointer might be altering it
- Crash in (otherwise reliable) system routine
 - Stack overflow might be occurring
- Memory allocation has side-effects
 - Corruption in heap can trickle down to the stack
- Bugs that go away in the debugger
 - The stack will (probably) be different

Conclusion

- Stack overflow
 - Is the compiler always right?
 - The architecture can introduce errors
 - Some errors aren't algorithm errors
 - Non-algorithm errors are hard to identify
 - Don't always assume your algorithm is wrong!
- Stack corruption can easily occur
 - Look out for this!
- Stack problems aren't always easy to find

COSC345

Software Engineering

The Heap
And
Dynamic Memory Allocation

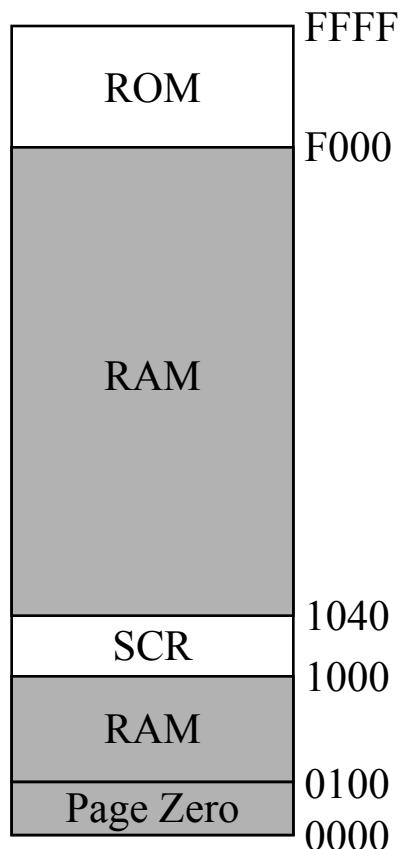
Outline

- Revision
- The programmer's view of memory
- Simple array-based memory allocation
- C memory allocation routines
- Virtual memory
- Swapping

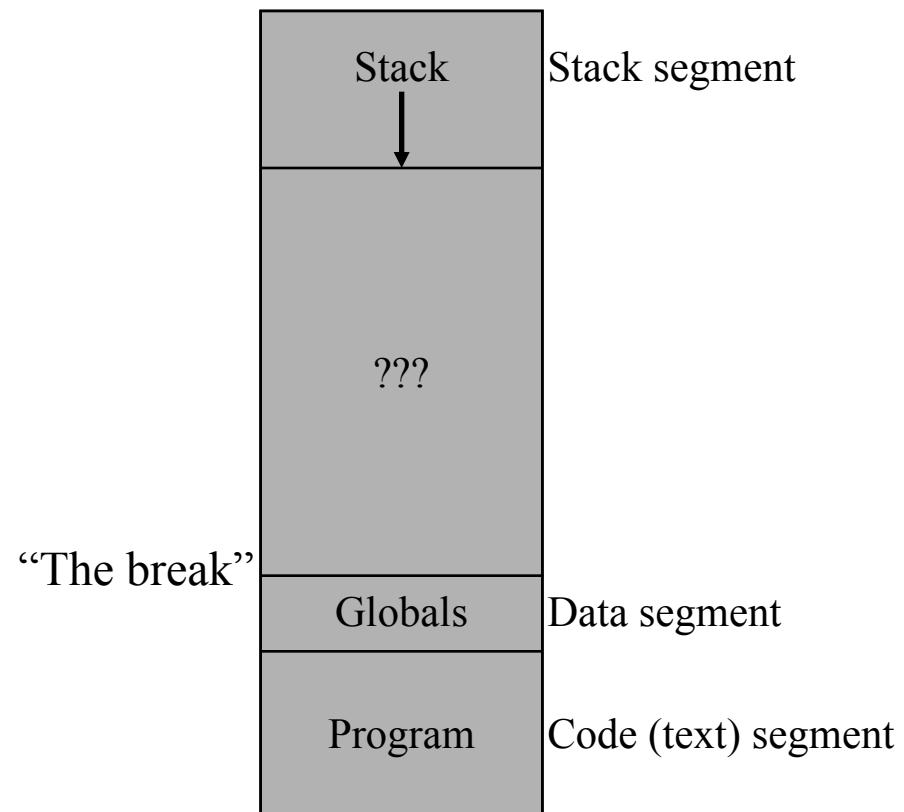
Revision

- From previous lectures
 - Using the von Neumann architecture
 - Linear address space
 - Every memory location exists
 - Stack is at top of memory
 - Program is at bottom of memory
- Also assume
 - At compile time
 - Program size is known
 - The amount of space needed for global variables is known
- So
 - The remaining unused memory is available

Computer Memory



Hardware Model
68HC11



Software Model

Program Segments

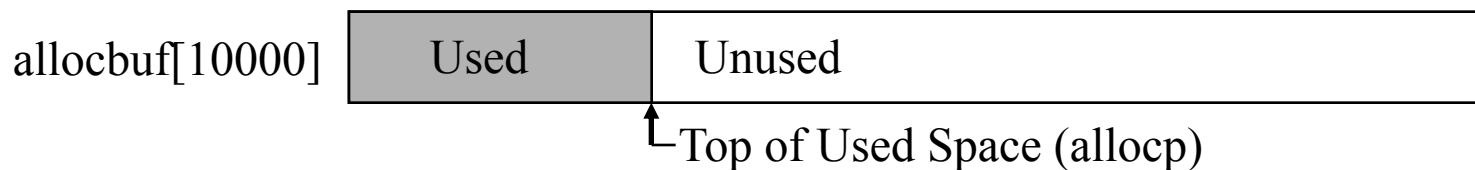
- Program
 - Low in memory (the code or text segment)
- Local Variables
 - On the stack (the stack segment)
- Global variables
 - Above the program and before the stack (the data segment)
- The Heap
 - Everything else (above the break & below the stack)

The Problems

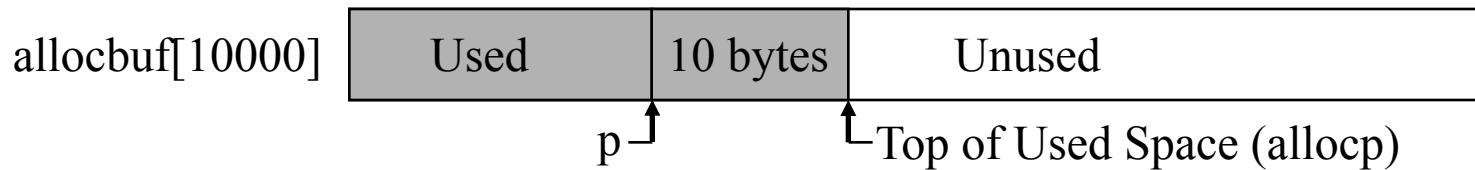
- Don't know how big a given structure will be until the program is running
- Don't know how much memory is needed before the program starts
- Not enough physical memory so must re-use it
- We need some kind of re-usable space
 - Memory allocation and deallocation

Array-Based Stacking Allocation

- Hand out pieces of a large character array



- $p = \text{alloc}(10)$



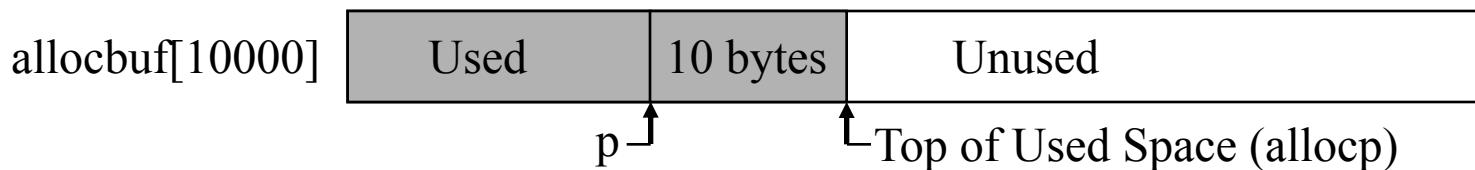
Array-Based Stacking Allocation

```
#define ALLOCSIZE 10000           /* size of available space */
static char allocbuf[ALLOCSIZE];  /* storage for alloc */
static char *allocp = allocbuf;   /* next free position */

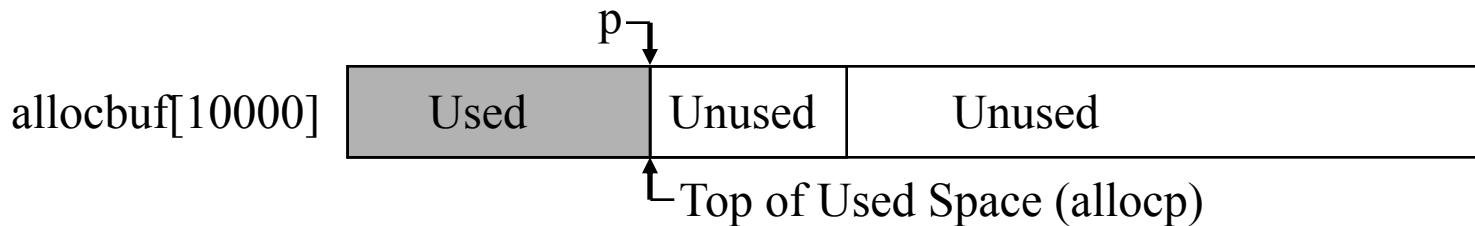
char *alloc(int n)              /* return pointer to n characters */
{
    if (allocbuf + ALLOCSIZE - allocp >= n)
        {                         /* it fits */
            allocp += n;
            return allocp - n;      /* old p */
        }
    else                          /* not enough room */
        return NULL;
}
```

Array-Based Stacking Free

- $p = \text{alloc}(10)$



- $\text{afree}(p)$



Array-Based Stacking Free

```
void afree(char *p) /* free storage pointed to by p */
{
if (p >= allocbuf && p < allocbuf + ALLOCSIZE)
    allocp = p;
}
```

Example

- What is the output of this program?

```
int main(int argc, char *argv[])
{
char *p, *q;

p = alloc(1);
q = alloc(2);
q[0] = 'q';
p[0] = 'p';
p[1] = '\0';
printf("%c %c\n", p[0], q[0]);
}
```

Example

- What is the output of this program?

```
int one_main(int argc, char *argv[])
{
char *p, *q, *r;

p = alloc(1);
p[0] = 'p';

q = alloc(2);
q[0] = 'q';
q[1] = 'q';

afree(p);

r = alloc(3);
r[0] = 'r';
r[1] = 'r';
r[2] = 'r';

printf("%c %c\n", q[1], r[2]);
}
```

Example

- What is the output of this program?

```
int main(int argc, char *argv[])
{
char *p;
int *q;

//p = alloc(1);
q = (int *)alloc(sizeof(int));
*q = 123456;
printf("%d\n", *q);
}
```

- What if the comment is removed (i.e. that code runs)?
- Is the output *always* the same?

Problems

- Must free in reverse order to allocation
- Allocated memory is not aligned correctly
- Memory overwrites cause data corruption
- Heap overflow causes stack corruption
 - And stack overflow causes heap corruption

Allocate and Free

- This array-based allocator is good when we
 - Want to allocate many small units and free all at once
 - Never need to free the memory
 - The OS does it for you when your program terminates
- But what about type-safe languages?
 - In some you can allocate an array of objects and return references to those objects, then re-use them later
 - You end up with separate zones for each object type
 - You pay the price of (i.e. time for) allocation once

SWIFT

```
class thing
{
    var x : Int = 0;
}

class allocator
{
    let allocbuf = Array<thing>(unsafeUninitializedCapacity: 10000)
    {
        buffer, initializedCount in

        for element in 0 ..< 10000
        {
            buffer[element] = thing()
        }
    }

    initializedCount = 10000
}

var allocp : Int = -1
}
```

SWIFT

```
extension allocator

{
    func alloc() -> thing
    {
        allocp += 1
        return allocbuf[allocp]
    }

    func tell() -> Int
    {
        return allocp
    }

    func free(_ to: Int)
    {
        allocp = to
    }
}
```

SWIFT

```
var thing_allocator = allocator()
```

```
var p = thing_allocator.alloc()
```

```
var q = thing_allocator.alloc()
```

```
p.x = 1
```

```
q.x = 3
```

```
thing_allocator.free(-1)
```

```
var r = thing_allocator.alloc()
```

```
r.x = 4
```

```
print(p.x, q.x, r.x)
```

Allocate and Free

- General purpose allocation needs
 - A list of unused (free) memory
 - Details of the allocation size of a piece of memory
 - To correctly align allocated memory

The Header

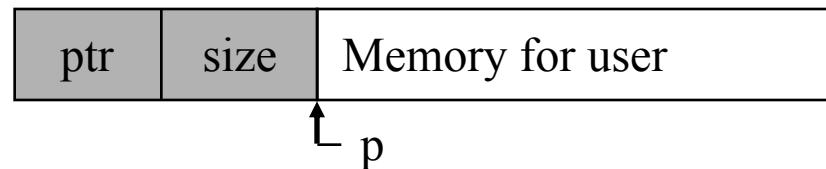
- To keep a free list we need to know how big the allocated unit was

```
struct header
{
    struct header *ptr;          /* next block if on free list */
    unsigned size;               /* size of this block */
};
```

- Allocate enough for user and the header

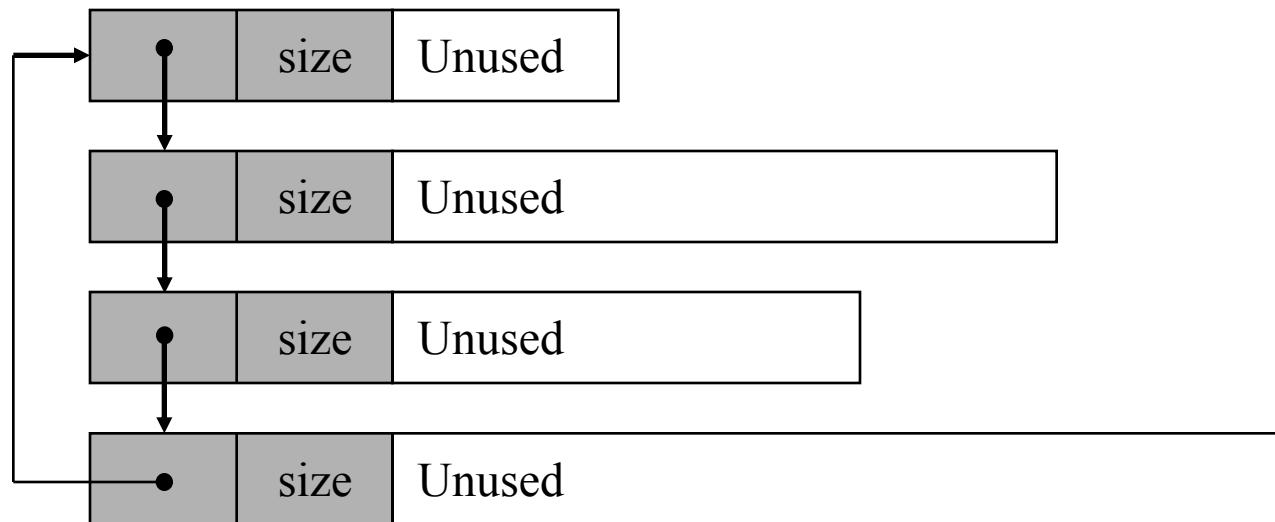
```
needed = sizeof(struct header) + nbytes;
```

- When allocating return a “fake” to the user



The Free List

- Chain the blocks together when free is called
 - Chunks kept in order by increasing addresses
 - Merge adjacent blocks into one larger block



- There are alternative solutions
 - Buckets

Possible Allocation Schemes

- Best Fit
 - Scan the entire list
 - Use the smallest block large enough
 - Leaves large free areas untouched
 - Tends to leave small holes in the address space (fragmentation)
 - Slow
- First Fit / Next Fit
 - Use the first block in the list
 - Fragmentation no worse than best fit
 - Fast
- Other schemes
 - Worst fit, etc.

Word Alignment

- Some hardware and OSs force “word” alignment
 - Force the header to be aligned on worst case boundary

```
typedef long Align;           /* for alignment to long boundary */

union header
{   /* block header */
    struct
    {
        union header *ptr; /* next block if on free list */
        unsigned size; /* size of this block */
    } s;
    Align x; /* force alignment of blocks */
};

typedef union header Header;
```

Allocate Chunks

- Make sure all allocation is of *Header* sized chunks

```
nunits = (nbytes + sizeof(Header) - 1) / sizeof(header);
```

- And make room for the header itself

```
nunits = (nbytes + sizeof(Header) - 1) / sizeof(header) + 1;
```

- Now everything will be aligned in units of Header
- A free list is needed

```
static Header *freep = NULL; /* start of free list */
```

- Use a sentinel

```
static Header base; /* empty list to get started */
```

Malloc()

```
void *malloc(unsigned nbytes) /* malloc: general-purpose storage allocator */
{
Header *p, *prevp;
unsigned nunits;

nunits = (nbytes + sizeof(Header) - 1) / sizeof(header) + 1;
if ((prevp = freep) == NULL)
{
    base.s.ptr = freeptr = prevptr = &base;
    base.s.size = 0;
}
for (p = prevp->s.ptr; ; prevp = p, p = p->s.ptr)
{
    if (p->s.size >= nunits)
        {
            if (p->s.size == nunits)
                /* big enough */
                /* exactly */
                prevp->s.ptr = p->s.ptr;
            else
                {
                    p->s.size -= nunits;
                    p += p->s.size;
                    p->s.size = nunits;
                }
            freep = prevp;
            return (void *) (p + 1);
        }
    if (p == freep)
        if ((p = morecore(nunits)) == NULL)
            return NULL;
}
}
```

Morecore()

- Gets more memory from OS – calls sbrk()
 - Equivalent to a call to alloc() (from earlier)

```
#define NALLOC 1024 /* minimum #units to request */
static Header *morecore(unsigned nu) /* morecore: ask system for more memory */
{
    char *cp;
    Header *up;

    if (nu < NALLOC)
        nu = NALLOC;
    cp = sbrk(nu * sizeof(Header));           /* add to the break */
    if (cp == (char *) -1)                    /* no space at all */
        return NULL;

    up = (Header *) cp;
    up->s.size = nu;

    free((void *) (up+1));

    return freep;
}
```

Free()

```
void free(void *ap) /* free: put block ap in free list */
{
Header *bp, *p;

bp = (Header *)ap - 1;                                /* point to block header */
for (p = freep; !(bp > p && bp < p->s.ptr); p = p->s.ptr)
    if (p >= p->s.ptr && (bp > p || bp < p->s.ptr))
        break;                                         /* freed block at start or end of arena */

if (bp + bp->size == p->s.ptr)
{
    bp->s.size += p->s.ptr->s.size;                /* join to upper nbr */
    bp->s.ptr = p->s.ptr->s.ptr;
}
else
    bp->s.ptr = p->s.ptr;

if (p + p->size == bp)                               /* join to lower nbr */
{
    p->s.size += bp->s.size;
    p->s.ptr = bp->s.ptr;
}
else
    p->s.ptr = bp;
freep = p;
}
```

Problems

- Can overwrite the header (underrun)
 - Corrupts the header (size becomes corrupt)
- Can write past the end of a block of memory (overrun)
 - Will write into what-ever is there
- Can overwrite the free list (heap corruption)
 - The list of free memory chunks becomes corrupt
- Can use pointers to free memory (dangling pointer)
 - Can write to a piece of memory that has been reused
- Can forget to free memory (memory leak)
 - Eventually memory will all be used and heap enters stack
 - Our malloc() leaks!

More Problems

- Can free the same block twice (double free)
- Can fragment the heap (heap fragmentation)
 - Malloc() fails even though there's loads of memory
- Can run out of memory (out of memory)
 - Often caused by a leak

Other Allocation Routines

- `void *calloc(size_t count, size_t size)`
 - Call `malloc()` then set memory to all zero

```
return memset(malloc(count * size), 0, count * size);
```
- `void *realloc(void *ptr, size_t size)`
 - Resize a block
 - If smaller then

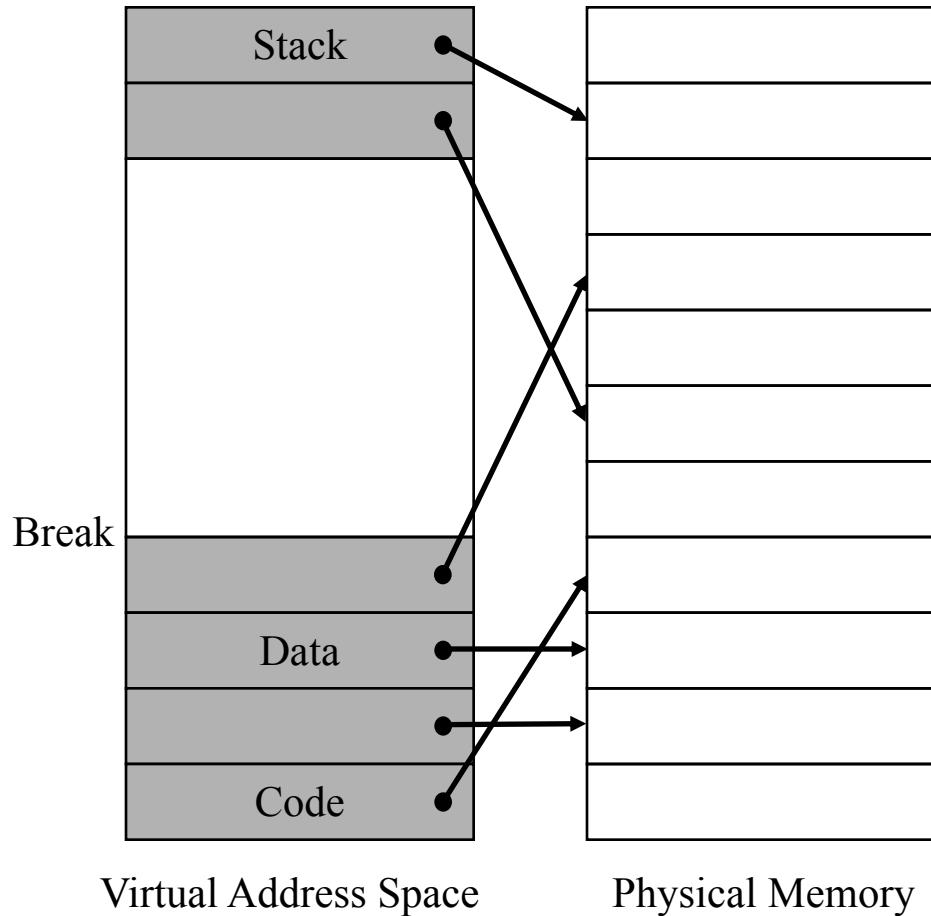
```
return ptr;
```
 - If larger then

```
ans = memcpy(malloc(size), size, ptr);
free(ptr);
return ans;
```
- `char *strdup(const char *str)`
 - Allocate space for a string and copy it there

```
return strcpy((char *)malloc(strlen(str) + 1), str);
```

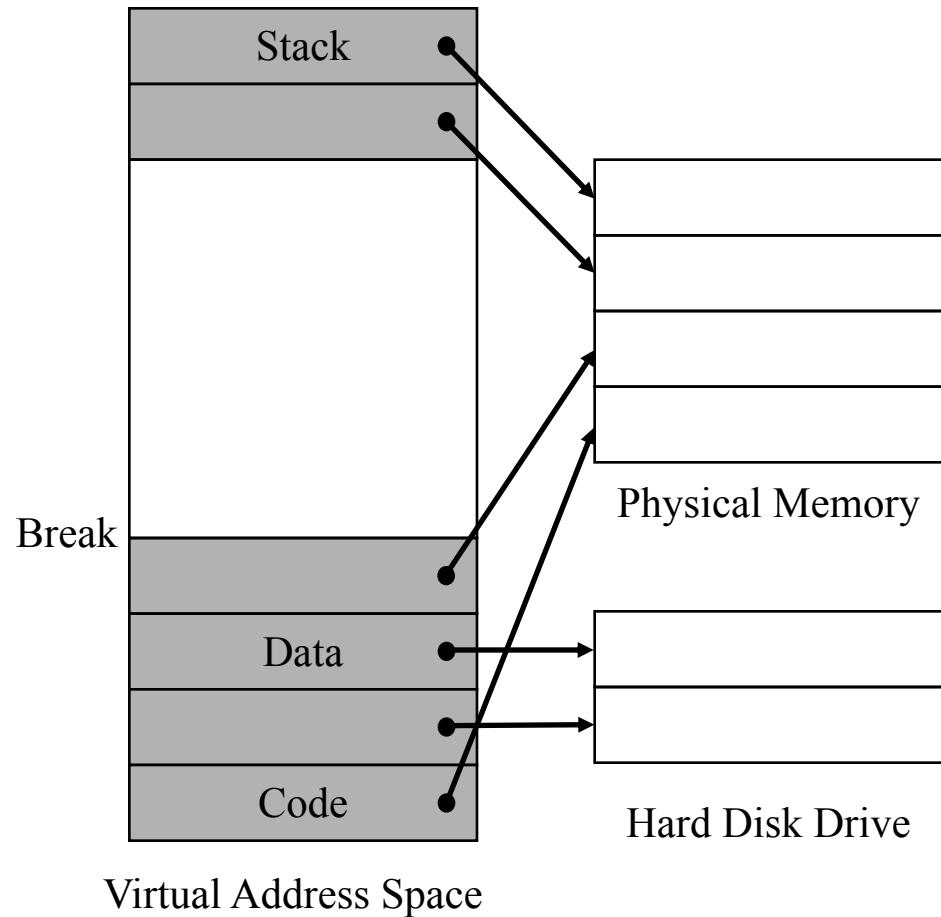
Virtual Memory (with MMU)

- Address space divided into pages
- The MMU maps from virtual to physical addresses
- Non-existing address access causes interrupt (page fault)
- Interrupt service routine called
 - Bus error?
- Extend virtual address space
 - Add physical pages to virtual address space
 - UNIX: brk()



Demand Page Swapping

- When there isn't enough physical memory the disk is used to hold pages
- When a page fault occurs the OS “swaps” a page in memory for one on disk
- OS is constantly trying to free up rarely-used physical memory in favor of often used virtual pages



Swapping

- Memory access time: 100ns
- Solid State Disk access time : $10\mu\text{s} - 100\mu\text{s}$
 - Assume SSD time at $50\mu\text{s}$
- Rotational Disk I/O time: $1\text{ms} - 10\text{ms}$
 - Assume HDD time at 5ms
- Swapping to SSD takes 500 times longer than
- Swapping to HDD takes 50,000 times longer
- Perhaps we should write programs that don't swap!
- B. Gregg (2013), Systems Performance: Enterprise and the Cloud

References

- B. Kernighan and D. Ritchie, *The C Programming Language*, Chapter 5 & 8
- D. Knuth, *The art of computer programming Volume 1 / Fundamental Algorithms* (2nd ed.), Section 2.5
- B. Gregg (2013), *Systems Performance: Enterprise and the Cloud*

Some code taken from: B. Kernighan and D. Ritchie, *The C Programming Language*

COSC345

Software Engineering

Managing Memory Managers

Outline

- Typical problems (from previous lectures)
- Tracking malloc() calls
- Catching calls to zero length malloc()
- Eradicating un-initialized memory
- Bulletproofing
- Catching leaks
- Catching buffer over-runs and under-runs
- Assumption for this lecture: we do *not* have access to the internals of the memory management routines: malloc(), free(), realloc(), etc.

Typical Problems

- Allocate zero length blocks
- Allocate a block and use its un-initialized contents
- Free a block then use it
- Call realloc then use the old pointer
- Allocate a block then lose the pointer to it
 - Memory leak
- Read and write beyond the boundaries of a block
 - Over-run and under-run
- Fail to notice any of these

malloc() Tracking

- Put a wrapper around malloc()
- my_malloc.h

```
#ifdef MALLOC_DEBUG
#define my_malloc(s) do_my_malloc(__LINE__, __FILE__, s)
void *do_my_malloc(long line, char *file, size_t size);
#else
#define my_malloc(s) malloc(s)
#endif
```

- my_malloc.c

```
#include <stdio.h>
#include <stdlib.h>

void *do_my_malloc(long line, char *file, size_t size)
{
    printf("%s(%d): log, malloc(%d)\n", file, line, size);
    return malloc(size);
}
```

malloc() Tracking

- main.c

```
#include <stdlib.h>
#define MALLOC_DEBUG
#include "my_malloc.h"

int main(int argc, char *argv[])
{
    my_malloc(100);
    return 0;
}
```

- Output

```
main.c(6): log, malloc(100)
```

- Why use #ifdef MALLOC_DEBUG ?

Zero Length Blocks

- Strictly forbidden by the ANSI standard
- Catch all cases and return NULL
- my_malloc.c

```
void *do_my_malloc(long line, char *file, size_t size)
{
if (size == 0)
{
    printf("%s(%d): error, zero length malloc\n", file, line);
    return NULL;
}
#ifndef MALLOC_DEBUG_ALL
else
    printf("%s(%d): log, malloc(%d)\n", file, line, size);
#endif

return malloc(size);
}
```

Uninitialized Memory

- Leads to unpredictable behavior – so initialize it!
- my_malloc.c

```
#define BAD_MEM 0xCC
void *do_my_malloc(long line, char *file, size_t size)
{
char *mem;

if (size == 0)
{
    printf("%s(%d): error, zero length malloc\n", file, line);
    return NULL;
}
#ifndef MALLOC_DEBUG_ALL
else
    printf("%s(%d): log, malloc(%d)\n", file, line, size);
#endif

mem = malloc(size);
if (mem == NULL)
{
    printf("%s(%d): error, malloc failure (NULL returned)\n", file, line);
    return NULL;
}
return memset(mem, BAD_MEM, size);
}
```

BAD_{_}MEM:0xCC or 0xA3

- S. Maguire, *Writing Solid Code*
 - Must look like garbage but not be garbage
 - 0x00, 0xFF, 0x01 all look like valid values
 - Maguire recommends 0xCC on PC 0xA3 on Macintosh
 - If word-alignment is enforced then make it odd

```
p = malloc(sizeof(int)); **p = 5; /* will crash */
```
 - As an index into an array it is large (and noticeable)

```
p = malloc(sizeof(int)); a[*p]=5; /* p is very large */
```
 - If ever called it will crash

```
p = malloc(sizeof(int)); p(); /* will crash */
```
 - Easy to spot long sequences in a debugger
 - Easy to spot the value in a printf()
- All done to make it crash at the first opportunity
 - Increase instability when the program goes wrong
- Many use DEADBEEF (see:<https://en.wikipedia.org/wiki/Hexspeak>)

Bulletproofing

- Bulletproofing
 - Making a program (module / class / method) robust to bad, incorrect, or unexpected input
- Bulletproofing malloc()
 - Forbid zero-length calls
 - Forbid the return of un-initialised memory
- Why should we bulletproof?
- Why should we not bulletproof?
- How else can we bulletproof malloc() and free()?

Leak Detection

- Keep a list (or a tree) of in-use memory
 - For brevity source code not included here

- Four routines needed:

- Add a node to the tree

```
void *my_mem_add(void *mem, long size, char *file, long line)
```

- Delete a node from the tree

```
void *my_mem_delete(void *mem)
```

- Find a node in the tree and return its size

```
long my_mem_size(void *mem)
```

- List all nodes in the tree

```
void my_mem_leaks(void)
```

- This list is called the in-use list

Changes to malloc()

```
void *do_my_malloc(long line, char *file, size_t size)
{
void *mem;

if (size == 0)
{
printf("%s(%d): error, zero length malloc\n", file, line);
return NULL;
}
#ifndef MALLOC_DEBUG_ALL
else
    printf("%s(%d): log, malloc(%d)\n", file, line, size);
#endif

mem = malloc(size);
if (mem == NULL)
{
printf("%s(%d): error, malloc failure (NULL returned)\n", file, line);
return NULL;
}
memset(mem, BAD_MEM, size);
if (my_mem_add(mem, size, file, line))
    return mem;
else
{
free(mem);
return NULL;
}
}
```

Changes to free()

- From my_malloc.h

```
#define my_free(p) do_my_free(__LINE__, __FILE__, p)
void do_my_free(long line, char *file, void *p);
```

- From my_malloc.c

```
void do_my_free(long line, char *file, char *mem)
{
size_t size;

size = my_mem_size(mem);
if (size > 0)
{
    my_mem_delete(mem);
    memset(mem, BAD_MEM, size);
    free(mem);
}
}
```

- do_my_free()

- Clears the unused memory (set equal to BAD_MEM)
- Removed it from the in-use list

- Can we fit p=NULL into the macro somehow?

Example

- From main.c

```
int main(int argc, char *argv[])
{
char *g;
int index;

atexit(my_mem_leaks);
g = my_malloc(10);

for (index = 0; index <= 10; index++)
    g[index] = 0;

return 0;
}
```

- Output

```
main.c(11): warning leaked 10 bytes
```

- A program cannot possibly tell where it *should* have been freed, only where it *was* allocated

The Special-Case of realloc()

- Problem
 - realloc() *sometimes* moves memory
 - Leads to irreproducible behavior
 - Example: dangling pointers in a tree (node->str)
- Solution
 - Debug version
 - Always move the block
 - Release version
 - Avoid moving the block if possible
- Remember to debug *both* versions
- *Always* test both debug and release versions

Uses of the In-Use List

- At termination of program list all in-use memory
 - These are program leaks

```
atexit(my_mem_leaks);
```
- Does a pointer point to valid memory?
 - Check the size is greater than zero
- Validate pointer before calling system routines
 - Prevent memory overwrites
 - Is destination a valid pointer in call to strcpy()?
- Validate block-size before calling system routines
 - Prevent memory overwrites
 - Is destination large enough in call to strncpy()?
- Pair free() and malloc() calls
 - The allocation location is in the in-use list
- Can this technique can be extended to the stack?

Write Outside Buffer

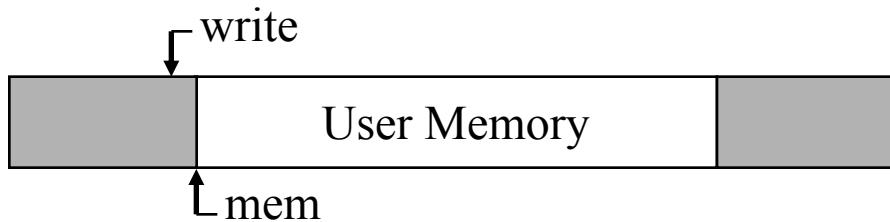
- Write past end of buffer (memory over-run)

```
char *mem = my_malloc(10);  
for (index = 0; index <= 10; index++)  
    mem[index] = 0;
```



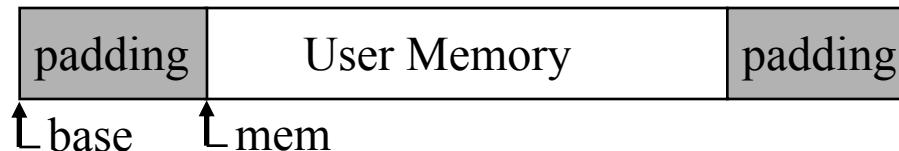
- Write before beginning of buffer (memory under-run)

```
char *mem = g = my_malloc(10);  
*mem-- = 0;  
*mem-- = 0;
```



Catching Outside Buffer Writes

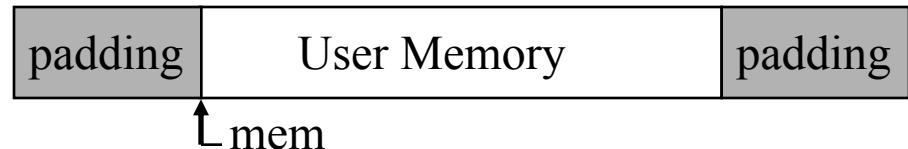
- In `do_my_malloc()` insert padding at each end



- Similar to last lecture's “header”
 - Allocate $\text{size} + 2 * \text{sizeof(padding)}$ bytes
 - Word align everything correctly
 - Put one at each end of the block
 - Initialize the whole block (`BAD_MEM`)
 - Return $\text{mem} + \text{sizeof(padding)}$
 - This hides the existence of the padding

Catching Outside Buffer Writes

- In do_my_free(), check the padding is BAD_MEM



- In do_my_free()
 - Check under-writes (underflow)

```
if (!my_pad_check(mem - sizeof(padding), sizeof(padding)))
    printf("%s(%d): error, memory under-run\n", file, line);
```
 - Check over-writes (overflow)

```
if (!my_pad_check(mem + my_mem_size(mem), sizeof(padding)))
    printf("%s(%d): error, memory over-run\n", file, line);
```

• my_pad_check()

```
long my_pad_check(char *where, long size)
{
char *ch;

for (ch = where; ch < where + size; ch++)
    if (*ch != (char)BAD_MEM)
        return 0;

return 1;
}
```

- How (and when) can this check fail?

More Besides...

- These techniques do not catch everything
 - Dangling pointers to re-allocated memory
 - Memory writes outside the padding
- Thought experiment: is it possible to
 - Check the padding during runtime?
 - Keep a list of all used *and* unused memory?
 - Verify the heap is OK?
 - Verify the free-list and the in-use list?
 - Verify every single memory access in the program?
 - Set $p = \text{NULL}$ after each call to $\text{free}(p)$?
- How would we do all this?

Conclusions

- Memory leaks are easily caught
- Memory over-runs and under-runs are easily caught
- Memory size mismatches are easily caught
- Uninitialized memory blocks easily spotted in debugger
- Always manage the memory manager
- Wait a second...
 - Shouldn't there be a library that does all this?

References

- S. Maguire, *Writing Solid Code*, Chapter 3

COSC345

Software Engineering

Memory Checkers

and

Dr Memory

Outline

- Typical Errors
- Tools
 - Source code libraries
 - Hardware OS Traps
 - Heap replacement
 - Source code instrumentation
 - Object code instrumentation
 - A very old (but still interesting) comparison
 - Valgrind

Memory Errors

- Typically
 - Leaks
 - Over-runs and under-runs
 - Dangling pointers
- Symptoms
 - Unexpected and apparently random crashes
 - In system heap routines
 - In previously stable places
- These bugs are hard to find, but important to fix
 - Unpredictable and occur in a different place to the symptoms
 - But often remain unidentified and unfixed
- There are many tools to help us find these errors

Downloadable Libraries

- Many including: MEMWATCH (<https://memwatch.sourceforge.net/>)
 - Add a header file to your source code files, and compile with MEMWATCH defined or not
 - ANSI C
 - Logging to file or user function using TRACE() macro
 - Fault tolerant, can repair it's own data structures
 - Detects double-frees and erroneous free's
 - Detects unfreed memory
 - Detects overflow and underflow to memory buffers
 - Can set maximum allowed memory to allocate, to stress-test app
 - ASSERT(expr) and VERIFY(expr) macros
 - Can detect wild pointer writes
 - Support for OS specific address validation to avoid GP's (segmentation faults)
 - Collects allocation statistics on application, module or line level
 - Rudimentary support for threads (see FAQ for details)
 - Rudimentary support for C++ (disabled by default, use with care!)

Downloadable Libraries

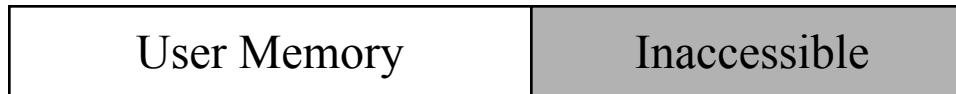
- Other tools include:
 - Largest allocated chunk
 - Smallest allocated chunk
 - Maximum memory used
 - Largest number of concurrent allocations
- There are dozens of downloadable tools with similar behaviour

Hardware and OS Traps

- Hardware already traps some errors
 - Read or Write to a NULL pointer (protected zero page)
 - Stack overflow (protected page at bottom of stack)
- UNIX
 - Segmentation fault
 - Reading or writing somewhere not permitted (but exists)
 - E.g. write into code segment
 - Bus error
 - Access somewhere the computer physically cannot address
 - E.g. write outside allocated virtual address space
- Use the Virtual Memory Manager to heap check?
 - Electric Fence

Electric Fence (efence)

- Detects two types of error
 - Memory over-runs (*or* under-runs)
 - Access to freed memory
- Electric Fence
 - Places each chunk before an inaccessible block (no r/w)



- Over-runs cause a segmentation fault
- Same approach for under-runs (but not concurrently)
- Freed blocks are marked inaccessible
- Very little run-time overhead
 - But “wastes” a lot of virtual address space

Heap Replacement

- `malloc()` and `free()` are very slow
 - We saw that in a previous lecture
- Replace with faster routines
 - Different approaches
 - Size based
 - Used in Object Oriented languages
 - Buddy system
 - Allocate 2^n bytes (split a 2^{n+1} block if none available)
 - Where there is a minimum n (e.g. 6 (32-bytes))
 - Check the heap at the same time

HeapAgent (Windows)

- Replace heap routines with calls to HeapAgent
- Each allocated block is initialized, has padding, and has a header
 - Header contains
 - Name of allocating function (file and line)
 - Size of allocation
 - Read only / no-free / no-realloc status



- Blocks are reinitialized on free
- All heap calls verify their parameters and the block
- Heap is verified as a background task
 - No impact as threaded at idle priority
 - Catches over-runs after they occur but before a free

Source Code Instrumentation

- Pre-process the source code as part of the build
- Inserts
 - Calls to checking functions everywhere that
 - Allocates, frees, or reallocates memory
 - Assigns to a pointer
 - Reads a pointer
 - References an array
 - Calls a function
 - Both before and after the call
 - Parameter checks for all system routines (API checks)
- Instrumented programs are slow to run
- Examples
 - BoundsChecker, Insure++

BoundsChecker Example

- Original

```
if (m_hsession)
    gblHandles->ReleaseUserHandle(m_hsession);
if (m_dberr)
    delete m_dberr;
```

- Instrumented

```
if (m_hsession)
{
    _Insight_stack_call(0);
    gblHandles->ReleaseUserHandle(m_hsession);
    _Insight_after_call();
}
_Insight_ptr_check(1994, (void **) &m_dberr, (void *) m_dberr);
if (m_dberr)
{
    _Insight_deletea(1994, (void **) &m_dberr, (void *) m_dberr, 0);
    delete m_dberr;
}
```

Source Instrumentation Limits

- Does not have control over the heap
 - Errors are errors because “improper” use is identified
 - Can’t catch all heap abuse (some dangling pointers)
- Prone to false positives
 - Pointers make it hard to tell what “proper” use is!
- Instrumentation process prone to inserting errors
 - The debug and release programs are different source code
- Relies on module instrumentation
 - Source code must be available
 - Can’t (easily) catch errors in OS APIs and C-RTL
- These preprocessors can change compiler options
 - E.g. turn off warnings
 - Introduces bugs otherwise caught by the compiler

Object Code Instrumentation

- Also known as object code insertion (OCI)
- Inserts checking code around every instruction that
 - References, allocates, or deallocates memory
- Every object file (and DLLs) is instrumented
 - This includes the OS and the C-RTL
- Inserted instructions are machine code checks
- Example : Purify (Windows / UNIX)

Purify

- Purify detects (before they occur)
 - Array bounds errors
 - Accesses through dangling pointers
 - Uninitialized memory reads
 - Free of non-freeable memory
 - Mismatched new / free()
 - Overlapping source and destination blocks
 - Memory leaks
- Problems:
 - The instrumentation can introduce errors
 - It is CPU specific, sensitive to
 - Compiler upgrades
 - Operating system changes
 - Processor upgrades

Compile-Time Comparison

- From article referenced on final slide (from 1997)
 - Comparison of *old* versions of each approach
 - Some of MFC (~110 KLOC)
 - Averaged over three runs

| <i>Product</i> | <i>Compile Time</i> | <i>Factor</i> |
|--------------------|---------------------|---------------|
| Visual C++ (debug) | 23:25 | 1 |
| HeapAgent 3.0 | 23:25 | 1 |
| Purify 4.0 | 24:20 | 1.04 |
| BoundsChecker 4.0 | 100:55 | 4.31 |

- Conclusion: source code instrumentation is slow

Analysis

- HeapAgent
 - Does not affect compile time
 - Heap manipulation DLLs are replaced by HeapAgent
 - The actual application (except heap management) is run
- Purify
 - Instruments the run-time version
 - Requires debug version at run-time
 - Requires more disk space
 - Must store instrumented version somewhere
- Boundschecker
 - Compile-time instrumentation is complex

Run-Time Comparison

- From article referenced on final slide
 - GNU Bison 1.25. (11 KLOC)
 - Measured execution real-time
 - Averaged over three runs

| <i>Product</i> | <i>Run Time</i> | <i>Factor</i> |
|-------------------------|-----------------|---------------|
| Visual C++ (debug) | 0:04 | 1 |
| HeapAgent | 0:04 | 1 |
| Purify | 0:44 | 11 |
| BoundsChecker (quick) | 6:56 | 104 |
| BoundsChecker (maximal) | 66:18 | 994 |

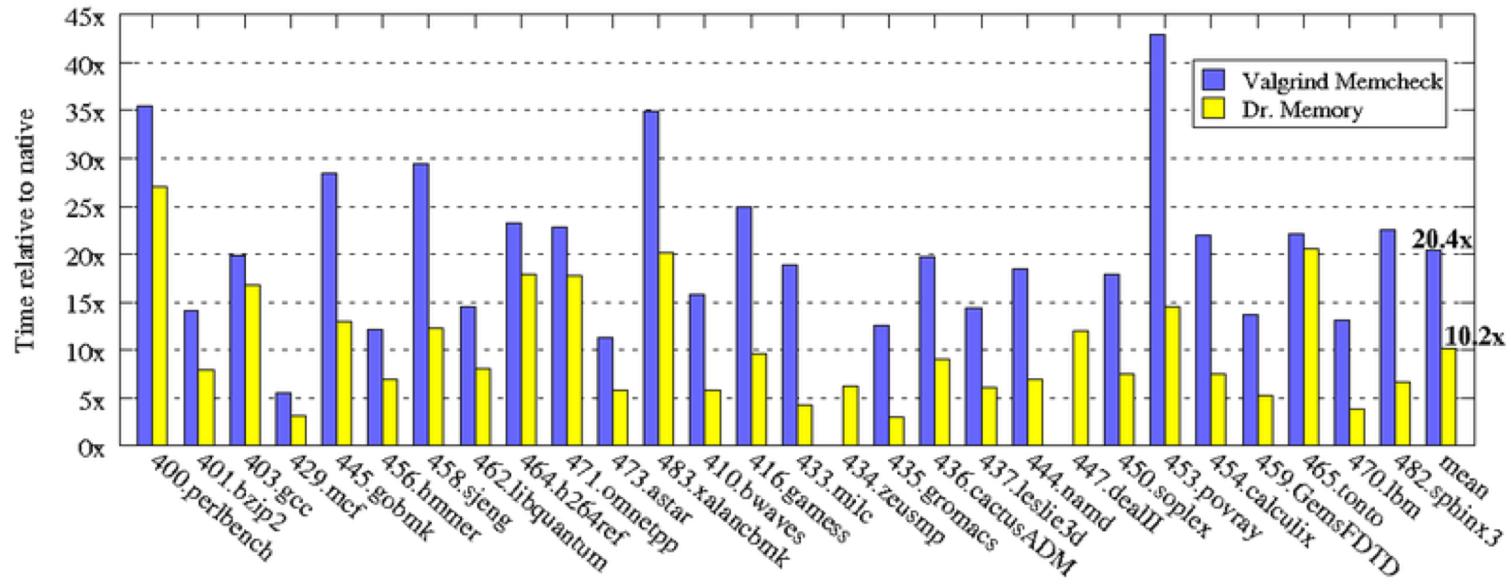
- Conclusion: source code instrumentation is slow

Analysis

- HeapAgent
 - Only watches the heap
 - Check using a background thread (idle cycles)
 - Padding chosen to cause CPU exceptions
 - Implements its own heap (using SmartHeap)
 - Checks are implemented *in* the heap routines
- Purify
 - Checking code is complex (because the CPU is complex)
 - Causes an application to be larger
 - Affects the CPU instruction pipeline
 - Affects the CPU cache
 - Can't share DLLs with other apps
 - Large state table tracks the state of every byte of memory
- BoundsChecker
 - Normally fast operations become procedure calls
 - Dereference a pointer becomes a procedure call!

Dr Memory (Windows)

- Virtual CPU with JIT-instrumentation
- Reads (and uses) debug data read from executable
 - Programming language independent
 - Supports ARM and Intel
- Adds checking code to every memory access
 - Instruments your program, the OS and C-RTL



Dr Memory Checks

- Accesses of uninitialized memory
- Accesses to unaddressable memory
 - Including outside of allocated heap units and heap underflow and overflow
- Accesses to freed memory
- Double frees
- Memory leaks
- Handle leaks (windows)
- API usage errors (windows)
- Accesses to un-reserved thread local storage slots.

Dr Memory Example

- **example.c**

```
1. #include <stdio.h>
2. #include <stdlib.h>
3.
4. int main(int argc, char *argv[])
5. {
6.     char *buffer;
7.
8.     buffer = malloc(10);
9.     putchar(*buffer);
10.    buffer = malloc(10);
11.    buffer[10] = 'B';
12.    free(buffer);
13.    *buffer = 'A';
14.    free(buffer);
15.    return 0;
16. }
```

- **makefile**

```
example.exe : example.c
             cl /Zi /MT /EHsc /Oy- /Ob0 example.c
```

- **drmemory example.exe**

Dr Memory Output

```
Dr. Memory version 2.6.0 build 0 built on Sep 21 2023 17:38:58
Windows version: WinVer=105;Rel=2009;Build=22621;Edition=ProfessionalWorkstation
Dr. Memory results for pid 13668: "example.exe"
Application cmdline: "example.exe"
Recorded 124 suppression(s) from default C:\Program Files (x86)\Dr. Memory\bin64\suppress-
default.txt
```

Error #1: UNINITIALIZED READ: reading register al

```
# 0 write_text_ansi_nolock [minkernel\crts\ucrt\src\appcrt\lowio\write.cpp:431]
# 1 _write_nolock [minkernel\crts\ucrt\src\appcrt\lowio\write.cpp:69]
# 2 _write_internal [minkernel\crts\ucrt\src\appcrt\lowio\write.cpp:65]
# 3 main [C:\Users\andrew\programming\temp\example.c:9]

Note: @0:00:00.300 in thread 7732
Note: instruction: cmp    %al $0x0a
```

Error #2: UNADDRESSABLE ACCESS beyond heap bounds: writing 0x000001de8064c18a-0x000001de8064c18b 1 byte(s)

```
# 0 main [C:\Users\andrew\programming\temp\example.c:11]
Note: @0:00:00.300 in thread 7732
Note: refers to 0 byte(s) beyond last valid byte in prior malloc
Note: prev lower malloc: 0x000001de8064c180-0x000001de8064c18a
Note: instruction: mov    $0x42 -> (%rcx,%rax)
```

Dr Memory Output

```
Error #3: UNADDRESSABLE ACCESS of freed memory: writing 0x000001de8064c180-0x000001de8064c181 1 byte(s)
# 0 main [C:\Users\andrew\programming\temp\example.c:13]
Note: @0:00:00.300 in thread 7732
Note: prev lower malloc: 0x000001de8064c150-0x000001de8064c15a
Note: 0x000001de8064c180-0x000001de8064c181 overlaps memory 0x000001de8064c180-0x000001de8064c18a
      that was freed here:
Note: # 0 replace_free [D:\a\drmemory\drmemory\common\alloc_replace.c:2710]
Note: # 1 main [C:\Users\andrew\programming\temp\example.c:12]
Note: instruction: mov $0x41 -> (%rax)
```

Error #4: INVALID HEAP ARGUMENT to free 0x000001de8064c180

```
# 0 replace_free [D:\a\drmemory\drmemory\common\alloc_replace.c:2710]
# 1 main [C:\Users\andrew\programming\temp\example.c:14]
Note: @0:00:00.316 in thread 7732
Note: prev lower malloc: 0x000001de8064c150-0x000001de8064c15a
Note: memory was previously freed here:
Note: # 0 replace_free [D:\a\drmemory\drmemory\common\alloc_replace.c:2710]
Note: # 1 main [C:\Users\andrew\programming\temp\example.c:12]
```

Error #5: LEAK 10 direct bytes 0x000001de8064c150-0x000001de8064c15a + 0 indirect bytes

```
# 0 replace_malloc [D:\a\drmemory\drmemory\common\alloc_replace.c:2580]
# 1 main [C:\Users\andrew\programming\temp\example.c:8]
```

Dr Memory Output

FINAL SUMMARY:

DUPLICATE ERROR COUNTS:

SUPPRESSIONS USED:

ERRORS FOUND:

| | |
|------------------|---|
| 2 unique, | 2 total unaddressable access(es) |
| 1 unique, | 1 total uninitialized access(es) |
| 1 unique, | 1 total invalid heap argument(s) |
| 0 unique, | 0 total GDI usage error(s) |
| 0 unique, | 0 total handle leak(s) |
| 0 unique, | 0 total warning(s) |
| 1 unique, | 1 total, 10 byte(s) of leak(s) |
| 0 unique, | 0 total, 0 byte(s) of possible leak(s) |

ERRORS IGNORED:

1 potential error(s) (suspected false positives)

(details: C:\Users\andrew\AppData\Roaming\Dr. Memory\DrMemory-example.exe.13668.000\potential_errors.txt)

1 potential leak(s) (suspected false positives)

(details: C:\Users\andrew\AppData\Roaming\Dr. Memory\DrMemory-example.exe.13668.000\potential_errors.txt)

20 unique, 97 total, 26880 byte(s) of still-reachable allocation(s)
(re-run with "-show_reachable" for details)

Details: C:\Users\andrew\AppData\Roaming\Dr. Memory\DrMemory-example.exe.13668.000\results.txt

A Note on JAVA

- Problems in Java
 - References to objects no-longer needed
 - Very bad if its a large dynamic structure (tree, list, etc)
 - System resource problems
 - Open files that should be closed
 - External “non-JAVA” memory
 - Java Native Interface(JNI)
 - OS resources (bitmaps, etc)
- These all need checking too
 - Some of these tools can now work with JAVA

References

- MicroQuill Software Publishing, Inc, (1997)
Comparing and contrasting the runtime error detection technologies used in HeapAgent™ 3.0, Purify NT® 4.0, and BoundsChecker Pro™ 4.0.
http://www.microquill.com/heapagent/ha_comp.htm
- This article is an excellent introduction to the three different technologies and much of this lecture is from it. It's published by the authors of HeapAgent, and it is now very old.
- Dr. Memory: <https://drmemory.org/>

COSC345

Software Engineering

Linkers Loaders Libraries

Outline

- Sources of error
 - A bunch of statistics
- Compiling
- Linking
- Loading
- Libraries
 - Static libraries
 - Overlays
 - Shared libraries
- DLLs

Revision: Sources of Errors

- Analysis
 - Misunderstanding of the process
 - Leads to specification errors
 - Misunderstanding of the specification
 - Leads to implementation errors
- Architecture
 - A design of the software that doesn't fit the needs
 - Inappropriate algorithms
 - E.g. linear search where binary search needed
- Construction
 - Logic errors in the program
 - Need debugging
 - Integration errors
 - Re-implement some interfaces

Revision: Effect of Size on Errors

- Errors per KLOC increases with project size
- Consistent for each developer
 - Regardless of programming language

| Project Size (KLOC) | Density (Err / KLOC) |
|---------------------|----------------------|
| <2 | 0-25 |
| 2-16 | 0-40 |
| 16-64 | 0.5-50 |
| 64-512 | 2-70 |
| >512 | 4-100 |

Revision: Effect of Size on Productivity

- Boehm, Gray, Seewaldt
 - Compared productivity of small and large groups
 - Small groups 39% more productive than larger groups
 - Small: 2 developers
 - Large: 3 developers

| Project Size (KLOC) | LOC / month |
|---------------------|-------------|
| <2 | 333-2000 |
| 2-16 | 200-1250 |
| 16-64 | 125-1000 |
| 64-512 | 67-500 |
| >512 | 36-250 |

Revision: Problem and Solution

- As a project becomes larger
 - The productivity decreases
 - The error rate increases
 - Communication takes longer
- Developers should work in small groups
 - Directed by a “chief programmer”
 - See lecture notes for “Team Organization”
- Developers should build small components
 - Divide a large project into small parts

From Engineering

- Reuse of existing components
 - Not just nails but engines and turbines
- Components tried and tested and work
 - So error rate can be reduced by using them
- Software reuse
 - Concepts (patterns)
 - Functions
 - Objects (libraries)
 - Build objects and expose interfaces
- Specify interfaces early
 - This way parallel development can start
 - Agree on the procedural contract (input and output)

Concept Reuse

- Reuse a concept already shown to work
 - Algorithms from a textbook
 - Algorithms from previous projects
 - Patterns
- Coding is only 1/6 (16%) of the development time
 - Re-implementation is (often) an acceptably low cost

Function Reuse

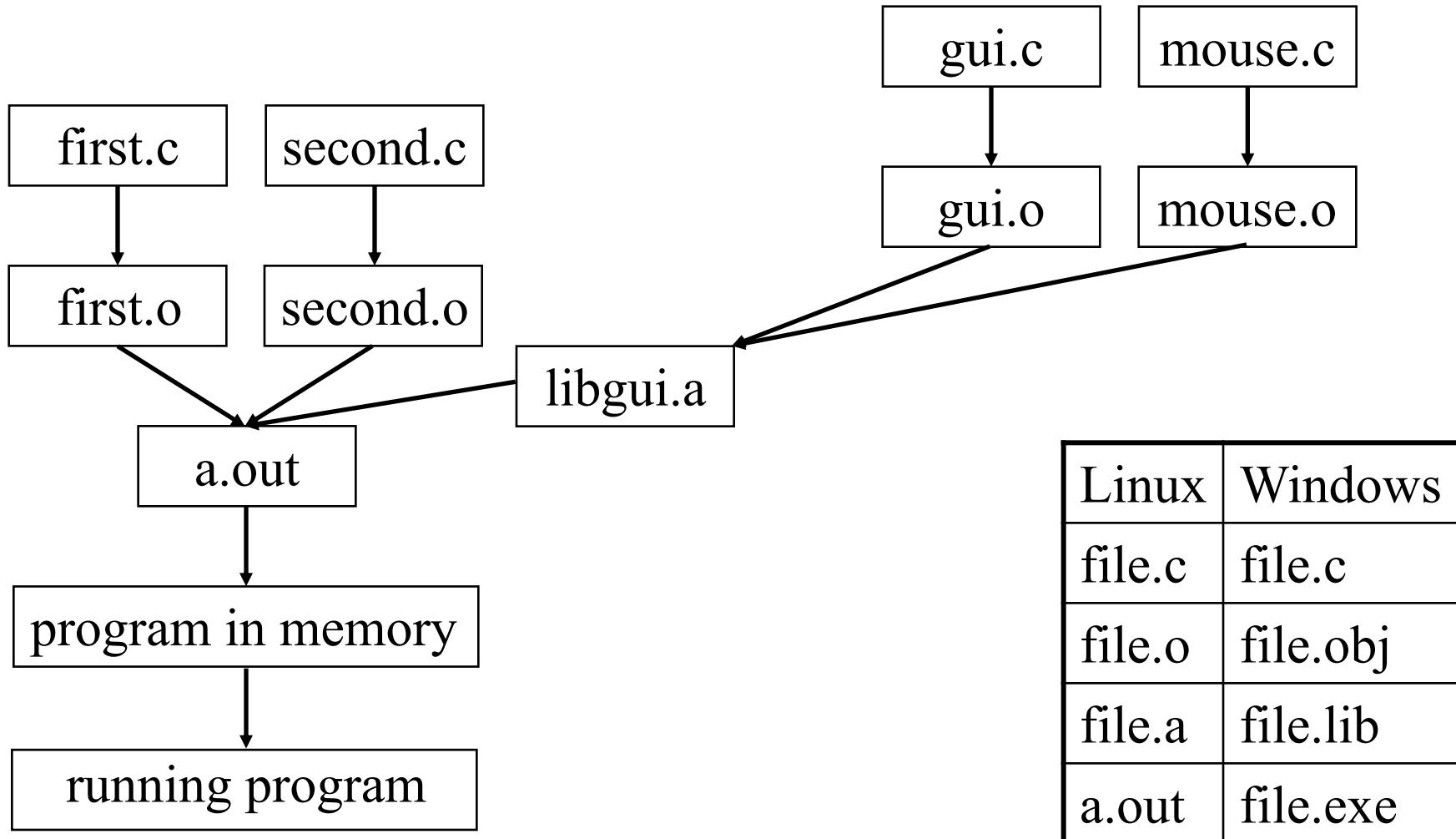
- Source code from the web
 - Web-sites from algorithms textbooks
 - R. Sedgewick, Algorithms in C++,
 - <http://www.cs.princeton.edu/~rs/Algs3.cxx1-4/code.txt>
 - SourceForge
 - Archive of (many) open-source development projects
 - GitHub / Bitbucket
 - Archive of (many) open-source projects
 - Source code from prior projects
 - In your own archives
 - Don't have an archive? Why not?
 - Keep all source code in a version control system

Libraries

- A file containing a collection of related functions
 - Math, GUI, etc.
- Libraries give us
 - Functional reuse of object code
 - Functional reuse of programmer's work
 - Easy maintenance
 - Faster compile times
 - Because libraries are pre-compiled for the current project

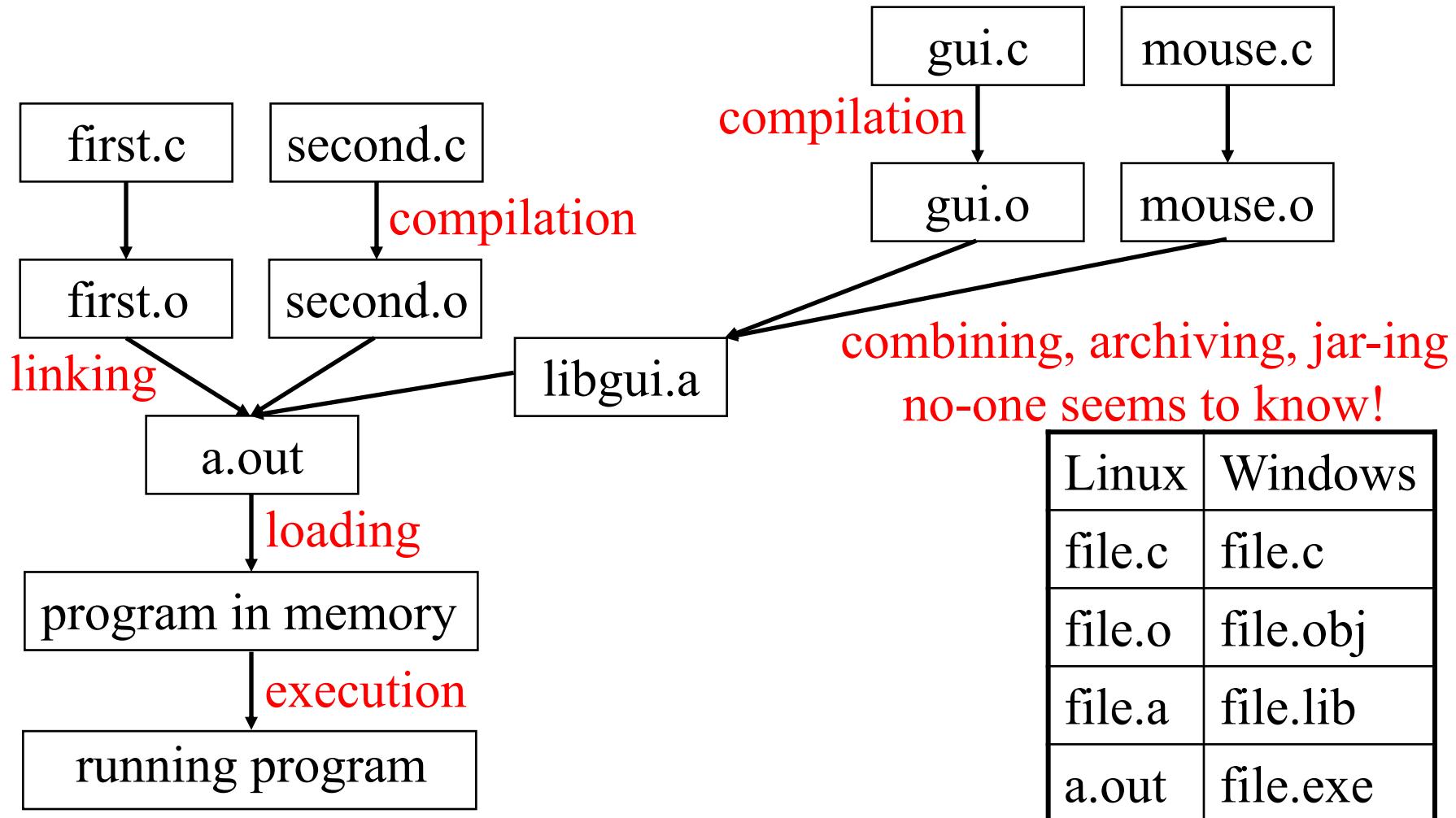
From Source To Execution

- What is responsible for each step (each arrow)?



From Source To Execution

- What is responsible for each step (each arrow)?



Separate Source Files

- `use.c`

```
extern int another;  
  
int main(void)  
{  
    another = 1234;  
}
```

- `declare.c`

```
int another;
```

- How does the compiler know:
 - Where **another** is stored in memory?
- How can the compiler produce the assembly code?

The Compiler

- Leave gaps in assembly code when referencing externs
- `use.c`

```
extern int another;  
int main(void)  
{  
    another = 1234;  
}
```

- Compiler output for `use.c`

```
0001 18c0          sect 0  
0002              _main:  
0003 18c0 cc 04 d2      ldd #1234  
0004 18c3 fd 00 00      std  _another  
0005 18c6 39          rts
```

- On line 0004 `another` is at location 0000!

The Compiler

- Allocate space for global variables
- declare.c

```
int another;
```

- Compiler output for declare.c

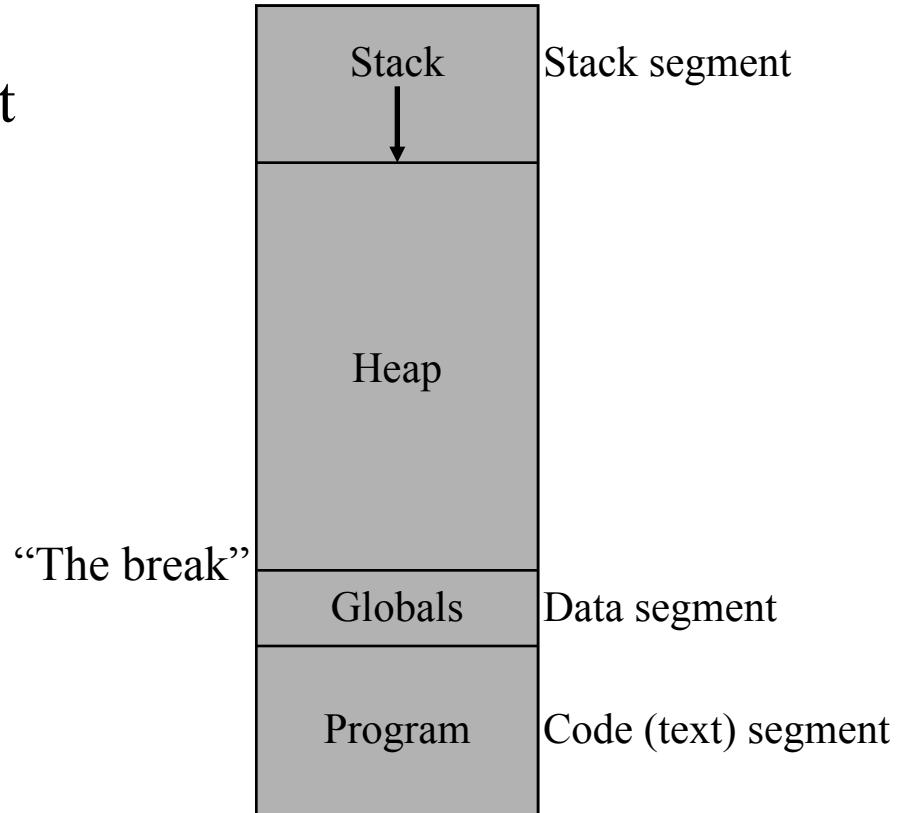
```
0001 4000          sect 1
0002                  _another:
0003 4000          RMB  2
```

- On line 0003 space is allocated for **another**

Revision: Segments

- Program
 - The code or text segment
- Local Variables
 - The stack segment
- Global variables
 - The data segment
- In the example

| <i>Segment</i> | <i>Example</i> |
|----------------|----------------|
| Code | sect 0 |
| Data | sect 1 |
| Stack | |



Software Model

The Linker

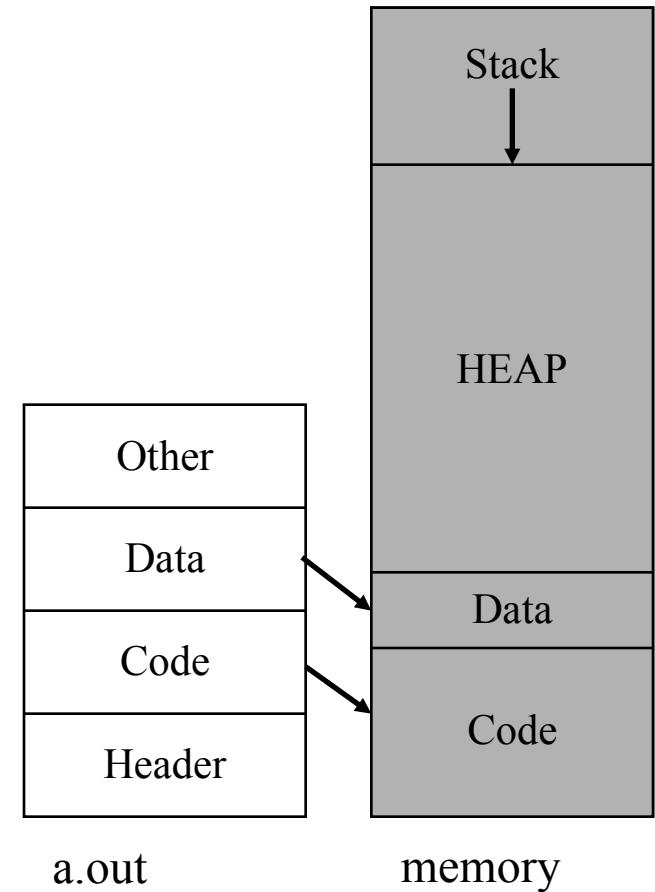
- Loads a set of *object* files and outputs an *executable* file
- Each input file is a set of segments
 - Code / Data
 - Symbol table
- Pass 1
 - Scan the input files to compute the segment sizes
 - Collect the symbol tables together
- Process
 - Allocate locations for each symbol
 - Layout the symbols in the output (executable) file
- Pass 2
 - Read and relocate the object code
 - Replace symbol references with memory locations
 - Copy segments into the output (executable) file

The Combined (Linked) Program

```
0001                                * define starting addresses
0002 18c7                          sect 0          * code
0003 1800                          org $1800
0004 0000                          sect 1          * data
0005 4000                          org $4000
0006 7ffb                         stackbase equ $7ffb
0007
0008                                *
0009                                *
0010 1800                         sect 0
0011 1800 8e 7f fb                lds #stackbase
0012 1803 bd 18 c0                jsr _main
...
0002
0003 18c0 cc 04 d2
0004 18c3 fd 40 00
0005 18c6 39
0006
0001 4000
0002
0003 4000
                                _main:
                                ldd #1234
                                std _another
                                rts
                                L1.use:
                                sect 1
                                _another:
                                RMB 2
```

The Loader

- Read the executable file
- Allocate memory space for it
- Load each segment
- Initialize the stack (if needed)
 - Create stack segment (if needed)
- Set up environment, etc.
- Jump to program start
 - Initializes the stack (if needed)
- Sometimes the stack is created and initialised by the loader, sometimes this is done by the program



Relocation

- In some systems (old and embedded systems)
 - Multiple programs in memory at one time
 - No virtual address space
- Executable format has a patch or relocation table
- Loader
 - Loads the executable at some base location
 - For every direct memory address:
 - Adds the base to the address
 - Uses the patch table to do this

Relocation

- In more modern systems
 - All references are relative to the Program Counter (PC)
 - Intel RIP-relative addressing
 - So executables never need to use absolute addresses
- Some memory management units have special “tagging” for different executables in a single address space (for efficiency)
- Some hardware requires special attention
 - E.g. Intel 8086 / 8088 segmentation

Static Libraries

- Just a collection of object files stored together
 - In Unix they are created using **ar**
 - The archive program
 - Using Windows they are created using **LIB**
- Linux

```
gcc -c first.c
gcc -c second.c
ar -r my.a first.o second.o
gcc -c use.c
gcc use.o my.a
```

- Windows
- ```
cl -c first.c
cl -c second.c
lib /out:my.lib first.obj second.obj
cl -c use.c
link use.obj my.lib
```

# DLL / SO / ActiveX

- DLL: Dynamic link libraries (Windows)
- SO: Shared objects (Linux)
  - Load and bind at run time
    - Allows library to change after program written
    - If done with relocation then very little binding is needed
- VBX / OCX / ActiveX / etc.
  - Dynamic link and load of objects
  - Load at run time
    - Interrogate the ActiveX Object to see what methods it supports
      - Allows object binding at runtime
        - » So you can change the behaviour based on what is loaded

# Advantages and Disadvantages

- Advantages
  - Reduced overall development costs
  - Reduced maintenance costs
  - Reduced error rates
  - Increased productivity
- Disadvantages
  - Finding suitable libraries is difficult
  - Understanding the libraries is hard
    - And must be done before use
  - Stability must be ensured
  - The library must be tested

# Considerations

- Support
  - What happens if the supplier goes out of business?
- Source code
  - Is it available – do you want it available
- Software lifecycle
  - Does the lifecycle of the library match yours
- Portability
  - What platforms will you support in the future?
    - Does the supplier support that platform?
- Skills of the developers
  - Are the suppliers skilled to produce the library?
  - Are your developers skilled to use the library?
- How (mission / life) critical is the software?

# References

- J. Levine, *Linkers and Loaders*
- S. McConnell, *Code Complete*, Chapter 21
  - Tables in these notes taken from this chapter
- I. Sommerville, *Software Engineering*, Chapter 18

# COSC345

# Software Engineering

Components

# Outline

- What are components
- Interfacing and integration
- Types:
  - Frameworks, services, data storage
- Multitiered architectures
  - Decoupling and abstraction
- Distributed Components
  - COM and DCOM (COM+)
- Virtual machines and Docker
- Interchange formats
- Distributed component software models

# What are Components?

- Different sorts of pre-constructed pieces used to build software including:
  - Modules, libraries, frameworks, services
- You need to know:
  - How to interact with each component
  - When the component should be connected
    - Compile-time (library) vs. run-time (DLL or JAR)
  - The component's lifecycle?
    - Linked by compiler
    - Contained within another process
    - Contained within the Operating System
    - Distributed
    - Etc.

# Interacting with Components

- How tight is the integration with the programming language?
  - Native interface calls (APIs)
  - Operating System interactions (IPC or RPC)
  - Language extensions
    - SWIFT **extension**, C++ operator overloading, etc.
  - Operating System extensions
    - Relational Databases etc.
  - Some require data type conversions
- Component interaction
  - **Synchronous**: typically use function or method calls
  - **Asynchronous**: non-blocking calls with call-backs or interrupts

# Integration

- Could be done in the pre-processor
  - Our memory checking code from previous lectures
- Often done by the compiler
  - Other peoples classes (e.g. JPEG library)
- Can be runtime class-loader, e.g., Java
- May be entirely dynamic and at run-time
  - Dynamic link libraries (DLLs) and shared objects (SOs)
  - Your program is already running before it's even known if the DLL or SO is available to load
  - Downside: DLL-hell
    - The need to track dependencies

# Lifecycle

- Directly integrated?
  - No separate component lifecycle
- DLLs risk unexpected versions
  - But the OS might host multiple versions of library
- Component may be in a separate process
  - Component's start and stop may be independent of the app
  - Requires inter-process communications of some sort
- Components may be in a different OS (virtualisation)
- Possibly on a different set of machines (distribution)

# Types, Functions, and OO

- Just a set of **type definitions**
- More typically **code** too, e.g. functions or methods
- **Object Oriented** approaches provide types *and* code
  - OO languages support visibility restriction for code isolation
- Languages that support **modules** provide namespace isolation, irrespective of OO access control to classes
  - Prevents more than one component having a method of (globally) the same name
- Collections of types, methods, and classes (pre-implemented) is usually called a **library** or **API**

# Frameworks

- May impose structure on the way you code your app
  - May require a particular filesystem structure
  - Might be sensitive to your type and variable names
  - Often aiming to make your code more compact:
    - **Convention over Configuration (CoC)** promoted in Ruby on Rails: Database table names (by default) map from variable names
    - **Don't repeat yourself (DRY)** principle may cause hidden effects
- Frameworks may help manage heterogenous concerns:
  - e.g., web frameworks not just HTML, but also CSS & JavaScript

# Side Note: DRY

- DRY: Don't Repeat Yourself!
  - “Every piece of knowledge must have a single, unambiguous, authoritative representation within a system” (Hunt & Thomas) including “"database schemas, test plans, the build system, even documentation""
  - Reduce repetition of code
    - Increases abstraction and pattern use (e.g. collection classes)
  - Advantages:
    - Increased maintainability
    - Increased readability
    - Increased reuse
    - Easier testing

# Side Note: WET

- WET: Write Everything Twice!
  - Or “write every time:”, “we enjoy typing” or “waste everyone's time”
- Common in multi-tiered architectures
  - Adding a comment field on a form in a web application
    - The text string “comment” might be repeated in the label, the HTML tag, in a read function name, a private variable, database DDL, queries, and so on.
    - A DRY approach eliminates that redundancy by using frameworks that reduce or eliminate all those editing tasks except the most important ones, leaving the extensibility of adding new knowledge variables in one place
- From: [https://en.wikipedia.org/wiki/Don%27t\\_repeat\\_yourself](https://en.wikipedia.org/wiki/Don%27t_repeat_yourself)

# Services

- Typically much more heavyweight than libraries
- Usually a separate process responding to requests
  - Often asynchronous even if they appear synchronous
    - Usually multi-threaded
- Independent operation in terms of:
  - Logging
  - Resource use
  - Failure handling

# Structured Data Storage

- Programming languages usually don't include DBs
  - Even if they do provide interfaces to databases
- Service approach: communicate with a database
  - Uses client library that is linked to your code
  - Talks to a separate process (service) running the DB
- Framework approach: object-relational mapping (ORM)
  - Handles common case of saving / restoring state of variables
  - Database table names auto-determined from type names
  - Database field types auto-determined from types of variables

# Multitiered Architectures

- Many uses of services can be abstracted into a design pattern, e.g., multitiered architectures
- Web application three-tier architecture:
  - Presentation layer
    - Handles formatting templates
  - Business logic layer
    - Code specific to the application
  - Persistence layer
    - Use a back-end relational database
- Large-scale applications may include a cache tier
  - E.g. Search Engines have many caches
    - Results cache, snippet cache, document cache, etc.

# Decoupling and Abstraction

- Components should provide a *separation of concerns*
  - Otherwise the component does not save on engineering
- **Abstraction** allows developers to ignore detail
  - The interface of the object, not the implementation
  - Downside: may complicate the understanding of the code
- **Decoupling** ensures that abstractions work well
  - Not relying on internals of other components (classes)
  - Downside: may slow down your software

# Inter-Process Communication (IPC)

- When components are in a different *process* we need to interact with them somehow
  - Typical approaches:
    - **Shared memory**
      - Needs very careful coordination, and trust
    - **Data stream**
      - TCP/IP network socket, etc.
    - **Message passing**
      - D-Bus, etc.
    - Via the **filesystem**
  - End result should ideally be robust and efficient
    - Might also consider support for logging, debugging, etc.

# Distributed Components

- When components are distributed, trust and efficiency are big issues
  - Can the system do the round-trip in less time than it would take to do the work locally
  - Do you trust the other end with your information
    - Is the communications open to an attack (man-in-the-middle)
  - Is the remote service available any other way?
    - E.g. Using the Google APIs

# Microsoft's Component Object Model

- COM provides a language-independent notion of objects that allows software components to interact
  - COM is a runtime framework
    - Insulated from compiler changes
  - COM forces separation of interfaces from implementation
    - Interfaces are specified in an implementation agnostic “language” (IDL)
  - COM facilitates inter-process communication
  - Distributed COM (DCOM) works across networks
- So:
  - Same-process communication, inter-process communications, or distributed communications
    - Programmers need not concern themselves with which
    - Systems are configured as needed

# Microsoft's Component Object Model

- Underlies Windows
  - OLE, ActiveX, DirectX, etc.
- Implemented in
  - Apple Core Foundation
  - .NET
- Underlying technology in
  - Windows Runtime (WinRT)
- COM components are typically distributed as DLLs
- Similar systems include:
  - CORBA, JavaBeans, etc.

# CORBA

- Distributed OO paradigm for heterogeneous systems
  - Programming language and OS independent
- Objects defined using an interface definition language
  - Mappings provided from programming languages to IDL
- Provides distribution transparency
  - An object instance might be local, or remote
  - Convenient, until failures occur
    - Your code must be robust to large access latency differences
- Similar to COM and DCOM, but not proprietary
  - Same advantages and disadvantages

# OS-Integrated Components

- Java and Windows have runtime systems that support embedding of objects within an app
  - **Objects** can be considered to be components
- In contrast: Unix applications are more likely developed independently, but still need to interact
  - **Applications** can be considered to be components
  - Linux D-Bus allows apps to talk to each other

# Example: Pipes (Unix & Windows)

- Very widespread component model
  - Use binary data, but commonly just text
  - E.g. Print the highest 2 numbers less than 1000 that contain a 3
    - `seq 1 1000 | grep 3 | tail -2` gives 983 and 993
- A pipe has independent writer and reader processes
- A Pipe has a limited capacity
  - It has an underlying buffer
- Writes to a pipe when buffer is full will block
  - Until the buffer empties
- Reads from pipe when buffer is empty block
  - Until there's data to read

# Virtual Machines

- Service-based components can be packaged along with their host operating systems
- Originally provided emulation across platforms
  - Recent CPUs and software (hypervisors) allow VMs to run at near-native speeds
- VMs occupy physical resources (memory, etc)
  - Too heavyweight to be useful as software components?

# Docker and Containers

- Containers are like VMs, but *share one kernel*
  - Memory efficient and very quick to start
  - Link parts of their storage and network to the host OS
  - Container file systems are distributed efficiently
    - Common layers are factored out, such as a base Linux layer
    - Can store deltas over the base layer
      - A DB container might only store file system differences
- Components package user-spaces into software components
  - Can link components together via network interfaces
    - TCP/IP or RPC

# RESTful Interfaces

- Representational State Transfer (REST)
- Manipulate web resources using *stateless* operations
  - Resource is something that can be identified
    - A web URL
  - Has a small set of methods
    - HTTP's GET, POST, PUT, DELETE
  - Representations of resources are returned in replies
    - A server may respond with XML, HTML, JSON, etc.
    - RESTful services may provide human and machine readable data
- Responses from server can refer to other resources
  - Client software should be able to discover possible actions
    - Similar to ActiveX controls

# Data Interchange

- Serialising & deserialising data:
  - Application specific file-format?
  - Standardised serialisation format (often via library)
    - Schema might be application-specific
    - Can manipulate data without knowing application semantics
  - Use another component to effect storage and retrieval
    - ORM
    - Relational database
  - Don't forget about internationalisation
    - Often easier to use a pre-existing component to deal with different writing systems (left-to-right or right-to-left) and alphabets (CKJ or Greek) than to invent your own solution

# JavaScript Object Notation (JSON)

- JavaScript has a standard representation for data
  - Which is human readable
- JSON is syntax of a JavaScript “values” but without including code (or comments!)
- Well supported by libraries in many languages
  - But there can be bugs

# Other Interchange Formats

- Extensible Markup Language (XML)
  - Tree structured, human readable text
  - Element can contain key/value attributes
  - Syntax supports inclusion of comments
  - Structural constraints with:
    - DTD or XML Schema
- YAML Ain't Markup Language (YAML)
  - Superset of JSON (in terms of data types)
  - Removes superfluous bracket and brace syntax
  - Relies on indentation for structure (like Python)
  - Includes comments for human readers / writers
- Binary systems such as Google Protobuf

# Service Oriented Architectures (SOA)

- Software components at organisational scale
  - Different vendors offer micro-services that work together
    - e.g., “Process a customer order”, i.e. service is a **business process**
- Dynamically connect components over a network
  - Component roles include service provider and service broker
    - Brokers facilitate **hierarchical service design**, and service markets
  - SOA often implemented using Web Services (WS) technology
  - SOA can also be implemented using CORBA, REST, etc.

# Function as a Service (FaaS)

- With common interchange formats (JSON, etc.) and execution model (REST, etc.) it is often possible build an app by gluing services together
- Each component
  - Scales independently
  - Provides logging and monitoring
- The aim is to build a “serverless” architecture for the app on a pay-as-you-use basis
  - Bootstrapping is inexpensive (pay as you use)
  - Complexity is reduced (use external components)
  - Maintenance is managed by each external service
- Components might be as small as a single function!

# References

- See the Wikipedia articles on
  - COM
  - JSON, XML
  - FaaS, SaaS, PaaS, etc.

# COSC345

# Software Engineering

Ethics and Morals

# Outline

- Positives and Negatives
- GDPR
- ACM Code of Ethics
- Old exam questions

# Positives and Negatives

- Technologists are progressing the state of the art
  - But often less good at considering consequences
- Most systems have positive and negative effects:
  - Email
    - [+] paperless; efficiency
    - [-] spam; phishing; scams
  - Social Media
    - [+] support groups
    - [-] fake news; election damage
  - Tor
    - [+] bypassing oppressive regimes
    - [-] dark web

# Positives and Negatives

- This doesn't just apply to apps
  - CPU speculative execution
    - [+] speed
    - [-] security (Spectre)
  - AI
    - [+] automating boring jobs
    - [-] people need new jobs

# Your Roll

- You may be working on a part, but consider the whole
  - Implications on society may not come from your code
    - They may come from unanticipated use of your code
    - Or from the work as a whole
  - Important cross-cutting concerns can be raised by anyone
    - Security consequences
    - Privacy effects
    - User support

# Your Roll

- Just because it's legal doesn't mean its ethical!
  - Hold your work to a high standard
    - And that of your employers too
- Regulation is emerging that addresses some concerns
  - GDPR
  - ACM Code of Ethics

# GDPR

- User privacy requires careful handling of data
  - Software engineers will effect this handling and often software engineering is not well understood
    - By the law
    - By politicians
- Nonetheless common principles apply; some are in the European Union's **General Data Protection Regulation**
  - Protects EU citizens
    - Often understood as protecting all users
    - Alternative is to handle EU users differently: impractical

# GDPR Principles

- Personal data may not be processed unless a data subject (user) has provided informed consent
  - Data subjects can withdraw this consent at any time
- European Union Agency for Network and Information Security specifies what needs to be done to achieve privacy and data protection by default
- Must employ a *data protection officer*
  - Who manages compliance
- Violators may be fined up to €20 million or 4% of the annual worldwide turnover, whichever is greater

# GDPR Organisation Requirements

- Controller and processor
  - No personal data may be processed unless done under one of the six lawful bases:
    - If the data subject has given consent
    - To fulfil contractual obligations with a subject, or for tasks at the request of a subject who is entering into a contract
    - To comply with a data controller's legal obligations
    - To protect the vital interests of a subject or another individual
    - To perform a task in the public interest or in official authority
    - For the legitimate interests of a data controller or a third party, unless these interests are overridden by interests of the data subject or her or his rights according to the Charter of Fundamental Rights (especially in the case of children)

# GDPR Organisation Requirements

- Controller and processor
  - When data is collected, users must be clearly informed
    - The extent of data collection
    - The legal basis for processing of personal data
    - How long data is retained
    - If data is being transferred to a third-party
    - If the data is being transferred outside the EU
    - Any decision-making that is made on a solely algorithmic basis
  - Users must be informed of their rights under the GDPR
  - Users must be provided with contact details for the data controller and their designated *data protection officer*

# GDPR Organisation Requirements

- Pseudonymisation
  - Must transform personal data so that the result cannot be attributed to a specific user without the use of other data
    - E.g. encryption. The decryption key must be kept separately
- Records of processing activities
  - Electronic records must be maintained by the organisation
  - Made available to supervisory authority on request
  - Must contain all of the following information:
    - Details of the data controller
    - The purposes of the processing
    - Categories of data subjects and personal data
    - Categories of recipients
    - Time limits for erasure
    - Security measures

# GDPR Organisation Requirements

- Security of personal data
  - Must notify the supervisory authority within 72 hours after becoming aware of the data breach
  - Users must be notified if a high risk impact is likely
    - Not required if the data is unintelligible to any person who is not authorised to access it (i.e. encrypted)
- Data protection officer
  - Must appoint a data protection officer
    - With expert knowledge of data protection law and practices
      - Assist the controller in monitoring compliance
      - Can be a current member of staff or can be outsourced
      - The contact details must be published
      - Organisations outside the EU must appoint an EU-based representative for their GDPR obligations

# GDPR Organisation Requirements

- Remedies, liability and penalties
  - Violators may be fined up to €20 million or 4% of the annual worldwide turnover, whichever is greater

# GDPR User Rights

- Rights of the user
  - Transparency and modalities
    - Must provide information ‘in a concise, transparent, intelligible and easily accessible form, using clear and plain language’
  - Information and Access
    - Must provide an overview of the categories of data that are being processed as well as a copy of the data
    - Must provide details about the processing, including purpose, with whom the data is shared, and how the data was acquired
  - Rectification and erasure
    - Must delete personal data within 30 days of request (on a number of specified grounds)
  - Right to object and automated decisions
    - Can object to processing personal information for marketing, sales, or non-service related purposes

# GDPR User Rights

- Rights of the user
  - Revoke consent
    - Doing so must not be harder than it opting in
  - Can file complaints with a Data Protection Authority
    - National independent body examines complaints

# ACM Code Of Ethics

- General Ethical Principles
  - 1.1 Contribute to society and to human well-being, acknowledging that all people are stakeholders in computing
  - 1.2 Avoid harm
  - 1.3 Be honest and trustworthy
  - 1.4 Be fair and take action not to discriminate
  - 1.5 Respect the work required to produce new ideas, inventions, creative works, and computing artifacts
  - 1.6 Respect privacy
  - 1.7 Honor confidentiality

# ACM Code Of Ethics

- Professional Responsibilities
  - 2.1 Strive to achieve high quality in both the processes and products of professional work
  - 2.2 Maintain high standards of professional competence, conduct, and ethical practice
  - 2.3 Know and respect existing rules pertaining to professional work
  - 2.4 Accept and provide appropriate professional review
  - 2.5 Give comprehensive and thorough evaluations of computer systems and their impacts, including analysis of possible risks
  - 2.6 Perform work only in areas of competence
  - 2.7 Foster public awareness and understanding of computing, related technologies, and their consequences
  - 2.8 Access computing and communication resources only when authorized or when compelled by the public good
  - 2.9 Design and implement systems that are robustly and usably secure

# ACM Code Of Ethics

- Professional leadership principles
  - 3.1 Ensure that the public good is the central concern during all professional computing work
  - 3.2 Articulate, encourage acceptance of, and evaluate fulfillment of social responsibilities by members of the organization or group
  - 3.3 Manage personnel and resources to enhance the quality of working life
  - 3.4 Articulate, apply, and support policies and processes that reflect the principles of the Code
  - 3.5 Create opportunities for members of the organization or group to grow as professionals
  - 3.6 Use care when modifying or retiring systems
  - 3.7 Recognize and take special care of systems that become integrated into the infrastructure of society

# ACM Code Of Ethics

- Compliance with the code
  - 4.1 Uphold, promote, and respect the principles of the Code
  - 4.2 Treat violations of the Code as inconsistent with membership in the ACM

# How?

- How are *you* going to guarantee the safety of *your* code?
  - **Technical** mechanisms
    - Testing, static analysis, etc.
  - **Organisational** mechanisms
    - Code review, version control (audit trail), etc.
- Be open about what you cannot guarantee
  - So managers can plan
  - Hiding problems may lead to higher net cost
  - Openness doesn't guarantee problems get fixed
  - At worst, be a whistle-blower

# Engineer It In Early

- Bad feeling about the software?
  - Raise it and record it!
  - It reduces your potential liability
- Ensure there's a plan to handle unexpected problems
  - Risks (see previous lectures)
  - Test the plan
    - Is it workable?
- Retrofitting security tends not to work effectively
  - Ensure you're satisfied with the way designs work
  - Ensure that temporary hacks / workarounds get reviewed

# 2023 Exam Question

The recent BBC News article “On the warpath: AI’s role in the defence industry” (August 2023) starts:

**Alexander Kmentt doesn’t pull his punches: “Humanity is about to cross a threshold of absolutely critical importance,” he warns.**

The disarmament director of the Austrian Foreign Ministry is talking about autonomous weapons systems (AWS). The technology is developing much faster than the regulations governing it, he says. “This window [to regulate] is closing fast.”

A dizzying array of AI-assisted tools is under development or already in use in the defence sector.

Discuss the moral and / or ethical responsibilities of the developers of such technology, include references to the ACM Code of Ethics.

# 2022 Exam Question

AI tools such as OpenAI, and GitHub CoPilot are remarkably good at writing routines and programs in many different languages. This is because they have been trained on many millions of lines of code, code that is assumed to be correct and working. But these tools can, and do, write code that has bugs.

A piece of code from a tool such as GitHub CoPilot might end up in a life-critical application such as a self-driving car, an airplane, or a pacemaker. And it might go wrong resulting in a human death. If this happens then who should be held accountable for that death? There are several possible candidates, including the author of the code used in the AI training, the person who put the AI training dataset together, the person who did the training, the person who asked the AI for the code, and the person who integrated that code into the overall application. ***Choose one of these people and discuss their moral and / or ethical responsibilities, include reference to the ACM Code of Ethics.***

# 2021 Exam Question

A BBC News article published on 19 August 2021 (*Twitter tests ‘misleading’ post report button for first time*), states that “Many of the large social media networks have been accused of not doing enough to fight the spread of disinformation during the Covid pandemic and US election campaigns”. Is it the responsibility of a social media platform to censor the utterances of their users? If so, then is it the responsibility of Microsoft to censor the writings in Office 365 and stored in the cloud? If not, then should all content be legal? ***To what extent should social media platforms censor the utterances of their users? Discuss how your answer aligns with the ACM Code of Ethics.***

# 2020 Exam Question

On 29th August 2020 the media reported on a chip implanted into the brain of a pig. The chip sent Bluetooth signals to a computer when it detected neural activity in the pig's snout as a consequence of the pig looking for food. Neuralink's founder, Elon Musk, claimed the initial use in humans will be to help paraplegic and tetraplegic patients, "If you can sense what people want to do with their limbs, you can do a second implant where the spinal injury occurred and create a neural shunt".

However Musk also hypothesized other uses including "conceptual telepathy" where people can communicate device to device without writing or speaking, their thoughts being transferred directly. He even claimed that "In the future you will be able to save and replay memories".

Neuralink is clearly sitting on the boundary between what today is ethical and what is not. Restoring limb control to paraplegics could have a profound and positive effect on the lives of may thousands of people. Replaying memories could be the entertainment medium of the future. However, controlling someone else's limbs, or forcibly uploading someone else's memories, or telepathically pushing advertising directly into someone's brain might have a negative and profound effect on someone's life.

Choose one side of the debate: Either for or against Neuralink continuing this work. If you are against, then discuss why it should stop. If you are for, then discuss how to know when the work becomes unethical.

# 2019 Exam Question

Social media companies such as Facebook are not only collecting information on which web sites you visit, but also where (physically) you go by tracking you via the GPS on your phone. Moreover, they are collecting labeled images of individuals from which they can deduce other biometric data such as height, eye colour, and so on. Such information is incredibly valuable for advertising (the reason it is being collected), and it is valuable for law enforcement.

Amazon was recently in the news because of an agreement with “at least 200 law enforcement agencies to carry out surveillance via its Ring doorbells” [BBC News, 1/8/2019]. These doorbells were sold as devices that send live video to a customer’s computer (or phone).

Your mobile phone service provider knows the location of your phone whenever it is turned on and attached to the mobile network. Consequently, they know where you are when you have your phone with you.

Increasingly every action we perform online and offline is tracked. Some people believe this is for the betterment of society, while others believe it is creating a surveillance state which will create a worse society. Choose one side of this debate and discuss the software engineer’s ethical responsibility in building products that track users and their behaviour.

# 2018 Exam Question

The news is filled with stories of deepfakes and fake news, but there is a genre of entertainment known as mockumentary, and sites like The Onion specialise in satire. Discuss three things: The dividing line between ethical fakes and unethical fakes; the role of the software engineer in this; and your responsibility as a software engineer.

# 2017 Exam Question

With the automation of just about everything, there is a real possibility that paid employment (as we know it today) will no longer exist in the near future as all jobs will be automated. Some have suggested a Universal Basic Income (UBI) as a solution — everyone gets paid enough to survive regardless of whether there is work for them to do or not. Experiments are underway in some countries (such as Finland), and others will experiment soon (such as Canada). Discuss:

- (a) How the technology industry is (or is not) responsible for this problem (you can discuss one side, the other side, or both sides).
- (b) The ethical obligation the technology industry has to either “fix” or “prevent” mass unemployment (you can discuss one side, the other side, or both sides).
- (c) How you would contribute to the technology industry if you did not need to worry about the cost of living.
- (d) The potential downside (or upside) to the technology industry of a Universal Basic Income (you can discuss one side, the other side, or both sides).

# 2016 Exam Question

Google is well known to be developing a driverless car. Apple is suspected to be doing likewise. Uber is expected to have driverless cars on the streets before the end of 2016. Explain some of the ethical issues (and how you would resolve them) that the software development team must face when developing driverless cars.

# References

- Much of this lecture comes directly from Wikipedia and the ACM
  - General Data Protection Regulation
    - [https://en.wikipedia.org/wiki/General\\_Data\\_Protection\\_Regulation](https://en.wikipedia.org/wiki/General_Data_Protection_Regulation)
  - ACM Code of Ethics and Professional Conduct
    - <https://www.acm.org/code-of-ethics>
- University of Otago Past Exam Papers
  - <https://www.otago.ac.nz/library/exams/>