Structured Programming

Dr Ian Reid
4 lectures, Michaelmas Term

http://www.robots.ox.ac.uk/~ian/Teaching/B16
B16 Software Engineering

- 4 courses
  - Computer Communications and Networking
    - How devices and computers communicate
    - The internet
  - Structured Design and Programming
    - Basic Principles of Software Engineering
    - Writing structured code in a procedural language
  - Object Oriented Programming
    - Understanding the key principles of object-oriented design and programming
  - Operating Systems
    - The glue between the programmer and controlling the real-world

It is hard to underestimate the role that computers play in modern life. In your Engineering lives after graduation it is almost inconceivable that you will not have to use computers. But the aim of this course is to go beyond use, and to develop understanding of how to build software and how that software will fit in with the “bigger picture”. This might be for an Engineering firm, maybe for a bank (boo), or it might be part of a research project such as those we offer in Information Engineering.

To that end this course tries to equip you with knowledge of how to go about designing good software (course 2 and 3 above), and how that software might fit in with the real world: the key bits we cover here are (i) a computer’s operating system (course 4), the glue that links your programs with the underlying hardware (which may be your basic PC stuff, or more excitingly, it might be sensors and actuators; and (ii) communications between computers (course 1), especially that thing that has revolutionised our lives in the last 20 years, the internet.
B16 Software Engineering

Meta learning outcomes:

• A clear understanding of the importance of good design practice in software, and the role that structured and object oriented programming ideas play in this.

• An understanding of how to develop applications that can interact with the outside world (i.e., programs as part of bigger engineering systems) via the operating system and via inter-computer and computer-device communications.

• Have an understanding of how ideas in 4th year options and projects can actually be implemented in software.
Software Engineering vs structured programming

Not really a course about software engineering...

1. Software engineering
   - Mostly about concepts/principles of design, modularity, abstraction, encapsulation, etc
2. Structured programming
   - Revision, coding in C and Matlab, control flow, variables, data types
   - The mapping from high-level language to architecture (including memory)
3. Functions
   - Code re-use, parameters, libraries
4. Data structures
   - Structures/classes, arrays
5. Algorithms
6. Recursion

David Parnas ("Structured programming: a minor part of software engineering", Information Processing Letters, 88(1-2), 2003), one of the leading researchers on software engineering in the world has argued that “Software Engineering is ... a branch of Engineering specialising in software intensive products. ........it is argued that writing programs, though obviously an essential step in the process, is only a small part of Software Engineering.” That said, it is this essential step that this course is primarily looking at.

I assume you have some skill in writing computer programs in Matlab. You might need to brush up on this. If you are lucky you will have seen C or other programming languages (eg Python) in your extra-curricular life, but this is not essential. The overall aim of this course is to take you beyond the narrow syntactical knowledge you acquired in the Matlab lab. To do this we will look in quite close detail at some things you have already seen, like functions, because a crystal clear understanding of these will greatly help in understanding how and why various programming practices are “good” and some (many!) are “bad”. I will try to use a mixture of both Matlab and C/C++ to illustrate the general principles.
Learning Outcomes

• The course will aim to give a good understanding of basic software design methods, and emphasize the need to produce well-structured maintainable computer software. The course will concentrate on principles, but these will be reinforced with examples in Matlab and C/C++ programming languages. Specifically, by the end of this course students should:
  • understand concepts of basic program design techniques that can be applied to a variety of programming languages
  • understand the need for structured programming in software projects
  • have a good understanding of the mechanics of function calls and of recursion
  • Have a good understanding of the role, uses and advantages of compound data structures
  • be able to recognise and to produce and/or maintain well structured programs
Texts

- Goodrich et al., *Data structures and algorithms in C++*, Wiley, 2004

The first is a comprehensive “classic” text book on software engineering, now in its 8th edition.

Wirth is a classic. Some aspects are outdated, but it remains very sound.

Leveson is very interesting and contains valuable lessons, but is really for peripheral reading.

Lipmann and Lajoie, and Goodrich et al, are C++ text books. The C++ bible is by Stroustrup.
The Role of Computing in Engineering

• Computing is ubiquitous in engineering. Why?
• Awesome speed of modern, everyday computers makes complicated analysis and simulation possible across all domains.
• Applications in design and modelling. Far beyond the reach of the mortal human engineer. Indeed many modelling problems are utterly infeasible without modern computers and software.
• In embedded systems, computers can provide a level of power, speed, flexibility and control not otherwise possible (e.g., mobile phone).
• Computing is “cheap” (but exercise this argument with care).
• Software is the key…

Computing is cheap: well, we have to be careful here. Leveson in her seminal book on computers and safety (see course texts) debunks this “myth”, along with several others. Computer hardware is cheap relative to other electromechanical devices; but the cost of writing and certifying reliable and safe software may be enormous.
Example: mobile phone

- Even simple mobile phones rely on software
- Typical phone has a microcontroller (SIM card) with a small program
  - Drive GUI
  - Control devices (keypad, microphone, a/d, dsp, decoder)
Example: Sizewell B

- Nuclear power station (PWR), on-stream in 1995
- Software used extensively in the design
- Software for control!
  - first UK reactor to use software in its Primary Protection System)

Sizewell B was Britain’s first (only?) Pressurised Water Reactor. It is also notable because the Primary Protection System uses software extensively. This software is thus an (extreme!) example of so-called safety-critical software; i.e. unexpected operation could lead to disaster. Over 100,000 lines of code were validated in a process that was controversial, hugely time-consuming and at greatly expensive.
Example: A380

- A380
- 1400 separate programs
- There is a software project just to manage all the software!
- Clearly safety-critical features of the software

The A380 incorporates 1400 separate computer programs. These govern everything from air-conditioning and lighting to engine management and flight-surface control. Indeed there are so many components to the software that there is a separate software project just to manage all the software!

CAD software, of course, has played a major role in the design, but has in fact been at the root of major delays of as much as two years I production. Two of the partners (French and German) used different versions of the CAD software which were unable to interface to one-another. Sigh!

Like its predecessors (the first was the A320), the A380 is a fly-by-wire aircraft. This means that some of the software is safety-critical...

Like its predecessors (the first f-b-w commercial airliner was the A320) it uses fbw for weight savings and also implements a “Flight Envelope Protection System” (i.e. the computer will overrule attempts to take the aircraft outside its safe flying limits).

The A320 used 7 computers running software from two separate vendors (the aim being to minimise the possibility that the same bug would appear simultaneously.

See comp.risks archives for extensive discussion of the A380, and if you go back to the archives from late 1980s you will see a lot of discussion about several A320 incidents; more on this in a bit.
Example: NPfIT

- NHS National Plan for IT
- Plan to provide electronic care records for patients
- Connect 30000 GPs and 300 hospitals
- Provide secure access to records for healthcare professionals
- Provide access for patients to their own records via "Healthspace"

NPfIT is one of the biggest, if not the biggest ever, civil information technology programmes. Sounds like a great plan. Or does it? See later for a little discussion about this project.
Software engineering versus programming

• Software engineering is about more than just programming/coding
• It is about design principles and methodologies that yield programs that are
  • Robust
  • Manageable
  • Reusable

Programmers of course want to produce code that is correct; i.e. it produces the right output for all expected inputs. But beyond this software should ideally be robust, in that it is capable of gracefully handling inputs not explicitly defined for its application.

Code that is well designed and implemented will facilitate current and future maintenance, make debugging more easy, and potentially expedite future changes.

Well designed software, which has clearly defined interfaces to other code, can be reused to improve correctness, save time, etc. For example, you would not reimplement Matlab’s trig functions since these are already implemented in an efficient manner in the Matlab function library, and have a well defined interface (eg put in an angle in radians and get out sine of the angle). Re-inventing the wheel here introduces unnecessary complexity and the possibility of errors. A well-encapsulated module can be re-used.
Software vs “other” engineering

- How is software engineering similar to other engineering?
- Abstraction and Modularity
  - Consider free-body diagram
  - Thevenin/Norton
  - Low output impedance / High input impedance
  - Digital computer
  - We return to these concepts later…

In many respects software engineering is no different from other forms of engineering. In designing a bridge, or a building, or a machine, or an electronic circuit, or a control system, the engineer uses a variety of techniques to make the analysis/design of an apparently complex system more tractable. Two key ideas that transcends all engineering disciplines are those of abstraction and modularity. In statics and dynamics the engineer will consider the behaviour of a component by use of a free-body diagram in which the role of neighbouring/connected components is abstracted as a set of forces. The component can then be considered as a module. In control theory (for that matter any dynamic modelling) we use transfer functions to model dynamic behaviour and a block diagram is an abstract representation of a set of modules dynamic modules. In linear, passive component circuit analysis Norton’s and Thevenin’s theorems enable a complex circuit to be abstracted as a voltage and/or current source and a resistance. And when dealing with transistors and op-amps, one tried to ensure that parts of a circuit have high input impedance and low output impedance so that they can be considered as separate modules with simple interactions (eg output voltage of circuit 1 equals input voltage of circuit two). Of course this reaches an extreme in the design of a digital computer (see A2) which lives in (and is built with) an analogue world, but which is built using logic gates, an abstraction of underlying analogue electronics. In P2/A2 you saw how the transistors/op-amps could be combined to yield flip-flops and and/or (etc) gates. Then how these could be combined into registers. And how registers (a higher level abstraction) could be combined to form a control unit and a data unit, and ultimately to yield a micro-processor.
Abstraction: free-body diagram

Block A
- 49.05 N
- T
- Fr
- N

Block B
- 19.62 N
- T

A
- 5 kg

B
- 2 kg
Modularity: Op-amp buffer

- Unity gain buffer
- $V_{out} = V_{in}$
- Very high input impedance, very low output impedance
Software vs “other” engineering

• How is software different to other engineering?
• Pure, weightless, flexible
• Capacity to incorporate massive complexity
• No manufacturing defects, corrosion, aging

Use of software is attractive for these reasons.

Consider Airbus A320, huge weight savings in fly-by-wire design. Wires carrying control signals will not fatigue like cables in traditional design.

The most complex engineering systems in existence today are almost either software, or heavily reliant on software.

A computer's behaviour can be easily changed by modifying its software. In principle, flexibility is a good thing, since major changes can be effected quickly and at apparently low cost.

Software does not age in the sense that it will always do the same thing, so it will not fatigue/fail like other engineering systems (distinguish this from the hardware on which the software runs which is of course built from digital circuits).

Software cannot be made more reliable by having multiple copies of it. Each copy will behave identically and have exactly the same failure modes.

Furthermore, it has been observed that software teams writing code independently to perform the same tasks will often make the same mistakes!
Intrinsic difficulties with software

- Analogue versus discrete state systems
- The “curse” of flexibility
  - Can encourage unnecessary complexity
  - Redefinition of tasks late in development – shifting goal-post
- Complexity and invisible interfaces
  - Standard way of dealing with complexity is via modularity
  - But this alone is not enough because interfaces can be subtle and invisible, and here too there is a need to control complexity
- Historical usage information
  - Unlike physical systems, there is a limited amount of experience about standard designs

Discrete behaviour means we cannot interpolate. A bridge can be tested at extreme loads and we can assume it will work for lesser loads, but with software, every input is a separate, distinct situation. Further, the failure behaviour need not be related in any way to the normal behaviour.

A computer's behaviour can be easily changed by modifying its software. Though potentially advantageous (see previous slide), ease of change can encourage unnecessary complexity and introduce errors, because it blurs the distinction between what can be achieved versus what should be achieved. It can encourage a shifting of goalposts in specifications.

As mentioned earlier, a standard way in all engineering disciplines is to attach complexity via modularity. In physical systems, the physical separation of functions can provide a useful guide for decomposition into modules: the spatial separation of these modules limits their interactions. Makes their interactions easy to trace and makes introducing new interactions difficult. With software there are no such barriers: complex interfaces are as “easy” to produce as simple ones, and the interfaces/interactions between software components can be invisible or not obvious.

Historical data: Consider aeroplane, or bridge design. The starting point for such designs is usually based on sound historical designs, and data collected about such designs over a long period [c.f. Millennium bridge, a new design of bridge that met with unexpected problems]. With much software it is specially constructed and therefore there is very little in the way of historical data to go on.
When software projects go wrong

- A320, Habsheim and Strasbourg

Air France Flight 296 was a chartered flight of a newly-delivered fly-by-wire Airbus A320 operated by Air France. On June 26, 1988, as part of an air show it was scheduled to fly over Mulhouse-Habsheim Airport at a low speed with landing gear down at an altitude of 100 feet, but instead slowly descended to 30 feet before crashing into the tops of trees beyond the runway. Three passengers were killed. The cause of the accident is disputed, as many irregularities were later revealed by the accident investigation. This was the first ever crash involving an Airbus A320.

On 20th Jan 1992 another incident involving an A320 in which the aircraft descended much too rapidly without the flight crew noticing until it was too late. The official report states that from the moment of descent until impact, the aircraft's high descent rate was adopted and maintained. The official conclusion on the cause of the accident was "pilot error". The "pilot error" in this case was the confusion of the "flight-path angle" (FPA) and "vertical speed" (V/S) modes of descent. These were selectable on the Flight Management and Guidance System (FMGS) console. The pilots were inadvertently in V/S when they should have been in FPA mode. The error was not noticed on the console itself, due to the similarity of the number format display in the two modes. The other cues on the Primary Flight Display (PFD) screen and elsewhere (e.g., altitude and vertical speed indicator) were not noticed since the pilots were overloaded following a last-minute change of flight plan, and presumably were concentrating on the Navigational Display.

The report criticised the ergonomic design of the descent-mode selector for being too easy to misread or mis-set.

Officially neither of these accidents was caused by software. However the additional complexity introduced – which also of course has advantages – in these cases contributed to the accidents.
When software projects go wrong

• London Ambulance Service
  • 1992, computerised ambulance despatch system fails
• Therac-25
  • 2 people died and several others exposed to dangerous levels of radiation because of software flaws in radiotherapy device
• OSIRIS
  • £5M University financial package
  • Expenditure to date more like £20-25M
• NPfIT?
  • NHS £12 billion IT project
• comp.risks is a great source of others...

LAS: project to computerise ambulance despatch. To be completed in 6 months. A consortium of Apricot, Systems Options and Datatrak made the cheapest bid and was awarded the contract. Because of time pressures, and the lack of experience of the development team in dealing with safety-critical systems, fundamental flaws in system design, and inadequate consultation with users, the system went “live” even though there were 81 known errors. It ran for a day and a half before being shut down. A further 10-day trial was abandoned and the LAS reverted to manual operation. From the Independent, 30 Oct 1992, “Computer specialists yesterday said that the system blamed for this week’s crisis at the London Ambulance Service appeared to ignore basic tenets for software where breakdown would put lives at risk. The failure of the computer system over 36 hours on Monday and Tuesday, which was said to have cost between 10 and 20 lives, raised serious questions about the way it was designed and tested, experts said. Yesterday, the software company involved, Systems Options, refused to comment.”

Therac-25: See Leveson, especially Appendix A

OSIRIS: Classic failures of trying to build a system to serve two different needs and neglecting one. Commissioned in 01/02. In Nov 2003 departmental administrators highlight 10 points critical to operation not adequately addressed, in particular the ability to generate reports central to management of finances. System goes live in April 2004 with major omissions/flaws and University lucky to escape without serious financial meltdown.
NHS National programme for IT: NPfIT

- Plan to provide electronic care records for patients
- Connect 30000 GPs and 300 hospitals
- Provide secure access to records for healthcare professionals
- Provide access for patients to their own records via “Healthspace”

- Laudable?
- Realistic?
  - Software Engineering specialists have their doubts
  - Ross Anderson (Prof of Security Engineering, Cambridge Computing Laboratory) writes in his blog “I fear the whole project will just continue on its slow slide towards becoming the biggest IT disaster ever”.

NPfIT is one of the biggest, if not the biggest ever, civil information technology programmes. Original budget of £2.3 billion over 3 years, by June 2006 this was estimated to be £12 billion over 10 years. In April and June 2006 in an Open Letters, 23 leading professors of Computer Science wrote to the Health Select Committee calling for a public enquiry:

“As computer scientists, engineers and informaticians, we question the wisdom of continuing NPfIT without an independent assessment of its basic technical viability. We suggest an assessment should ask challenging questions and issue concrete recommendations where appropriate, e.g.:

- Does NPfIT have a comprehensive, robust:
  - Technical architecture? - Project plan? - Detailed design?

Have these documents been reviewed by experts of calibre appropriate to the scope of NPfIT?

- Are the architecture and components of NPfIT likely to:
  - Meet the current and future needs of stakeholders? - Support the need for continuous (i.e., 24/7) healthcare IT support and fully address patient safety and organisational continuity issues? - Conform to guidance from the Information Commissioner in respect to patient confidentiality and the Data Protection Act?

- Have realistic assessments been carried out about the:
  - Volumes of data and traffic that a fully functioning NPfIT will have to support across the 1000s of healthcare organisations in England?
  - Need for responsiveness, reliability, resilience and recovery under routine and full system load?

In April 2007, the Public Accounts Committee of the House of Commons issued a 175-page damning report on the programme. The Committee chairman, Edward Leigh, claimed “This is the biggest IT project in the world and it is turning into the biggest disaster.” The report concluded that, despite a probable expenditure of 20 billion pounds “at the present rate of progress it is unlikely that significant clinical benefits will be delivered by the end of the contract period.”
Software life-cycle

- Software development stages
  - Specification
  - Design
  - Implementation
  - Integration
  - Validation
  - Operation/Maintenance/Evolution
- Different types of system organise these generic activities in different ways
- *Waterfall approach* treats them as distinct stages to be signed off chronologically
- In practice usually an iteration of various steps

*Specification*: engineers and customers collaborate to specify objectives (what should it do?) and constraints. This process would typically be iterative, refining vague ambiguous objectives into more formal, tightly specified ones that clarify assumptions.

*System/Software Design*: the requirements specification is then mapped onto an overall architecture of the system (both hardware and software, probably including choices of programming language). Software design involves identifying and describing the fundamental software abstractions and their relationships (i.e. identifying individual software components and designing their interfaces).

*Implementation*: software design is realised as a set of units or programs or modules. Unit/modules are tested to ensure each meets its specification.

*Integration and Validation*: components brought together and tested rigorously to ensure conformity to user-requirements and correctness.

*Operation/Maintenance/Evolution*: system installed, modifications to adapt to changing user-requirements; maintenance involves correcting errors that were not discovered earlier, improving the implementation and enhancing the services as new requirements are discovered.

Because software is invisible, signoff will often involve production of copious documentation.
Requirements

- Vague initial goals
- Iterative refinement
- Leading to more precise specification
- Example
  - Calculate the n-bounce trajectory of a lossy bouncing ball.
  - Refine this to consider
    - What does the statement \textit{actually} mean?
    - Physics
    - Initial conditions
    - Air-resistance?
    - Stopping criterion (criteria)?
  - Now, think about how to design/implement

What does the statement n-bounce trajectory mean? We could either assume that this means “given a set of initial conditions, work out n”, or it could mean, “given n, work out what the initial conditions would need to be”. This may inform how we set about the task, or how we represent the trajectory (or if we even represent it at all).
Validation/Verification

- Verification: does the system confirm to spec?
- Validation: does it actually do what it was supposed to?
- Top-down vs bottom-up testing
- Black-box vs white-box testing
- Impossibility of exhaustive testing

Verification checks the system against its specification, while validation checks that it actually meets the real needs; the two may be different if there were problems moving from the vague mission statement to the formal requirements in the design process.

Testing at the module/unit level should happen concurrently with implementation so that errors are found as quickly as possible.

Top down testing involves coding top-level components first and substituting trivial “stubs” for the lower-level functionality that has not yet been implemented (e.g. writing a simple function that returns a number corrupted by a random element to simulate a sensor module). Bottom up testing involves building the lowest level components first and testing them using a “test harness”.

Black-box testing checks a component by examining its outputs for a set of controlled inputs to check they match the specification. White-box, on the other hand, examines the internal structure of a module exercising every line of code and every decision point. It can be useful for checking limiting conditions which are a common source of error and may be missed in black-box testing.

As mentioned previously, interpolation is not possible in the discrete world of computer hardware and software, but exhaustive testing is impossible for all but the simplest (trivial) systems.
Extreme programming (XP)

- Proposed in the late 90s as a reaction to problems with “traditional” development processes
- Takes extreme position compared with waterfall approach
- Appropriate for small-medium sized projects
  - Teams of pairs of programmer, programming together
  - Incremental development, frequent system releases
  - Code constantly refined, improved, made as simple as possible
  - Do not design for change; instead change reactively

Incremental development supported through small, frequent releases of code; requirements specified via a set of customer scenarios forming the basis for planning;

Customer involvement in the development cycle, defining acceptance tasks, etc

Pairs of programmers work very closely together, collective ownership of code

Change supported through regular releases, and continuous integration

Simplicity maintained by use of simple designs that deliberately do not anticipate future requirements. Instead constant refactoring to improve code quality.

Extreme programming takes the position that most design for change is wasted effort. Instead, code is re-factored: the programming team is constantly looking for improvements and implements them immediately; the software is then ideally easy to understand, facilitating change as/when necessary.

Top down design

- Here want to keep in mind the general principles
  - Abstraction
  - Modularity

- Architectural design: identifying the building blocks
- Abstract specification: describe the data/functions and their constraints
- Interfaces: define how the modules fit together
- Component design: recursively design each block

Top down design means breaking the problem down into components (modules) recursively. Each module should comprise related data and functions, and the designer needs to specify how these components interact – what their dependencies are, and what the interfaces between them are. Minimising dependencies, and making interfaces as simple as possible are both desirable to facilitate modularity.

By minimising the ways in which modules can interact, we greatly limit the overall complexity, and hence limit unexpected behaviour, increasing robustness.

Because a particular module interacts with other modules in a carefully defined manner, it becomes easier to test/validate, and can become a reusable component.

Note the reference to the general engineering principles of abstraction and modularity. A word about each in its software context:

**Abstraction:** the idea here is to distil the software down to its fundamental parts, and describe these parts precisely, but without cluttering the description with unnecessary details such as exactly how it is implemented. The abstraction specifies what operations a module is for, without specifying how the operations are performed. A function header with a sensible name is an example of such an abstraction. The Matlab function header:

```
function y = sin(x)
```

tells us almost enough about this function to use it without ever needing to know how it is implemented (of course it doesn’t quite, since this specification does not expose the units of the angle x).

**Modularity:** the aim is to define a set of modules each of which transcribes, or encapsulates particular functionality, and which interacts with other modules in well defined ways. We will explore this in the context of functions when we look at side-effects and global variables. The ore complicated the set of possible interactions between modules, the harder it will be to understand. The bottom line is that humans are only capable of understanding and managing a certain degree of complexity, and it is quite easy (but bad practice) to write software that exceeds this capability.
Modular design

- Procedural programming: tend to focus on algorithms
- Object-oriented programming: tend to focus on data structures

In 1975 Niklaus Wirth, famous Swiss computer scientist (investor of Pascal, Modula 2), wrote a seminal text called *Algorithms + Data Structures = Programs*.

While the top-down design methodology is a general tool, how we approach it will potentially be language dependent.

In procedural programming the design effort tends to concentrate on the functional requirements and recursively subdivides the functionality into procedures/functions/subroutines until each is a simple, easily understood entity.

Examples of procedural languages are C, Pascal, Fortran, Matlab

In object-oriented programming the design emphasis shifts to the data structures and to the interfaces between objects.

Examples of object-oriented languages are C++ and Java.

This distinction is not hard and fast, and it is possible and beneficial to think about data structures as part of the fundamental design effort even when using procedural languages.
Structured programming

- Top-down vs bottom-up
- Both are useful as a means to understand the relations between high-level and low-level views of a program
- Top-down
  - Code high level parts using “stubs” with assumed functionality for low-level dependencies
  - Iteratively descend to lower-level modules
- Bottom-up
  - Code and test each low-level component
  - Need “test harness” so that low-level can be tested in its correct context
  - Integrate components
- Not hard-fast rules; combination often best

Donald Knuth interview, 7/12/1993:

“You can create the parts in whatever order is psychologically best for you. Sometimes you can create them from the bottom up. Bottom-up means that you know somehow that you probably need a subroutine that will do something, so you write it now while you're ready, while you're psyched for it. With this bottom-up programming, your pencil gets more powerful every page, because on page nine you've developed more tools that you can use on page ten... your pencil is stronger.

With top-down programming you start at the beginning and say "I'm going to do this first and then this, and then this"... but then you have to spell out what those are--- you can wind up gasping for breath a hundred pages later when you finally figure out how you're actually going to do those things!

Top-down programming tends to look very nice for the first few pages and then it becomes a little hard to keep the threads going. Bottom-up programming also tends to look nice for a while, your pencil is more powerful, but that means you can also do more tricky stuff. If you mix the two in a good psychological way, then it works, even at the end.

I did this with TeX, a very large program: 500+ pages of code in the book. Throughout that entire program, all those lines of code, there was always one thing that had to be the next thing I did. I didn't really have much choice; each step was based on what I'd done so far. No methodology would teach me how to write a piece of software like that, if I followed it rigorously. But if I imagined myself explaining the program to a good competent programmer, all that this long program was, then there was just this one natural way to do it. The order in which the code appears in the book is the order in which I wrote it.”
Simple design tools

- Flow chart

- Pseudo-code
  - Wait for alarm
  - Count = 1
  - While (not ready to get up and count <= 3)
    - Hit snooze button
    - Increment count
  - Climb out of bed

Traditional flow charts are a rather cumbersome and outmoded way of designing code. There is a strong emphasis on the sequences of operations and decisions that is probably better represented using pseudo code [picture from EDraw website]

Pseudo-code is representation of program structure that is an intermediate between natural language and proper programming language. The structure of the program is expressed by the formatting (note indentation, etc).
Data flows, on the other hand, can be very useful in helping to abstracting the system architecture, and begin to define interfaces. One notation uses rounded rectangles to represent functional processing, rectangles to represent data stored, and arrows between boxes to represent data movement.

Sommerville p175
Simple design tools

• State diagram

State machines represent how a system responds to internal and external stimuli, by showing how the internal state of the system changes in response to events. This representation should be familiar from P2/A2 Digital Logic and Computer Architecture courses.

Unlike the data-flow model, the state machine representation does not show the flow of data within the system. State machines are often useful abstractions when dealing with real-time systems, because these are often driven by external events.

In the example the diagram (taken from Sommerville p176) shows how the states of a simple microwave oven vary.
Lecture 2: Structured Programming

In which we consider basic programming constructions…
Basic coding techniques: flow control

• Pretty much any program can be specified using:
  • Sequences of instructions
    • { Do A; Do B; Do C }
  • Conditional instructions
    • If (condition) Do A
  • Repetitions (loops)
    • While (condition) Do A

• These semantic concepts are implemented in different high-level programming languages using different syntax

Syntax and semantics are terms applied here to programming languages but they derive from concepts that apply to language in general.

Syntax refers to the ways symbols may be combined to create well-formed programs (or sentences) in the language. Syntax deals solely with the form and structure of symbols in a language without any consideration given to their meaning.

Semantics reveals the meaning of syntactically valid strings in a language. For natural languages, this means correlating sentences and phrases with the objects, thoughts, and feelings of our experiences. For programming languages, semantics describes the behaviour that a computer follows when executing a program in the language. We might disclose this behaviour by describing the relationship between the input and output of a program or by a step-by-step explanation of how a program will execute on a real or an abstract machine.

(definitions from Slonneger and Kurtz, http://www.cs.uiowa.edu/~slonnegr/plf)
Basic coding techniques: variables

- Pretty much any program that does something useful requires the use of variables:
  - A symbolic name associated with a data storage location and its contents in memory
  - Value (contents) generally changes during the course of program execution.

- Assignment
  \[ \text{<Variable>} = \text{<Expression>} \]

- Declaration
  \[ \text{<Type>} \text{<Variable>;} \]

- Again, the idea of a variable is a semantic concept which is implemented in differently in different high-level programming languages.

A variable is a label for some of the computer’s memory storage.

The type of the variable tells the program how to interpret the values in the memory.

A declaration is a statement in the program that causes the program to allocate space in memory to store the variables value, and also tells the program explicitly what type of values/data will be stored. Since previously you have mostly had exposure only to matlab, this may be a new concept. Matlab has only one atomic data type, the floating point number, and does an implicit allocation of memory whenever you use a new variable name, so there is no need for explicit declaration. Many other languages (eg C/C++) require you to declare variables before use.
It is useful to think about what a variable *really* is (or actually how it maps to the architecture of a typical computer). It represents as set of memory locations, each of which stores a byte of information.

How these values are interpreted depends on the *type* of the variable.

In this instance the variable *myvar* occupies 4 bytes of the computer's memory.

If *myvar* is interpreted as a character string, each value is an ASCII code for a letter.

If *myvar* is interpreted as an interger it represents the value 66726564hex or 1,718,773,092 decimal

If *myvar* is interpreted as a 4-byte IEEE floating point number then it has the value 1.68302 x 10^22
implementation in Matlab and C

\[ N = 10; \]
\[ tot = 0; \]
\[ totsq = 0; \]
\[ for i=1:N \]
\[ \quad tot = tot+i; \]
\[ \quad totsq = totsq+i^2; \]
\[ \text{end} \]
\[ \text{cout} \ll tot \ll \text{endl}; \]
\[ \text{cout} \ll totsq \ll \text{endl}; \]

Of course a much more efficient way of doing this in Matlab would be to make use of the fact that the in-built data type is a matrix and that this could be achieved as:

\[ x = [1:N]; \]
\[ tot = \text{sum}(x); \]
\[ totsq = \text{sum}(x.*x); \]

Things to note are sequences of instructions. Note the different role played by the semicolon in the two languages. In C/C++ the ; denotes sequential statements. In contrast in Matlab the sequence is implicit from the different lines and the semicolon suppresses the output which would (implicitly) print the value of the variable to the console.

In Matlab a group of sequential statements is bracketed by an initial statement (in this case "for" and and "end" statement. In C/C++ a group of sequential statements is denoted using "{ }").

C/C++ are strongly typed languages. This means variables must be declared before use, and given an explicit type. In Matlab when a new variable name is encountered it is dynamically created. All variables in matlab are matrices of real numbers. In this case “tot” and “totsq” (and for that matter “i”) are simply 1x1 matrices (i.e. Scalars).

Note the unfamiliar "<<" notation used in the C code. This is actually C++ code and I have been very naughty; C++ implements i/o using "streams" and the streaming operators "<<" and ">>". "cout" is the standard output stream (the console) and so the last two lines should be interpreted as writing the value of tot followed by a new line to the console. Standard C-code would use

\[ \text{printf("%d\n", tot);} \]
\[ \text{printf("%d\n", totsq);} \]
Implementation in Matlab and C

N = 10;
ton = 0;
totsq = 0;
for i=1:N
    tot = tot+i;
totsq = totsq+i^2;
end
tot
totsq

int i, N=10;
int tot = 0;
int totsq = 0;
for (i=1; i<N; i++) {
tot += i;
totsq += i*i;
}
cout << tot << endl;
cout << totsq << endl;

Also of note here is the way the code has been laid out.

This may seem a trivial point, but it is remarkable how much more readable (and therefore debuggable or reusable) well laid out code is.

In particular note that the statements inside the loop are indented so that the structure of the code is clear even before looking at the detail of the code.
Notes on coding style

• Use meaningful variable names
• Use comments to supplement the meaning
• Indent code for each block/loop
• Encapsulate groups of statements sensibly in functions
• Encapsulate related data sensibly in data structures
• Design top down
• Code bottom-up or top-down, or a combination

All of these points are aimed at producing code which is clear and as simple as it needs to be. This in turn will aid with debugging, modification and maintenance of the code.

Encapsulation is an important concept and we will discuss this in more detail with regard to functions and to data structures later in the course and in the Object Oriented (OO) Programming course.
Matlab vs C

• Matlab and C are both procedural languages
• Matlab is an **interpreted language**
  • each statement decoded and executed in turn
• C is a **compiled language**
  • each module (.c file) is converted into assembly language
  • The interfaces between the modules are
    • Shared global data
    • Function calls from one module to another
  • This is resolved at **link** time when the modules are **linked**
    together into an **executable**

Note that Matlab does provide object-oriented support, but I do not discuss the details in this course. C++ is an object-oriented language. I will cover O.O. programming next term. Here I will occasionally use C++ but only the non-OO features which are almost identical to C.

Matlab is an interpreted language (or at any rate, it is best thought of as being interpreted). Each statement is decoded and executed in turn.

C is a compiled language
Procedural programming

• Aim is to break program down into functional units
  • procedures or functions
  • Set of inputs, set of outputs
• In Matlab and C this procedural building block is the *function*
• Understanding functions…
Organisation of Matlab programs

- A Matlab “program” may be a script or function
  - i.e. a sequence of instructions
- This script or function will typically call a bunch of other functions
- Functions are stored in .m files
- Multiple functions can be stored in one .m file, but only first is visible outside
  - The others are local functions
  - Part of the recursive subdivision of the problem

Confusingly in Matlab, (and in my view irritatingly) the name of the function in the function call is the filename, not the name that appears in the function header. Work that out! In general, therefore, it is sensible practice to have filename and function name identical.
Matlab file organisation

FUNC.m  foo.m  bar.m

FUNC
foo
bar
Each .c file contains the implementation details of one or more functions and global variables
Each is compiled independently of other code modules isolation to produce an object file (.o)
Functions

• Function definition
• Function call
• Function prototype
• Scope (local versus global data)
• Parameters and return value(s)
• Function call
• Low-level implementation of function calls
• Recursion
In Matlab this would live in a .m file. In C/C++ this would live in a .c or .cpp file.

The "//" is a C++ style comment. A C comment would be "/*" at the start and "*/" at the end.
Function call

- Distinguish between
  - The function definition
    - Defines the set of operations that will be executed when the function is called
    - The inputs
    - The outputs
  - And the function call
    - i.e. actually using the function
      - `fact(10)`
      - `a = 6;`
      - `z = fact(a);`

- Formal vs Actual parameters
  - `a = 6;`
  - `z = fact(a);`
  - `[V,D] = eig(A);`

- Return value(s)
  - The value of a function evaluation is the return value

The *formal parameters* are the parameters that appear in the function definition. They are placeholders for the values of the inputs; in this way they are analogous to the use of a variable in an algebraic expression. In the case of `fact` there is only one formal parameter, `n`.

The *actual parameters* are the values that get passed in at the time the function is called. In the examples above the actual parameters are 10, and 6 (since this is the value of “a” at the time `fact(a)` is called.

Carrying the analogy of algebraic expression forward, the function call is an evaluation of the expression at a particular value, given by the current value of the actual parameter(s).

C permits only one return value (can be a significant limitation!). This value is indicated in the definition using the “return ...” statement. This return value becomes the “value of the function”.

Matlab, in contrast, allows multiple return values. Matlab does not use the “return ...” notation. It is rather more elegant: the placeholders to be used for the outputs are indicated in the function header as “function [a,b,c] = myfunc(d,e,f)”. Here `d,e,f` are input placeholders and `a,b,c` are output placeholders. In the `eig` example, the first output is assigned to the matrix `V`, and the second to the matrix `D`. 
Function prototype

- The function prototype provides enough information to the compiler so that it can check that it is being called correctly
- Defines the interface
  - Input (parameter), output (return value)

myexp.h file

```c
float myexp(float x);
```

myexp.c file

```c
float myexp(float x)
{
    const float precision = 1.0e-6;
    float term=1.0, res=0.0;
    int i=0;
    while (fabs(term)>precision) {
        res += term;
        i++;
        term = pow(x,i)/fact(i);
    }
    return res;
}
```

The implementation detail of myexp is in the .c file.

In order to use the function, we do not need to know the internal detail from the .c file, just the function prototype in the header (.h) file. If you design a module that uses the function myexp, you need to include the prototype at the top of the code; more usually you put the prototype into a file with a .h extension and use the compiler directive #include "myexp.h"

The whole C standard library is built like this, and to use functions from it you issue #include <math.h>. During the linking phase of Modules are then linked together to resolve undefined references.

The prototype encapsulates the function interface and hides the implementation detail.

Though Matlab does not enshrine the prototype in quite the same explicit way, if it helps, think of the descriptions returned by “help xxxx” in matlab as being the prototypes: you find out the interface without being exposed to unnecessary implementation detail.

Just for completeness, here is a version of myexp in Matlab:

```matlab
function y = myexp(x)
    precision = 1e-6;
    term=1;
    y=0;
    i=0;
    while abs(term)>precision
        y = y + term;
        i = i+1;
        term = x^i/factorial(i);
    end
end
```
Scope: local variables

- Variables which are declared inside a function are *local variables*
- They cannot be “seen” outside the function (block) in which they are declared
- A local variable exists only for the duration of the current function execution
- It is declared as a new variable every time the function is called
- It ceases to exist when the function returns
- It does not “remember” its value between calls

Note that in a Matlab context, variables are not declared explicitly. When a variable is used for the first time it is declared. If this use involves looking up the variable’s value it will cause an error: it must be an assignment, which implicitly declares the variable and sets its initial value. A common source of error in C programs is to forget to initialise the value of a variable before using it – the declaration does not set the value, it just allocates some memory.
Scope: global variables

- Global variables exist outside all functions
- A global variable is visible inside functions
- If there exist two variables, one local, one global, with the same name, then the local one takes precedence within its local scope
- C and Matlab behave differently
  - C will use a global if no local exists
  - Matlab only uses a global if the programmer explicitly requests it
- Globals should be used with caution because their use inside a function compromises its encapsulation

Global variables are usually the resort of the lazy programmer. In general good practice dictates that you declare variables as close to where they are used as possible, and restrict their scope as much as possible.

Matlab guards against “indiscriminate” use of globals within functions by assuming that all variables are local and giving an error if there is a reference to an unknown local variable – unless the variable has been explicitly declared global inside the function, viz:

    global varname;

In contrast, if there is no local variable that matches a variable in an expression, then C checks to see if there is a global one and uses that if there is.
Encapsulation

- Want the function to behave in the same way for the same inputs
  - encapsulate particular functional relationship

- But if the function depends on a global it could behave differently for the same inputs

- Live example using `myexp`

In the example, the inputs are the same to function “myexp”, but the output is different.

We have used a global variable “precision” to govern the roundoff error in our exp function (we might think this is a good idea because we could use it for all functions involving a calculation of this sort, like sin and cos, etc).

Later, we quite understandably used the term precision for the reciprocal of variance (as is common terminology), forgetting to declare it as a local variable in function “_tmain”. Resetting this to 1/variance drastically affects the output of myexp!

It is easy to say “you made a silly mistake not declaring precision in _tmain”, but this misses the point.

One way around this would be to make the global precision a const, viz:

```c
const precision = 1.0e-6;
```

Then the assignment in _tmain would generate an error at compile time.

However again the main point is that by making exp depend on something global, we effectively make its interface less transparent.
Function encapsulation

Input parameters → Output values

Hidden input

Input parameters → Output values

Hidden output
Side-effects

- Could set value of a global variable in a function
- Again this compromises the function’s encapsulation
  - Causes a side-effect
  - An implicit output, not captured by the interface
- Makes it difficult to re-use code with confidence
- c.f. C and Matlab function libraries
  - Set of re-usable routines with well defined interfaces

- In small projects maybe not a big problem
- Hugely problematic in bigger projects, especially when multiple programmers working as a team
- Complicates interfaces between components, possibly in unintended ways

There is a temptation to use “easy” techniques like global variables rather than rigorously defining function interfaces, but this is the resort of the lazy programmer and can lead to very obscure errors. Do not be fooled by the simple examples in the lecture. The complexity of a program even for relatively simple tasks grows rapidly to the point where it is very easy to make unknowing errors in coding.

In large projects, with multiple programmers working in teams this type of error can create enormous problems – one team may have no conception of the consequences of its implementation choices on another team. Errors relating to poor interface design may be obscure and hard to trace and eliminate. At best this creates expense, at worst serious risk of operational failure.

Because of the way so-called object-oriented programming encapsulates related data and functions together, emphasizing the interfaces between objects, it has become the paradigm of choice for large projects. Implement an object correctly, with the right interface, and it can be used with confidence as a building block. However, this does not mean that C cannot be used for large projects or that it’s not important to think about interfaces in C; in some respects this is even more important because the language itself doesn’t “force” it on us.
Whenever a function is called, the stack is used to store:

- return location (JSR, RTS machine code)
- parameter values
- local variables
- return values

When a function returns the stack shrinks as the local variables are removed. Execution returns to wherever the function was called from (as given by the return location), and the parameters are removed from the stack. Finally, the return values are removed from the stack. Recall that we said local variables don’t remember their values — this is why. Each time a function is called, the local variables are declared anew and allocated space on the stack.
Libraries

Well developed, generally useful functions are typically placed in *libraries*.

*Example: libm.a*

- *libm.a* is the C maths library that contains well-tested, efficient implementations of mathematical functions like `sin()`, `cos()`, `sqrt()`, etc.
- The function prototypes are in *math.h* and this is all the programmer needs to know. The function implementation detail is hidden from the programmer.

*Example: matlab toolboxes*

Function libraries are an example of encapsulation. The details of the implementation are unimportant and indeed could change if a new more efficient way of achieving the same end is discovered, providing the *interface* (ie the way the function is called, as given by its prototype) remains the same. From the perspective of the applications programmer it suffices to know that the functions have already been implemented by someone else and can be relied upon. Thus he/she stands on the shoulders of giants…
Pass by value/reference

```cpp
int i=5, j=10;
swap(i,j);
cout << i << " " << j << endl;
```

### Pass by value

```cpp
void swap(int a, int b)
{
    int temp = a;
    a = b;
    b = temp;
    return;
}
```

### Pass by reference

```cpp
void swap(int& a, int& b)
{
    int temp = a;
    a = b;
    b = temp;
    return;
}
```

Usually best to think of parameters as being *values*. All parameters in Matlab are passed by value.

If you change the values of the formal parameters in the function, this has no effect (see example of "swap" when `a` and `b` are passed by value).

However in C (well, actually, C++) you can specify that parameters are *passed by reference*. This is done with the `&` symbol. Instead of placing the value of an actual parameter on the stack, the location of the actual parameter is placed on the stack. The compiled machine code uses *indirect addressing* to access the parameters (recall from A2?)

One reason for this is to overcome the limitation that C only permits one return value.

If not exercised carefully and in a completely transparent way, changing pass-by-reference parameters in a function can destroy its encapsulation. One sensible way of managing this would be to have all input parameters first in the list, followed by any "output parameters", and make this clear in any comments and/or documentation.

Note, also, that arrays in C/C++ are always passed by reference. This means that changes to array elements in a function "stick" – they remain after the function returns. This is not the case in Matlab.
Here we see what the activation record looks like in each case.
Aside: pass by reference in C

```c
int i=5, j=10;
swap(&i,&j);
printf("%d %d\n", i, j)
```

Pass by reference in C

```c
void swap(int* a, int* b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
    return;
}
```

Unfortunately, C does not provide for explicit reference parameters (this feature is in C++ only). Instead we must make use of pointers.

A variable declared as a pointer (using the * notation) is memory location containing the address of another variable (the object pointed to). The & operator applied to a variable returns a pointer to that variable (i.e. it returns a memory address). The notation *a dereferences a pointer so that it refers to the contents of the location pointed to. Thus in the code above, we explicitly pass to swap the addresses of the variables i and j (i.e. the parameters to the function swap are both pointers to int), and then within swap we manipulate the contents of these memory locations by dereferencing the pointers.

This is pretty nasty! I provide this so you can recognise it. Unfortunately in C the use of pointers is almost unavoidable: it is used as here, as the way to achieve pass-by-reference, but pointers are also essential in C for building advanced data structures (see later).

In C++ many of these issues can be avoided. We have already seen that C++ explicitly provides for pass-by-reference. Note that the activation record looks exactly the same for the code above to that of pass-by-reference in C++. What C++ does is to remove the need for the programmer to worry about dereferencing the pointers in the function (the compiler will take care of that automatically), and therefore producing a cleaner and safer interface between the caller and called function.

C++, via the Standard Template Library (see next term) also provides lots of useful advanced data structures in which the use of pointers is hidden safely away from the programmer (i.e. encapsulated!).
Passing functions as parameters

• Consider an algorithm for numerical solution of a differential equation
  \[ Ydot = -Y \]

• The code may look like this:

```matlab
function [ta, Ya] = Euler(span, Y0, N)
  h = (span(2) - span(1))/(N-1); % step size
  ta = [span(1):h:span(2)]'; % column vector of times
  Ya = ta * Y0; % get Y array right size
  Ya = Ya .* 0; % benign initialisation
  Ya(:,1) = Y0; % initial condition
  for n = 2:N
    t = n*h;
    Ya(n,:) = Ya(n-1,:) + h * (-Ya(n-1,:)); % ydot + y = 0, hence ydot = -y
  end
end
```

• But we don’t want to have to re-write the code for every function we want to find the roots of!

```matlab
function [ta, Ya] = Euler(FnName, span, Y0, N)
  h = (span(2) - span(1))/(N-1); % step size
  ta = [span(1):h:span(2)]'; % column vector of times
  Ya = ta * Y0; % get Y array right size
  Ya = Ya .* 0; % benign initialisation
  Ya(:,1) = Y0; % initial condition
  for n = 2:N
    t = n*h;
    Ya(n,:) = Ya(n-1,:) + h * feval(FnName, t, Ya(n-1,:)); % ydot + y = 0, hence ydot = -y
  end
end
```
Passing functions as parameters

- More generally: \( \frac{dx}{dt} = f(x, t) \)

- What is \( f \) and can we encapsulate the Euler method without building \( f \) into it?

```matlab
function [tvec, x] = Euler(FnName, span, x0, N)

h = (span(2) - span(1))/(N-1); % step size

tvec = [span(1):h:span(2)]';      % column vector of times
x = xvec * x0;                            % get Y array right size
x(1,:) = x0;                                % benign initialisation

for n = 2:N
    t = n*h;
    x(n,:) = x(n-1,:) + h * feval(FnName, t, x(n-1,:));
end

end
```

- \( \text{FnName} \) is a user-defined function that takes current value of \( x \), and returns \( xdot = f(x, t) \)

Need to create a function to use with it. Consider:

```matlab
function xdot = MyFuncFirstOrder(t, x)
    xdot = -0.1*x;
end
```

Call with something like

```matlab
[t,x] = Euler('MyFuncFirstOrder',[0 10], 1.0, 200);
```

where 1.0 is the starting value, [0 10] is the time interval over which we want to compute \( Y \), and 200 is the number of steps (hence stepsize \( h = (10-0)/200 = 0.05 \))
Passing functions as parameters

- C code to contrast:
  ```c
  float MyFuncFirstOrder(float t, float Y) {
    return -0.1*Y;
  }
  
  float Euler(float FnName(float,float), float t, float h, float x0) {
    return x0 + h * FnName(t, x0);
  }
  
  int main() {
    int i;
    float Y[200], stepsize = 0.05;
    Y[0] = 1.0;
    for (i=1; i<200; i++) {
      Y[i] = Euler(MyFuncFirstOrder, i*stepsize, stepsize, Y[i-1]);
      printf("Y[%d] = %f\n", i, Y[i]);
    }
  }
  ```

Need to create a function to use with it. Consider:

```plaintext
function Ydot = MyFuncFirstOrder(t, Y)
  Ydot = -0.1*Y;
```

Call with something like

```plaintext
[t,Y] = Euler('MyFuncFirstOrder',[0 10], 1.0, 200);
```

where 1.0 is the starting value, [0 10] is the time interval over which we want to compute Y, and 200 is the number of steps (hence stepsize \( h = (10-0)/200 = 0.05 \))
Recursion

- Recursion is the programming analogue of induction:
  - If \( p(0) \) and \( p(n) \) implies \( p(n+1) \)
  - Then \( p(n) \) for all \( n \)

- Define a function in terms of
  - Itself
  - Boundary conditions

- For example
  - Factorial: \( n! = n \times (n-1)! \), \( 0! = 1 \)

When either designing recursive functions, or considering if a particular problem can (or should!) be solved recursively, it is useful to think about ways in which the problem can be subdivided into one or more problems with the same structure as the original. In some cases there are no gains to be had over a simple iterative version, and in other cases a naïve recursive version can be very inefficient (consider a simple implementation to compute the \( n \)th fibonacci number, \( \text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2) \) -- a naïve implementation would end up computing \( \text{fib}(2) \) many times. Exercise: can you think of a smart way of doing this in Matlab?).

However the power of recursion lies in (i) the fact that recursive implementation are often beautifully elegant and map cleanly onto a problem specification; (ii) significant gains can be had using a divide-and-conquer approach. Typically one starts with a problem and then divides it into two with the same structure, recursing, then combines the two in a clean way. See final lecture.

Examples of this are the sorting algorithm due to C A R Hoare (QuickSort), with a run-time of \( O(N\log N) \), and the Fast Fourier Transform due to Cooley and Tukey.
Recursion example: factorial

• Live demo

function y = factorial(n)

if n==0 || n==1
    y = 1;
else
    y = n*factorial(n-1);
end

What’s wrong with this code?

Consider factorial(-1)…
Recursion: example 2

Multiple recursion:

const int SIZE=256;
Bitmap im[SIZE][SIZE];

void fill(int x, int y, int old_colour, int new_colour) {
    if (x>=0 && x<SIZE && y>=0 && y<SIZE) {
        if (im[y][x]==old_colour) {
            im[y][x] = new_colour;
            fill(x-1,y,old_colour,new_colour);
            fill(x+1,y,old_colour,new_colour);
            fill(x,y-1,old_colour,new_colour);
            fill(x,y+1,old_colour,new_colour);
        }
    }
    return;
}
Data types and data structures

- C/C++ predefine a set of atomic types
  - `bool, char, int, float, double`

- C/C++ provides mechanism for building compound data structures
  - `struct (class)`
  - `Array`

- Matlab supports arrays/matrices (of course)
- Matlab also supports structures (and classes)
C/C++: struct and class

- A **struct** (class in C++) is a compound data type which encapsulates related data into a single entity

```c
typedef struct {
    double re, im;
} Complex;
```

- Defines how a variable of this type will look

```c
int i;
Complex z;  
```

The data elements that make up the struct/class are known as **fields**.

The concept of the C++ class is rather richer than the C struct. I will cover classes in more detail in the lectures next term. For now the key idea is that a struct/class encapsulates related data together into a single entity.
Example: VTOL state

- Represent current state as, say, a triple of numbers and a bool, (position, velocity, mass, landed)
- Single variable represents all numbers
  - Better abstraction!

```c
typedef struct {
  double pos, vel, mass;
  bool landed;
} State;
```

```
State s;
```

We could represent the state as a set of four “atomic” variables. However this would not capture the conceptual relationship between the variables, and would result in code has a more complicated interface. The controller would need to take 4 input variables instead of one! Likewise the simulator output would be four quantities. Harder to understand, less close to our data flow diagram.
Accessing struct/class members

State s;

s.pos = 1.0;
s.vel = -20.0;
s.mass = 1000.0;
s.landed = false;

s.pos = s.pos +
  s.vel*deltat;

Thrust =
  ComputeThrust(s);

• In Matlab introduce
  structure fields
  without declaration

  s.pos = 1.0;
  s.vel = -20.0;
  ...

  Thrust =
    ComputeThrust(s);

Data fields are accessed using the “.” (dot) operator.

“s” is a variable of type “State”. In object-oriented terminology, “s” is an
instance of “State”.

Accessing struct/class members

State s;

s.pos = 1.0;
s.vel = -20.0;
s.mass = 1000.0;
s.landed = false;

s.pos = s.pos + s.vel*deltat;

Thrust = ComputeThrust(s);

Or in Matlab declare a class

classdef State
    properties
        pos;
        vel;
    end
end

s = State();
s.pos = 1.0;
s.vel = -20.0;
...

Thrust = ComputeThrust(s);

The advantage of the latter matlab code is that trying to access a non-existent field in State will generate an error. If we used the version on the previous slide in which the fields are defined without predefining the structure then the typo

s.vek = -20.0

will create a new field “vek” in the struct s, giving us fields pos, vel and vek.
Output parameters

Image ReadImage(const string filename, bool& flag);

bool ReadImage(const string filename, Image& im);

- Input: filename (type string)
- Output:
  - im (type Image)
  - boolean flag indicating success/failure

function [Image, errflag] = ReadImage(filename)

- Basically the same, but cleaner in Matlab!

As shown on the previous slide, the use of classes is another way of returning multiple values from a function in C. Recall that an alternative is to use reference parameters.

How do you decide which to use? If the data really are conceptually related, then put the data together in a class/struct. If they are not then consider using reference parameters. An example of when this might happen would be if you want to return some data unless there is an error, in which case you want to flag the error. Two possibilities are shown in the slide. I would recommend the second form:

    bool ReadImage(const string filename, Image& im);

The informed reader would see this as one input parameter (indicated by the const keyword which means the function is not allowed to change its value), and one output parameter (Image& im is a reference to an image).

This function will return an image, read from the file specified, but also return a boolean value to indicate if the read was successful.

In Matlab, the cleaner looking function header (prototype) would be that makes it absolutely clear what the inputs and outputs are:

    function [Image, errflag] = ReadImage(filename)
Arrays

- An array is a data structure containing a numbered (indexed) collection of items of a single data type

```c
int a[10];
res = a[0] + a[1] + a[2];

Complex z[20];
State s[100];

for (t=1; t<100; t++) {
    s[t].pos = s[t-1].pos + s[t-1].vel + 0.5*g;
    s[t].vel = s[t-1].vel + g - GetThrust(s[t-1], burnrate)/s[t-1].mass;
    s[t].mass = s[t-1].mass - burnrate*escapevel;
}
```

Of course arrays are supported pretty much as atomic types in Matlab.
Multi-dimensional arrays

double d[10][5];

has elements:

d[0][0]  d[0][1]  ...  d[0][4]
.
.
.
d[9][0]  d[9][1]  ...  d[9][4]

C/C++ provides no inbuilt runtime checking of array bounds, so beware – it is possible to access, eg d[1][7]. Because of the way arrays are stored, this will actually return the value stored in d[2][2] (why??). If you are lucky, your program will crash with a “Segmentation Fault” or something nasty like this. If you are unlucky, it will apparently run but give nonsensical results and be very difficult to trace.
Dynamic structures

Arrays are static structures that live either in global memory space or on the stack.

It is often useful to be able to create space for variables (objects)\textit{dynamically at runtime}.

This is done in a memory area known as the \textit{heap}.

Useful dynamic structures:

- Linked lists
- Trees
- Graphs

What’s this got to do with software engineering and structured programming? The same patterns of data occur repeatedly in lots of different problems. The three dynamic structures mentioned are examples. It makes sense to re-use standard designs of these structures. This is made more formal in C++ through the Standard Template Library, and the general idea of design patterns but for now we will look at typical ways of creating these structures so that you don’t need to re-invent the wheel (which of course may be error prone).
Pointers and dynamic memory allocation

- Recall that a pointer is a variable that contains an address in memory
- The heap is an area of memory that can be allocated/freed at run-time by the program

Global variables are effectively allocated at compile time: the compiler knows everything it needs to know before the program runs about how much space is required, so they can be “built in” during compilation. Local variables are allocated at function call time. They always occupy the same amount of space so that the activation record is always the same size. Again this means that much can be done at compile time in terms of mapping variable names to memory addresses.

However it is often the case that space is needed at run-time. The heap is an area of memory available to the program for run-time allocations, and the data structures that make use of the heap are known as dynamic structures, because their size and form can change during the course of program execution.

Note that many languages offer much cleaner, safer support for dynamic memory allocation than C/C++. Indeed in some languages it is completely transparent to the programmer. Nevertheless, having some understanding of what happens and how to implement dynamic structures in C/C++ is a good thing, providing you don’t use them! 😊.
Dynamic memory in C

- In C dynamic (runtime) memory management is performed using
  - malloc()
  - free()

DataStructure *d;
d = malloc(sizeof(DataStructure));

DataStructure darray[];
Darray = calloc(N, sizeof(DataStructure));

In C a chunk of memory is allocated using the function call malloc(), which takes a size (in bytes) and returns a pointer to the bit of memory. The usual way of using this, therefore is as follows

DataStructure *d;
d = malloc(sizeof(DataStructure));

When this bit of memory is no longer needed, it is released back to the system using free()

free(d);

which releases the bit of memory pointed to by d (where the size is implicit in the type of object d points to).

Note the potential for memory leaks when a bit of the heap is not freed, but the pointer that references it disappears (a common pitfall when doing dynamic memory allocation). For example:

void func(Data d)
{
    Data *pd;
    pd = (Data *)pd = malloc(sizeof(Data));
    // no do some processing
    return;
}

Here dp, the pointer to the data, lives on the stack but references data on the heap (allocated using malloc). However when the function returns, dp ceases to exist, leaving the bit of memory it points to “orphaned”, and with no way to recover it.

Anyway, just to emphasize the point, usually the use of pointers is to be cautioned against. However having a good understanding of the workings, and actually using something are not the same thing. I recommend: (i) understand dynamic memory and especially the useful data structures it supports; (ii) find another way of solving your problem. In C++ there is a library called STL (Standard Template Library) that provides for lots of useful dynamic data structures, in which the detail of memory allocation/deallocation is hidden from the programmer. This is sound code re-use that avoids lots of potential errors.
linked list

Insertion of a list element

```c
struct list_elt {  
    Data d;  
    struct list_elt *next;
};

typedef struct listelt ListElement;

ListElement *linked_list=NULL;
```

Removing a list element

```c
ListElement *newElement = (ListElement *)malloc(sizeof(ListElement));
newElement->next = current->next;
current->next = newElement;
```

```c
ListElement *temp = current->next;
current->next = temp->next;
free(temp);
```

Defining a list element:

```c
struct list_elt {  
    Data d;  
    struct list_elt *next;
};

typedef struct listelt ListElement;
```

Set the list to have no elements:

```c
ListElement *linked_list=NULL;  
```

Insert an element at node current:

```c
ListElement *newElement = (ListElement *)malloc(sizeof(ListElement));  
newElement->next = current->next;
current->next = newElement;
```

Remove an element at current->next:

```c
ListElement *temp = current->next;
current->next = temp->next;
free(temp);
```

// frees the memory pointed to by temp (ie old current->next)
Trees

Consider binary tree

```c
struct node {
    Data d;
    struct node *left;
    struct node *right;
};

typedef struct node TreeNode;
```

**Depth first traversal**

Recursive *depth-first* traversal:

```c
void traverse(TreeNode *t)
{
    if (t==NULL) return;

    traverse(t->left);
    traverse(t->right);

    return;
}
```

This code will descend to the left-most, deepest node before hitting the end condition and exploring the right node of the deepest non-terminal node, etc.
A graph comprises a set of nodes and connecting edges

classdef graph
    properties
        nodes
        edges
        adjacency
    end
end

A simpler alternative that is amenable to MATLAB implementation comprises a two arrays and a matrix:

    an array of N vertices
    an array of M edges
    an NxN adjacency matrix in which the ij entry is either e, if the edge links the i and j vertices, or 0 if there is no edge
Algorithms

• A precise set of instructions (a “recipe”) for how to solve some problem
• We’ll look at some basic algorithms in sorting, searching and numerical computing to give a flavour
• We'll also look at:
  – The order of an algorithm
  – Aspects of formal proofs of algorithms, especially the idea known as loop invariants
• A common task is to sort a set of objects

• Suppose we have a set of $N$ objects where each object $i = 0, ..., N - 1$ has some property or key $a[i]$ associated with it which can be assessed as larger or smaller than some other object’s key $a[j]$.

• Then sorting into ascending order consists of relabelling the objects such that the keys are sorted
  
  $a[0] \leq a[1] \leq a[2] \leq \ldots \leq a[N - 1]$
Insertion sort

Algorithm:
- Each iteration considers an element and inserts it into the sub-array to the left at its correct position, shifting the other elements up as necessary
- Start at the first element and work up

- Simple to code and quite efficient for small arrays/lists
Insertion sort

Pseudo code:

Input: a[1]....a[N]

for i=2 to N
    key = A[i];
    j=i-1;
    // Insert the item into the sorted list a[1....j-1]
    // First shuffle up as far as necessary
    while (j>=1 && key<A[j])
        j=j-1;
    endwhile
    // Then insert
    A[j] = key;
endfor
Order of an algorithm

- Characterize functions according to their growth rates
  - different functions with the same growth rate represented using the same order
- “Big O” notation
  - describes the limiting behaviour of a function when the size of the arguments tend towards a particular value (usually infinity)
- Insertion sort:
  - Loop N times
  - Within each loop we do 1, then 2, then 3, etc operations
  - Total operations Sum(1:N) = N(N-1)/2
  - \( O(N^2) \)
Divide and conquer

- Divide and conquer is a recursive paradigm:

  - Solve(problem)
    - If problem difficult:
      - Subdivide problem into subproblems
      - For each subproblem Solve(subproblem)
      - Combine subproblem solutions
    - Return solution

- Note the recursive call
- Some of the best known and most famous (and useful) algorithms are of this form, notably quicksort and FFT


Complexity analysis of divide and conquer

- N items in first sub-problem
- N/2 items in each subproblem next level
- N/4 at next level, etc
  ...
- 2 items solved trivially

- \( \log_2(N) \) levels in the tree
- If combining subproblems at each level \( O(N) \), then overall complexity

\( O(N\log_2 N) \)
Merge sort

Works as pretty much a direct application of divide and conquer.

A pair of items are “trivially” sorted using a single comparison.

Two lists of sorted elements are merged by starting at the beginning of each and choosing the smaller element from the two lists to insert, moving to the next element in the list from which the smaller one came, and repeating the process.
Merge sort

MERGE-SORT(list)
if length(list) > 2 then
    Split list into firsthalf and remainder
    MERGE-SORT(firsthalf)
    MERGE-SORT(remainder)
    // combine firsthalf and remainder to give ordered list
    list = MERGE-SORTEDLISTS(firsthalf, remainder)
else if length(list) == 2 then
    // Swap if needed ...
    if (listhead > listtail) then
        temp = listhead
        listhead = listtail
        listtail = temp
    end if
else
    // Do nothing at all. The list with one element is sorted!
end if
return (list)
Insertion versus Merge Sort

InsertSort vs MergeSort

Time to Sort (sec)

Size of List

1 10 100 1000 10000 1e+06
Root finding

- Find zeros of a non-linear (scalar) function of one variable
- Method of bisection
  - Given $x_1$ and $x_2$ straddling root
  - Consider $f(r)$ where $r = (x_1+x_2)/2$
  - If $f(r)$ close to zero return $r$
  - If signs of $f(r)$ and $f(x_1)$ differ
    - look in $[x_1,r]$
  - Else
    - Look in $[r,x_2]$
Program correctness: Loop invariants

- The key to understanding many algorithms is understanding what action a loop or loops are performing.
- A loop invariant is a relation among program variables that is true when control enters a loop, remains true each time the program executes the body of the loop, and is still true when control exits the loop.
- Understanding loop invariants can help us analyze programs, check for errors, and derive programs from specifications.
  - The desired outcome of an algorithm can often be specified by a loop invariant and a terminating criterion (post-condition).
  - Together with precondition this can specify the entire algorithm.

Note this last point: we can use loop invariants in two ways. Either we can take existing code and check its correctness. Or we could design an algorithm on the basis of a loop invariant that forms part of the specification.
Loop invariants

- General formulation

```java
... // the Loop Invariant must be true at the start
while ( TEST CONDITION ) {
  // top of the loop
  // Loop Invariant must be true here
  ...
  // bottom of the loop
}
// Termination + Loop Invariant = Goal  ...
```

- Example: Consider insertion sort

We will consider the case of insertion sort in a moment. It comprises two loops. In each loop we aim for a single iteration to move us closer to some overall objective. A useful/meaningful invariant is one that ensures the goal gets closer at each iteration. We possibly start with an idea that we want to sort numbers like cards (see the picture). Then we need to articulate what precisely we mean by each step (ie design our invariants), then we can implement code to make sure the invariants are satisfied.
Loop invariants

- Example: insertion sort

for i=2 to N
  Invariant 1: \(a[1...i-1]\) is a sorted permutation of the original \(a[1...i-1]\)
  key = \(A[i]\);
  j=i-1;
  // Insert the item into the sorted list \(a[1...j-1]\)
  // First shuffle up as far as necessary
  while (j>=1 && key\(<A[j])
    Invariant 2: \(A[j...i]\) are each \(\geq key\)
    j=j-1;
  endwhile
  // Then insert
  \(A[j] = key\);
Endfor
Termination: i=N+1

Prove correctness by proving that the loop invariants hold, and that the combination of Invariant 1 with the terminating condition achieves the goal.

First, note that Invariant 1 is true initially because in the first iteration \(i=2\) \(A[1..i-1]\) is \(A[1]\) and a single element is always a sorted list.

In order to show that Invariant 1 is maintained by the loop and true during the next iteration, we examine the body of the loop. We require that after the last line of the outer loop \(A[j] = key\) that \(A[1..i]\) is a sorted permutation of the original \(A[1..i]\). We show this is true by examining Invariant 2.

Invariant 2 is true in the first iteration of the inner loop because \(j = i-1\) and we tested to ensure that key\(<A[j]\), and also \(A[i]=key\).

The inner loop maintains this invariant because the statement \(A[j+1] = A[j]\) moves a value in \(A[j]\), known to be \(> key\), into \(A[j+1]\) which also held a value \(\geq key\). Thus this statement does not change the validity of the invariant.

Upon termination of the inner loop, we know the following things about the array \(A\):
\(A[1..j]\) are sorted and \(<= key\)
\(A[i] = key\)
\(A[j+1...i]\) are sorted and \(> key\)

Thus, Invariant 1 is maintained after an iteration of the loop. The outer loop terminating condition is \(i=N+1\). So finally we for Invariant 1 that \(A[1...i-1] = A[1...N]\) is sorted.
Search

• Consider we have binary search tree, ie:
  – the left subtree of a node contains only nodes with keys less than the
    node’s key
  – the right subtree of a node contains only nodes with keys greater than
    the node’s key
  – both left and right subtrees are also binary search trees.

TreeNode *TreeSearch(Node *n, Key k)
{
    if (n==NULL)
        return NULL;
    else if (k < n->d.key)    // n->d is same as (*n).d
        return TreeSearch(n->left, k);
    else if (k > n->d.key)
        return TreeSearch(n->right, k);
    else
        return n;
}

• Complexity is $O(\log N)$

Recall we discussed the tree data structure earlier. Trees can be used to store data
leasing to very efficient and elegant search. Suppose we have a binary search tree,
as defined above, and we wish to determine if the tree contains a node with a certain
value of key. A recursive search algorithm very naturally suggests itself.
Minimum spanning tree

See tute sheet
Concept summary

• Software engineering principles
  – Abstraction
  – Encapsulation
  – Modularity
• Managing complexity
• Functions
  – Encapsulation of related instructions to create “Black-box” functionality
  – Code re-use
• Data structures
  – Encapsulation of semantically related data together into appropriate structures
• Algorithms
  – Designing algorithms
  – Standard algorithms
  – Loop invariants

Exam questions? See tute sheet to follow.