Aims (from Intro)

- To round off C programming skills for non-Computing majors
- Topics to be covered include:
  - structures
  - dynamic memory allocation
  - linked lists
  - trees
  - files & file operations
  - algorithms for searching, hashing and sorting
Learning Outcomes (from Intro)

- Be able to write C programs to specification using simple array, list and tree data structures and algorithms
  - understand in broad terms the structure and operation of computer systems
  - be able to make use of files for storing and retrieving data
  - be able to design large programs consisting of multiple source files
  - know how to test and debug C programs
  - be able to read and understand C code
Assessment (from Intro)

labs = total mark for 10 best labs (out of 10)

ass1 = mark for assignment 1 (out of 5)
ass2 = mark for assignment 2 (out of 10)
ass3 = mark for assignment 3 (out of 15)

exam = mark for final exam (out of 60)
okExam = (exam >= 25/60) (after scaling)

bonus = contribution to Wiki (up to 2)

mark = labs + ass1 + ass2 + ass3 + exam + bonus (capped at 100)

grade = HD|DN|CR|PS if mark >= 50 && okExam
       = FL if mark < 50 && okExam
       = UF if !okExam
Exam

- On at 8:45am, 4 November in CSE labs
  - Seating arrangement to be posted on course web
- 3 Hour CLOSED book exam
  - 4 written questions – 30 marks
    - Data structure design
    - Dynamic memory allocation
    - Argument stack & stack frames
    - Time complexity of given algorithm
  - 3 practical questions – 30 marks
    - Array
    - Linked list
    - Graph
Exam organisation

- C reference supplied (see link off web site)
Course evaluation

- Please complete the course evaluation
  - Particularly interested to know how course delivery could be improved
  - How objectives could be clarified
  - What worked well, what didn’t

- Immediate feedback
  - Text
  - Course advice
Review of main points: Weeks 1 - 8
The von Neumann Model

- Processor: control, calculation
- Memory: data & program storage
- Input/Output: interface to world
From high-level to low-level languages

High Level Language Program

Compiler

Assembly Language Program

Assembler

Machine Language Program

Machine Interpretation

Control Signal Specification

\[ \text{temp} = v[k] \]
\[ v[k] = v[k+1] \]
\[ v[k+1] = \text{temp} \]  
Real machines can’t execute C (or Haskell or…).

\[ \text{lw} \] $15, 0($2)  
\[ \text{lw} \] $16, 4($2)  
\[ \text{sw} \] $16, 0($2)  
\[ \text{sw} \] $15, 4($2)  
Real machines execute their own machine code

\begin{verbatim}
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
\end{verbatim}

\[ \text{ALUOP}[0:3] \leq \text{InstReg}[9:11] \land \text{MASK} \]  
(Register Transfer Level language)
Execution Cycle

- **Instruction Fetch**
  - Obtain instruction from program storage

- **Instruction Decode**
  - Determine required actions and instruction size

- **Operand Fetch**
  - Locate and obtain operand data

- **Execute**
  - Compute result value or status

- **Result Store**
  - Deposit results in storage for later use

- **Next Instruction**
  - Determine successor instruction
Memory & Execution

Topics to be covered:

- Memory representation of program data
- Execution of C idiom on a processor

Approach:

- MIPS processor, memory model & assembly language used as representative example
- Expect you to understand representation of data and how common constructs are executed
- Do not expect you to be able to translate C program fragments into assembly language
- MIPS reference provided for students who wish to understand assembly language in more detail
MIPS Memory Map
- where are programs & data stored?

Locally declared variables and function parameters

Data dynamically allocated by your program (see later)

Your program’s global variables are stored here
Main (Primary) Memory
- Program view of memory

- Effectively: a large (slow) array of registers

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>1004</td>
<td>0 0 0 0 0 0 0 1</td>
</tr>
<tr>
<td>1008</td>
<td>0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>100C</td>
<td>F F F F F F F F</td>
</tr>
<tr>
<td>1010</td>
<td></td>
</tr>
</tbody>
</table>

- Accessed by giving an address (location, offset) and reading/writing data there.
Memory allocation

- Depends on the type of data and the processor:

<table>
<thead>
<tr>
<th>Data type</th>
<th>Number of bytes typically allocated</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
</tr>
<tr>
<td>int, float</td>
<td>4</td>
</tr>
<tr>
<td>double, long</td>
<td>8</td>
</tr>
<tr>
<td>Pointer e.g. int *, char *</td>
<td>4</td>
</tr>
<tr>
<td>Array e.g. int a[10], char s[20]</td>
<td>Num elements * base data type</td>
</tr>
<tr>
<td>String constant</td>
<td>Num chars + 1</td>
</tr>
<tr>
<td>Other constants</td>
<td>Size of data type</td>
</tr>
</tbody>
</table>
3. Structures

Oliver Diessel
odiessel@cse.unsw.edu.au

Last updated: 10:10 11 Aug 2008
Structures

- Topics to be covered:
  - Using the keyword `typedef`
  - Primitive vs compound data types in C
  - Structures
  - Pointers to structures
  - How to pass structures as arguments to functions
  - How to return structures from functions
  - Combining structures

Slide credits: Vladimir Pervouchine
typedef – when is it useful?

- Meaningful names:

```c
typedef int phone_number;
typedef float currency;

phone_number my_number = 57384;
```

- Allow for easy changes:

```c
typedef float number;

number complex_calculation(number a, number b) {
    number c = log(a+b);
    ...  
    return c;
}
```
**typedef** as a shorthand for compound types

- **Primitive types:**
  ```
  int, double, float, char, ...
  ```

- **Compound types:**
  ```
  arrays and strings
  char S[MAX_LENGTH];
  float grades[NUM_ELEMENTS];
  ```

  ```
  typedef char surname_t[MAX_LENGTH];
  typedef float float_array[NUM_ELEMENTS];
  ```

  ```
  surname_t S;
  float_array grades;
  ```

  ```
  surname_t class_list[NUM_STUDENTS];
  ```
New compound type: structure

typedef struct {
    surname_t surname;
    surname_t given_name;
    int year_of_birth;
    int month_of_birth, day_of_birth;
    int height;
    int ID;
} person_record;

person_record player;
person_record team[MAX_TEAM_SIZE];
Exercise

Define a structure to account for this situation:

Cars have six-character registration numbers, and two dates associated with them – the date the car was first registered, and the date that the current registration expires. Each car also has fields (40-byte strings) for manufacturer, make, body type, and colour; and a field to record the number of owners it has had.

(Moffat)
4. Dynamic Memory Allocation

Oliver Diessel
odiesel@cse.unsw.edu.au

Last updated: 10:00 14 Aug 2008
Structures

Topics to be covered:

- Defining the size of an array
- Using `malloc()` and `free()`
- Control over the life-span of variables
- Changing the elements of an array in a function
- Returning arrays, strings and structures from functions
- Use of the `assert()` function
- Other memory allocation functions

Slide credits: Vladimir Pervouchine
Dynamic memory allocation

- **sizeof()**:
  
  ```
  sizeof(int); sizeof(char); sizeof(int*);
  sizeof(my_string);  // returns size of
  sizeof(custom_array);  // declared array
  ```

- **Allocation**:
  ```
  int n; // number of elements
  scanf("%d", &n); // read n
  int* custom_array = (int*)malloc(sizeof(int)*n);
  ```

- **Deallocation**:
  ```
  free(custom_array);
  ```

  Size of one element * Number of elements
Self-Referential Structures

- We can define a structure containing within it a pointer to the same type of structure

  ```c
  typedef struct lnode Lnode;
  struct lnode {
    int data;
    Lnode *next;
  };
  ```

- These “self-referential” pointers can be used to build larger “dynamic” data structures out of smaller building blocks
List Operations

- Fundamental List operations:
  - create a new node with specified data
  - search for a node with particular data
  - insert a new node into the list
  - remove a node from the list

- Other operations are possible and can be added as needed
- Lists also form the basis for useful data structures like stacks and queues
Stacks

- A stack is a collection of items such that the last item to enter is the first one to exit, i.e. “last in, first out” (LIFO)
- Based on the idea of a stack of books, or plates
Stack functions

- **Essential Stack functions:**
  - `push()` // add new item to stack
  - `pop()` // remove top item from stack

- **Additional Stack functions:**
  - `top()` // fetch top item (but don’t remove it)
  - `size()` // number of items
  - `isEmpty()`
Queues

- A queue is a collection of items such that the first item to enter is the first one to exit, i.e. “first in, first out” (FIFO)
- Based on the idea of queuing at a bank, shop, etc.
Queue functions

- **Essential Queue functions:**
  - `enqueue()` // add new item to queue
  - `dequeue()` // remove front item from queue

- **Additional Queue functions:**
  - `front()` // fetch front item (but don’t remove it)
  - `size()` // number of items
  - `isEmpty()`
Implementing Stacks and Queues

- A stack can be implemented using a linked list, by adding and removing at the head \[ \text{push()} \] \text{and} \ \text{pop()} \]

- For a queue, we need to either add or remove at the tail
  - can either be done efficiently?

**Note:** Both stacks and queues could just as easily be implemented using arrays.
  - A linked list is usually preferred because the memory used adapts to the amount of data stored
Example of a binary tree

Path length = 1
Level 1

Level 2

Depth 3 (Level 3)
Defined by path length from root to leaf

Height of tree = 3
Defined by longest path length from root to leaves
Binary Search Tree

- If all insertions in the tree follow the rule (at each node) that
  - if the new item has a key less than the key at this node, it is recursively inserted into the “left” subtree, and
  - if the new item has a key greater than the key at this node, it is recursively inserted into the “right” subtree,
then the tree is a binary search tree, and the leaves and internal nodes appear in sorted order from left to right across the tree.

- A binary search tree is a binary tree in which the objects are ordered from left to right across the tree.
Assignment 1 using BSTs

- Need functions to...
  - search a BST for a word
  - insert a word into a BST
  - print out the BST
  - free up the memory used by the BST
8. Function Calls & Stack Frames

Oliver Diessel
odiessel@cse.unsw.edu.au

Last updated: 11:30 11 Sep 2008
Functions

What exactly happens when we execute...

```
L1:
    z = f(x, y);
L2:
    ...
```

1. values of x and y are passed to f
2. control is transferred to function f
3. local variables of f are created
4. code for f executes
5. return value is set up
6. local variables of f are removed
7. return value is copied into z
8. execution resumes at statement L2

In this lecture, we examine how each of these steps are performed in detail
Function Calls

- First consider how to transfer control to a function and then back again.
- When we go to \( f \), we need to remember where to return to.
- The “jump and link” instruction \texttt{jal} provides this:

  \[
  \begin{align*}
  \text{L1: } & \ldots \\
  \text{jal } & f \# \text{Reg}[31] \leftarrow \&\text{L2} \text{ (store return address)} \\
  & \# \text{ PC} \leftarrow \&f \text{ (fetch next instruction from } f) \\
  \text{L2: } & \ldots 
  \end{align*}
  \]

- How to get back:

  \[
  \begin{align*}
  \text{f: } & \ldots \text{ # code for ”f”} \\
  \text{jr } & \$31 \# \text{PC} \leftarrow \text{Reg}[31] \text{ (fetch next instruction from return address)}
  \end{align*}
  \]

Recall: the program counter (PC) is a special register that stores the address of the next instruction word to fetch and execute.
Function Calls

A potential problem: \texttt{f} calls \texttt{g}.

\begin{verbatim}
L1: ...
jal f  # main calls "f"
L2: ...
f: ...
    jal g  # "f" calls "g"
L3: ...
    jr $31  # what value in $31?

    g: ...
    ...  # code for "g"
    jr $31
\end{verbatim}
Function Calls

A potential problem: \texttt{f} calls \texttt{g}.

L1: ...
jal \texttt{f} \ # main calls "f"
L2: ...

\texttt{f}: ...
jal \texttt{g} \ # "f" calls "g"
L3: ...
jr $31 \ # what value in $31?
g: ...
... \ # code for "g"
jr $31
Function Calls

A potential problem: \texttt{f} calls \texttt{g}.

L1: ...
   \texttt{jal f} # main calls "f"
L2: ...

f: ...
   \texttt{jal g} # "f" calls "g"

L3: ...
   \texttt{jr $31} # what value in $31?

\texttt{	extcolor{red}{g}: ...}
\texttt{...} # code for "g"
   \texttt{jr $31}
A potential problem: \( f \) calls \( g \).

L1: ...

jal \( f \)  # main calls "f"

L2: ...

f: ...

jal \( g \)  # "f" calls "g"

L3: ...

jr $31  # what value in $31?

So \( f \) needs to save its “return address”.

In another register?

That’s ok ... unless \( f \) calls \( g \) calls \( h \) calls ...
Function Calls

If main calls \( f \) calls \( g \) calls \( h \) ... 
...then \( h \) finishes, then \( g \) finishes, then \( f \) finishes 
and we’re back in main

Function call/return uses last-in, first-out (LIFO) protocol ⇒ use a stack of return addresses
MIPS Memory Map
- where are programs & data stored?

Program Address Space

- **Reserved for O/S**
- **Text Segment**
- **Data Segment**
- **Dynamic data**
- **Static data**
- **Program code**
- **Function call data**
- Stack

Function parameters, return addresses, and locally declared variables

Data dynamically allocated by your program (*malloc*, etc.)

Your program’s global variables are stored here

Words of program code (main and functions) – where PC points to
Stack Frames

- Inside a function, we need access to:
  - arguments
  - local variables

- When the function terminates, need to retrieve:
  - register values from the caller
  - previous stack pointer
  - return address

- All of these are located on the stack. Thus, a small region on the stack is associated with the invocation of each function.

- This region is referred to as a stack frame
GCC Stack Frame

Memory

Stack Pointer

$sp

High address

Stack grows down

Low address
gcc Stack Frame

- Memory
  - Stack Pointer $sp$
  - High address
  - Stack grows down
  - saved values of registers from calling function
  - Set up call in calling function
- Low address
Stack Frame

Memory

High address

Stack grows down

saved values of registers from calling function

arguments

Push excess arguments onto stack

Low address

stack pointer $sp
Stack Frame

- High address
- Stack grows down

Memory

- saved values of registers from calling function
- arguments
- previous frame pointer

In called function, push FP value of the caller

Low address

Stack Frame

- previou stack pointer $sp, $fp

In calling function, push FP value of the caller

- saved values of registers from calling function

- arguments

- previous frame pointer
gcc Stack Frame

Memory

High address

Stack grows down

saved values of registers from calling function

arguments

frame pointer

$fp

stack pointer

$sp

previous frame pointer

return address

In called function, push return address

Low address
Stack Frame

Memory

saved values of registers from calling function

arguments

previous frame pointer

return address

other registers (such as $s0-7)

High address

Stack grows down

Low address

frame pointer $fp

stack pointer $sp

In called function, push any other callee saved registers
### Stack Frame

The Stack Frame is a memory location used by the compiler to manage function calls. Here is a breakdown of its components:

- **Memory**
  - **High address**
  - **Stack grows down**

- **Saved values of registers from calling function**

- **Arguments**
  - **Previous frame pointer**
  - **Return address**

- **Other registers (such as $s0-7)**

- **Local variables**

The stack pointer ($sp$) and frame pointer ($fp$) are used to navigate the stack frame:

- **Stack pointer** ($sp$)
  - Points to the current position on the stack.

- **Frame pointer** ($fp$)
  - Points to the previous frame pointer in the stack.

When a function is called, space is created for local variables by creating a new stack frame below the current one, which is identified by the frame pointer. This process allows efficient management of temporary variables and function arguments.
Stack Frame

Memory

- saved values of registers from calling function
- arguments
- previous frame pointer
- return address
- other registers (such as $s0-7)
- local variables

High address Stack grows down

Low address

- Stack Pointer $sp
- Frame Pointer $fp

Points to first word stored by called function

Local data for active function
Streams

- C always opens 3 files, or *streams* when a program starts running
  - `stdin` is the standard input file, usually associated with the keyboard, but can take input from a file using Unix input redirection; functions `scanf` and `getchar` take their input from `stdin`
  - `stdout` – standard output file; associated with the terminal screen; can write to a file using Unix output redirection; `printf` and `putchar` send their output to `stdout`
  - `stderr` – standard error output file; usually associated with the terminal screen; can be redirected to a file using Unix redirection
10. Hashing

Oliver Diessel
odiesel@cse.unsw.edu.au

Last updated: 15:00 17 Sep 2008
File Operations

Topics to be covered:
- Dictionaries and searching
- Hashing
  - Dense and sparse integer ranges
  - Character strings