Lectures 6-7  Inf2C - Computer Systems: Intro to C

Boris Grot

School of Informatics
University of Edinburgh
Previous lectures

- **MIPS**
  - Arithmetic and memory
  - Control flow: branches and jumps
  - Function calls and the stack
Lectures 6-7: Intro to C

- **Motivation:**
  - C is both a high and a low-level language
  - Very useful for systems programming
  - Fast!

- This intro assumes knowledge of Java
  - Focus is on differences
  - Most of the syntax is the same
  - Most statements, expressions are the same
Performance: C vs. the rest

Source: http://attractivechaos.github.io/plb/
Outline

- A simple program; how to compile and run
- Major differences with Java
- Data types and composite data structures
- Arrays and strings
- Pointers
- Other issues
  - Memory regions
  - C Preprocessor
  - Portability
The hello world program

#include<stdio.h>

int main(void)
{
   // This is a comment
   printf("Hello world!\n");
   return 0;
}

Linux/DICE shell commands
Compile: gcc hello.c
Run: ./a.out
Major differences with Java

- C is not object oriented
  - C programs are collections of **functions**, like Java methods, but not class-based.
  - No inheritance, subtyping, dynamic dispatch in C

- C is not interpreted
  - A C program is **compiled** into an executable machine code program, which runs directly on the processor
  - Java programs are compiled into a **byte code**, which is read and executed by the Java interpreter (which is just another program)
C is less “safe”

- Run-time errors are not ‘caught’ in C
  - The Java interpreter catches these errors before they are executed by the processor
    - Example: array out-of-bounds exception
  - C run-time errors happen for real and the program crashes (or not 😐)

- The C compiler trusts the programmer!
  - Many mistakes go un-noticed, causing run-time errors and leaving systems vulnerable to security exploits
Memory management is different

- In Java
  - All objects dynamically allocated
  - Usable objects recycled automatically by garbage collection

- In C
  - No objects, only data structures
  - Some data structures statically allocated, others dynamically
  - Dynamically-allocated storage must be reclaimed (or freed) once the data structures there are no longer needed.
    - Major source of error, particularly when the programmer forgets to free the memory, resulting in memory leaks.
C has pointers …

- Pointers are special variables that reference (or point to) another variable
  - Similar to Java references

- We have already seen pointers in assembly:
  \l w \$t1,0(\$s2)
  - \$s2 is a pointer
  - C pointers are the same thing! (more later)
Built-in data types

- The usual basic data types are there:
  - **char** 8 bits
  - **short** 16
  - **int** 16, 32, 64 (same as machine word size)
  - **long** 32, 64
  - **float** 32
  - **double** 64

- Data type sizes are machine dependent
  - Unlike Java where an int is always 32 bits
- Normally signed. Unsigned available too
- No boolean type exists
  - for any number (int, char, ...): 0 false, other true
Composite data structures - struct

- Structures are like objects, but their types have no methods, unlike classes:
  ```c
  struct point {
      int x, y;
      // can include other data types and other structs
  } p1;
  struct point p2;
  ```

- Components accessed using “.” operator
  ```c
  p1.x = 2;
  ```
In memory: structures

```c
struct point {
    int x;
    int y;
} p1;
```

sizeof(point) = 8

What does `p1.y` translate into in MIPS?

```
addi $t0, $s0, 4 // $s0 points to the starting addr of p1
lw   $t0, (0)$t0 // load p1.y into $t4
```
User-defined types

- Define names for new or built-in types
  
  ```c
  typedef <type> <name>;
  ```

- Example:
  ```c
  typedef unsigned char byte;
  typedef struct {
      inx x;
      int y;
  } point;
  ...
  point p1, p2;
  ```
Arrays

- Syntax of C arrays similar to Java
- As in Java, C arrays have fixed size
- Example declarations of array:
  ```c
  int m[] = {5, 8, 10}; // size fixed to 3
  int n[2][10];        // two-dimensional array
                      // with 2 rows and 10 cols
  point p[4];          // array of 4 structs
  ```
- C arrays have no knowledge of their length
  - No checking that indexes are within bounds
- In C, close relationship between arrays and pointers
  - Pointers commonly used to pass arrays between functions
Strings

- C strings are simply arrays of type char
  - Encoded in 8 bits using ASCII

- They end with '\0', the null character
  char s[10]; // up to 9 characters long

- String initialisation
  char s[10] = "string"; // '\0' implied
  char s1[] = “string, too”; // length=12

- C rule for arrays:
  - Cannot store more chars than reserved at declaration
  - But bounds are not checked!
Strings – common operations

- **Assignment:** `strcpy(s, "string");`
- **Length:** `strlen(s)`
- **To get the 6\(^{th}\) character:** `s[5]`  
  - First char at position 0, as in Java arrays
- **Comparison,** `strcmp(s1, s2)` returns:
  - 0 when equal
  - Negative number when lexicographically `s1 < s2`
  - Positive when `s1 > s2`
- **Must** `#include<string.h>` **to call the functions**
  - Type: `man string` to see what’s available
Pointers

- We have seen pointers in assembly:
  ```assembly
  lw $t1,0($s2)
  ```
- $s2 points to the location in memory where the “real” data is kept
- $s2 is a register, but there’s nothing stopping us to have pointers stored in memory like “normal” variables
C pointers

- A C pointer is a variable that holds the address of a piece of data

- Declaration:
  int *p; // p is a pointer to an int
  - The compiler must know what data type the pointer points to

- Basic pointer usage:
  p = &i; // p points to i now
  *p = 5; // *p is another name for i

- & - address of operator.  * dereference operator

why?
Pointers as function arguments

- In Java
  - an argument with primitive type is passed by value (function gets copy of value)
  - an argument with class type is passed by reference (function gets reference to value)

- In C
  - All arguments passed by value
  - To get effect of `pass by reference’, use an argument with a pointer type
Example – the swap function

```c
void swap_wrong(int a, int b) {
    int t=a;
    a=b; b=t;
}
```

`swap_wrong` swaps the local variables `a, b` which are unknown outside of the function

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

Function call: `swap(&x, &y);`
C allows arithmetic on pointers:

```c
int a[10];
int *p;
p = a;  // p points to a[0]. Same as p = &a[0]
p+1 points to a[1]
  - Note that &a[1] = &a[0]+1
  - The compiler multiplies +1 with the data type size
In general: p+i points to a[i], *(p+i) is a[i]
Also valid: *(a+i) and p[i]
  - but cannot change what a points to. It’s not a variable
```
Practice questions

The following questions refer to the picture on the left:

- What is the machine value of \( p+1 \)?
- How can you get the effect of \( a[2]=5 \) using \( p \)?
- Which of the following looks suspicious (i.e., likely incorrect)?
  A. \( a[2]-p \)
  B. \( a[2]-*p++ \)
  C. \&a[2]-p
- Would the “suspicious” expression generate a runtime error?
More pointer arithmetic

Common expressions:

- `*p++` use value pointed by `p`, make `p` point to next element
- `*++p` as above, but increment `p` first
- `(*p)++` increment value pointed by `p`, `p` is unchanged

- Special value `NULL` used to show that a pointer is not pointing to anything (e.g., `p=NULL`)
  - `NULL` is typically 0, so statements like `if (!p)` are common

- Dereferencing a `NULL` pointer is a very common cause of C program crashes
Example – pointer arithmetic

Return the length of a string:

```c
int strlen(char *s)
{
    char *p=s;
    while (*s++ !='\0');
    return s-p-1;
}
```

- Argument/variable s is local, so we can change it
- Pointer increment, dereference and comparison all in one! No statement in the loop body
- Note pointer subtraction at return statement
More fun with strings & pointers

```c
char s1[10] = "Bob";
char s2[10] = "Bob";

if (s1 == "Bob")
    // do x
else if (s1 == s2)
    // do y
else
    // do z
```

Which statement (x, y, or z) is executed?
Dynamic memory allocation

- Pointers are not much use with statically allocated data
- Library function `malloc` allocates a chunk of memory at run time and returns the address

```c
int *p;
if ((p = malloc(n*sizeof(int))) == NULL) {
    // Error
}
...
free(p); // release the allocated memory
```
Pointers to pointers

- Consider an array of strings:
  ```c
  char *strTable[10];
  ```
- The strings are *dynamically allocated* ⇒ any size
- But the table size is fixed to 10 strings
- What if we don’t know the number of strings ahead of time?
  - Need to be able to provision array size on demand
  - That is, need to dynamically allocate the storage for the array of strings

  ```c
  char **strTable;
  ```
Pointers to pointers - details

Space must be allocated both for the table and the strings themselves
  – Pointer to pointer!

1  char **strTable;
2  strTable = malloc(n*sizeof(char *));
3  for (i=0; i < n; i++) {
4    // s gets a string of length l
5    *(strTable+i) = malloc(l*sizeof(char));
6    strcpy(strTable[i], s);
7  }
8  // strTable[i][j] == *((*(strTable+i)+j)
Memory regions and management

- **Memory areas**
  - *Heap*: dynamically allocated storage
  - *Stack*: for function/method local variables
  - *Static*: for data live during the entire program lifetime

- **In Java**
  - All objects on heap
  - Unusable objects on heap recycled automatically by garbage collection

- **In C**
  - Data structures in all 3 areas
  - Programs must explicitly free-up heap storage that is no longer needed
Memory regions in detail

- **Stack**: Managed by the compiler.
  - Managed by the programmer (in C) or the system (in Java).
  - Initialized when the process starts.
  - Variables for function calls are stored on the stack.

- **Heap**: Managed by the programmer (in C) or the system (in Java).
  - Variables for data structures are stored on the heap.

- **Static data**: Located in memory space and initialized when the process starts.

- **Instructions**: Contains machine code that the processor executes.

**Diagram:**

- **SP (Stack Pointer)**
- **PC (Program Counter)**
Categories of variables in C

- Global variables (statically allocated)
  - Defined outside of functions
  - Have *lifetime* of program and *scope* to file end
  - *extern* declarations extend scope before definition and to other files
  - Declare *static* to hide from other files

- Local (*automatic*) variables (allocated on stack)
  - Defined inside a function
  - Not available outside function
  - Distinct storage for each function invocation
  - Declare *static* for same storage for all invocations
Compilation units

- Programs are divided into *compilation units*
  - Provide degree of modularity
  - Each commonly has main file (.c) for source code
    - *Header* files (.h) *declare* public interfaces of units
- Each compiled separately to relocatable object code
  - Allows creation of object-code libraries
- A *linker* assembles these into an *executable*, resolving references between units
- A *loader* sets up the executable program in memory and initialises data areas, prior to program being run
  - Loader also computes addresses for Jump instructions
Declaration vs Definition

- Declaration: inform the compiler of the existence of a variable or function
  
  ```c
  void swap(int *a, int *b); // in .h file
  ```

- Definition: provide function body; allocate memory for globals
  
  ```c
  void swap(int *a, int *b) { // in .c file
    int temp = *a;
    *b = a;
    *a = temp;
  }
  ```
Compilation units example

A.h:

```c
int array_len;       // global
extern int MAX_SIZE; // global, defined elsewhere

// function declarations
void swap(int *a, int *b);
```

A.c:

```c
#include "A.h"

// function definition
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

main.c:

```c
#include <stdio.h>
#include "A.h"

int main(void) {
    int a = 5;
    int b = 15;
    swap(a, b);
}
```

Error?
The C pre-processor: \texttt{cpp}

- Includes – imports header files
  \begin{verbatim}
  #include <stdio.h>
  #include "A.h"
  \end{verbatim}

- Text substitution, e.g. define constants
  \begin{verbatim}
  #define NAME value
  \end{verbatim}

- Macros (inline functions)
  \begin{verbatim}
  #define MAX(X,Y) (X>Y \ ? \ X \ : \ Y)
  \end{verbatim}

- Conditional compilation
  \begin{verbatim}
  ifdef DEBUG
  Printf("Debugging message");
  endif
  \end{verbatim}

\texttt{gcc -DDEBUG ...}
That’s all folks

- Not all C features have been covered, but this introduction should be enough to get you started
- Useful things to learn on your own:
  - Standard input/output: `printf`, `scanf`, `getc`, ...
  - File handling: `fopen`, `fscanf`, `fprintf`, ...
- Look over past exam papers for simple C programming exercises
Coursework 1

- Assigned now, due in 2 weeks
  - Deadline: Wed, 26 Oct, 16:00h

- Task A: split a character string into words
  - Given: a C implementation
  - Your job: convert it to MIPS

- Task B: find the most commonly-occurring word in a character string
  - Given: C and MIPS implementations of Task A
  - Your job: write C and MIPS code for Task B
Coursework 1 (con’d)

- **Task A example:**
  
  input: The first INF2C-CS coursework  
  output:  
  The  
  first  
  INF2C  
  CS  
  coursework

- **Task B example:**
  
  input: The first INF2C-CS coursework comes first  
  output:  
  6  # number of unique words  
  2  # max frequency  
  1  # number of unique max-frequency words  
  first  # one of the max-frequency words
Coursework 1 (con’d)

Task B: you will need to store the unique words in memory as one big character array.

Two options:
1) Contiguous representation

```plaintext
my \0 name \0 is \0
```

2) Fixed width representation

```plaintext
my \0 name \0 is \0
```

*Max word length (inc. terminating char)*
A (friendly) note on plagiarism

- Don’t do it!!!!

- We use special software (MOSS, etc) to electronically cross-check all submissions
  - Unaffected by variable renaming, code reshuffling, etc.

- Remember: if you’re sharing your code, you’re just as guilty as the person taking it.