Computer Systems Architecture

http://cs.nott.ac.uk/~txa/g51csa/

Thorsten Altenkirch and Liyang Hu

School of Computer Science
University of Nottingham

Lecture 11: Pointers and References

What does the following C program print?

```
void swap(int x,int y) {
  int z;
  z = x;
  x = y;
  y = z;
}

int main() {
  int a,b;
  a = 2;
  b = 3;
  swap(a,b);
  printf("a=%d, b=%d\n",a,b);
}
```

References in C

- We can declare pointer types in C, e.g.

  `int *x;`

  means that `x` holds a pointer to an integer.
- To dereference a pointer we also use `*`, e.g. `*x` has type `int`.
- The operator `&` returns a pointer to a variable.
- E.g. if we have declared

  `int y`

  then `& y` has type `int *`, *pointer to an integer*.

What does the following C program print?

```
void swap(int *x,int *y) {
  int z;
  z = *x;
  *x = *y;
  *y = z;
}

int main() {
  int a,b;
  a = 2;
  b = 3;
  swap(&a,&b);
  printf("a=%d, b=%d\n",a,b);
}
```

The University of Nottingham

What about Java?

- Java hasn't got pointer types.
- Basic datatypes are always passed *by value*.
- Objects, arrays and strings are passed as references.
- Java avoids pointer bugs, which are common and hard to deteect.

What does the following Java program print?

```
class Int {
    int n;
    Int(int m) { n = m; } }

public class Swap {
    static void swap(Int x, Int y) {
        int z;
        z = x.n;
        x.n = y.n;
        y.n = z; }
    public static void main(String args[]) {
        Int a = new Int(1);
        Int b = new Int(2);
        swap(a,b);
        System.out.println("a="+a.n+" b="+b.n); }
}
```

swap in MIPS

```
        .data
aa:     .word 1
bb:     .word 2
        .text
        .globl main
main:   la $a0, aa
        la $a1, bb
        jal swap            # swap(&a,&b);
        ...                 # print a,b

swap:   # x=$a0, y=$a1, z=$t0
        lw $t0,($a0)    # z = *x;
        lw $t1,($a1)
        sw $t1,($a0)    # *x = *y;
        sw $t0,($a1)    # *y = z;
        jr $ra
```

Arrays

- One of the most basic data structures in CS
- Usually a block of consecutive elements in memory
  - All same size (s bytes); same offset from one to the next
  - The i^{th} element is at offset $i \times s$ bytes from beginning
  - Looking up an element of the array is termed '*indexing*'
- Characterised by constant-time indexing
  - No more faster to look up `xs[0]` than `xs[42]`
  - Contrast this with a linked-list[1] (not in this course)
- We can implement arrays using pointer arithmetic
- e.g. Assembly equivalent of an `int[]` in Java/C would be...
  - a consecutive block of word-sized signed integers
  - represented by its starting address and length

[1] like lists in Haskell

Using (Integer) Arrays: C

```
int as[8] = { 3, 1, 4, 1, 5, 9, 2, 6 };

int array_max(int xs[], int length) {
  int i,m;
  m = INT_MIN;
  for(i=0; i<length; i++) {
    if(m < xs[i])
      m = xs[i];
  }
  return m;
}

int main() {
  printf("max = %d\n",array_max(as,8));
}
```

Using (Integer) Arrays: Assembly, Part 1

```
array_max: # $a0: array address, $a1: array length
    li $v0, 0x80000000      # MIN_VALUE
    li $t0, 0               # i = 0
    j am_cond
am_loop:
        sll $t1, $t0, 2         # 4 * i
        add $t1, $a0, $t1       # xs + 4*i bytes
        lw $t1, ($t1)           # lookup xs[i]
        addi $t0, $t0, 1        # i++
        bge $v0, $t1, am_cond   # if(m < xs[i])
            move $v0, $t1       #     m = xs[i]
am_cond:
    blt $t0, $a1, am_loop   # i < length?
    jr $ra
```

Using (Integer) Arrays: Assembly, Part 2

```
        .data
as:     .word 3, 1, 4, 1, 5, 9, 2, 6
        .text
        .globl main
main:   addi $sp, $sp, -4
        sw $ra, 0($sp)
        la $a0, as      # $a0 = address of as
        li $a1, 8       # $a1 = as.length
        jal array_max   # array_max(as, as.length)
        move $a0, $v0
        li $v0, 1       # print_int
        syscall
        lw $ra, 0($sp)
        addi $sp, $sp, 4
        jr $ra
```

The University of Nottingham

Strings

- Java strings are opaque objects of class `String`
- Assembly strings are arrays of ASCII characters, or bytes
  - End marked with a NUL, rather than storing its length
- You've already used them before
  - with the `.asciiz` directive
  - and the `print_string` syscall
- What else can we do with strings?

String length in C

```
int strlen(char *s) {
  int l;
  l = 0;
  while(*s != 0) {
    s++;
    l++;
  }
  return l;
}

int main() {
  printf("%d\n",strlen("hello"));
}
```

String Length in Assembler

```
strlen: # s=$a0,l=$v0
li $v0, 0                 # l = 0 ;
j strlen_cond
strlen_loop:
addi $v0, $v0, 1          # l++
strlen_cond:
lbu $t0, ($a0)
addi $a0, $a0, 1          # s++
bne $t0, $zero, strlen_loop  # while(*s != '\0')
jr $ra
```

String Length in Assembler

```
        .data
hello:  .asciiz "hello"
        .text
        .globl main
main:   la $a0,hello
        jal strlen
        move $a0,$v0
        li $v0, 1       # print_int
        syscall
        li $v0,10
        syscall          # exit
```

strcat, 1st attempt

```
char* strcat(char *s, char *t) {
  char *r;
  r = s;
  while(*s != '\0') s++;
  do {
    *s = *t;
    s++;
    t++;
  } while(*t != '\0');
  return r;
}

int main() {
  printf("%s\n",strcat("hello ","world"));
}
```

Oops!

```
sean:code txa$ strcat1
Bus error
```

Dynamic Data

- So far we've only dealt with *static* data
  - Contents may change, but size and location doesn't
  - Same sense as the `static` keyword in Java
- In Java, `"hello" + "world"` contatenates two strings
  - But neither of the original strings are modified
  - Instead a new string is created on the *heap*
- The heap is a much larger pool of memory than the stack
  - In C we can allocate data using `malloc`
  - Unused data can be returned by using `mfree`
- Storage allocated on the heap persist across procedures
  - Caller can't access stack storage

The University of Nottingham

strcat, 2nd attempt

```
char* strcat(char *s, char *t) {
  char *r,*u;
  r = (char *) malloc(strlen(s)+strlen(t)+1);
  u = r;
  while(*s != '\0') {
    *u = *s;
    s++;
    u++;
  }
  do {
    *u = *t;
    u++;
    t++;
  } while(*t != '\0');
  return r;
}
```

Horrors of Memory Leaks

```
int main() {
  char *s;
  while(1) {
    s = malloc(1000);
    *s='x';
    printf(".");
  }
}
```

Horrors of Memory Leaks

- Program uses up all memory and will eventually crash.
- Small leaks hard to discover: may run for a long time

Java version

```
public class Foo {
    public static void main(String[] args) {
        while(true) {
            int[] as = new char[1000];
            as[0] = 'x';
            }}}
```

Automatic Garbage Collection

- Java has automatic garbage collection
  - Inaccessible objects are periodically freed by JVM
  - SPIM doesn't/can't have automatic garbage collection
- Can you write a Java program which runs out of memory?