C “Review”

Part 1

November 2, 2010
C Programming

• Some syntax similarities with Java
  – Loop control: for, while, do-while
  – If statement
  – Logical Operators

• Some differences
  – No class concept: C uses structures
  – No (?) Boolean type
    * typedef enum {FALSE,TRUE} bool;
  – Pointer concept
  – Declared variables are not initialized…
  – No type checking!
Operator Review

Relational operators

== != > < >= <=

• What is the difference between ‘==’ and ‘=’? ‘=’ is called the assignment operator.

Logical operators

|| && !

Arithmetic operators

+= -= *= /= %=

Bit-wise operators

>>>= <<= ^= |= &= ~=

When combined with the assignment operator, operators are called ‘compound operators’
Basic C Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
</tr>
<tr>
<td>long</td>
<td>4</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
</tr>
<tr>
<td>long double</td>
<td>12</td>
</tr>
</tbody>
</table>

Sizes differ across platforms!
These sizes are typical of a 32-bit system
Affects portability of code, assumptions!
Constant Types

const int c = 5;

#define PI 3.14
#define NAME "root"
#define MAGIC_NUMBER 0x2A

When these constants appear in the code.. what type does the compiler give them?
Code Example

#define PAGE_SIZE 4096
#include <stdio.h>

int main () {
    int memory_addr = 4200;
    int page_num;
    page_num = memory_addr / PAGE_SIZE;
    printf("Page %d\n", page_num);
}

Warmup Exercise: Compile and Run this code.

In ANSI C, variable declarations must come at the start of the block (or function)
Exercise 1

Write a C program that prints out the size of the following data types in bytes and bits: char, int, long, float, double

Hint: The `sizeof` built-in will return the size in bytes of its argument, for example `sizeof(char)`
The Stack

The stack (call stack) is a region of memory that is used by the C compiler

- Storing local variables
- Saving register values
- Passing and returning data between functions

A special register, the **stack pointer**, points to the current top (bottom) of the stack.

Another register, the **frame pointer**, points to the base of the current function's **frame**
How the Stack Works

On a function call, the return address is stored on the stack and the frame and stack pointers ($fp, $sp) are adjusted.

Some registers are saved on the stack, moving $sp.
Pointers

A pointer is a memory address.

"Pointers" solve two problems:
- Sharing data without copying
- Facilitate complex data structures (trees, etc.)

How many bytes is a memory address in 32-bit architectures? 64-bit?
Unary "star" operator
* symbol : 1st use

With variables: * is used to declare pointers

```c
int *intPtr;
int * intPtr;
int* intPtr;
char *x, y, z='a';
```

What is the difference between the first three declarations?
What type is y?
What type is z?
Address-of unary operator &

• The & operator returns the memory address of a variable.

```c
int main() {
    int x = 4;
    int *y;
    y = &x;
}
```

What is y's value?
"star" (Indirection) operator
Unary * symbol : 2nd use

Used to access the value at the pointer's memory address:

```c
int main() {
    int x;
    int *y;
    x = 4;
    y = &x;
    printf("%d", *y);
    *y = 3;
    printf("%d", x);
}
```

What is the output from this code?

accessing the value using * is called "dereferencing" or "following the pointer"
The other uses of * in C

• * is the binary operator for multiplication
• * is used for C-style commenting, where /* begins a comment block, and */ ends a comment block. Comment blocks can not be nested (one */ ends ALL preceding /*)

• Note: // comments are new to ANSI C.
Pointers

What happens when this code is run?

```
#include <stdio.h>
int main() {
    int x;
    int *y;
    x = 4;
    y = x;
    printf("%d", y);
    printf("%d", *y);
}
```
Manipulating Data with Pointers

```c
#include <stdio.h>

void foo(int *z) {
    (*z)++;
}

int bar(int *z) {
    printf("%d\n", *z);
    return (*z - 1);
}

int main() {
    int x;
    int *y;
    x = 4;
    y = &x;
    foo(y);
    X = bar(&x);
    printf("%d\n", *y);
}
```
A pointer must have a good memory address before "dereferencing"

Declaring a pointer variable gets space to store a memory address. Remember that C does not initialize variables.

**Option 1:** Assign a pointer an address from a variable or another pointer...

```c
int main () {
    int *x
    int *y
    int z = 4;
    y = &z;
    x = y;
    printf("%d", *x);
}
```

```
x 0x00000204
y 0x00000208
z 0x00000212
```

CSCI 135
Computer Architecture I
the Heap

```c
void myCalculation (int num_elements) {
    int my_array[num_elements];
    my_array[0] = 5;
}
```

This won’t compile!

The compiler needs to know the size of all local variables at compile time, so it can generate code to adjust the stack for function calls.
the Heap (continued)

• The operating system provides a large section of memory called "the heap" for use by programs.
• C provides two functions to help use the heap:
  \- void* malloc(size_t n);
  \- free(pointer);

What is the void* type?
the Heap (continued)

• **void** is a generic pointer. It is a memory address for any type of data (int, char, etc.).

• When the compiler dereferences a pointer, it needs to know what the type of the data is.

```c
int main() {
    void* x;
    int y = 4;
    x = &y;
    printf("%d", *x);
}
```

A void pointer (void*) cannot be dereferenced - it has no ‘type’.
Casting

- Sometimes we want to change data from one type to another.
- Casting lets us do this.
- Example: `getchar()` returns an integer.
  ```c
  char x;
  x = (char) getchar();
  ```
- This will cause C compiler to change the integer returned by `getchar()` into a `char`
using malloc

Option 2: Assign a pointer to a memory address in the heap returned from malloc

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
    int *x;
    x = (int*) malloc(sizeof(int));
    if (!x)
        fprintf(stderr, "unable to allocate x\n");
    else {
        *x = 5;
        printf("%d", *x);
    }
}
```

What does (int*) do here?

What does sizeof() do here?
using malloc (continued)

• `malloc` returns a `void*`, which should be cast to the appropriate type pointer.

```c
int *x;
x = (int*) malloc(5 * sizeof(int));
```

• `sizeof(type)` returns the number of bytes in a given type. Very useful with structures, more to come later…
malloc failure

• If malloc fails, it will return a memory address of 0, so we should check for failure

```c
#define BUFFER_SIZE 20

char* x;
x = (char*)malloc(sizeof(char)*BUFFER_SIZE);
if (!x) {
    fprintf(stderr,"unable to allocate x\n");
    exit(0);
}
```

• fprintf() outputs to a file, in this case the file specified is stderr, or standard error. exit(0) aborts the application, you may not always want to exit on a failed allocate.
Using Array Notation with Pointers

```c
char* GetEntryFromUser() {
    int index;
    int *user_entry;
    user_entry=(int*)malloc(20*sizeof(int));
    for (index = 0; index < 20; index++) {
        printf("Please enter # %d\n", index);
        scanf("%d", &(user_entry[index]));
        printf("%d", user_entry[index]);
    }
}
```

- You've malloc'ed space for 20 integers on the heap.
- `user_entry[index]` is another way to "dereference" the index'th integer.
using Address Notation

```c
char* GetEntryFromUser() {
    int index;
    int *user_entry;
    user_entry=(int*)malloc(20*sizeof(int));
    for (index = 0; index < 20; index++){
        printf("Please enter # %d\n", index);
        scanf("%d", (user_entry + index));
        printf("%d", (*(user_entry + index)));
    }
}
```

Integers are 4 bytes. When `index=1`, Do we get a "misaligned" read of the integer in bytes 1-4 of the heap? You actually read 2nd integer from bytes 4-7. The compiler adds `index*sizeof(int)` to the base address; it knows `user_entry` points to type int.
Handling Memory with C

Have a mental picture (or a real drawing) of how your C code is using memory.

The C programming model is that the programmer knows exactly what they want to do...

So why should the C compiler enforce meaningless rules?
void myInit(int *array, int size) {
    int i;
    for(i = 0; i < size; i++) {
        array[i] = i*(i + 1);
    }
}

int main( int argc, char *argv[] ) { 
    int *a;
    /* malloc an array with size 5 */
    /* pass the array to myInit */
    myInit(, 5);
    /* use printf to print out all the numbers. */
    /* then free the memory declared with malloc */
}
You made it!
More to come next time...