Overview

- C Syntax
- Important Tidbits in C
- Pointers
- Dynamic Memory Allocation
- Arrays
- Strings
- Common Pointer Mistakes
- Operators

Review: What is Subject about?

- Coordination of many levels of abstraction

Review: Programming Levels of Representation

- High Level Language Program (e.g., C)
- Assembly Language Program (e.g., ARM)
- Machine Language Program (ARM)

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
ldr r0, [r2, #0]
ldr r1, [r2, #4]
str r1, [r2, #0]
str r0, [r2, #4]
```

```
1110 0101 1001 0010 0000 0000 0000 0000
1110 0101 1001 0010 0000 0000 0000 0100
1110 0101 1000 0010 0001 0000 0000 0000
1110 0101 1000 0010 0001 0000 0000 0100
```

```
```

COMP 3221
Microprocessors and Embedded Systems

Lecture 2: C-Language Review - 1

http://www.cse.unsw.edu.au/~cs3221

July, 2003

Saeid Nooshabadi

Saeid@unsw.edu.au
Review: What will You learn in COMP 3221?

° Learn big ideas in Microprocessors & Interfacing
  • 5 Classic components of a Computer
  • Principle of abstraction, used to build systems as layers
  • Data can be anything (integers, floating point, characters): a program determines what it is
  • Stored program concept: instructions just data
  • Principle of stack and stack frames
  • Compilation v. interpretation thru system layers
  • Principle of Locality, exploited via a memory hierarchy (cache)

Review: 5 Classic Components of a Computer

Control
Datapath
Memory
Processor
Input
Output
Network/Bus
ALU
Registers

Quick Survey

° How many of you have experience with:
  Java?
  C++?
  C?

° Important: You will not learn how to code in C in this one lecture! You’ll still need some sort of C reference for this course.

Compilation (#1/3)

° C compilers take C and convert it into an architecture specific machine code (string of 1s and 0s).
  • Unlike Java which converts to architecture independent code.
  • Unlike Haskell/Scheme environments which interpret the code.

° But how is it architecture specific?
  • You’ll know the answer to this by the end of next week.
Advantages of C-style compilation:

- Great run-time performance: generally much faster than Haskell or Java for comparable code (because it optimizes for a given architecture)
- OK compilation time: enhancements in compilation procedure (Makefiles) allow only modified files to be recompiled

Disadvantages of C-style compilation:

- All compiled files (including the executable) are architecture specific, depending on both the CPU type and the operating system.
- Executable must be rebuilt on each new system.

C Syntax

```c
#include <stdio.h>
int main (void) {
    unsigned int exp = 1;
    int k;
    /* Compute 2 to the 31st.*/
    for (k=0; k<31; k++) {
        exp = exp * 2;
    }
    ...
    return 0;
}
```

C Syntax: General

- Very similar to Java, but with a few minor but important differences
- Header files (.h) contain function declarations, just like in C++.
- .c files contain the actual code.
- `main ()` is called by OS
- `main` can have arguments (more on this later):
  ```c
  int main (int argc, char *argv[])
  ```
- In no argument correct form is:
  ```c
  int main (void)
  ```
- Comment your code:
  - only `/* */` works
  - `//` doesn’t work in C
  - gcc accepts `//`
C Syntax: Declarations

- All declarations must go at the beginning of a C block, before assigning any values.

Examples of incorrect declarations:
```c
test c = 0;
c = c + 1;
char d;     /* error */
for (int i = 0; i < 10; i++)
```

C Syntax: Structs

- C uses structs instead of classes, but they’re very similar.

Sample declaration:
```c
struct alpha {
  int a;
  char b;
};
```

- To create an instance of this struct:
  ```c
  struct alpha inst1;
  ```

- Read up on more struct specifics in a C reference.

True or False?

- What evaluates to FALSE in C?
  1. 0 (integer)
  2. NULL (pointer: more on this later)

- What evaluates to TRUE in C?
  1. everything else...

- No such thing as a Boolean type in C.

Address v. Value (#1/2)

- Consider memory to be a single huge array:
  1. Each cell of the array has an address associated with it.
  2. Each cell also stores some value.

- Don’t confuse the address referring to a memory location with the value stored in that location.
Address vs variable (#2/2)

1 word = 4 Bytes = 32 bits

Addresses

<table>
<thead>
<tr>
<th>Address</th>
<th>Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>80</td>
<td>24</td>
</tr>
<tr>
<td>84</td>
<td>65</td>
</tr>
<tr>
<td>88</td>
<td>32</td>
</tr>
<tr>
<td>92</td>
<td>90</td>
</tr>
</tbody>
</table>

Variables

<table>
<thead>
<tr>
<th>Variable Address</th>
<th>Variable Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>84</td>
</tr>
<tr>
<td>b</td>
<td>65</td>
</tr>
<tr>
<td>c</td>
<td>32</td>
</tr>
<tr>
<td>d</td>
<td>90</td>
</tr>
</tbody>
</table>

Pointers in C (#1/6)

° An address refers to a particular memory location. In other words, it points to a memory location.

° Pointer: High Level Language (in this case C) way of representing a memory address.

° More specifically, a C variable can contain a pointer to something else. It actually stores the memory address that something else is stored at.

Address vs variable (#2/6)

1 word = 4 Bytes = 32 bits

Addresses

<table>
<thead>
<tr>
<th>Address</th>
<th>Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>80</td>
<td>24</td>
</tr>
<tr>
<td>84</td>
<td>65</td>
</tr>
<tr>
<td>88</td>
<td>32</td>
</tr>
<tr>
<td>92</td>
<td>90</td>
</tr>
</tbody>
</table>

Variables

<table>
<thead>
<tr>
<th>Variable Address</th>
<th>Variable Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>84</td>
</tr>
<tr>
<td>b</td>
<td>65</td>
</tr>
<tr>
<td>c</td>
<td>32</td>
</tr>
<tr>
<td>d</td>
<td>90</td>
</tr>
</tbody>
</table>

Pointers in C (#3/6)

° Why use pointers?
  • If we want to pass a huge struct or array, it’s easier to pass a pointer than the whole thing.
  • In general, pointers allow cleaner, more compact code.

° So what are the drawbacks?
  • Pointers are probably the single largest source of bugs in software, so be careful anytime you deal with them.
Pointers in C (#4/6)

- To declare a pointer, just precede the variable name with a *

- Examples:
  - int *a;
  - int b;
  - int *ptr, var, var2;

- Warning: In the third example above, the variable ptr is a pointer to an integer, while var and var2 are actual integer variables. The asterisk only applies to one variable.

Pointers in C (#5/6)

- Mixing int and char pointers

- Examples:
  - int *a;
  - char *b;
  - int *ptr, var;
  - Char *chptr, ch;

- Notice that a pointer can only point to one type (int, char, a struct, etc.).

- An int* cannot point to a character, or vice versa.

- Why not?
  - Safety: Pointers are known to cause problems to careless programmers, so limit what a pointer can do.
**Pointer Arithmetic (#2/4)**

° What about array of integers (4 Byte size).

° ptr+1 will return a pointer to the next integer array element as well.

```c
int a[8];
```

```plaintext
24 a[0]
65 a[1]
32 a[2]
90 a[3]
45 a[4]
55 a[5]
11 a[6]
88 a[7]
```

**Pointer Arithmetic (#3/4)**

° What if we have an array of large structs?

• C takes care of it:
  In reality, ptr+1 doesn’t add 1 to the memory address, but rather adds the size of an array element.

```plaintext
Struc #1
24 a
65 b
32 c
90 d

Struc #2
45 a
55 b
11 c
88 d

Struc #n
```

**Pointer Arithmetic (4/4)**

° So what’s valid pointer arithmetic?

• Add an integer to a pointer.
• Subtract 2 pointers (in the same array).
• Compare pointers (<, >, etc.).
• Compare pointer to NULL (indicates that the pointer points to nothing).

° Everything else is illegal since it makes no sense:

• adding two pointers, multiplying pointers, etc.

**Questions**

° Which one of the following are invalid?

- pointer + integer  
  OK
- integer + pointer  
  Not OK
- pointer + pointer  
  Not OK
- pointer – integer  
  OK
- integer – pointer  
  Not OK
- pointer – pointer  
  OK
- compare pointer to pointer  
  OK
- compare pointer to integer  
  Not OK
- compare pointer to 0  
  OK
**Pointer Usage**

- Once a pointer is declared:
  - use `&` to return a pointer to an existing variable (the memory address of the variable)
  - use `*` to return the value pointed to by a pointer variable

- Example:
  ```c
  int *ptr, var, var2;
  var = 5;
  ptr = &var;
  var2 = *ptr;
  ```

**Dynamic Memory Allocation (#1/4)**

- After declaring a pointer:
  ```c
  int *ptr;
  ```
  `ptr` doesn’t actually point to anything yet. We can either:
  - make it point to something that already exists, or
  - allocate room in memory for something new that it will point to...

**Dynamic Memory Allocation (#2/4)**

- Pointing to something that already exists:
  ```c
  int *ptr, var, var2;
  var = 5;
  ptr = &var;
  var2 = *ptr;
  ```
  `var` and `var2` have room implicitly allocated for them.

**Dynamic Memory Allocation (#3/4)**

- To allocate room for something new to point to, use `malloc` (with the help of a typecast and `sizeof`):
  ```c
  ptr = (int *) malloc (sizeof(int));
  ```
  - Now, `ptr` points to a space somewhere in memory of size `sizeof(int)` in bytes.
  - `(int *)` simply tells the compiler what type (in this case) will go into that space (called a typecast).
Dynamic Memory Allocation (#4/4)

° Once `malloc` is called, the memory location might contain anything, so don’t use it until you’ve set its value.
° After dynamically allocating space, we must dynamically free it:
  ```c
  free (ptr);
  ```
° Use this command to clean up.

Arrays (#1/3)

° Declaration:
  ```c
  int ar[12];
  ```
declares a 12-element integer array.
° Accessing elements:
  ```c
  ar[num];
  ```
returns the numth element.

Arrays (#2/3)

° **Key Concept:** An array variable is a pointer to the first element.
° **Consequences:**
  - `ar` is a pointer
  - `ar[0]` is the same as `*ar`
  - `ar[2]` is the same as `*(ar+2)`
  - We can use pointer arithmetic to access arrays more conveniently.

Arrays (#3/3)

° **Pitfall:** An array in C does **not** know its own length.
° **Consequence:** We can accidentally access off the end of an array.
° **Segmentation faults** and **bus errors:** These are VERY difficult to find, so be careful.
Strings

- A C String is just an array of characters:
  ```c
  char *str = "hello";
  ```
- `str` points to the 'h'
- The next five elements are:
  - 'e', 'l', 'l', 'o', and NULL (`\0`)
- **Key Detail:** A C String is always terminated with a NULL, which is why they're called null-terminated strings.

Review: C memory allocation

- **Address**
  - Stack: Space for saved procedure information
  - Heap: Explicitly created space, e.g., malloc(); C pointers
- **Variables declared once per program**
- **Static**: Variables declared implicitly

Arguments to Functions

- Arguments can be:
  - passed by value: Make a copy of the original argument (doesn't really affect types such as integers).
  - passed by reference: Pass a pointer, so the called function makes modifications to the original struct.
- Passing by reference can be dangerous, so be careful.

Arguments to Functions: Example

```c
int a = 1, b = 1;
*alpha = &a, *beta = &b;
void point_less (int a, int b) {
  a = 0;
  b = 0;
}
void point_full (int *a, int *b) {
  *a = 0;
  *b = 0;
}
point_less(a, b);
/* After calling point_less, a and b are unchanged */
point_full(alpha, beta);
/* After calling point_full, a and b are changed */
```
Common Pointer Mistakes (1/2)

° Declare and write:
  int *p;
  *p = 10; /* WRONG */

° What address is in p?
  • Answer: NULL; C defines that memory address 0 (same as NULL) is not valid to write to.

° Remember to malloc first.

Common Pointer Mistakes (2/2)

° Copy pointers v. values:
  int *ip, *iq, a = 100, b = 200;
  ip = &a; iq = &b;
  *ip = *iq; /* what changed? */
  ip = iq;   /* what changed? */

Important Logical Operators

° Logical AND (&&) and bitwise AND (&) operators:
  char a=4, b=8, c;
  c = a && b;
  /* After this statement */
  c = a && b;
  /* After this statement */
  c = a & b;
  /* After this statement */

° Similarly logical OR (||) and bitwise OR (|) operators:

Important Shift Operators

° Logical AND (&&) and bitwise AND (&) operators:
  char a=4, b=8, c;
  c = a << 2;
  /* After this statement */
  c = a << 2;
  /* After this statement */

° Similarly logical OR (||) and bitwise OR (|) operators:
Things to Remember (#1/2)

° All declarations go at the beginning of each function.
° Only 0 and NULL evaluate to FALSE.
° All data is in memory. Each memory location has an address to use to refer to it and a value stored in it.
° A pointer is a High Level Language version of the address.

Things to Remember (#2/2)

° Use malloc and free to allow a pointer to point to something not already in a variable.
° An array name is just a pointer to the first element.
° A string is just an array of chars.