CS 350 - C Primer

Tyler Szepesi

January 16, 2013

Topics

1. Why C?
2. Data Types
3. Memory
4. Files
5. Endianness
6. Resources

Why C?

- C is exteremely flexible and gives control to the programmer
  - Allows users to break rigid rules, which are enforced by other languages (Java, C++, Scheme)
- C exposes the programmer to the underlying memory management
- C is lightweight, lots of external features
  - Barebones functionality builtin

Data Types

- A data type characterizes the size and operations that are permitted on the data
  - Also defines how data is stored in memory
- We are using a 32-bit MIPS machine
  - unsigned/signed char = 1 byte
  - unsigned/signed int = 4 bytes
  - unsigned/signed long = 4 bytes
  - unsigned/signed long long = 8 bytes
  - All pointers = 4 bytes

Structs

- A Struct is a aggregation of other data types
    - Also called a record type
- Its size depends on its member types
- Struct members are accessed with the '.' operator

Structs

Example:

```
struct foo_struct{
    unsigned int x;
    char a; char b; char c; char d;
};

sizeof (unsigned int) = 4
sizeof (struct foo_struct) = 8

struct foo_struct inst;
inst.x = 1; inst.a = 2;
```

Structs

- Structs are "usually" layed out in the way they are defined
  - Previous example
    - First 4 bytes are reserved for the int, and then 4 * 1 byte for the chars
    - The first byte of data will belong to the int

Pointers *

- C allows the programmer to manipulate memory locations directly
- For declaration, use the dereference operator '*'
  - int *ptr;
- A pointer is a variable which holds the address of a memory location
- The data type of the memory location is inferred by the pointer type (ie.: a pointer to an integer)

Pointers

- Usually a pointer is assigned the address of a variable
- Easily done with the reference operator '&'

```
int someVar = 1;
int *ptr = &someVar;
```

- Can also use casts to create custom pointers

```
int *ptr2 = (int*)(0xFFFF0000);
unsigned long var = 0xFFFF0000;
int *ptr2 = (int*)var;
```

Using Pointers

- The contents of a pointer is an address

```
printf("->0x%x", ptr);
->0xFF001200
```

- To access the content of the location use the dereference operator '*'

```
printf("->%d", *ptr);
->1
```

Pointers and Functions

- To be able to use "pass-by-reference" semantics we must pass a pointer
  - void foo (int *ptr, char *cptr);
  - foo can now change the value of the variables which ptr and cptr reference

```
int foo (int *ptr, char *
    cptr) {
    *ptr = 1;
    *cptr = 'a';
    return 10;
}
```

```
int a = 0; b = 0;
char c = '';
printf("%d,%d,%c",b,a,c);
b = foo (&a, &c);
printf("%d,%d,%c",b,a,c);
=>0, 0,
=>10,1,a
```

Arrays

- Arrays are a collection of elements of 1 data type
  - int a[10];
  - This reserves an array of size 10
  - What is the size of this in bytes (on MIPS)?
  - struct foo_struct[10];
- Much like the struct, each element is layed out contiguously in memory

```
if &a[3] = 0x10, then &a[4] = 0x14, etc
```

Pointers and Arrays

- In C, arrays are managed with pointer arithmetic
- The name given to an array, is in reality a pointer
- The '[]' operator performs the following operation:

```
int a[10];
a[4] == *(int)(&a[0] + 4 * sizeof(int))

&a[0] == a // a is in reality of type (int*) and
    contains the address of the first element
```

Pointers Example

```
#include <stdio.h>
#include <stdlib.h>
struct foo_struct {
    unsigned int x;
    char a; char b; char c; char d;
};

int main () {
    int i;
    char a[40];
    unsigned int *iptr = (unsigned int *) a;
    struct foo_struct *sptr = (struct foo_struct *) a;
```

Pointers Example

```
    for (i=0; i<40; i++) {
        a[i] = (char) i;
    }

    for (i=0; i<10; i++) {
        printf("%2d = 0x%08x\n", i, iptr[i]);
    }

    printf("x = 0x%08x a = %d b = %d c = %d d = %d\n",
        sptr[0].x, (int) sptr[0].a, (int) sptr[0].b,
        (int) sptr[0].c, (int) sptr[0].d);

    exit(0);
}
```

Pointers Answer

```
0 = 0x00010203
1 = 0x04050607
2 = 0x08090a0b
3 = 0x0c0d0e0f
4 = 0x10111213
5 = 0x14151617
6 = 0x18191a1b
7 = 0x1c1d1e1f
8 = 0x20212223
9 = 0x24252627
x = 0x00010203 a = 4 b = 5 c = 6 d = 7
```

Strings

- C strings are arrays of type "char" with a trailing '\0' character

```
char str[12] = "Hello World";
{'H','e','l','l','o','','W','o','r','l','d','\0'}
str[4] = 'o', str[11] = '\0';
printf ("%c", *str) // => 'H' (Why?)
```

- You can create constant strings like this:

```
char *cstr = "Some String";
```

  - GCC will create the string as a constant in .bss

Strings Example

```c
#include <stdio.h>
#include <stdlib.h>

static char *alpha = "abcdefghijklmnopqrstuvwxyz";

int main () {
    char array[12];
    char *value = 0;
    int i;
    for (i=0; i<12; i++) {
        array[i] = alpha[i];
    }
```

Strings Example

```c
printf("addr of array = %p\n", &array);
printf("addr of array[0] = %p\n", &array[0]);
printf("*array = %c\n", *array);
printf("addr of value = %p\n", &value);
printf("addr of value[0] = %p\n", &value[0]);
printf("value = %p\n", value);
printf("\n");

value = array;
printf("addr of value = %p\n", &value);
printf("addr of value[0] = %p\n", &value[0]);
printf("value = %p\n", value);
printf("*value = %c\n", *value);
printf("\n");
```

Strings Example

```
    value = &array[4];
    printf("addr of value = %p\n", &value);
    printf("addr of value[0] = %p\n", &value[0]);
    printf("value = %p\n", value);
    printf("*value = %c\n", *value);
    printf("\n");

    exit(0);
}
```

Strings Answer

```
addr of array = 0x7ffff80
addr of array[0] = 0x7ffff80
*array = a
addr of value = 0x7ffff7c
addr of value[0] = 0x0
value = 0x0
addr of value = 0x7ffff7c
addr of value[0] = 0x7ffff80
value = 0x7ffff80
*value = a
addr of value = 0x7ffff7c
addr of value[0] = 0x7ffff84
value = 0x7ffff84
*value = e
```

Structs and Pointers

- A pointer can point to a struct

```c
struct foo_struct *sptr;
```

- When accessing members of a struct via a pointer use either:

```c
(*sptr).a   // The brackets are required, '.' has a
    higher precedence then '*' operator
sptr->a     // This is the shortcut for (*sptr).a
```

- Note that you first dereference the struct and then access the member

Volatile Variables

- Declaring a variable volatile will disable value based optimizations by the compiler
  - ie.: Dead Code elimination or value folding
- If variables memory is changed without compiler knowledge, we must use volatile
  - Changing the value in another thread
  - Hardware changing mapped variables

Volatile Variables

```c
int main (int argc, char **
    argv) {
    int test;
    for (i=0;i<10;i+=1) {
        test += 1;
    }
    return test;
}
```

```c
int main (int argc, char **
    argv) {
    return 45;
}
```

```c
int main (int argc, char **argv) {
    volatile int test;
    for (i=0;i<10;i+=1) {
        test += 1;
    }
    return test;
}
```

Volatile Variables

```c
/* foo is memory mapped */
unsigned int foo
int main (int argc, char **
    argv) {
    foo = 0;
    while (foo != 255);
}
```

```c
/* foo is memory mapped */
unsigned int foo;
int main (int argc, char **
    argv) {
    while(true);
}
```

```c
/* foo is memory mapped */
volatile unsigned int foo;
int main (int argc, char **argv) {
    foo = 0;
    while (foo != 255);
}
```

Memory

- A program usually touches 3 sections fo memory
  - Stack
    - Local variables, determined at compile time
  - Heap
    - Dynamically allocated memory
  - Global Data
    - Constant values from the binary

Memory

- Stack grows down on MIPS
  - As new stack frames are created, the stack pointer is moved down
  - It is possible to run out of stack space
  - As stack frames become obsolete, the stack is "unwound" (space is reclaimed)
- Heaps are managed by the OS

Stack Memory

- To use stack memory, allocate local variables or pass arguments

```
void foo (int b) {
    int a;
}
```

  - The variable 'a' and 'b' is placed on the stack once foo is called
  - Be careful though! Using addresses of local variables after the function exits is WRONG!

- Recall CS 241 stack frame setups

Dynamic Memory

- To use memory on the heap, a memory region has to be allocated via malloc

```
void * malloc (size_t s);
char *str = (char*)malloc(sizeof(char) * 21);
```

- Storage allocated via malloc has to be freed once done using it

```
free (str);
```

- This is important, as this memory region is otherwise claimed until the next restart

Stack Memory

Example:

```
int * foo() {
    int var = 3;
    return &var;
}   /* At this point the stack region will be unwound */
    /* The address of var is no longer valid! */

void main () {
    int *a;
    a = foo();
    printf("%d", *a); /* This will produce erratic results
        */
}
```

Stack Memory

Example:

```
int * foo() {
    int var = 3;
    int *tmp = (int*) malloc (sizeof(int));
    *tmp = var;
    return tmp;
}

void main () {
    int *a;
    a = foo();
    printf("%d", *a);
}
```

Writing Files (Binary)

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main () {
    int i, rc, fd;
    unsigned int array[40];
    for (i=0; i<40; i++) {
        array[i] = i;
    }
    fd = open ("test-output", O_WRONLY | O_CREAT);
    if (fd < 0) {
        exit(1);
    }
```

Writing Files (Binary)

```
    rc = write (fd, array, sizeof(array));

    if (rc < 0) {
        exit(1);
    }

    close (fd);
    exit(0);
}

% cat test-output
#@u
   !$%
```

Reading Files (Binary)

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#define PRE_ROW (4)

int main() {
    int i, rc, fd;
    unsigned int array[40];
    fd = open("test-output", O_RDONLY);
    if (fd < 0) {
        exit(1);
    }
    rc = read(fd, array, sizeof(array));
    if (rc < 0) {
        exit(1)
    }
```

Reading Files (Binary)

```
    printf("offset = %4d : ", 0);
    for (i=0; i<40; i++) {
        printf("0x%08x " array[i]);

        if (((i+1) % PER_ROW) == 0) {
            printf("\noffset = %4d : ",
                (i+1)*(sizeof(unsigned int)));
        }
    }

    printf("\n");
    close(fd);
    exit(0);
}
```

Reading Files (Binary)

```
offset = 0 : 0x00000000 0x00000001 0x00000002 0x00000003
offset = 16 : 0x00000004 0x00000005 0x00000006 0x00000007
offset = 32 : 0x00000008 0x00000009 0x0000000a 0x0000000b
offset = 48 : 0x0000000c 0x0000000d 0x0000000e 0x0000000f
offset = 64 : 0x00000010 0x00000011 0x00000012 0x00000013
offset = 80 : 0x00000014 0x00000015 0x00000016 0x00000017
offset = 96 : 0x00000018 0x00000019 0x0000001a 0x0000001b
offset = 112 : 0x0000001c 0x0000001d 0x0000001e 0x0000001f
offset = 128 : 0x00000020 0x00000021 0x00000022 0x00000023
offset = 144 : 0x00000024 0x00000025 0x00000026 0x00000027
offset = 160 :
```

Endianness

- System 161 uses Big-Endian semantics
- Intel x86 uses Little-Endian
- Endianness defines the byte order in memory
  - Little-Endian has the least significant byte first
  - Big-Endian has the most significant byte first
- Big-Endian is what we are used to

Endianness

```
Example: x = 0xdeadbeef /* 3735928559 */
Little-Endian:
Least significant byte at lowest address
Word addressed by address of least significant byte

0 .. 7 8 .. 15 16 .. 23 24 .. 31
[ ef ] [ be  ] [  ad  ] [  de  ]

Big-Endian:
Most significant byte at lowest address
Word addressed by address of most significant byte

0 .. 7 8 .. 15 16 .. 23 24 .. 31
[ de ] [ ad  ] [  be  ] [  ef  ]
```

Reading Files (Binary): Little-Endian

```
offset = 0 : 0x00000000 0x01000000 0x02000000 0x03000000
offset = 16 : 0x04000000 0x05000000 0x06000000 0x07000000
offset = 32 : 0x08000000 0x09000000 0x0a000000 0x0b000000
offset = 48 : 0x0c000000 0x0d000000 0x0e000000 0x0f000000
offset = 64 : 0x10000000 0x11000000 0x12000000 0x13000000
offset = 80 : 0x14000000 0x15000000 0x16000000 0x17000000
offset = 96 : 0x18000000 0x19000000 0x1a000000 0x1b000000
offset = 112 : 0x1c000000 0x1d000000 0x1e000000 0x1f000000
offset = 128 : 0x20000000 0x21000000 0x22000000 0x23000000
offset = 144 : 0x24000000 0x25000000 0x26000000 0x27000000
offset = 160 :
```

Kthxbye

```
exit(0)
```

Questions?

Useful Resources

- cplusplus.com
- Wikipedia
- Linux/Unix man pages
  - man printf
  - man open
- Library functions are in chapter 3 of man
  - man 3 strcpy
- System calls are in chapter 2 of man
  - man 2 open