C and Assembly Language

September 10, 2019

Today

- Strings and structs, left over from last time
- Compiling with **gcc**
- C Language topics
  - Arrays and pointers
  - Typedefs
  - Structs
  - Memory Management
- Introduction to X64 Assembly Language

Strings

- In C, strings are simply arrays (pointers) of type **char**
- The end of the string is marked with a **0x00** byte
- Example: The string ”105” is represented by four bytes:

  0x 31 30 35 00
- Pitfall: Writing a long string into a small space. You may overwrite other data

Structs

- Heterogeneous records, like Java objects
- Typical linked list declaration:

```
typedef struct cell {
  int value;
  struct cell *next;
} cell_t;
```

- Usage:

```
cell_t c;
c.value = 42;
c.next = NULL;
```

How many bytes are allocated for **c**? for **p**?

- Usage with pointers:

```
cell_t *p;
p->value = 42;
p->next = NULL;
```

Find the error →

p->next is an abbreviation for **(*p).next**

gcc, Typical Compilation

From the Data Lab:

```
$ gcc -O -Wall -lm -o btest bits.c btest.c decl.c tests.c
$ ./btest
```

Easier:

```
$ make
$ ./btest
```


gcc Option Summary

- Output options
  - Default is **a.out**
  - **-o <filename>**, output goes to the named file
  - **-c**, compile but do not link; output goes to **program.o**
  - **-S**, assemble only; output goes to **program.s**
- Optimization options (uppercase “Oh,” not zero!)
  - **-O, -O1, -O2, -O3, -Og**
- Debugging option: **-g**, include symbolic debugging information
- Warning option example: **-Wall**
- Library option example: **-lm**, link with the math library

`gcc` and File Types

Some files from the Datalab:

`Makefile`		
`README`		
`bits.c`	Source file	
`bits.h`	Include file	
`bits.o`	Object file	**`gcc -c bits.c`**
`bits.s`	Assembly listing	**`gcc -S bits.c`**
`btest`	Executable program	**`gcc -o btest btest.c bits.o decl.o`**
`btest.c`		
`btest.h`		
`btest.o`		
`dlc`		
...		

Managing Compilation with `make`

- **`make`** is a command that reads **`Makefile`**
- Example extracted from the Datalab **`Makefile`:**

```
# Makefile that builds btest and other helper programs for the CS:APP data lab
#
CC = gcc
CFLAGS = -O -Wall
LIBS = -lm

all: btest fshow ishow

btest: btest.c bits.c decl.c tests.c btest.h bits.h
    $(CC) $(CFLAGS) $(LIBS) -o btest bits.c btest.c decl.c tests.c

fshow: fshow.c
    $(CC) $(CFLAGS) -o fshow fshow.c
```

- Actions taken *only* when sources are newer than target
- **`all`** is assumed when no target is given on the command line

Preprocessor Directives

- **`#include <filename>`**
- **`#include "filename"`**
  - Usually include header files, with extension **`.h`**
- **`#define PI 3.14`**
- **`#define TIMESFOUR(j) ((j)<<2)`**
  - Textual substitution--parentheses are important!
- **`#if #elif #else #endif`**

```
#ifndef _STDIO_H_
#define _STDIO_H_

    All of the code

#endif /* _STDIO_H_ */
```


Memory Management—Creation

- **`void *malloc(size_t size);`**
  - Allocates a block of (at least) size bytes
  - Returns a pointer to the start of the block
  - The block exists (on the heap) until it is explicitly recycled
- Usage:
  - **`cell_t *cp = (cell_t *)malloc(sizeof(cell_t));`**
  - Must cast the result of **`malloc`**

Memory Management—Recycling

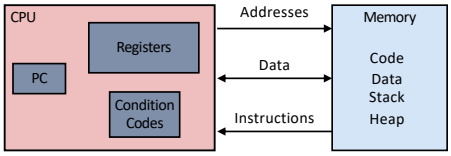
- **`void free(void *ptr);`**
  - ptr must be a value previously returned by **`malloc`**
  - Recycles the previously allocated memory block
  - Error to free twice
  - "Memory leak" when we forget to free
- Usage:
  - **`free(cp);`**
  - No need for cast; **`void*`** is compatible with any pointer type

More on **`malloc`** and **`free`** later in the course!

New Topic: X64 Assembly Language

- Intel Pentium: 64 bit instruction set
- Evolutionary design, going back to 8086 in 1978
  - Basis for original IBM Personal Computer, 16-bits
- Other languages are translated into X64 instructions and then executed on the CPU
  - Actual instructions are sequences of bytes
  - We give them mnemonic names

Assembly/Machine Code View



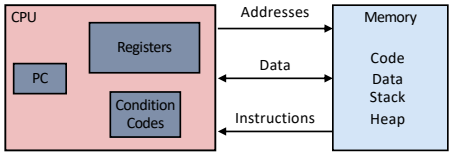
Programmer-Visible State

- PC: Program counter
- 16 Registers
- Condition codes

Memory

- Byte addressable array
- Code and user data
- Stack to support procedures

Assembly/Machine Code View



Repeat forever:

- Fetch instruction at address in PC
- Execute the instruction
- Update PC

X86-64 Integer Registers

%rax	%eax	%r8	%r8d
%rbx	%ebx	%r9	%r9d
%rcx	%ecx	%r10	%r10d
%rdx	%edx	%r11	%r11d
%rsi	%esi	%r12	%r12d
%rdi	%edi	%r13	%r13d
%rsp	%esp	%r14	%r14d
%rbp	%ebp	%r15	%r15d

- Can reference low-order 4 bytes (also low-order 1 & 2 bytes)

X86-64 Register Usage Conventions

%rax, function result	%r8
%rbx	%r9
%rcx, fourth argument	%r10
%rdx, third argument	%r11
%rsi, second argument	%r12
%rdi, first argument	%r13
%rsp, stack pointer	%r14
%rbp	%r15

Callee-saved registers are in yellow

Assembly Characteristics: Data Types

- "Integer" data of 1, 2, 4, or 8 bytes
  - Data values
  - Addresses (untyped pointers)
- Floating point data of 4, 8, or 10 bytes
- Code: Byte sequences encoding series of instructions
- No aggregate types such as arrays or structures
  - Just contiguously allocated bytes in memory

Assembly Characteristics: Operations

- Perform arithmetic function on register or memory data
- Transfer data between memory and register
  - Load data from memory into register
  - Store register data into memory
- Transfer control
  - Unconditional jumps to/from procedures
  - Conditional branches

Compiling into Assembly

```
long plus(long x, long y);

void sumstore(long x, long y,
              long *dest) {

    long t = plus(x, y);
    *dest = t;

}
```

```
sumstore:
   pushq   %rbx
   movq    %rdx, %rbx
   call    plus
   movq    %rax, (%rbx)
   popq    %rbx
   ret
```

Obtain assembly listing (on pom-itb-cs2) with command

```
gcc –Og –S sum.c
```

Produces the file **`sum.s`**

May get very different results on different machines!

Machine Instruction Example

```
*dest = t;
```

- C Code
  - Store value **`t`** where designated by **`dest`**

```
movq %rax, (%rbx)
```

- Assembly
  - Move 8-byte value to memory
    - Quad words in x86-64 parlance
  - Operands:
    - **`t:`** Register **`%rax`**
    - **`dest:`** Register **`%rbx`**
    - **`*dest:`** Memory **`M[%rbx]`**

```
0x40059e:  48 89 03
```

- Object Code
  - 3-byte instruction
  - at address **`0x40059e`**

Disassembling Object Code

```
0000000000400595 <sumstore>:
  400595:  53              push   %rbx
  400596:  48 89 d3        mov    %rdx,%rbx
  400599:  e8 f2 ff ff ff  callq  400590 <plus>
  40059e:  48 89 03        mov    %rax,(%rbx)
  4005a1:  5b              pop    %rbx
  4005a2:  c3              retq
```

- Disassembler
  - **`$ objdump –d sum`**
  - Useful tool for examining object code
  - Analyzes bit pattern of series of instructions
  - Produces approximate rendition of assembly code
  - Can be run on either **`a.out`** (complete executable) or **`.o`** file

Alternate Disassembly

```
Dump of assembler code for function sumstore:
 0x0000000000400595 <+0>: push   %rbx
 0x0000000000400596 <+1>: mov    %rdx,%rbx
 0x0000000000400599 <+4>: callq  0x400590 <plus>
 0x000000000040059e <+9>: mov    %rax,(%rbx)
 0x00000000004005a1 <+12>:pop    %rbx
 0x00000000004005a2 <+13>:retq
```

- Using the gdb Debugger

```
$ gdb sum
(gdb) disassemble sumstore
(gdb) x/14xb sumstore
```

  - Examine the 14 bytes starting at **`sumstore`**