Please note that some of the resources used in this assignment require a Stanford Network Account and therefore may not be accessible.

CS107 Handout 19
Spring 2008 April 28, 2008

Assignment 5: Raw Memory

Brought to you by Julie Zelenski and Jerry Cain.

Bits and Bytes

The first several weeks of CS107 are all about solidifying your understanding of memory: arrays, pointers, typecasts, parameter-passing, etc. and then supplementing it with a better understanding of compile-time language implementation: the layout and organization of memory, what kind of code is generated from a compiler, how the runtime memory structures (stack and heap) are managed, and so on. The implementation problem set is a break from the hard-core C coding. It's your chance to try your hand at generating machine code, to do some in-depth experimentation with the compiler and the debugger in order to dissect your programs. **Recall that you're not expected to hand anything in for this assignment**. You're to work on the problems independently, compare your solutions to the answer key (to be provided on Friday), and ask questions where needed.

To Be Completed By: May 7th at 7:00 p.m.

How to compile a program without a Makefile

You may want to write little test programs for the lab problems. You're used to relying on the provided Makefiles we wrote to build multi-module programs, but it is possible to just directly invoke the compiler at the shell. Refer to the big UNIX handout for lots of details— here is just a brief summary:

- To compile and link one file into an executable, you use:

```
% gcc binky.c -o binky
```

  which says to compile the file **`binky.cc`**, link it, and name the resulting executable **`binky`**.

- To compile multiple files into one executable, you first compile each file separately:

```
% gcc -c streamtokenizer.c -o stremtokenizer.o
% gcc -c wordgames.c -o wordgames.o
```

  This compiles the file **`streamtokenizer.c`** into the output file **`streamtokenizer.o`** and stops without linking it. (same for **`wordgames.c`**). And then you invoke **`gcc`** again to link like this:

```
% gcc streamtokenizer.o wordgames.o -o word-games
```

  that takes the two compiled object files and links them together into one executable called **`word-games`**.

- You can also add other flags for compiling after the **`gcc`** such as **`-Wall`** (to generate all warnings), **`-g`** (to include debugging information, and **`-O`** (for optimization).

Problem 1: Binary numbers and bit operations

Since computers work entirely in binary, learning how to manipulate numbers in the base two system can sometimes come in handy. Here are a few suggestions of some exercises to test out your understanding of binary numbers.

- Try converting a few decimal numbers into binary form, do some binary arithmetic with them (add, subtract, maybe even a multiply), and convert the result back to decimal to verify you have it correct. This lets you know you are on top of the basic workings and know how to do all that tedious carrying.
- Write a test program to find out what happens when you overflow the range of a variable, such as adding two **`short`**s that are both very large. Is any error reported on the overflow? How is the result related to what the desired answer would have been? What happens when you do the same thing with unsigned **`short`**s instead of signed?
- Write a program that assigns values between different-sized integer types. What happens when you go from a smaller-sized type to a larger? What about the other direction? How is the result related to the original number? Does this match your understanding of the binary representation?

In addition to the usual logical AND, OR, and NOT connectives, there are bitwise versions of these operations available in C. The bitwise AND (expressed with single &) compares its two operands bit-by-bit and reports which bits were 1 in both, 0 otherwise. For example:

```
unsigned char a = 12, b = 5, c;
c = a & b;
```

Doing a bitwise AND on 00001100 (12 in binary) and 00000101 (5 in binary) gives the result 00000100, since the two patterns have only one bit in common.

There is a bitwise OR operator (single |) that works similarly to logical OR, but operates at the bit level. There is also a bitwise exclusive OR (^) that reports which bits are on in one operand, but not both. The bitwise NOT (~) is a unary operator that just inverts all of the bits in its operand. Bit manipulations are used in a variety of situations (graphics, robotics, cryptography, etc.) especially when you need to work with a form of packed data.

- How could you use bit operations to determine the remainder of a number when divided by 4 (or any power of two for that matter?) How could use a bit approach to determine if an integer would lose data when assigned to a `short` or a `char`?
- How can you use bit operations to convert a number from negative to positive or vice versa? (Refer back to the "two's complement" representation for negative numbers we showed in class and try to work through what the patterns are in terms of bits.)
- One neat feature of the bitwise XOR operation is that it is completely invertible. If you XOR **`a`** with **`b`** and then XOR the result with **`b`** again, you get back **`a`** (trace this out for yourself). This makes this operation useful for encryption and decryption. How

could you construct a simple program that encrypts a file using XOR and a specified "key"? What would happen if you run the program twice in a row using the same key?

Probelm 2: ASCII and extended ASCII

In lecture, we showed the binary polynomial representation used for the integer-derived types. Characters belong to the integer family and although computers agree on the bit pattern used to represent the number 10 or 250, the mapping from number to character is where things break down. The ASCII character set establishes mappings for 0 to 127 that are used by all computers, but the extended ASCII characters, from 128 to 255, varies from system to system.

In **`/usr/class/cs107/assignments/assn-5-raw-memory`**, there is a file called **`ascii.txt`** that contains a table of all possible characters. Try viewing this file on UNIX and then again on Macintosh or a PC (either by using ftp to transfer the file or viewing the file with a Web browser) and observe how the exact same bit pattern can be interpreted as different characters on different systems. Which number-to-character mappings seem to be reliable? Which are not?

What implications does this have for using extended ASCII characters in a Web page intended to be viewed by all? What about sending those characters in e-mail exchanged between different computers? Does this exercise help explain any weird character translation problems you may have run into in the past?

Problem 3: It's just bits and bytes

One key idea we've been stressing is that memory is just one big glob of bytes. You can't tell whether the byte at address **`0x1024`** is a character or the first or third byte in an integer, whether it's initialized, whether it's in use, or anything meaningful at all from the bits stored there. Many bit patterns will have reasonable values in several interpretations. The `x` command in **`gdb`** will allow you to examine memory in all variety of interpretations (try "**`help x`**" in **`gdb`** for details on how to use it), so you can try "What if the contents of **`0x1024`** were a **`float`**, what would its value be? What if it were a machine instruction?" and so on. Try these few experiments and report your findings:

- Put the string **`"hi!"`** somewhere in memory and use gdb's **`x`** command to see how those 4 characters would be interpreted if treated as an integer. What about as a machine instruction?
- Use **`x`** to learn what the integer bit pattern **`3`** would be re-interpreted if treated like a `float`. Does the result turn out to be **`3.0`**?

This type of **`x`** operation is also available at runtime using a typecast. For safety reasons, most languages don't expose the typecast mechanism to the programmer and therefore constrain type conversion to a limited set of operations that actually make sense. C, on

the other hand, allows the typecast to be freely used in (almost) all situations, allowing the programmer immense control over interpretation of data, but also providing opportunity for lots of errors. You can do a lot of very twisted and weird things with typecasts, most of which you probably shouldn't even want to do. To show off your new prowess as a master of data manipulation, here are a few pressing needs you can solve. Try to construct a code snippet using a typecast that will:

- Print out the contents of a 4-byte **`struct`** as though it were a 4-byte **`float`**.
- Directly copy the first four characters of a string and assign them into an integer variable.
- Report whether the architecture is big or little endian. Recall big-endian means the most significant byte of a multi-byte value is at the lowest address, vice versa for little-endian.

Problem 4: Identical Outputs

As we've stressed in lecture, type information is used at compile-time to make decisions about how many bytes of data to load/store and calculating offsets and the like, but all that is left at runtime is a sequence of instructions and data laid out in memory for it to operate on. Looking at the generated code tells you little about what a given 4 bytes is: an **`int`**? a **`struct fraction *`**? a sequence of **`4`** characters? It is possible for many different C functions to compile to the same output— the exact same sequence of CPU instructions is appropriate if the functions access the same memory locations in the same pattern. For example, look at this set of machine instructions:

```
R2 = Mem[SP]
R3 = R2 + 4
R4 = Mem[SP + 4]
Mem[R4] = R3
```

a) Assuming that **`SP`** holds the address of the last local variable (and others follow at higher addresses), build a C snippet with variables of only pointer type (any type of pointer okay) that will compile to the above sequence of machine instructions. The code should not have any typecasts.

b) Construct another function with just one local variable involving only integer type (array or **`structs`** of integers also okay) that compiles to the same sequence of machine instructions. You can use typecasts if necessary.

Problem 5: Find the linked list

You are debugging a program. You suspect that the program has allocated a linked list containing the sequence of numbers (1,2,3), but you're not sure. If the list is present, the three elements will use the following type definition and will be allocated and set up in the normal way with calls to **`operator new`** (which in turn calls **`malloc`**).


```
struct list {
    int data;
    struct list *next;
};
```

Write a function that looks through the heap to see if a (1,2,3) list is present anywhere. Assume that the function takes no inputs but returns true if the list is present and false otherwise. You can assume that **`kHeapStart`** (a **`void *`**) and **`kHeapSize`** (an **`int`**) are global constants defined for you. Your function should not dereference any invalid pointers. Do not worry about alignment restrictions or rounding up allocations to some larger value.

Problem 6: Homestar Runner: The System Is Down

You are to generate code for the following nonsense code. Don't be concerned about optimizing your instructions or conserving registers. We don't ask that you draw the activation records, but it can only help you get the correct answer if you do. Be clear about what assembly code corresponds to each line of code.

a) Consider the following record definition:

```
typedef struct {
   int coachz;
   short *thecheat[2];
   homestarrunner **strongbad;
} homestarrunner;

void pompom(homestarrunner strongmad, homestarrunner *marzipan)
{
line 1    char bubs[4];
line 2    bubs[*bubs] = *(marzipan->thecheat[strongmad.coachz]);
line 3    ((homestarrunner *)(strongmad.thecheat))->strongbad += *(int *)bubs;
}
```

Generate code for the entire **`pompom`** function.

b) Now generate code for the **`puppetthing`** function. You needn't draw the stack frame out, but it can only help.

```
homestarrunner **puppetthing(homestarrunner *marshie,
                             homestarrunner& mrshmallow)
{
   return (**puppetthing(&mrshmallow, *marshie)).strongbad;
}
```

Problem 7: The Hitchhiker's Guide To The Galaxy

Consider the following type and function definition:

```
      typedef struct {
         short **arthur;
         char ford[12];
         int trillian;
         short zaphod[6];
      } galaxy;

      static galaxy *hitchhikersguide(galaxy *mice, short **dolphins);
      static short *thanksforallthefish(galaxy marvin, int *deepthought)
      {
line 1   marvin.zaphod[100] = deepthought[*marvin.ford];
line 2   ((galaxy *)((galaxy *)(marvin.zaphod))->ford)->trillian = **marvin.arthur;
line 3   return hitchhikersguide(&marvin + 1, marvin.arthur)->arthur[10];
      }
```

Generate code for the entire **`thanksforallthefish`** function.

Problem 8: C++'s Dark Side

Given the following C++ **`class`** definition, generate code for **`jedimaster::luke`** method. Assume that the parameters have already been set up for you, and don't worry about returning from the method. Be clear about which code pertains to which line. Recall that C++ references are automatically dereferenced pointers, and k-argument methods are really (k + 1)-argument functions, because the address of the receiving object is quietly passed in as the bottommost parameter. The address of the first instruction of the **`anakin`** method is synonymous with **`<jedimaster::anakin>`**.

```
      class jedimaster {

         public:
            int luke(jedimaster *macewindu, jedimaster obiwan);
            int& anakin(short *padme, jedimaster& leia);

         private:
            short council[4];
            short *yoda;
      };

      int jedimaster::luke(jedimaster *macewindu, jedimaster obiwan)
      {
line 1   obiwan.yoda += macewindu->council[40];
line 2   return obiwan.anakin((short *) &obiwan, *this);
      }
```