CS107 Handout 28
Autumn 2000 October 25, 2000

Assignment 2ab Solutions

Due to Julie Zelenski (`zelenski@cs.stanford.edu`)

Although we don’t require you to hand in your solutions to HW2 for grading, we expect you to work through the problems and you are responsible for this material. In particular, make sure you are well-versed in the layout and operation of the stack and heap, can trace and follow even complicated and nasty C code, and know how to generate machine code for accessing variables and arrays, arithmetic calculations, branches and loops, array addressing, function calls, and so on. It’s all fairly straightforward, but the more you practice, the quicker and easier it will be for you do the tracing and translating without making mistakes. This is quite important for doing well on the midterm!

You might want to read through this only after you've made a genuine attempt to thoughtfully answer all questions from HW2a and b.

Problem 1: Binary numbers and bit operations

· A few examples are below. Checking your results in decimal is the easiest way to verify you’ve got it correct.

```
 58 = 32 + 16 + 8 + 2           00111010
+47 = 32 + 8 + 4 + 2 + 1      + 00101111
105                             01101001  = 64 + 32 + 8 + 1 = 105

100 = 64 + 32 + 4               01100100
-27 = 16 + 8 + 2 + 1          - 00011011
 73                             01001001  = 64 + 8 + 1 = 73

20 = 16 + 4                     00010100
*3 =  2 + 1                   * 00000011
60                              00010100
                               00010100
                                00111100 = 32 + 16 + 8 + 4 = 60
```

· Adding one to the maximum signed short (32767) gives the most negative short –32768. Adding one to the maximum unsigned short (65535) returns zero. In both cases, it is doing the usual binary arithmetic and carrying the last bit into the sign bit or off the end, wrapping back into range. No error is reported in either case.

· Assigning from larger to smaller type just truncates to the lower byte(s). For example, assigning the short 1025 to a char produces 1. The other direction has no problem or truncation.

- The remainder of a number when divided by 4 can be found in the bits in the places below 4 (i.e. the bits in the 2 and 1's place). Thus we want to get the value of just those bits from the number. A bitwise AND can pull out a specific subset of bits, by AND'ing with something that has just those bits on. The integer 3 has only the 2 and 1 bits on, so if we take a number and bitwise AND it with 3, we can quickly compute its remainder when divided by 4:

  ```
  printf("Remainder of %d is %d\n", num, num & 3);
  ```

  If the upper byte of a short is all zero bits, then we can assign it to a char without data loss. Using a similar approach as above, we can extract just the high order bits from a number to see whether they are all zeros. The short with all the upper bits on is 32512 (32767-255).

  ```
  printf("High byte of %d clear? %d \n", num, (num & 32512) == 0));
  ```

  There are easier and more portable ways to compute the number to AND against (using bitmask or bit shifting), but we won't get into it for now.

- To change a number's sign in two's complement, we need to invert all the bits and add one. Inverting all the bits can be done with a bitwise XOR with that number that has all bits on. This operation will turn all bits previously off on and all bits on off, exactly what we want. The value with all bits on is –1.

  ```
  printf("Start with %d, change sign %dn", num, (num ^ -1) + 1);
  ```

- Here's a simple encryption / decryption program that reads a file from standard input and writes the result to standard output. It encrypts every character with a constant key of 'Z'. You could adapt it to encrypt by shorts or ints and using different keys if you felt like it. If you run this program twice on the same file, you will get the original contents back.

  ```
  #include <stdio.h>

  #define KEY 'Z'

  int main(void)
  {
     int ch;

     while ((ch = getc(stdin)) != EOF)
        putc(ch ^ KEY, stdout);
     return 0;
  }
  ```

Problem 2: Bits and Bytes

Most people got this right away. Each card represents a different bit (the number in the upper left corner) and the numbers on each the card are those that have that bit on. If you take the cards handed back to you and add the bits together you get the chosen number. Voila!

Problem 3: ASCII and extended ASCII

As expected, the mapping for characters in the bottom half of the range is well-established, but the upper half is full of variation. For example, the ASCII value 153 is mapped to the trademark symbol on the Mac, but to the u with a circumflex over it on UNIX.

Setting up a web page or sending an email with extended ASCII characters is therefore not a reliable activity. Unless the receiver is using a system with the same character mappings, there is no guarantee your intention is properly interpreted.

Problem 4: Floating point numbers

- When a floating point operation overflows, the result is set to the special pattern for positive or negative "Infinity" to indicate it is out of range. This is unlike integer overflow that just silently wraps around on overflow. Part of the IEEE floating point specification includes dedicating a few reserved bit patterns to represent these special values.
- Floating point division by zero results in a positive or negative Infinity, except for the case of 0/0, which gives the result NaN (Not a number), another of the special floating point patterns. Integer division by zero raises an arithmetic exception at runtime.
- Adding a small number (or even a fairly large number that is still not close to FLT_MAX) won't change the number at all. This demonstrates the resolution of the floats becomes fairly coarse when the exponent is large. This is just part of the tradeoff of the floating point representation. You can represent very large numbers, but at a loss of precision.
- The first two doubles report that they are ==, but the other two do not, showing the kind of error that floating point approximation can introduce into even simple calculations. All four numbers compare as approximately equal within the value of DBL_EPSILON.

Problem 5: It's just bits and bytes

This is from an epic, results can differ due to machine endianness or instruction encoding:

- The string `"hi!"` as an integer: `1751720192`, as an instruction: `call 0xa1a59c00`.
- The integer `3` as float: `4.20389539e-45`, which is definitely not `3.0`.
-
```
struct binky b;
printf("Printing 4-byte struct as float %f\n", *(float *)&b);
```

·
```
   int i;
   char *s = "Hi there!";
   i = *(int *)s;
```

·
```
   short s = 1;         // s will have 1 in LSB, 0 in MSB
   char ch = *(char *)&s;     // read out first byte of short
   printf("Endian is %s.\n". (ch == 0 ? "big" : "little"));
```

·
```
   float f = 3.14159;
   printf("Value %d\n", f);   // printf will use unconverted bits
```

Problem 6: Performance lab

· These are the results running on one of the epics. All times are expressed in seconds.

Version	**User time**	**Real time**
SwapInt	1.3	1.8 – 2.6
SwapAny	14.3 - 14.7	21.7 - 29.4

I ran about a dozen trials each to observe any interesting variations. Note that while real time ("elapsed time") can range somewhat, user time ("CPU time") is fairly consistent, usually varying only by tenths of a second. Real time includes time spent waiting while the processor is being used for other jobs so it isn't a very interesting or reliable measure of performance.

The generic approach shows some significant overhead costs relative to the hard-wired version, it is practically an order of magnitude slower in the time trials— wow!

· Here are the results for the `SwapInt` function experiments:

Version	**User time**	**Real time**
extra parameter	1.3	1.9 - 2.5
using heap	10.7 - 10.9	13.8 – 16.0
using memcpy	4.3 - 4.4	6.2 - 10.6

The extra parameter seemed to have no appreciable effect on the performance. Note that the `sizeof` operator is completely compile-time, so there is no runtime cost of using it. The heap appears to be the real culprit, using `malloc` and `free` as opposed to the stack for storage is responsible for about 75% of the performance cost, while `memcpy` is contributing the remaining 25%. Very interesting indeed!

Problem 7: Identical outputs

There are an infinite number of functions that can be created— that's the point of the question. Lots of different-looking C code can compile to the same sequence of machine instructions. If looking only the generated code at run-time, you can't distinguish between them.

·

```
void PtrOnly(void) {
   int *a, **b;
   *b = &a[1];    // or equivalently, *b = a + 1
}
```

·

```
void IntOnly(void) {
   int arr[2];
   *(int *)arr[1] = arr[0] + 4;
}
```

- The instructions for `SwapInt` and `SwapFloat` are exactly identical. Since our swapping operation is only concerned with the size of the pointee, not its representation, two identically-sized types will generate the same sequence of instructions. For different sized types, the load (`ld`) and store (`st`) instructions change to `lduh` and `sth` (for load and store half byte, the equivalent of our "`.2`") for shorts and do `ldub` and `stb` to store single bytes (our "`.1`") for chars. The rest of the instructions remain the same.

- All of the unoptimized functions are 15 machine instructions and when optimized, compile down to just 5 instructions, for a reduction of 2/3— not bad! Allowing the compiler to optimize can produce a pretty dramatic improvement.

Problem 8: Find the linked list

The basic strategy is to scan the heap from bottom to top, at each location try to interpret the bytes as though they were the head node of such a list. We check what would be the data field and if it matches what we want, we access what would be the next field, verify that it points into a valid area of the heap and then dereference and see if what follows matches the rest of the list.

```
#define HEAP_END ((char *)HEAP_START + HEAP_SIZE - sizeof(struct list))

bool isListOnHeap(void)
{
   void *pos;

   for (pos = HEAP_START; pos <= HEAP_END; ((char *)pos)++) {
      struct list *current = (struct list *)position;

      if ((current->data == 1) && isInHeap(current->next)) {
         current = current->next;
         if ((current->data == 2) && isInHeap(current->next)) {
            current = current->next;
            if (current->data == 3) && (current->next == NULL))
               return true;
         }
      }
   }

   return false;
}
```

```
bool isInHeap(const void *ptr)              // Helper function
{
   return (ptr >= HEAP_START && ptr <= HEAP_END);
}
```

Problem 9: Experimenting with `malloc` and Purify

These answers came from using `malloc` on an epic, your mileage may vary.

· Asking `malloc` for a region of 0 bytes returns a heap address (which turns out be a node of size 8 bytes, the smallest handed out by the epic's `malloc`) that you can actually write to with no ill effects (until you get past the 8 bytes). I would think returning `NULL` (as the malloc on the Alpha machines do) would be a better choice here. If the user took care to check for a `NULL` return from `malloc`, then the problem could be discovered immediately.

  Purify doesn't report an error when you try to malloc a `0` byte region, but any attempt to write/read to that area will immediately result in an ABW or ABR error.

· The standard `realloc` appears to correctly detect when the pointer passed wasn't in the heap and returns `NULL` to indicate failure. If the user `asserts` on `NULL` return from `realloc`, they will immediately be informed of their mistake.

  Purify reports this as a FNH (freeing non-heap memory) error, most likely raised when Purify attempts to free the previous block after moving its contents.

· Writing off the end of a heap-allocated block is definitely a bad thing, but the observed symptom depends upon how much you overrun, what you write, and what's next to it in the heap. For example, overrunning just a byte or two often causes no problems, since the node is likely to be a bit larger than you asked for. But writing 5-10 bytes off the end (or more), is likely to cause some havoc. You will be writing into the space of the neighboring block. Since the epic's malloc uses a pre-node header, when you run off the end, you are writing into the node header of your next neighbor. The next time that neighboring piece's header is looked at (to determine its size in a `free` or `realloc` call, for instance), `malloc` often will crash.

  If you make this mistake under Purify, you will get the all-too-familiar ABW (array bounds write) error. Note that this error doesn't always mean the memory was an array and you indexed past the end. Although incorrect array indexing is one likely cause, an ABW error is reported for an attempt to write to space beyond the end of any malloc node.

· Freeing a `NULL` pointer appears to do nothing. No ill effects observed. The man page says this is a no-op. Purify doesn't report any error either.

- Freeing a string constant doesn’t seem to give consistent results. Sometimes it goes by quietly, but other times it appears to corrupt the heap such that a following `malloc` call would crash. It wasn't clear what the pattern was that would trigger a crash and what wouldn't. Purify reports a FNH (freeing non-heap memory) error. It also gives you a little warning about how some `malloc` implementations might allow this, but that it is not a portable practice.

- Attempting to free something in the stack appears to do nothing, or at least no bad results from it could be observed. Purify reports a FNH (freeing non-heap memory) error with same warning as above.

- Trying to `free` a heap pointer twice seems to do nothing (as long as the pointer wasn't reallocated in the meantime, in which case it will prematurely `free` that). Since most `malloc` implementations need to track the `free`d status of each node, it is usually easy to detect a double `free` and handle gracefully.

  Purify reports a FUM (freeing unallocated memory) and tells you how many times the block has been freed and where in the code, to help you identify the problem.

- When a piece is actually freed (i.e. after the little grace period on UNIX expires), the first 4 bytes of the node are changed (it uses those bytes to store the next pointer chaining together the free list), so attempting to use that freed region will meet with immediate and unpleasant surprises. However, this only seemed true for small (< 16 byte) nodes being returned to heap. Larger ones didn't show this effect. Under Purify, reading from a freed region gives a FMR (free memory read), writing to it produces a FMW (free memory write).

Problem 10: Code Generation

- First off, it helps to have a picture to get the details right:

struct garden
(20 bytes)

	type	**size**	**offset**
cucumber	double *[2]	8	12
onion.leaves	short[2]	4	8
onion.roots	int **	4	4
zucchini	float	4	0

main AR
(8 bytes)

		type	**size**	**offset**
	return addr	address	4	4
SP ->	backyard	struct garden *	4	0

```
SP = SP - 4                    ;; make space for local variable

R1 = M[SP]                     ;; load value of backyard
R2 = M[R1 + 12]                ;; load value of backyard->cucumber[0]
R3 = R1 + 40                   ;; add 2*sizeof(struct garden) to backyard
                               ;; (remember &backyard[2] => backyard + 2)

SP = SP - 8                    ;; make space on stack for parameters
M[SP + 4] = R3                 ;; assign second param
M[SP] = R2                     ;; assign first param

CALL <Grow>                    ;; save return addr, start executing Grow

SP = SP + 8                    ;; clean up space on return
R4 = ItoF RV                   ;; convert return value from int to float
R5 = M[SP]                     ;; load backyard
M[R5] = R4                     ; assign to backyard->zucchini

SP = SP + 4                    ;; clean up locals
RET                            ;; return from function
```

.

Grow AR
(20 bytes)

		type	**size**	**offset**
	corner	struct garden *	4	16
	broccoli	char **	4	12
	return addr	address	4	8
	spinach	int *	4	4
SP ->	peppers	short[2]	4	0

· As always, the first step is making space for locals:

```
SP = SP - 8   ;; make space for locals

#1  corner->onions.roots = &spinach

R1 = SP + 4             ;; compute address of spinach
R2 = M[SP + 16]         ;; load value of corner
M[R2 + 4] = R1          ;; store address in corner->onion.roots

#2  spinach[*spinach] = *((*(struct vegetable *)peppers).leaves);

R1 =.2 M[SP + 4]        ;; load value of *(*peppers.leaves)
R2 = M[SP + 4]          ;; load value of spinach
R3 = M[R2]              ;; load *spinach
R4 = R3 * 4             ;; multiply by sizeof(int)
R5 = R2 + R4            ;; add to array base
M[R5] = R1              ;; store at that element
```

One pretty tricky step is the first instruction of line #2. You could equivalently re-write the right-hand side of the C statement as `((struct vegetable *)peppers)->leaves[0]` which may make it easier for you to follow. Remember that for arrays declared on the stack, the name of an array is equivalent to the address of its 0th element. What location is the right-hand side location actually referring to after you untangle it?

```
#3  *(int *)(broccoli + peppers[0]);

 R1 =.2 M[SP]          ;; load value of peppers[0]
 R2 = R1 * 4           ;; multiply offset by sizeof(char *)
 R3 = M[SP + 12]       ;; load value of broccoli
 R4 = R3 + R2          ;; add ptr to offset
 RV = M[R4]            ;; deref, load value to return register
 SP = SP + 8           ;; clean up locals
 RET                   ;; return
```

Problem 11: Compilation Process

The preprocessor

The point of this exercise is to understand the consequences of how the preprocessor expands macros, which is by simple find-and-replace text substitution.

```
#define ADDRESS_OF_NTH(base, n, elemSize) (char *)base + n*elemSize;
```

· Consider this reasonable-looking use of the macro:

```
ADDRESS_OF_NTH(darray->elements, n-1, darray->elemSize)
```

Here's the result after the preprocessor does its job:

```
(char *)darray->elements + n-1*darray->elemSize;
```

Due to the precedence of multiplication over addition, this will not multiply the quantity (n-1) by the element size, but instead will subtract one element size from n. This will happen to be correct if the element size is 1, but in all other cases, we end up far off from where we intended.

To fix this, we need to enclose the macro arguments in parentheses so that we ensure that when the macro is expanded, we won't get such surprises:

```
#define ADDRESS_OF_NTH(base, n, elemSize) (char *)(base) + (n)*(elemSize);
```

For similar, but not quite the same reasons, we also should enclose the entire macro expansion in another set of parenthesis to avoid ill effects when combining it in a larger expression. Can you give an example where this lack of parentheses would get you into trouble?

· Consider this attempt to use the macro to copy a new element into a darray:

```
memcpy(ADDRESS_OF_NTH(darray->elements, n, darray->elemSize),
       newElem,
       darray->elemSize);
```

Here's the result after the macro expansion:

```
   memcpy((char *)darray->elements + n*darray->elemSize;,
          newElem,
          array->elemSize);
```

Look carefully at that errant semi-colon which appeared in the middle of our parameter list for memcpy. The macro definition ends with a semicolon (an easy mistake to make with since you're so used to ending all lines with semicolons) which will be included every time the macro is expanded. For some uses, this extra semicolon won't cause problems, but there are definitely legitimate uses of the macro which cannot tolerate the semicolon this macro is dragging along with it. The worst part about this is that you're likely to get very weird errors from the compiler complaining about the line where you make the macro call, which looks just fine in the source text. The fix here is, of course, to just remove the ending semicolon.

Macro expansion comes with a large variety of quirks due to syntax, parentheses, side-effects, repeated evaluation, etc., which makes it difficult to write robust macros. In general, it's best to avoid them for all but the most trivial uses.

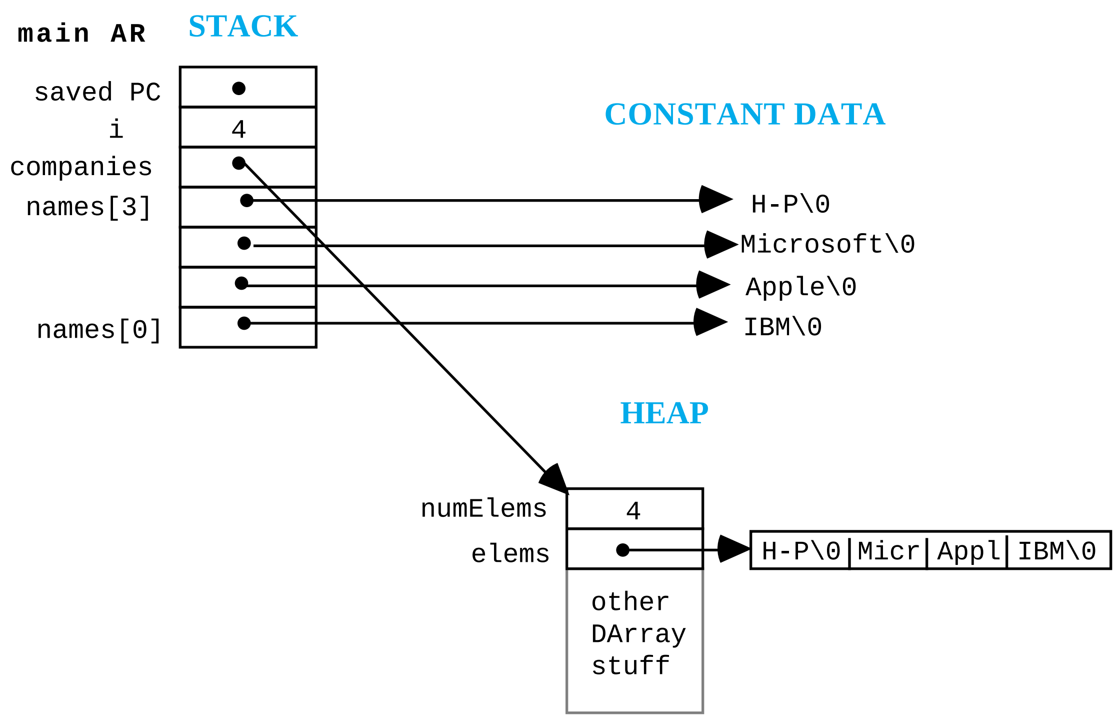
The compiler and linker

- When you call a function that has not yet been declared, the compiler flags it with the warning "Implicit declaration of function binky." It is just a warning and doesn't halt compilation. If you used the function correctly (i.e. called it with the correct number of arguments), then there won't be any problems at runtime, but if you didn't call it correctly, your code wasn't type-checked (since the compiler didn't see the prototype) and can have runtime problems if the arguments you passed were not correct.
- The compiler is satisfied if the call to a function matches its declaration, and it leaves a placeholder for the address of the function definition that the linker will resolve into one of the other modules or libraries. If no definition is found when linking, the linker halts with the fatal error "Undefined symbol binky".
- The compiler catches the same name reused within one file and complains with a "redefinition of binky" (or potentially a "conflicting types for binky" if the two definitions don't match). This is a fatal error. If the functions are in two separate files and not visible to each other when compiling, they each compile fine, but when the linker integrates the two modules, it will report the fatal error "symbol binky is multiply defined". It does not matter if the prototypes match, the linker doesn't even have access to type information, it just verifies there is one and only one definition of each `extern` symbol.
- The compiler does the type-checking to verify that a call to a function matches its declared prototype. If there is a mismatch for a function pointer parameter, it will give a warning about incompatible pointer types. Putting in a typecast will hush up the compiler and will let it compile "cleanly". However, at runtime, unpleasant results come from having the caller-side setup put a different arrangement of parameters

onto the stack than the called function pointer will expect to find there. Casting a function pointer is pretty much never a good idea.

Problem 12: Drawing and understanding memory

Here's a sketch of the state of memory right after the loop and before the call to `ArrayMap`:



The code prints:

```
H-P
MicrApplIBM
ApplIBM
IBM
```

As you should be well aware of by now, the `DArray` always refers to elements by address, so you pass pointers to elements to `ArrayAppend/Insert`, get pointers to elements from `ArrayNth`, and pointers to elements are passed as arguments to the comparator and map functions. This is always true even if the element itself is already a pointer.

Rather than correctly passing a `char**` when inserting, this code passes a `char*`. When the `DArray` copies 4 bytes from that location, it will then be copying the first four characters of the string, rather than copying the pointer. In the map callback function, there is a similar error in that the `void*` elem pointer is cast as a `char *`, not `char **`.

In some weird sense, two errors sort of cancel each other out. When we pass the pointer to `printf`, it prints characters from that address until it finds a null character. It makes a valiant attempt at printing the strings, but they only contain the first four characters and get all garbled together because the `NULL char` isn't found until the very end.