Midterm Exam Key

CSE 110

21 July 1992

1 Tracing

1.1 10 Points

Since the value of `name` is a pointer to the first element (the 5) of the array `name`, `p` now points to the first element.

Similarly, since the value of `name` is a pointer to an element of an array, `name + 1` points to the next element, so `q` now points to the second element, the 23.

So the first `printf` prints out `5 5 23`.

`*(p++)` says to get what `p` points to, throw it away, and then bump up the value of `p`. To bump up the value of a pointer means to make it point to the next element of the array, so `p` now points to the second element of `name`, the 23. The `*` here is a total red herring—`p++` alone would have done the same thing.

`(*q)++` says to get what `q` points to, throw it away, and then bump it up by 1. `q` was pointing to the second element of `name`, the 23, so that element gets bumped up to 24. Note that the value of `q` didn't change: it's still pointing to the same place; rather, the value stored in that place changed. The `*` here was not a red herring because `q++` means to bump up `q`, rather than what `q` points to.

In any case, Since `p` was pointing to the second element of the array `name`, which is now 24 instead of 23, `*p` is 24. `q[0]` is completely identical with `*(q + 0)` and so is synonymous with `*q`, whose value is 24.

So the second `printf` prints `24 24`.

1.2 10 Points

`t` starts at 4.

The condition in the `while` statement is **false** if `t-1` is zero, **true** otherwise. So the `while` loop will coninue until `t` becomes 1. It's not 1 yet, so we go into the loop.

We print `4`.

The condition in the `if` statement is **false** if `t%3` is zero, **true** otherwise.

`t%3` is the remainder when `t` is divided by 3, and so is 0 if and only if `t` is a multiple of 3. So when `t` is a multiple of 3, we take the `else` clause; otherwise we take the `if` clause.

First time through, `t` is not a multiple of 3, so we take the `if` part of the statement and `t` gets the value 9.

We print `9`.

Second time through, `t` is a umltiple of 3, so we take the `else` clause. `t` gets the value 3.

We print `3`.

Second time through, `t` is a umltiple of 3, so we take the `else` clause. `t` gets the value 1.

Now the `while` condition, `t-1`, is zero (**false**) so we exit the loop. that's the end of the program.

1.3 10 Points

`a` gets 5, `b` gets 23. We print `5 23`.

We call `scramble`. `x` gets 23 and `y` gets a pointer to `a`, whose value is 5.

Inside of `scramble`, we print `23 5`. Then `temp` gets 24, `x` gets 5, and `*y` gets 24. Since `y` is pointing to `a`, this actually changes the value of `a` to 24. We then print `5 24`. Then we return from `scramble`; `x` and `y` are destroyed.

We print the values of `a` and `b`. `a` was changed by `scramble` to 24, but `scramble` couldn't possibly change `b`, because it didn't know where `b` was, so `b` must still be 23. We print `24 23`.

We call `scramble`. `x` gets 24 and `y` gets a pointer to `b`, whose value is 23.

Inside of `scramble`, we print `24 23`. Then `temp` gets 25, `x` gets 23, and `*y` gets 25. Since `y` is pointing to `b`, this actually changes the value of `b` to 25. We then print `23 25`. Then we return from `scramble`; `x` and `y` are destroyed.

We print the values of `a` and `b`. `b` was changed by `scramble` to 25, but `scramble` couldn't possibly change `a`, because it didn't know where `a` was, so `a` must still be 24. We print `24 25`.

2 Writing Code

These are sample solutions; yours will of course be at least a little different.

2.1 10 Points

```
long int pow(int n, int p)
{
  long int power = 1;
```

```
  for ( ; p; p--)
    power *= n;

  return power;
}
```

I can't help but show you another solution which is less srtaightforard, but much, much faster. If p is 1,000, then the loop in the program above executes 1,000 times. But the loop in the program below gets the same answer and only executes 10 times.

```
long int pow2(int n, int p)
{
  /* Has value n^(2^i) the i'th time through the loop */
  long int np = n;

  long int answer = 1;

  while (p) {
    if (p%2)
      answer *= np;
    p = p/2;
    np *= np; /* np had n^(2^i), now has n^(2^(i+1)). */
  }

  return answer;
}
```

2.2 20 Points

```
void strrev(char *s)
{
  char *e;
  int left=0;                       /* Count of characters left to swap */
  char temp;                        /* For swap */

  /* If length of string is less than 2, don't bother.
   * Note short-circuiting here---it's VERY IMPORTANT.
   * If s has length 0, so that s[0] == '\0', it's ESSENTIAL that the
   * computer does NOT go and try to look at s[1] anyway, because that's
   * off the end of the array. */
  if (s[0] == '\0' || s[1] == '\0')
    return;

  /* e starts at string end, s at beginning.  We march s forward
     and e backwards, simultaneously, swapping the characters that
     e and s point to.  Thus the first character swaps with the last,
     the second with the next-to-last, and soforth. */

  /* First, point e at end of s and compute length of s: */

  for (e=s; *e != '\0'; e++)
```

```
    left++;
  /* e now points to NUL character at end of s. left is the number of
     characters we have to swap. */

  e--;
  /* e now points at last character in s. */


  while (left > 1) {            /* If there's only one character left,
                                   it's in the middle of the string anyway,
                                   so stop. */


    temp = *s; *s = *e; *e = temp; /* Swap characters at beginning and end */
    s++; e--;                      /* Move towards middle of string  */
    left -= 2;                     /* two fewer characters to swap. */
  }

}
```

2.3 20 Points

```
int get_int(int min, int max, char *prompt)
{
  int response;                         /* User's resposnse */
  int c;

  do {
    printf("%s", prompt);
    while (scanf("%d", &response) != 1)
      /* Discard non-numeric input up through next white space. */
      while ((c = getchar()) != ' ' && c != '\t' && c != '\n')
        /* nothing */ ;

    /* When we're out of the while loop, 'response' definitely has
       some integer value in it. */

  } while (response < min || response > max);

  return response;
}
```

3 Debugging

3.1 5 Points

The specific bug I had in mind was this: If getline returns the string `"%s %d %s %d %s %d %s %d''`, then printf will see the `%s` and `%d`s, and will expect to receive eight more arguments, four <pointer to `char`>s and four <`int`>s. They won't be there, of course, because we didn't pass them, and so `printf` will get garbage values instead. `printf` will print out garbage <`int`> values for the `%d`

conversions. The garbage <pointer to `char`>s will tell `printf` to go looking for string data in all sorts of unlikely and random places, some of which will be full of garbage and others of which will not exist and cause a program failure when `printf` tries to look at them.

The correct program has `printf("%s", s);` instead of `printf(s);`.

Although this was the answer I had in mind, I awarded credit for any reasonable bug that could possibly cause the behavior I described. One answer I thought of after I wrote the problem, but which nobody turned in, was that `getline` might be unable to get enough memory to store the input line, but it has to return a pointer anyway. If it returns a garbage pointer we're hosed. It *should* return `NULL` under these circumstances, but we failed to check for that, and so we're hosed that way too.

3.2 5 Points

It's better to write `NUMBER_OF_NOSTRILS` instead of putting 2's all over your program because it comunicates to the person reading the code just what the `2` is supposed to represent, and distinguishes it from other `2`'s such as those that represent completely different things, such as the number of rabbits in a brace. (`RABBITS_IN_BRACE`.

If you write `NUMBER_OF_NOSTRILS * AVERAGE_HAIRS_PER_NOSTRIL` it's obvious what you're computing; on the other hand, the number `1346` doesn't communicate anything.

3.3 5 Points

When a function is called, all its local variables, including the parameter variables that hold its arguments, are created from scratch and initialized with the appropriate values. Parameter variables are initialized with the values of the arguments the function was passed. So `var` here is created fresh when `set_to_57` is called, initialized with the argument that the caller passed in, and destroyed again when `set_to_57` returns. `var` is not the variable that the caller wanted to change; it is a different variable which is initialized with the same value.

Thus `p` points to `var` itself, and not to the variable that the caller wanted to change. The statement `*p = 57` changes the value of the variable tt var, but not the value of the variable that the caller wanted to change. When `set_to_57` returns, `var` is destroyed anyway, so this function still doesn't do anything.

3.4 5 Points

The problem here is very simple. Since `9` and `5` are both `<int>`s, the `/` in the expression `(9/5)` means integer division, and to drop fractions from the result. Thus `(9/5)` means 1, and not 1.8. To fix the problem, change `(9/5)` to `1.8`.