CS33 Homework Assignment 1 Solutions

Fall 2019

1. Arguments to C functions are "passed by value", meaning that the called function is given copies of its arguments. Thus, in the following code, *func1* is given a copy of the argument *x*.

```
int func1(int a) { a = a+2;
printf("a + 2 = %d\n", a);

return 1; } int main() {
     int x = 3;
     func1(x);
     if (x != 3)

   printf("shouldn't happen\n"); return 0; }
```

An alternative approach to passing arguments is to pass them "by reference" — the called function receives a reference to the variables provided by the caller. In our above example, if *x* were passed to *func1* by reference, whenever *func1* refers to *a*, it's really referring to *x*. If this were done, then *x*'s value would appear to be changed in *main* on return from *func1*.

As shown in class, we can achieve the effect of passing by reference by using pointers. Thus we might rewrite the above code as follows:

```
int func2(int *a) {
*a = *a+2; printf("*a + 2 =

%d\n", *a); return 1; } int

main() {
     int x = 3;
     func2(&x);
     if (x != 3)

   printf("should happen\n"); return 0; }
```

Yet another approach to passing arguments is to pass them by "value-result". This, as with pass by value, entails passing to the called function the current value of the argument. But on return from the function, the possibly modified value of the argument is passed back to the caller, updating the original argument.

Consider *func3* below.

```
int func3(int a1, int a2)
{
  a2 = 2; a1 =
1; return 1; }
```

a) One might ask whether calls to *func3* have the same effect if *func3's* arguments are passed by reference as they do if its arguments are passed by value-result. It turns out that there are cases in which the effects are different. Provide an example showing this; i.e., provide an example of passing arguments to func3 such that their values, after the call, are different depending on the argument-passing technique. You may assume that arguments are copied in the order in which they appear in the argument list.

Answer
:

```
int main()
{
     int x = 10;

     func3(x,
     x);

     printf("%d\n", x);


   return 0; }
```

If the arguments are passed by reference, what's printed is "1". But if the arguments are passed using value-result, what's printed is "2". (And in standard C, which uses pass by value, what would be printed is "10".)

C is sometimes adapted for use in remote procedure call (RPC) environments in which the calling function and the called function are on separate computers. Thus, when placing a call, the arguments are communicated to the called function over a network, the called function is computed on the remote machine, and any return values are transferred back to the caller over the network.

A problem with this approach is that, if implemented naively, pointers won't work: one passes a pointer to the remote machine, but the remote machine can't use it, since the value pointed to is on the calling machine. To get around this problem, rather than simply pass the pointer, the value the pointer points to is transferred to the remote machine. A new pointer is created on the remote machine that points to this copy of the transmitted value. When the remote procedure returns, the modified value is transmitted back to the caller, replacing what the caller's original pointer pointed to.

For example, looking back at the example containing *func2*, when it is called by *main*, the pointer to *x* is not passed to *func2*, but the value pointed to is (3). In *func2*, a new pointer (*a*) is created that points to the *3* that's been copied to the computer on which *func2* is running. What *a* points to is incremented by 2, and the modified value (5) is passed back to *main*. This modified value overwrites what the original pointer (*&x*) pointed to, thus *x*'s value is updated to be 5.

Consider *func4* below.

```
int func4(int a1[], int size1, int a2[], int size2) {

a1[0] = 8; a2[size2-1] = 8; return 1; }
```

If *func4* is implemented as a remote procedure, using a version of C that's been adapted as discussed above, then when it is called the arrays represented by its first and third arguments are transmitted over the network to its machine — they're effectively copied. On return from *func4*, the arrays are transmitted back to the caller, updating their original values.

b) Is it always the case that calling *func4* using standard C pass-by-value in a non-remote environment will provide the same result as calling it in a remote environment using a version of C adapted for RPC?

Answer
:

No. This is essentially the same question as was asked in part a. In normal C, the contents of an array is effectively passed by reference. But in the remote version, it's effectively passed by value-result. So consider the following calling function:

```
int main()
{
     int A[2] = {0, 1};

     func4(A, 2, A, 2);

     printf("%d, %d\n", A[0], A[1]);

   return 0; }
```

If the arguments are passed by value (as in normal C) in a non-remote environment, what's printed is "8, 8". But if the arguments are passed as described above in a remote environment, what's printed is "0, 8".