BGN: 2191A

P4

Q4

a. The C method does not store the length of the string. It instead stores all of the characters of the string in consecutive memory locations, followed by a null byte.

   One effect of this difference is that in C, strings cannot contain null bytes (as they would be interpreted as the end of the string rather than a part of it) whereas in BCPL they can.

   Another effect is that the “strlen” operation (or BCPL equivalent thereof) would be slower in C than in BCPL. In C this operation is O(n) where n is the length of the string, as the entire string has to be traversed until the null byte at the end is detected. Whereas, in BCPL, this operation is O(1), as the position where the length is stored is known, and so the length can be directly read from memory.

b. The callee should be responsible for allocating memory for the strings. Unless memory is an extremely scarce resource, the callee should make copies of the keys and values on the heap, and store in the dictionary the pair of char pointers to these copies.

   If the callee instead simply used the char pointers provided as function arguments, and the caller mutates its local copy of the key, then this would also mutate the corresponding key in the dictionary. Likewise with the values. This can lead to unexpected and difficult-to-debug side-effects.

c. This can make differences to compiler errors. For example, if assigning to the “mydata”, if it is declared as an array, you will get an error regarding the fact that array variables are not assignable.

   If “mydata” is only read from, and the program is not being linked with a translation unit which defines “mydata”, then in both cases you will see a linker error regarding an undefined reference to “mydata”.

   One difference in run-time operation is that sizeof(mydata) will return the number of bytes in the when using a pointer, and the number of elements when using an array.

   It would help code readability and maintainability to use a shared header file. In this file, the programmer could declare “mydata”, and in one of the translation units, define it. Then, all of the translation units could include this header, and the

programmer would have access to helpful IntelliSense which in many cases will inform them if they are using the variable incorrectly.

d. Assuming ints are 32-bit, doubles are 64-bit, pointers are 32-bit

    1. At run-time, i1 would contain an int whose value is f1 rounded towards zero (i.e., floor if f1 is potivie, ceil if f1 is negative). The compiler would explicitly perform the double to int conversion, shifting the mantissa to the left according to the exponent, and discarding carry bits, storing the result in i1.

    2. At compile time, the address of i1 would simply be copied into the pointer variable p2. At run time, p2 would be a pointer to a character whose ASCII value is the left-most (most-significant if big-endian, least-significant if little-endian) byte of i1.

    3. At compile time, the char stored at the address pointed to by p2 (which happens to be the same address as the left-most byte of i1) would be dereferenced. The dereferenced byte would be padded with 3 additional bytes (padding on the left if big-endian, on the right if little-endian). These bytes would be 0x00 if the first bit of the dereferenced byte is 0, and 0xFF if the first bit is 1. This padded sequence of bytes would be stored on the stack as the variable i3. At run time, i3 will have the same value as the left-most (most-significant if big-endian, least-significant if little-endian) byte of i1.