Problem Set 1 Solutions

Problem 0

To truncate a C string, we simply need to insert a terminating null character before the end of the string. This can be done as follows:

```
void TruncateString(char* source, int index)
{
    source[index] = '\0';
}
```

Problem 1

`StringDuplicate` needs to allocate a buffer capable of holding the entire string (plus the terminating null!), then must copy over the source string. Here's one solution:

```
char* StringDuplicate(char* source)
{
    char* result = new char[strlen(source) + 1];
    strcpy(result, source);
    return result;
}
```

Note that writing `*result = *source` in place of `strcpy(result, source)` will not work as expected because it will only copy over the first character in the string.

Problem 2

Here are two versions of `CountFrequency`:

```
int CountFrequency(char* source, char toFind)
{
    int frequency = 0, length = strlen(source);
    for(int k = 0; k < length; ++k)
        if(source[k] == toFind) ++frequency;
    return frequency;
}

int CountFrequency(char* source, char toFind)
{
    int frequency = 0;
    for(; *source != '\0'; ++source)
        if(*source == toFind) ++frequency;
    return frequency;
}
```

For those of you planning on working on professional C code, there's a slightly more cryptic version of the second implementation of this function that relies on a few quirks of C and C++. First, nonzero

values evaluate to false, so `while(*ptr != '\0')` and `while(*ptr)` are equivalent. Second, it's possible to combine the `++` and `*` operators into a single pointer operation. While horridly cryptic, the statement `*ptr++` is interpreted by the C++ compiler as `*(ptr++)`. The `ptr++` increments `ptr`, then returns the original value of `ptr`. Thus `*ptr++` means "return the value pointed at by `ptr`, then increment `ptr` by one step." It is therefore possible to rewrite the second version of this function as

```
int CountFrequency(char* source, char toFind)
{
    int result = 0;
    while(source)
        if(*source++ == toFind) ++result;
    return result;
}
```

I do not encourage writing code in this style, since it's both dense and cryptic. However, you might encounter it in practice, so it's good to know about.

Problem 3

Writing `GetSubstring` is easy if you have `strncpy`, but is a bit trickier if you have to code it by hand. Here's one solution:

```
char* GetSubstring(char* source, int start, int length)
{
    char* result = new char[length + 1];
    for(int k = 0; k < length; ++k)
        result[k] = source[k + start];
    result[length] = '\0';
    return result;
}
```

In practice, because most of the C string manipulation routines are written in lightning-fast assembly code, you should almost certainly use `strncpy` to accomplish this instead of a hand-written `for` loop.

Problem 4

While there are many solutions to this problem, I personally find this one most attractive:

```
bool StrCaseEqual(char* str1, char* str2)
{
    /* Keep looping until we hit the end of either string. */
    for(; *str1 != '\0' && *str2 != '\0'; ++str1, ++str2)
        if(tolower(*str1) != tolower(*str2))
            return false;

    /* The strings are not equal unless we hit the end of both strings. */
    return *str1 == '\0' && *str2 == '\0';
}
```

Common errors on this problem included using `strlen` (no library functions allowed!) and improperly handling the case where the strings are different lengths.

Problem 5

`CreateRepetitiveString` can be implemented as follows:

```
char* CreateRepetitiveString(char ch, int numCopies)
{
    char* result = new char[numCopies + 1]; // Need space for null!
    memset(result, ch, numCopies);
    result[numCopies] = '\0';
    return result;
}
```

Problem 6

The modified version of this code would be written as follows:

```
void TruncateString(char* source, int index)
{
    assert(index >= 0 && index <= strlen(source));
    source[index] = '\0';
}
```

A common C/C++ technique is to use unsigned integers (integers whose values cannot be negative) to keep track of the lengths of strings and lists. If we use an unsigned integer as the second parameter to the function (or, better, the special type `size_t` exported by `<cstddef>`), we can eliminate the check `index >= 0`.

Problem 7

The car drives off a cliff because `InitCriticalInfrastructure` wasn't called. `assert` statements are typically disabled in release builds to improve efficiency, so if you insert code with side effects into an `assert` statement you can run into trouble. There are two ways to fix this. First, you can manually check the result of the function, as shown here:

```
if(!InitCriticalInfrastructure())
    abort();
```

Alternatively, if you're *absolutely sure* that `InitCriticalInfrastructure` will never fail, you can separate out the call to `InitCriticalInfrastructure` and the check in the `assert` as follows:

```
bool result = InitCriticalInfrastructure();
assert(result);
```

As an interesting aside, most imperative languages like C++ do not have a way of controlling side effects in functions – that is, any function can have side effects besides just returning a value. This makes writing a good `assert` function tricky since it's impossible to tell if the argument causes side effects. However, other languages like Haskell have the idea of side effects encoded into the return types of functions, so it's possible to check at compile-time that you aren't making this sort of mistake. In C++, however, programmer vigilance is your only safeguard.