Chapter 7 Review Questions

1. The three steps involved in using a function are:
    a. Providing a function definition
    b. Providing a function prototype
    c. Calling the function

2. Function definitions:
    a. `void igor();`
    b. `float tofu(int foo);`
    c. `double (mpg (double foo, double bar);`
    d. `long summation (long foo[], int size);`
    e. `double doctor (const string foo);`
    f. `void ofcourse (boss foo);`
    g. `string plot(map *foo);`

3.
```
void foo (int bar[], int size, int value)
{
   for (int i=0; i < size; i++)
        bar[i] = value;
}
```

4.
```
void foo (int * begin, int * end, int value)
{
   for (int * i = begin; i != end; i++)
        *i = value;
}
```

5.
```
double foo (const double name[], int size)
{
   double largestVal = 0;

   for (int i = 0; i < size; i++)
        if (name[i] > largestVal)
             largestVal = name[i];

   return largestVal;
}
```

6. For fundamental types (e.g., double, int), c++ only passes a copy of the value to the function, not the original value itself. As a result, the original value is already protected, making the `const` qualifier redundant.

7. A c-style string can take three forms:
    a. An array of char
    b. A quoted string constant (also called a *string literal*)
    c. A pointer-to-char set to the address of the string

8.
```
int replace (char * str, char c1, char c2)
{
   int replacements = 0;

   while (*str)
   {
        if (*str == c1)
        {
             *str = c2;
             replacements++;
        }
        str++;
   }
   return replacements;
}
```

9. String constants behave the same as array names. Thus, * “pizza” returns the value of the first element in the array or, “p”. Furthermore, since “taco” would return “t”, “taco”[2] returns “c”.

10. A structure, `glitz`, could be passed by value as follows:

```
void foo (glitz bar); // function prototype
foo (bar); // function call
```

The same structure could be passed by address as follows:

```
void foo (glitz *bar); // function prototype
foo (&bar); // function call
```

Structures are treated similar to fundamental types, in that C++ only passes a copy of the structure to the function. Thus the values in the original structure are protected, but creating this copy takes extra time. Passing by address saves time but forgoes the protection that passing by value offers. However, one can use the const modifier to protect the structure as follows:

```
void foo (const glitz *bar); // function prototype
foo (&bar); // function call
```