void print(const char *str); // #5

When you then use a `print()` function, the compiler matches your use to the prototype that has the same signature:

```c
print("Pancakes", 15); // use #1
print("Syrup"); // use #5
print(1999.0, 10); // use #2
print(1999, 12); // use #4
print(1999L, 15); // use #3
```

For example, `print("Pancakes", 15)` uses a string and an integer as arguments, and that matches prototype #1.

When you use overloaded functions, be sure you use the proper argument types in the function call. For example, consider the following statements:

```c
unsigned int year = 3210;
print(year, 6); // ambiguous call
```

Which prototype does the `print()` call match here? It doesn't match any of them! A lack of a matching prototype doesn't automatically rule out using one of the functions, for C++ will try to use standard type conversions to force a match. If, say, the only `print()` prototype were #2, the function call `print(year, 6)` would convert the `year` value to type `double`. But in the code above there are three prototypes that take a number as the first argument, providing three different choices for converting `year`. Faced with this ambiguous situation, C++ rejects the function call as an error.

Some signatures that appear different from each other can't coexist. For example, consider these two prototypes:

```c
double cube(double x);
double cube(double & x);
```

You might think this is a place you could use function overloading, for the function signatures appear to be different. But consider things from the compiler's standpoint. Suppose you have code like this: