After preprocessing, this declaration becomes

```
int i1, i2, i3;
```

The ## operator isn't one of the most frequently used features of the preprocessor; in fact, it's hard to think of many situations that require it. To find a realistic application of ##, let's reconsider the MAX macro described earlier in this section. As we observed then, MAX doesn't behave properly if its arguments have side effects. The alternative to using the MAX macro is to write a max function. Unfortunately, one max function usually isn't enough; we may need a max function whose arguments are `int` values, one whose arguments are `float` values, and so on. All these versions of max would be identical except for the types of the arguments and the return type, so it seems a shame to define each one from scratch.

The solution is to write a macro that expands into the definition of a max function. The macro will have a single parameter, `type`, which represents the type of the arguments and the return value. There's just one snag: if we use the macro to create more than one max function, the program won't compile. (C doesn't allow two functions to have the same name if both are defined in the same file.) To solve this problem, we'll use the ## operator to create a different name for each version of max. Here's what the macro will look like:

```
#define GENERIC_MAX(type)          \
type type##_max(type x, type y)    \
{                                  \
  return x > y ? x : y;            \
}
```

Notice how `type` is joined with `_max` to form the name of the function.

Suppose that we happen to need a max function that works with `float` values. Here's how we'd use GENERIC_MAX to define the function:

```
GENERIC_MAX(float)
```

The preprocessor expands this line into the following code:

```
float float_max(float x, float y) { return x > y ? x : y; }
```

General Properties of Macros

Now that we've discussed both simple and parameterized macros, let's look at some rules that apply to both:

- *A macro's replacement list may contain invocations of other macros.* For example, we could define the macro TWO_PI in terms of the macro PI:

  ```
  #define PI      3.14159
  #define TWO_PI  (2*PI)
  ```

  When it encounters TWO_PI later in the program, the preprocessor replaces it by `(2*PI)`. The preprocessor then *rescans* the replacement list to see if it