The # Operator

Macro definitions may contain two special operators, # and ##. Neither operator is recognized by the compiler; instead, they're executed during preprocessing.

Q&A The # operator converts a macro argument into a string literal; it can appear only in the replacement list of a parameterized macro. (The operation performed by # is known as "stringization," a term that I'm sure you won't find in the dictionary.)

There are a number of uses for #; let's consider just one. Suppose that we decide to use the `PRINT_INT` macro during debugging as a convenient way to print the values of integer variables and expressions. The # operator makes it possible for `PRINT_INT` to label each value that it prints. Here's our new version of `PRINT_INT`:

```
#define PRINT_INT(n) printf(#n " = %d\n", n)
```

The # operator in front of `n` instructs the preprocessor to create a string literal from `PRINT_INT`'s argument. Thus, the invocation

```
PRINT_INT(i/j);
```

will become

```
printf("i/j" " = %d\n", i/j);
```

We saw in Section 13.1 that the compiler automatically joins adjacent string literals, so this statement is equivalent to

```
printf("i/j = %d\n", i/j);
```

When the program is executed, `printf` will display both the expression `i/j` and its value. If `i` is 11 and `j` is 2, for example, the output will be

```
i/j = 5
```

The ## Operator

The ## operator can "paste" two tokens (identifiers, for example) together to form a single token. (Not surprisingly, the ## operation is known as "token-pasting.") If one of the operands is a macro parameter, pasting occurs after the parameter has been replaced by the corresponding argument. Consider the following macro:

```
#define MK_ID(n) i##n
```

When `MK_ID` is invoked (as `MK_ID(1)`, say), the preprocessor first replaces the parameter `n` by the argument (`1` in this case). Next, the preprocessor joins `i` and `1` to make a single token (`i1`). The following declaration uses `MK_ID` to create three identifiers:

```
int MK_ID(1), MK_ID(2), MK_ID(3);
```