Next: Function-like Macros, Up: Macros

3.1 Object-like Macros

An *object-like macro* is a simple identifier which will be replaced by a code fragment. It is called object-like because it looks like a data object in code that uses it. They are most commonly used to give symbolic names to numeric constants.

You create macros with the '`#define`' directive. '`#define`' is followed by the name of the macro and then the token sequence it should be an abbreviation for, which is variously referred to as the macro's *body*, *expansion* or *replacement list*. For example,

```
#define BUFFER_SIZE 1024
```

defines a macro named `BUFFER_SIZE` as an abbreviation for the token `1024`. If somewhere after this '`#define`' directive there comes a C statement of the form

```
foo = (char *) malloc (BUFFER_SIZE);
```

then the C preprocessor will recognize and *expand* the macro `BUFFER_SIZE`. The C compiler will see the same tokens as it would if you had written

```
foo = (char *) malloc (1024);
```

By convention, macro names are written in uppercase. Programs are easier to read when it is possible to tell at a glance which names are macros.

The macro's body ends at the end of the '`#define`' line. You may continue the definition onto multiple lines, if necessary, using backslash-newline. When the macro is expanded, however, it will all come out on one line. For example,

```
#define NUMBERS 1, \
                2, \
                3
int x[] = { NUMBERS };
     ==> int x[] = { 1, 2, 3 };
```

The most common visible consequence of this is surprising line numbers in error messages.

There is no restriction on what can go in a macro body provided it decomposes into valid preprocessing tokens. Parentheses need not balance, and the body need not resemble valid C code. (If it does not, you may get error messages from the C compiler when you use the macro.)

The C preprocessor scans your program sequentially. Macro definitions take effect at the place you write them. Therefore, the following input to the C preprocessor

```
foo = X;
#define X 4
bar = X;
```

produces

```
foo = X;
bar = 4;
```

When the preprocessor expands a macro name, the macro's expansion replaces the macro invocation, then the expansion is examined for more macros to expand. For example,

```
#define TABLESIZE BUFSIZE
#define BUFSIZE 1024
TABLESIZE
     ==> BUFSIZE
     ==> 1024
```

`TABLESIZE` is expanded first to produce `BUFSIZE`, then that macro is expanded to produce the final result, `1024`.

Notice that `BUFSIZE` was not defined when `TABLESIZE` was defined. The '`#define`' for `TABLESIZE` uses exactly the expansion you specify—in this case, `BUFSIZE`—and does not check to see whether it too contains macro names. Only when you *use* `TABLESIZE` is the result of its expansion scanned for more macro names.

This makes a difference if you change the definition of `BUFSIZE` at some point in the source file. `TABLESIZE`, defined as shown, will always expand using the definition of `BUFSIZE` that is currently in effect:

```
#define BUFSIZE 1020
#define TABLESIZE BUFSIZE
#undef BUFSIZE
#define BUFSIZE 37
```

Now `TABLESIZE` expands (in two stages) to `37`.

If the expansion of a macro contains its own name, either directly or via intermediate macros, it is not expanded again when the expansion is examined for more macros. This prevents infinite recursion. See Self-Referential Macros, for the precise details.