The # Operator

- Macro definitions may contain two special operators, `#` and `##`
- Neither operator is recognized by the compiler; instead, they are executed during preprocessing
- The `#` operator
  - converts a macro argument into a string literal
  - can appear only in the replacement list of a parameterized macro
- The operation performed by `#` is known as "*stringization*"

The # Operator

- There are a number of uses for `#`; let us 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

The # Operator

- Our new version of `PRINT_INT`:

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

- The invocation

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

  will become

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

- The compiler automatically joins adjacent string literals, so this statement is equivalent to

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


The ## Operator

- The `##` operator can "*paste*" two tokens together to form a single token
- If one of the operands is a macro parameter, pasting occurs after the parameter has been replaced by the corresponding argument

The ## Operator

- A macro that uses the ## operator:

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

- A declaration that invokes MK_ID three times:

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

- The declaration after preprocessing:

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


General Properties of Macros

Several rules apply to both simple and parameterized macros

- ***A macro's replacement list may contain invocations of other macros***

  Example:

```
#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 contains invocations of other macros

General Properties of Macros

- ***The preprocessor replaces only entire tokens***

  Macro names embedded in *identifiers*, *character constants*, and *string literals* are ignored

  Example:

```
#define SIZE 256

int BUFFER_SIZE;

if (BUFFER_SIZE > SIZE)
  puts("Error: SIZE exceeded");
```

  Appearance after preprocessing:

```
int BUFFER_SIZE;

if (BUFFER_SIZE > 256)
  puts("Error: SIZE exceeded");
```


General Properties of Macros

- ***A macro definition normally remains in effect until the end of the file in which it appears***

  Macros do not obey normal scope rules

  A macro defined inside the body of a function is not local to that function; it remains defined until the end of the file

- ***A macro may not be defined twice unless the new definition is identical to the old one***

  Differences in spacing are allowed, but the tokens in the macro's replacement list (and the parameters, if any) must be the same

General Properties of Macros

- ***Macros may be "undefined" by the* `#undef` *directive***

  The `#undef` directive has the form

  `#undef` *identifier*

  where *identifier* is a macro name

  One use of `#undef` is to remove the existing definition of a macro so that it can be given a new definition

Parentheses in Macro Definitions

- The replacement lists in macro definitions often require parentheses in order to avoid unexpected results
  - If the macro's replacement list ***contains an operator***, always *enclose the replacement list in parentheses*:

    ```
    #define TWO_PI (2*3.14159)
    ```
  - Also, *put parentheses* ***around each parameter*** every time it appears in the replacement list:

    ```
    #define SCALE(x) ((x)*10)
    ```
- Without the parentheses, we can not guarantee that the compiler will treat replacement lists and arguments as whole expressions

Parentheses in Macro Definitions

- An example that illustrates the need to put parentheses around a macro's replacement list:

```
#define TWO_PI 2*3.14159
/* needs parentheses around replacement list */
```

- During preprocessing, the statement

```
conversion_factor = 1/TWO_PI;
```

becomes

```
conversion_factor = 1/2*3.14159;
```

The division will be performed before the multiplication and the end result will be storing `ZERO` in `conversion_factor`

Parentheses in Macro Definitions

- Each occurrence of a parameter in a macro's replacement list needs parentheses as well:

```
#define SCALE(x) (x*10)
  /* needs parentheses around x */
```

- During preprocessing, the statement

```
j = SCALE(i+1);
```

becomes

```
j = (i+1*10);
```

This statement is equivalent to

```
j = i+10;
```

Conditional Compilation

- The C preprocessor recognizes a number of directives that support ***conditional compilation***
- This feature permits the inclusion or exclusion of a section of program text depending on the outcome of a ***test performed by the preprocessor***

The `#if` and `#endif` Directives

- Suppose we are in the process of debugging a program
- We would like the program to print the values of certain variables, so we put calls of `printf` in critical parts of the program
- Once we have located the bugs, it is often a good idea to let the `printf` calls remain, just in case we need them later
- Conditional compilation allows us to leave the calls in place, but have the compiler ignore them

The `#if` and `#endif` Directives

- The first step is to define a macro and give it a nonzero value:

```
#define DEBUG 1
```

- Next, we will surround each group of `printf` calls by an `#if` `#endif` pair:

```
#if DEBUG
printf("Value of i: %d\n", i);
printf("Value of j: %d\n", j);
#endif
```


The `#if` and `#endif` Directives

- During preprocessing, the `#if` directive will test the value of `DEBUG`
- Since its value is not zero, the preprocessor will leave the two calls of `printf` in the program
- If we change the value of `DEBUG` to zero and recompile the program, the preprocessor will remove all four lines from the program
- The `#if` `#endif` blocks can be left in the final program, allowing diagnostic information to be produced later if any problems turn up

The `#if` and `#endif` Directives

- General form of the `#if` and `#endif` directives:

  `#if` *constant-expression*
  …
  `#endif`
- When the preprocessor encounters the `#if` directive, it evaluates the *constant-expression*
- The `#if` directive treats *undefined identifiers* as macros that have the value `0`
- If the value of the expression is `0`, the lines between `#if` and `#endif` will be removed from the program during preprocessing
- Otherwise, the lines between `#if` and `#endif` will remain

The `defined` Operator

- The preprocessor supports three operators: `#`, `##`, and `defined`
  - `#` and `defined` are unary operators
  - `##` is a binary operator
- When applied to an identifier, `defined` produces the value `1` if the identifier is a currently defined macro; it produces `0` otherwise
- The `defined` operator is normally used in conjunction with the `#if` directive

The `defined` Operator

- Example:

```
#if defined(DEBUG)
…
#endif
```

What is the difference between:
`#if defined(DEBUG)`
and
`#if DEBUG`

- The lines between `#if` and `#endif` will be included only if `DEBUG` is defined as a macro
- It is not necessary to give `DEBUG` a value:

```
#define DEBUG
```

- The parentheses around `DEBUG` are *not required*:

```
#if defined DEBUG
```


The `#ifdef` and `#ifndef` Directives

- The `#ifdef` directive tests whether an identifier is currently defined as a macro:

  `#ifdef` *identifier*

- The effect is the same as

  `#if defined(`*identifier*`)`

- The `#ifndef` directive tests whether an identifier is *not* currently defined as a macro:

  `#ifndef` *identifier*

- The effect is the same as

  `#if !defined(`*identifier*`)`

The `#elif` and `#else` Directives

- `#if`, `#ifdef`, and `#ifndef` blocks can be nested just like ordinary `if` statements
- When nesting occurs, it is a good idea to use an increasing amount of indentation as the level of nesting grows
- Some programmers put a comment on each closing `#endif` to indicate what condition the matching `#if` tests:

```
#if DEBUG
…
#endif /* DEBUG */
```


The `#elif` and `#else` Directives

- `#elif` and `#else` can be used in conjunction with `#if`, `#ifdef`, or `#ifndef` to test a series of conditions:

  `#if` *expr1*
  Lines to be included if expr1 is nonzero
  `#elif` *expr2*
  Lines to be included if expr1 is zero but expr2 is nonzero
  `#else`
  Lines to be included otherwise
  `#endif`

- Any number of `#elif` directives—but at most one `#else`—may appear between `#if` and `#endif`

Uses of Conditional Compilation

- Conditional compilation has other uses besides debugging
  - ***Writing programs that are portable to several machines or operating systems***

    Example:

```
#if defined(WIN32)
…
#elif defined(MAC_OS)
…
#elif defined(LINUX)
…
#endif
```


Uses of Conditional Compilation

- ***Conditional compilation makes it possible to check whether a macro is currently defined and, if not, give it a default definition:***

```
#ifndef BUFFER_SIZE
#define BUFFER_SIZE 256
#endif
```

Uses of Conditional Compilation

– ***Temporarily disabling code that contains comments***

A `/*...*/` comment can not be used to "*comment out*" code that already contains `/*...*/` comments

An `#if` directive can be used instead:

```
#if 0
Lines containing comments
#endif
```