/****** Steve Vegdahl (vegdahl@ogicse.ogi.edu) ******/

Possible advantages of using macros:

- You can define generic operations (macro are not type-checked), as in

  ```
  #define next2(ptr)  ptr->next->next
  ```

  which will work on any pointer that points to a struct containing a next field of the same pointer-type.

- Generally leads to faster code due to the lack of function-call overhead.

- You can pass things other than expression-values to macro, including types, labels, other macros, statements, and return such things as a macro's "value." For example, you can define a macro to 'malloc' a block of memory, including doing the proper type-casting with something like

  ```
  #define      tmalloc(typeName)   ((typeName *)malloc(sizeof(typeName)))
  ```

- You can define a macro that side-effects a variable without having to use the '&' operator, as in

  ```
  #define      nextPtr(ptr) (ptr = ptr->next)
  ```

  A function to do the same operation would have to be invoked as

  ```
  nextPtr(&myPtr)
  ```

  and would not be generic. (On the other hand, things that look like functions, but perform non-apparent side-effects can also be confusing.)

- A macro that contains errors (e.g. syntax) need not be commented out as long as it is unused.

- Macros in some contexts where functions cannot, as in:

  ```
  switch (abs)
  {
  case foo(2, 6):           /* foo must be a macro here      */
        ...
  }
  ```

Disadvantages of using macros include:

- No checking (e.g. syntax, type-checking) is done at the macro definition, but only at its invocation. This can lead to some pretty non-intuitive error-messages by the time the compiler figures out what's going on. Additionally, arguments promotions are not done for macros.

- Code is often bulkier because it is expanded in-line.

- You cannot pass a pointer to a macro like you can to a function.

- In many debuggers, you cannot invoke a macro. If you have defined a function to print out a data object in some readable way, you can generally invoke it during debugging; had a macro been used to print the data object instead, you may be out of luck.

- The scope rules of macros are different than for functions.
  - a function name can be overridden by a declaration in an inner scope
  - if a function refers to a "free" name, within its body, the definition of the name that used is the definition that is in force during the function's definition. If a macro refers to a name within its definition, the name definition of the name that is used is that which is in force during the macro's definition. Thus, in

```
int    i;
#define      iAddMacro(n) (i += n)
int    iAddFunc(int n)     { return i+= n;     }
void   myFunc(void)
{
       int    i;
       iAddMacro(4);/* adds 4 to local var 'i' */
       iAddFunc(4); /* adds 4 to global var 'i'      */
}
```

  Thus, when a macro is used that refers to a free variable, you take the risk that the user of the macro might inadvertently use that name.

- Precedence rules are different for macro calls and function calls:

```
#define      times(a, b)  a * b
x = times(i + j, k) /* expands into 'i+j*k' == 'i+(j*k)'    */
```

  C-macro uses are advised to avoid such problems by parenthesizing expressions involving macroparemeters, as in:

```
#define      times(a, b)  ((a) * (b))
```

- When a macro expands into a sequence of statements, it is possible that the trailing statements might "dangle", as in:

```
#define      incIandJ()   i++; j++
if (a > b) incIandJ();    /* expands into 'if (a>b) i++; j++      */
```

  where the 'j++' is outside the scope of the if-statement. It doesn't work to "fix" it like:

```
#define      incIandJ()   { i++; j++; }
```

  because

```
if (a > b) incIandJ(); else k++;
```

  will then expand into

```
if (a > b) { i++; j++; }; else k++;
```

  causing a syntax error at the 'else' due to the extra ';'. A correct (but somewhat kludgy) solution to this kind of problem was posted to this newsgroup several months ago by someone:

```
#define      incIandJ()   do { i++; j++ } while (0)
```

  This is a statement-still-expecting-semicolon, that executes both increment-statements exactly once.

- If a macro expands into code that contains one or more goto-labels, these labels-name could conflict those already in function (e.g. if such macro is invoked twice within the same function).

- If an formal parameter is used more than once in the same macro, and the side-effects, the side-effect may be applied twice, as in:

```
#define      max(a, b)    (a>b) ? a : b
max(x++, y++);            /* either 'x' or 'y' will be incremented
                             twice (behavior is undefined?)*/
```

  Similarly, a macro might cause one of its operands to remain unevaluated, as in:

```
#define      check(a, b)  { if (a > 0) a += b; }
check(0, z++);            /* 'z' is not incremented here          */
```

- A macro whose body contains a statement (e.g. to implement a loop) may not return a value.

- A macro cannot be defined that takes a variable number of arguments.

- Recursive macros will not work as you might expect. The ANSI C standard requires that macro expansions not be done for any call that is recursive. (presumably to prevent the macro-expander from performing infinite recursion).

- If a macro-definition spans multiple lines, you need to type (a stupid) '\' in order for your macro to continue.

- If you change a function definition, you generally only have to recompile the file where the function is defined (unless the function's interface changes). If you change a macro definition, you generally have to recompile all the files that use the macro.

```
/****** David Fox (fox@cs.nyu.edu)                    ******/
```

typedef vs enum

- enum improves readability. Using enum makes it clear that the constant (e.g. TRUE and FALSE) are related. The constants are defined in one spot. If, later on, we want to allow for MAYBE, we can just add it to our list (e.g. {TRUE, MAYBE, FALSE}) without having to stop and think what value MAYBE should have.

- The compiler can enforce strong type checking with enumerated types. We can enlist the compiler's help to make sure that we don't assign some nonsense value or magic number to a flag that should always be TRUE or FALSE.

- Enumerated types have the advantage of being symbols, available to the debugger, whereas #define never even make it to the compiler.