Defining a C Function that has a Varying Number of Arguments

Under ANSI C, a function with a varying number of arguments is defined by using a trailing ellipsis (`...`) in the argument list, declaring that there may be additional arguments, number and type unspecified. The ellipsis can only be used if there is at least one named argument. All named arguments must precede the ellipsis. For a function defined in this fashion, argument data types must be specified in-line; e.g.,

```
    int fnv(int x, ...);
```

[the data type for the named argument “`x`” is specified “in-line”]

There is no built-in mechanism for determining the data type of an argument passed to the function via the varying list. The programmer must devise means (such as the format string used by `printf`) to pass in this information. Moreover, the only way to reliably determine how many arguments are in the list is for the calling program to pass it in, either explicitly (as with the `printf` format string) or implicitly via a delimiter argument (a typical delimiter value used is `NULL`).

Preprocessor macros included from `<stdarg.h>` are used to access the argument list for this kind of function. The macros are:

- `va_list`
- `va_start`
- `va_arg`
- `va_end`

The basic approach is to use `va_arg` for traversing the argument list. `va_list` and `va_start` are for set-up. `va_end` is a compiler dependent means for assuring a clean finish for the function.

Example macro usage:

1. The macro call
   ```
   va_list lptr;
   ```
   sets up a `typedef` for a pointer variable named `lptr` for argument lists.

2. The macro call
   ```
   va_start(lptr,last-named);
   ```
   points `lptr` to the first un-named argument (assuming that "`last-named`" is the last named argument before the ellipsis).

3. The loop structure
   ```
   while ((val=va_arg(lptr, int)) != NULL)
   ```
   captures each parameter value in `val` as type `int`. `va_arg` advances `lptr` along the argument list each time it is called. Note that the data type to apply to the current argument is specified in the call to `va_arg`. The loop ends when a `NULL` input is encountered, so the calling program must have a `NULL` argument delimiting its list of input values. In contrast, the `printf` format string implicitly provides a counting mechanism, avoiding the need to require a delimiting argument.

4. Finally,
   ```
   va_end(lptr);
   ```
   may need to be called before finishing up. On some systems, if `va_end` is not called, the result of the function is undefined.

Example: (full function)

```
/****************************************************/
/*          University of North Florida             */
/* Department of Computer and Information Sciences */
/*         Charles N. Winton - 10/22/2001           */
/* bits function, variable number of arguments      */
/* for setting specified bits of integer x          */
/* NOTES:                                           */
/*   1. usage format: x = bits(x,<sr>,<list>);      */
/*   2. <sr>='s' sets to 1  <sr>='r' resets to 0    */
/*   3. bit positions listed in order from bit 0    */
/*   4. a trailing 0 is needed to delimit           */
/*   5. <stdarg.h> is discussed in K&R p. 155-56    */
/****************************************************/

#include <stdarg.h> /* library of macros for "..." */

int bits(unsigned int x, char sr, ...)
  {
    int a, b=0;

    va_list listptr;       /* declare list pointer */
    va_start(listptr, sr); /* start after var sr   */

    /* step through list: cover case of initial 0  */
    b+=1<<va_arg(listptr,int);
    while ((a=va_arg(listptr,int)) != 0)
      b+=1<<a;            /* use shift to get 2^a  */

    va_end(listptr);      /* clean up for return   */

    if (sr == 's') return(x|b);
    if (sr == 'r') return(x&~b);
    else return(x);
  }
```

As an example of typical usage, the function call

```
    x = bits(x,'s',0,3,17,0);
```

to set bit positions 0, 3, and 17 of variable `x` to 1, accomplishes this by calculating `b`$=2^0+2^3+2^{17}$ and then ORing `x` with `b` [*x*|*b*].