9.9. Variable numbers of arguments

It is often desirable to implement a function where the number of arguments is not known, or is not constant, when the function is written. Such a function is `printf`, described in Section 9.11. The following example shows the declaration of such a function.

```
int f(int, ... );

int f(int, ... ) {
        .
        .
        .
}

int g() {
        f(1,2,3);
}
```

Example 9.5

In order to access the arguments within the called function, the functions declared in the `<stdarg.h>`header file must be included. This introduces a new type, called a `va_list`, and three functions that operate on objects of this type, called`va_start`, `va_arg`, and `va_end`.

Before any attempt can be made to access a variable argument list,`va_start` must be called. It is defined as

```
#include <stdarg.h>
void va_start(va_list ap, parmN);
```

The `va_start` macro initializes `ap` for subsequent use by the functions`va_arg` and `va_end`. The second argument to `va_start`, *parmN* is the identifier naming the rightmost parameter in the variable parameter list in the function definition (the one just before the , ...). The identifier *parmN* must not be declared with `register` storage class or as a function or array type.

Once initialized, the arguments supplied can be accessed sequentially by means of the `va_arg`macro. This is peculiar because the type returned is determined by an argument to the macro. Note that this is impossible to implement as a true function, only as a macro. It is defined as

```
#include <stdarg.h>
type va_arg(va_list ap, type);
```

Each call to this macro will extract the next argument from the argument list as a value of the specified type. The `va_list` argument must be the one initialized by `va_start`. If the next argument is not of the specified type, the behaviour is undefined. Take care here to avoid problems which could be caused by arithmetic conversions. Use of `char` or short as the second argument to `va_arg` is invariably an error: these types always promote up to one of `signed int` or `unsigned int`, and `float` converts to `double`. Note that it is implementation defined whether objects declared to have the types `char`, `unsigned char`, `unsigned short` and unsigned bitfields will promote to `unsigned int`, rather complicating the use of `va_arg`. This may be an area where some unexpected subtleties arise; only time will tell.

The behaviour is also undefined if`va_arg` is called when there were no further arguments.

The *type* argument must be a type name which can be converted into a pointer to such an object simply by appending a `*` to it (this is so the macro can work). Simple types such as `char` are fine (because `char *` is a pointer to a character) but array of char won't work (`char []` does not turn into

‘pointer to array of char’ by appending a `*`). Fortunately, arrays can easily be processed by remembering that an array name used as an actual argument to a function call is converted into a pointer. The correct *type* for an argument of type ‘array of char’ would be `char *`.

When all the arguments have been processed, the `va_end` function should be called. This will prevent the `va_list` supplied from being used any further. If `va_end` is not used, the behaviour is undefined.

The entire argument list can be re-traversed by calling `va_start` again, after calling `va_end`.
The `va_end`function is declared as

```
#include <stdarg.h>
void va_end(va list ap);
```

The following example shows the use of `va_start`, `va_arg`, and `va_end` to implement a function that returns the biggest of its integer arguments.

```
#include <stdlib.h>
#include <stdarg.h>
#include <stdio.h>

int maxof(int, ...) ;
void f(void);

main(){
        f();
        exit(EXIT_SUCCESS);
}

int maxof(int n_args, ...){
        register int i;
        int max, a;
        va_list ap;

        va_start(ap, n_args);
        max = va_arg(ap, int);
        for(i = 2; i <= n_args; i++) {
                if((a = va_arg(ap, int)) > max)
                        max = a;
        }

        va_end(ap);
        return max;
}

void f(void) {
        int i = 5;
        int j[256];
        j[42] = 24;
        printf("%d\n",maxof(3, i, j[42], 0));
}
```

Example 9.6