

El código fuente [descárgalo aquí](#)

Hace poco estuve buscando cierta información sobre C y recurrí a toda la bibliografía que he acumulado a lo largo de los años. Mientras buscaba me sorprendió ver que sólo dos de mis fuentes hablaban de cómo hacer funciones con una lista de argumentos de longitud variable. Una de ellas lo mencionaba, sin explicar nada, y me remitía a otra bibliografía. Menos mal que pude refugiarme en el único libro imprescindible para un programador de C: "El lenguaje de programación C"

de

Brian W. Kernigham y Dennis M. Ritchie

.

Las funciones cuya lista de argumentos es variable pueden llegar a ser útiles en nuestros desarrollos, así que intentaremos explicar en qué consiste esta técnica de la que, aun sin conocerla, os aprovecháis, pues es la base de la familia de funciones printf y scanf.

Podemos tener este número variable de argumentos gracias a la potencia y flexibilidad de C -donde podemos trabajar a nivel de direcciones de memoria-. Veremos [en qué se basa](#) y cómo nos ayuda una

[librería de C a crear este tipo de funciones](#)

. También programaremos

[un ejemplo](#)

para verlo más claro.

Fundamentos, demos gracias a la pila

Para entender cómo es posible crear este tipo de funciones es interesante comprender, aunque sólo sea de manera superficial, cómo funciona un compilador. En especial el mecanismo de llamadas a funciones.

Cuando el compilador traduce el código fuente a código máquina utiliza una pila para realizar ciertas tareas. La pila es, básicamente, una zona de almacenamiento temporal en memoria a la cual se hace referencia con un puntero. A través de este puntero se puede conocer sus contenidos. Cuando se llama a una función, se crea una nueva zona en ella un frame- donde se almacena cierta información, como por ejemplo, la dirección de retorno, las variables automáticas de la función, etc. A nosotros, lo que nos interesa es que en esa zona, también se copian los argumentos de la función.

En C los argumentos se pasan por valor, es decir, se hacen copias de los mismos, y por eso podemos usarlos como variables dentro del cuerpo de la función, sin miedo a que varíen fuera, y debemos trabajar con punteros cuando queremos el comportamiento contrario, es decir el paso de argumentos por referencia.

Funciones en C con lista de argumentos variable

Escrito por adrianvaca

Domingo, 20 de Marzo de 2011 12:53 -

Veamos un caso sencillo. Tenemos la siguiente función y llamada:

```
codeDivStart()void funcion(int a, int b);  
funcion(10, 20);
```



En el frame de la pila tendremos los valores de dichas variables. Por tanto, si la dirección de memoria del primer parámetro es X, el contenido [X] es 10 en adelante, usaré la convención de los corchetes para indicar el contenido de una posición de memoria-. Además, el segundo parámetro estará en XXXX+sizeof(int), por tanto [X+sizeof(int)] es 20.

Parece claro que las variables están contiguas en memoria. Si tenemos ahora una función y la llamada:

```
codeDivStart()void funcion(char *s , ... );  
funcion("Estos son dos números: %d y %f", 10, 0.5);
```



Entonces, como vimos antes, sabemos que [X], por ser un puntero contiene la dirección de memoria donde se almacena la letra 'E'. Pero además, sabemos que de existir otro parámetro este estaría en la posición X+sizeof(char *), y que el siguiente estaría en X+sizeof(char *)+sizeof(tipo) -el tipo es desconocido a priori-. Es aritmética de punteros pura y dura. De este modo, es sencillo obtener todos los argumentos que se pasen a la función, sin importar cuantos sean ni de que tipo.

En realidad, esto no tiene porque suceder de este modo tan ideal, dependerá de la arquitectura, el operativo, el compilador, etc. Sin embargo, este modelo de posiciones contiguas para los argumentos es la base del sistema que usa C, y ayudará al lector a entender el mecanismo que se nos proporciona.

#include <stdarg.h>, o cómo hacer este tipo de funciones

En el fichero de cabecera estándar stdarg.h se definen una serie de macros y un tipo de dato, que nos permiten gestionar las listas de argumentos variables. Su implementación concreta depende de estructuras internas del compilador, pero básicamente se basa en el modelo teórico que hemos explicado antes.

Funciones en C con lista de argumentos variable

Escrito por adrianvaca

Domingo, 20 de Marzo de 2011 12:53 -

Una función de este tipo se define de manera sencilla: como una función normal a la que, después del último argumento declarado se añaden puntos suspensivos.

```
codeDivStart()tipo funcion (tipo last, ... );
```

Una vez declarada, su implementación utilizará cuatro construcciones definidas en el fichero `stdarg.h`. A continuación veremos la estructura de cada una y cuál es su función:

1. `va_list...` es un tipo de dato que define un puntero a la lista de argumentos variable, se maneja a través de las macros siguientes.

2. `void va_start(va_list pa, last)...` macro que inicializa un puntero `va_list`, usando para ello `last` que es el nombre del último argumento antes de la lista variable, es decir, el último argumento cuyo tipo conoce la función.

3. `tipo va_arg(va_list pa, tipo)...` macro que devuelve el valor del siguiente argumento -apuntado por el puntero `pa` y de tipo `tipo`-, que puede ser de cualquier tipo válido que pueda pasarse como argumento a una función. Además, esta macro, manipula `pa` haciendo que apunte al siguiente argumento de la lista para que invocaciones sucesivas devuelvan los valores del resto de argumentos.

4. `void va_end(va_list pa)...` macro que manipula un retorno normal de la función cuya lista de argumentos variable fue inicializada por `va_start`.

Visto en pseudocódigo el proceso para manejar la lista variable es:

```
codeDivStart()tipo funcionVariable(last , ...) {  
  
    va_list pa;  
  
    tipo_X argumento_de_tipo_X;  
  
    va_start(pa,last);  
  
    while (quedanArgumentos)  
        argumento_de_tipo_X = va_arg(pa, tipo_X);
```

Funciones en C con lista de argumentos variable

Escrito por adrianvaca

Domingo, 20 de Marzo de 2011 12:53 -

```
    va_end(pa);  
}
```

El orden es innegociable. Debe definirse el puntero de tipo `va_list` y debe inicializarse con `va_start` para poder trabajar. Al terminar hay que llamar siempre a `va_end`.

Hay que prestar atención a unos detalles cuando usemos estas construcciones:

1. La lista variable siempre va al final de los argumentos.
2. Tiene que haber al menos un argumento definido `-last-`, para poder tener una referencia con la que acceder al resto de argumentos.
3. Dado que la dirección de `last` es utilizada por `va_start` no debe ser ni una variable register, ni una función, ni un array. La razón queda como ejercicio para el lector ¡vaya!, siempre quise escribir eso-.
4. Si no hay próximo argumento, o si tipo no es compatible con el tipo del próximo argumento, se producirán errores impredecibles al llamar a `va_arg`.
5. Ojo con los tipos. Tened en cuenta que por defecto C hace promoción de los parámetros pasados, así, si pasáis un `float`, este será promovido a `double` y la sentencia `va_arg(pa, float)` será incorrecta. Lo mismo con `shorts`, etc. Normalmente el compilador nos avisará de esto.
6. Las macros nos ayudan a trabajar de modo portable. No importa, por ejemplo, que un `int` tenga 32 o 16 bits. Las macros y el compilador se encargan de lidiar con la diferencia. Nosotros sólo vemos una lista de variables de distintos tipos.

Controlando el número de argumentos

Controlar el número de argumentos es importante para evitar errores. Pongamos como ejemplo algo muy habitual cuando andamos un poco despistados: equivocarnos en un `printf`. Si escribimos lo siguiente...

```
codeDivStart()printf("%d %d %d", 3, 4, 5, 6);  
printf("%d %d %d", 3, 4);
```

...obtendremos la salida de abajo el último número puede variar de una máquina a otra-.

```
codeDivStart()3 4 5  
3 4 134513729
```

Pasarse en el número de argumentos no parece muy problemático, sin embargo, quedarse corto obtiene resultados extraños. El compilador está buscando un entero pero encuentra memoria sin inicializar basura- que en este caso particular equivale a un 134513729 si se interpreta como un entero.

Hay que tener mucho cuidado. En este sencillo ejemplo, tenemos un problema visual, que no parece muy grave. ¿Y si el argumento fuese un puntero? los efectos colaterales podrían ser terribles, podríamos tener un fallo de segmento o uno de esos bugs casi imposibles de depurar.

Visto lo visto, parece que `stdarg.h` nos ofrece un mecanismo un poco escaso. Sería mejor algo similar a los argumentos de línea de comandos en `main int argc y char* argv[]`-, o al menos tener una macro `va_count`, o similar, que nos dijese cuántos argumentos tenemos. Por desgracia no es así y tenemos que usar otra aproximación donde, como programadores, tenemos que hacer explícito el número de los mismos.

En general, hay dos modos de hacerlo:

1. Con un contador... uno de los argumentos fijos es un entero que indica el número de argumentos variables que debemos esperar. No permite indicar el tipo de los argumentos a menos que usemos un artificio adicional como indicarlos con constantes simbólicas-. Suele usarse con funciones que reciben argumentos de un tipo concreto.

2. Con una cadena de formateo... el estilo que usa la familia de funciones `printf` y `scanf`. La función recibe un argumento que indica de alguna manera la cantidad de argumentos variables esperados y el tipo de los mismos.

Un ejemplo sencillo

Ya va siendo hora de hacer algo más tangible y de terminar este artículo. Para ello, construiremos paso a paso un pequeño ejemplo con dos funciones y un `main` de prueba, explicando que vamos haciendo.

1. Función para sumar un número variable de dobles: usando un argumento para contar.

Esta sencilla función toma un número variable de argumentos de tipo double para sumarlos. Tiene un argumento fijo: un entero operandos que indica la longitud de la lista variable.

El bucle para recorrer dicha lista itera sobre el argumento operandos hasta que se hace cero, indicando que no quedan más. Este ejemplo es una implementación casi directa del pseudocódigo que vimos antes y no presenta mayores complicaciones.

```
codeDivStart()double multiSuma(int operandos, ...) {  
  
    double resultado = 0.0;  
    va_list pa;  
  
    va_start(pa, operandos);  
  
    while (operandos--) {  
        resultado += va_arg(pa, double);  
    }  
  
    va_end(pa);  
  
    return resultado;  
}
```

2. printf entre paréntesis: usando una cadena de formateo.

Esta función parece más compleja, pero en el fondo es similar a la anterior. Vamos a hacer una implementación reducida de un printf sólo para cadenas y enteros- que rodea entre paréntesis las variables que escribe. La hemos llamado pprintf.

Esta función usa una cadena de formateo, que en nuestro caso, tiene dos marcas, ?s para variables de tipo cadena y ?d para las de tipo entero.

El tratamiento de la lista variable de argumentos es idéntico al realizado en el ejemplo anterior, salvo que ahora si que podemos usar tipos distintos véase el switch-. En este caso, la iteración no es sobre la lista de argumentos, sino sobre la cadena de formateo hemos definido un puntero char* p para recorrerla con aritmética de punteros-. Leemos cada carácter y lo escribimos en pantalla con putchar- a menos que sea un '?' que nos indica que busquemos un argumento más cuyo tipo se define en el siguiente carácter -*++p-.

```
codeDivStart()void pprintf(char *formato, ...) {

    char *p;
    va_list pa;

    va_start(pa, formato);

    for (p = formato; *p; p++) {

        if (*p != '?') {
            putchar (*p);
            continue;
        }

        switch (*++p) {
            case 'd':
                printf("(%d)", va_arg(pa, int));
                break;
            case 's':
                printf("(%s)", va_arg(pa, char *));
                break;
            default:
                putchar(*p);
                break;
        }
    }
    va_end(pa);
}
```

3. Probando lo anterior

Ya para terminar escribimos un pequeño main para probar nuestras dos funciones:

```
codeDivStart()int main (void) {

    printf("Resultado es %g", multiSuma(3, 0.3, 0.1, 0.2));
    pprintf("Probemos a poner parentesis a una ?s y a un ?d", "cadena", 25);

    return 0;
}
```

Como era de esperar, la salida es:

Funciones en C con lista de argumentos variable

Escrito por adrianvaca

Domingo, 20 de Marzo de 2011 12:53 -

codeDivStart()Resultado es 0.6

Probemos a poner paréntesis a una (cadena) y a un (25)

El código fuente [descárgalo aquí](#)