El lenguaje C

1. Funciones Recursivas

Para algunos tipos de problemas es útil tener funciones que se llaman a si mismas. Una *función recursiva* es una función que se llama a si misma.

Primero consideraremos la recursión en forma conceptual y a continuación examinaremos varios programas que contienen funciones recursivas.

Las funciones recursivas se definen definiendo:

1. caso base: son casos simples. Si la función es llamada con el caso base la función simplemente devuelve un resultado.
2. caso recursivo: la función es llamada con un problema más complejo. Para hacer factible la recursión este último debe parecerse al problema original. El paso de recursión puede dar como resultado muchas llamadas recursivas a la función con problemas más sencillos que el original.

A fin de que la recursión en forma eventual se termine, cada vez que la función se llame a si misma debe ser sobre una versión ligeramente más sencilla que el problema original. Esta secuencia de problemas más pequeños deben de converger en el paso base.

Ejemplo 1: Factorial

El factorial de un entero no negativo, escrito $n!$ y pronunciado factorial de n , es el producto:

$$n \cdot (n-1) \cdot (n-2) \dots 1$$

con $1! = 1$ y $0! = 1$.

El factorial de un entero *numero* mayor o igual a 0 puede ser calculado en forma iterativa utilizando for como sigue:

```
factorial=1;
for (cont=numero;cont >= 1;cont - - )
    factorial *= cont;
```

Una definición recursiva de la función factorial se obtiene al observar la siguiente relación:

$$n! = n \cdot (n-1)!$$

la definición de la función recursiva correspondiente a factorial es:

```

long factorial(int numero)
{
    if (numero <= 1) return 1;
    else return (numero * factorial(numero-1));
}

```

utilizamos variables de tipo long porque la función factorial crece rápido.

Ejemplo 2: Fibonacci

La serie de Fibonacci:

0,1,1,2,3,5,8,13,21,...

empieza con 0 y 1 y tiene la propiedad que cada número subsecuente es la suma de los dos números previos.

Puede ser definida en forma recursiva por:

```

fibonacci(0)=0
fibonacci(1)=1
fibonacci(n)=fibonacci(n-1)+fibonacci(n-2)

```

el programa respectivo es:

```

long fibonacci (int n)
{
    if (n==0 || n==1) return n;
    else return fibonacci(n-1)+fibonacci(n-2);
}

```

Cada vez que se invoca a fibonacci se prueba el caso base, es decir si n es 0 o 1. Si esto es verdadero se regresa n. Si n es mayor que 1 el paso de recursión genera dos llamadas recursivas cada una de las cuales es para un problema ligeramente más sencillo que el original (valores de n menores).

Ejemplo 3

Consideremos una función que recibe un número n e imprime los números del n al 1:

```

void mi_funcion(int cont)
{
    if (cont==0) return;
    else {
        printf(" %d ",cont);
        mi_funcion(--cont);
        return;}
}

```

observar que llamamos recursivamente con -- cont. Si pusieramos en cambio cont --, la función entraria en loop ya que llamaria con cont sin restar.

Ejemplo 4: El triangulo de Pascal

Lo utilizamos para calcular las C_m^n . Este número cuenta la cantidad de formas de escoger m objetos de un total de n objetos distintos. Les llamamos *combinaciones* de m objetos en n.

El triangulo de Pascal se construye como sigue: al principio se coloca un 1 (que corresponde a C_0^0). Para cada renglón subsecuente, digamos para el renglón n, se coloca un 1 a la izquierda y un 1 a la derecha que corresponden con C_0^n y C_n^n respectivamente. Los elementos restantes se calculan sumando los dos números que tiene justo arriba a la izquierda y a la derecha, es decir $C_m^n = C_{m-1}^{n-1} + C_m^{n-1}$ para todo $0 < m < n$.

```

      1
     1 1
    1 2 1
   1 3 3 1
  1 4 6 4 1

```

El programa para calcular combinaciones es el siguiente:

```

int comb(int n,int m)
{
    if ((n==0) || (n==m)) return 1;
    else return comb(n-1,m-1) + comb(n-1,m);
}

```

Ejemplo 5: printd

Este ejemplo consiste en imprimir un número como un string de caracteres. Llamaremos recursivamente con el cociente de dividir entre 10. Por ejemplo, supongamos tenemos el número 3267. Llamaremos recursivamente con 326, con 32 y con 3. Cuando llegamos a 3 el cociente es 0 por lo cual no realizamos ninguna llamada recursiva. Luego de la llamada recursiva o si no hay llamada recursiva, imprimimos el módulo del número entre 10, por lo cual se imprimira: 3 (módulo de 3), 2 (módulo de 32), 6 (módulo de 326) y 7 (módulo de 3267).

El programa es el siguiente:

```

void printd (int n)
{
    int i;

    if (n !=0)
    {
        putchar('-');
        n= -n;
    }
    if ((i=n/10)!=0) printd(i);
}

```

```
    putchar(n % 10 + '0');  
}
```

2. Recursión en comparación con iteración

Compararemos los dos enfoques y analizaremos porqué el programador debe escoger un enfoque sobre el otro en una situación en particular.

Tanto la iteración como la recursión se basan en una estructura de control: la iteración utiliza una estructura de repetición, la recursión utiliza una estructura de selección. Tanto la iteración como la recursión implican repetición: la iteración utiliza la estructura de repetición en forma explícita, la recursión consigue la repetición mediante llamadas de función repetidas. La iteración y la recursión ambas involucran una prueba de terminación: la iteración termina cuando falla la condición de continuación del ciclo, la recursión termina cuando se reconoce un caso base.

La recursión tiene muchas negativas. La sobrecarga de llamadas de la función puede resultar costoso tanto en tiempo de procesador como en espacio de memoria. Cada llamada recursiva genera otra copia de la función, esto puede consumir gran cantidad de memoria. La iteración por lo regular ocurre dentro de una función por lo que no ocurre la sobrecarga de llamadas repetidas de función y asignación extra de memoria.