Principios de Programación

El lenguaje C

1. Introducción, Funciones

La mayor parte de los programas de computo que resuelven problemas de la vida real son mucho mayores que los programas que hemos visto.

La experiencia ha mostrado que la mejor forma de desarrollar y mantener un programa grande es construirlo a partir de piezas menores o módulos, siendo cada uno de ellos más facil de manipular que el programa original. Esta técnica se conoce como *divide y venceras* .

C fue diseñado para hacer funciones eficientes y faciles de usar. Los programas C consisten generalmente de varias funciones pequeñas en vez de pocas grandes.

1.1. Módulos de programa en C

Una función es un fragmento de código que realiza una tarea bien definida. Por ejemplo, la función printf imprime por la salida estandar los argumentos que le pasamos.

En C los módulos se llaman funciones. Por lo general en C los programas se escriben combinando nuevas funciones que el programador escribe con funciones "preempacadas"disponibles en la *biblioteca estandar de C*.

La biblioteca estandar de C contiene una amplia colección de funciones para llevar a cabo cálculos matemáticos, manipulaciones de cadenas, entrada/salida, y muchas otras operaciones útiles. Esto facilita la tarea del programador porque estas funciones proporcionan muchas de las capacidades que los programadores requieren.

Las funciones se invocan mediante una *llamada de función*. La llamada de función especifica el nombre de la misma y proporciona información (en forma de *argumentos*) que la función llamada necesita a fin de llevar acabo su tarea.

1.2. Funciones

Las funciones permiten a un programador modularizar un programa.

Todas las variables declaradas en las definiciones de función son *variables locales* (son conocidas sólo en la función en la cual están definidas).

La mayor parte de las funciones tienen una lista de parámetros. Los parámetros proporcionan la forma de comunicar información entre funciones. Los parámetros de función son también variables locales.

Existen varios intereses que dan motivo a la "funcionalización"de un programa. El enfoque de divide y venceras hace que el desarrollo del programa sea más manipulable. Otra motivación es la reutilización del software - el uso de funciones existentes como bloques constructivos para crear nuevos programas.

Cada función deberá limitarse a ejecutar una tarea sencilla y bien definida y el nombre de la función deberá expresar claramente dicha tarea.

Si no se puede elegir un nombre consiso es probable que la función esté intentando ejecutar demasiadas tareas diversas.

1.3. Definiciones de función

Cada programa que hemos presentado ha consistido de una función llamada **main** que para llevar a cabo sus tareas ha llamado funciones estandar de biblioteca. Veremos ahora como los programadores escriben sus propias funciones personalizadas.

Considere un programa que utiliza una función **cuadrado** para calcular los cuadrados de los enteros del 1 al 10.

```
/* Funcion que calcula cuadrado de un numero. */

int cuadrado(int);

main ()
{
      int x;

      for (x=1; x <= 10; x++)
          printf(" %d ", cuadrado(x));
      printf ("\n");
      system("PAUSE");
}


/* Definicion de funcion */

int cuadrado(int y)
{
    return y*y;
}
```

Se imprime:

1 4 9 16 25 36 49 64 81 100

Es conveniente dejar lineas en blanco entre definiciones de función para mejorar la legibilidad del programa.

La función cuadrado es invocada dentro de printf. Ésta recibe una copia del valor de x en el parámetro y. A continuación calcula y*y. El resultado se regresa a la función printf en main donde se llamo a cuadrado y printf despliega el resultado. Éste proceso se repite diez veces utilizando la estructura de repetición for.

La definición de cuadrado muestra que esta función espera un parámetro entero. La palabra reservada **int** que precede al nombre de la función indica que la función devuelve un resultado entero. El enunciado **return** en la función cuadrado pasa el resultado del cálculo de regreso a la función llamadora.

La linea

```
int cuadrado(int);
```

es un **prototipo de función**. El int dentro del paréntesis informa al compilador que cuadrado espera recibir del llamador un valor entero. El int a la izquierda del nombre de la función le informa al compilador que la función regresa al llamador un resultado entero. El compilador hace referencia al prototipo de la función para verificar que la llamadas contengan el tipo de regreso correcto, el número correcto de argumentos, el tipo correcto de argumentos y el orden correcto de los argumentos.

El formato de una definición de función es:

```
tipo_de_regreso nombre_de_funcion(lista de parametros)
{

        declaraciones

        enunciados
}
```

El nombre de la función es cualquier identificador válido. El tipo de regreso es el tipo del resultado que se regresa al llamador. Se coloca como tipo **void** cuando la función no regresa un valor.

La lista de parámetros consiste en una lista separada por comas que contiene las declaraciones de los parámetros recibidos por la función al ser llamada. Si una función no recibe ningun valor la lista de parámetros es void. Para cada parámetro debe especificarse un tipo.

En el caso en que la función no reciba parametros se coloca void entre parentesis o simplemente los parentesis sin parametros (esto incluye a main).

Las declaraciones junto con los enunciados dentro de las llaves forman el cuerpo de la función. Bajo ninguna circunstancia puede ser definida una función en el interior de otra función.

Existen diferentes formas de regresar el control al punto desde el cual se invocó una función. Si la función no regresa un resultado el control se devuelve cuando se llega a la llave derecha que termina la función o al ejecutar el enuciado *return*. Si la función regresa un resultado el enunciado *return expresion* devuelve el valor de expresión al llamador y termina la ejecución de la función.

Ejemplo 1

En el siguiente ejemplo se utiliza una función **máximo** definida por el programador para determinar y regresar el mayor de tres enteros. Este valor es regresado main mediante un enunciado return.

```
/* Hallo el mayor de tres enteros */

int maximo(int,int,int);

main ()
```

```
{
      int a,b,c;

      printf("Ingrese tres enteros: ");
      scanf(" %d %d %d", &a, &b, &c);
      printf("El maximo es: %d\n", maximo(a,b,c));
      system("PAUSE");
}
/* Definicion de la funcion maximo */
int maximo(int x,int y,int z)
{
   int max=x;

   if (y>max) max=y;
   if (z>max) max=z;
   return max;
}
```

Ejemplo 2

En el siguiente ejemplo se define una función que calcula el **cubo** de un entero. main lee un valor, e imprime su cubo.

```
/* Hallo el cubo de un entero */

int cubo(int);

main ()
{
      int a;

      printf("Ingrese un entero: ");
      scanf(" %d", &a);
      printf("El cubo de %d es: %d\n", a, cubo(a));
      system("PAUSE");
}
/* Definicion de la funcion cubo */
int cubo(int x)
{
   return x*x*x;
}
```

Argumentos formales y reales

Llamamos argumentos formales a los parámetros de la función y reales a los argumentos que aparecen en la llamada a la función.

1.4. Prototipos de funciones

Una de las características más importante del ANSI C es el prototipo de función. El compilador utiliza los prototipos de función para verificar las llamadas de función.

Es conveniente incluir prototipos de función para aprovechar la capacidad de C de verificación de tipo. Se debe utilizar las directrices de preprocesador de C (#include) para obtener los prototipos de funciones definidos en un archivo aparte.

No es necesario especificar nombres de parámetros en los prototipos de función, pero se pueden especificar para mejorar la comprensión del cdigo fuente. Es necesario colocar punto y coma al final de un prototipo de función.

Una llamada de función que no coincida con el prototipo de la función causará un error de sintaxis.

Otra característica importante de prototipos de función es la **coerción** de argumentos, es decir, obligar a los argumentos al tipo apropiado. Estas conversiones pueden realizarse si se siguen las reglas de promoción de C (por ejemplo tengo un parametro real y la función es llamada con un entero).

La conversión de valores a tipos inferiores por lo general resulta en un valor incorrecto.

Si el prototipo de función de una función no ha sido incluido en un programa el compilador forma su propio prototipo utilizando la primera ocurrencia de la función, ya sea la definición de función o una llamada a dicha función.

En los ejemplos anteriores los prototipos son:

```
int cuadrado(int);

int maximo(int,int,int);

int cubo(int);
```

1.5. Devolución de valores

Una función en C slo puede devolver un valor. Para devolver dicho valor, se utiliza la palabra reservada return cuya sintaxis es la siguiente:

return expresion

Donde expresión puede ser cualquier tipo de dato salvo un array o una función. Adems, el valor de la expresión debe coincidir con el tipo de dato declarado en el prototipo de la función. Por otro lado, existe la posibilidad de devolver múltiples valores mediante la utilización de estructuras.

Dentro de una función pueden existir varios return dado que el programa devolverá el control a la sentencia que ha llamado a la función en cuanto encuentre la primera sentencia return.

Si no existen return, la ejecución de la función continúa hasta la llave del final del cuerpo de la función (}). Hay que tener en cuenta que existen funciones que no devuelven ningún valor. El tipo de dato devuelto por estas funciones puede ser void, considerado como un tipo especial de dato. En estos casos, la

sentencia return se puede escribir como return sin expresión o se puede omitir directamente .

Por ejemplo:

```
void imprime_linea()
{
    printf("esta funcion solo imprime esta linea");
    return;
}
```

es equivalente a :

```
void imprime_linea()
{
    printf("esta funcion solo imprime esta linea");
}
```

1.6. Archivos de cabecera

Cada biblioteca estandar tiene un *archivo de cabecera* correspondiente que contiene los prototipos de función de todas las funciones de dicha biblioteca y las definiciones de varios tipos de datos y de constantes requeridas por dichas funciones.

El programador puede crear archivos de cabecera personalizados. Estos deben terminar en .h. Un archivo de cabecera definido por el programador puede ser incluido utilizando #include.

Cuando un programa utiliza un número elevado de funciones, se suelen separar las declaraciones de función de las definiciones de las mismas. Al igual que con las funciones de biblioteca, las declaraciones pasan a formar parte de un fichero cabecera (extensión .h), mientras que las definiciones se almacenan en un fichero con el mismo nombre que el fichero .h, pero con la extensión .c.

1.7. Acceso a una función

Para que una función realice la tarea para la cual fue creada, debemos acceder o llamar a la misma. Cuando se llama a una función dentro de una expresión, el control del programa se pasa a ésta y sólo regresa a la siguiente expresión de la que ha realizado la llamada cuando encuentra una instrucción return o, en su defecto, la llave de cierre al final de la función.

Generalmente, se suele llamar a las funciones desde la función main, lo que no implica que dentro de una función se pueda acceder a otra función.

Cuando queremos acceder a una función, debemos hacerlo mediante su nombre seguido de la lista de argumentos que utiliza dicha función encerrados entre paréntesis. En caso de que la función a la que se quiere acceder no utilice argumentos, se deben colocar los paréntesis vacos.

Cualquier expresión puede contener una llamada a una función. Esta llamada puede ser parte de una expresión simple, como una asignación, o puede ser uno de los operandos de una expresión ms compleja. Por ejemplo:

```
a=cubo(2);
calculo=b+c/cubo(3);
```

Debemos recordar que los argumentos que utilizamos en la llamada a una función se denominan argumentos reales. Estos argumentos deben coincidir en el número y tipo con los argumentos formales o parámetros de la función. No olvidemos que los argumentos formales son los que se utilizan en la definición y/o declaración de una función.

Los argumentos reales pueden ser variables, constantes o incluso expresiones más complejas. El valor de cada argumento real en la llamada a una función se transfiere a dicha función y se le asigna al argumento formal correspondiente.

Generalmente, cuando una función devuelve un valor, la llamada a la función suele estar dentro de una expresión de asignación, como operando de una expresión compleja o como argumento real de otra función.

Sin embargo, cuando la función no devuelve ningún valor, la llamada a la función suele aparecer sola. Por ejemplo:

```
imprime_linea();
```

2. Cómo llamar funciones. Llamadas por valor y por referencia.

Hasta ahora siempre hemos declarado los parámetros de nuestras funciones del mismo modo. Sin embargo, éste no es el único modo que existe para pasar parámetros.

La forma en que hemos declarado y pasado los parámetros de las funciones hasta ahora es la que normalmente se conoce como "por valor". Esto quiere decir que cuando el control pasa a la función, los valores de los parámetros en la llamada se copian a .ºbjetos"locales de la función, estos .ºbjetos"son de hecho los propios parámetros.

Lo veremos mucho mejor con un ejemplo:

```
#include <stdio.h>
#include <iostream.h>

int funcion(int n, int m);

main ()
{
      int a, b;
      a = 10;
      b = 20;

      printf("valores de a,b = %d , %d", a, b);
```

```
        printf("valor de funcion(a,b)= %d",funcion(a, b));
        printf("valores de a,b = %d , %d", a, b);
        printf("valor de funcion(10,20) = %d", funcion(10,20));
        system("PAUSE");
}

int funcion(int n, int m)
{
    n = n + 2;
    m = m - 5;
    return n+m;
}
```

Bien, qu es lo que pasa en este ejemplo

Empezamos haciendo a = 10 y b = 20, despus llamamos a la función "funcionçon las objetos a y b como parámetros. Dentro de "funcion.esos parámetros se llaman n y m, y sus valores son modificados. Sin embargo al retornar a main, a y b conservan sus valores originales. Por qu?

La respuesta es que lo que pasamos no son los objetos a y b, sino que copiamos sus valores a los objetos n y m.

Pensemos, por ejemplo, en lo que pasa cuando llamamos a la función con parámetros constantes, es lo que pasa en la segunda llamada a "funcion". Los valores de los parámetros no pueden cambiar al retornar de "funcion", ya que esos valores son constantes.

Si los parámetros por valor no funcionasen así, no sera posible llamar a una función con valores constantes o literales.

Cuando los argumentos se pasan en llamada por valor se efectúa una copia del valor del argumento con el cual se invoca y se asigna a la variable local (parámetro) correspondiente al argumento. Las modificaciones que la función realice a la variable correspondiente al parámetro no afectan a las posibles variables con las que se invocó la función.

Si queremos que los cambios realizados en los parámetros dentro de la función se conserven al retornar de la llamada, deberemos pasarlos por referencia. Esto se hace declarando los parámetros de la función como referencias a objetos. Por ejemplo:

```
#include <stdio.h>
#include <iostream.h>

int funcion(int *n, int *m);

main ()
{
        int a, b;

        a = 10; b = 20;
        printf("valores de a,b = %d , %d", a, b);
        printf("valor de funcion(a,b)= %d",funcion(&a, &b));
```

```
        printf("valores de a,b = %d , %d", a, b);
        /* printf("valor de funcion(10,20) = %d", funcion(10,20));
        es ilegal pasar constantes como parametros cuando estos
        son referencias (1) */
        system("PAUSE");
}
int funcion(int *n, int *m)
{
    n = n + 2;
    m = m - 5;
    return n+m;
}
```

En este caso, los objetos a y b tendrn valores distintos después de llamar a la función, a saber a valdrá 12 y b valdrá 15 . Cualquier cambio de valor que realicemos en los parmetros dentro de la función, se hará también en los objetos de la llamada.

Esto quiere decir que no podremos llamar a la función con parámetros constantes, como se indica en (1). Un objeto constante no puede tratarse como objeto variable.

Cuando un argumento es pasado en llamada por referencia, las modificaciones que la función realice a la variable correspondiente al parámetro se realizan en las posibles variables con las que se invocó la función.

La llamada por valor debería ser utilizada siempre que la función llamada no necesite modificar el valor de la variable original del llamador.

La llamada por referencia debe ser utilizada solo cuando se necesite modificar la variable original.

Veremos otro modo de pasar parámetros por referencia.

2.1. Más ejemplos:

Veamos un ejemplo. Escribamos una función **intercambio** que reciba dos argumentos enteros y los intercambie. Escribamos la función del siguiente modo:

```
void intercambio(int x, int y)
{
      int temp;

      temp=x;
      x=y;
      y=temp;
}
```

los argumentos están pasados por valor, entoces si llamo a la función del siguiente modo : intercambio(a,b), a y b no son modificados.

Veamos como sería la función y su invocación en el caso de llamada por referencia. La función se declara:

```
void intercambio(int *x, int *y)
```

y en la invocación de la función utilizamos &, en este ejemplo:

```
intercambio(&a,&b);
```

Hay otra forma de realizar llamada por referencia, es colocando en la declaración de la función & del siguiente modo:

```
void intercambio(int &x, int &y)
```

y en la invocación de la función no necesitamos &, en este ejemplo la invocación sería:

```
intercambio(a,b)
```

3. Generación de números aleatorios

Desarrollaremos un programa para ejecución de juegos que incluye varias funciones.

Utilizaremos la función rand() existente en la biblioteca estandar de C para generar números aleatorios. Considere el enunciado:

```
i=rand();
```

Esta función genera un entero entre 0 y RAND_MAX. Éste último valor debe ser por lo menos 32767 que es el valor máximo de un entero de dos bytes.

Si rand en verdad produce enteros aleatorios cualquier número entre 0 y RAND_MAX tiene la misma oportunidad (o probabilidad) de ser elegido cada vez que rand es llamado.

Para generar enteros aleatorios entre 0 y n (donde n es un valor menor que RAND_MAX) tomamos módulo n del entero devuelto por rand().

Comencemos desarrolando un programa para simular 20 tiradas de un dado de 6 caras, e imprimamos el valor de cada tirada. El prototipo de rand se encuentra en stdlib.h.

El programa es el siguiente:

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

main ()
{
      int i;

      for (i=1;i <= 20; i++)
      {
          printf(" %10d", 1 + (rand() % 6));
          if (i % 5 == 0) printf("\n");
      }
}
```

se imprimen 5 números por linea (para ello se utiliza el if de dentro del for).

Para ver que los números ocurren aproximadamente con la misma probabilidad simularemos 6000 tiradas de un dado. Cada entero del 1 al 6 debe ocurrir aproximadamente 1000 veces.

Definiremos un programa que tira el dado 6000 veces y cuenta la cantidad de veces que aparece cada número.

El programa es el siguiente:

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

main ()
{
      int valor, cant, frecuencia1=0; frecuencia2=0,
      frecuencia3=0, frecuencia4=0, frecuencia5=0, frecuencia6=0;

      for (cant=1;cant<=6000;cant++)
      {
          valor=1 + rand() % 6;
          switch (valor)
          {
                  case 1: ++frequencia1; break;
                  case 2: ++frequencia2; break;
                  case 3: ++frequencia3; break;
                  case 4: ++frequencia4; break;
                  case 5: ++frequencia5; break;
                  case 6: ++frequencia6; break;
          }
      }
      printf(" %s %13s\n", "Numero", "Frecuencia");
      printf("1 %13d\n", frecuencia1);
      printf("2 %13d\n", frecuencia2);
      printf("3 %13d\n", frecuencia3);
      printf("4 %13d\n", frecuencia4);
      printf("5 %13d\n", frecuencia5);
      printf("6 %13d\n", frecuencia6);
}
```

El resultado de una ejecución del programa anterior da:

Numero	Frecuencia
1	987
2	984
3	1029
4	974
5	1004
6	1022

Advertir que en el switch anterior no se necesita default.

Si ejecutamos la función que tira 20 dados por segunda vez, aparece impresa exactamente la misma salida. La función rand de hecho genera números pseudoaleatorios. Cada vez que el programa se ejecute el resultado es el mismo.

Existe otra función estandar de biblioteca **srand**. Ésta función toma un argumento entero unsigned para que en cada ejecución del programa, rand produzca una secuencia diferente de números aleatorios.

Veamos el programa anterior de nuevo:

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

main ()
{
      int i;
      unsigned x;

      printf("Ingrese numero: ");
      scanf(" %u", &x);
      srand(x);

      for (i=1;i <= 10; i++)
      {
          printf(" %10d", 1 + (rand() % 6));
          if (i % 5 == 0) printf("\n");
      }
}
```

en el programa, srand toma un valor x como argumento. El prototipo de srand se encuentra en <stdlib.h>. A distintos valores de x corresponden distintos números en rand().

3.1. Ejemplo: un juego de azar

Las reglas del juego son las siguientes:

Un jugador tira dos dados. Cada dado tiene 6 caras. Las caras contienen 1,2,3,4,5 y 6 puntos. Una vez que los dados se hayan detenido se calcula la suma de los puntos en las dos caras superiores. Si a la primera tirada la suma es 7 o bien 11 el jugador gana. Si en la primera tirada la suma es 2, 3 o 12 el jugador pierde (gana la casa). Si en la primera tirada la suma es 4,5,6,8,9 o 10 el jugador debe seguir jugando hasta que o bien se repite dicho número o hasta que salga 7. En el primer caso el jugador gana en el segundo pierde.

El siguiente programa simula este juego. Note que el jugador debe tirar dos dados en cada tirada. Definimos una funcion **tirada** para tirar los dados, calcular e imprimir su suma. Esta función no tiene argumentos pero regresa un entero que es la suma de los dos dados.

El jugador puede ganar o perder en cualquier tirada. Se define una variable **estado** que lleva registro de esto.

Cuando se gana el juego estado se setea en 1. Cuando se pierde el juego, estado se setea en 2. De lo contrario estado se setea a 0 y el juego debe continuar.

El programa es el siguiente:

```
int tirada(void);

main ()
{
      int estado, suma, resultado;
      srand(time(NULL));
      suma = tirada();
      switch(suma)
      {
       case 7 : case 11 : estado=1; break;
       case 2 : case 3 : case 12 : estado=2; break;
       default: estado=0; resultado=suma;
                printf("Tirada dio %d\n",resultado);
                break;
      }
      while (estado==0)
      {
       suma=tirada();
       if (suma == resultado) estado=1;
       else if (suma==7) estado=2;
      }
      if (estado==1) printf("Jugador gana\");
      else printf("Jugador pierde\n");
}

int tirada(void)
{
    int dado1,dado2,suma;

    dado1 = 1 + (rand() % 6);
    dado2 = 1 + (rand() % 6);
    suma = dado1 + dado2;
    printf("Jugador sumo %d + %d = %d\n", dado1, dado2, suma);
    return suma;
}
```

La instrucción srand(time(NULL)) hace que la computadora lea su reloj para obtener automaticamente un valor distinto en el mismo dia. La función time devuelve la hora actual del dia en segundos. Este valor es convertido a un entero unsigned y utilizado en la generación de números aleatorios. La función time toma NULL como argumento (time puede proporcionar al programador

una cadena representando la hora del dia, NULL deshabilita esta capacidad). El prototipo correspondiente a time se encuentra en <time.h>.

Veamos el juego. Despues de la primera tirada, si el jugador gano o perdio (el juego se termino) la estructura while es salteada. El programa continúa con la estructura if/else que imprime si el jugador gano o perdio.

Después de la primera tirada, si el juego no ha terminado, guardamos la suma en resultado. La ejecución continua con la estructura while pues el estado es 0. Dentro del while se tiran los dados, si la suma de estos coincide con resultado el estado se pone en 1, si coincide con 7 el estado se pone en 2, sino continuará la ejecución del while hasta que el jugador gane o pierda. Finalmente se imprimira si el jugador ganó o perdió.