EL LENGUAJE DE PROGRAMACIÓN C

Estructura básica de un programa

- Un programa en C tiene la siguiente estructura mínima:

```
main () {
 // código

 }
```

- Donde:
  - `main` es la función que se ejecutará al iniciar el programa. Siempre debe haber un `main`.
  - El programa deberá guardarse en un archivo que se llame `nombre.c`.
  - El código del programa deberá ir entre las llaves de la función `main`.
- El sitio https://www.mycompiler.io/es provee un ambiente para probar estos programas en línea.

Estructura básica de un programa

- Ejemplo

```
#include <stdio.h>
int main() {
  double distancia, velocidad, tiempo;
  velocidad = 120; // km/hr
  tiempo = 2.5; // en horas
  distancia = velocidad * tiempo; // 300 km
  printf("%f", distancia); escribe 300
}
```

- Las variables se declaran al inicio del código.
  - es buena práctica hacerlo pero no siempre es necesario.
- El texto precedido por doble barra (//) son comentarios.
  - Los comentarios no se ejecutan, también se puede usar `/* … */`
- `printf` despliega el resultado en la pantalla.

Estructuras de control de flujo

- Programa que calcula la suma de los números pares entre 0 y 10.

```
int main() {
 int suma;
 suma = 2 + 4 + 6 + 8;
 printf("%d", suma); // escribe 20
}
```

- Pero ¿para sumar los números pares entre 0 y 10.000?,

```
númeroPar = 2;
númeroPar = númeroPar + 2; // línea 2
númeroPar = númeroPar + 2; // línea 3

hasta llegar al último par:

númeroPar = númeroPar + 2; // línea 4999
```

Repetición (while)

```
#include <stdio.h>
int main() {
    int suma, númeroPar;
    suma = 0;
    númeroPar = 2;                  // el primer número par
    while (númeroPar < 10000) {
      suma += númeroPar;
      númeroPar += 2;               // siguiente número par
    }
  printf("%d", suma);
}
```

- La estructura **`while`** repite una serie de instrucciones mientras se cumpla una condición dada. Dicha condición se cierra entre paréntesis a la derecha del **`while`**, mientras que las instrucciones que se han de repetir se cierran entre llaves ({}). En el ejemplo, se repetirá la suma mientras “númeroPar” sea más pequeño que 10.000.

Selección (if)

```
#include <stdio.h>
int main() {
    int número, suma;
    número = 1;                 // primer número del intervalo
    suma = 0;
    while (número <= 9999) { // último número del intervalo
      if (número % 2 == 0) { // ¿es par?
        suma += número;       // sí, entonces lo suma
       }
      número += 1;              // siguiente número del intervalo
    }
    printf("%d", suma);
}
```

- La estructura **`if`** ejecuta una serie de instrucciones si se cumple una determinada condición. Dicha condición se cierra entre paréntesis a la derecha del **`if`**, mientras que las instrucciones que a ejecutar se cierran entre llaves ({}). Opcionalmente, la estructura if acepta una extensión **`else`**, que especifica una serie de instrucciones que se han de ejecutar en caso de que la condición no se cumpla.

NOTA

- El cálculo de la suma de números pares es un buen pretexto para ilustrar el uso de estructuras de control.
  - Tiene la característica de consumir tiempo de máquina (control de **`while`**, consulta en **`if`**).

- Solución alternativa: si *N* es el último número a sumar, entonces sea *n* = *N / 2* y la suma se calcula con

$$S_{pares} = n * (n + 1)$$

  - El cálculo de la suma de pares *en un intervalo* se obtiene por diferencia.

Esquemas de recorrido y búsqueda

- Generalmente, se recurre a las repeticiones para:
  - Buscar un elemento de un conjunto.
  - Recorrer todos los elementos de un conjunto.
- Los esquemas de recorrido tienen siempre la misma estructura:

```
while (haya_elementos) {
  hacer_acción(elemento_actual);
  siguiente_elemento();
}
```

Esquemas de recorrido y búsqueda

- Ejemplo: encontrar el primer múltiplo de 13 entre 760 y 800:

```
int número;
número = 760;
while (número % 13 != 0)
   número++;
```

- Empezando por 760, se va incrementando la variable mientras `número` no sea múltiplo de 13.
- Pero si no hubiere un múltiplo de 13 en el intervalo ¡tendríamos una respuesta errónea!

Desarrollo de programas

- Se debe comprobar que el bucle termina alguna vez y la razón de ello.

```
#include <stdio.h>
int main() {
    int número = 760;
    while (número <= 800 && número % 13 != 0)
        número++;
    if ((número - 1) % 13 != 0)  // al menos un múltiplo
        printf("%d", número);
    else                         // no hay múltiplo
        printf("No se ha encontrado");
    }
```

- Es importante observar que la condición “elemento_encontrado” siempre irá negada. Esto siempre será así porque las instrucciones del bucle deben repetirse *mientras no se haya encontrado* el elemento que se busca.

Definición y uso de funciones

- Para encontrar el primer número primo de un intervalo dado podría usarse el esquema de búsqueda:

```
encontrado = es_primo el primer_número_del_intervalo;
while (!encontrado && hay_mas_numeros_en_el_intervalo) {
  encontrado = es_primo el siguiente_número;
}
if (encontrado) {
// número encontrado
}
else {
// número no encontrado
}
```

- De hecho, dado un número *N*, la única forma de comprobar que es primo es dividiéndolo entre todos los números primos entre 2 y N – 1, ambos incluidos (discutible).

Definición y uso de funciones

- Solución simple para determinar si un número es primo o no.

```
int número, divisor, esPrimo;
número = 37; // número a comprobar
divisor = 2;
while (número % divisor != 0 && divisor < número)
  divisor ++;
esPrimo = divisor == número; //true o false
```

- Hay dos condiciones que gobiernan el **`while`**: la que comprueba que “divisor” no sea divisor entero de “número” (`número % divisor != 0`) y la que comprueba que no se haya salido de rango (`divisor < número`).

Buscar primos en un intervalo

```
#include <stdio.h>
int main() {
   int inicio, fin;
   int número, divisor;

   inicio = 10; fin = 20;
   printf("Buscando primos entre %d y %d\n", inicio, fin);

   for (número = inicio; número <= fin; número++) {
      divisor = 2;
      while (número % divisor != 0 && divisor < número)
         divisor ++;

      if (divisor == número)  //  encuentra un primo
         printf(" *** %d\n", número);
   }
}
```

Funciones

- Repetir trozos de código dentro de un programa tiene serios inconvenientes:
  - Reduce la legibilidad. Como se puede apreciar en el programa, cuesta discernir entre el código que pertenece a la búsqueda del número primo y el que comprueba si un número lo es. Las variables se mezclan y las instrucciones también.
  - Dificulta la mantenibilidad. Si se detecta un error en la función que comprueba si un número es primo o se desea mejorarlo, las modificaciones deben propagarse por todos y cada uno de los bloques repetidos. Cuantas más repeticiones, más difícil será introducir cambios.
  - Aumenta la probabilidad de errores. Ello es consecuencia de la baja legibilidad y mantenibilidad. Una corrección o mejora que no se ha propagado correctamente por todos los fragmentos repetidos puede generar nuevos errores. La corrección de éstos, a su vez, puede generar otros nuevos, y así indefinidamente.
- Al final, la repetición de bloques de código conduce a códigos inestables con un costo de mantenimiento más elevado en cada revisión. Afortunadamente, todos estos inconvenientes se pueden salvar mediante el uso de funciones.

Funciones

- Función que determina si un número es primo o no

```
int esPrimo(int número) {
  int divisor;
  divisor = 2;
  while (número % divisor != 0 && divisor < número)
    divisor ++;
  return (divisor == número); // true o false
}
```

Uso de la función

- `primo = esPrimo(19);`

determina si el número 19 es primo:

- asigna a la variable `primo` (de tipo **`int`**) el resultado de la función `esPrimo` para el valor 19.
- La llamada a una función también termina en punto y coma.
- Al ejecutarse la función, el parámetro `número` valdrá 19. De la misma forma, al terminarla, se asignará a la variable primo el valor de la expresión que hay a la derecha del **`return`**

Buscar primos en un intervalo (versión 2)

```
#include <stdio.h>

int esPrimo(int número) {
  int divisor;
  divisor = 2;
  while (número % divisor != 0 && divisor < número)
    divisor ++;
  return (divisor == número); // true o false
}

int main() {
  int inicio, fin;
  int número, divisor, esPrimo(int);

  inicio = 10; fin = 20;
  printf("Buscando primos entre %d y %d\n", inicio, fin);

  for (número = inicio; número <= fin; número++) {

     if (esPrimo(número))  //  si encuentra un número primo lo muestra por pantalla
        printf(" *** %d\n", número);
    }
}
```

Funciones

- Dos consecuencias importantes:

  - La lógica del subprograma es completamente independiente de aquella del programa principal. Es decir, al programa principal poco le importa cómo se comprueba si un número es primo, ni que variables se utilizan para hacer dicha comprobación.

  - Programa y subprograma se comunican mediante una interfaz basada en variables bien definida.

Observaciones

- Es notable que, a veces, obtener todas las instancias de solución de un problema puede ser más simple que obtener sólo una.
- Como ejemplo de lo anterior, diseñe, construya y pruebe un programa que obtenga sólo el primer número primo de un intervalo (use secuencias, if y while).

Aspectos importantes

- Los parámetros de una función actúan como variables. Se puede operar con ellos como si fueran variables.
- Los parámetros de una función son una copia de los parámetros de entrada. Si pasamos una variable como parámetro a una función y, dentro del código de la función se modifica el parámetro, el valor de la variable no se verá alterado.
- Al espacio entre llaves que encierra el código de la función se le llama “ámbito de la función”. Las variables declaradas dentro de la función sólo pueden “verse” dentro del ámbito de la función. Es decir, no se puede acceder a ellas desde fuera de la función. Lo mismo es aplicable a los parámetros.

Aspectos importantes

- Una función se caracteriza a partir de su nombre y el tipo de los parámetros de entrada.
  - No está permitido en C definir funciones con igual nombre y direrentes parámetros (como en C++ o Java)
- Una función puede que no devuelva ningún resultado o bien, que su resultado se descartará. En este caso, el tipo de la función será **`void`** (vacío) y estaremos ante lo que se llama un procedimiento.