Receta y Testing en C

Cátedra Programación II Octubre 2023

1. Construcción de Programas

Como vimos anteriormente en **Python**, la receta que aplicamos consta de los siguientes pasos:

1. diseño de datos;
2. signatura y declaración de proposito.
3. ejemplos;
4. definición de la función;
5. evaluar el código de los ejemplos;
6. realizar modificaciones en caso que el paso anterior genere errores.

Vamos a ver esto en **C**, aplicado al mismo ejemplo que vimos en *Racket* y *Python*:

Escribir un programa que convierta una temperatura medida en un termómetro Fahrenheit a una temperatura en Celsius.

1.1. Item 1: Diseño de datos

¿Cómo representamos la información?

```
1 // Representamos temperaturas mediante números
```

1.2. Item 2: Signatura y declaración de propósito

La signatura de una función indica qué parámetros recibe (cuántos y de qué tipo) y qué datos retorna. La declaración de propósito es una breve descripción sobre qué hace la función.

En el caso de los problemas sencillos que abordaremos, deberemos decidir:

a) cuáles son los datos de entrada que se nos proveen,
b) cuáles son las salidas que debemos producir, y
c) cuál es la relación entre todos ellos.

```
1 // Representamos temperaturas mediante números enteros
2 // farCel: Int -> Int
3 // El parámetro representa una temperatura en Fahrenheit y,
4 // se retorna su equivalente en Celsius.
```

1.3. Item 3: Ejemplos

Luego de los pasos anteriores, podemos saber el resultado de la función para algunos valores de entrada

```
// Representamos temperaturas mediante números enteros
// farCel: Int -> Int
// El parámetro representa una temperatura en Fahrenheit y,
// se retorna su equivalente en Celsius.
//
// entrada: 32, salida: 0
// entrada: 212, salida: 100
// entrada: -40, salida: -40
```

1.4. Item 4: Definición del programa

¡Escribimos código!

Traducir a un lenguaje de programación el diseño que elegimos en el punto anterior.

```
// Representamos temperaturas mediante números enteros
// farCel: int -> int
// El parámetro representa una temperatura en Fahrenheit y,
// se retorna su equivalente en Celsius.
//
// entrada: 32, salida: 0
// entrada: 212, salida: 100
// entrada: -40, salida: -40

int farCel(int f) {
   return (f-32)*5/9;
}
```

En el caso del lenguaje **C**, como se puede observar, la signatura queda explícita en la definición de la función ya que el lenguaje nos obliga a poner el tipo de cada uno de los argumentos, así como también el tipo de retorno. Es por esto que, en general, en la práctica no es necesario poner la signatura como comentario, sino que basta con que se explicite en la definición de la función. Sin embargo, es importante que, antes de comenzar a definir la función, tengamos claro qué argumentos va a tomar y qué va a retornar.

1.5. Item 5: Evaluar el código en los ejemplos

¿Qué significa?

Aplicar el código generado en los ejemplos que se diseñaron y verificar que los resultados obtenidos coincidan con lo esperado. Para esto, armamos un `main` que invoque los casos de ejemplos y mostramos la salida obtenida:

```
int main() {
   printf("farCel(32) = %d\n", farCel(32));
   printf("farCel(212) = %d\n", farCel(212));
   printf("farCel(-40) = %d\n",  farCel(-40));
   return 0;
}
```

Cuya salida al ejecutarlo va a ser:

```
farCel(32) = 0
farCel(212) = 100
farCel(-40) = -40
```

1.6. Item 6: Realizar modificaciones en caso de error

¿Qué significa?

Encontrar los problemas que se detectaron en la ejecución de los ejemplos y solucionarlos.

2. Un Ejemplo: Conversión de Unidades Métricas

Problema 1. Al leer un artículo en una revista que contiene información de longitudes expresadas en millas, pies y pulgadas, correspondientes al sistema anglosajón de unidades no métricas, necesitamos convertir esas distancias a metros. Para ello decidimos escribir un programa que convierta las longitudes del **sistema anglosajón** al **sistema métrico decimal**.

Diseño de datos

En este caso el problema es sencillo:

```
// Representamos millas, pies, pulgadas y metros
// como números flotantes (con coma).
```

Declaración de propósito

```
// Representamos millas, pies, pulgadas y metros
// como números flotantes (con coma).
// convierteAMetros toma tres argumentos. El primero
```

```
// representa una longitud en millas, el segundo una
// longitud en pies y el tercero en pulgadas.
// El valor de retorno representa su equivalente en metros.
```

Ejemplos

Para construir los ejemplos, tenemos que encontrar la relación existente entre las unidades que tomamos de entrada y la salida que vamos a generar. Para esto, buscando en Internet, encontramos la siguiente tabla:

- 1 milla = 1609.344 m
- 1 pie = 0.3048 m
- 1 pulgada=0.0254 m

Por lo que podemos decir que

```
// Representamos millas, pies, pulgadas y metros
// como números flotantes (con coma).
// convierteAMetros toma tres argumentos. El primero
// representa una longitud en millas, el segundo una
// longitud en pies y el tercero en pulgadas.
// El valor de retorno representa su equivalente en metros.
//
// entrada: 1, 0, 0; salida: 1609.344
// entrada: 0, 1, 0; salida: 0.3048
// entrada: 0, 0, 1; salida: 0.0254
// entrada: 1, 1, 1; salida: 1609.6742
```

Definición del programa

Ahora, vamos a escribir el código de la función en **C**, entendiendo que si una longitud se expresa como L millas, F pies y P pulgadas, su conversión a metros se calculará como:

$$M = 1609.344 \times L + 0.3048 \times F + 0.0254 \times P$$

```
// Representamos millas, pies, pulgadas y metros
// como números flotantes (con coma).
// convierteAMetros toma tres argumentos. El primero
// representa una longitud en millas, el segundo una
// longitud en pies y el tercero en pulgadas.
// El valor de retorno representa su equivalente en metros.
//
// entrada: 1, 0, 0; salida: 1609.344
// entrada: 0, 1, 0; salida: 0.3048
// entrada: 0, 0, 1; salida: 0.0254
```

```
// entrada: 1, 1, 1; salida: 1609.6742

double convierteAMetros(double numeroMillas, double numeroPie, double ↩
    numeroPulgadas) {
     double metros;
     metros = 1609.344 * numeroMillas + 0.3048 * numeroPie + 0.0254 * ↩
    numeroPulgadas;
     return metros;
}
```

Evaluar el código en los ejemplos

Aquí tendríamos que verificar los ejemplos mencionados, por lo que el `main` quedaría asi:

```
int main() {
   printf("convierteAMetros(1,0,0) = %lf\n", convierteAMetros(1,0,0));
   printf("convierteAMetros(0,1,0) = %lf\n", convierteAMetros(0,1,0));
   printf("convierteAMetros(0,0,1) = %lf\n", convierteAMetros(0,0,1));
   printf("convierteAMetros(1,1,1) = %lf\n", convierteAMetros(1,1,1));
   return 0;
}
```

Cuya salida al ejecutarlo va a ser:

```
convierteAMetros(0,1,0) = 1609.344000
convierteAMetros(0,1,0) = 0.304800
convierteAMetros(0,0,1) = 0.025400
convierteAMetros(1,1,1) = 1609.674200
```

Con este ejemplo podemos ver cómo aplicaríamos la receta a cada función que forme parte del programa.

3. ¿Cómo testear los ejemplos?

Para automatizar los casos de prueba en **C** vamos a utilizar la librería **assert.h** la cual vamos a tener que incluir como cabecera en el programa:

```
#include <assert.h>
```

La librería *assert* define una macro `assert` que toma como parámetro una condición `test`:

```
#define assert ( test )
```

Esta macro se expande como un bloque `if` que comprueba la condición `test` y, dependiendo de si es verdadera o no, puede abortar el programa. Si `test` se evalúa como **0** (es decir,

si es falsa), entonces se aborta el programa y se imprime un mensaje de salida en el que se incluyen la condición `test`, el nombre del archivo fuente y el número de línea en la que se llamó a `assert()`. Si la condición resulta verdadera, la ejecución del programa continúa de manera normal.

Vamos a ver esto en detalle tomando como referencia el primer ejemplo que vimos modificando la función `main` para que llame a una función de testeo que utilice la macro `assert`:

```
1  #include <stdio.h>
2  #include <assert.h>
3
4  // Representamos temperaturas mediante números enteros
5  // farCel toma un parámetro que representa una temperatura
6  // en Fahrenheit y retorna su equivalente en Celsius.
7  //
8  // entrada: 32, salida: 0
9  // entrada: 212, salida: 100
10 // entrada: -40, salida: -40
11
12 int farCel(int f) {
13    return (f-32)*5/9;
14 }
15
16 void test_farCel() {
17    assert(farCel(32) == 0);
18    assert(farCel(212) == 100);
19    assert(farCel(-40) == -40);
20 }
21
22 int main() {
23    test_farCel();
24    return 0;
25 }
```

Como se puede ver, estamos verificando que el valor obtenido en la llamada a la función `farCel` coincida con lo esperado. Al cumplirse todos los casos, los tests se pasan sin problemas (y sin aviso también!). Si, por ejemplo, hubieramos puesto en la línea 17:

```
1     assert(farCel(212) == 120);
```

la salida que obtendríamos seria:

```
1 main: receta.c:17: test_farCel: Assertion ‘farCel(212) == 120’ failed.
2 Abortado (‘core’ generado)
```