Funciones

Esp. Ing. José María Sola, profesor.

Revisión 1.0.0
2018-05-14

Tabla de contenidos

1

Introducción a Funciones

1.1. Ejemplo Introductorio

1.1.1. En Notación Matemática

$$f: \mathbb{R} \rightarrow \mathbb{R} / f(x) = 2x + 1$$

1.1.2. En C++

```
double f(double); //<2>
double f(double x){ return 2*x + 1; } //<5>
```

1.1.3. Analogía: Relación entre Conjuntos

$$f: \mathbb{R} \rightarrow \mathbb{R}$$

```
double f(double); //<2>
```

1.1.4. Analogía: Definición de la Relación

$$f(x) = 2x + 1$$

```
double f(double x){ return 2*x + 1; } //<5>
```

1.1.5. Aplicación e Invocación de Prueba

$$f(3) = 2 \cdot 3 + 1 = 7$$

```
assert( 2*3+1 == 7 );
assert( 7 == f(3) ); //<4>
```

1.1.6. Programa Completo C++

```
/* f
[stem]
++++
f:RR->RR
//
f(x)=2x+1
++++
f(3)=2*3+1=7
JMS
2018 */

#include <cassert> ❶
#include <iostream>

double f(double); ❷

int main(){
 std::cout << f(3);  ❸
 assert( 2*3+1 == 7 );
 assert( 7 == f(3) ); ❹
}

double f(double x){ return 2*x + 1; } ❺
```

❶ Incluye las declaraciones para usar `assert`. En C++ es erróneo incliur `assert.h`.

❷ *Declaración* ó *prototipo* de la función, indica tipo de la función, nombre y tipo de sus parámetros, los nombres de los parámetros son opcionales.

❸ *Invocación* de la función, con un *argumento*. En este caso, el valor de la invocación se envía a `cout`.

❹ *Invocación* de la función, con un *argumento*. En este caso, el objetivo es probar la correcta *implementación* de la función. El valor de la invocación se compara con el valor esperado, si no son iguales, finaliza la ejecución del programa con un error descriptivo.

5. *Definición* o *implementación* de la función: Debe incluir los nombres de los *parámetros*, el *cuerpo* de la función debe estar entre llaves e incluir el *retorno* de un valor.

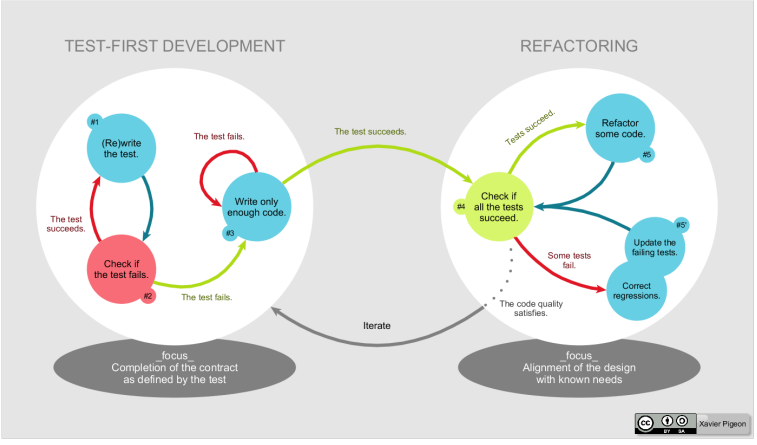
1.1.7. Orden de Escritura versus Orden de Desarrollo

En relación a la funciones, el compilador espera encontrar primero la declaración, luego la invocación de prueba, y por último la definición. Pero el orden de desarrollo recomendado es:

1. Primero escribir las pruebas.
2. Luego escribir el prototipo que coincida con las pruebas.
3. Por último, implementar la función de manera tal que *pase* las pruebas escritas

Una vez que el prototipo está estable, se sugiere seguir los pasos de *Test Driven Development*[TDD]:

1. Agregar un test, ejecutarlo y verlo fallar (*fail*).
2. Escribir el código para que pase el test, ejecutarlo y verlo pasar (*pass*).
3. Refactorizar el código para mejorar su calidad, sin agregar funcionalidad (*refactor*).
4. Relizar el *commit*.
5. Repetir ciclo.



1.2. Funciones de Escalares (1→1)

Estas son funciones que dado un valor generan otro, son funciones de una sola variable.

1.2.1. Constante

$$y: \mathbb{Z} \rightarrow \mathbb{Z} / y(x) = 42$$

```
/* y
[stem]
++++
y:ZZ->ZZ // y(x)=42
++++
JMS
2018
*/

#include <cassert>
#include <iostream>

int y(int); ❶

int main(){
 std::cout << y(-9); ❷
 assert( 42 == y(1) ); ❸
}

int y(int x){return 42;} ❹❺
```

❶ Declaración ó prototipo de la función.

❷ Invocación de prueba de la función que envía resultado a `cout`.

❸ Invocación de prueba de la función mediante `assert`.

❹ Definición o implementación de la función.

❺ La función constante es un caso especial, el parámetro no se usa, así que puede recibir cualquier valor. Los compiladores emiten un *warning* (advertencia) sobre esta situación; aunque compile y ejcute correctamente, puede haber un error *pragmático*.
Es por eso que debemos optar entre a) atender el *warning*, ya sea mediante el uso efectivo o eliminación del parámetro, o b) ignorarlo.

Al ser un ejemplo, decidimos ignorar el warning. Más adelante vamos a ver otra forma de definir una función constante.

1.2.2. Identidad

$$id : \mathbb{Z} \to \mathbb{Z} / id(x) = x$$

```
/* Id
[stem]
++++
id:ZZ->ZZ // id(x)=x
++++
JMS
2018 */

#include <cassert>
#include <iostream>

int Id(int);

int main(){
 std::cout << Id(42); ❶
 assert( 7 == Id(7) );
}

int Id(int x){return x;}
```

❶ Este es el último ejemplo que probamos mediante envíos a `cout`, en próximos ejemplos vamos a usar simplemente `assert`.

1.2.3. Sucesor

A partir de este ejemplo, vamos a incluir solo los puntos más importantes del desarrollo de cada función:

1. Definición matemática.
2. Prototipo.
3. Pruebas.
4. Implementación.

$$\text{suc}:\mathbb{Z}\rightarrow\mathbb{Z}/\text{suc}(x)=x+1$$

```
int Suc(int);
```

```
assert( -41 == Suc(-42) );
assert(   0 == Suc( -1) );
assert(   1 == Suc(  0) );
assert(  42 == Suc( 41) );
```

```
int Suc(int x){return x+1;}
```

1.2.4. Negación

$$\text{neg}:\mathbb{Z}\rightarrow\mathbb{Z}/\text{neg}(x)=\ -x=(-1)\cdot x$$

```
int Neg(int);
```

```
assert( -7 == Neg(  7) );
assert(  0 == Neg(  0) );
assert( 42 == Neg(-42) );
```

```
int Neg(int x){return -x;} // return -1*x;
```

1.2.5. Doble

$$\text{double}:\mathbb{Z}\rightarrow\mathbb{Z}/\text{double}(n)=2n$$

```
int Double(int);
```

```
assert( -14 == Double(-7) );
assert(   0 == Double(0) );
assert(  42 == Double(21) );
```

```
int Double(int n){return 2*n;}
```

1.3. Funciones Partidas (*Piecewise*)

Estas funciones se pueden implementar en C++ con el *operador condicional*, también conocido como ***el** operador ternario*, ya que el resto de los operadores son unarios o binarios.

Selección

Otra alternativa para implementar las funciones partidas es mediante *control de flujo de ejecución estructurado*, en particular con la estructura *selección*, que C++ implementa con las sentencias `if`, `if-else`, y `switch`. En próximas secciones vamos a presentar esta alternativa.

1.3.1. Absoluto

$$\text{abs}:\mathbb{Z}\rightarrow\mathbb{Z}/\text{abs}(x)=|x|=\sqrt{x^2}=\begin{cases}-x & x<0\\ x & \text{e.o.c.}\end{cases}$$

```
int Abs(int);
```

```
assert( 42 == Abs(-42) );
assert(  0 == Abs(  0) );
assert( 42 == Abs( 42) );
```

```
int Abs(int x){return x<0 ? -x : x ;}
```

Como C++ no impone restricciones en la cantidad de espaciadores (i.e., espacio, tab, nueva línea), tan solo que deben estar entre *tokens* que de otra manera serían un solo token, una forma alternativa de formatear el código es la siguiente:

```
int Abs(int x){ return
```

```
 x<0 ? -x :
        x ;}
```

Este es el estilo que vamos a aplicar en el resto del texto, con el objetivo de ser más claro y establecer una analogía más fuerte entre matemática y C++. En matemática, perimero se escribe el resultado y luego la condición:

$$-x \quad x < 0$$

En C++, perimero se escribe la condición, y luego el resultado:

```
x<0 ? x
```

1.3.2. Step

```
   ^             /
   |            /
   |           /
   6      )---[
   |     /
   |    /
---3---[
   |
   |
<--0---4--7---11---->
```

$$\text{Step}:\mathbb{Z} \to \mathbb{Z} / \text{Step}(x) = \begin{cases} 3 & x < 4 \\ x - 1 & 4 \le x < 7 \\ 6 & 7 \le x < 11 \\ x - 5 & \text{e.o.c.} \end{cases}$$

```
double Step(double);
```

```
assert( 3   == Step(-2  ) );
assert( 3   == Step( 0  ) );
```

```
assert( 3   == Step( 3  ) );
assert( 3   == Step( 4  ) );
assert( 4.5 == Step( 5.5) );
assert( 6   == Step( 7  ) );
assert( 6   == Step( 8  ) );
assert( 6   == Step(11  ) );
assert( 8   == Step(13  ) );
```

```
double Step(double x){return
 x<4           ? 3   :
 4<=x and x<7  ? x-1 :
 7<=x and x<11 ? 6   :
                 x-5 ;}
```

1.4. Funciones de Varias Variables — Campo Escalar: $2^{+}\rightarrow 1$

1.4.1. Promedio

$$\text{avg}: \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{R} / \text{avg}(a, b) = \frac{a+b}{2}$$

El dominio $\mathbb{Z} \times \mathbb{Z}$ también puede denotarse como $\mathbb{Z}^2$.

```
double Avg(int,int); ❶
```

❶ Los parámetros se separan con coma. Los nombres de los parámetros son opcionales.

```
assert( 3   == Avg( 2, 4) ); ❶
assert( 1   == Avg( 1, 1) );
assert( 0   == Avg( 0, 0) );
assert( 0   == Avg(-1, 1) );
assert( 1.5 == Avg( 1, 2) );
```

❶ En la invocación, los argumentos también se separan con coma.

```
double Avg(int a, int b){ ❶
```

```
 return (a+b)/2.0; ❷
}
```

❶ Cada parámetro debe tener su tipo especificado, no es válido escribir `Avg(int a,b)`.

❷ La división es cerrada entre `int` s, por eso se divide por un `double`. La adición da un `int`, pero se convierte a un `double` implícitamente, se aplica *promoción de tipo* automática.

1.4.2. Máximo

$$\text{Max} : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z} / \text{Max}(a, b) = \begin{cases} a & a > b \\ b & \text{e.o.c.} \end{cases}$$

```
int Max(int,int);
```

```
assert(  0 == Max(  0, 0) );
assert( -1 == Max( -1,-1) );
assert( 42 == Max( 42,42) );
assert( 42 == Max( -2,42) );
assert( 42 == Max( 42,-2) );
assert( -2 == Max(-42,-2) );
```

```
int Max(int a,int b){return
 a>b ? a :
       b ;}
```

1.5. Funciones Sin Datos o Sin Resultados

Matemáticamente, las funciones toman un dato y producen un resultado. El dato es un elemento del conjunto dominio y el resultado del conjunto imagen. Existe un caso extremo donde al función no recibe un dato pero genera un resultado, es decir el dominio es vacío. Pero matemáticamente no existen funciones que no produzcan resultados, es decir, que tengan imagen vacía[EmptyFunctions]. Vamos a ver que programáticamente esas "funciones", sí existen, son frecuentes, y tienen utilidad.

1.5.1. Dominio Vacío: Funciones sin Datos (0→1)

A continuación vamos a ver dos ejemplos de funciones que retornan resultados sin recibir ningún dato como parámetro, es decir, su dominio está vacío.

Constante sin Parámetros

$$Const42 : \varnothing \rightarrow \mathbb{Z} / Const42() = 42$$

En la sección función constante con un parámetro vimos una versión de la definición, donde el dominio no era vacío pero el valor de la función era independiente del parámetro. A continuación vemos otra forma de definirla, esta vez, con dominio vacío.

Declaración.

```
int Const42();
```

Prueba.

```
assert( 42 == Const42() );
```

Definición.

```
int Const42(){return 42;}
```

La función no recibe argumentos pero puede generar un valor, en este caso 42, gracias que en la propia definición de la función está el valor 42 *hardcodeado*.

Solicitar Nombre

$$\text{SolicitarNombre} : \varnothing \rightarrow \Sigma^*$$

Esta función tampoco recibe argumentos, pero logra generar un valor al extraer un dato del objeto global `cin` mediante el operador `>>`. Es importante notar que esa extracción produce un *efecto de lado*, en este caso, cambio de estado en el objeto `cin`.

Declaración.

```
string SolicitarNombre();
```

Invocación.

```
cout << SolicitarNombre(); // envía a cout el nombre retornado por la
 función SolicitarNombre.
```

Definición.

```
string SolicitarNombre(){
 cout << "Ingrese un nombre simple: "; // No acepta nombre con espacios.
 string nombre;
 cin >> nombre;
 cout << "\n";
 return nombre;
}
```

Transparencia Referencial y Efecto de Lado

La función `Const42` y todas las funciones vistas en las anteriores secciones tienen *transparencia referencial*[RefTrans] porque pueden ser reemplazadas por el valor que generan sin afectar el comportamiento del programa y su valor solo depende de sus parámetros.

En contraste, la función `SolicitarNombre` tiene opacidad; como en cada invocación su valor varía, no puede ser reemplazada por su valor sin afectar el comportamiento del programa. Adicionalmente, posee *efecto de lado*[SideEffect]. Esto implica que su invocación cambia el estado del sistema, en este caso, el objeto `cin`.

Las funciones con transparencia referencial y sin fecto de lado son llamadas *funciones puras*[PureFunc].

1.5.2. Imagen Vacía: Funciones sin Resultados ($0^{+}\rightarrow 0$)

Las funciones `void` son las funciones que no retornan resultados; tienen prototipos de la forma `void f(...);`. El tipo `void` es el tipo que no contiene valores, esto es, su conjunto de valores es vacío. Como estas funciones no

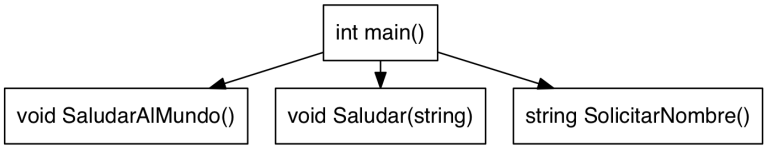
retornan un valor, se las invoca por su *efecto de lado*, es decir, por cambios que pueden producen sobre objetos de nuestros sistema.

Saludos

Como ejemplo introductorio vamos a desarollar un programa que invoque y defina, además de `string SolicitarNombre()`, las siguientes funciones funciones `void`:

- `void SaludarAlMundo()`: Envía `"Hola, Mundo!\n"` por `cout`.
- `void Saludar(string)`: Envía `"Hola, <nombre>!\n"` por `cout`, donde *nombre* es el string parámetro.

Una *Carta Estrcuturada* (sic) ó *Structure Chart*[StructChart] presenta la relación de invocación entre las funciones, es decir, "*quien llama a quien*". La siguiente es la carta estructurada del programa:



Este es el programa completo:

```
/* Hi
JMS
2018 */
#include <iostream>
#include <string>

using std::string; ❶
using std::cout;
using std::cin;

int main(){ ❷
 /*
 [stem]
 ++++
 "SaludarAlMundo": O/ -> O/
 ++++
 */
```

```
 void SaludarAlMundo(); ❸
 SaludarAlMundo();

 /*
 [stem]
 ++++
 "Saludar": Sigma* -> O/
 ++++
 */
 void Saludar(string);
 Saludar("Mundo");
 Saludar("León");

 /*
 [stem]
 ++++
 "SolicitarNombre": O/ -> Sigma^**
 ++++
 */
 string SolicitarNombre();
 Saludar(SolicitarNombre()); ❹
}

void SaludarAlMundo(){
 cout << "Hola, Mundo!\n";
}

void Saludar(string s){
 cout << "Hola, " << s << "!\n";
}

string SolicitarNombre(){
 cout << "Ingrese un nombre simple: "; // No acepta nombre con espacios.
 string nombre;
 cin >> nombre;
 cout << "\n";
 return nombre;
}
```

❶ Incorporamos al namespace global estos tres nombres.

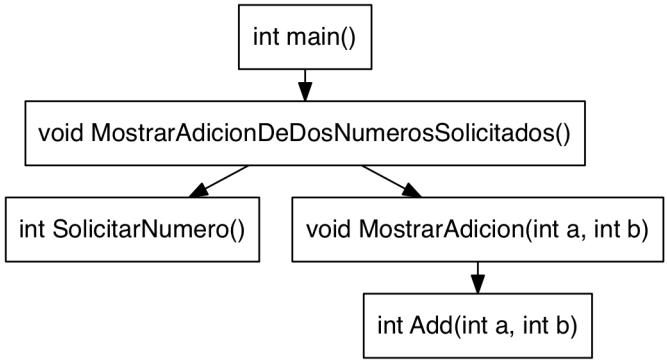
❷ En C++ la *definición* de `main` tiene paréntesis vacíos significa que la función no tiene parámetros; el significado en C es otro.

❸ En C++ la *declaración* de la función paréntesis vacíos significa que la función no tiene parámetros; el significa en C es otro.

❹ El resultado de `SolicitarNombre` es el argumento de `Saludar`.

Mostrar Adición de Dos Números Solicitados

El siguiente ejemplo es un poco más complejo que el anterior; su carta estructurada es más profunda ya que hay funciones que invocan a otras funciones. El programa resuelva el problema: Leer dos números de la entrada estándar y enviar el resultado de su adición a la salida estándar.




```
/* AskAddShow
JMS
2018 */
#include <iostream>

int main(){
 /*
 [stem]
 ++++
 "MostrarAdicionDeDosNumerosSolicitados": O/ -> O/
 ++++
 */
 void MostrarAdicionDeDosNumerosSolicitados();
 MostrarAdicionDeDosNumerosSolicitados();
}

void MostrarAdicionDeDosNumerosSolicitados(){
 /*
 [stem]
 ++++
 "SolicitarNumero": O/ -> ZZ
 ++++
```

```
 [stem]
 ++++
 "MostrarAdicion": ZZ xx ZZ -> O/
 ++++
 */
 std::cout << "Ingrese dos números enteros para conocer su suma:\n";
 int SolicitarNumero();
 void MostrarAdicion(int, int);
 MostrarAdicion(
  SolicitarNumero(),
  SolicitarNumero()
 ); ❶
}

int SolicitarNumero(){
 std::cout << "Ingrese un número entero: ";
 int n;
 std::cin >> n;
 std::cout <<  '\n';
 return n;
}

void MostrarAdicion(int a, int b){
 /*
 [stem]
 ++++
 Add:ZZ xx ZZ -> ZZ // Add(a,b)=a+b
 ++++
 */
 int Add(int, int);
 std::cout << "La suma es: " << Add(a,b) << '\n';
}

int Add(int a, int b){return a+b;}
```

❶ Esta invocación es especial. Para resolverla primero hay que resolver las dos invocaciones a `SolicitarNumero`. Los strings resultantes de sendas inovaciones a `SolicitarNumero` son los argumentos a `MostrarAdicion`.

2

Recursividad en Funciones Puras

Las funciones recursivas son las que en su definición incluyen por lo menos una referencia, ya sea directa o indirecta, a la misma función. Por eso las funciones recursivas denotan *repeticiones*.

> ## Iteración
>
> Otra alternativa para representar repetición es mediante *control de flujo de ejecución estructurado*, en particular con la estructura *iteración*, que C ++ implementa con las sentencias `for`, `while` , y `do-while`. En próximas secciones vamos a presentar esta alternativa.

2.1. Sumatoria de los Primeros *n* Naturales

$$\text{Sum} : \mathbb{N} \rightarrow \mathbb{N} / \text{Sum}(n) = \begin{cases} 0 & 0 \\ n + \text{Sum}(n - 1) & \text{e.o.c.} \end{cases}$$

```
unsigned Sum(unsigned);
```

```
assert(          0 == Sum(      0) );
assert(          1 == Sum(      1) );
assert(          3 == Sum(      2) );
assert(         10 == Sum(      4) );
assert(         55 == Sum(     10) );
assert(  705082704 == Sum( 100000) ); // 100K
```

```
//assert(1784293664 == Sum(1000000)); // 1M -> Segmentation fault.
```

```
unsigned Sum(unsigned n){return
 n == 0 ? 0           :  ❶
          n + Sum(n-1);} ❷
```

❶ Caso base, el caso "terminador".

❷ Caso recursivo.

3

Parámetros in, out e inout

4

Funciones y Campos Vectoriales

$(0^{+} \rightarrow 2^{+})$

Bibliografía

https://en.wikipedia.org/wiki/Test-driven_development.

https://math.stackexchange.com/questions/789123/why-is-there-no-function-with-a-nonempty-domain-and-an-empty-range

https://en.wikipedia.org/wiki/Side_effect_(computer_science)

https://en.wikipedia.org/wiki/Referential_transparency

https://en.wikipedia.org/wiki/Pure_function

https://en.wikipedia.org/wiki/Structure_chart