# Funciones

Esp. Ing. José María Sola, profesor.

Revisión 2.0.0
2023-05-29

# Tabla de contenidos

# 1

# Introducción a Funciones

Comenzamos con la *función lineal* como un ejemplo introductorio muy simple que refleja las grandes similitudes entre el concepto de función en matemática y en ciencias de la computación y sirve para presentar los conceptos principales, Usemos como ejemplo una .

## 1.1. Especificación Matemática

La *especificación* matemática indica *qué hace* la función mediante expresiones matemáticas, no indica *cómo hace* para lograr el resultado en lenguaje de programación.

Repasemos las dos partes de la especificación matemática partes, separadas por el *tal que* ( $/$ ):

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

1. Primero tenemos el nombre de la función, $f$, junto con los conjuntos que relaciona: $\mathbb{R} \rightarrow \mathbb{R}$.
2. Luego, la fórmula que define la relación: $f(x) = 2x + 1$

## 1.2. Implementación en C++

Su contrapartida en C++ también tiene en dos partes:

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

❶ La *Declaración* de la función ó *prototipo*, que indica tipo de la función, nombre y tipo de sus parámetros; los nombres de los parámetros son opcionales.

❷ La *Definición* o *implementación* de la función, que defina el *método*, *forma*, o *manera* de calcular los resultados; análogo a la fórmula matemática. Debe incluir los nombres de los *parámetros* que se usan (pueden no usarse todos) y el *cuerpo* de la función que debe estar entre llaves e incluir el *retorno* de un valor (ya vamos a ver más adeltante que hay funciones que no *retornan* resultados).

## 1.3. Aplicación o Uso de la Función

Matemáticamente, la *aplicación* de la función es el remplazo del *parámetro* o *variable libre* por un *argumento* que es un valor perteneciente al dominio. En este ejemplo de aplicación buscamos calcular $f(3)$, lo representamos con dos ecuaciones:

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

En C++ esta acción se la llama *invocación*, y en el siguiente ejemplo usamos `assert` para modelar las ecuaciones matemáticas y probar nuestro desarrollo:

```
assert( f(3) == 2*3+1 ); ❶
assert( 2*3+1 == 7 );
```

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

## 1.4. Estructura del Programa en C++

Este es el programa completo en C++:

```
/* f
[asciimath_]
++++
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( f(3) == 2*3+1 ); ❹
 assert( 2*3+1 == 7 );
}

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

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

❷ *Declaración* de la función.

❸ *Invocación* de la función, con un *argumento*. En este caso, el valor resultante de la invocación se lo usa como operando del operador inserción (`<<`) para enviarlo a `cout`.

❹ *Invocación* como parte de una afirmación o ecuación (`assert`) para verificar que la implementación de nuestra función es correcta.

❺ *Definición* o *implementación* de la función.

C++ requiere que antes de invocarse a una función, esa función tenga un prototipo que indique como invocarla correctamente. En C++ ese prototipo se logra mediante una declaración o mediante una definición.

El ejemplo usa declaraciones para eso, así vemos que la declaración está antes de la invocación, en este caso, antes de `main`; mientras que la definición se ubica debajo de `main`.

¿Qué ocurre si removemos la declaración y ubicamos la definición antes de la invocación? Si nuestro programa tiene muchas funciones definidas, que es el caso común, la función `main` quedaría relegada al final de nuestro código fuente,

y no es un estilo esperado en C++ idiomático. Por otro lado, usar la definición como prototipo impone restricciones en el orden en que ese definen, orden que no puede ser definido para funciones mutuamente recursivas.

Por eso, usamos el estilo del C++ idiomático: primero prototipos, seguido de la definición de `main`, seguido por la definición de las funciones.

## 1.5. Proceso de Desarrollo

Este proceso tiene como objetivo lograr una implementación de una función a partir de una especificación matemática. Aunque usamos C++, este proceso es aplicable a cualqueir lenguaje de programación compilado.

1. **Especificación** matemática.
2. Diseño de casos de **prueba**.
3. Codificación de las ecuaciones o `assert` (aseveraciones) en C++ que invoca la función según los casos de prueba diseñados. Intentar compilar, el error principal debe ser que la *función no está **declarada***.
4. Declaración del **prototipo**. Debe haber concordancia entre las pruebas y el pruebas y el prototipo. Intentar compilar, si hay concordancia, el error principal debe ser que la *función no está **definida***.
5. Definición de la función. Por último, y debajo de la función `main` codificar la definición de la función que implementa el método de cálculo.
6. **Compilar**. Si hay error revisar los pasos anteriores.
7. **Ejecutar**. El programa debería finalizar sin mensajes, si hubo algún mensaje de error es porque las aseveraciones fallaron, lo cual estamos ante la presencia de un *bug*. Debemos revisar si hay concordancia entre la especificación matemática, las pruebas, y la implementación en C++.

## 1.6. Conclusiones

Comparación entre Matemática y C++

| **Matemática** | **C++** |
|---|---|
| Relación: Dominio, Codominio | Prototipo: tipo de la función, nombre, y tipo de los parámetros. |

| **Matemática** | **C++** |
|---|---|
| $f: \mathbb{R} \to \mathbb{R}$ | `double f(double);` |
| Ecuaciones | Aseveraciones con `assert` para hacaer pruebas |
| $f(3) = 7$ | `assert( f(3) == 7 );` |
| Fórmula | Definición |
| $f(x) = 2x + 1$ | `double f(double x){return 2*x+1;}` |

Podemos sintetizar el *proceso de desarrollo* en estos pasos:

1. Especificar.
2. Escribir las pruebas con `assert`.
3. Implementar.
4. Compilar, ejecutar, y evaluar resultados.

Continuamos la introducción tres conceptos nuevos y ejemplos que los ilustran:

- Funciones escalares.
- Funciones partidas.
- Funciones de varias variables.

# 2

# Funciones y Campos Escalares

Estas son funciones que retornan exactamente un valor.

## 2.1. Funciones de Escalares (1→1)

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

### *2.1.1. Constante*

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

```
#include <cassert>
#include <iostream>

int y(int); ❶

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

int y(int){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 porque el parámetro no se usa, y se puede usar cualquier valor como argumento.

## Parámetros que no se usan

La función `y` tiene un parámetro pero que no los usamos Si en la definición le damos un nombre los compiladores, *generalmente*, emiten un *warning* (advertencia) indicando que el parámetro no se usa.

```
int y(int x){return 42;} // WARNING: paramenter x is unused.
```

Una alternativa es *ignorar* el warning pero, aunque lamentablemente habitual, es una mala práctica porque los warnings puede ser en realidad un *errores pragmáticos* (i.e., *bugs*).

Otra alternativa es indicar al compilador que no analice esta situación; y lo podemos hacer de dos formas:

1. Por medio de opciones al momento de compilar, lo cual acarrea varias desventajas:
   - El tratamiento de la situación ya no está evidenciado en el código.
   - No sabemos si el compilador que estemos usando tiene esa opción.
   - Generalizamos la situación a todas las funciones y no solo a una particular.
2. La otra forma de indicar al compilador que no analice la situación es mediante el *atributo* `[[maybe_unused]]` justo antes del parámetro. Aunque mucho más simple y directo, este atributo no fue pensado para este caso de uso, y lamentablemente no existe `[never_used]]` o similar.

La solución que aplicamos es parte de C++ moderno, y el lenguaje C la está por incorporar: *Parámetros sin nombre*. Se indica el tipo, pero no se nombra el parámetro, muy similar a los nombres de parámetros opcionales de las declaraciones.

Más adelante, en Sección 3.1.1, “Constante sin Parámetros”, vamos a ver una alternativa más, que va a ser la definitiva.

### 2.1.2. Identidad

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

```
#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`.

### 2.1.3. 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;}
```

### *2.1.4. Negación*

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

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

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

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

### *2.1.5. Twice (Dos Veces)*

$$\text{Twice}:\mathbb{Z}\to\mathbb{Z}/\text{Twice}(n) = 2n$$

```
int Twice(int);
```

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

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

## 2.2. 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.

### *2.2.1. 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 ;}
```

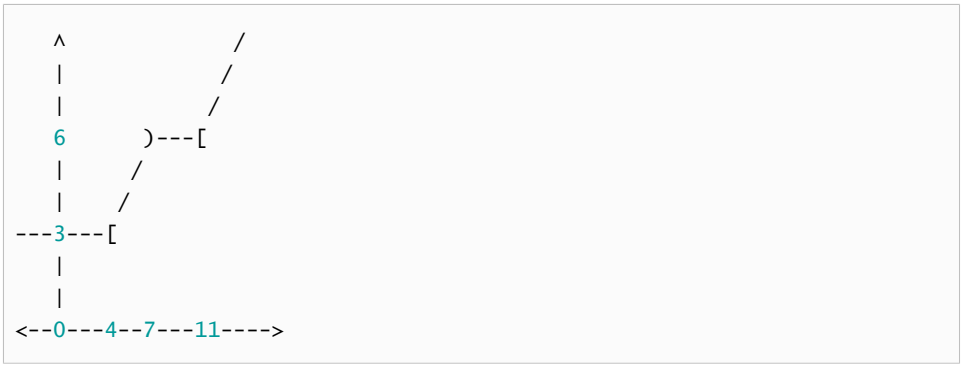
Este es el estilo que vamos a aplicar en el resto del texto, con el objetivo de ser más claro y establecer una analogía más fuerte entre matemática y C++. En matemática, primero se escribe el resultado y luego la condición:

$$-x \qquad x < 0$$

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

```
x<0 ? x
```

### *2.2.2. Step*




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

```
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 ;}
```

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

### *2.3.1. 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.

### 2.3.2. 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 ;}
```

# 3

# Funciones Sin Datos ni Resultados y Efecto de Lado

Matemáticamente, las funciones toman un dato y producen un resultado. El dato es un elemento del conjunto dominio y el resultado del conjunto imagen.

En programación, cuando una función sigue la regla que sus resultados dependen exclusivamente de sus datos y que no produce *efectos de lado* (efectos colaterales, *side effects*) [SideEffect], decimos que es una *función pura* [PureFunc]. Una función pura es ídentica al concepto matemático de funciones.

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 en el contexto de la programación esas *"funciones"*, sí existen, son frecuentes, y tienen utilidad; también vamos analizar si realmente no reciben datos ni producen resultados.

## 3.1. 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.

### *3.1.1. 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.

```
int Const42();
```

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

```
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*.

## C vs C++: Sin Parámetros

Para todas las versiones de C++, una lista de parámetros vacía en la *declaración* o *definición* **siempre** significó que no se esperan argumentos. Para C es así sólo a partir de la versión 2023.

```
int f();     //   C++: Sin parámetros
int f(void); //     C: Sin parámetros
int f();     //  C<23: Sin chequeo de argumentos
int f();     // C>=23: Sin parámetros
```

### *3.1.2. 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 `>>`.

**Cambio de estado y argumentos implícitos**

> Es importante notar que esa extracción produce un *efecto de lado*, en este caso, cambio de estado en el objeto `cin`.

> Analicemos esta situación, si la función tiene acceso a `cin` y modifica su estado, ¿no es el estado de `cin` un argumento implícito de la función? Y de la misma manera, el nuevo estado de `cin` ¿no es un resultado de la función?

```
string SolicitarNombre();
```

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

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

## 3.2. 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 argumentos.

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*. 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*.

## 3.3. 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.

### *3.3.1. Saludos*

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

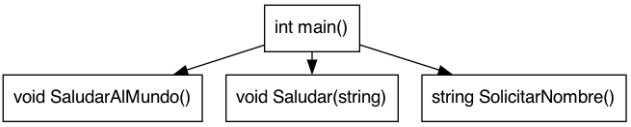
1. $$\text{SaludarAlMundo} : \varnothing \rightarrow \varnothing$$

   `void SaludarAlMundo()`: Envía "`Hola, Mundo!\n`" por `cout`.

2. $$\text{Saludar} : \Sigma^* \rightarrow \varnothing$$

   `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
```

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


 void Saludar(string);
 Saludar("Mundo");
 Saludar("León");


 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.

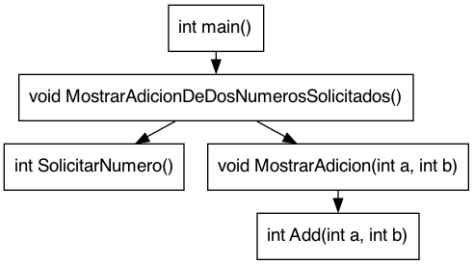
❷ En C++ la *definición* de `main` tiene paréntesis vacíos significa que la función no tiene parámetros.

❸ En C++ la *declaración* de la función paréntesis vacíos significa que la función no tiene parámetros.

❹ El resultado de `SolicitarNombre` se usa como argumento de `Saludar`.

### *3.3.2. 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(){
 void MostrarAdicionDeDosNumerosSolicitados();
 MostrarAdicionDeDosNumerosSolicitados();
}

void MostrarAdicionDeDosNumerosSolicitados(){

 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){
 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 invocaciones a `SolicitarNumero` son los argumentos a `MostrarAdicion`.

# 4

# 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.

## 4.1. 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 Paso inductivo, recursivo.

# 5

# Rol de los Parámetros: In, Out, e InOut

Los parámetros de una función cumplen el rol de ser los datos, es decir lo *dado* como *input* (entrada), a la función para calcular su resultado. Asimismo, el *output* (salida) o resultado del cálculo es el valor de la expresión de invocación.

Pero vamos a ver que los parámetros pueden utilizarse también como otro como mecanismo para generar el *output* (resultado o salida) de la función.

Es decir que expandimos los posibles roles de los parámetro. Un parámetro puede tene el rol de *input* pero ahora también podemos crear parámetros con rol de *output*, e inclusive también parámetros que cumplen dos roles a la vez: *input-output*.

Al *rol del parámetro* también se lo conoce como *modo del parámetro*. Los roles o modos de los parámetros toman los siguientes nombres cortos: *in*, *out*, e *inout*.

Como un ejemplo simple de uso de roles de parámetros vamos utilizar la función *Sumar*, que tiene la siguiente especificación:

$$Suma : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z} / Suma(a, b) = a + b$$

Y a continuación vemos una declaración, una invocación, y una definición válida, siguiendo los conceptos que vimos antes.

```
#include <cassert>
int main(){
 int Suma1(int,int);
```

```
 int i{5};
 assert( 7 == Suma1(2,i) );
}
int Suma1(int a, int b){
 return a+b;
}
```

En el anterior ejemplo, los parámetros `a` y `b` cumplen el rol *in*, y la evaluación de la expresión de invocación es el resultado de la función, que se implementa como el *valor de retorno de la función*.

## 5.1. Parámetro Out

En este ejemplo, el tercer parámetro es *out*.

```
#include <cassert>
int main(){
 void Suma2(int,int,int&);
 int s;
 Suma2(2,5,s);
 assert( 7 == s );
}
void Suma2(int a, int b, int& c){
 c=a+b;
}
```

La declaración indica que espera tres argumentos, los primeros dos son `int` y el segundo es una referencia a un `int`. La función no retorna valor por que lo *devuelve* por el tercer argumento.

En la invocación el tercer argumento no es dato, es el lugar donde se va a alojar el resultado o salida de la función. La variable `s` se declara pero no se incializa porque su valor va a ser escrito por por la función.

El parámetro `c` es una referencia a la variable `s`, así que si modificamos `c` estamos modificando `s`.

Para pensar. ¿Es correcto inicializar `s`? ¿Por qué? ¿Puede el compilador detectar la situación?

# 6

# Funciones y Campos Vectoriales

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

# 7

# Pasaje de Argumentos

# 8

# Changelog

**2.0.0+2023-05-29**

- Reestructuración y extensión del del paper.
- Se ocultan la mayoría de las notaciones en `asciimath`.
- Más explicaciones en la introducción
    - Especificación vs. implementación
    - Estructura del programa.
    - Terminología
    - Proceso de desarrollo
    - Se explica parámetros no usados.
    - Comparación entre Matemática en C++
- La explicación de `assert` se movió a un paper todavía no publicado.
- La sección "Funciones Sin Datos o Sin Resultados" tienen su propio capítulo.
- Se renombró la función `Doble` a `Twice` porque se asemejaba mucho al tipo `double`.
- Notaciones en C y C++ de lista de parámetros vacía.
- Se detalla el concepto de efecto de lado y función pura.
- Nueva sección: Rol de los Parámetros: In, Out, e InOut.

**1.0.0+2018-05-14**

- Versión inicial.

# Bibliografía


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

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

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