Capítulo 4

**Introducción**

Ya sabemos realizar saltos (condicionales o no) para lograr que el microprocesador ejecute instrucciones situadas en **otras posiciones de memoria**. En esta sesión aprenderemos a hacer **subrutinas**, que son trozos de código que realizan una **función específica**, y que al terminar **devuelven el control** a punto siguiente en el que **fueron invocadas**. Esto es importantísimo, porque nos permite **reutilizar** código fácilmente, y **organizar** nuestros programas. Para resolver un problema complejo, **dividimos** el código en trozos más pequeños: **las subrutinas**

Las subrutinas es lo que permite que en los **lenguajes de alto nivel** (python, C, Ada, C++...) se implementen **funciones**, **procedimientos** o **métodos**. Por eso usaremos indistintamente los términos **subrutina** o **función** para referirnos al mismo concepto. Se podría decir que una subrutina es una función en alto nivel.

## Ideas claves: Organización y reutilización

Las **funciones** tienen dos misiones fundamentales:

* **Organizar y estructurar** nuestro código: Aplicamos la idea de "divide y vencerás". Para resolver un problema complejo lo dividimos en pequeños trozos (funciones). Cada función realiza una operación muy concreta
* **Reutilizar el código**: El mismo código lo podemos invocar desde diferentes partes de nuestro programa, ahorrando memoria, y haciendo más fácil el mantenimiento y depuración de los programas

Esto es válido en **cualquier lenguaje de programación**. Cuando trabajamos en ensamblador, las funciones se llaman **subrutinas**

## Programa principal y una subrutina

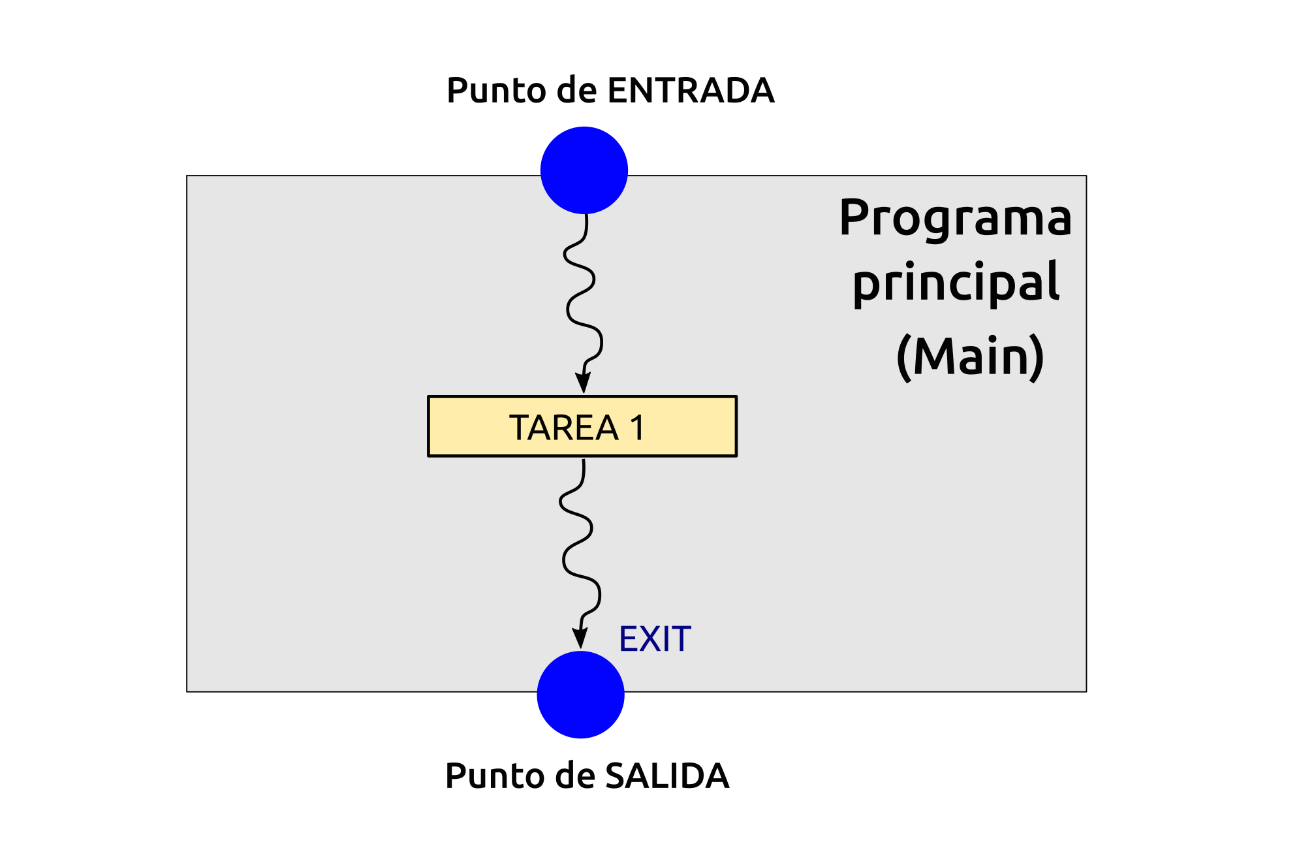
Para comprender el **mecanismo de funcionamiento de las subrutinas** partiremos de lo que ya conocemos e iremos introduciendo el resto de conceptos poco a poco. Partiremos de un **ejemplo** en el que hay que realizar una **tarea genérica** y terminar. Esta tarea puede ser cualquiera, pero la representaremos mediante la **impresión en la consola** del mensaje **"TAREA 1"**

Vamos ir viendo las diferentes **formas de organizarlo**

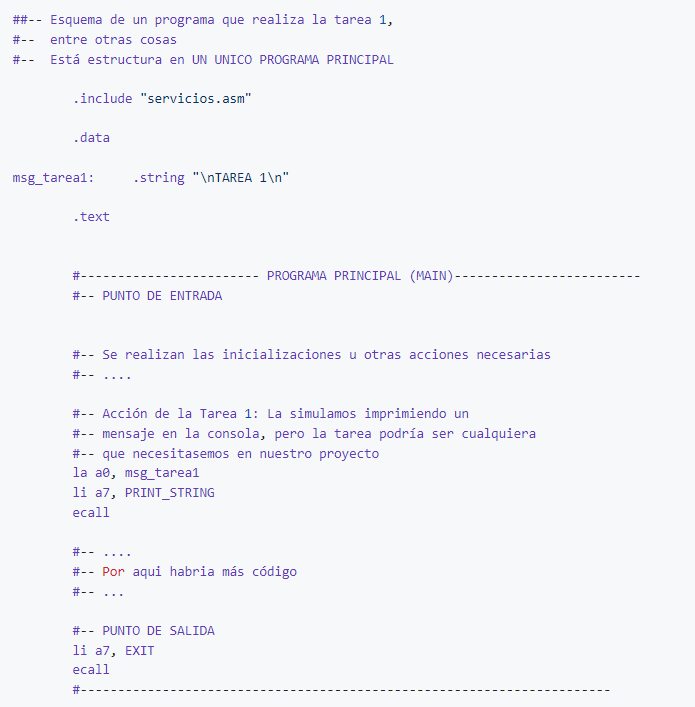
### Programa principal

Definimos **programa principal** como el **código** que está comprendido entre el **PUNTO DE ENTRADA** y **EL PUNTO DE SALIDA**. El sistema operativo pasa el control a nuestro programa, y se empieza a ejecutar su **primera instrucción** (que en el caso del simulador RARs sabemos que la primera del segmento de código: 0x00400000). Luego realiza la **Tarea 1** y finalmente termina

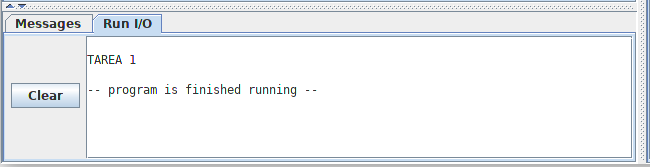
Esto lo **esquematizamos** con el siguiente **diagrama**



El **programa** de ejemplo, que realiza la **tarea 1**, es el siguiente:



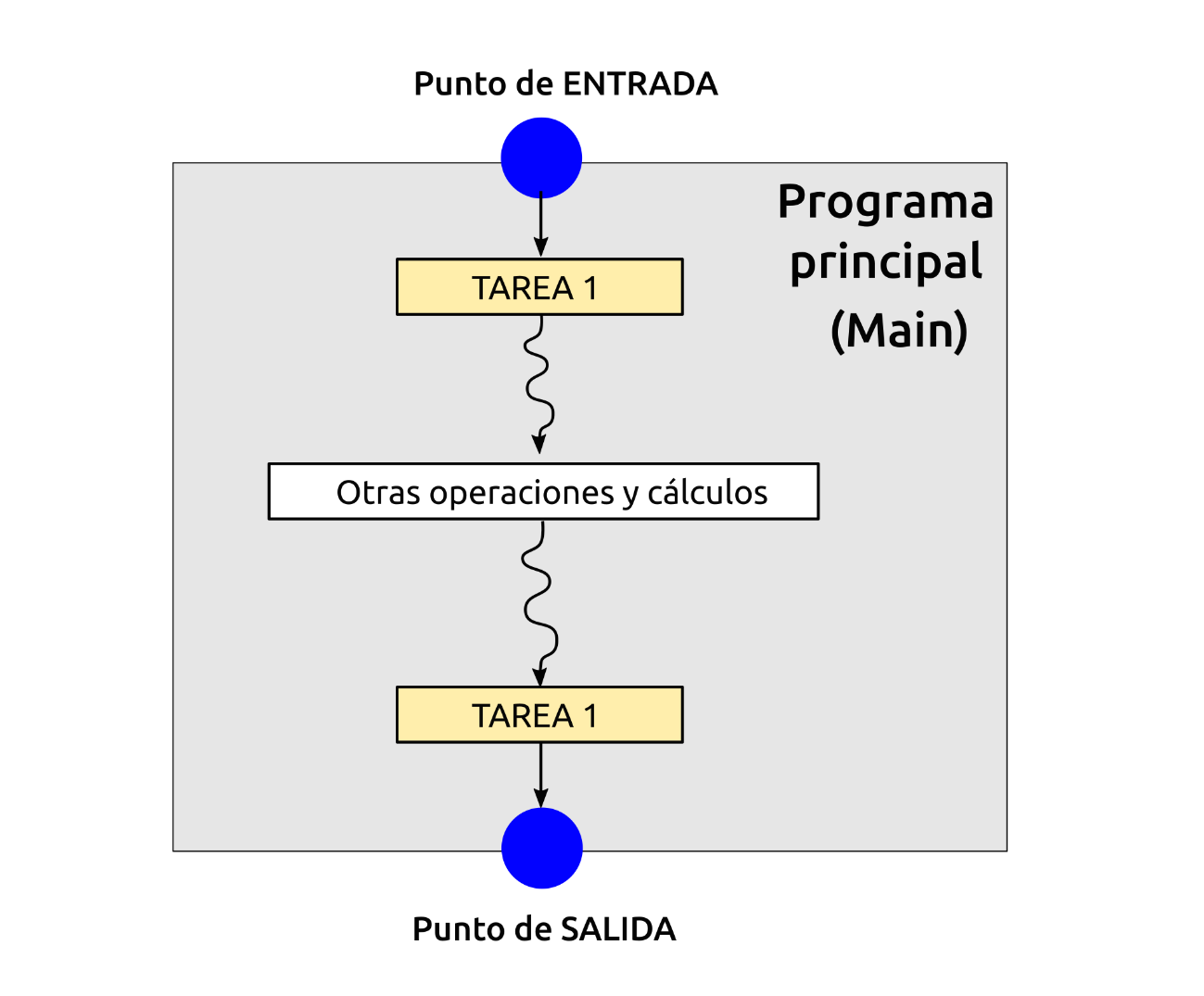
Lo **ensamblamos** y lo **ejecutamos**. En la consola vemos el mensaje de que la TAREA 1 se ha ejecutado



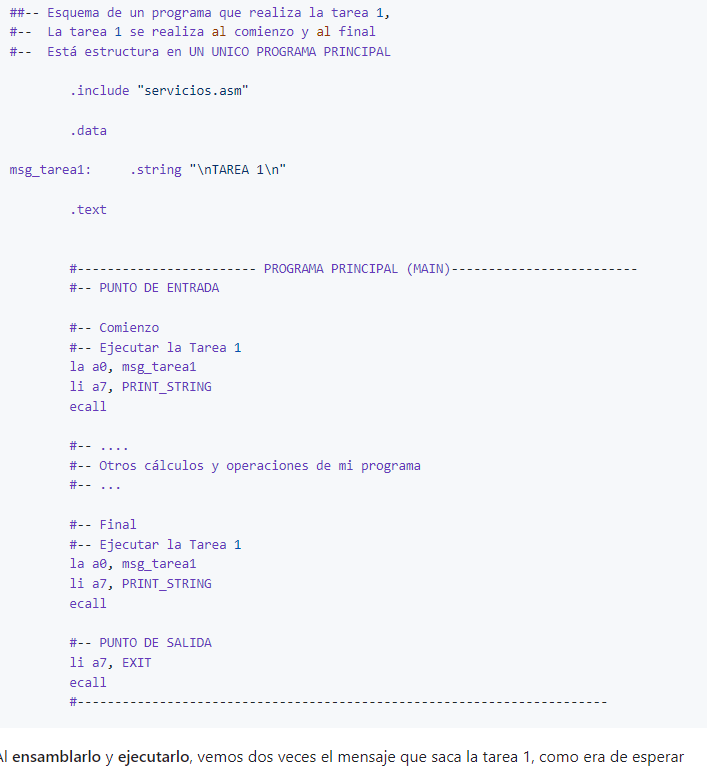
### Repetición de la tarea 1

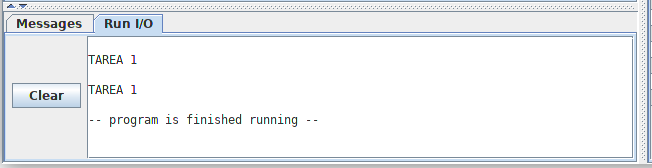
Supongamos que ahora nuestro proyecto ha sufrido **modificaciones** y es necesario **ejecutar la Tarea 1 dos veces**: una al **comenzar**, y otra al **terminar**

Con lo que sabemos hasta ahora, nuestra única opción es hacer "copy" y "paste" del **código de la tarea 1**. El **esquema** quedaría ahora así:



Y este es el **código**:





### Análisis del programa

Como se trata de un ejemplo muy sencillo, y la tarea 1 tiene muy pocas líneas de código, nos puede parecer una solución correcta. Sin embargo **NO LO ES**. Tiene los siguientes **problemas**:

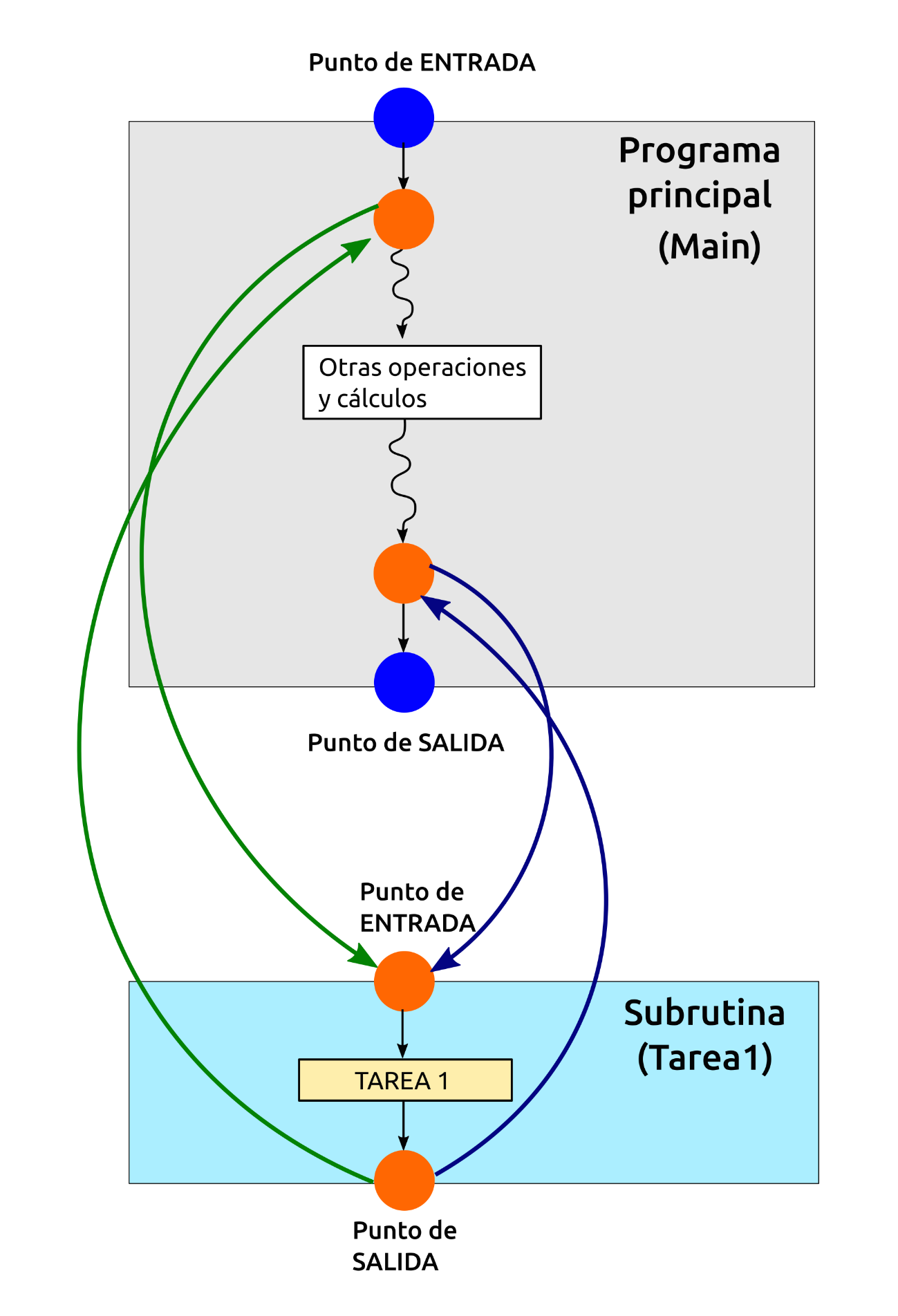
* **Duplicación de código**: En dos partes del programa estamos duplicando el código, por lo que nuestro programa va a aumentar de tamaño. Si la tarea 1 fuese más compleja y tuviese muchísimas instrucciones, crecería mucho en tamaño
* **No escalable**: ¿Y si ahora fuese necesario ejecutar la tarea 3 veces? Habría que triplicar su tamaño... Esta solución, no escala. No es general
* **Difícil de mantener**. Si hay que modificar la tarea 1 para que añadir más funcionalidad... hay que hacer en dos sitios. Esto es propenso a errores y es una **mala práctica de programación**

### **Solución: La tarea 1 es una SUBRUTINA**

El **diseño correcto** es convertir la **tarea 1** en una **función** o **subrutina**: Es un fragmento de código que realiza una **tarea específica**, y que el programa principal debe invocar cuando considere necesario. El nuevo diseño de mi programa está dividido en **dos partes**:

* **El programa principal**: Es el que se comienza a ejecutar, llama a la función tarea1 al comienzo realiza los cálculos pertinentes, vuelve a invocar la tarea1 y termina (punto de salida)
* **Una subrutina para realizar la tarea 1**

El **esquema** es el siguiente:



Ahora el **programa principal** realiza una **llamada a la subrutina**. Se ejecuta la **tarea 1**. Y esta **devuelve el control** al punto del programa principal desde donde se **invocó**

El **programa principal** realiza el resto de operaciones. Y por último vuelve a **invocar** a la subrutina tarea 1. Cuando termina, le **devuelve el control** al mismo punto desde donde fue invocada

Estas son las **ideas importantes**:

* La **subrutina** tiene un **ÚNICO PUNTO DE ENTRADA**, y un **ÚNICO PUNTO DE SALIDA**. Por supuesto, igual que ocurre con el programa principal, podría haber varios puntos de salida. Sin embargo, es una **buena práctica de programación** el tener sólo un único punto de salida
* La **subrutina** devuelve el control al punto desde donde fue invocada. Por tanto, existe un mecanismo que debe **recordar** a qué dirección **retornar**. ¿Cuál es este mecanismo?

Con este nuevo esquema se **solucionan** los problemas anteriores. Ahora sólo hay un **único código** que ejecuta la tarea 1 (NO está duplicado). El programa principal simplemente lo invoca dos veces, pero no está duplicado en memoria. Si es necesario, el programa principal lo puede **invocar tantas veces como quiera**, sin incrementar apenas el tamaño del programa

Y por supuesto, esta subrutina es más **fácil de mantener**. Normalmente las subrutinas las implementan personas diferentes. El **jefe de proyecto** hace la división de las tareas, y cada ingeniero se encarga de asegurar que su subrutina funciona correctamente

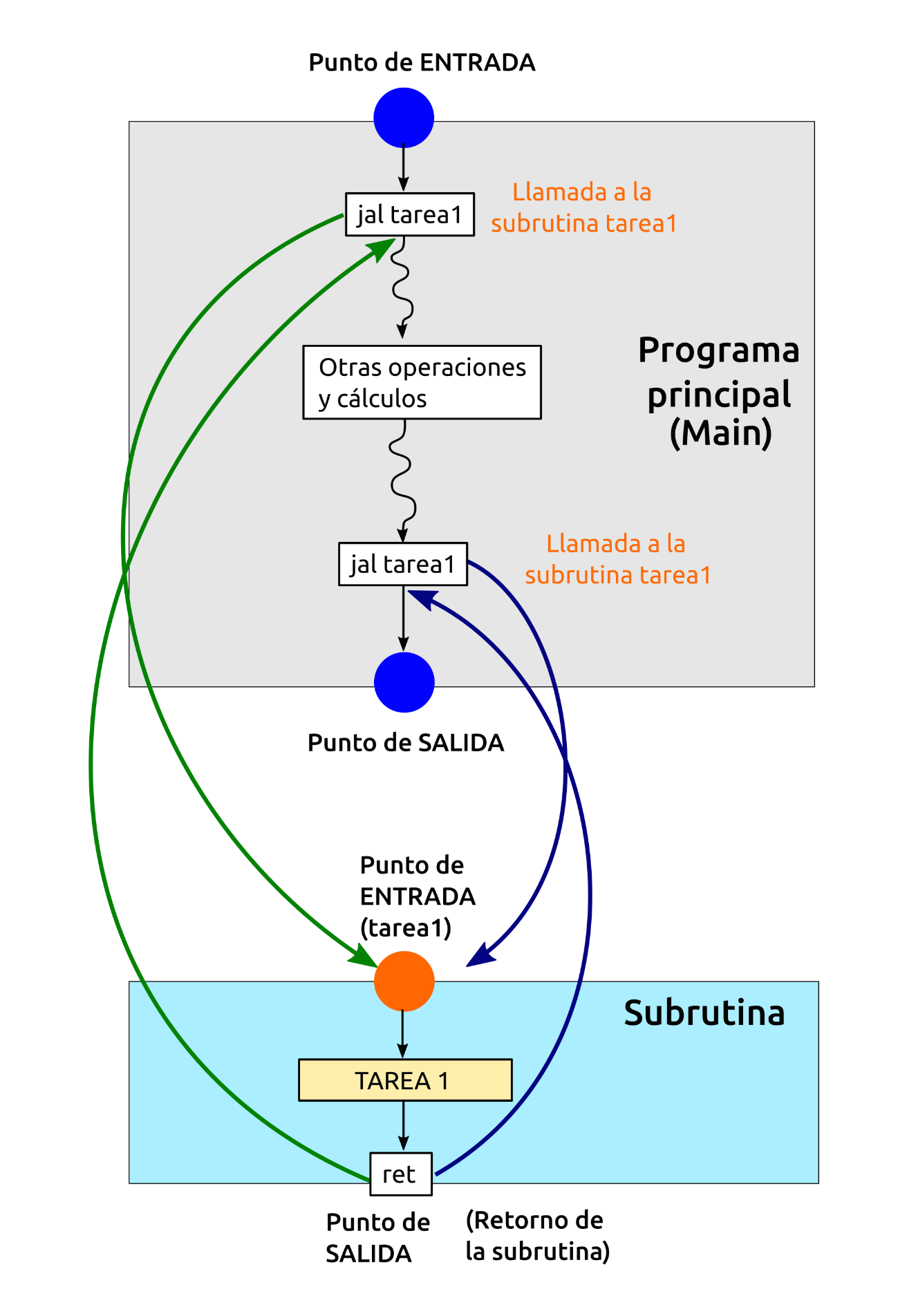
### **Las instrucciones jal/ret**

El **punto de entrada** de una subrutina lo definimos mediante una **etiqueta**, como hacemos siempre. Es la dirección donde está la **primera instrucción de la subrutina**

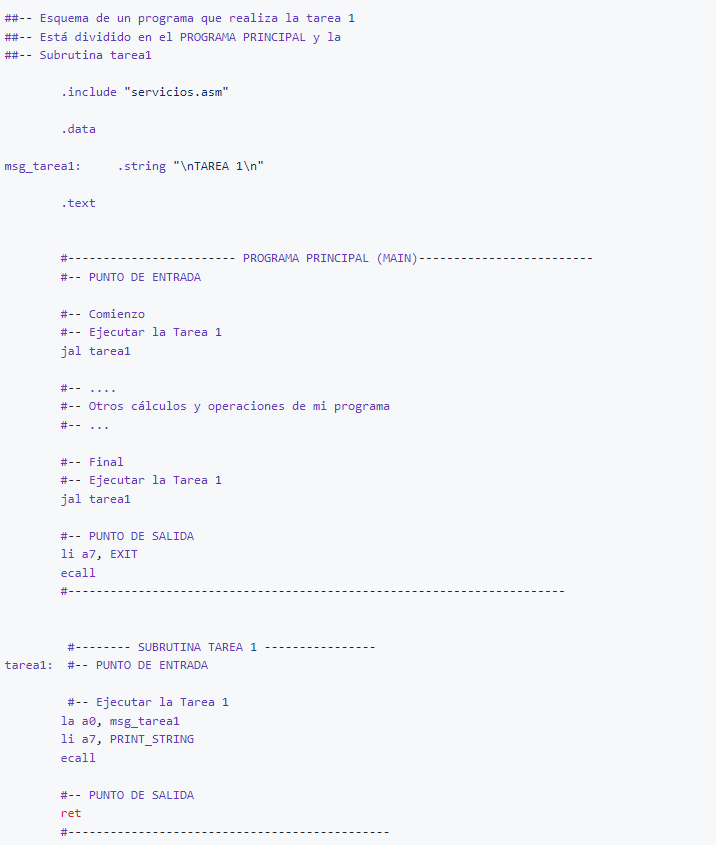
La **llamada** se realiza con la **instrucción jal**. Esta instrucción realiza un **salto incondicional** a la subrutina, pero **almacena la dirección de retorno**: es la dirección de la siguiente instrucción a jal

El **punto de salida** de la subrutina se define con la **instrucción ret**. Cuando se ejecuta esta instrucción, se realiza un **salto incondicional** a la dirección de retorno, que había sido almacenada previamente por la instrucción jal

Estas ideas se esquematizan en esta **figura**



El ejemplo anterior lo vamos a separar en el **programa principal** y la **subrutina tarea1**. Este es el **código**:



### **La magia de la instrucción ret**

**Ejecutamos** el programa anterior **paso a paso**. La **magia de las subrutina**s está en que al ejecutar la **instrucción ret**, siempre **se retorna a la instrucción siguiente al jal** usado para su invocación. Así, el primer jal está en la línea 19 (Dirección 0x00400000). Se invoca tarea1 y cuando termina devuelve el control a la siguiente instrucción, que está en la línea 27 (Dirección 0x00400004)

La **siguiente instrucción** es otro **jal**, que vuelve a llamar a la **subrutina tarea1**. Al ejecutar el ret, ahora se **devuelve el control** a la instrucción de la línea 30 (dirección 0x00400008)