75.40 Algoritmos y Programación I Curso 4
Introducción a la Modularización en C

Dr. Mariano Méndez
Juan Pablo Capurro
Martin Dardis
Matias Gimenez

Facultad De Ingeniería. Universidad de Buenos Aires

23 de agosto de 2022

1. Divide et Impera!

Hace ya miles de años los romanos habían comprendido que una estrategia efectiva que podía ser aplicada a la política y a la guerra se plasmaba en la frase "Divide et impera" (Divide y Vencerás), la autoría de la misma se le otorga al emperador romano Julio César. El concepto básico detrás de esta frase es la siguiente: esta técnica permite a un poder central compuesto por un número relativamente pequeño de personas, gobernar y dominar a una población mucho más numerosa, y de una forma relativamente simple.

Esta idea puede ser utilizada también como técnica de resolución de problemas. Dado un determinado problema complejo de resolver, la idea detrás de "divide et impera" implica que la resolución del mismo se obtiene dividiendo el problema inicial en partes (o subproblemas) más simples tantas veces como sea necesario, hasta que la solución de cada una de las partes sea obvia. Una vez encontrada la solución de cada una de las partes (o subproblema) el problema inicial queda resuelto por la composición o combinación de la solución de cada una de las partes.

Este método de resolución de problemas puede ser descrito como un algoritmo.

1.1. Programación Top-down

Esta técnica de programación conocida como top-down (descendente) basa la construcción de un programa a partir de una especificación general o de alto nivel de lo que el mismo debe hacer, posteriormente se descompone esta especificación en piezas más y más sencillas hasta alcanzar un nivel que corresponda a las **acciones primitivas** del lenguaje en el cual se implementará el programa. Una **acción primitiva**, es *toda aquella acción que puede ejecutarse sin necesidad de algún tipo de explicación para aquel que la ejecuta*. Se debe tener en cuenta que una acción primitiva en un lenguaje de programación, puede no serlo en otro.

Básicamente es el concepto propuesto por el César en la frase "divide et impera", ver Figura **??**. Esta técnica de programación sustenta sus principios en:

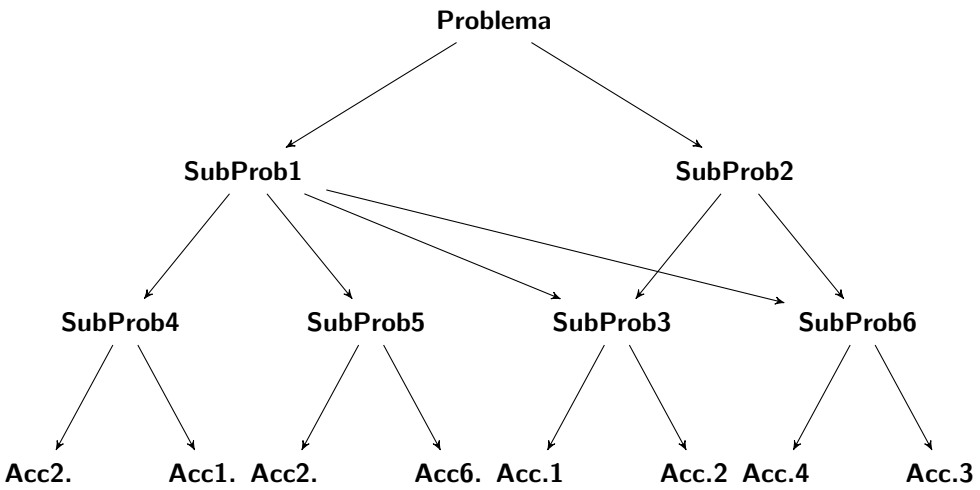

Figura 1: programación Top-Down

1. La descomposición o la partición.
2. Los refinamientos sucesivos.

Uno de los promotores de esta técnica fue Niklaus E. Wirth, y la plasmó en el artículo Program Development by Stepwise Refinement [**?**].

Desventajas:

1. Complica la realización de pruebas.
2. El programa ejecutable completo se obtiene muy al final del proceso de desarrollo.
3. Tiende a generar una partición del problema muy específica y relacionada a ese caso en particular.

1.2. Programación Bottom-up

La programación Bottom-up o ascendente por el contrario es una técnica de programación que a partir de **acciones primitivas** del lenguaje construye poco a poco acciones cada vez más complejas hasta llegar a obtener todas las necesarias para la construcción del programa, ver Figura **??**. Este tipo de técnica se utilizará hacia el final de la materia y se continuará profundizando en los cursos posteriores de programación.

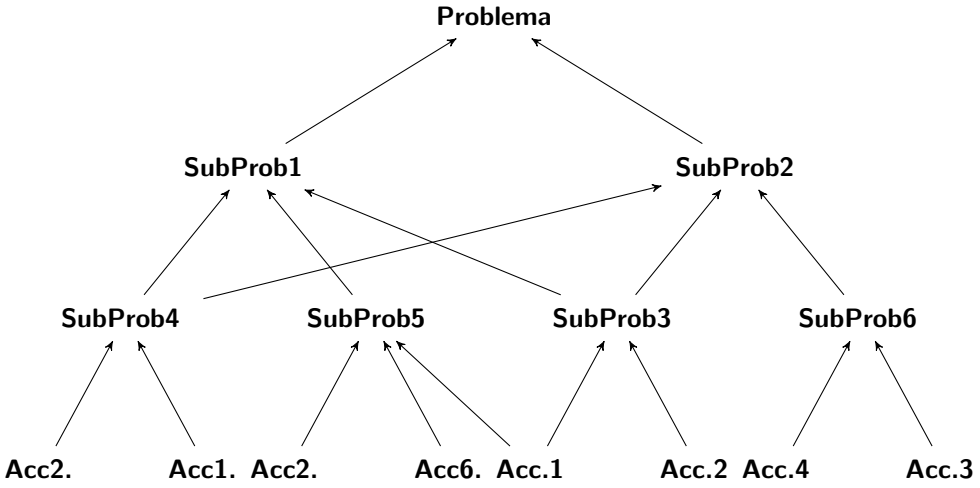

Figura 2: programación Bottom-Up

Este técnica de programación sustenta sus principios en :

1. La composición.
2. En el concepto de "hágalo usted mismo!" (por ejemplo un mueble comprado en un supermercado).

Desventajas:

1. Es necesario tener una buena cantidad de partes ya construidas.

2. ¿Qué es la Modularización?

En programación un módulo corresponde a alguna de las partes en la que un determinado problema fue dividido, es decir que un módulo resuelve alguno de los subproblemas que conforman al problema original. Un módulo puede ser pensado como una parte de un programa. En la práctica muchos programadores lo utilizan como sinónimo de subprograma (concepto que veremos a continuación). Se debe tener en cuenta que un módulo no es estrictamente un único subprograma sino que también puede ser un conjunto de varios de ellos. **La modularización en definitiva es construir un programa basándose en módulos independientes**.

3. Modularización en C

La pieza fundamental para la modularización en C se denomina **función**. Una función permite al programador modularizar un programa. Según sus creadores [**?**]:

"Las funciones dividen tareas grandes de computación en varias más pequeñas y, permiten la posibilidad de construir sobre lo que otros ya han hecho, en lugar de comenzar desde cero.... C ha sido diseñado para hacer que las funciones sean eficientes y fáciles de usar. Generalmente los programas en C consisten en muchas funciones pequeñas en lugar de unas pocas grandes."

3.1. Funciones

En primer lugar tenemos que develar que ya se ha utilizado el concepto de función sin saberlo. Cuando se construye un programa en C y se escribe:

```c
int main () {

}
```

En realidad se está usando la función main de C. El punto de entrada al programa, es decir, el punto a partir del cual se empieza a ejecutar el mismo. En C existen dos tipos de funciones: las funciones de la biblioteca estándar y las funciones definida por el programador.

3.1.1. Funciones Definidas por el Programador

Esencialmente la estructura de una función en C es la siguiente:

```c
tipo_retorno nombre_función (lista de parámetros){
        declaraciones

        acciones

}
```

```c
tipo_retorno nombre_función (tipo_1 Par_1, tipo_2 Par_2, ... , tipo_N Par_N){

   /* declaraciones */

   /* acciones */
   acción_1;
   acción_2;
   acción_3;
   acción_4;
   acción_5;
   . . . .
   acción_N;

}
```

Tipo de retorno: El tipo de retorno corresponde al tipo de dato del valor que devolverá la función tras ser ejecutada. La acción que permite la ejecución de una función se denomina *invocación*. Existen dos roles entre una función y aquel que la utiliza. El rol de la parte del programa que hace uso de una función se llama "invocador o llamador". El rol de la función al ser utilizada por alguna parte del programa se denomina "invocada o llamada". Si a una función no se le asigna tipo de retorno esta devolverá por defecto un valor de tipo entero. La instrucción que se encarga de devolver el valor de la función y devolver el control al invocador es **return**. La sintaxis de return corresponde a:

```c
return expresion;
```

Nombre de la función: Toda función debe tener un nombre que cumpla con las reglas sintácticas de los identificadores válidos en C.

Lista de parámetros: Un parámetro es una variable utilizada para recibir valores de entrada en una función. La lista de parámetros consiste en una lista separada por comas que contiene *las declaraciones* de los parámetros recibidos por la función al ser invocada.

Declaraciones y Acciones: Dentro del cuerpo de una función se espera encontrar declaraciones de variables, necesarias para alcanzar el objetivo de la función. Estas variables declaradas dentro de la función se denominan variables **locales**, una variable es local dentro del bloque de programa, en este caso una función, en la cual es declarada. También se espera encontrar acciones y estructuras de control.

3.1.2. Ejemplo 1:

```c
# include <stdio.h>

int cuadrado (int numero){
        return numero*numero;
}

int main(){
        int i;
        printf("\n");
        for ( i=1 ; i<=10 ; i++){
                printf ( "%d  ", cuadrado(i));
        }
        printf("\n\n");

        return 0;
}
```

resultado: 1 4 9 16 25 36 49 64 81 100

3.1.3. Prototipo de una Función

Se denomina prototipo de una función o firma, al tipo de dato de retorno + el nombre de la función + la lista de tipos de los parámetros de la misma. En el lenguaje de programación C se requiere que una función esté declarada antes de ser utilizada por su llamador. Pero es posible poder dilatar la implementación de la misma dentro del programa con solo indicar su prototipo. Por ejemplo:

```c

# include <stdio.h>

int cuadrado(int);//prototipo de la       función 'cuadrado'

int main(){
        int i;
        printf("\n");
        for ( i =1; i <=10 ; i ++){
                printf ( "%d  ", cuadrado(i));
                }
        printf("\n\n");

        return 0;
}

int cuadrado (int numero){
        return numero*numero;
        }
```

3.2. Funciones sin Valor de Retorno

En muchos lenguajes de programación existe el concepto de **procedimiento**, éste es *comparable a una función que no retorna valor alguno*. En C, no existen los procedimientos como una estructura separada de las funciones. Para definir lo que en otros lenguajes de programación es conocido como un procedimiento en C se utiliza una función con valor de **retorno nulo**. Para ello existe el tipo de dato **void**. Este tipo de dato utilizado como valor de retorno de una función indica que la misma no devuelve valor alguno. También puede ser usado en la lista de parámetros formales, para indicar que la función no recibe parámetros:

```c
void un_procedimiento(parametro_1,parametro_2,...,parametro_N);
```

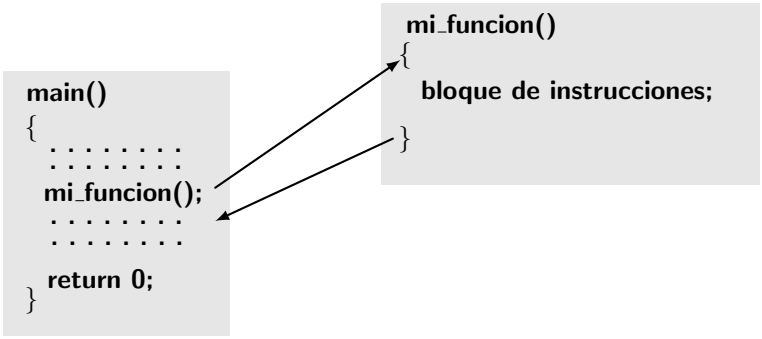

Figura 3: Esquema básico del funcionamiento del llamado a una función sin valor de retorno

3.3. Nombres de Funciones

A este punto hay que prestarle bastante atención. Un nombre de una función debe definir o ser equivalente a una acción. Por ejemplo:

```
long obtener_cuadrado (int un_numero){
    ....
    ....
}

long calcular_factorial (int un_numero){
....
....
}
```

Si una función tiene un nombre como por ejemplo:

```
int obtener_cuadrado_mostrandolo_por_pantalla (int un_numero){
....
....
}
```

Es un indicativo que estamos frente a un problema de diseño y de mala programación. A su vez si el nombre de la función indica que ésta realiza una única acción y en su interior hace más cosas también nos encontramos frente al mismo problema:

```
long obtener_cuadrado (int un_numero){
    long cuadrado=(un_numero*un_numero);

    printf("%d",cuadrado);
    return cuadrado;
}
```

```
long obtener_cuadrado (int un_numero){
        return (un_numero * un_numero);
}
```

3.4. Más sobre Funciones

La idea principal, una vez conocido el concepto de función en C, es utilizar este concepto para aplicar la idea de **Divide y Vencerás**. Es decir el concepto de función permite dividir el programa en un conjunto de funciones que sirvan para resolver el problema.

Para ello las funciones deben seguir algunas de las siguientes reglas:

- Deben tener pocas lineas.
- Su nombre debe denotar el **hacer algo** o **responder algo**, no ambas cosas.
- La lista de parámetros debe estar en (7+-2).
- Los nombres de los parámetros deben seguir la misma reglas que los nombres de las variables.
- Es buena idea que tiendan a ser genéricas, es decir, poder reutilizarla en casos donde las condiciones son similares.

Además, se debe tener en cuenta que solo se podrán usar variables declaradas en el scope o ámbito de la función que se esta ejecutando. Ya sea porque se obtienen de los parámetros, sean globales o declaradas en la mismo.

4. Recursividad

La recursividad es una característica de ciertos problemas los cuales son solucionables mediante la solución de una instancia del mismo problema. En informática la recursividad está fuertemente asociada a las funciones y a los tipos de datos. En este apunte se estudia su relación con la Implementación de las funciones recursivas. Se dice que *una función es recursiva si en el cuerpo de la función hay una llamada a sí misma*. Para poder ver cual sería este caso, es conveniente mirar algunos problemas matemáticos cuya naturaleza es recursiva, por ejemplo, los números factoriales. Un número factorial se define de la siguiente forma:

$$n\,! = \begin{cases} 1 & si\ n = 0 \\ (n-1)\ n! \times n & si\ n > 0 \end{cases}$$

En este caso puede verse como en la definición de factorial se invoca a factorial, este es uno de los mejores ejemplos del concepto de recursivo. Por el momento la versión conocida, para determinar el factorial de un número, debe estar construida en forma iterativa:

```
long factorial (int un_numero){
        long producto;
        int  contador;

        producto=1;
        contador=un_numero;
        while ( contador > 0 ){
                producto *= contador;
                contador--:
        }
        return producto;
}
```

¿Será posible escribir una versión recursiva del mismo algoritmo?

En el lenguaje de programación C es posible construir funciones recursivas de la misma forma que en matemática. Para ello lo único que hay que hacer es llamar otra vez a la función dentro de su ámbito. Para lograr realizar esto se utiliza el **stack o pila de ejecución**, este no es más que la parte del programa donde se mantiene la información sobre cual función es la que está siendo ejecutada, sus variables y parámetros. Todos los programas tienen su propio stack, que es creada cuando comienza a ejecutarse, ver Figura **??**.

Si se observa detalladamente una función recursiva debe cumplir con ciertas reglas para que funcione correctamente, etas son:

- debe poseer una condición de corte
- debe poseer una llamada recursiva, es decir así misma dentro de la función.

```
long factorial  (int un_numero){
        if (un_numero > 0 )
                factorial= un_numero *  factorial(un_numero-1);
        else
                return 1;
}
```

llamada recursiva

condición de corte

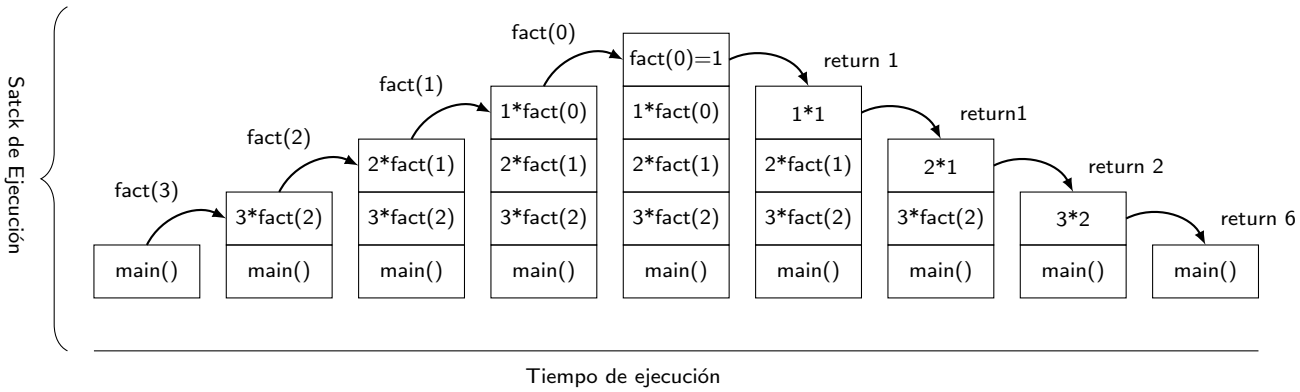

Figura 4: stack de ejecución

Ejemplo: Se le solicita al usuario que ingrese un número por teclado, luego se calcula e imprime por pantalla el resultado de la Serie de Fibonacci para dicho número.

```
long int fibonacci(numero){
        if (numero < 2)   // Condicion de corte
                return numero;

        return fibonacci(numero-1)+fibonacci(numero-2); //Llamado recursivo para numero-1 y luego para numero-2
}

int main (){

        int numero;
        scanf("%i",&numero);

        long int resultado = fibonacci(numero);

        printf("El Fibonacci de %i es %lu",numero,resultado);
        return 0;

}
```

4.1. Recursividad Indirecta

Se dice que una funcion usa recursividad directa, si la misma se llama a sí misma. Por otro lado existe otra forma de recursividad, la **recursividad indirecta**, que se dá cuando una función f llama a una función g, la función g llama a la función g y así sucesivamente... En la Figura **??** se puede ver el esquema de funcionamiento de la recursividad indirecta:

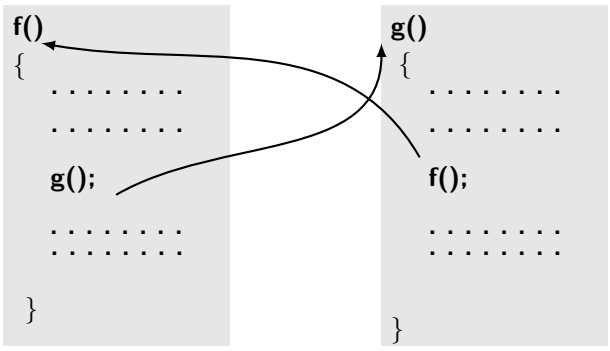

Figura 5: Simulacion de recursividad mutua o indirecta

Al igual que en la recursividad directa ambas funciones deben tener un punto de corte.

Ejercicio 1

La Granja de los Conejos Inmortales Tomás el granjero le compró al brujo del pueblo un par de conejos inmortales recién nacidos con la intención que se reproduzcan y tener su propio ejercito de conejos inmortales super adorables

y poder así conquistar el mundo. Tomás le preguntó al brujo que es lo que tenía que tener en cuenta a la hora de tener su propia granja de conejos inmortales y el brujo le comentó lo siguiente:

1. Un par de conejos bebe tardan un mes en crecer.
2. Un par de conejos adulto tardan un mes en producir otro par de conejos bebe.
3. A partir de que un par de conejos adulto tienen un par de bebés, producen uno nuevo cada mes indefinidamente.

1. Tomás inmediatamente se da cuenta que va a ser difícil llevar la cuenta de cuántos conejos tendrá en el futuro así que nos pidió que lo ayudemos y hagamos una serie de algoritmos en C que determinen cuántos conejos bebe, adultos y totales tendrá el n-ésimo mes.

Solución del ejercicio ??

Análisis: en primer lugar se debería poder escribir un ejemplo a mano del problema planteado.

Mes	Bebes	Adultos
1	1	0
2	0	1
3	1	1
4	1	2
5	2	3
6	3	5
7	5	8

Figura 6: Simulacion de recursividad mutua o indirecta

Para saber la cantidad de conejos adultos basta con mirar la tabla, y se puede observar que:

$$Adultos(n) = \begin{cases} 0 & si\ n = 0 \\ \\ Bebes(n-1) + Adultos(n-1) & si\ n > 0 \end{cases}$$

para saber la cantidad conejos bebes en un determinado tiempo n, basta ver en la tabla que :

$$Bebes(n) = \begin{cases} 1 & si\ n = 0 \\ \\ Adultos(n-1) & si\ n > 0 \end{cases}$$

Con la definición de estas funciones ya se puede implementar el problema en forma computacional. **Implementación:**

```c
#include <stdio.h>

//Linea de compilacion: gcc resuelto.c -o resuelto -Wall -Werror -std=c99 -g

/*
 * Todas estas funciones devuelven la cantidad de parejas
 * de conejos totales, bebes y adultas respectivamente.
 * Pre: El numero del mes es mayor o igual a 0
 */
unsigned int cantidad_total(unsigned int mes);
unsigned int cantidad_bebes(unsigned int mes);
unsigned int cantidad_adultos(unsigned int mes);

int main(){
  unsigned int mes;
  printf("El codigo funciona con cualquier nÃºmero, pero es recomendable no
  poner numeros mayores a 40 porque puede llegar a tardar mucho\n", );
  printf("Mes nÃºmero:");
  scanf("%d", &mes);
  printf("La cantidad total de parejas es:%d\n", cantidad_total(mes));
  printf("La cantidad de parejas bebes es:%d\n", cantidad_bebes(mes));
  printf("La cantidad de parejas adultas es:%d\n", cantidad_adultos(mes));
  return 0;
}

unsigned int cantidad_total(unsigned int mes){
  return cantidad_bebes(mes)+cantidad_adultos(mes);
}
```

```
29
30 unsigned int cantidad_bebes(unsigned int mes){
31   /*
32    * La cantidad de bebes en un mes es igual a la cantidad
33    * de adultos del mes anterior. Si el mes es el primero,
34    * la cantidad de bebes es 1.
35    */
36
37   if(mes == 1)
38     return 1;
39
40   return cantidad_adultos(mes-1);
41 }
42
43 unsigned int cantidad_adultos(unsigned int mes){
44   /*
45    * La cantidad de adultos en un mes es igual a la cantidad
46    * de adultos y bebes del mes anterior. Si el mes es el primero,
47    * la cantidad de adultos es 0.
48    */
49
50   if(mes == 1)
51     return 0;
52
53   return cantidad_adultos(mes-1)+cantidad_bebes(mes-1);
54 }
```

5. Archivos de cabecera (Header file)

Se denomina archivo de cabecera o header file, o include file, o en español archivo de inclusión, en el ámbito de los lenguajes de programación C y C++, al archivo, normalmente en forma de código fuente, que el compilador incluye de forma automática al procesar algún otro archivo fuente. Es típico y muy común que los programadores especifiquen la inclusión de los header files por medio de pragmas al comienzo (head o cabecera) de otro archivo fuente.

Un header file contiene, normalmente, una declaración directa de subrutinas, variables, u otros identificadores. Aquellos programadores que desean declarar identificadores estándares en más de un archivo fuente pueden colocar esos identificadores en un único header file, que se incluirá cuando el código que contiene sea requerido por otros archivos.

En la mayoría de los lenguajes de programación, los programadores tienden a construir los programas mediante la composición de elementos de menor tamaño, en el caso de C, en funciones. Estas pueden ser reutilizadas en más de un programa.

5.1. Un Ejemplo Práctico

Supongamos que se quiere utilizar en un determinado programa una función que sume dos números enteros. Para ello se debe escribir la siguiente porción de código fuente:

```
1
2 int sumar (int numero_uno, int numero_dos){
3     return numero_uno+numero_dos;
4 }
5
6 int triplicar (int numero){
7     return sumar(numero,sumar(numero,numero));
8 }
```

Si el programador quisiera volver a utilizar la misma función en otro programa C debería, por lo que se conoce hasta el momento, declarar nuevamente la función o el prototipo de la misma. Si esto se repite, es decir, el programador quiere o necesita usar esta función en más de un programa en C deberá escribirla múltiples veces, una por programa que desee realizar. Esto acarrea dos implicancias. La primera, es que hay que reescribir la función más de una vez, lo que puede acarrear problemas ya que es muy tedioso repetir la misma porción de código fuente muchas veces sin cometer errores. La segunda implicancia está en que a la hora de realizar una modificación, el cambio hay que plasmarlo en cada una de las copias de la función desparramada por el código fuente.

Para solucionar esto C nos proporciona los archivos de cabecera. En ellos se puede definir funciones, variables y constantes y utilizarlas en tantos programas se requiera con sólo hacer referencia al archivo cabecera en el que se ha definido. Archivo aritmetica.h:

```
1 /* File aritmetica.h */
```

```
#ifndef ARITMETICA_H
#define ARITMETICA_H

int sumar(int numero_uno, int numero_dos);// prototipo de la funcion sumar

#endif /* ARITMETICA_H */
```

En este archivo sólo se incluirán los prototipos de las funciones es decir, el tipo de retorno, el nombre y el tipo de los parámetros que recibe. Posteriormente se implementará en un archivo llamado aritmetica.c la función sumar:

```
/* File aritmetica.c */
#include "aritmetica.h"

int sumar(int numero_uno, int numero_dos){
        return numero_uno + numero_dos;
}
```

Así queda definida e implementada la función sumar en el archivo aritmetica.h (definición) y aritmetica.c (implementación). En este punto se puede utilizar la función sumar desde cualquier cualquier programa C con tan sólo utilizar el archivo aritmetica.h:

```
/* File Ejemplo.c */
#include <stdio.h>
#include "aritmetica.h"

int triplicar(int numero){
        return sumar(numero, sumar(numero,numero));
}

int main(){

        int valor = 1;
        int suma, triple;
        suma = sumar(valor, valor);
        triple = triplicar(valor);
        printf("\n La suma es : %d y el triple es %d \n\n", suma, triple);

        return 0;
}
```

5.2. Inclusión de archivos

Existe una herramienta llamada el preprocesador de C (cpp), cualquier compilador de C la primera acción que realiza es ejecutar al preprocesador. El preprocesador es un programa que realiza transformaciones literales de código fuente. Este está compuesto por una serie de directivas. Una de estas es la directiva #include. Cada vez que el preprocesador encuentra una directiva #include reemplazará esta línea por el contenido completo del archivo que se encuentra entre <> o entre "". Por ejemplo la linea #include<stdio.h> se reemplazará con el archivo de cabecera del sistema con ese nombre. En ese archivo de cabecera se declara, entre otras muchas cosas, la función printf().

Inicialmente se diferenciaba entre archivos de cabecera del sistema (<>) y los que eran creados por los programadores (""). En la actualidad los compiladores de C y los entornos de desarrollo actuales disponen de facilidades para indicar dónde se encuentran los distintos archivos de cabecera.

Nota: #include normalmente obliga a usar protectores de #include o la directiva #pragma once para prevenir la doble inclusión, porque si se incluye más de 1 vez el mismo archivo, (dependiendo del contenido) puede causar que se intente declarar varias veces las mismas funciones o tipos de variable, lo que va a generar un error al compilar, esto se intenta prevenir de la siguiente forma:

```
#ifndef __ARCHIVO_H__
#define __ARCHIVO_H__

/*... declaraciones de funciones, etc. ...*/

#endif
```

Como resultado, al intentar incluirse el archivo por segunda vez, la operación " ifndef " va a dar falso porque __ARCHIVO_H__ ya estaba definido la primera vez que se incluyó, y a consecuencia se saltea todo el bloque hasta llegar al "endif" que suele estar al final del archivo."

6. La Biblioteca Estándar de C

El estándar ANSI C define una serie de funciones que deben estar presentes en todo compilador de C. Usando únicamente estas funciones podemos asegurar la portabilidad de los programas.

Estas funciones se pueden agrupar en las siguientes categorías:

- Funciones de E/S.
- Funciones de cadenas y de caracteres.
- Funciones matemáticas.
- Funciones de asignación dinámica de memoria.
- Otras funciones.

La forma de acceso a las mismas es usando los archivos de cabecera donde están definidas las características y elementos necesarios de las funciones. Dado que estas cabeceras están escritas en el estándar de C, se ha utilizado en este apunte la traducción de Darío Álvarez Gutiérrez:

6.1. Descripción de los Archivos de Cabecera de la Biblioteca Estándar de C

Según el estándar de ANSI C estos son los archivos cabecera de la biblioteca estándar de C [**?**].

assert.h Define la macro assert y el símbolo NDEBUG. Se usa para diagnósticos del programa.

ctype.h Define rutinas de clasificación y conversión para caracteres.

errno.h Define macros para las condiciones de error, EDOM y ERANGE y la variable entera errno.

float.h Define símbolos para los valores máximos y mínimos de los números en coma flotante.

limits.h Define símbolos para los valores extremos de los tipos enteros.

locale.h Declara las funciones necesarias para adaptar los programas a un país determinado. Define la estructura lconv.

math.h Declara las funciones matemáticas y la constante HUGE_VAL.

setjmp.h Define el tipo de datos jmp_buf usado por las rutinas setjmp y longjmp.

signal.h Define símbolos y rutinas necesarios para la gestión de condiciones especiales.

stdarg.h Define las macros que facilitan la manipulación de listas de argumentos de longitud variable.

stddef.h Define los tipos estándar ptr_diff, size_t, wchar_t, el símbolo NULL, y la macro offsetof.

stdio.h Define tipos y macros necesarios para el paquete de Entrada/Salida Estándar. Define las secuencias predefinidas stdin, stdout, stderr y stdprn. Declara rutinas de EntradaSalida.

stdlib.h Declara las funciones de utilidad como las rutinas de conversión de cadenas, generador de números aleatorios, rutinas de asignación de memoria, y rutinas de control de procesos.

string.h Declara las rutinas de manipulación de strings.

time.h Define el tipo de datos time_h, la estructura de datos tm, y declara las funciones de tiempo.

Muchas de estas funciones están muy ligadas a la estructura del sistema UNIX (como las de manejo de señales). Sin embargo, al ser parte de la biblioteca estándar, se garantiza que funcionan en otros entornos.

Algunos de estos archivos simplemente son usados a su vez por otros archivos de cabecera. Importantes son ctype.h, errno.h, math.h, setjmp.h, signal.h, stdio.h, stdlib.h, string.h y time.h.

A continuación describiremos alguno de los archivos cabecera más importantes.

6.2. stdbool.h

La finalidad de esta biblioteca es introducir un nuevo tipo de variable (bool) que represente los valores logicos 0 (falso logico) y 1 (verdadero logico), lo que vuelve menos tediosa la lectura del codigo.

```
/* Copyright (C) 1998, 1999, 2000, 2009 Free Software Foundation, Inc.

This file is part of GCC.

GCC is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 3, or (at your option)
any later version.

GCC is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

Under Section 7 of GPL version 3, you are granted additional
permissions described in the GCC Runtime Library Exception, version
3.1, as published by the Free Software Foundation.

You should have received a copy of the GNU General Public License and
a copy of the GCC Runtime Library Exception along with this program;
see the files COPYING3 and COPYING.RUNTIME respectively.  If not, see
<http://www.gnu.org/licenses/>.  */

/*
* ISO C Standard:  7.16  Boolean type and values  <stdbool.h>
*/

#ifndef _STDBOOL_H
#define _STDBOOL_H

#ifndef __cplusplus

#define bool        _Bool
#define true        1
#define false        0

#else /* __cplusplus */

/* Supporting <stdbool.h> in C++ is a GCC extension.  */
#define _Bool        bool
#define bool        bool
#define false        false
#define true        true

#endif /* __cplusplus */

/* Signal that all the definitions are present.  */
#define __bool_true_false_are_defined        1

#endif
```

6.3. stdio.h

Funciones del archivo de cabecera stdio.h entrada y salidad de datos:

Tarea	Nombre de la función
Crear o abrir un fichero	fopen, freopen
Cerrar un fichero	fclose
Borrar o renombrar un fichero	remove, rename
Lectura con formato	fscanf, scanf
Escritura con formato	fprintf, printf, fvprintf, vprintf
Lectura de un carácter	fgetc, fgetchar, fputc, fputchar
Lectura de una línea	fgets, gets
Establecer posición L/E	fseek, fsetpos, rewind
Obtener posición L/E	fgetpos, ftell
Lectura binaria	fread
Escritura binaria	fwrite
Volcado de buffer	fflush
Comprobación de error/EOF	clearerr, feof, ferror
Gestión de ficheros temp.	tmpfile, tmpnam
Control de buffers	setbuf, setvbuf
evolver un carácter al buffer	ungetx

int rename(const char *oldname, const char *newname);
Renombra el fichero a cuyo nombre apunta oldname, por el de newname.

int vfprintf(FILE *stream, const char *format, va_list arglist);
int vprintf (const char *format, va_list arglist);
Funcionan como printf, sólo que en lugar de escribir los argumentos directamente como parámetros se pasa una lista de argumentos (similar a argv)

char *fgets(char *s, int n, FILE *stream);
Lee una cadena del fichero, hasta encontrar fin de línea o leer n-1 caracteres.

int fgetpos(FILE *stream, fpos_t *pos);
Obtiene la posición del puntero de lectura y escritura del fichero, en un formato que puede usar luego fsetpos.

int fsetpos(FILE *stream, const fpos_t *pos);
Establece la posición de lectura y escritura del fichero, usando el formato dado por fgetpos.

long ftell(FILE *stream);
Devuelve la posición del puntero de lectura y escritura del fichero, expresada como un desplazamiento en bytes desde su comienzo. Puede usarse este valor para hacer luego una llamada a fseek.

int ferror(FILE *stream);
Indica si está activo el indicador de error del fichero, devolviendo un número positivo.

void clearerr(FILE *stream);
Resetea los indicadores de error y de fin de fichero del fichero.

6.4. Conversión de Datos: stdlib.h

Funciones de conversión:

Tarea	Nombre de la función
Convertir cadena a un flotante	atof, strod
Convertir cadena a entero	atoi
Convertir cadena a entero largo	atol, strtol
Convertir cadena a unsigned long	stroul

int atoi(const char *s);
long atol(const char *s);
double atof(const char *s);
Convierten la cadena s en un entero, entero largo y doble precisión, respectivamente. Indican error devolviendo cero.

6.5. Funciones Matemáticas: math.h

Funciones del archivo de cabecera math.h

Tarea	Nombre de la función
Funciones trigonométricas	acos, asin, atan, atan2, cos, sin, tan
Potencias y logaritmos	exp, frexp, ldexp, log, log10, pow
Raíz cuadrada	sqrt
Valor absoluto	abs, fabs
Redondear números flotantes	ceil, floor
Funciones hiperbólicas	cosh, sinh, tanh
Descomponer flotante en	modf
entero y fracción	
Resto de división flotante	fmod
Aritmetica de enteros	abs, div, labs, ldiv
Generar números aleatorios	rand, srand

double cos(double x);
double sin(double x);
double tan(double x);

double exp(double x);
Devuelve e^x.

double frexp(double x, int *exponent);
Devuelve la mantisa y en exponent el exponente de x.

double pow(double x, double y);
Devuelve x^y.

double sqrt(double x);
Raíz cuadrada de x.

double ceil(double x);
Redondea un flotante hacia arriba
double floor(double x);
Redondea un flotante hacia abajo.

void srand(unsigned seed);
Inicializa el generador de números aleatorios.

int rand(void);
Devuelve un número aleatorio entre 0 y RAND_MAX.

6.6. Conversión Y Clasificación De Caracteres

Funciones del archivo de cabecera ctype.h

Tarea	Nombre de la función
Clasificar un caracter	isal num, isalpha, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit
Convertir de mayúscula a minúscu la	tolower
Convertir de minúscula a mayúscu la	toupper

int isalnum(int c); int islower(int c);
int isalpha(int c); int isprint(int c);
int isascii(int c); int ispunct(int c);
int iscntrl(int c); int isspace(int c);
int isdigit(int c); int isupper(int c);
int isgraph(int c); int isxdigit(int c);
Devuelven verdadero si el caracter c es alfanumérico, alfabético, de control, dígito, se puede imprimir (sin el espacio), está en minúscula, se puede imprimir (incluye el espacio), es de puntuación (imprimibles menos alfabéticos y espacio), es un separador (espacio, tabulador, etc.), está en mayúscula o es un dígito hexadecimal.
int tolower(int c);
Devuelve el carácter convertido a minúscula.
int toupper(int c);
Devuelve el carácter convertido a mayúscula

6.7. Manipulación De Cadenas Y De Buffers: string.h

Tarea	Nombre de la función
Longitud de una cadena	mblen, strlen
Comparación de dos cadenas	memcmp, strcmp, strncmp, strcoll, strxfrm
Copia y concatenación	memcpy, memmove, strcat, strcpy, strncat, strncpy
Búsqueda de carácter o subcadena	memchr, strchr, strcspn, strpbrk, strrchr, strspn, strstr
Extraer tokens de un string	strtok
Rellenar buffer con carácter	memset
Obtener la cadena de un error	strerror

size_t strlen(const char *s);
Devuelve la longitud de una cadena.

int strcmp(const char *s1, const char*s2);
Compara las cadenas s1 y s2, devolviendo 0 si s1 == s2, negativo si s1 ¡s2 o positivo si s1 ¿s2.

int strncmp (const char *s1, const char *s2, size_t maxlen);
Como strcmp, mirando como máximo maxlen caracteres.

char *strcat(char *dest, const char *src);
Añade la cadena src al final de la cadena dest.

char *strcpy(char *dest, const char *src);
Copia la cadena src sobre la cadena dest.Esta función es muy usada para almacenar cadenas no conocidas en tiempo de compilación en arrays de caracteres.

char *strncat(char *dest, const char *src, size_t maxlen);
char *strncpy(char *dest, const char *src, size_t maxlen);
Como strcat y strcpy sólo que como máximo usan maxlen caracteres.

Todas estas funciones devuelven el puntero a la cadena dest. No comprueban que esté reservado el espacio necesario y sobreescriben memoria si no lo está.

char *strchr(const char *s, int c);
Busca en la cadena la primera ocurrencia del caracter c.

char *strrchr(const char *s, int c);
Busca la última ocurrencia.

char *strstr(const char *s1, const char *s2);
Busca en la cadena s1 la primera ocurrencia de la subcadena s2.

char *strpbrk(const char *s1, const char *s2);
Busca en la cadena s1 la primera ocurrencia de cualquier carácter que esté en la cadena s2.
Todas estas funciones devuelven el valor null si no tienen éxito. En otro caso, devuelven un puntero a la ocurrencia encontrada.

char *strtok(char *s1, const char *s2);
s1 es una cadena compuesta por tokens de texto. Los separadores de los tokens son los caracteres de la cadena s2. La primera llamada a s2 devuelve el primer token encontrado. Es decir devuelve un puntero al primer carácter del token en s1 y escribe un nulo a continuación del token (en s1). Puede tratarse entonces la cadena como el token directamente. Las siguientes llamadas a strtok, usando NULL como primer parámetro devuelven los siguientes tokens. La cadena de separadores s2 puede ser distinta entre llamada y llamada. Cuando ya no hay más tokens devuelve NULL.

char *strerror(int errnum);
Devuelve una cadena que describe el mensaje de error del sistema de número errnum.

6.8. Control De Procesos: stdlib.h

Tarea	Nombre de la función
Ejecutar un comando del shell	system
Terminar un proceso	abort, exit
Gestión de errores	assert, perror
Obtener entorno del proceso	getenv
Instalar gestor de señales y generarlas	raise, signal
Salto no local de una función a otra	longjmp, setjmp
Instalar rutinas de terminación	atexit

int system(const char *command);
Llama al shell para que ejecute el comando contenido en la cadena. Si no tiene éxito devuelve -1, si lo tiene devuelve 0.

void abort(void); Aborta el proceso, imprimiendo un mensaje de error y devolviendo al sistema un código de error de 3.

void exit(int status); Termina el proceso, devolviendo al sistema el código de error status.

void assert(int test); /* assert.h */ Comprueba la condición test. Si es falsa, aborta el programa e imprime en el error estándar un mensaje de error, indicando la condición, el fichero y la línea donde se produjo.

void perror(const char *s); /* stdio.h */ Imprime el mensaje de error asociado al último error producido en una función del sistema (indicado por el valor de la variable errno). El array char *sys_errlist[] contiene los mensajes de error, e int sys_nerr indica el número de mensajes existentes.

int atexit(atexit_t func);
Registra la función func como función de salida. Cuando termina el programa se llama a la función (*func)(). Cada llamada a atexit registra una nueva función de salida. Se pueden registrar hasta 32 funciones, que se van llamando en orden inverso.

char *getenv(const char *name);
Devuelve el valor asociado a la variable de entorno especificada en name (TERM = vt100, etc.). Si la variable no está devuelve NULL.

int raise(int sig); /* signal.h */
Envía la señal sig al proceso en ejecución. Si se ha instalado un manejador para esa señal, se ejecuta el manejador, si no, se ejecuta la acción por defecto para la señal.

void (*signal(int sig, void (*func) (int sig)))(int);
Instala una función para manejar la señal sig. La función manejadora debe aceptar un parámetro de tipo entero. signal devuelve un puntero a la anterior función manejadora de la señal. Si no puede devuelve NULL. Están definidas un número dado de señales y dos manejadores estándar (ignorar y por defecto).

int setjmp(jmp_buf jmpb);
Marca la situación actual del proceso en jmpb, para que pueda ser usado en un longjmp posterior y devuelve 0.

void longjmp(jmp_buf jmpb, int retval); Restaura el estado en que estaba el proceso cuando se llamó a setjmp, haciendo que para el proceso parezca que la llamada a setjmp retorna con un valor de retval.

7. Ejemplos

7.1. Calcular el factorial de un número (Recursivo)

```c
#include <stdio.h>

int calcular_factorial(int numero){

    if(numero == 1){           //condición de corte
        return 1;
    }

    return ( numero * calcular_factorial(numero-1));  //llamada recursiva
}


int main(){

    int valor = 4;
    int factorial;

    factorial = calcular_factorial(valor);
    printf("El factorial de 4 es: %i\n", factorial);

    return 0;
}

/*En este ejemplo calcularemos el factorial de 4. Veamos qué sucede:
Con la primer llamada a la función "calcular_factorial" se crea un ámbito (ámbito 1)
en el cual la variable numero vale 4.
Con la segunda llamada se crea un segundo ámbito (ámbito 2), donde numero vale 3.
En la tercer llamada, ámbito 3, numero vale 2.
Cuarta llamada, ámbito 4, numero vale 1. Por la condición de corte, el ámbito 4
retorna 1.

La ejecución vuelve al ámbito 3 donde el valor de la variable numero se multiplica
por el retorno del ámbito 4:
2*1 -> el ámbito 3 retorna 2.

De nuevo en el ámbito 2, se multiplica el valor de numero de este (3) por el retorno
del ámbito 3:
3*2 -> el ámbito 2 retorna 6.

Por último, en el ámbito 1, se multiplica el valor de numero (4) por el retorno del
ámbito 2:
4*6 -> ámbito 1 retorna 24.

Ahora el valor de factorial es 24. Finalmente, se imprime por pantalla:
"El factorial de 4 es: 24"*/
```

8. Ejercicios Resueltos

8.1. Qui-Gon en la primaria

Cuando niño, el maestro Qui-Gon, era un alumno bastante duro, y le costaban mucho las operaciones matemáticas elementales. Para ayudarlo a entender, vamos a armar algunas operaciones para que el pequeño Qui-Gon pueda usar de ejemplo. En este caso, necesitamos que implemente una multiplicación de números naturales.

8.1.1. Posible Solución

```c
unsigned int multiplicar(unsigned int factor1, unsigned int factor2) {
    long multiplicacion;

    multiplicacion=(factor1 * factor2)
    return multiplicacion;
}
```

8.1.2. Posible Solución 2

```c
unsigned int multiplicar(unsigned int factor1, unsigned int factor2) {
return (factor1 * factor2);
}
```

8.2. Alimentando al monstruo

El proyecto Estrella de la muerte está estancado, y el emperador está furioso. Todos conocemos su fama de poco piadoso y antes de que enfoque su ira en nosotros debemos solucionar cómo obtener la descomunal energía que necesita la estrella de la muerte. Dado que nos encontramos en el espacio, podemos contar con energía solar en los sistemas que cuentan con soles, pero para los sistemas oscuros, la elección ideal es la energía nuclear. También sabemos los siguiente: Capacidad de generación de un panel Solar: 2w/h Capacidad de generación de una planta nuclear de Plutonio: 700 kw/h Capacidad de generación de una planta nuclear de Uranio: 400 kw/h **Requerimiento**: *Generar una función que calcule la la energía total generada por hora, tomando como base la cantidad de paneles solares y el rendimiento de los reactores nucleares (de plutonio y de uranio).* Nota: Exprese el resultado en w/h.

8.2.1. Posible Solución

```c
#define POTENCIA_SOLAR 2
#define POTENCIA_PLUTONIO 700000
#define POTENCIA_URANIO 400000

double energia_por_hora(unsigned int cantidad_paneles_solares, unsigned int
    cantidad_reactores_Plutonio, unsigned int cantidad_reactores_uranio) {

    double potencia_total;

    potencia_total=potencia_total+(POTENCIA_SOLAR*cantidad_paneles_solares);
    potencia_total=potencia_total+(POTENCIA_PLUTONIO*cantidad_reactores_plutonio);
    potencia_total=potencia_total+(POTENCIA_URANIO*cantidad_reactores_Uranio);

    return potencia_total;
}
```

8.2.2. Posible Solución 2

```c
#define POTENCIA_SOLAR 2
#define POTENCIA_PLUTONIO 700000
#define POTENCIA_URANIO 400000

double energia_por_hora(unsigned int cantidad_paneles_solares, unsigned int
    cantidad_reactores_Plutonio, unsigned int cantidad_reactores_uranio) {

    double potencia_total;

    potencia_total+=(POTENCIA_SOLAR*cantidad_paneles_solares);
    potencia_total+=(POTENCIA_PLUTONIO*cantidad_reactores_plutonio);
    potencia_total+=(POTENCIA_URANIO*cantidad_reactores_Uranio);

    return potencia_total;
}
```

¿En qué difiere con la anterior? ¿Cuál es mejor?

8.2.3. Posible Solución 3

```c
#define POTENCIA_SOLAR 2
#define POTENCIA_PLUTONIO 700000
#define POTENCIA_URANIO 400000

double energia_por_hora(unsigned int cantidad_paneles_solares, unsigned int
    cantidad_reactores_Plutonio, unsigned int cantidad_reactores_uranio) {

double potencia_total;

return (POTENCIA_SOLAR*cantidad_paneles_solares)+(POTENCIA_PLUTONIO*
    cantidad_reactores_plutonio)+(POTENCIA_URANIO*cantidad_reactores_Uranio);
}
```

¿En qué difiere con la anterior? ¿Cuál es mejor?

8.3. La ambición de Darth Vader

El propósito del maligno ser es dominar la galaxia y para eso necesita un gran ejército. Su plan de alistamiento consistía en la clonación de Stormtroopers, empezando con uno, y doblando su ejército día a día. El segundo día tendría 2, el 3ro 4, el 4to 8 y así sucesivamente. Escribir una función que permita saber cuántos soldados tendrá Darth Vader en su Ejército Imperial al cabo de N días (la cantidad de días se recibe como parámetro).

8.3.1. Posible Solución

```
int calcular_clones(int dias){
        int clones=1;

        for(int i=1; i<dias; i++)
            clones*=2;

        return clones;
}
```

8.3.2. Posible Solución Utilizando Recursividad

```

int calcular_clones(int dias){

        if(dias == 1)
            return 1;

        return calcular_clones(dias-1)*2;
}
```

8.4. Mantener el control

Darth Vader sabe que la mejor forma de mantener el control de la galaxia es mediante la constante vigilancia de Stormtroopers en cada planeta. La cantidad de Stormtroopers a posicionar en cada región de un planeta guarda relación con la cantidad de rebeldes dentro de una población de dicha región. - Por cada rebelde debe haber 2 Stormtroopers. - La cantidad de rebeldes en una población es igual al 10 % de la población. Requerimiento: Escribir una función que reciba la población y devuelva la cantidad de Stormtroopers necesarios para mantener el control.

8.4.1. Posible Solución

```
unsigned int calcular_cantidad_soldados(unsigned int poblacion){

        int cant_rebeldes = 0.1 * poblacion;
        unsigned int cant_troopers = cant_rebeldes * 2;

        return cant_troopers;
}
```

8.4.2. Posible Solución 2

```
unsigned int calcular_cantidad_soldados(unsigned int poblacion){

    return  ( 0.1 * poblacion) * 2;

}
```

¿En qué difiere con la anterior? ¿Cuál es mejor?