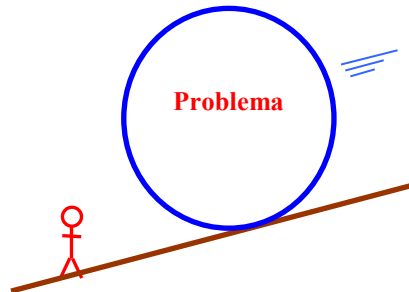
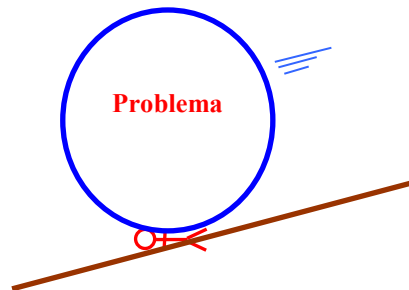


PROGRAMACIÓN MODULAR

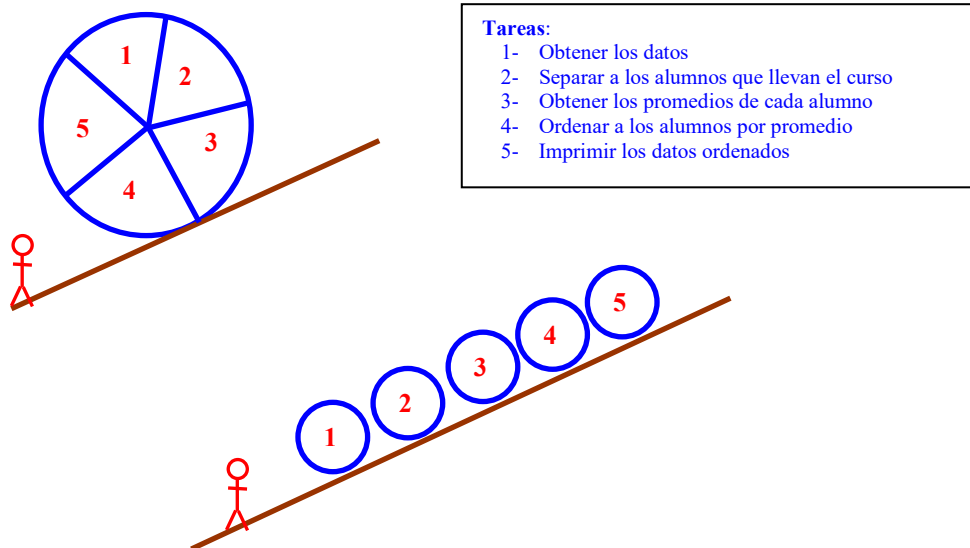
Si nos plantearan el siguiente problema: "Emitir un reporte en el que aparezcan los alumnos de un curso de acuerdo con el promedio obtenido, de mayor a menos nota", quizá la primera sensación que tengamos al leer el problema y saber que lo tenemos que resolverlo se parezca mucho a la figura siguiente:



Al parecer el problema es cómo esta roca, tan complejo o grande que quizá no lo podamos solucionar o detener, y nos pasará por encima aplastándonos.



Sin embargo, si luego de un análisis comprendemos que el problema es un conjunto de tareas, la situación puede cambiar, esto debido a que cada tarea puede ser considerada como un problema independiente y por lo tanto estamos ahora ya no ante un problema muy grande, sino ante varios problemas más sencillos.

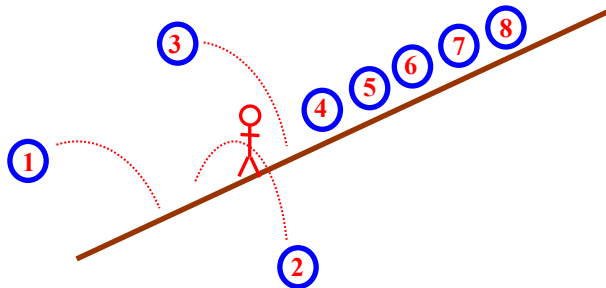


Una vez descritas las tareas que tiene el problema, se coge una a una las tareas y se les empieza a analizar. Es muy probable que esas tareas sean aún muy complejas, pues bien, a esas tareas les podemos aplicar el método anterior, esto es subdividir las en una serie de subtareas, por ejemplo:

Tarea: Obtener los promedios de cada alumno

- 1- Tomar un alumno
- 2- Calcular el promedio de prácticas (PPr).
- 3- Calcular el promedio de laboratorios (Ple).
- 4- Aplicar la fórmula $(2 \times PPr + plan + 3 \times Ex1 + 4 \times Ex2) / 10$ para obtener la nota del curso
- 5- Si hay más alumnos, repetir los pasos 1 al 5

Con esto Si la misma técnica se aplicase a cada tarea se podría descomponer el problema en partes mucho más pequeñas o simples que finalmente nos permita resolver el problema inicial.



Observe que las tareas que se han llegado a establecer, además de darnos una idea general de lo que tenemos que hacer para solucionar el problema, tienen algunas características.

En primer lugar, cada tarea es independiente de la otra, esto quiere decir que si tomamos una de las tareas podemos tratar de resolverla independientemente si se tiene o no una solución para las otras. Por ejemplo la tarea 2, "Separar los alumnos que llevan el curso", teniendo la lista de alumnos podemos separarlos independientemente de saber cómo se obtuvo la lista (por el teclado, por un archivo de textos, etc.), tampoco interesa aquí qué se va a hacer con la lista seleccionada (se va a ordenar por nombres, por promedio ponderado o por un curso, o si se va o no a imprimir y en qué medio). Esta característica hace que, como habrá podido darse cuenta, podamos empezar a solucionar el problema por cualquiera de sus tareas, sin necesidad que sea en un orden determinado, y más aún podemos solucionar una parte del problema sin tener idea de cómo se va a solucionar alguna otra tarea.

Otra característica de esto es que las tareas que se describen no están detalladas, no se menciona por ejemplo el nombre del archivo de donde se obtendrán los datos, qué método se va a emplear para ordenar los datos, ni la forma cómo se va a obtener la nota del curso. Esos detalles se irán colocando conforme ataquemos cada uno de los problemas.

Esta metodología de solución de problemas ha sido utilizada por muchos años con gran éxito y se denomina "**Diseño descendente**", se espera, en este texto, que usted pueda llegar a dominar esta metodología.

Otra característica de esta metodología es que una vez definida la lista de tareas de un problema se puede conformar varios equipos de personas, a cada equipo se le puede entregar una tarea que deberá resolver. Esto hará que la solución del problema se dé más rápidamente, porque estarán trabajando en paralelo. Cada grupo dará solución a una tarea, que entregará en lo que se denomina un "módulo" con todas las sub tareas resueltas, luego se juntarán los módulos y se tendrá la solución para el problema inicial, esto se conoce con el nombre de "**programación modular**".

Otra ventaja que se puede apreciar en esta forma de trabajar es que si luego de implementar la solución nos damos cuenta que una parte del programa no es muy eficiente, por ejemplo, se demora mucho en mostrar los resultados, podemos reemplazar el módulo por otro que resuelva la misma tarea, pero más eficiente, sin tener que modificar todo el programa.

Implementación de la programación modular

Los diferentes lenguajes de programación modernos permiten implementar programas haciendo uso de la técnica de programación modular. Las unidades básicas que nos brindan los lenguajes de programación para este fin se denominan **"funciones"**

Una función se puede definir como un conjunto de instrucciones que tiene como finalidad realizar un proceso y eventualmente obtener un valor resultante. Si nosotros vemos dentro la biblioteca de funciones **math** (observe que en C++ no se pone el **.h** como se hace en C), podemos encontrar por ejemplo que la instrucción: **a = pow(3,2);** está formada por una función (**pow**) que realizará el proceso de coger el valor de 3 y elevarlo a la potencia 2 y el resultado se lo entrega a la variable "a".

Declaración, implementación y uso de funciones

Una función se declara de la siguiente manera:

```
<Tipo de retorno>Nombre de la función (<Parámetros (Tipos)>);
```

Según esto:

```
int factorial (int);
```

Declara una función denominada "factorial" que recibe como parámetro un valor entero. La función devolverá o retornará como resultado un valor entero. Seguramente la función calculará el factorial de un número entero dado como parámetro.

```
double longitudDeSegmento (double, double, double, double);
```

Declara una función denominada "longitudDeSegmento" que recibe como parámetro cuatro valores de punto flotante y retorna un valor de tipo double. Probablemente la función recibirá dos coordenadas (x₁, y₁), (x₂, y₂) y retornará la longitud del segmento que lo forma.

La implementación de la función consiste en desarrollar el código necesario para poder realizar el proceso o el cálculo del resultado esperado por la función. Una función se implementa de la siguiente manera:

```
<Tipo de retorno>Nombre de la función (<Parámetros (Tipos)>){
    <Instrucción>
    <Instrucción>
    ...
    return expresión;
}
```

Ejemplos de implementación de funciones se aprecia a continuación:

```
int factorial (int num){
    int fact=1;
    for (int n = num; n > 0; n--){
        fact *= n;
    }
    return fact;
}

double longitudDeSegmento (double x1, double y1, double x2, double y2){
    return sqrt (pow(x2-x1, 2) + pow(y2-y1, 2));
}
```

Una vez declara e implementada la función, se puede usar como "pow" o "sin".

Bibliotecas de funciones

En el lenguaje C++ se puede trabajar modularmente de muchas formas, sin embargo, nos centraremos en una forma ligada al concepto de "Reutilización de código". La idea es plantear el trabajo con funciones pensando que cada vez que desarrollemos un proyecto no lo hagamos todo desde cero. Esto es que cuando diseñemos por primera vez un grupo de funciones las coloquemos, desde un principio, en un archivo especial, de modo que la próxima vez que necesitemos alguna de esas funciones solo tengamos que incorporar ese archivo al proyecto sin tener que volverlo a escribir, reutilizando el código. En otras palabras, el trabajo en este nuevo proyecto se limitará a repetir de modo similar lo que hacemos cuando necesitamos usar por ejemplo la función pow, esto es colocar la definición de esa función mediante la cláusula #include y luego simplemente utilizarla. Ese archivo especial del que hablamos se suele denominar "**Biblioteca de funciones**" y la idea es que en una de esas bibliotecas se coloque un grupo de funciones que estén relacionadas de alguna forma, por ejemplo, funciones como la media, mediana, moda, desviación estándar, etc. se pueden agrupar en una biblioteca de funciones estadísticas, funciones para graficar líneas, círculos, cuadriláteros, etc. pueden agruparse en una biblioteca de funciones gráficas.

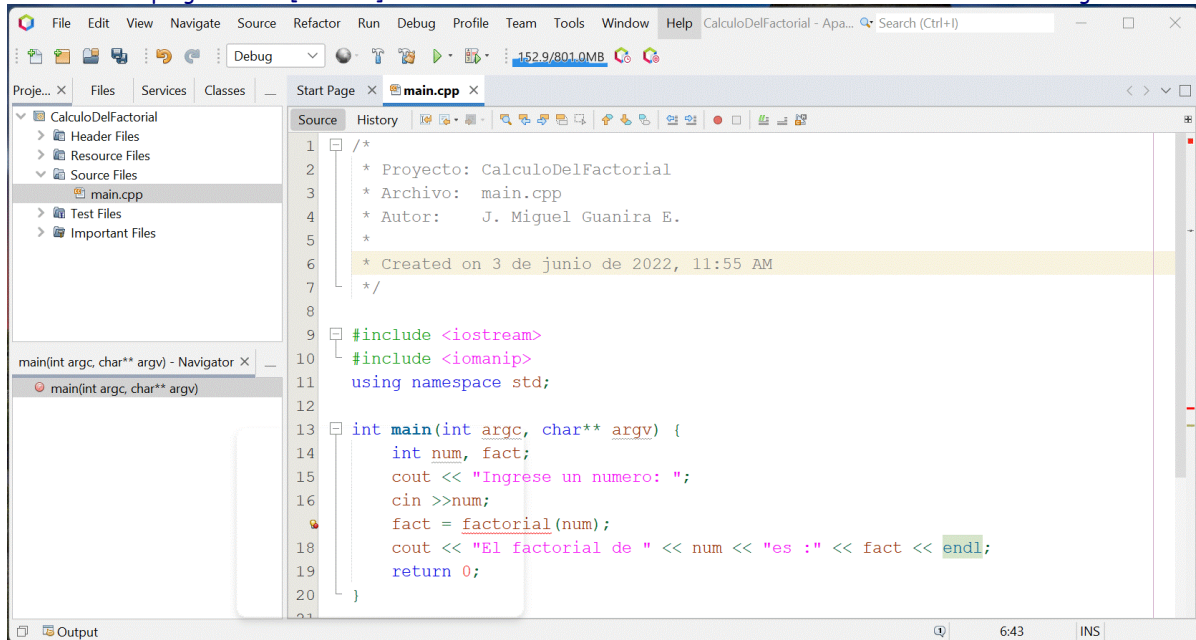
Un ejemplo explicará lo que queremos hacer:

Supongamos que queremos desarrollar un programa que nos permita calcular el factorial de un número, esto es, se ingresa un número entero y el programa calcula y nos devuelve su factorial.

Entonces las tareas que debe realizar el programa sería:

1. Leer el valor entero
2. Calcular el factorial
3. Mostrar el resultado

Plasmar este programa en NetBeans es una tarea sencilla, simplemente creemos un proyecto y escribamos el código siguiente:

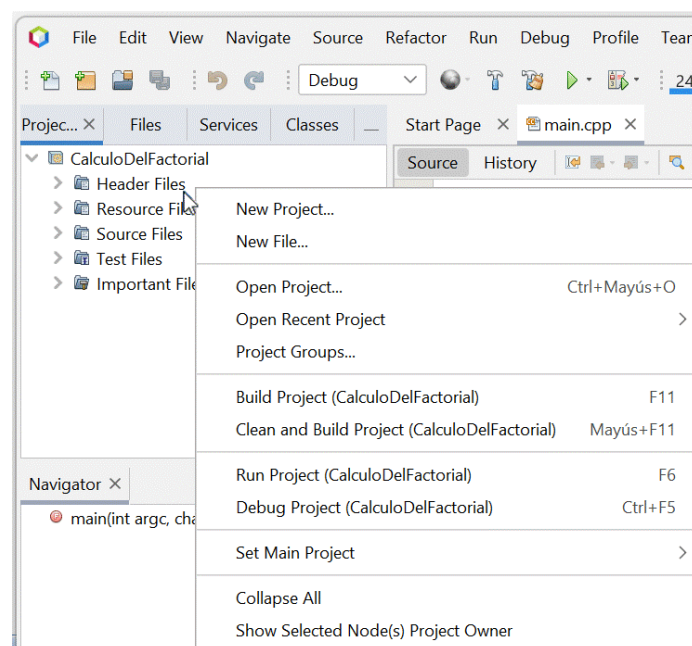


Observe que nuestro programa usará una función "factorial" de la misma forma que usamos las funciones pow o sin, con la diferencia que nuestra función factorial está subrayada en rojo, lo que significa que no se reconocerá el significado la palabra "factorial" por lo que si compilamos el programa nos dará una error.

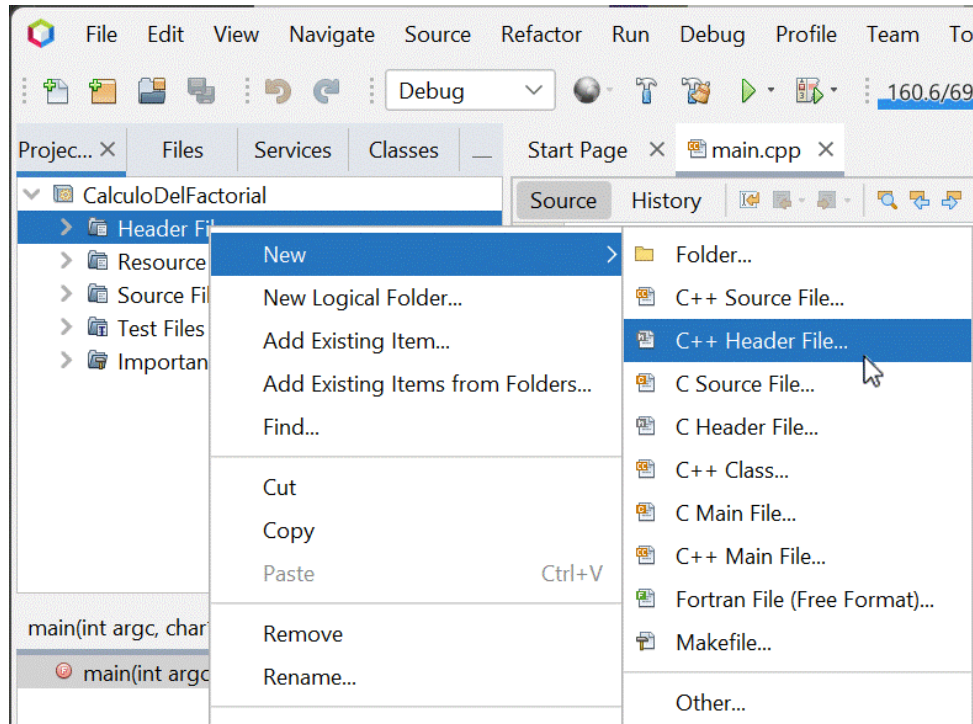
Para corregir el error debemos realizar dos tareas:

1. Declara la función factorial
2. Implementar la función factorial.

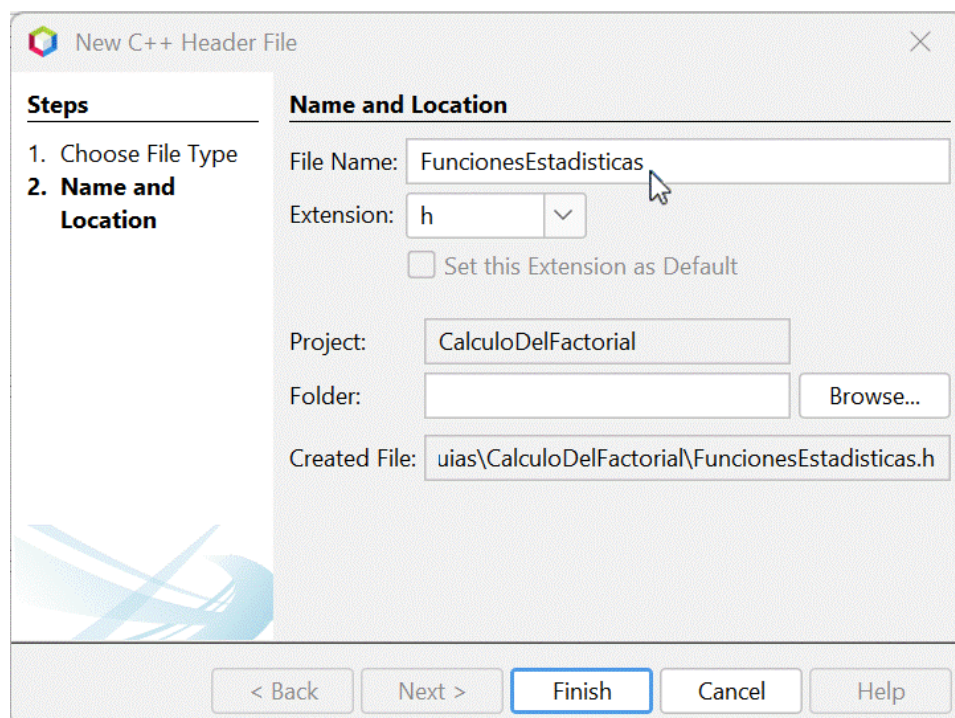
Estas dos tareas deben ser hechas pensando siempre en la reutilización del código. Entonces, debemos crear primero "un archivo de cabecera", "header file" o "archivo.h" donde coloquemos allí su declaración. Debe tener en cuenta que nuestro ejemplo solo define una función, pero, para que esto sea práctico, deberíamos pensar en varias funciones que se coloque en este archivo de cabecera. Para esto nos dirigimos a las carpetas del proyecto y en la carpeta "Header Files" presionaremos el botón derecho de mouse para que se despliegue un menú como se ve a continuación:



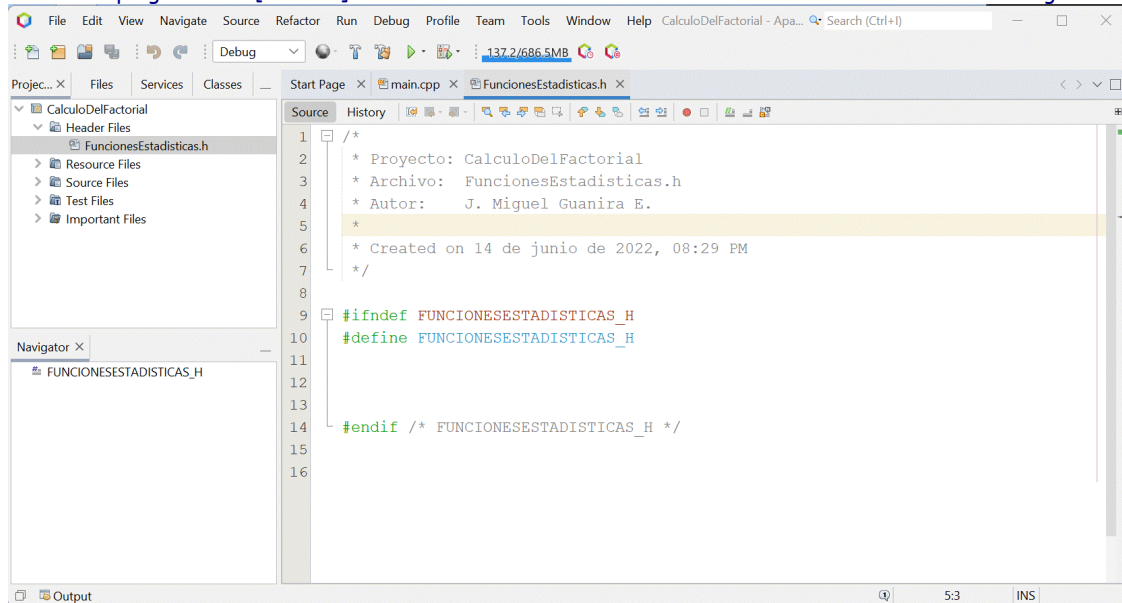
Allí nos dirigimos a la opción "Nuevo" y se desplegará otro menú, allí se elegirá la opción "C++ Header File...", como el que se aprecia a continuación:



Inmediatamente aparecerá una ventana, en ella, en el recuadro "File Name" colocaremos un nombre al archivo, este no debe ser el nombre de la función porque, como veremos luego no solo va a contener la definición de la función "factorial" sino todas las que queramos, por ejemplo, podríamos ponerle "FuncionesEstadisticas". Como vemos a continuación:

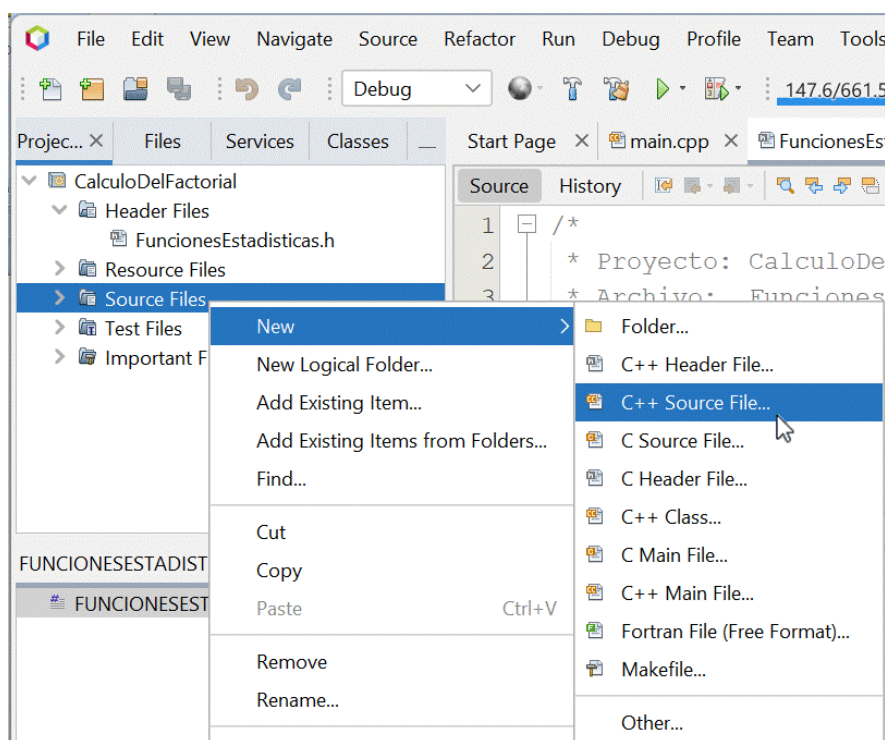


Luego presionamos el botón "Finish" con lo que podrá observar lo siguiente:

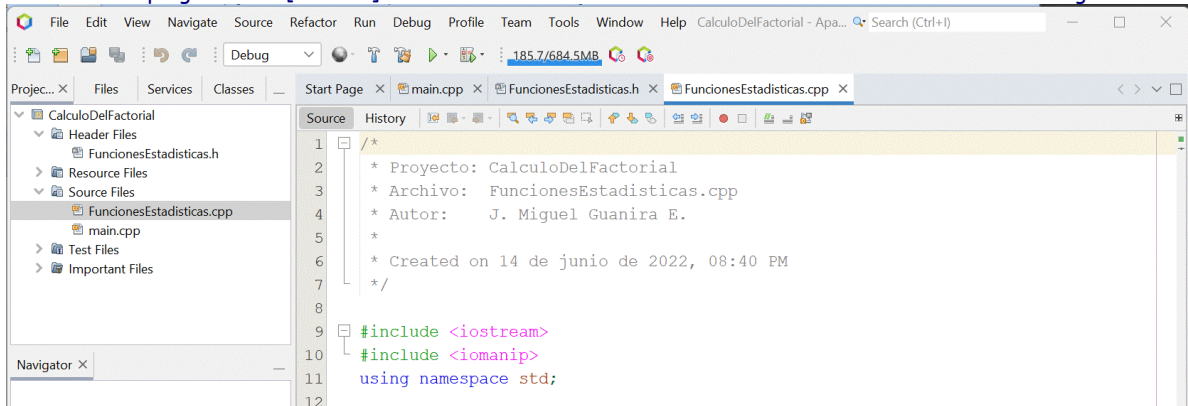


Como puede ver se ha creado el archivo "FuncionesEstadisticas.h".

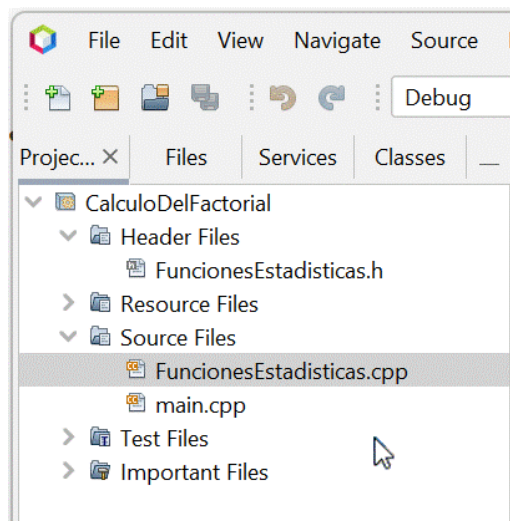
Luego crearemos el archivo donde colocaremos la implementación de nuestra función "factorial" (y todas las que queremos re utilizar). Para esto repetiremos la creación del archivo como lo hicimos con el "Header Files" pero ahora con el "Source Files", como se ve en la figura siguiente:



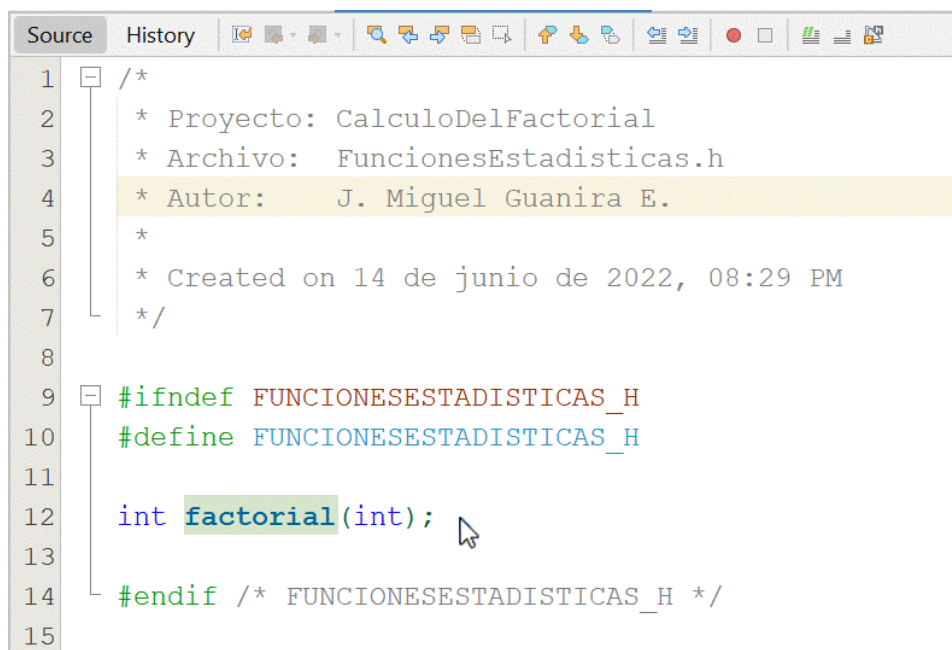
En la ventana que aparece coloque siempre como nombre (como una buena práctica de programación) el mismo que usó para el archivo de cabecera, en este caso FuncionesEstadisticas. Con esto ahora tendremos otro archivo con extensión ".cpp" como se ve a continuación:



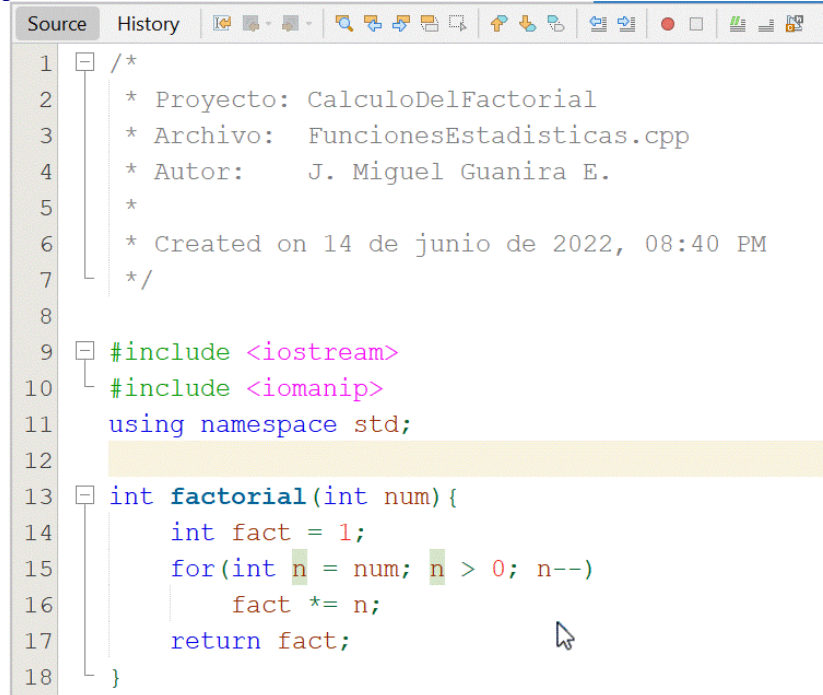
Observe que su proyecto ahora está formado por tres archivos. En la ventana de proyectos a la izquierda del NetBeans se puede ver esto.



Finalmente escribiremos el código correspondiente, en este caso en el archivo de cabeceras, colocaremos el encabezado de la función factorial, como se ve a continuación:



En el archivo fuente "FuncionesEstadisticas.cpp" escribiremos la implementación de la función, como se muestra a continuación:

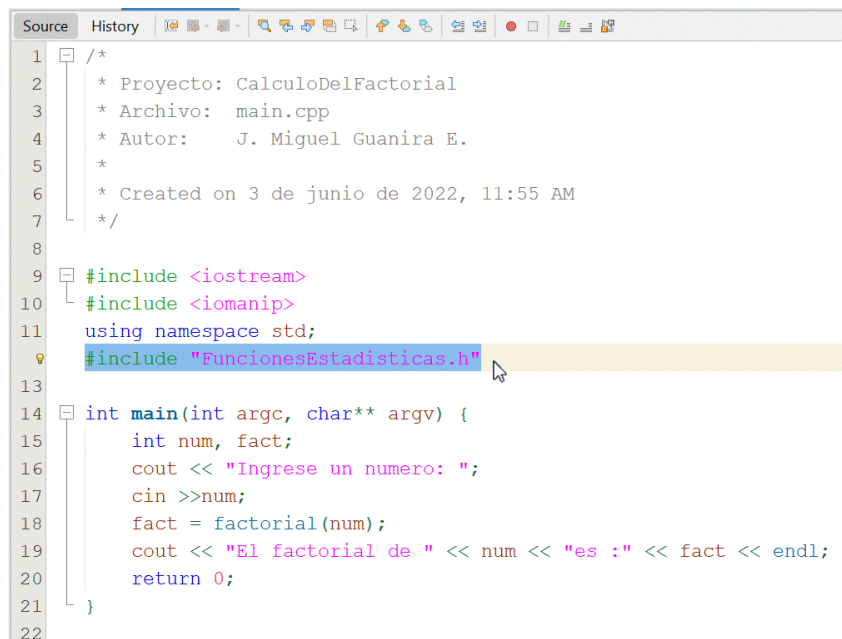


```

1  /*
2   * Proyecto: CalculoDelFactorial
3   * Archivo:  FuncionesEstadisticas.cpp
4   * Autor:    J. Miguel Guanira E.
5   *
6   * Created on 14 de junio de 2022, 08:40 PM
7   */
8
9  #include <iostream>
10 #include <iomanip>
11 using namespace std;
12
13 int factorial(int num){
14     int fact = 1;
15     for(int n = num; n > 0; n--)
16         fact *= n;
17     return fact;
18 }

```

Finalmente debemos saber que cuando compilemos el proyecto, cada módulo con extensión ".cpp" se compilará de manera independiente y luego se enlazará al programa ejecutable ".exe", por lo tanto, cualquier función que utilicemos en un módulo debe ser declarada antes de utilizarla. Por lo tanto, así como cuando usamos en main la función **pow** necesitamos incluir (**#include**) la biblioteca **math.h**, la definición de nuestra función factorial debe ser incluida en todos los módulos en donde se use. Regresando a nuestro ejemplo, por esa razón, para que se pueda compilar el módulo "main.cpp" del programa debemos colocar la definición de la función factorial colocando la orden **#include** correspondiente como se ve a continuación:



```

1  /*
2   * Proyecto: CalculoDelFactorial
3   * Archivo:  main.cpp
4   * Autor:    J. Miguel Guanira E.
5   *
6   * Created on 3 de junio de 2022, 11:55 AM
7   */
8
9  #include <iostream>
10 #include <iomanip>
11 using namespace std;
12 #include "FuncionesEstadisticas.h"
13
14 int main(int argc, char** argv) {
15     int num, fact;
16     cout << "Ingrese un numero: ";
17     cin >> num;
18     fact = factorial(num);
19     cout << "El factorial de " << num << " es : " << fact << endl;
20     return 0;
21 }
22

```

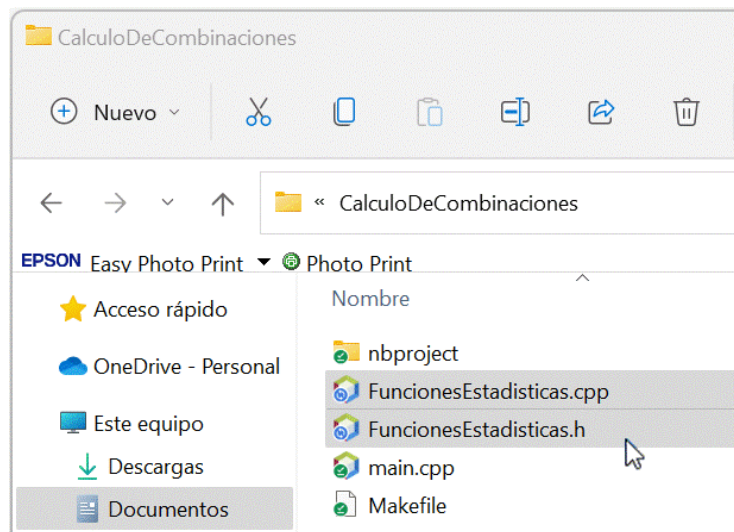
Una vez realizado esto, su programa debe ejecutarse sin problemas. Sin embargo, lo más importante de este proceso es que la próxima vez que en un programa necesitemos realizar el cálculo del factorial ya no tendremos que volver a escribirla, solo la usaremos como usamos la función **pow**.

Reutilización de código

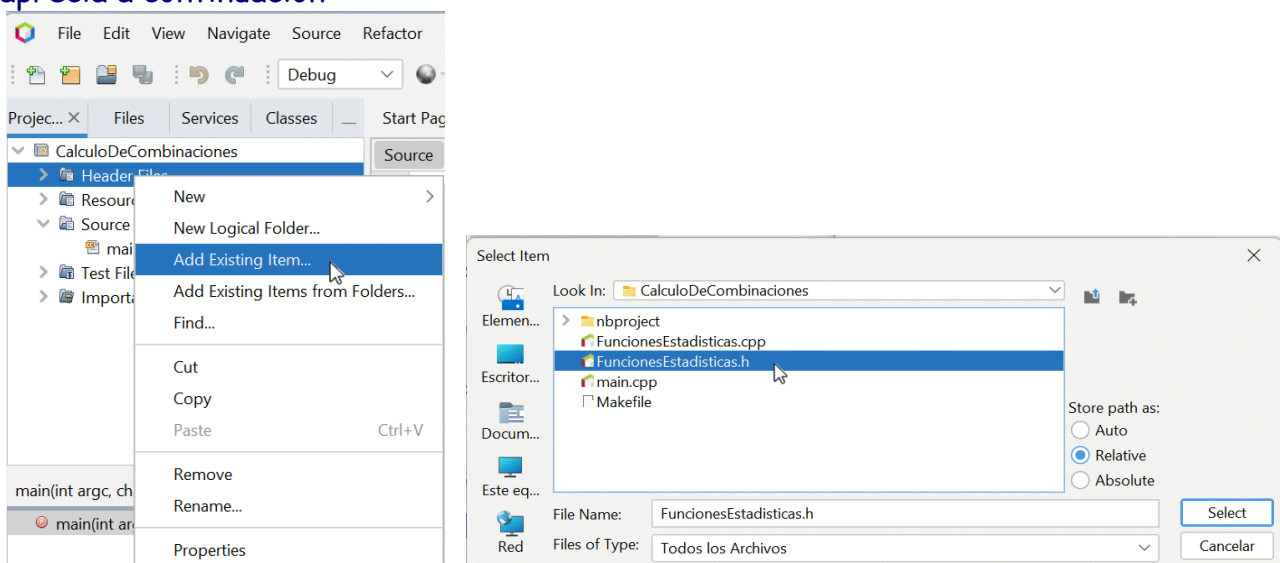
Ahora veremos cómo podemos implementar un proyecto que requiera el cálculo del factorial de un número sin volverlo a escribir. En el siguiente ejemplo queremos desarrollar un proyecto en el que se calcule las combinaciones de "n" elementos tomados de "p" en "p" sin repetición. Para esto debemos emplear la siguiente fórmula:

$$C_p^n = \binom{n}{p} = \frac{n!}{p! \cdot (n-p)!}$$

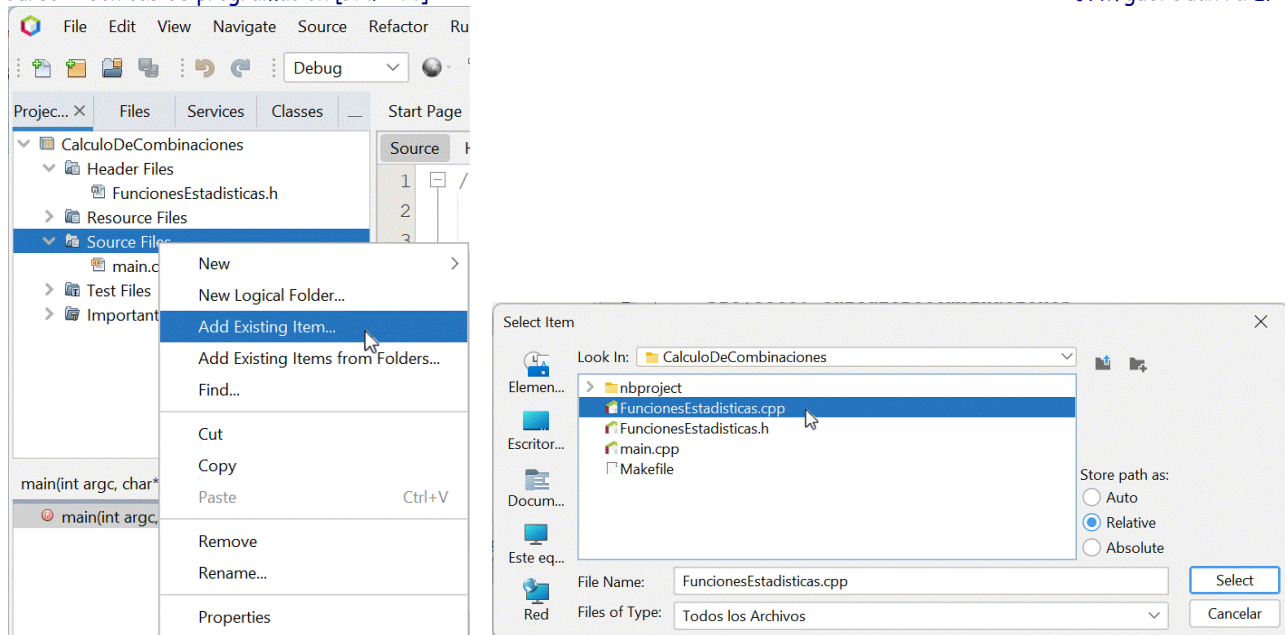
Creemos entonces en NetBeans el proyecto "CalculoDeCombinaciones". Luego, antes de comenzar a escribir el código del programa, debemos copiar, en la carpeta que contiene nuestro nuevo proyecto, el archivo de cabecera (.h) y de implementación (.cpp) donde están la definición e implementación de nuestra función factorial que se encuentra en la carpeta del proyecto anterior. La carpeta debe quedar como se muestra a continuación:



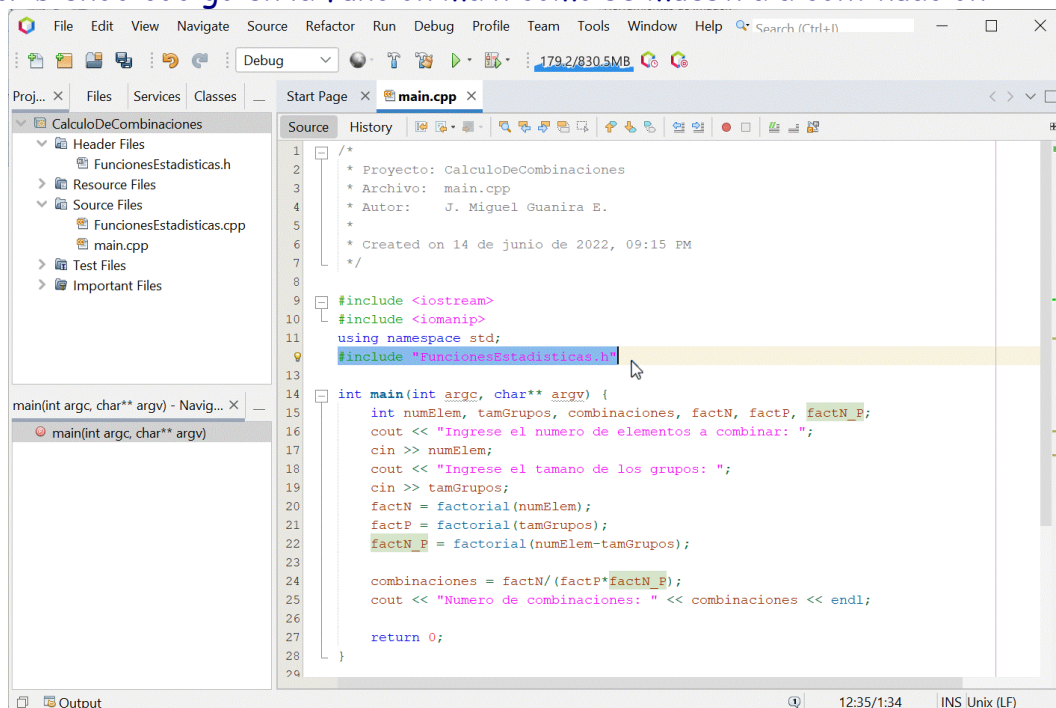
Luego debemos ligar al proyecto estos archivos, para esto nos dirigimos al NetBeans y en la ventana izquierda haremos algo similar a lo que hicimos en el anterior proyecto, pero en el menú que se despliega elegiremos la opción "Add Existing Item...", y en la ventana que aparece seleccionamos el archivo "FuncionesEstadistica.h" como se aprecia a continuación:



De la misma manera lo hacemos con el archivo que contiene la implementación de la función factorial, como se ve a continuación:



Una vez hecho esto, podemos escribir y ejecutar nuestro programa de combinaciones solo escribiendo código en la función main como se muestra a continuación:



Tipos de variables

Variables Globales: Una variable global es aquella variable que ha sido declarada fuera del ámbito de cualquier función, tienen como característica principal que, dependiendo de la ubicación de su declaración, pueden ser utilizadas dentro del código de las funciones del programa.

Variables Locales: Las variables locales, son variables que se definen dentro de una función, estas variables se crean en el instante en que se empieza a ejecutar la función y se destruyen cuando esta termina. Si la función es invocada varias veces en un programa, las variables locales declaradas en él, se crearán y destruirán tantas veces como la función sea invocada.

Esta característica hace que las variables definidas como locales, no puedan ser utilizadas por otra función, incluso dos funciones podrían definir sus propias variables locales y emplear los mismos nombres sin que se afecten unas contra otras.

Variables Estáticas: Las variables estáticas son variables que, como las variables locales, sólo se pueden emplear en la función que la declaró, pero a diferencia de éstas últimas no se destruyen cuando la función es nuevamente invocada, manteniendo el valor que tenía cuando terminó la ejecución la función la vez anterior.

La manera de declarar una variable estática es muy simple, ya que sólo hay que anteponer la palabra "**static**" a la declaración de la variable

Referencias

Una referencia es una variable que ocupa el mismo espacio de memoria que otra variable.

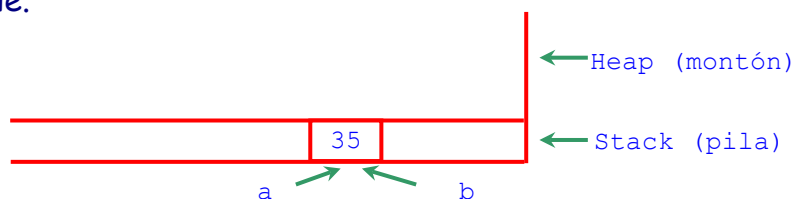
El siguiente ejemplo ilustra esta definición:

```
int a = 35;
int &b = a;

cout << "A = " << a << endl;
cout << "B = " << b << endl <<endl;

b = 179;
cout << "A = " << a << endl;
cout << "B = " << b << endl;
```

En el ejemplo, "a" es una variable entera común y corriente. Sin embargo, "b" también es una variable entera, pero al declararla anteponiéndole el operador & e igualándola a la variable "a", se convierte en una variable espacial, una "referencia", el sistema la ubicará a la variable "b" en el mismo espacio que la variable "a". La figura siguiente muestra ese detalle.



Las dos líneas siguientes imprimen los valores contenidos en las variables "a" y "b" y ambas mostrarán el valor de 35. En las tres últimas líneas vemos que se cambia solo el valor de "b" pero como ocupa el mismo espacio que "a", la variable "a" también se verá afectada y al imprimir sus valores ambas mostrarán 179.

Debe entender que la variable "b" NO ES UN PUNTERO, solo ocupa el mismo espacio que la variable "a".

Parámetros de una función

Al igual que se requieren que se ingrese información a los programas para que estos se puedan ejecutar de acuerdo a las necesidades del momento, en la mayoría de los casos se requiere introducir información a las funciones para que éstas puedan realizar su trabajo. Piense en la función sin o pow, ambas requieren de datos para realizar su

trabajo, sin requiere del valor de un ángulo en radianes, pow eleva un número a una potencia, por lo que se necesita de los dos valores para obtener el resultado esperado. Esta información, que es introducida a las funciones se denomina parámetros o argumentos.

Parámetros por valor: La característica principal de este tipo de parámetro es que, pase lo que pase dentro de la función, el valor de la variable que se use como parámetro no será alterado.

Para entender esto, analicemos el siguiente programa:

```
#include <iostream>
#include <iomanip>
using namespace std;
#include "FuncionesEstadisticas.h"

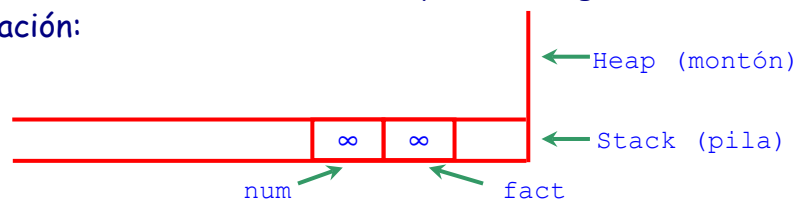
int main(int argc, char** argv) {
    int num, fact;
    cout << "Ingrese un numero: ";
    cin >> num;
    fact = factorial(num);
    cout << "El factorial de " << num << "es : "
         << fact << endl;
    return 0;
}
```

```
// FuncionesEstadisticas.cpp

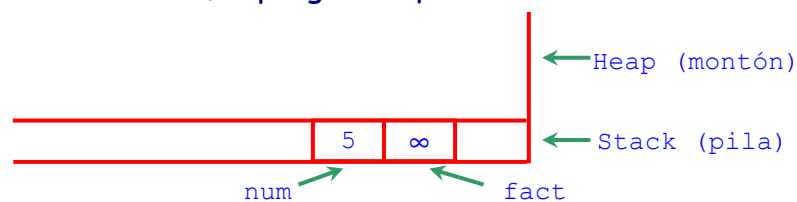
int factorial (int x){
    int f=1;
    for(int n = x; n>0; n--){
        f *= n;
    }
    return f;
}
```

El programa se ejecuta de la siguiente manera:

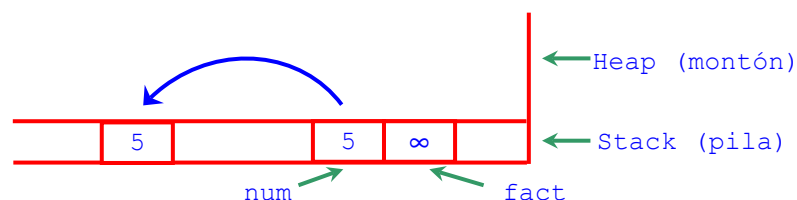
1. La ejecución de la función main empieza declarando las variables num y fact. Estas variables son colocadas, sin inicializar, en la pila del segmento de datos, como se muestra a continuación:



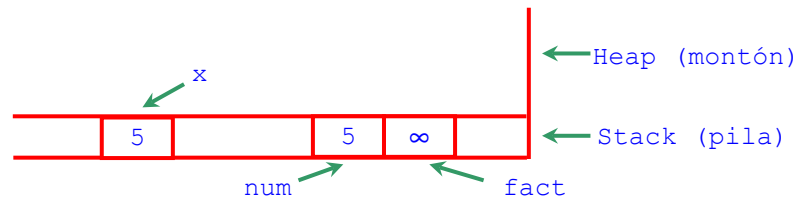
2. Luego se lee un valor para la variable num, colocándose ese valor en la posición de memoria asignada a esa variable, supongamos que se lee el valor de 5. Entonces:



3. Enseguida se llama a la función factorial, el programa toma el valor de la variable num y lo envía a otra parte de la pila, como se ve a continuación:



4. Luego el sistema entrega el control a la función factorial, en ese momento el sistema relaciona el nombre del parámetro "x" con el espacio de memoria donde se encuentra el valor enviado, como se aprecia a continuación:



5. A partir de allí la función factorial se ejecuta utilizando, cuando se requiera, el valor asignado a la variable "x", cualquier modificación que se haga a la variable "x" no afectará a la variable "num" definida en la función main. Es por eso que al parámetro se le nombra como parámetro por valor, porque envía el valor de la variable que se usa como argumento, a la función.

Parámetros por referencia: se denominan parámetros por referencia a aquellos parámetros que al ser modificados en la función modifican también a la variable empleada como parámetro. Para realizar esto, el lenguaje C++ emplea el concepto de **referencia**. A continuación, explicamos este proceso mediante el siguiente programa:

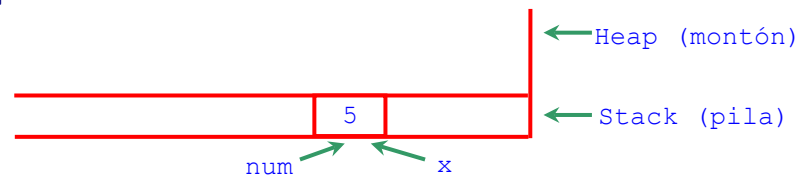
```
#include <stdio.h>
#include <stdlib.h>
#include "FuncionesEstadisticas.h"

int main(int argc, char** argv) {
    int num;
    cout << "Ingrese un numero: ";
    cin >> num;
    f(num);
    cout << "El valor de num es: " << num << endl;
    return 0;
}

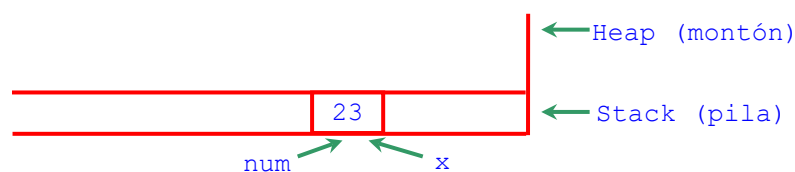
// FuncionesEstadisticas.h
void f (int &);

// FuncionesEstadisticas.cpp
void f (int &x){
    x =23;
}
```

1. Observe que en el llamado a la función "f" el parámetro es colocado como **num**, al igual que se hizo con la función "factorial" del ejemplo anterior, por lo tanto, no hay diferencia en el llamado entre un parámetro por valor y uno por referencia. La diferencia está en la declaración como en la implementación. Eso es, en la declaración se coloca un & luego del tipo de dato y en el caso de la implementación se coloca el & entre el tipo de dato y el parámetro. Esto quiere decir que el parámetro "x" se define como una referencia de la variable "num" y por lo tanto ocupa el mismo espacio de memoria como se ve a continuación:



2. Al ser "x" una referencia de la variable "num", cualquier modificación que se realice a "x" en la función modificará también el valor de la variable "num"



Observe que a diferencia del lenguaje C, el C++ no requiere el uso de punteros para el paso de parámetros por referencia.

Solución de problemas empleando diseño descendente y programación modular

A continuación, se presenta un problema que ilustra esta metodología.

Se desea escribir un programa en lenguaje C que permita imprimir las facturas que se han elaborado por las ventas realizadas en una tienda.

Para realizar esta labor, se cuenta con tres archivos de texto con la información requerida. El primer archivo, denominado "Productos.txt", contiene la información de todos los productos que comercializa la tienda. Este archivo, con la finalidad de simplificar la solución del problema y que así nos podamos concentrar en la metodología, es similar al que se muestra a continuación:

Productos.txt	
123456	1199.90
345678	345.50
626262	689.99
...	

En este archivo se encuentra en cada línea el código del producto y su precio unitario.

El segundo archivo, denominado "Clientes.txt", contiene la información de todos los clientes que han comprado alguna vez algún producto en la tienda. De manera similar que, en el archivo anterior, el archivo es parecido al que se muestra a continuación:

Clientes.txt			
17000095	[Hijar Pairazaman Jenny Delicia]	5	39827505
81092545	[Cabrera Canales Guillermo Edric]	3	48450092
21000270	[Justino Cruz Williams]	2	69282573
...			

En este archivo se encuentra en cada línea, el DNI del cliente, su nombre (encerrado entre corchetes []), el código de la ciudad donde vive y su teléfono.

El tercer archivo contiene la información de las facturas que se han emitido en la tienda en un periodo de tiempo. El archivo contiene en cada línea la información necesaria para poder elaborar cada factura, esto es: el número de factura, la fecha en que se emitió, el DNI del cliente que hizo la compra y la lista de productos comprados, aquí sólo se consigna el código y la cantidad comprada de cada producto, pudiendo haber muchos productos en una línea. El archivo es similar al que se muestra a continuación:

Facturas.txt						
10001	23/1/2018	81300045	445566	5	890988	10...
10002	1/2/2018	23764590	345678	3		
10036	5/2/2108	11223344	626262	1	445566	21 ...
...						

Finalmente, el programa deberá imprimir las facturas de una manera similar a lo que se muestra a continuación:

TIENDA DE ABARROTES TP			
=====			
Numero:	10001	Fecha:	10/03/2020

DNI:	85054426	Nombre:	Alamo Pairazaman Miguel Roberto
Ciudad:	6	Telefono:	45595843
=====			
CODIGO	CANTIDAD	P.U.	SUBTOTAL

956872	4	96.08	384.32
488600	6	10.49	62.94
606509	11	103.27	1135.97
...

Total:			3631.82
=====			
Numero:	10002	Fecha:	10/03/2020
...			

Solución: Al tratarse de un problema complejo. La solución debe plantearse empleando un diseño descendente, modulando el programa mediante funciones. El aplicar el diseño descendente nos permitirá ir encontrando la solución en el camino e ir escribiendo el código del programa aun así no tengamos una idea muy clara de cada detalle de la solución completa.

Primero analicemos a grandes rasgos las tareas que hay que realizar para elaborar el problema; por un lado, se tendrá que manejar esos archivos descritos en el problema, por lo que habrá que definir las correspondientes variables de archivo. Luego una tarea que debemos hacer es preparar los archivos (asignarlos, abrirlos, verificar su apertura) para que podamos trabajar con ellos, luego que los archivos estén abiertos habrá que procesar los datos para conseguir la impresión de las facturas. Finalmente se deberá cerrar esos archivos.

Entonces escribamos estas tareas como primer paso al desarrollo del problema:

```
int main(int argc, char** argv) {
    // Tarea 1.- Definir las variables de archivo y prepara los archivos
    // Tarea 2.- Emitir las facturas
    // Tarea 3.- Cerrar los archivos
    return 0;
}
```

Analicemos las tareas, la primera es una tarea trivial y repetitiva y, por razones que se estudiarán más adelante, no tenemos por ahora herramientas para poder manejarla en una función por lo que tendremos que implementarla en la función main. Las otras tareas si las podremos implementar con funciones. Por lo que el programa lo podemos modificar de la forma siguiente:

```
#include <iostream>
#include <fstream>
using namespace std;
#include "FuncionesAuxiliares.h"

int main(int argc, char** argv) {
    // Tarea 1.- Definir las variables de archivo y prepara los archivos
    ifstream archFact ("Facturas.txt",ios::in);
    if(not archFact.is_open()){
        cout << "ERROR: No se pudo abrir el archivo Facturas.txt" << endl;
        exit(1);
    }

    ifstream archProd ("Productos.txt",ios::in);
    if(not archProd.is_open()){
        cout << "ERROR: No se pudo abrir el archivo Productos.txt" <<endl;
        exit(1);
    }

    ifstream archCli ("Clientes.txt",ios::in);
    if(not archCli.is_open()){
        cout << "ERROR: No se pudo abrir el archivo Clientes.txt" << endl;
        exit(1);
    }

    ofstream archRepDeFacturas ("ReporteDeFacturas.txt",ios::out);
    if(not archRepDeFacturas.is_open()){
        cout << "ERROR: No se pudo abrir el archivo ReporteDeFacturas.txt"
            << endl;
        exit(1);
    }
    // Tarea 2.- Emitir las facturas
    emitirReporteDeFacturas(archFact, archProd, archCli, archRepDeFacturas);

    // Tarea 3.- Cerrar los archivos
    // El destructor cierra automáticamente los archivos
    return 0;
}
```


Luego de haber escrito este código podríamos decir que hemos concluido la elaboración de la función main, ya no requerimos agregarle algo más. Sin embargo, como hemos visto anteriormente, aún no hemos terminado con el proyecto, tenemos que declarar e implementar la función **emitirReporteDeFacturas**. Lo importante de esto es que hay que observar que, dejando de lado la declaración y apertura de los archivos, el código de la función main es muy simple y fácil de entender.

Observe que la tarea 3, que consiste en cerrar los archivos, no implementar ya que la variable de archivo cuenta con un método (función) que se ejecuta automáticamente y que cierra los archivos.

Una buena práctica de programación es la de procurar desarrollar funciones de poca extensión de modo que se pueda entender fácilmente. Una función que tenga más de 20 líneas de código es sospechosa de no haber sido analizada correctamente, que es ineficiente y hasta que puede contener errores.

Declaración e implementación de la función: **emitirReporteDeFacturas**.

Como la función es propia del problema que estamos resolviendo y difícilmente pueda ser reutilizado en otros programas, definiremos una sola biblioteca de funciones para este fin. Con la metodología explicada anteriormente definiremos los archivos "FuncionesAuxiliares.h" y "FuncionesAuxiliares.cpp", el primero con el siguiente código:

```
/*
 * Proyecto: Facturas
 * Archivo:  FuncionesAuxiliares.h
 * Autor:    J. Miguel Guanira E.
 *
 * Created on 16 de junio de 2022, 08:00 PM
 */

#ifndef FUNCIONES_AUXILIARES_H
#define FUNCIONES_AUXILIARES_H

void emitirReporteDeFacturas(istream &, istream &, istream &, ostream &);

#endif /* FUNCIONES_AUXILIARES_H */
```

La función emitirReporteDeFacturas es mucho más complicada, primero debemos decidir por qué archivo empezaremos la tarea, esto es muy importante, de no hacerlo correctamente no llegaremos a solucionar el problema. En este caso, si queremos emitir las facturas una a una, debemos empezar precisamente por el archivo de facturas. La información que tiene este archivo, si es cierto que no está completa, registra los DNI y códigos de los productos en cada factura, por lo que con esos datos podemos buscar en los otros archivos lo que nos falta. Sin embargo, si comenzamos por el archivo de clientes, tendremos el problema que allí solo tenemos la información de cada cliente, no hay allí relación con los otros, lo mismo pasa con el archivo de productos.

Entonces empezaremos por el archivo de facturas. Como en ese archivo los datos de cada factura están todos en una línea, la función deberá implementar un ciclo iterativo en el que se en cada ciclo se procese una factura. El esquema será similar al siguiente:

```
while (true){
    // Procesar una factura
}
```

Los datos de una factura se pueden dividir en tres partes, los datos de la factura propiamente dicho (número de factura y fecha), los datos del cliente (solo el DNI) y los datos de productos que compró (códigos y cantidades).

Si observamos el reporte final, apreciaremos que con los primeros datos podemos imprimir la información de la factura, con el DNI de cliente podemos buscar los datos que nos faltan del cliente en el archivo de clientes, sin embargo, esta tarea puede ser algo compleja y como por ahora no sabemos cómo hacerlo, simplemente lo anotaremos como una tarea pendiente. El procesamiento de los productos también parece ser complicado por lo que haremos lo mismo que con la búsqueda de los clientes. La impresión del encabezado y subtítulos del reporte también podríamos encargarla a funciones, esto para que el código se más simple.

De acuerdo a esto, la función emitir facturas se puede implementar de la siguiente manera:

```
#include <iostream>
#include <fstream>
#include <iomanip> #include "FuncionesAuxiliares.h"
#define MAX_CAR_LINEA 60

void emitirReporteDeFacturas(istream &archFact, istream &archProd,
                             istream &archCli, ostream &archRepDeFacturas){
    // Tarea 2.1.- Imprimirlos encabezados
    imprimirEncabezado(archRepDeFacturas);
    // Tarea 2.2.- Repetir por cada factura
    while(true){
        // Tarea 2.1.- Leer e imprimir los dato de la factura: Número y la fecha
        leerEImprimirDatosDeLaFactura(archFact,archRepDeFacturas);
        if(archFact.eof()) break;
        // Tarea 2.3.- Leerel DNI y completar e imprimir los datos del cliente
        leerEImprimirLosDatosDelCliente(archFact,archCli,archRepDeFacturas);
        // Tarea 2.4.- Leer los datos de las compras del cliente, Completarlos
        // y mostrarlos en el reporte con sus totales
        imprimirSubTiulos(archRepDeFacturas);
        leerEImprimirLosDatosDeLosProductos(archFact,archProd,
                                             archRepDeFacturas);
    }
}
```

Las funciones pendientes deberán declararse en el archivo FuncionesAuxiliares.h así:

```
#ifndef FUNCIONES_AUXILIARES_H
#define FUNCIONES_AUXILIARES_H

void emitirReporteDeFacturas(istream &, istream &, istream &, ostream &);

void imprimirEncabezado(ostream &);
void leerEImprimirDatosDeLaFactura(istream &, ostream &);
void leerEImprimirLosDatosDelCliente(istream &, istream &, ostream &);
void imprimirSubTiulos(ostream &);
void leerEImprimirLosDatosDeLosProductos(istream &,istream &, ostream &);

#endif /* FUNCIONES_AUXILIARES_H */
```

Las funciones **imprimirEncabezado** e **imprimirSubTiulos** son muy simples de desarrollar y lo menos importante en el problema, por lo que las dejaremos para el final.

Empezaremos por la función **leerEImprimirDatosDeLaFactura**, aquí leeremos los dos primeros datos de archivo de facturas y los imprimiremos en el reporte:

```

void leerEImprimirDatosDeLaFactura(ifstream &archFact,
                                   ofstream &archRepDeFacturas){
    int numFact, dd, mm, aa;
    char c;
    archFact >> numFact;
    if(archFact.eof())return;
    archFact >> dd >> c >> mm >> c >> aa;
    archRepDeFacturas << setw(9) << "Numero: " << setw(6) << numFact
        << setw(10) << "Fecha: " << setfill('0') << setw(2)<< dd
        << '/' << setw(02) << mm << '/' << setw(4) << aa << setfill(' ')
        << endl;
    imprimeLinea(archRepDeFacturas, '-', MAX_CAR_LINEA);
}

```

Ahora analicemos lo que debemos hacer para imprimir los datos del cliente al que se le emitió la factura. Primero habrá que lee el siguiente dato del archivo de facturas, el DNI, y luego debemos buscar esa dato en el archivo de clientes. El código la mostramos a continuación.

```

void leerEImprimirLosDatosDelCliente(ifstream &archFact, ifstream &archCli,
                                     ofstream &archRepDeFacturas){
    int dni;
    archFact >> dni;
    buscarEImprimirElCliente(dni, archCli, archRepDeFacturas);
    imprimeLinea(archRepDeFacturas, '=', MAX_CAR_LINEA);
    // while(archFact.get()!='\n');
}

```

Observe que la tarea de buscar al cliente la encargamos a otra función, esto es una buena práctica de programación, toda operación de búsqueda debe hacerse en una función independiente.

Para realizar la búsqueda debemos recorrer línea por línea el archivo de clientes, hasta encontrar el cliente cuyo DNI coincida con el que buscamos. Para hacer esto, primero debemos asegurarnos que el proceso empiece en la primera línea del archivo. Para esto usaremos un método (función) definido en la variable de archivo denominado **seekg**. Luego debemos tomar en cuenta que apenas encontremos al cliente debemos interrumpir inmediatamente el proceso iterativo.

El código de la función lo mostramos a continuación:

```

void buscarEImprimirElCliente(int dni, ifstream &archCli,
                              ofstream &archRepDeFacturas){
    int dniCli, ciudad, telefono;

    archCli.seekg(0, ios::beg); //El indicador del archivo se pone al inicio
    while(true){
        archCli >> dniCli;
        if(archCli.eof()) break;
        if(dni == dniCli) break; // Si se encontró se sale del while
        else
            while(archCli.get()!='\n');// Si no se encontró se descarta la línea
    }

    if(dni == dniCli){ //Si lo encontró
        archRepDeFacturas << setw(5) << "DNI:" << setw(10) << dni
            << setw(10) << "Nombre: ";
        leeImprimeNombreDelimitado(archCli, archRepDeFacturas, '[', ' ');
        archCli >> ciudad >> telefono;
        archRepDeFacturas << endl << setw(9) << "Ciudad: "
            << setw(6) << ciudad << setw(12) << "Telefono: "
            << setw(8) << telefono << endl;
    }
    else{ // Si NO lo encontró
        cout << "ERROR: No se encontro el DNI: " << dni << endl;
        exit(1);
    }
}

```

La impresión del nombre del cliente no es una tarea sencilla, por lo que se la encargamos a otra función. El código es el siguiente:

```
void leeImprimeNombreDelimitado(istream &archCli,
                                ostream &archRepDeFacturas, char ini, char fin){
    char c;
    int numCar=0, numBlancos;
    while((c = archCli.get()) != ini);
    while(true){
        c = archCli.get();
        if(c == fin)break;
        numCar++;
        archRepDeFacturas.put(c);
    }
    archRepDeFacturas << endl;
}
```

En la función **leerEImprimirLosDatosDeLosProductos**, debemos desarrollar un ciclo iterativo en el que en cada ciclo leamos el código y la cantidad comprada de un producto, buscar el precio del producto, calcular el subtotal, acumular el total e imprimir todos esos datos. El código será el siguiente:

```
void leerEImprimirLosDatosDeLosProductos(istream &archFact,
                                          istream &archProd, ostream &archRepDeFacturas){
    int codPro, cant;
    double subTotal, total=0;
    double precio;
    archRepDeFacturas.precision(2);
    archRepDeFacturas << fixed;
    while(1){
        archFact >> codPro >> cant;
        precio = buscarProducto(codPro, archProd);
        if(precio >= 0.0){
            subTotal = cant*precio;
            total += subTotal;
            archRepDeFacturas << setw(10) << codPro << setw(13) << cant
                << setw(15) << precio << setw(16) << subTotal << endl;
        }
        if(archFact.get() == '\n') break;
    }
    imprimeLinea(archRepDeFacturas, '-', MAX_CAR_LINEA);
    archRepDeFacturas << setw(44) << "Total: " << setw(10) << total << endl;
    imprimeLinea(archRepDeFacturas, '=', MAX_CAR_LINEA);
}
```

La función **buscarProducto** es parecida a la que busca el cliente, pero como esta solo devuelve un dato (el precio), este valor lo devolveremos por la de la orden return. Siempre que se espere que una función de búsqueda devuelva un valor, se debe considerarse que el valor buscado no sea encontrado y ese caso se debe devolver un valor que así lo indique, en este caso como se trata del precio de un producto, un valor aceptable sería -1 ya que no puede haber un precio negativo. El código será el siguiente:

```
double buscarProducto(int codPro, istream &archProd){
    int codigo;
    double precio;
    archProd.seekg(0, ios::beg);
    while(true){
        archProd >> codigo;
        if(archProd.eof())break;
        archProd >> precio;
        if(codigo == codPro) return precio; // Lo encontró
    }
    return -1.0; // No lo encontró
}
```

Finalmente implementamos las tareas más sencillas.


```

void imprimirEncabezado(ofstream &archRepDeFacturas){
    archRepDeFacturas << setw(40) << "TIENDA DE ABARROTES TP" << endl;
    imprimeLinea(archRepDeFacturas, '=', MAX_CAR_LINEA);
}

void imprimirSubTiulos(ofstream &archRepDeFacturas){
    archRepDeFacturas << setw(10) << "CODIGO" << setw(16) << "CANTIDAD"
        << setw(12) << "P.U." << setw(17) << "SUBTOTAL" << endl;
    imprimeLinea(archRepDeFacturas, '-', MAX_CAR_LINEA);
}

void imprimeLinea(ofstream &arch, char car, int cant){
    for(int i=0; i<cant; i++){
        arch.put(car);
    }
    arch << endl;
}

```

El archivo **FuncionesAuxiliares.h** quedará finalmente así:

```

#ifndef FUNCIONES_AUXILIARES_H
#define FUNCIONES_AUXILIARES_H

void emitirReporteDeFacturas(ifstream &, ifstream &, ifstream &, ofstream &);
void leerEImprimirDatosDeLaFactura(ifstream &, ofstream &);
void leerEImprimirLosDatosDelCliente(ifstream &, ifstream &, ofstream &);
void leerEImprimirLosDatosDeLosProductos(ifstream &, ifstream &, ofstream &);
void imprimirEncabezado(ofstream &);
void imprimirSubTiulos(ofstream &);
void buscarEImprimirElCliente(int, ifstream &, ofstream &);
void leeImprimeNombreDelimitado(ifstream &, ofstream &, char, char);
double buscarProducto(int, ifstream &);
void imprimeLinea(ofstream &, char, int);

#endif /* FUNCIONES_AUXILIARES_H */

```