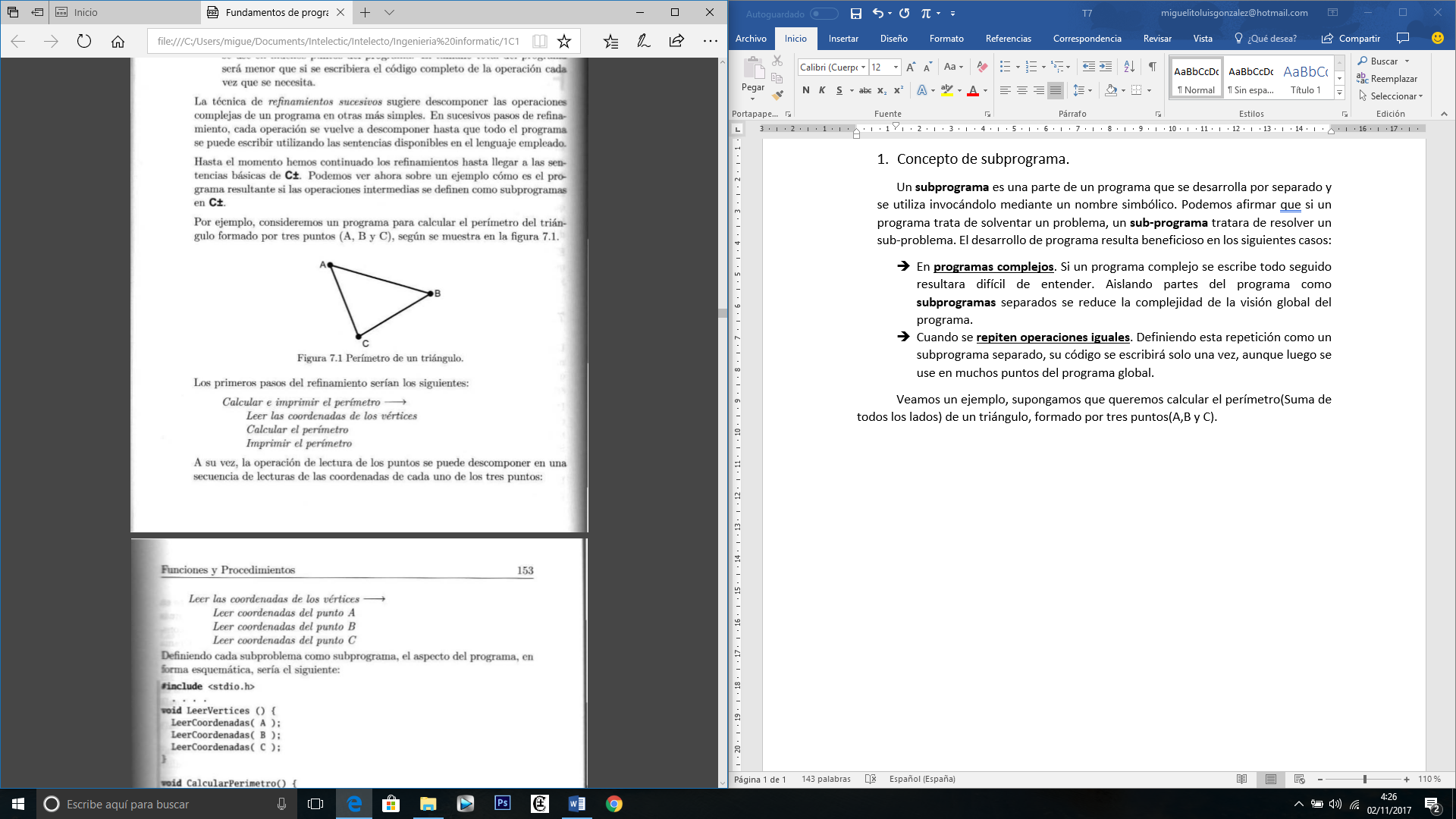
**Tema 7: Funciones y Procedimientos.**

1. **Concepto de subprograma.**

Un **subprograma** es una parte de un programa que se desarrolla por separado y se utiliza invocándolo mediante un nombre simbólico. Podemos afirmar que, si un programa trata de solventar un problema, un **sub-programa** tratara de resolver un sub-problema. El desarrollo de programa resulta beneficioso en los siguientes casos:

* En **programas complejos**. Si un programa complejo se escribe todo seguido resultara difícil de entender. Aislando partes del programa como **subprogramas** separados se reduce la complejidad de la visión global del programa.
* Cuando se **repiten operaciones iguales**. Definiendo esta repetición como un subprograma separado, su código se escribirá solo una vez, aunque luego se use en muchos puntos del programa global.

Veamos un ejemplo, supongamos que queremos calcular el perímetro (Suma de todos los lados) de un triángulo, formado por tres puntos (A, B y C).

Los primeros pasos de refinamiento serían los siguientes:

*Calcular e Imprimir el perímetro 🡪*

*Leer las coordenadas de los vértices.*

*Calcular el perímetro.*

*Imprimir el perímetro.*

Recordemos que la técnica de refinamientos sucesivos consistía en descomponer operaciones complejas en operaciones más simples. Si transformáramos esas acciones simples en subprogramas. El programa se vería así:

#include <stdio.h>

void LeerVertices () {

LeerCoordenadas( A );

LeerCoordenadas( B );

LeerCoordenadas( C );

}

void CarcularPerimetro{

……………..

}

void ImprimirPerimetro{

………………………….

}

Int main() {

LeerVetices();

CalcularPerimetro();

ImprimirPerimetro();

}

1. **Funciones.**

Cuando se diseña y desarrolla un programa aparecen con frecuencia operaciones significativas que da como resultado un valor simple y único en funcion de ciertos parámetros. Este tipo de operaciones se pueden considerar programas más concretamente funciones. Una función es un tipo de subprograma que calcula como resultado un valor único a partir de otros valores dados como argumento asemajandoze bastante a la idea matemática de función. Por ejemplo:

|  |  |
| --- | --- |
| *Potencia* |  |
| *Volumen de un cubo* |  |
| *Area de un triángulo* |  |
| *Distancia entre dos puntos* |  |

* 1. **DEFINICIÓN DE FUNCIONES EN C++.**

El primer paso en el manejo de una función es declarar su interfaz. Esta declaración incluye su nombre, los argumentos que necesita con el correspondiente tipo para cada uno de ellos, y el tipo de resultado que proporciona. En ++ esto se realiza escribiendo una cabecera de funcion de la siguiente forma.

TipoResultado NombreFuncion (Tipo1 argumento1, Tipo2 argumento2,….){

*Codigo del subprograma;*

*}*

*Algunos ejemplos:*

*|****float*** *Potencia (* ***float*** *x,* ***int*** *n)*

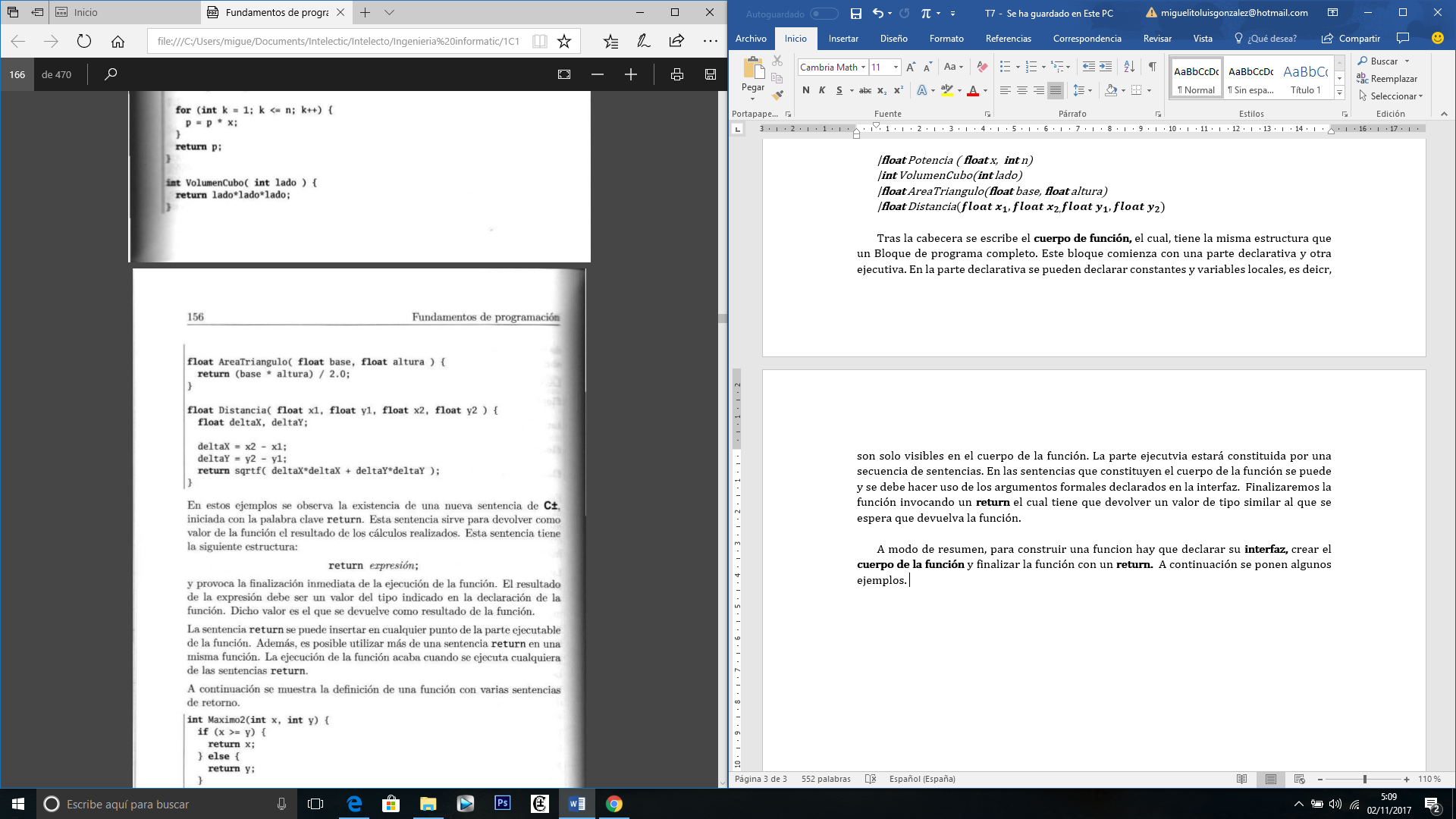
*|****int*** *VolumenCubo(****int*** *lado)*

*|****float*** *AreaTriangulo(****float*** *base,* ***float*** *altura)*

*|****float*** *Distancia*

Tras la cabecera se escribe el **cuerpo de función,** el cual, tiene la misma estructura que un Bloque de programa completo. Este bloque comienza con una parte declarativa y otra ejecutiva. En la parte declarativa se pueden declarar constantes y variables locales, es decir, son solo visibles en el cuerpo de la función. La parte ejecutvia estará constituida por una secuencia de sentencias. En las sentencias que constituyen el cuerpo de la función se puede y se debe hacer uso de los argumentos formales declarados en la interfaz. Finalizaremos la función invocando un **return** el cual, finaliza la ejecución de la función y tiene que devolver un valor de tipo similar al que se espera que devuelva la función. Cabe destacar que se pueden usar más de un **return**, por ejemplo, en un IF, en cada una de las opciones del if puede ponerse un return.

A modo de resumen, para construir una funcion hay que declarar su **interfaz,** crear el **cuerpo de la función** y finalizar la función con un **return.** A continuación se ponen algunos ejemplos.



* 1. **Uso de funciones.**

Para usar una funcion en los cálculos de un programa se invoca dicha función escribiendo su nombre y a continuación entre parentesis, los valores concretos de los argumentos separados por coma. Esta invocación de la función representa un valor del tipo de la función, que podra ser usado como operando en una expresión aritmetica o en cualquier parte del programa en que sea valido escribir una expresión de ese tipo.

Denominaremos Argumentos Reales, a aquellas variables, constantes…etc. Que se envien desde el programa principal, o desde un subprograma a otro subprograma y argumentos Formales, a aquellos argumentos que se manipulan dentro del subprograma objeto. Por ejemplo:

#include <stdio.h>

int sumar (int a, int b){ 🡪 a y b son **argumentos formales**

int result;

result = a + b;

}

int main(){

int num1,num2;

Sumar(num1,num2); 🡪 num1 y num2 son **argumentos reales**

}

Al invocar una función los valores suministrados para la función tienen que corresponder en número y tipo con los argumentos de la definición (argumentos formales). Por ejemplo:

|VolumenCubo( ladoCubo) > 27

|valorPotencia = Potencia( base, exponente);

|area = AreaTriangulo(Distancia(xA,yA,xB,yB),medidaAltura))

* 1. **Funciones predefinidas.**

Se consideran funciones predifinidas las que forman parte del propio lenguaje de programación. Estas funciones están siempre disponibles en cualquier programa. Algunos lenguajes de programación tienen un repertorio de funciones amplio. Los lenguajes C y C++ disponen de pocas funciones predefinidas, además, estas funciones prefinidas sólo resultan útiles en uso avanzado del lenguaje, por ello, no se usa en este curso este tipo de funciones.

* 1. **Funciones estándar.**

Al realizar programas en C+- podemos utilizar funciones que estén definidas en módulos ya redactados de antemano. Algunos módulos constituyen *librerías estándar* y estan disponibles en la mayoria de compiladores C/C++. Tal y como se dijo al hablar de las operaciones de lectura y escritura o para el manejo de caraceres, a diferencia de las funciones predefinidas para utilizar una función estandar antes en la parte de “directiva de compilación” del programa se deben invocar las librerias que contienen estan funcion es estandar. Por ejemplo, las funciones estándar de la librería <ctype.h> se centran principalmente en el manejo de las diferentes clases de caracteres. Este modulo incluye:

|  |  |
| --- | --- |
| bool isalpha( char c ) | Indica si c es una letra |
| bool isascii( char c ) | Indica si c es un carácter ascci |
| bool isblank(char c) | Indica si c es un carácter de espacio o tabulación. |
| bool isdigit( char c ) | Indica si c es un digito decimal |
| bool islower(char c) | Indica si c es una letra minúscula. |
| bool isspace( char c ) | Indica si c es un espacio en blanco o un salto de línea. |
| bool isupper( char c ) | Indica si c es una letra mayúscula |
| char tolower( char c) | Devuelve la minúscula correspondiente a c. |
| Char toupper( char c) | Devuelve la mayúscula correspondiente a c. |

El tipo de resultado **bool** sirve para indicar si un resultado es verdadero o falso. Se introducira en el tema 9. En lo referente a funciones matemáticas se dispone del módulo de funciones estándar **math.h** ; este modulo esta compuesto por un gran número de funciones matemáticas con nombres distintos dependiendo del tipo de argumento y el tipo de resultado, algunas son las siguientes:

|  |  |
| --- | --- |
| float sqrtf ( float x ) | Raíz cuadrada de x |
| float expf ( float x ) | Exponencial de e^x |
| float logf( float x ) | Logaritmo neperiano de x |
| Float powf( float x, float y) | Potencia de x^y |
| Float sinf(float x) | Seno de x |
| Float cosf(float x) | Coseno de x |
| Float tanf(float x) | Tangente de x |
| Float atanf(float x) | Arcotangente de x |
| Float roundf(float x) | Valor de x redondeado a entero |

1. **Procedimientos.**

Un **procedimiento** es un subprograma que realiza una determinada acción. A diferencia de una función, un procedimiento no tiene como objetivo devolver un dato obtenido por cálculo, se trata de una sentencia o una grupo de sentencias que realizan una acción y permite darles un nombre por el que se las puede identificar posteriormente. Estas sentencias se pueden parametrizar mediante una serie de argumentos. Así, otra forma de ver a los procedimientos es como acciones parametrizadas. Por ejemplo, en el desarrollo de un programa podemos identificar acciones como:

*Trazar una línea de longitud dada.*

*Imprimir el resultado*

*Ordenar dos balores*

*Leer las coordenadas de un punto.*

* 1. **Definicion de procedimientos**

La definición en C+- de un procedimiento es prácticamente igual a la de una función:

**void** NombreProcedimiento ( Tipo1 argumento1, Tipo2 argumento2,…..){

Acciones a realizar.

}

La diferencia principal es que no se declara el tipo de valor del resultado, ya que no existe dicho valor. La palabra reservada void es la que indica que no hay un resultado de ningún tipo. Además, con cierta frecuencia interesa definir procedimientos sin argumentos. En estos casos sólo es necesario dar el nombre, y no habra nada entre los parentesis. También dentro de los procedimientos se puede añardir la expresión **return;** Esta acción sirve para deterner la ejecución del procedimiento.

* 1. **Uso de un procedimiento.**

Para usar un procedimiento hay que invocarlo escribiendo su nombre y a continuación, en caso de que los haya, los valores de los argumentos particulares en esa llamada separados por comas, escribiendo la siguiende sentencia:

*NombreProcedimiento( argumento1, argumento2, …. );*

El uso de procedimientos produce la siguientes acciones:

1. Se evaluan las expresiones de los valores de los argumentos.
2. Se asignan dichos valores a los correspondientes argumentos formales.
3. Se ejecuta el codigo de la definición de procedimiento, hasta alcanzar el final de bloque o una sentencia de retorno.
4. El programa que invocó al procedimiento continúa en el punto siguiente a la sentencia de llamada.

* 1. **Procedimientos estándar.**

Al igual que las funciones estándar, en los módulos estándar asocaicados a cada compilador de C/C++ se dispone de diversos procedimientos estándar que pueden utilizarse sin más que hacer uso de la directiva **#include** del correspondiente modulo. Por ejemplo, para lectura y escritura de datos se usa el modulo <stdio.h> .

1. **Paso de argumentos.**

La manera fundamental de comunicar información entre las sentencias de un subprograma y el programa que lo tiliza es mediante los argumentos. En C+- existen dos formas distintas de realizar esta comunicación, que se denominan, *paso de argumentos por valor y paso de argumentos por referencia****.***

Llamaremos **argumentos reales** a los argumentos que son enviados desde un programa a un subprograma; y llamaremos **argumentos formales** a los argumentos que se escriben en la interfaz del subprograma.

* 1. **Paso de argumentos por valor.**

Los argumentos representan valores que se transmiten del programa que llama al sub-programa. En el caso de las funciones, hay además un valor de retorno. Que es el valor de la función que se transmite desde el subprograma hacia el programa que llamó. Los **argumentos reales** en la llamada del subprograma pueden darse en general en forma de expresiones, cuyos tipos de valor deben ser compatible en asignación con los tipos de los **argumentos formales**.

El modo de **paso de argumentos por valor** implica que los elementos usados como argumentos en la llamada al subprograma no pueden ser modificados por la ejecución de las sentencias del subprograma. Es decir, los datos que se envían desde el programa principal (**argumentos reales**) al sub-programa (convirtiéndose en **argumentos formales**) pueden ser modificados, remplazados…etc. Pero estos cambios no afectarán a los datos que hay en el programa principal. La metodología que sigue es la siguiente:

1. Se evalúan las expresiones de los argumentos reales usados en la llamada.
2. Los valores obtenidos se copian en los argumentos formales.
3. Los argumentos formales se usan como variables locales dentro del subprograma, es decir, que si sufren cambios los argumentos formales no alteraran a los argumentos reales.
   1. **Paso de argumentos por referencia.**

Cuando queremos que el subprograma modique las variables que se envian desde el programa principal hacemos un **paso de argumentos por referencia**. Para espicificar que se esta realizando un paso de argumentos por referencias utilizamos el simbolo “ & “ de la siguiente manera:

TipoResultado Nombre (Tipo Argumento **&** argumento,….. )

Con el símbolo “&” estamos indicando que el argumento real puede ser modificado en el subprograma.

La metodología que sigue es la siguiente:

1. Se evalúan las expresiones de los argumentos reales usados en la llamada.
2. Los valores obtenidos se copian en los argumentos formales.
3. Los argumentos formales se usan como variables dentro del subprograma, pero si sufren cambios, dichos cambios alteraran también a las variables del programa principal.
4. **Estructura de bloques.**

La definición de un subprograma esta limitada por una cabecera o interfaz, y un bloque de cógigo que es el cuerpo del subprograma. Ese bloque de código constituye una barrera de visibilidad que hace que los elementos declarados dentro del cuerpo de un subprograma no sean visibles desde el exterior.

Los elementos definidos en el ámbito más externo son elementos *globales***,** mientras que los elementos definidos en el interior del bloque de un subprograma son elementos locales a dicho subprograma.

1. **Recursividad de subprogramas.**

Cuando un subprograma se hace una llamada así mismo podemos afirmar que se trata de un programa recursivo. Para verlo de manera más clara utilizaremos el algoritmo para el calculo de un número natural.

