**TEMA 9: Definición de tipos.**

En este tema veremos:

* Tipos escalares definidos.
* Tipo **bool.**
* Definición de tipos estructurados de datos (**Array** o formación; **Struct** o registro).
* Cadena o vectores de caracteres.

1. **Tipos definidos.**

Los lenguajes de alto nivel nos permiten definir nuestros propios tipos de datos. Los tipos pre-definidos: **int, char** y **float** nos sirven para manejar números y caracteres. Pero si queremos diseñar, por ejemplo, un programa para jugar al ajedrez resulta mucho más adecuado utilizar datos que representan de manera más directa a los peones, caballos, torres, alfiles, reyes y damas. Lo mismo ocurre para manejar otro tipo de datos como los días de la semana, deportes, colores, alimentos…etc.

El tipo de dato establece los posibles valores que puede tomar ese dato. Además, al igual que sucedía con los tipos predefinidos, a cada nuevo tipo que se define se asocian un conjunto de operaciones que se pueden realizar con él. Por tanto, la definición de tipos supone crear un nuevo nivel de abstracción dentro del programa.

Para declarar los tipos de datos se realiza antes de la declaración del programa principal o los subprogramas, sin embargo, la inicialización si se puede declarar dentro del programa principal o subprograma antes de la parte ejecutiva. Dicha declaración se inicia con la palabra clave **typedef.** Por ejemplo:

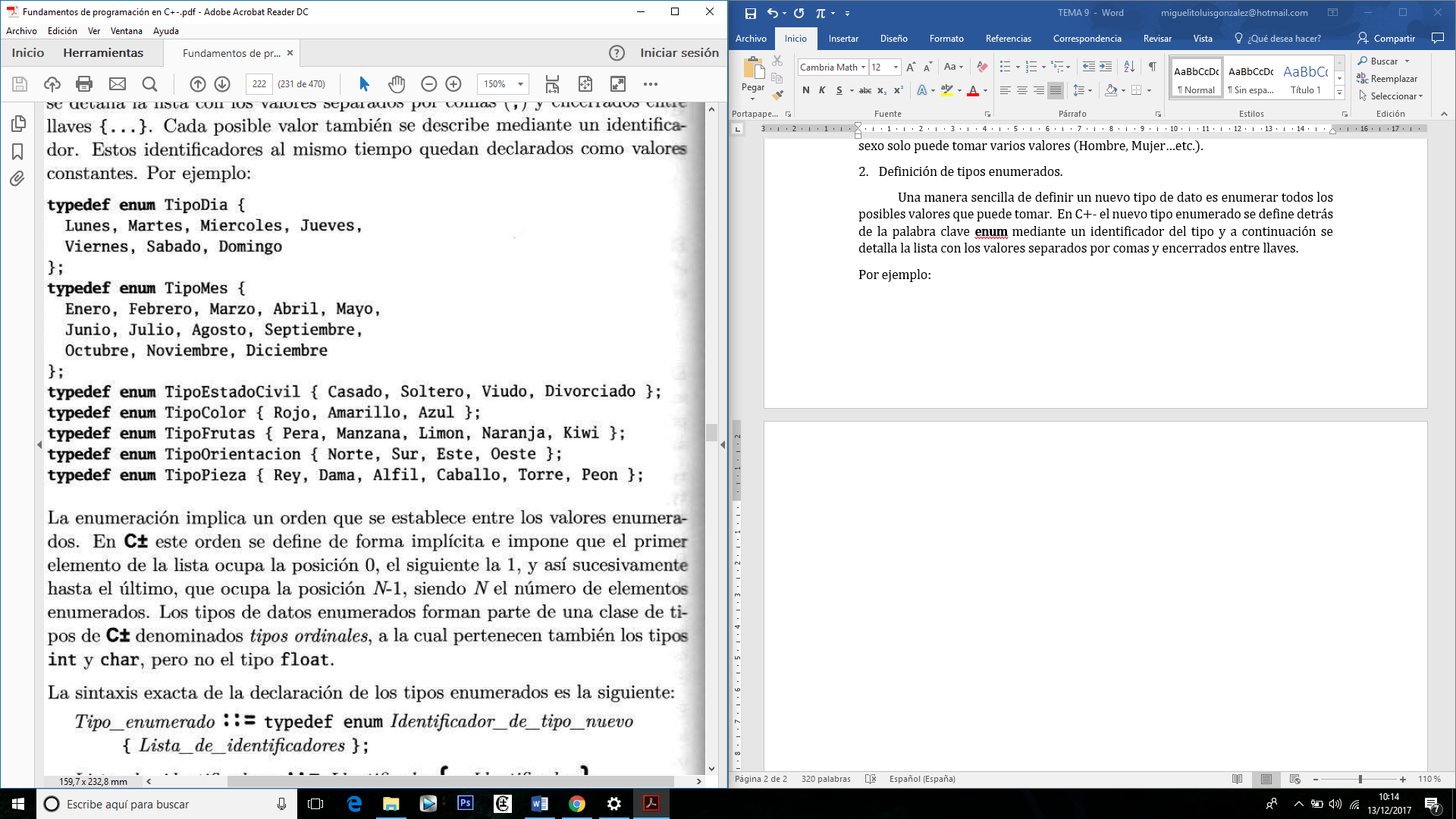
* **typedef int** TipoEdad; 🡪 typedef significa “Voy a declarar un nuevo tipo dato diseñado por mi”
* **typedef chat** TipoSexo;
* **typedef float** TipoAltura;

Esta declaración de tipos no establece ninguna especificación, como que la edad no puede ser un número negativo, ni mayor que un cierto número, o que el sexo solo puede tomar varios valores (Hombre, Mujer…etc.).

1. **Definición y uso de tipos enumerados.**

Una manera sencilla de definir un nuevo tipo de dato es enumerar todos los posibles valores que puede tomar. En C+- el nuevo tipo enumerado se define detrás de la palabra clave **enum** mediante un identificador del tipo y a continuación se detalla la lista con los valores separados por comas y encerrados entre llaves.

Por ejemplo:



ESTOS SON LOS ÚNICOS VALORES QUE PUEDE TOMAR EL DATO “TipoDia”

La enumeración implica un orden que se establece entre los valores enumerados. En c+- ese orden se define de forma implícita e impone que el primer elemento de la lista ocupa la posición 0, el siguiente la 1, y así sucesivamente. Los tipos de datos enumerados forman parte de una clase de tipos de C+- denominados tipos ordinales, a la cual también pertenecen los tipos int y char, pero no el tipo float.

USO DE TIPOS ENUMERADOS

Los tipos enumerados se emplean de manera similar a los tipos predefinidos. El identificador de tipo se puede emplear para definir variables de ese tipo, y los identificadores de los valores enumerados se emplean como las constantes con nombre. Usando las definiciones anteriores podemos escribir:

**TipoDia** diaSemana;

**TipoColor** colorCoche = Rojo; (También podemos inicializar un variable dándole un valor directamente)

**TipoMes** mes;

diaSemana = Lunes;

coloCoche = azul;

mes = Marzo;

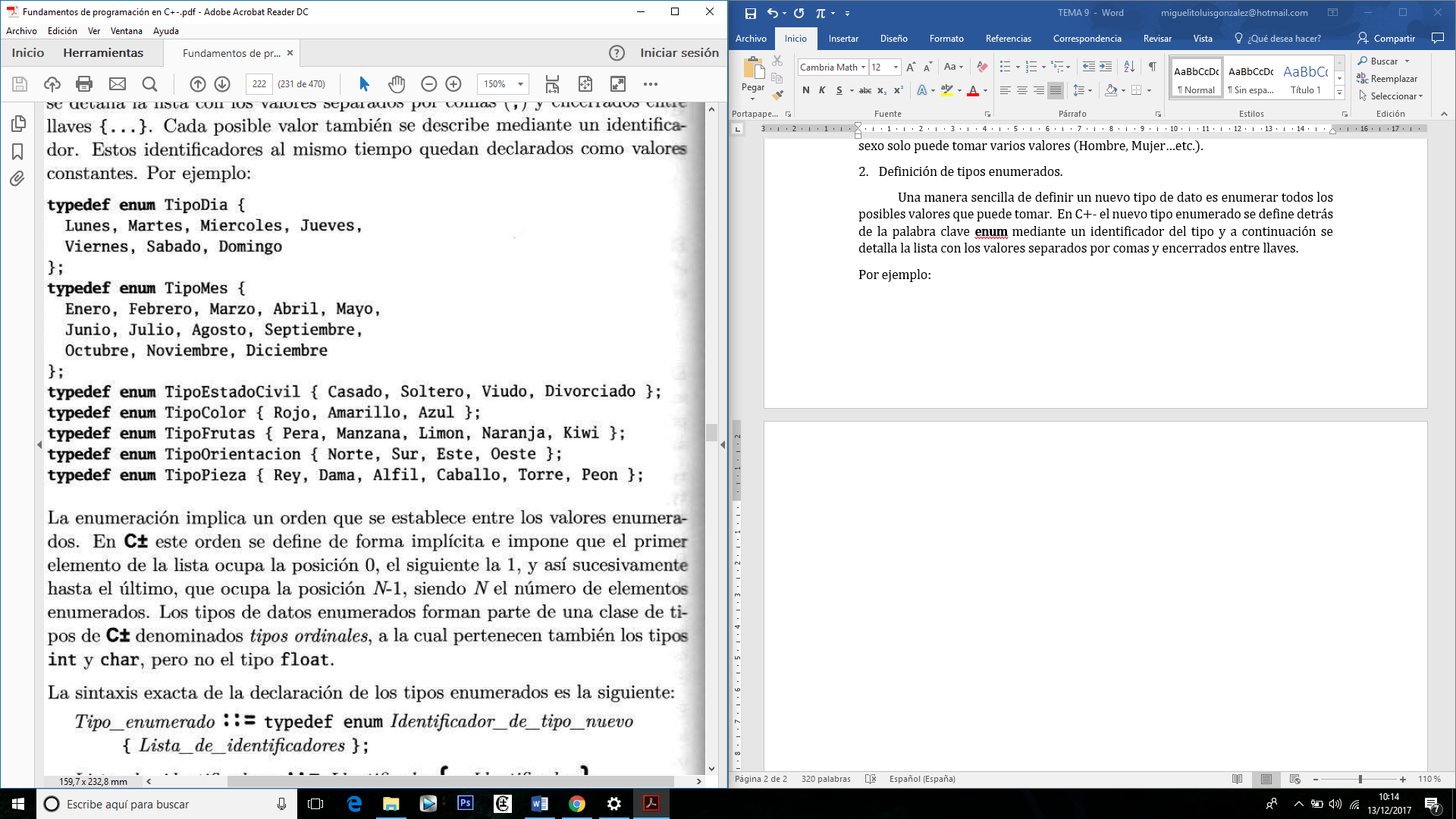
Puesto que los valores enumerados tienen un valor numérico implícito (primer elemento enumerado posee el valor numérico 0, segundo 1… hasta N-1 siendo N el número de elementos enumerados), podremos emplear con ellos los operadores de comparación para escribir sentencias del tipo:

if(mes >= Julio){…}

while (diaSemana < Sabado){…}

if (colorCoche = Rojo) {…}

Al igual que para el resto de los tipos de datos ordinales (char, float y tipos enumerados) se puede utilizar la notación **int(e)** para obtener la posición de un valor en la lista de valores del tipo. Por ejemplo, se cumple que:



**int (Lunes) == 0;**

**int (Martes) == 1;**

**int (Domingo) == 6;**

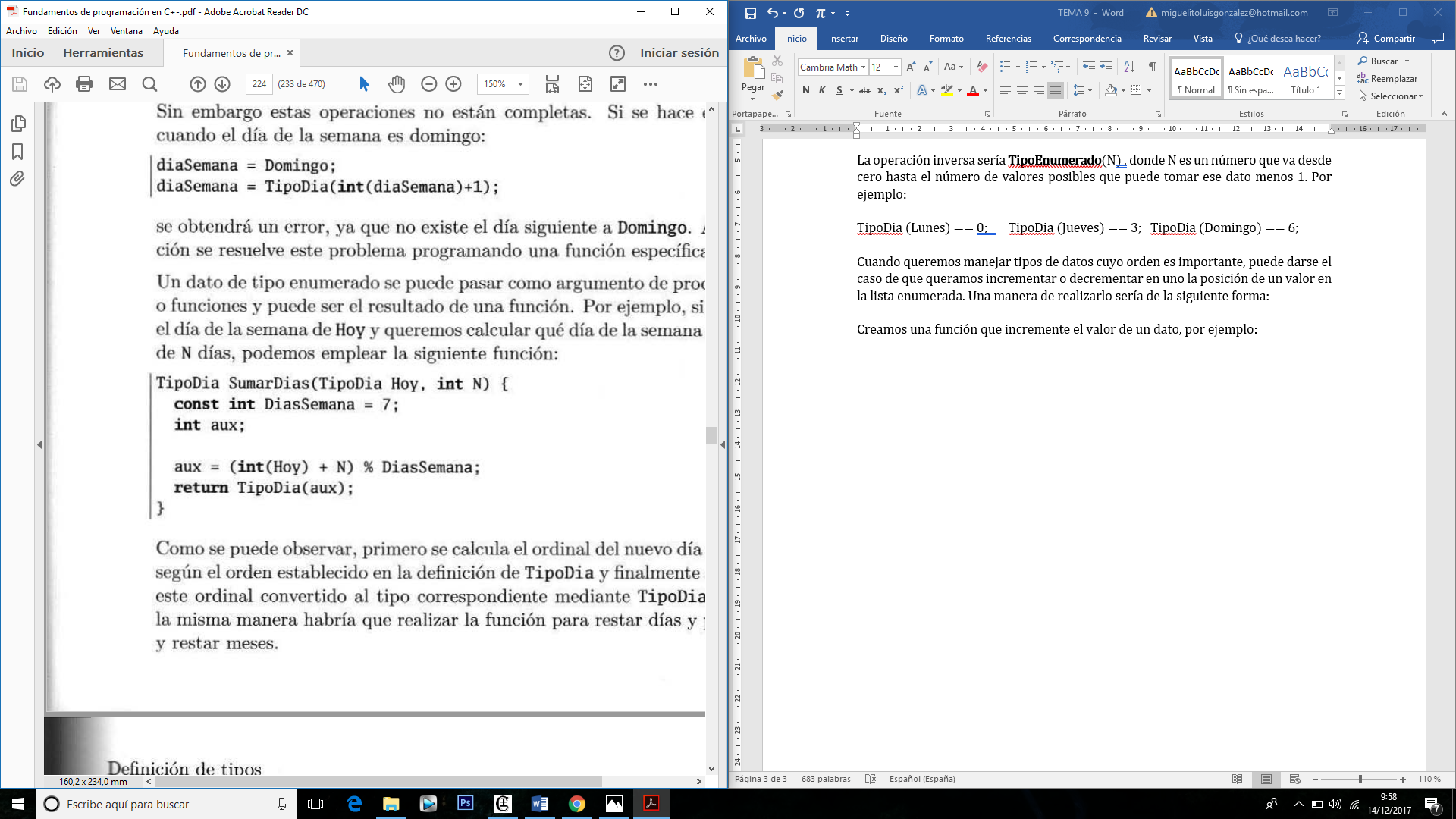
La operación inversa sería **TipoEnumerado**(N), donde N es un número que va desde cero hasta el número de valores posibles que puede tomar ese dato menos 1.

Por ejemplo:

TipoDia (Lunes) == 0; TipoDia (Jueves) == 3; TipoDia (Domingo) == 6;

Cuando queremos manejar tipos de datos cuyo orden es importante, puede darse el caso de que queramos incrementar o decrementar en uno la posición de un valor en la lista enumerada. Una manera de realizarlo sería de la siguiente forma:

Creamos una función que incremente el valor N dias del dato TipoDias, por ejemplo:

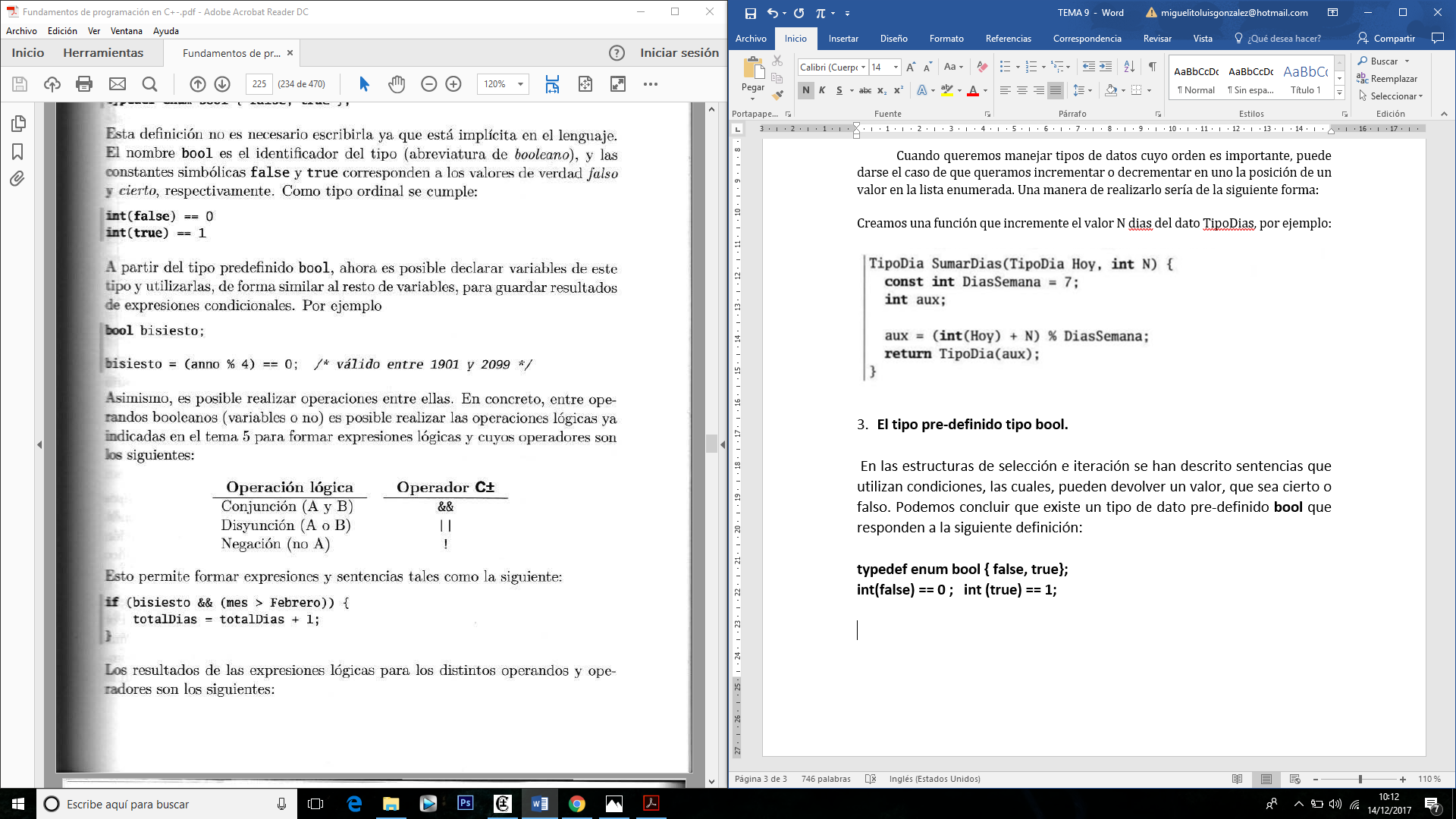


1. **El tipo pre-definido tipo bool.**

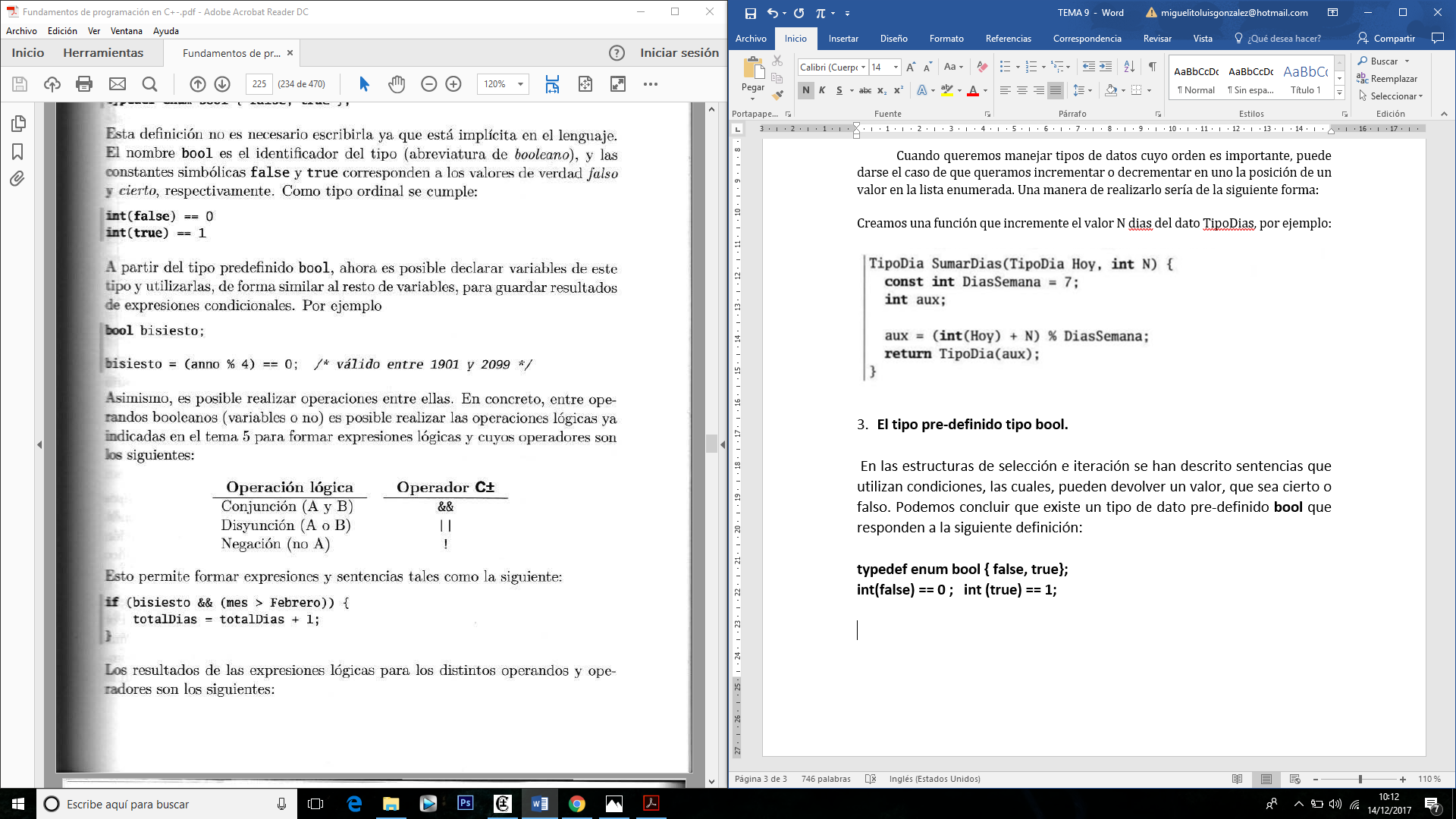
En las estructuras de selección e iteración se han descrito sentencias que utilizan condiciones, las cuales, pueden devolver un valor, que sea cierto o falso. Podemos concluir que existe un tipo de dato pre-definido **bool** que responden a la siguiente definición:

**typedef enum bool** { false, true};

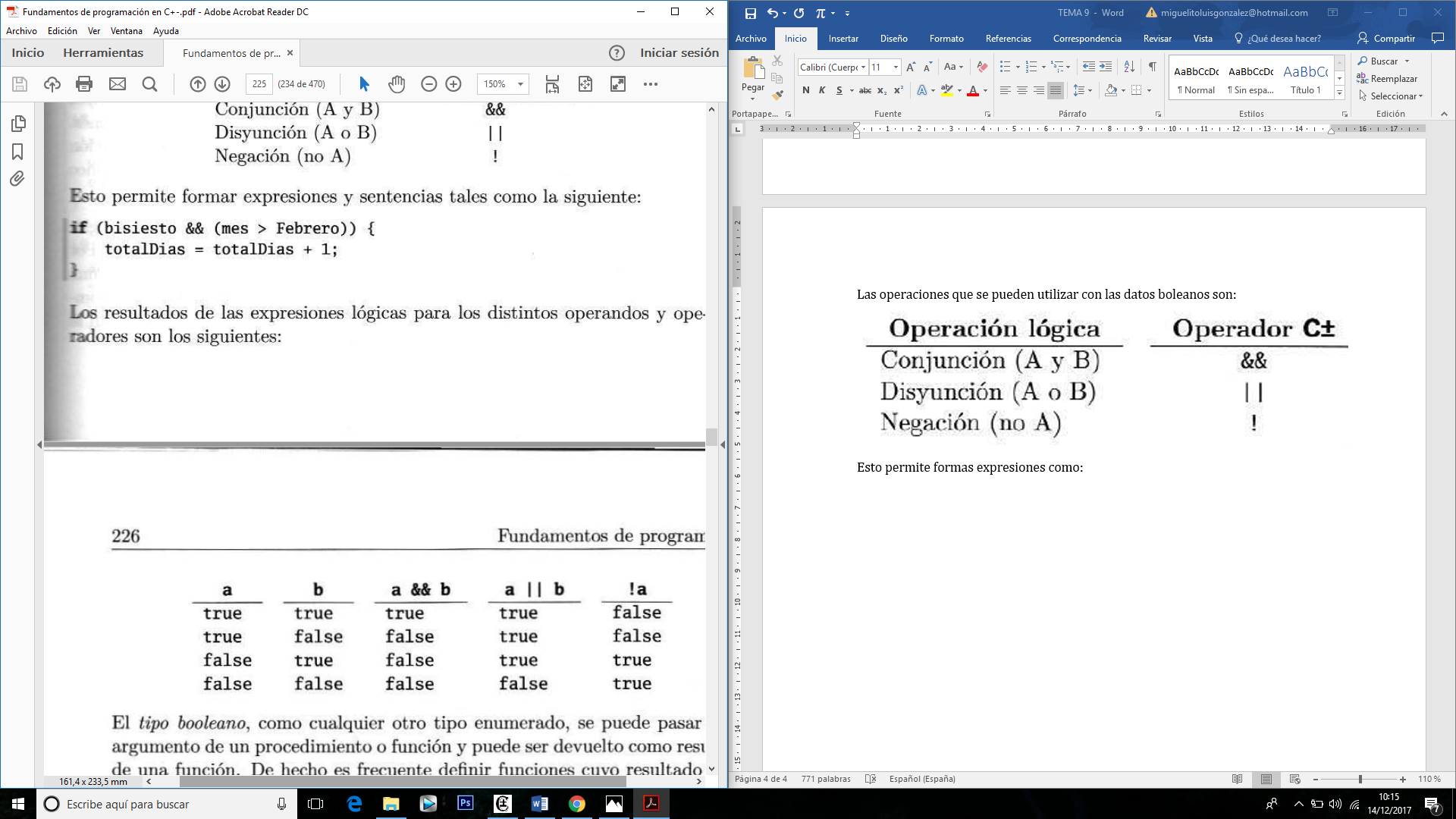
**int**(false) == 0 ; int (true) == 1;

Una aplicación de este tipo de dato puede ser: 

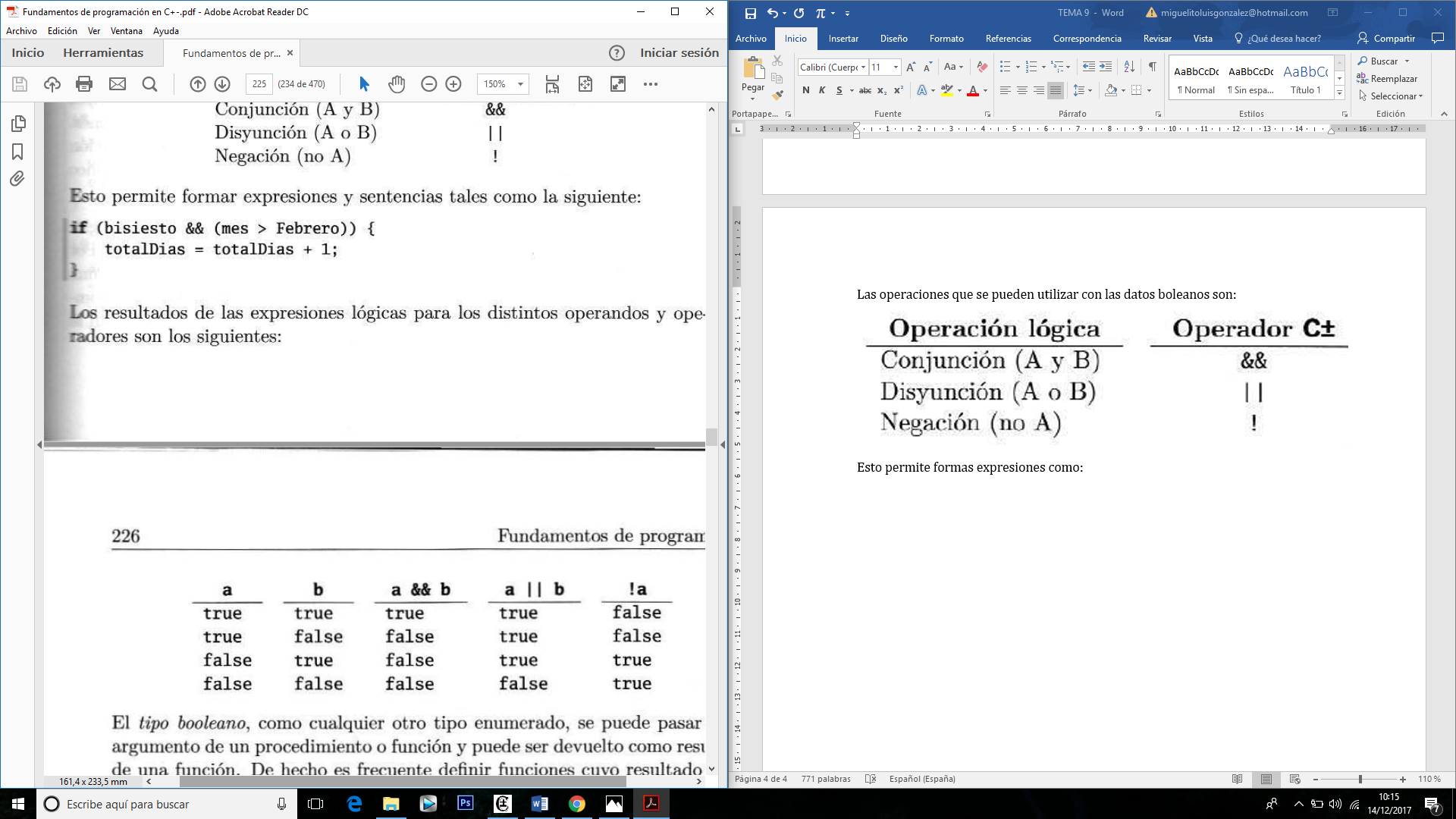
Las operaciones que se pueden utilizar con las datos boleanos son:



Esto permite formas expresiones como:



Los resultados de las expresiones lógicas con los distintos operandos son:



El tipo de funciones que devuelven un valor booleano reciben el nombre de “predicado”.

1. **Tipos estructurados.**

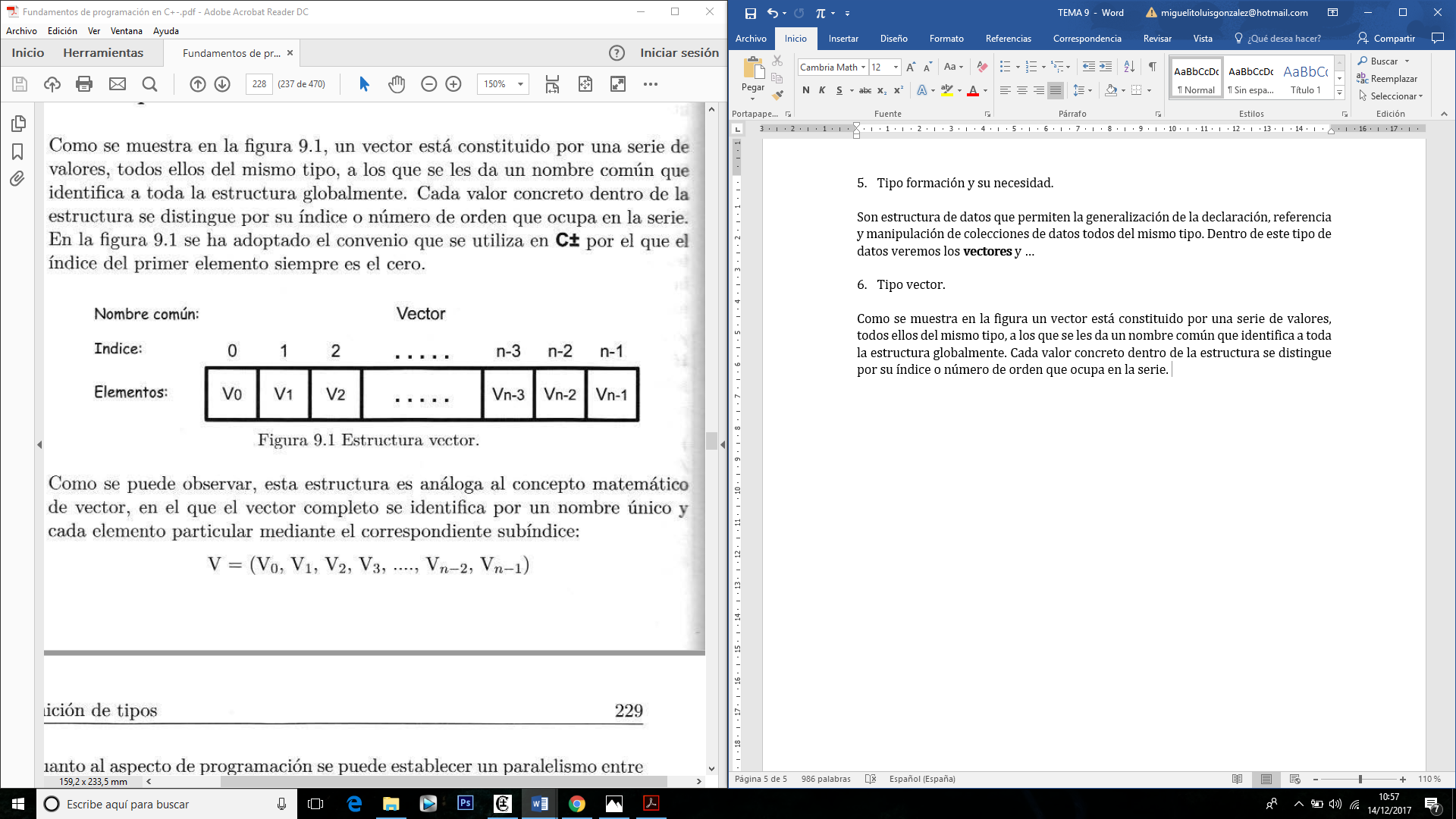
Todos los tipos de datos presentados hasta este momento se denominan tipos escalares, y son datos simples, en el sentido de que no se pueden descomponer. En muchas aplicaciones resulta conveniente manejar globalmente elementos de información que agrupan colecciones de datos. Por ejemplo, se puede manejar como dato único una fecha que está compuesta por día, mes y año. Un **tipo estructurado** de datos o estructura de datos, es un tipo cuyos valores se construyen agrupando datos de otros tipos más sencillos. Cada elemento simple se denomina componentes.

1. **Tipo formación y su necesidad.**

Son estructura de datos que permiten la generalización de la declaración, referencia y manipulación de colecciones de datos todos del mismo tipo. Dentro de este tipo de datos veremos los **vectores, vectores de caracteres** y **los registros.**

1. **Tipo vector.**

Como se muestra en la figura un vector está constituido por una serie de valores, todos ellos del mismo tipo, a los que se les da un nombre común que identifica a toda la estructura globalmente. Cada valor concreto dentro de la estructura se distingue por su índice o número de orden que ocupa en la serie.



## Declaración de vectores en BFN

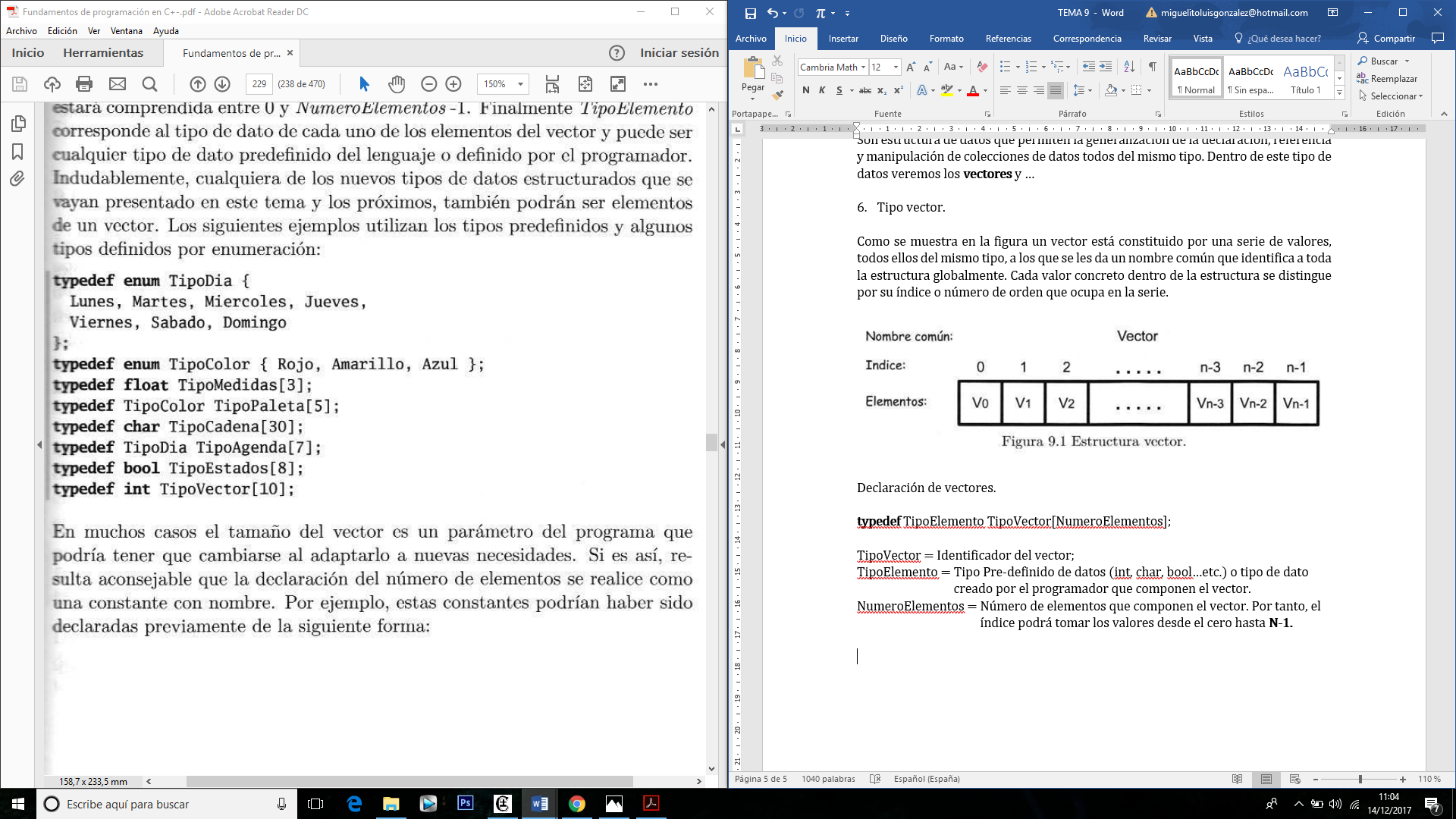
**typedef** TipoElemento TipoVector[NumeroElementos];

TipoVector = Identificador del vector;

TipoElemento = Tipo Pre-definido de datos (int, char, bool…etc.) o tipo de dato creado por el programador que componen el vector.

NumeroElementos = Número de elementos que componen el vector. Por tanto, el índice podrá tomar los valores desde el cero hasta **N-1.**

Un ejemplo de la creación de vectores es la siguiente:



DECLARACIÓN

Hayque diferenciar entre **inicializar una variable** y **declarar** una variable. La Declaración de tipos de datos(enum, vector, struct…etc) se realiza antes de la escritura de sub-programas y programa principal, en cuanto a la inicialización de vectores para dar valores a cada una de las posiciones dentro del vector, basta con ponerlas entre corchetes separadas por comas. Si se quiere crear un vector de caracteres, cada carácter esta entre puntillas(‘c’) y si se trata de cadena de caracteres cada una está entre comillas (“Hola Mundo”).

INICIALIZACIÓN

TipoMedidas Longitud = {12.3, 20.7,0};

TipoColor ColoresPrimarios = {Rojo, Azul, Rojo, Rojo, Azul}

**OPERACIONES** **CON ELEMENTOS DE LOS VECTORES**

La mayoría de las operaciones que se pueden realizar con vectores hay que hacerlas elemento por elemento, La referencia a un dato concreto dentro del vector se realiza mediante la mención del vector inicializado y la posición o índice del dato, dentro del vector, que se quiera tratar. Por ejemplo:

**typedef int** vector[5];🡪 Declaro un vector denominado “mivector” con 5 elementos.

vector mivector = {1,2,3,4,5} 🡪 Inicializo el vector dándole los valores que quiera.

mivector[1]== 2 🡪 la posición 1 de mi vector es igual a 2.

mivector[0]== 1🡪 La posición 0 de mi vector es igual a 1.

mivector[1] = mivector[2]+mivector[3]; 🡪 El elemento de la posición 1 será igual a la suma de los elementos de la posición 2 y 3.

mivector[3] = 3\*mivector[0] + 2\*mivector[0];

NOTA: Hay que tener en cuenta el tipo de elemento que se trata dentro del vector, por ejemplo, si tenemos un vector de tipo enteros y otro de tipo decimales, no podemos hacer operaciones aritméticas entre elementos de los distintos vectores, a no ser que se haga un cambio de tipo de datos previamente.

También se pueden manejar los índices de un vector mediante variables, por ejemplo:

int i,j;

i = 3;

j = 5;

vector[i] = vector1[j+i]+vector[j\*2];

Si el vector tiene “N” elementos el índice solo puede tomar valor desde cero hasta N-1.

**OPERACIONES GLOBALES CON VECTORES**

Con operaciones globales con vectores se refiere a hacer operaciones entre distintos vectores. Por ejemplo, para copiar un vector a otro vector, en C+- debemos usar un bucle for que recorra todos los elementos del primer vector y los copie en el vector 2.

for (int i = 0; i < N\_elementos; i++) {

vectorDos[i] = vectorUno[i];

}

**PASO DE ARGUMENTOS DE TIPO DE VECTOR**

También podemos utilizar los vectores como argumentos de procedimientos o funciones. Hay que tener en cuenta que si enviamos un vector a un procedimiento sin especificar que no puede ser alterado, este sub-programa podría cambiar el valor de los datos dentro del vector. Para especificar que no se quiere cambiar ningún dato dentro del vector, basta con poner un **const** en los argumentos del vector, en la interfaz del sub-programa.

**typedef int TipoVector[5]; “Declaramos vector con 5 elementos”**

**void LeerVector (TipoVector v){...}🡪 El vector v modificara el vector real, que se envía desde el programa principal.**

**void LeerVector (const TipoVector v){...}🡪 El vector v no modificara el vector real.**

**VECTOR DE CARACTERES: CADENA (*String)***

En C+- cualquier vector cuya declaración sea “**typedef char** Nombre[N]” será considerado una cadena o *String.* Una cadena de caracteres es un vector en el que se pueden almacenar textos de diferentes longitudes. Para distinguir la longitud útil en cada momento se reserva siempre espacio para un carácter más, y se hace que toda cadena termine con un carácter nulo ‘\0’ situado al final.

**DECLARACIÓN**

**typedef char string[21]🡪 Declaro un “string” con 21 espacios, uno de ellos reservado para el carácter nulo (‘\0’).**

**INCIALIZACIÓN**

**String MyString;**

**MyString = “Me llamo Juan”; Si me paso de 21 caracteres se produciría un error.**

C+- dispone de una librería **String** (<string.h>) que facilita el manejo de cadenas. Las funciones más comunes que la componen son:

|  |  |
| --- | --- |
| strcpy(c1,c2) | Copia c2 en c1 |
| strcat (c1, c2) | Concatena c2 a continuación de c1. |
| Strlen (c1) | Devuelve la longitud de c1 |
| strcmp (c1, c2) | * Devuelve cero si c1 y c2 son iguales. * Menor que cero si c1 precede a c2 en orden alfabético. * Mayor que cero si c2 precede a c1 en orden alfabético. |

No es posible realizar asignaciones globales entre cadenas:

**Typedef char** string[20];

String apellido, nombre;

**apellido = “Gonzalez”; strcpy(apellido, “gonzalez”,);**

**nombre = apellido; strcpy(nombre,apellido);**

También se pueden usar las cadenas en procedimientos como en las funciones de la librería stdio.h “printf” y “scanf”.

En la función **printf** podemos realizar la escritura de cadenas de caracteres como los tipos pre-definidos, poniendo %s estamos indicando que se imprimirá una cadena de caracteres. Por ejemplo:

**Typedef** char String[50];

String idioma = “inglés”;

String nombre, apellido;

String dirección = “Gran via 23”

**Printf(“**Idioma: %s Nombre: %s Apellido: %s**”, idioma, nombre, apellido)**

El procedimiento de escritura solo escribe el contenido de la variables hasta el primer carácter nulo ‘\0. ’

En la función **scanf** también podemos utilizar el mismo procedimiento con %s, por ejemplo:

**Typedef** char String[50];

# Printf (“Nombre y apellido?”);

scanf (“%s%s”,&Nombre, &apellido);

1. **Tipo Tupla y su necesidad**

Otra forma de construir un dato estructurado consiste en agrupar elementos de información usando el esquema de tupla o agregado. En este esquema el dato estructurado esta formado por una colección de componentes, cada uno de los cuales puede ser de un tipo diferente.

**Tupla:** Colección de elementos componentes, de diferentes tipos, cada uno de

los cuales se identifica por un nombre. El orden de los componentes de una tupla es hasta cierto punto arbitrario, para localizar el dato no se tiene en cuenta su posición sino el identificador dentro de la tupla que el programador le ha designado.

**TIPO REGISTRO**

Las tuplas pueden usarse en programa de C+- definiéndolos como estructuras de tipo registro o **struct,** el cual, es una colección de elementos de información llamados campos.

**DECLARACIÓN DE REGISTROS**

**typedef struct** Identificador {

Tipo-Campo1 nombre-campo-1;

Tipo-Campo2 nombre-campo-2;

...........

Tipo-CampoN nombre-campo-N;

}

Recordemos que normalmente la declaración no debe ir ligada a la inicializar.

**typedef enum** {

Enero, Febrero, Marzo, Abril, ...,Diciembre;

}

**typedef struct** TipoFecha {

**int** dia;

TipoMes mes;

**int** anno;

}

TipoFecha ayer, hoy;

Otra manera:

TipoFecha ayer = {12, Marzo, 2009};

TipoFecha hoy = {13, Marzo, 2009};

Se puede realizar la copia de un registro a otro, siempre y cuando tenga el mismo número de campos y cada campo sea del mismo tipo comparándolo con los campos del otro registro.

Ayer = hoy ;🡪 Copiar en “Ayer” la fecha contenida en “hoy”

También se pueden enviar argumentos de tipo registro a otros sub-programas

**void** LeerFecha (TipoFecha &Fecha){

…

}

Normalmente las operaciones con datos dentro de la estructura se realizan mediante el uso de cada uno de los campos por separado. Mencionar un campo concreto se realiza mediante la siguiente manera.

Nombre-Registro.Nombre-Campo

Cabe destacar que se pueden enviar únicamente un campo a otras funciones siempre y cuando este campo coincida en tipo de Dato con el de los argumentos formales del sub-programa.