UNIVERSIDAD DE MALAGA
DPTO. DE LENGUAJES Y C. DE LA COMPUTACION
E.T.S. DE INGENIERIA INFORMATICA

FUNDAMENTOS DE LA PROGRAMACIÓN

TEMA IV

TIPOS DE DATOS ESTRUCTURADOS

En el tema 2 de esta asignatura Fundamentos de la Programación hemos estudiado los tipos de datos simples en C++. En este tema abordaremos los tipos de datos estructurados, es decir, aquellos que se definen como composición de otros tipos de datos ya definidos, ya sean simples o estructurados. En C++ los tipos estructurados son el registro, el tipo *array* y la cadena de caracteres.

IV.1. Registros

Un registro es una colección de elementos que pueden ser de tipos distintos. O lo que es lo mismo, en la estructura registro no existe un único tipo base. A cada componente de la estructura se le llama campo, y podrá ser de cualquier tipo (simple o estructurado).

IV.1.1 Tipo de Datos *struct*

Para definir un tipo registro se utilizará la palabra reservada ***struct*** seguida del identificador del tipo y de la relación de componentes entre llaves, terminada en ';'. Para cada componente se indica el tipo e identificador del componente terminado en ';'. Ejemplo:

```
// tipo enumerado
enum Tmes {
      enero, febrero, marzo, abril, mayo, junio, Julio,
      agosto, sepiembre, octubre, noviembre, diciembre
};

// tipo registro
struct TFecha {
      unsigned dia;
      Tmes mes;
      unsigned agno;
};
```

Una vez definido un tipo registro se pueden declarar variables y constantes de dicho tipo:

```
const TFecha HOY={16, diciembre, 2010}
TFecha f_nac,f_ant;
```

Los identificadores dia, mes y agno representan los nombres de los elementos componentes del registro TFecha, denominados campos. Los nombres de los campos de un registro son locales a él, por lo que no hay conflicto con otros nombres en el programa.

Una referencia a un campo de un registro consiste en el nombre de la variable (o constante) registro y el nombre del campo, separados por un punto.

Se puede acceder a un componente concreto del registro de forma directa, de forma que con un campo de un registro se podrá hacer cualquier operación admisible para el tipo de dicho campo, como la operación de asignación:

```
f_nac.dia = 5 ;
f_nac.mes = enero;
f_nac.agno = 2000;
```

Con respecto a operaciones con registros completos, es posible asignar registros completos si son del mismo tipo:

```
f_ant = f_nac;
```

Así como inicializar una variable o una constante de tipo registro:

```
TFecha f_nac = { 24, marzo, 1984 };
```

Pero sin embargo, C++ no permite realizar comparaciones de igualdad o desigualdad entre registros:

```
if (f_ant == f_nac ) { ... } // ERROR
if (f_ant != f_nac ) { ... } // ERROR
if (f_ant <  f_nac ) { ... } // ERROR
```

Tampoco permite la lectura y escritura directa sobre dispositivos de E/S estándares. Este tipo de operaciones debe realizarse componente a componente.

Los campos de un registro pueden ser de cualquier tipo, simple o estructurado:

```
enum TMes {
            enero, febrero, marzo, abril, mayo, junio, julio, agosto,
            septiembre, octubre, noviembre, diciembre
      };
struct TFecha{
      unsigned dia;
      TMes mes;
      unsigned agno;
      };
struct TEmpleado{
      unsigned codigo;
      float  sueldo;
      TFecha fecha_ingreso; // un registro dentro de otro
      };
```

IV.1.2 Registros como Parámetros

Un registro se puede pasar tanto ***por valor*** como ***por referencia*** *a un subprograma*. Sin embargo, y por motivos de eficiencia, el paso de registros no se suele hacer ***por valor***, ya que es una operación muy costosa tanto en tiempo de CPU como en consumo de memoria, por lo que siempre se suelen pasar ***por referencia***, y en el caso de ser un parámetro de entrada, se pasa ***por referencia constante*** para evitar que pueda ser modificado y que se siga cumpliendo que es un dato exclusivamente de entrada. Ejemplo:

```
void escribir_fecha(const TFecha& fech)
```

```
{
    // escritura del registro campo a campo
}
```

Un registro también puede ser el valor devuelto por una función. Pero por las mismas razones de eficiencia expuestas anteriormente respecto al paso de parámetros, es conveniente que el valor devuelto por una función no sea de un tipo estructurado. En ese caso, será preferible construir un procedimiento y que el valor se devuelva como parámetro de salida mediante paso ***por referencia***.

```
void leer_fecha(TFecha& fech)
{
    // lectura del registro campo a campo
}
```

En resumen, las operaciones que se pueden realizar sobre registros completos son la asignación, la inicialización de una variable o constante de tipo registro, y el paso como parámetro a subprogramas, bien por referencia si el parámetro es de salida o entrada/salida o por referencia constante, caso de ser parámetro de entrada

IV.1.3 Ejemplo con registros

Suponemos definida la siguiente estructura registro que representa un instante de tiempo:

```
struct TTiempo{
      unsigned horas,minutos,segundos;

};
```

Ejemplo: *Función para calcular los segundos transcurridos entre dos instantes de tiempo dados*

```
unsigned convsegundos(const TTiempo& t)
{
      return t.horas*3600 + t.minutos*60 + t.segundos;
}

unsigned calcsegundos(const TTiempo& t1, const TTiempo& t2)
{
      return convsegundos(t1)-convsegundos(t2);

}
```

Ejemplo: *Procedimiento para convertir un tiempo expresado en segundos a formato de horas, minutos, segundos.*

```
void convtiempo(unsigned seg, TTiempo& t)
      {
                t.horas= seg /3600;
                seg= seg % 3600;
                t.minutos  = seg /60;
                t.segundos= seg % 60;
      }
```

IV.2. ARRAYS

El *array* es un tipo de datos estructurado formado por un número fijo de elementos del mismo tipo. El tipo de los elementos del *array* se llama tipo *base* y puede ser cualquier tipo. Una estructura *array* se caracteriza porque se puede acceder de manera directa a sus elementos, a través de los valores de un tipo denominado *índice*. El tipo índice debe ser un ordinal de cardinalidad finita. El tipo *base* puede ser tanto simple como estructurado. En resumen, un *array* es una colección finita de datos homogéneos y ordenados:

Homogéneos: Todos los elementos que hay dentro de un *array* son del mismo tipo (tipo base)

Finita: El número de elementos de un *array* es finito

Tamaño fijo (Dimensión): El número de elementos de un *array* se establece en su definición y se conoce como dimensión del *array*. Esta dimensión establece también los valores del tipo *índice* del *array* (valores desde 0 hasta dimensión).

Ordenados: Los elementos de un *array* mantienen una posición concreta en función de los valores del tipo índice del *array*.

IV.2.1 Tipo de Datos *array*.

Para definir un tipo *array* se usa la palabra reservada **`typedef`** seguida por el tipo de los elementos (tipo *base*), el identificador del nuevo tipo *array*, y entre corchetes el número de elementos que lo componen (dimensión).

```
const int MAX CARACTERES = 30;
typedef char TNombre[MAX CARACTERES]; // Tipo Array de 30 char
typedef int TVector[10];              // Tipo Array de 10 int

TVector v1;       // variable de tipo Vector
TNombre nom;      // variable de tipo Nombre
```

Para acceder a cada elemento de una variable declarada de tipo *array* se usa su nombre seguido por una expresión entre corchetes (índice). El valor de dicha expresión indicará la posición en el *array* del elemento al que se quiere acceder.

El primer elemento de un *array* ocupa la posición 0 (`v1[0]` y `nom[0]` en el ejemplo), y el último elemento ocupa la posición dimensión–1 (`v1[9]` y `nom[MAX_CARACTERES-1]` en el ejemplo). Así mismo, es posible que el índice de acceso al elemento sea una expresión: `nom[i]`, `nom[i+1]`, `v1[i*2]`, etc.

Como en todo tipo estructurado, no está permitida la lectura y escritura directa sobre dispositivos de E/S estándares. Este tipo de operaciones debe realizarse elemento a elemento.

A diferencia de los registros, C++ no permite la asignación de *array*s completos, aunque sí es posible darles un valor inicial:

```
const TVector V1 = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
TVector v2 = { 4, 5, 6 }; // el resto tomara el valor 0
TVector v3; // valores indefinidos

v2 = V1;    // ERROR. Asignacion de arrays no permitida
```

C++ tampoco permite la comparación de *array*s, por lo tanto, dicha operación se deberá realizar elemento a elemento.

IV.2.2 Arrays como Parámetros

C++ no permite el paso de *array*s por valor. De hecho, aunque se intente pasar por valor se pasará automáticamente por referencia. Por lo tanto, nosotros siempre pasaremos los *array*s por referencia. En caso de parámetros de entrada utilizaremos el ***paso por referencia constante***, y en caso de parámetros de salida o de entrada/salida utilizaremos el ***paso por referencia***. Ejemplo:

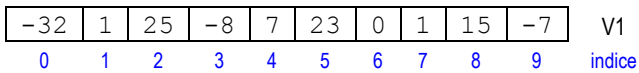
```
const int TAMANO = 20;
typedef int TVector[TAMANO];

void copiar(TVector& dest, const TVector& orig)
//copia orig en dest sin modificar origen
{
    for (int i = 0; i < TAMANO; ++i) {
        dest[i] = orig[i];
    }
}
```

C++ no permite que el valor devuelto por una función sea de un tipo *array*. En ese caso, se diseñará un procedimiento y el array el valor se devolverá como parámetro de salida mediante paso por referencia.

IV.2.2.1 Ejemplo con arrays unidimensionales

Crearemos un *array* unidimensional o vector, capaz de almacenar 10 enteros (tipo base *int*), y leeremos y escribiremos por pantalla su contenido.




```
#include <iostream>
```

```
using namespace std;
typedef int TVector[10];      // indexado desde la posición 0 a la 9

void LeerVector(TVector& v)
// lee valores para cada componente del vector
{
      for (unsigned i=0; i<10;i++){
            cout<<"Introduzca v["<<i<<"]: ";
            cin>>v[i];
      }
}
void ImprimirVector(const TVector& v)
// muestra por pantalla los valores del vector
{
      for (unsigned i=0; i<10;i++){
            cout<<"v["<<i<<"]: ";
            cout<<v[i]<<endl;;
      }
}
int main() {
      TVector v1;
      LeerVector(v1);
      ImprimirVector(v1);
      return 0;

}
```

IV.2.3 Problemas comunes al trabajar con arrays:

Inicialización: Las variables de tipo *array*, al igual que cualquier variable, deben ser inicializadas, bien mediante lectura por teclado de cada una de sus componentes o bien mediante asignaciones componente a componente.

Índice fuera de rango: Siempre hay que comprobar que se accede a una posición válida dentro del *array*. El lenguaje de programación C++ no comprueba que el valor del índice se encuentre dentro del rango válido para acceder a un elemento de un *array*. En caso de acceso (tanto para consultar su valor, como para modificarlo) fuera del rango válido, pueden ocurrir errores inesperados durante la ejecución del programa. De hecho, puede suceder que el error se manifieste en otro punto del programa, e incluso que no se manifieste visiblemente, y que permanezca oculto. Este hecho puede dar lugar a errores muy difíciles de detectar.

IV.2.4 Utilidad de los arrays

Consideremos el siguiente ejemplo: Una empresa tiene en plantilla 20 agentes de ventas (identificados por números del 1 al 20) que cobran comisión sobre la parte de sus operaciones comerciales que excede los 2/3 del promedio de ventas del grupo.

Se necesita un algoritmo que lea el valor de las operaciones comerciales de cada agente e imprima el número de identificación de aquellos que deban percibir comisión así como el valor correspondiente a sus ventas.

Este problema tiene 2 aspectos que juntos lo hacen difícil de programar con los mecanismos vistos hasta ahora:

Procesamiento similar sobre los datos de cada agente:

- leer ventas
- calcular promedio de ventas
- comparar niveles
- decidir si debe o no recibir comisión

Se necesita almacenar durante la ejecución del programa los valores de las ventas de cada agente: para el cálculo del promedio y para la comparación de niveles. Esto implica que necesitamos 20 variables para retener los valores de las ventas.

Un posible algoritmo sería:

```
#include <iostream>
using namespace std;
const float PORCION=2.0/3.0;
int main() {
	float ventas1,ventas2,ventas3, ventas4,...ventas20;
	//    ventas realizadas por cada agente
	float suma=0.0;   // acumulado de ventas
	float umbral;     // valor minimo para obtener comision

	cout<<"Introduzca las ventas del agente: "<<endl;
	cin>>ventas1;
	suma= suma+ventas1;
	cout<<"Introduzca las ventas del agente: "<<endl;
	cin>>ventas2;
	suma=suma+ventas2;
	cout<<"Introduzca las ventas del agente: "<<endl;
	cin>>ventas3;
	suma= suma+ventas3;
	cout<<"Introduzca las ventas del agente: "<<endl;
	cin>>ventas4;
	suma= suma+ventas4;
		...
	cout<<"Introduzca las ventas del agente: "<<endl;
	cin>>ventas20;
	suma= suma+ventas20;
	umbral= PORCION*(suma/20.0);
	if (ventas1>umbral)
	{
		cout<<"Ventas del Agente 1: "<<ventas1;
	}

	if (ventas2>umbral)
```

```
	{
			cout<<"Ventas del Agente 2: "<<ventas2;
	}
	if (ventas3>umbral)
	{
		cout<<"Ventas del Agente 3: "<<ventas3;
	}
	if (ventas4>umbral)
		{
			cout<<"Ventas del Agente 4: "<<ventas4;
		}
		...

	if (ventas20>umbral)
		{
			cout<<"Ventas del Agente 20: "<<ventas20;
		}
	return 0;
}
```

La escritura de este programa es, como se ve, bastante tediosa. El problema se agravaría más si en lugar de 20 hubiera 200 agentes de ventas.

Una mejor solución consiste en considerar estas variables de ventas como componentes de una estructura de datos *array*. Cada componente tendrá como índice asignado el número de identificación del correspondiente agente de ventas.

La definición del tipo *array* sería:

```
const unsigned NUMAGENTES=20;
typedef float TVentas[NUMAGENTES];//20 posiciones, indice desde 0..19
```

La variable de este tipo se declararía dentro del **main** como:

```
	TVentas ventas;
```

De forma que ventas[i] almacenará las ventas realizadas por el agente número i. Lo que va entre corchetes al referirse a un componente de un *array* puede ser cualquier expresión que al evaluarse dé como resultado un valor del tipo *indice* del *array*.

Con estos datos nuestro algoritmo podría escribirse:

```
#include <iostream>
using namespace std;
	const float PORCION=2.0/3.0;
	const unsigned NUMAGENTES=20;
	typedef float TVentas[NUMAGENTES];//20 posiciones, indice desde 0..19

	void leerVentas(TVentas& ventas)
	{
		for (unsigned i=0; i<NUMAGENTES;i++)
		{
			cout<<"Introduzca ventas del agente "<<i+1<<": ";
			cin>>ventas[i];
		}
```

```
}

float calcMedia(const TVentas& ventas)
{
	float suma=0.0;
	for (unsigned i=0; i<NUMAGENTES; i++)
	{
		suma= suma+ventas[i];
	}
	return suma/NUMAGENTES;
}

void imprimir(const TVentas& ventas, float umbral)
{
	for (unsigned i=0; i<NUMAGENTES; i++)
	{
		if (ventas[i]>umbral)
		{
			cout<<"Ventas del Agente "<<i+1<<" :"<<ventas[i];
		}
	}
}


int main() {
	TVentas ventas;
	float umbral;

	leerVentas(ventas);
	umbral= PORCION*calcMedia(ventas);
	imprimir(ventas,umbral);

	return 0;
}
```

IV.2.5 Arrays Multidimensionales

Al definir el tipo *array* intervienen siempre 2 tipos : el tipo *indice* y el tipo *base*. El tipo *indice* ha de ser un ordinal de cardinalidad finita, pero sobre el tipo *base* no existe restricción alguna. Al poder usarse cualquier tipo como tipo base de un *array*, esto incluye la posibilidad de usar tipos estructurados como tipo base de un *array*. De esta forma, en caso de que el tipo base de un *array* sea a su vez otro *array*, obtendremos *array*s multidimensionales.

Por ejemplo:

```
const unsigned Filas=5;
const unsigned Columnas=3;
typedef int TVectorFila[Columnas];
typedef TVectorFila TMatriz[Filas];  // matriz de 5 Vectores Fila
```

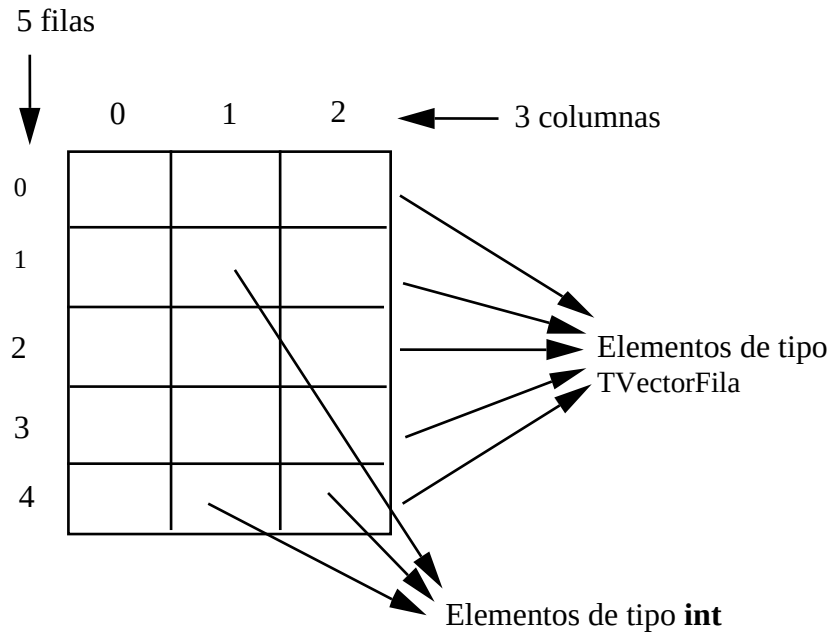
También es posible declararlo de la siguiente forma:

```
const unsigned Filas=5;
const unsigned Columnas=3;
typedef int TMatriz[Filas][Columnas];  // matriz 6x8
```

Si declaramos la variable

```
TMatriz a;
```

"a" será una estructura que puede contener 5 elementos de tipo `TVectorFila`, o lo que es lo mismo, 15 elementos de tipo **int**. Gráficamente:



A este tipo de estructuras se les denomina *Array*s bidimensionales. Los *array*s bidimensionales se utilizan a menudo para representar matrices matemáticas, de igual forma que un *array* (unidimensional) puede emplearse para representar un vector.

Podemos declarar una variable del tipo especificado anteriormente, y acceder a los componentes, tanto individualmente como por filas:

```
TMatriz a;

a[2]    // fila 2, es decir, un array de 3 elementos de tipo int
a[2][3]  // un elemento del tipo base int del array a, en concreto el
        //  entero almacenado en la fila 2, columna 3 del array
```

De igual forma, si hemos definido el *array* bidimensional TMatriz, podríamos construir un nuevo tipo *array* cuyo tipo base sea TMatriz, y obtendríamos un *array* de 3 dimensiones. Y así sucesivamente.

```
const unsigned Profundidad=3;
const unsigned Filas=5;
const unsigned Columnas=3;
typedef int TVectorFila[Columnas];
typedef TVectorFila TMatriz[Filas];  // array de 6 Vectores Fila
```

```
typedef TMatriz TCubo[Profundidad];
            //array con indice 3 y tipo base array bidimensional
```

IV.2.5.1 Ejemplo con Arrays Multidimensionales:

Vamos a construir un algoritmo que lee una matriz 6x8 de enteros (fila a fila), la almacena en un *array* bidimensional "a", deja los resultados de las sumas de los elementos de cada fila en un vector "b" y las sumas de los elementos de cada columna en un vector "c". Finalmente imprime los 3 *array*s con el formato siguiente:

```
a a a a a a a a b
a a a a a a a a b
a a a a a a a a b
a a a a a a a a b
a a a a a a a a b
a a a a a a a a b
c c c c c c c c
```

```
#include <iostream>
using namespace std;
const unsigned Filas=6;
const unsigned Columnas=8;
typedef int TVectorFila[Columnas];
typedef int TVectorColumna[Filas];
typedef TVectorFila TMatriz[Filas];  // array de 6 arrays TVectorFila


void leerMatriz(TMatriz& mat)
{

      for (unsigned fi=0; fi<Filas;fi++)
      {
            cout<<"Introduzca valores para fila "<<fi<<endl;
            for (unsigned co=0; co<Columnas; co++)
            {
                  cin>>mat[fi][co];
            }
      }
}

void escribirFila(const TVectorFila& fila)
{
      for (unsigned co=0; co<Columnas; co++)
      {
            cout<<fila[co]<<" ";
      }

}

int sumarFila(const TVectorFila& fila)
```

```
{
	int resultado=0;
	for (unsigned co=0; co<Columnas; co++)
	{
		resultado+= fila[co];
	}
	return resultado;
}


int sumarCol(const TMatriz& mat, unsigned co)
{
	int res=0;
	for (unsigned fi=0; fi<Filas; fi++)
	{
		res=res+ mat[fi][co];
	}
	return res;
}

int main() {
	TMatriz a;
	TVectorColumna b;
	TVectorFila c;
	leerMatriz(a);
	for (unsigned fi=0; fi<Filas; fi++)
	{
		b[fi]= sumarFila(a[fi]);
	}
	for (unsigned co=0; co<Columnas; co++)
	{
		c[co]= sumarCol(a,co);
	}
	for (unsigned fi=0; fi<Filas;fi++)
	{
		escribirFila(a[fi]);
		cout<<b[fi];
		cout<<endl;
	}
	escribirFila(c);

	return 0;
}
```

IV.2.6 Arrays Abiertos como Parámetros

Hay situaciones en las que deseamos implementar un subprograma que trabaje con *array*s, independientemente de su tamaño. De forma tal que el tamaño del *array* en el subprograma (parametro formal) se adecue al tamaño del *array* pasado durante la invocacion al mismo (parametro real). Para ello, el parámetro formal se declara especificando su tipo base, el identificador y el símbolo **`[]`** que indica que dicho parámetro es un *array* sin tamaño especificado, su tamaño dependerá del tamaño del parámetro real especificado en cada invocación al subprograma. El tipo base del parámetro real debe coincidir con el del parámetro formal. La información sobre el tamaño del *array* se pierde al pasarlo como *array* abierto, por lo que dicho tamaño se deberá también pasar como parámetro.

En caso de pasar *array*s multidimensionales como parámetros abiertos, sólo es posible poner la primera dimensión sin especificar.

El paso se realiza siempre por referencia sin necesidad de especificar el símbolo **&**, y para asegurar que no sea modificado en caso de información de entrada, se realizará el paso de parámetros ***constante***.

IV.2.6.1 Ejemplo de Arrays Abiertos

```
#include <iostream>
using namespace std;

const unsigned TAMANO_1 = 10;
const unsigned TAMANO_2 = 20;
typedef int Tvector_1[TAMANO_1];
typedef int Tvector_2[TAMANO_2];

void imprimir(int n,           // numero de elementos
              const int vct[]) // Parámetro de entrada: vector de enteros
                               // paso por referencia constante
{
    for (int i = 0; i < n; ++i) {
      cout<<" v["<< i <<"]: ";
        cout << vct[i] << " " ;
    }
    cout << endl;
}

void valores(int n,     // numero de elementos
             int vct[]) // vector de enteros (paso por referencia)
{
    for (int i = 0; i < n; ++i) {
      cout<<"Introduzca componente v["<<i<<"]: ";
        cin >> vct[i];
    }
}

int main()
{
    Tvector_1 vector_1;
    Tvector_2 vector_2;

    cout<<"Lectura de valores para el array 1: "<<endl;
    valores(TAMANO_1, vector_1);
    cout<<"Escritura de los valores del array 1: "<<endl;
    imprimir(TAMANO_1, vector_1);
    cout<<"Lectura de valores para el array 2: "<<endl;
    valores(TAMANO_2, vector_2);
    cout<<"Escritura de los valores del array 2: "<<endl;
    imprimir(TAMANO_2, vector_2);

    return 0;
}
```

Sólo es válido declarar *array*s abiertos como parámetros. No es posible declarar variables como *array*s abiertos.

IV.3. CADENAS DE CARACTERES.

En numerosos programas es frecuente encontrarnos con la necesidad de usar cadenas (o secuencia) de caracteres. Por ejemplo, si queremos implementar una agenda personal, ¿cómo representaríamos el nombre de las personas, su dirección, etc. ?

En general los lenguajes de programación (y en particular C++) presentan dos posibilidades: ofrecer un tipo Cadena como tipo predefinido en el lenguaje, o programar la idea de cadena como una parte más del problema, mediante una estructura *array* cuyo tipo base sea el tipo char.

En C++ el tratamiento de las cadenas de caracteres se puede realizar de estas dos formas diferentes: utilizando el tipo ***`string`*** proporcionado por la biblioteca estándar, o implementado sobre *array*s de caracteres al estilo "C" utilizando como carácter terminador el `'\0'`. En nuestros programas ***siempre*** utilizaremos el tipo ***`string`*** de la biblioteca estándar.

IV.3.1 El tipo de datos *string*

Un valor del **tipo predefinido** ***`string`*** será una secuencia de caracteres de **longitud indefinida** (**aunque limitada** por la implementación). Representaremos las literales constantes de tipo cadena entre comillas, por ejemplo:“Juan Mena”. Podremos definir tanto constantes como variables y parámetros.

En C++ el tratamiento de las cadenas de caracteres se realiza a través del tipo ***`string`*** proporcionado por la biblioteca estándar. Para ello deberemos incluir en nuestro programa las definiciones que de dicho tipo hace la biblioteca estándar mediante la siguiente sentencia al principio del programa:

```
#include <string>
```

Podremos definir tanto variables como constantes de dicho tipo de la siguiente forma:

```
const string nombre = "pepe garcia"; // string constante
string cadena;                       // variable cadena vacia
string otro = nombre;                // copia de nombre
```

Es posible realizar las siguientes operaciones con variables de tipo `string`:

- **Asignación**:

A una variable de tipo ***`string`*** se le puede asignar directamente tanto una cadena de caracteres (o carácter) constante como otra variable de tipo *string* (o de tipo carácter).

```
const string nombre = "jose";
```

```
string cadena;
string nmb;

cadena = nombre;     // asigna el valor de nombre a cadena
```

- **Concatenación** de dos cadenas (+):

En la concatenación, al menos uno de los dos primeros operandos debe estar declarado de tipo ***string***.

La operación de concatenación será imprescindible para añadir caracteres a una cadena porque no es posible acceder fuera del rango de la misma.

```
cadena += ' ';       // añade un espacio al final de cadena
cadena += "garcia";  // cadena valdra "jose garcia"
nmb = cadena + ' ' + "lopez"; // nmb = "jose garcia lopez"
nmb = "pepe" + "luis";        // ERROR
```

- **Comparación** lexicográfica:

Es posible comparar variables y constantes de tipo ***string*** mediante los operadores relaciones (== != > >= < <=) siempre y cuando al menos uno de los operandos esté declarado como de tipo ***string***:

```
if (nombre == "jose") { ... } if (nombre != "jose") { ... }
if (cadena > nombre)  { ... } if (nombre >= "jose") { ... }
if (cadena < nombre)  { ... } if (nombre <= "jose") { ... }
if ("pepe" == "pepe") { ... } // ERROR
```

También se puede utilizar la función **compare** que devuelve tres posibles valores:

```
int res;
res = cadena.compare(nombre);
    // si cadena < nombre  -> res <  0
    // si cadena == nombre -> res == 0
    // si cadena > nombre  -> res >  0
```

- **Longitud:**

Para saber el número de caracteres de un *string* se utiliza **size**

```
int n = cadena.size();
```

- **Acceso** al i-ésimo carácter de una cadena ([i]), donde i ∈ [0..nombre.size() - 1]

Se accede a los caracteres que la componen mediante indexación donde el primer elemento se encuentra en el índice `0`, y el último elemento en el índice `size()-1`:

```
string cadena = "pepe garcia";           // string constante
char primera_letra = cadena[0];          // toma valor 'p'
char ultima_letra = cadena[cadena.size()-1]; // toma valor 'a'
cadena[0] = 'P'; // pone el primer carácter de cadena a 'P'
```

- *NOTA IMPORTANTE: C++ no comprueba que el acceso a los elementos de una cadena mediante la indexación (vista anteriormente) se realiza dentro del rango correcto, por lo que un error en dicha indexación puede dar lugar a errores inesperados durante la ejecución del programa.*
- Obtener una **subcadena** a partir de una cadena ([i,t]).

$$i \in [0..cadena.size() - 1] \text{ y } t \in [0..cadena.size() - i]$$

Para obtener una subcadena a partir de una cadena dada se utiliza la operación **`substr`** de la siguiente forma. Dada una cadena `cad`, obtenemos una subcadena a partir de la posición `i` de la misma y con un tamaño `t` mediante:

```
cad.substr(i,t)
```

Por ejemplo:

```
string cadena = "Jose Antonio";
string seg_nombre;

seg_nombre = cadena.substr(5,7);  // obtenemos "Antonio"
      // subcadena de tamaño 7 a partir de la posición 5
```

- Realizar la **entrada** y **salida** de *string* mediante los operadores habituales:

```
cout << "Introduce nombre: ";
cin >> cadena; // salta espacios y lee hasta siguiente espacio
cout << " Nombre: " << cadena << endl;
```

El operador de entrada (>>) salta los espacios en blanco (y saltos de línea) hasta encontrar una cadena de caracteres, dicha cadena se leerá hasta encontrar un espacio en blanco o un salto de línea. Así mismo, tambien es posible leer una línea entera hasta el salto de línea, u otro delimitador que se especifique:

```
getline(cin, cadena);      // lee una linea completa
getline(cin, cadena, ';'); // lee hasta ';'
```

Una cuestión importante a tener en cuenta son las situaciones en las que se puede quedar la información en el buffer dando lugar a errores en la toma de datos.

La lectura de datos mediante flujos (>>) ignora los separadores existentes entre los diferentes elementos de entrada. Sin embargo, esto no es así para la operación `getline`**.** Esta operación lee del buffer de teclado la información que se encuentre sin diferenciar entre separadores y otros datos.

Por ejemplo, supongamos que tenemos la siguiente combinación de operaciones:

```
string nombre, apelidos;

cin >> nombre;
getline(cin, apellidos);
```

y la entrada para estas operaciones es:

```
Jose
Rodriguez
```

Si consultamos el contenido de las variables, `nombre` contendrá `Jose` pero `apellidos` contendrá la cadena vacía, pues tras haberse realizado la lectura de `Jose`**,** la operación `getline` (que toma los datos del buffer hasta que se encuentre un retorno de carro) lo que se encuentra en el buffer de teclado es el retorno de carro que el usuario pulsó tras introducir `Jose`.

La solución para ignorar esos separadores y realizar la toma de datos correcta es utilizar la operación `ignore`. Por ejemplo:

```
cin.ignore(100,'\n')
```

limpiará (extraerá) del buffer los siguientes 100 caracteres o bien hasta que se encuentre un retorno de carro. Así, el ejemplo anterior quedaría:

```
string nombre, appellidos;

cin >> nombre;
cin.ignore(100,'\n');
getline(cin, apellidos);
```

En este caso, la variable `nombre` contendrá `Jose` y la variable `apellidos` contendrá `Rodriguez`. Otra posibilidad será realizar una lectura previa sobre una variable con el único objetivo de eliminar todos los caracteres del buffer.

```
getline(cin, basura);
```

IV.3.2 *String*s como Parámetros

Respecto al paso de *string*s como parámetros, como se indicó anteriormente para el caso de registros, es posible pasarlos tanto ***por valor*** como ***por referencia***, sin embargo, y por motivos de eficiencia, el paso de tipos estructurados no se suele hacer ***por valor***, ya que es una operación muy costosa tanto en tiempo de CPU como en consumo de memoria, por lo que siempre se suelen pasar ***por referencia***, y en el caso de ser un parámetro de entrada, se pasa ***por referencia constante*** para evitar que pueda ser modificado.

De la misma forma que para los registros, es mejor ***que el valor devuelto por una función no sea de un tipo string.*** En ese caso, será preferible diseñar un procedimiento en el que el valor se devuelva como parámetro de salida mediante paso ***por referencia***.

IV.3.2.1 Ejemplo con string

Se desea leer una cadena de caracteres, y copiarla en orden inverso en otra cadena.

```
#include <iostream>
#include <string>
using namespace std;
void invertirCadena(const string& palabra, string& invertida)
{

	for (int i=palabra.size()-1;i>=0;i--)
	 {
		invertida= invertida+palabra[i];
	 }
}
int main()
{
	string entrada, invertida;
	cout<<"Escribir la palabra: ";
	cin>>entrada;

	invertirCadena(entrada, invertida);

	cout<<"La palabra "<<entrada<<" invertida es: "<<invertida<<endl;
	return 0;
}
```

IV.4. Resolución de Problemas usando Tipos Estructurados

Implementación de una Agenda personal: Se pretende diseñar un programa para la gestión de una 'agenda'. La información personal que será almacenada es la siguiente:

Nombre

Teléfono

Dirección

Calle

Número

Piso

Código Postal

Ciudad

Las operaciones a realizar con dicha agenda serán:

1. Añadir los datos de una persona
2. Acceder a los datos de una persona a partir de su nombre.

3. Borrar una persona a partir de su nombre.

4. Modificar los datos de una persona a partir de su nombre.

```
#include <iostream>
#include <string>
using namespace std;
unsigned const  MAXPERSONAS = 50;
int const NO_VALIDA    = -1;  //  Indica posición no valida del array
struct TDireccion
      {
         unsigned num;
         string calle;
         string piso;
         string cp;
         string ciudad;
      };

struct TPersona
      {
            string nombre;
            string tel;
        TDireccion direccion;
      };

typedef TPersona TPersonas[MAXPERSONAS]; / Array de 50 registros TPersona

struct TAgenda
{
      unsigned num_pers;  //número real de personas almacenada
      TPersonas pers;     //datos de dichas personas
};

void Inicializar(TAgenda& ag)
{
      ag.num_pers=0;    // agenda con 0 registros TPersona almacenados
}

 void LeerDireccion(TDireccion& dir)
 {
     cout<<"Introduzca calle: ";
     cin>>dir.calle;
     cout<<"Introduzca número: ";
     cin>>dir.num;
     cout<<"Introduzca piso: ";
     cin>>dir.piso;
     cout<<"Introduzca código postal: ";
     cin>>dir.cp;
     cout<<"Introduzca ciudad: ";
     cin>>dir.ciudad;
 }


 void LeerPersona(TPersona& per)
 {
      cout<<"Introduzca nombre: ";
      cin>>per.nombre;
      cout<<"Introduzca telefono: ";
      cin>>per.tel;
      LeerDireccion(per.direccion);
 }

  void EscribirDireccion(const TDireccion& dir)
```

```
  {
      cout<<dir.calle;
      cout<<dir.num;
      cout<<dir.ciudad;
  }

 void EscribirPersona(const TPersona& per)
 {
       cout<<"Nombre: "<<per.nombre;
       cout<<"Telefono: "<<per.tel;
       EscribirDireccion(per.direccion);
 }



int BuscarPersona(const string nombre, const TAgenda& ag)
/*
 * Busca una Persona en la Agenda
 * Devuelve su posicion si se encuentra, o bien -1 en otro caso
*/
{
      unsigned ind=0;
      int res;
      while ((ind < ag.num_pers) &&(nombre != ag.pers[ind].nombre))
            {
            ind=ind+1;
            }
      if (ind < ag.num_pers)
      {
            res=ind;
      }
      else
      {
            res= NO_VALIDA;
      }
      return res;
}
void AgnadirPersona(const TPersona& per, TAgenda& ag)
{
      int i;

      i=BuscarPersona(per.nombre, ag);
      if (i != NO_VALIDA)
      {
            cout<<"La persona ya se encuentra en la agenda"<<endl;
      }
      else
      {
            if  (ag.num_pers == MAXPERSONAS)
            {
                  cout<<"Agenda Llena."<<endl;
            }
            else
            {
                  ag.pers[ag.num_pers] =  per;
                  ag.num_pers = ag.num_pers + 1;
            }
    }
}

void BorrarPersona(const string& nombre, TAgenda& ag)
{
      int i;
      i= BuscarPersona(nombre, ag);
    if (i== NO_VALIDA)
```

```
    {
      cout<<"La persona no se encuentra en la agenda";
    }
    else
    {
      ag.pers[i]= ag.pers[ag.num_pers-1];
        ag.num_pers = ag.num_pers - 1;
     }
}

void ModificarPersona(const string& nombre, const TPersona& nuevosDatos,
TAgenda& ag)
{
      int i;
    i = BuscarPersona(nombre, ag);
    if (i == NO_VALIDA)
      {
                  cout<<"La persona no se encuentra en la agenda";
      }
    else
             {
            ag.pers[i] = nuevosDatos;
             }
 }

char Menu()
{
    char res;
    cout<<"A.- Anadir Persona"<<endl;
    cout<<"B.- Buscar Persona"<<endl;
    cout<<"C.- Borrar Persona"<<endl;
    cout<<"D.- Modificar Persona"<<endl;
    cout<<"X.- Salir"<<endl;

    do
      {
            cout<<"Introduzca Opcion: ";
            cin>>res;
      }while(res!='A'&&res!='B'&&res!='C'&&res!='D'&&res!='X');

    return res;
}

int main()
{
    TAgenda ag;
    char opcion;
    TPersona per;
    string nombre;
    int i;

    Inicializar(ag);
    do
    {
        opcion = Menu();
        switch (opcion)
        {
        case ('A'):
                                cout<<"Introduzca datos de Persona: "<<endl;
                                LeerPersona(per);
                                AgnadirPersona(per, ag);
                                break;
        case ('B'):
                                cout<<"Introduzca Nombre";
                                cin>>nombre;
```

```
                                i = BuscarPersona(nombre, ag);
                                if (i == NO_VALIDA)
                                  {
                                        cout<<"No está en la agenda"<<endl;
                                  }
                                else
                                {
                                      EscribirPersona(ag.pers[i]);
                                }
                                break;
        case ('C'):
                                cout<<"Introduzca Nombre"<<endl;
                                cin>>nombre;
                                BorrarPersona(nombre, ag);
                                break;
        case ('D'):
                                cout<<"Introduzca Nombre";
                                cin>>nombre;
                                cout<<"Inserte nuevos datos: "<<endl;
                                LeerPersona(per);
                                ModificarPersona(nombre, per, ag);
                                break;
        }

    }while(opcion!= 'X');
return 0;
}
```