Tema 5

Tipos estructurados

Índice general

5.1. Introducción

En los temas anteriores se han descrito las características de los tipos de datos básicos o simples (carácter, entero y coma flotante). Asimismo, se ha aprendido a definir y a utilizar constantes simbólicas utilizando `const` y `#define`. En este tema continuaremos el examen de los restantes tipos de datos de C++, examinando especialmente el tipo array (lista o tabla), los arrays de caracteres (como un tipo especial de array) y los registros o estructuras. Comenzaremos el tema con la descripción de un tipo de gran utilidad en C++: el tipo enumerado.

5.2. El tipo enumerado

Si quisiéramos definir una variable para que represente una situación de un problema en el que hay cinco posibles colores, (rojo, amarillo, verde, azul y naranja) podríamos hacer lo siguiente:

```
const unsigned int rojo = 0,
                   amarillo = 1,
                   verde = 2,
                   azul = 3,
                   naranja = 4;
unsigned int color1, color2;
```

Sin embargo, en C++ podemos definir un nuevo tipo ordinal que conste de esos cinco valores exactamente:

```
enum    colores {rojo, verde, azul, amarillo, naranja};
colores color1, color 2;
```

La declaración de un tipo de esta índole consiste en asociar a un identificador una enumeración de los posibles valores que una variable de ese tipo puede tomar.

Una misma constante no puede aparecer en dos definiciones de tipo.

El orden de los valores de estos nuevos tipos declarados por el programador será aquel en que aparecen dentro de la lista de enumeración de los elementos del tipo. El tipo enumerado es también escalar y ordinal. Por ejemplo, el siguiente código:

```
#include <iostream>

using namespace std;

enum colores{rojo, amarillo, verde};
enum mascolores{negro, azul, violeta};

int main(){
    colores color1;
    mascolores color2;

    color1=amarillo;
    cout << colores(1) << endl;
    cout << color1 << endl;
    color2=negro;

    return 0;
}
```

devuelve como resultado:

```
1
1
```

El resultado por pantalla es ése porque, internamente, C++ representa los valores enumerados con números naturales consecutivos desde el 0.

Al tratarse de tipos ordinales, los valores de tipo enumerado tendrán su predecesor (excepto el primero del tipo) y sucesor (excepto el último del tipo), por lo cual sería válido:

```
if (color1 < azul) ...
switch (color1)
    case rojo: ...
```

No existen operaciones aritméticas definidas para los tipos enumerados. Por lo tanto, sería inválido:

```
color1 = color1 + 1 ;
```

Los valores de tipo enumerado no pueden leerse ni escribirse directamente. Es cuestión del programador el implementar los procedimientos y funciones adecuados para la entrada / salida y para obtener el siguiente elemento y el anterior.

El siguiente ejemplo muestra en pantalla los diferentes valores del tipo enumerado colores:

```
colores SUC(const colores c) {
    int c_equiv;

    c_equiv = int(c);
    c_equiv++;
    return colores(c_equiv);
}

int main() {
    colores col;
    bool fin_bucle = false;
    col = rojo;
    while (! fin_bucle)
    {
```

```
            switch (col){
            case rojo:
                cout << "rojo" << endl;
                break;
            case amarillo:
                    cout << "amarillo" << endl;
                break;
            case verde:
                cout << "verde" << endl;
                break;
        }
        fin_bucle = (col == verde);
        if (!fin_bucle)
        {
            col = SUC(col);
        }
    }

    return 0;
}
```

5.3. El tipo array

Un array (lista o tabla) es una secuencia de objetos del mismo tipo. Los objetos se llaman elementos del array y se numeran consecutivamente 0, 1, 2, 3,... El tipo de elementos almacenados en el array puede ser de cualquier tipo de dato de C++, incluyendo tipos definidos por el usuario, como se describirá más tarde.

Los arrays se numeran, como se ha dicho arriba, consecutivamente, a partir del 0. Estos números se denominan índices y localizan la posición del elemento dentro del array, proporcionando acceso directo al array.

Si el nombre del array es a, entonces a[0] es el nombre del elemento que está en la posición 0 del array. En general, el i-ésimo elemento está en la posición i-1 del array.

Al igual que cualquier tipo de variable, se debe declarar un array antes de utilizarlo. Un array se declara de manera similar a otros tipos de datos, excepto que se debe indicar al compilador el tamaño o longitud del array. La sintaxis de la declaración es la siguiente:

```
tipo nombreArray[numeroDeElementos];
```

Por ejemplo, para crear una lista de 10 variables enteras, podemos hacer:

```
int numeros[10];
```

Esta declaración hace que el compilador reserve espacio suficiente para contener 10 valores enteros. Ejemplos:

```
int edad[5]; int pesos[30], longitudes[200];
float salarios[30];
double temperaturas[10];
char letras[25];
```

En los programas podemos referenciar los elementos utilizando expresiones para los índices. Por ejemplo, algunas referencias a elementos individuales, pueden ser:

```
edad[4]
ventas[total + 5]
bonos[mes]
salario[mes[i] * 5]
```

Es importante tener presente el hecho de que C++ no comprueba que los índices del array están dentro del rango definido. Así, por ejemplo, se puede intentar acceder a `numeros[12]` de un array declarado como `int numeros[10];` y el compilador no producirá ningún error, lo que puede provocar un fallo en el programa.

Otro punto importante a tener en cuenta es que en C++ no se puede hacer una asignación directa entre arrays con el operador = ni una comparación entre arrays con el operador ==. Para hacer una copia o una comparación hay que hacerla elemento a elemento:

```
int a[TAMARR], b[TAMARR]; int i; .....
// Copia de los valores de b en a
for (i = 0; i < TAMARR; i++) {
    a[i] = b[i];
}
```

5.3.1. Inicialización de un array

Se deben asignar valores a los elementos de un array antes de utilizarlos, tal como se asignan valores a variables. Por ejemplo, para asignar valores a cada uno de los elementos del array salarios, declarado como:

```
int salarios[5];
```

podríamos hacer:

```
salarios[0]=1500;
salarios[1]=2000;
salarios[2]=3000;
salarios[3]=9500;
salarios[4]=1200;
```

Cuando se inicializa un array, el tamaño del array se puede determinar automáticamente por las constantes de inicialización. Estas constantes se separan por comas y se encierran entre llaves, como en los siguientes ejemplos:

```
int numeros [5] = {2, 5, 19, 200, 3};
int n[] = {2, 3, -21}; // Esto declara un array de 3 elementos
char c[] = {'h', 'o', 'l', 'a'};    // Esto declara un array de
                                    // caracteres de 4 elementos
```

En C++ se pueden dejar los corchetes vacíos en una declaración sólo cuando se asignan valores a un array.

Se pueden asignar constantes simbólicas como valores numéricos.

Por último, se pueden asignar valores a un array utilizando bucles. Ejemplo:

```
int numeros[1000]; for (int i=0; i<1000; i++) {
numeros[i] = 0;
}
```

5.3.2. Arrays multidimensionales

Son aquellos que tienen más de una dimensión y, en consecuencia, más de un índice. Los arrays más usuales son los de dos dimensiones, también llamados tablas o matrices. Un array de dos dimensiones equivale a una tabla con múltiples filas y múltiples columnas.

Un array de dos dimensiones es, en realidad, un array de arrays. Es decir, es un array unidimensional en el que cada elemento, en lugar de ser un valor entero, un real o un carácter, es otro array. Este razonamiento puede hacerse extensivo a los arrays con un mayor número de dimensiones.

En la declaración de un array multidimensional en C++ es necesario que cada dimensión esté encerrada entre corchetes. Ejemplos:

```
char pantalla[25][40];
int puestos[6][8];
float matriz[10][20];
float cubo[10][10][10];
```

El acceso a los elementos de un array multidimensional se hace utilizando los índices de ese array, de la misma manera que se hacía con arrays de una dimensión. Por ejemplo, para acceder al elemento situado en la tercera fila, cuarta columna del array matriz del ejemplo anterior, haríamos: `matriz[3][4]`.

Los arrays multidimensionales se pueden inicializar, al igual que los de una dimensión, cuando se declaran. La inicialización consta de una lista de constantes separadas por comas y encerradas entre llaves, como en:

```
int tabla[2][3] = {2, 3, 4, 5, -2, 9};
```

o bien, los formatos:

```
int tabla[2][3] = {{2, 3,4},{5,-2,9}};
int tabla[2][3] = {{2, 3, 4},
                   {5, -2, 9}
                  };
```

El orden que se sigue cuando se inicializa un array multidimensional va de la primera dimensión a la última. En el caso del array tabla, el orden es tabla[0][0], tabla[0][1], tabla[0][2], tabla[1][0], tabla[1][1], etc.

Al igual que ocurría con los arrays unidimensionales, la inicialización de un array multidimensional puede hacerse utilizando bucles. Ejemplo:

```
int matriz[10][20];
for (filas = 0; filas<10; filas++) {
    for (columnas = 0; columnas < 20;columnas++)
    {
        matriz[filas][columnas]=0;
    }
}
```

5.3.3. Utilización de arrays como parámetros

Una función puede tomar como parámetro un array al igual que toma un valor de un tipo predefinido. Por ejemplo, se puede declarar una función que acepte un array de valores double como parámetro:

```
double SumaDeDatos(double datos[MAX]);
```

Esta función acepta como parámetros variables que sean arrays de `MAX` elementos de tipo double. No obstante, es muy útil definir funciones que acepten parámetros de tipo array de cualquier longitud. Eso se puede conseguir si no especificamos el tamaño concreto del array en la lista de parámetros formales:

```
double SumaDeDatos(double datos[]);
```

Veamos un ejemplo:

```
const MAX 100;
double datos[MAX];
double SumaDeDatos(double datos[], int n);

double SumaDeDatos(double datos[], int n)
{
    double Suma=0;
    int indice;
    for (indice = 0; indice < n; indice++)
    {
        Suma += datos[indice];
    }
    return Suma;
}
```

Esta función devuelve el valor de la suma de los n primeros elementos del array que se le pasa como parámetro.

Si el array que se pasa como parámetro es multidimensional, se puede dejar abierta la primera dimensión: `double SumaVectorMultidim (double v [ ][MAX1][MAX2]);`

Esta función acepta como parámetro arrays cuya segunda dimensión sea de tamaño `MAX1`, su tercera dimensión sea de tamaño `MAX2` y su primera dimensión sea de cualquier tamaño.

Por cuestiones de eficiencia, en C++ los arrays se pasan por referencia. Esto significa que cuando se utiliza un array como parámetro en una llamada a una función, no se hace una copia del parámetro real como en el caso de los tipos predefinidos, porque eso puede costar mucho tiempo. En su lugar, se copia una referencia a la dirección de memoria del primer componente del array. La consecuencia de este mecanismo en el

paso de parámetros es que las modificaciones que se hacen sobre el parámetro formal se conservan en el parámetro real, a diferencia de lo que ocurre con los tipos predefinidos.

Por tanto, se debe tener cuidado de no modificar el array en la función llamada de manera involuntaria. Para evitarlo, cuando el parámetro sea de entrada lo trataremos siempre como constante:

```
double SumaDeDatos(const double a[MAX]);
```

con lo que el compilador evitará cualquier modificación del contenido del array. Si, en cambio, el parámetro es de salida, lo definiremos sin considerarlo como constante:

```
double ModificarVector (double v[MAX]);
```

Por último, es importante reseñar la imposibilidad de devolver arrays como resultado de una función (utilizando la palabra `return`). Es decir, el siguiente ejemplo no está permitido y el compilador genera un error:

```
char InvertirCadena [100] (char cadena[100]); // ERROR!!!!!!!!
char [100] InvertirCadena (char cadena[100]); // ERROR!!!!!!!!
```

No obstante, es posible conseguir un "efecto"similar a la devolución de un array como resultado de una función haciendo uso del concepto de puntero, que se explicará en temas posteriores.

5.4. Cadenas de caracteres

Una cadena de caracteres es una secuencia de caracteres, tales como "BCDEFG". Es un tipo de datos muy útil, pero en C++ se puede implementar mediante un array de caracteres: `char cadena[]="BCDEF";`

La diferencia básica entre un array de caracteres y una cadena de caracteres es de tipo lógico. Una cadena no tiene por qué llenar todo el array, sino sólo la primera parte. Para diferenciar cuándo acaba una cadena dentro de un array usamos el carácter 0 (`'\0'` o `char(0)`) para indicar que los caracteres de las siguientes posiciones del array no son válidos para la cadena.

Es importante señalar que una cadena sin el carácter 0 como terminador no es una cadena correcta y eso tenemos que tenerlo en cuenta a la hora de reservar espacio en el array. Un array `cad[10]` puede contener una cadena de, a lo sumo, 9 caracteres, que estarían en las posiciones desde `cad[0]` hasta `cad[8]` y en `cad[9]` aparecería el carácter terminador.

El medio más fácil de inicializar una cadena de caracteres es hacer la inicialización en la declaración: `char cadena[7] = "BCDEFG";`

El compilador inserta automáticamente un carácter nulo al final de la cadena. No obstante, la asignación de valores a una cadena de caracteres se puede hacer de la siguiente manera:

```
cadena[0] = 'A';
cadena[1] = 'B';
cadena[2] = 'C';
cadena[3] = 'D';
cadena[4] = 'E';
cadena[5] = 'F';
cadena[6] = '\0';
```

Sin embargo, no se puede hacer una asignación directa al array, del siguiente modo:

```
cadena = "ABCDEF"; // Esto no es valido
```

Lectura y escritura de cadenas de caracteres

Cuando se introduce una cadena de caracteres por teclado, el carácter nulo de fin de cadena se introduce automáticamente cuando introducimos el retorno de carro. En el siguiente ejemplo, se muestra un algoritmo para el cálculo de la longitud de una cadena de caracteres que se introduce por teclado:

```
#include <iostream>

using namespace std;

unsigned int longitud(char []);

int main(){
    char cadena[100];

    cout << "Introduzca una cadena de 100 caracteres como maximo"
        << endl;
    cin  >> cadena;

    cout << "La longitud de la cadena es " << longitud(cadena)
```

```
            << " caracteres" << endl;

        return 0;
    }

    unsigned int longitud(char cad[]){
        unsigned int numCar = 0;

        while (cad[numCar] != '\0')
            numCar++;

        return numCar;
    }
```

Como puede comprobarse compilando y ejecutando el ejemplo anterior, la lectura usual de datos usando el objeto `cin` y el operador >> puede producir resultados inesperados. Por ejemplo, si en el caso anterior introducimos por teclado la cadena `"Pepe Sanchez"` nos dirá que tiene 4 caracteres. La razón es que la operación de lectura >> termina siempre que se encuentra un carácter separador, esto es, un blanco, un tabulador o un retorno de carro (`\n`). Para leer líneas completas se puede usar la función `getline()` junto con `cin`, en lugar del operador >>.

Esta función está sobrecargada y se define como:

```
getline(char p[], int tam);
getline(char p[], int tam, char term);
```

En la llamada a `getline` con dos parámetros, el primer parámetro es el array de caracteres donde se almacenará el resultado de la lectura. El segundo indica el número máximo de caracteres a leer. La lectura acabará cuando se lea un retorno de carro o se haya leído el máximo número de caracteres.

Si queremos que la lectura acabe cuando se llegue a un carácter distinto del retorno de carro podemos hacer la llamada a la función con tres parámetros, indicando en el tercer parámetro cuál es el carácter en el que acabamos la lectura.

Por ejemplo, para leer una línea completa, incluyendo los espacios en blanco:

```
cin.getline(cadena, 100);
```

En este caso, estamos limitados a líneas de hasta 100 caracteres de longitud.

Cuando se leen caracteres por teclado, el sistema operativo los va almacenando en una memoria intermedia, el denominado *buffer de teclado*. Cuando se hace una lectura, el programa coge de ese *buffer* de teclado los datos que necesita. Si en el *buffer* hay más datos de los necesarios para esa lectura, los caracteres restantes, que no son usados por el programa, quedan temporalmente almacenados en el *buffer* y pueden ser usados en futuras lecturas. Por ejemplo, con `cin >> car;` leemos un carácter por teclado y se lo asignamos a la variable `car`. El usuario, normalmente, actúa de la siguiente manera: introduce un carácter y pulsa la tecla `ENTER`, es decir, acaba la introducción del dato con un retorno de carro. El retorno de carro, no sólo indica que acaba la introducción de datos, sino que provoca que el carácter de retorno de carro (carácter 13 de la tabla ASCII, o $'\backslash n'$) quede introducido en el *buffer* de teclado.

También hay que hacer notar que el operador >> aplicada a la lectura de caracteres puede no funcionar como se espera. Se dice que el operador >> hace una lectura formateada, es decir, sólo lee aquellos caracteres que considera válidos y elimina de la lectura los que considera que son simplemente separadores, como los espacios en blanco, los tabuladores y los retornos de carro. Veamos cómo funciona un código como el siguiente:

```
// Este programa cuenta el numero de letras leidas
// hasta encontrar el retorno de carro

#include <iostream>

using namespace std;

int main () {

char letra;
int contador = 0;

cout << "Introduce la secuencia de letras: ";

do
{
    cin >> letra;
    contador++;
} while (letra != '\n');
```

```
cout << "El numero de letras es " << contador << endl;

return 0;
}
```

Supongamos que los datos de entrada son `En un lugar de la mancha \n`. El programa leería las letras `E` y `n` y después se saltaría el espacio en blanco para leer la `u`. Contaría 19 letras (es decir, no tendría en cuenta ningún espacio en blanco) y no acabaría. El programa no acaba porque como considera que el retorno de carro es un separador y no una letra válida, no lo lee, sino que espera que se introduzca una nueva letra para hacer la lectura `cin >> letra`. Por tanto, `letra` nunca toma como valor `\n` y el programa no acaba.

La función `get` sí lee todos los caracteres, incluidos los separadores y podemos usarla para hacer una versión corregida del programa anterior:

```
// Este programa cuenta el numero de letras leidas
// hasta encontrar el retorno de carro

#include <iostream>

using namespace std;

int main () {

char letra;
int contador = 0;

cout << "Introduce la secuencia de letras: ";

cin.get(letra);
while (letra != '\n')
{
    contador++;
    cin.get(letra);
};

cout << "El numero de letras es " << contador << endl;
```

```
return 0;
}
```

Este funcionamiento distinto del operador >> y las operaciones `get` y `getline` puede provocar errores al combinarlos. Por ejemplo, en el programa:

```
// Este programa lee un numero y despues un caracter
// que esta en la siguiente linea

#include <iostream>

using namespace std;

int main () {

char letra;
int numero;

cout << "Introduce los datos: ";

cin >> numero;
cin.get(letra);

cout << "El numero es " << numero << endl;
cout << "La letra es "  << letra  << endl;

return 0;
}
```

Si la entrada de datos es `25 \nc`, la lectura del número con >> salta los espacios en blanco iniciales y lee el 25. Cuando se lee el carácter, la operación `get` no salta los separadores y, en vez de la letra `c`, leería un espacio en blanco.

Para evitar esto, después de una lectura de caracteres con el operador >> podemos usar la función `ignore`, que permite ignorar (en otras palabras, sacar del *buffer* de teclado sin asignárselo a ninguna variable) los caracteres sobrantes, incluyendo separadores y caracteres de retorno de carro de lecturas previas. La función `ignore` está definida como:

```
ignore(int n=1, char term = eof);
```

Esta función saca caracteres del *buffer* de teclado hasta sacar tantos como indique el primer parámetro o encontrar el carácter indicado en el segundo parámetro. El código anterior se puede modificar incluyendo una llamada a `ignore`:

```
#include <iostream>

using namespace std;

int main () {

char letra;
int numero;

cout << "Introduce los datos: ";

cin >> numero;
cin.ignore(256,'\n');
cin.get(letra);

cout << "El numero es " << numero << endl;
cout << "La letra es "  << letra  << endl;

return 0;
}
```

En este caso, la función `ignore` ignorará como máximo 256 caracteres si no se ha encontrado antes el carácter de fin de línea.

Otras operaciones con cadenas de caracteres

La biblioteca `string.h` contiene las funciones de manipulación de cadenas utilizadas más frecuentemente. Los archivos de cabecera `stdio.h` y `iostream.h` también soportan E/S de cadenas. Algunas funciones de la biblioteca `string.h` de uso común y recomendado son:

Función	Cabecera de la función y prototipo
`strcat`	`char *strcat(char destino[], const char fuente[]);`
	Añade la cadena fuente al final de destino
`strcmp`	`int strcmp(const char s1[], const char s2[]);`
	Compara s1 a s2 y devuelve:
	0 si s1 = s2 $<$ 0 si s1 $<$ s2 $>$ 0 si s1 $>$ s2
`strcpy`	`char *strcpy(char destino[], const char *fuente[]);`
	Copia la cadena fuente en la cadena destino
`strlen`	`size_t strlen (const char s[]);`
	Devuelve la longitud de la cadena s

5.5. El tipo registro

Una estructura o registro es una colección de uno o más tipos de elementos denominados miembros o campos, cada uno de los cuáles pueden ser un tipo de dato diferente. El tipo de elementos almacenados en un registro puede ser de cualquier tipo de dato de C++, incluyendo tipos definidos por el usuario. El formato de la declaración es:

```
struct <nombre de la estructura> {
    <tipo de dato miembro 1> <nombre miembro 1>;
    <tipo de dato miembro 2> <nombre miembro 2>;
    ...
    <tipo de dato miembro n> <nombre miembro n>;
}
```

Ejemplo:

```
struct coleccion_CD{
    char titulo[30];
    char nombreArtista[20];
    unsigned int numCanciones;
    float precio;
    char fechaCompra[9];
}
```

Una vez declarada una estructura, podemos declarar variables de ese nuevo tipo registro. Ejemplo:

```
struct coleccion_CD c1, c2, c3;
```

Podemos hacer asignaciones directas entre variables del mismo tipo registro. Por ejemplo: `c1 = c2;` El acceso a los miembros de un registro se hace mediante el operador punto. Por ejemplo, las siguientes son expresiones válidas para el acceso al registro declarado más arriba:

```
strcpy(c1.titulo, "Turandot");
strcpy(c1.nombreArtista, "Puccini");
cout << c1.numCanciones << endl;
c1.precio=18.23 * (1 + 0.16);
cin.getline(c1.fechaCompra,9);
```

Los registros se comportan igual que un valor de tipo predefinido cuando se pasa como parámetro. El mecanismo por defecto es la copia en el parámetro formal, con lo que las modificaciones no se conservan en el parámetro real. Si queremos que el parámetro sea de salida lo definiremos con el operador `&`.

También es válido usar un registro como el tipo devuelto por una función.

5.6. Definición de tipos con nombre

Se dice que las variables de tipo array definidas en los apartados anteriores son de tipo anónimo porque el tipo al que pertenecen no tiene nombre. Esto provoca algunas incompatibilidades entre variables porque no se pueden definir del mismo tipo.

La instrucción `typedef` permite evitar esta situación. Esa instrucción sirve para darle un nombre nuevo a un tipo:

```
typedef <definicion del tipo> <nuevo nombre para el tipo>;
```

Por ejemplo, podemos darle a un tipo ya existente un nuevo nombre que sea más descriptivo en un programa concreto:

```
typedef float TRadianes;
```

En el caso de la definición de arrays, los tipos aún no tienen nombre y se les puede dar como el siguiente ejemplo:

```
typedef float TRadianes;
typedef int TVectorDeEnteros [TAMVEC];
```

Así, `TVectorDeEnteros` es un tipo que es un array de `TAMVEC` enteros y podemos declarar variables y parámetros de ese tipo.

```
TVectorDeEnteros v1, v2;
int F(const TVectorDeEnteros vv);
```

5.7. Ejemplo

Se pretende hacer un programa que para unos datos dados referentes a las temperaturas y pluviosidad de las provincias andaluzas, durante los 12 meses del año, calcule la media anual de cada provincia, y la media de la comunidad autónoma de cada mes. También calculará la media total de la comunidad.

```
        Enero     Febrero    Marzo     ...
Huelva  20.0, 10  22.5, 12   23.2, 35  ...
Sevilla 15.1, 20  14.7, 27   15.6, 25  ...
... ... ... ... ...
```

```
/*************************************************************
 * clima:
 *       Programa para el calculo de promedios de
 *       temperatura y pluviosidad de las provincias
 *       andaluzas a lo largo de todo el anyo.
 *
 * Programador:
 *
 * Fecha:
 *************************************************************/
#include <iostream>
#include <cstdlib>

using namespace std;

// Declaracion de constantes y tipos
const unsigned int NM_MESES = 12;
const unsigned int NM_PROV = 8;
enum TpProv {hu, se, ca, ma, co, gr, ja, al};
enum TpMes {enero, febrero, marzo, abril, mayo, junio, julio, \
            agosto, septiembre, octubre, noviembre, diciembre};
struct TpDat{
    float temp; // Temperatura
    float pluv; // Pluviosidad
};
```

```
typedef TpDat TpTabla [NM_PROV][NM_MESES];

// Declaracion de prototipos de funciones
// Escribe por pantalla el nombre del mes float
void escr_meses(const TpMes);

// Escribe por pantalla el nombre de la provincia
void escr_prov(const TpProv);

// Media anual de temperaturas
media_anual_temp(const TpTabla, const TpProv);

// Media anual de pluviosidad
float media_anual_pluv(const TpTabla, const TpProv);

// Media provincial de temperatura
float media_prov_temp(const TpTabla, const TpMes);

// Media provincial de pluviosidad
float media_prov_pluv(const TpTabla, const TpMes);

// Media de temperaturas de la comunidad
float media_com_temp(const TpTabla);

// Media de pluviosidad de la comunidad
float media_com_pluv(const TpTabla);
// Lee los datos de la tabla
void leer_tabla(TpTabla);

void main()
{
    TpTabla tbl; // Tabla de la Comunidad Autonoma Andaluza
    unsigned int IndMes, IndPro; // Indices para los bucles

    leer_tabla(tbl); // Lee la tabla por teclado
```

```
    // Medias provinciales
     for (IndMes=0; IndMes<NM_MESES; IndMes++){
         cout << "Temperatura media provincial del mes ";
         escr_meses(TpMes(IndMes));
         cout << ": " << media_prov_temp(tbl,TpMes(IndMes)) << endl;
         cout << "Pluviosidad media provincial del mes ";
         escr_meses(TpMes(IndMes));
         cout << ": " << media_prov_pluv(tbl,TpMes(IndMes)) << endl;
     }

     // Medias anuales por provincia
     for (IndPro=0; IndPro<NM_PROV; IndPro++){
         cout << "Temperatura media anual de la provincia ";
         escr_prov(TpProv(IndPro));
         cout << ": " << media_anual_temp(tbl,TpProv(IndPro)) << endl;
         cout << "Pluviosidad media anual de la provincia ";
         escr_prov(TpProv(IndPro));
         cout << ": " << media_anual_pluv(tbl,TpProv(IndPro)) << endl;
     }

     cout << "Temperatura media de la comunidad: "
          << media_com_temp(tbl)
          << endl;
     cout << "Pluviosidad media de la comunidad: "
          << media_com_pluv(tbl)
          << endl;

     system("PAUSE");
} // main

/*****************************************************************
 * escr_meses:
 *            Escribe en pantalla el nombre completo del mes que se
 *            pasa como parametro.
 *
 * Parametros de entrada:
```

```
 *            mes: mes del tipo TpMes
 ***************************************************************/
void escr_meses(const TpMes mes){
    switch (mes){
        case enero: cout << "Enero";
                break;
        case febrero: cout << "Febrero";
                break;
        case marzo: cout << "Marzo";
                break;
        case abril: cout << "Abril";
                break;
        case mayo: cout << "Mayo";
                break;
        case junio: cout << "Junio";
                break;
        case julio: cout << "Julio";
                break;
        case agosto: cout << "Agosto";
                break;
        case septiembre: cout << "Septiembre";
                break;
        case octubre: cout << "Octubre";
                break;
        case noviembre: cout << "Noviembre";
                break;
        case diciembre: cout << "Diciembre";
    }
} // escr_meses

/***************************************************************
* escr_prov:
*            Escribe en pantalla el nombre completo de la provincia
*            pasa como parametro.
*
* Parametros de entrada:
```

```
*            prov: provincia de tipo TpProv
*****************************************************************/
void escr_prov(const TpProv prov){
    switch (prov){
        case hu: cout << "Huelva";
                 break;
        case se: cout << "Sevilla";
                 break;
        case ca: cout << "Cadiz";
                 break;
        case ma: cout << "Malaga";
                 break;
        case gr: cout << "Granada";
                 break;
        case co: cout << "Cordoba";
                 break;
        case ja: cout << "Jaen";
                 break;
        case al: cout << "Almeria";
    }
} // escr_prov

/*****************************************************************
* media_anual_temp:
*                  Media anual de las temperaturas para una provincia
*                  de la Comunidad Autonoma Andaluza
*
* Parametros de entrada:
*            tbl: tabla con los datos de temperaturas y pluviosidad
*            prov: provincia
*****************************************************************/
float media_anual_temp(const TpTabla tbl, const TpProv prov){
    float media = 0.0;
    unsigned int IndMes;

    for (IndMes=0; IndMes<NM_MESES; IndMes++)
```

```
        media+=tbl[(unsigned int)prov][IndMes].temp;

    return media/NM_MESES;
} // media_anual_temp



/*****************************************************************
* media_anual_pluv:
*                  Media anual de la pluviosidad para una provincia
*                  de la Comunidad Autonoma Andaluza
*
* Parametros de entrada:
*            tbl: tabla con los datos de temperaturas y pluviosidad
*            prov: provincia
*****************************************************************/
float media_anual_pluv(const TpTabla tbl, const TpProv prov){
    float media = 0.0;
    unsigned int IndMes;

    for (IndMes=0; IndMes<NM_MESES; IndMes++)
        media+=tbl[(unsigned int)prov][IndMes].pluv;

    return media/NM_MESES;
}// media_anual_pluv

/*****************************************************************
* media_prov_temp:
*                  Media prov de las temperaturas para un mes
*
* Parmetros de entrada:
*            tbl: tabla con los datos de temperaturas y pluviosidad
*            mes: mes
*****************************************************************/
float media_prov_temp(const TpTabla tbl, const TpMes mes){
    float media = 0.0;
```

```
    unsigned int IndPro;

    for (IndPro=0; IndPro<NM_PROV; IndPro++)
        media+=tbl[IndPro][(unsigned int)mes].temp;

    return media/NM_PROV;
}// media_prov_temp



/*****************************************************************
* media_prov_pluv:
*                  Media prov de la pluviosidad para un mes
*
* Parametros de entrada:
*            tbl: tabla con los datos de temperaturas y pluviosidad
*            mes: mes
*****************************************************************/
float media_prov_pluv(const TpTabla tbl, const TpMes mes){
    float media = 0.0;
    unsigned int IndPro;

    for (IndPro=0; IndPro<NM_PROV; IndPro++)
        media+=tbl[IndPro][(unsigned int)mes].pluv;

    return media/NM_PROV;
}// media_prov_pluv

/*****************************************************************
* media_com_temp:
*                Media de la comunidad de temperatura
*
* Parametros de entrada:
*            tbl: tabla con los datos de la comunidad
*****************************************************************/
float media_com_temp(const TpTabla tbl){
```

```
    float media = 0.0;
    unsigned int IndPro, IndMes;

    for (IndMes=0; IndMes<NM_MESES; IndMes++)
        for (IndPro=0; IndPro<NM_PROV; IndPro++)
            media+=tbl[IndPro][IndMes].temp;

    return media/(NM_MESES*NM_PROV);
}// media_com_temp

/*****************************************************************
* media_com_pluv:
*               Media de la comunidad de pluviosidad
*
* Parametros de entrada:
*            tbl: tabla con los datos de la comunidad
*****************************************************************/
float media_com_pluv(const TpTabla tbl){
    float media = 0.0;
    unsigned int IndPro, IndMes;

    for (IndMes=0; IndMes<NM_MESES; IndMes++)
        for (IndPro=0; IndPro<NM_PROV; IndPro++)
            media+=tbl[IndPro][IndMes].pluv;

    return media/(NM_MESES*NM_PROV);
}// media_com_pluv

/*****************************************************************
* leer_tabla:
*            Lectura de los datos de temperatura y pluviosidad
*            de la Comunidad Autonoma Andaluza para un periodo
*            anual completo.
*
* Parametros de entrada/salida:
*            tbl: tabla con los datos
```

```
**************************************************************/
void leer_tabla(TpTabla tbl){
    unsigned int IndMes, IndPro;

    for (IndPro=0; IndPro<NM_PROV; IndPro++){
        cout << "Provincia: ";
        escr_prov(TpProv(IndPro));
        cout << endl;
        for (IndMes=0; IndMes<NM_MESES; IndMes++){
            cout << "     Mes: ";
            escr_meses(TpMes(IndMes));
            cout << endl;
            cout << "          Temperatura?: ";
            cin >> tbl[IndPro][IndMes].temp;
            cout << "          Pluviosidad?: ";
            cin >> tbl[IndPro][IndMes].pluv;
            cout << endl;
        }
    }
} // leer_tabla
```

Ejercicios

1. Escriba un programa que sume dos vectores de tipo base `int` e imprima el resultado en pantalla. Se considerarán datos de entrada la dimensión de los vectores y las componentes de cada uno. Suponemos que los vectores tendrán un máximo de 10 componentes. Ejecute el programa para un caso en que cada vector tenga más de 10 componentes. ¿Qué ocurre?

2. Lea un texto de teclado e imprima en pantalla un mensaje indicando cuantas veces aparecen dos letras contiguas e iguales en el mismo.

3. Las distancias en Kilómetros entre una serie de estaciones de tren son: 0 2.25 3.25 4.5 Escriba un programa que lea los datos anteriores y dé como resultado una tabla triangular que dé la distancia entre cualquier par de estaciones. Ejemplo:

	1	2	3	4
1	0.00E+0			
2	2.25E+0	0.00E+0		
3	5.50E+0	3.25E+0	0.00E+0	
4	1.00E+1	7.75E+0	4.50E+0	0.00E+0

4. Escriba un programa que efectúe la conversión de un número entero en base 10 a cualquier base. La introducción del número se hará en formato "número/base". Ejemplo: Introduzca dato: 723/4 (indica que el número 723 hay que convertirlo a base 4).

5. Escriba un programa que convierta un número entero expresado en cualquier base natural entre 2 y 10 a una base natural entre 2 y 10. Para ello la entrada se leerá según el formato: "BaseInicio/Numero/BaseDestino". Ejemplo: Introduzca dato: 4/123/8. (Indica que debe pasar el número 123 que está expresado en base 4 a base 8).

6. Escriba un programa que simule el comportamiento de una calculadora simple para aritmética fraccional. El resultado se expresará también en forma fraccional y simplificado al máximo. Supondremos que tanto el numerador como el denominador son números sin decimales. Las fracciones se leerán en formato Numerador/Denominador. Ejemplo: 34/7

7. En una entidad bancaria el número de cuenta de un cliente se va a formar añadiendo a una determinada cifra un dígito de autoverificación. Dicho dígito de autoverificación se calculará de la siguiente forma:

   a) Multiplicar la posición de las unidades y cada posición alternada por dos.

   b) Sumar los dígitos no multiplicados y los resultados de los productos obtenidos en el apartado anterior.

   c) Restar el número obtenido en el apartado 2 del número más próximo y superior a éste, que termine en cero. El resultado será el dígito de autoverificación. Codificar un programa que vaya aceptando números de cuenta y compruebe mediante el dígito de autoverificación si el número introducido es correcto o no. El proceso se repetirá hasta que se introduzca un cero como número de cuenta.

8. Se desea efectuar operaciones de suma de cantidades que se caracterizan por tener

una gran cantidad de dígitos. Escriba un programa que lea dos números, los sume e imprima el resultado en pantalla. Ejemplo:

```
Numero 1:  1111111111111111111111111111111111111111
Numero 2:  111111111111111111111111111111111111111
Resultado: 22222222222222222222222222222222222222222
```

Se considera que no se van a desear sumar números con más de 50 dígitos. Los números a sumar no contienen decimales.

9. Supongamos que deseamos evaluar a un determinado número de alumnos siguiendo el criterio de que aprobará aquel que supere la nota media de la clase. Escriba un programa que lea un número indeterminado de alumnos (como máximo 20, y las notas de tres evaluaciones, y como resultado emita un informe indicando para cada alumno las evaluaciones que tiene aprobadas y suspensas. Ejemplo de la salida que se debe obtener.

Alumno	Nota-1	Nota-2	Nota-3
Juan Lopez	Aprobado	Suspenso	Aprobado
Luis Garcia	Suspenso	Aprobado	Aprobado
Pedro Ruiz	Aprobado	Aprobado	Aprobado

10. Codifique un programa con las mismas especificaciones que el problema 8, pero para la operación de producto.

11. Escriba un programa que lea por teclado dos matrices MxN y NxP y devuelva como resultado la matriz producto MxP. Los valores M, N y P se introducirán por teclado antes de la lectura de las matrices. El tamaño máximo de M, N y P será una constante definida en el programa.

12. Implementar el tipo conjunto utilizando como base un array de booleanos. Utilizarlo para manejar un conjunto de colores (el tipo `TpColor` será un enumerado). Las operaciones básicas que deben diseñarse, además de las necesarias para introducir y extraer elementos en el conjunto y comprobar su presencia o ausencia en el mismo, son la unión y la intersección de conjuntos de colores.

13. Los alumnos de una clase desean celebrar una cena de confraternización un día del presente mes en el que puedan asistir todos. Se pide realizar un programa que recoja de cada alumno los días que le vendrían bien para ir a la cena e imprima las fechas concordantes para todos los alumnos. Los datos se introducirán por teclado y cada alumno escribirá una única línea con los días separados por espacios en blanco.

14. Implementar las funciones `LongitudCadena`, `CopiarCadena` y `ComparарCadenas`. Los prototipos de estas funciones serían:

```
// Esta funcion devuelve el numero de caracteres del parametro
int LongitudCadena(const TpCadena cadena);
// Esta funcion copia el contenido de origen en destino
void CopiarCadena(TpCadena &des, const TpCadena org);
// Esta funcion compara las dos cadenas y devuelve 0 si ambas
// son iguales, -1 si alfabeticamente cad1 es anterior a cad2,
// y 1 si cad2 es alfabeticamente anterior a cad1
int CompararCadenas(const TpCadena cad1, const TpCadena cad2);
```

15. Implementar las función `CadenaLong` que transforme una cadena de caracteres constituida sólo por dígitos ('0', '1', ...,'9') en un número entero sin signo. Utilizar como tipo numérico el tipo `unsigned long int`. El prototipo de esta función debe ser el siguiente:

```
unsigned long CadenaLong (const TpCadena cadena);
```

16. Implementar las función `LongCadena` que transforme un número entero sin signo en una cadena de caracteres constituida sólo por dígitos ('0', '1', ...,'9'). Utilizar como tipo numérico el tipo `unsigned long int`. El prototipo de esta función debe ser el siguiente:

```
LongCadena (const unsigned long numero, TpCadena & cadena);
```

17. Escriba un programa que se comporte como una calculadora que pida repetidamente un operador de conjuntos y dos operandos que sean conjuntos de letras minúsculas y que escriba el resultado de la operación. Los conjuntos estarán implementados según el ejercicio 12 de esta relación. Las operaciones se expresan como caracteres, siendo válidas las siguientes:

    - + Union de conjuntos
    - - Diferencia de conjuntos
    - * Interseccion de conjuntos

    El proceso se repetirá hasta que se introduzca como código de operación el carácter `'&'`. Los operadores y el resultado se expresan como cadenas de caracteres. Ejemplo:

```
Operacion = *
Operando1 = azufre
Operando2 = zafio
Resultado = afz
Operacion = -
Operando1 = abracadabra
Operando2 = abaco
Resultado = rd
Operacion = &
FIN
```

Nota para todos los ejercicios:

- No se permite el uso de variables globales.
- Documentar los programas.