Programación 1

Tema 11

Registros

Escuela de
Ingeniería y Arquitectura
Universidad Zaragoza

Índice

- Registros y campos
- Dominio de valores
- Representación externa
- Operaciones
- Problemas y ejemplos

Problema

- Gestionar información relativa a una persona:
  - Nombre
  - Apellidos
  - NIF
  - Fecha de nacimiento
  - Estado civil (casado, no casado)
  - Sexo

Registro

- **Registro** o **tupla**
  - Agrupan datos de igual o diferente naturaleza relacionados entre sí
- Para utilizarlos, hay que definir antes un **tipo registro**

Concepto de registro

En capítulos anteriores hemos trabajado con datos con estructura de tabla (vectores y matrices), datos agregados del mismo tipo a los que se accede a través de uno o más índices numéricos.

En este capítulo vamos a presentar un nuevo mecanismo de agregación de información que permite agrupar datos relacionados entre sí que pueden ser de igual o de diferente naturaleza. Los datos resultantes de esta agregación se denominan **registros** o **tuplas** de datos.

Fuente: **Javier Martínez.** «Capítulo 11. Estructuración agregada de datos». *Curso de Programación 1*. 2017

Sintaxis

```
/*
 * Definición de un nuevo tipo de dato con estructura de registro:
 */
struct NombreTipo {
    tipo_1 nombre_campo_1;
    tipo_2 nombre_campo_2;
   ...
    tipo_n nombre_campo_n;
};

/*
 * Definición de datos, simples o estructurados, del tipo definido
 * anteriormente:
 */
NombreTipo reg1, reg2;
NombreTipo reg3;
NombreTipo vector[100];
```

Concepto de registro

El tipo de dato se identifica mediante un nombre (`NombreTipo`). Este nombre se utiliza posteriormente para declarar datos de ese tipo. Un registro agrupa tantos datos como sea necesario. cada uno de ellos se suele denominar **campo del registro** o, simplemente, **campo**. Cada campo de un registro se declara especificando el tipo de dato asociado (`tipo_1`, `tipo_2`, etc.) y el nombre que el programador da al campo (`nombre_campo_1`, `nombre_campo_2`, etc.).

Fuente: **Javier Martínez.** «Capítulo 11. Estructuración agregada de datos». *Curso de Programación 1*. 2017

Ejemplos

```
struct Fecha {
    int dia, mes, anyo;
};

Fecha hoy;
Fecha cumplesClase[60];
```

Ejemplos

```
struct Nif {
    int  dni;    // número del DNI
    char letra; // letra asociada al DNI
};
```

Sintaxis

```
<definiciónTipoRegistro>
  ::= “struct” <identificador> “{”
      { <definiciónCampo> }
      “}” “;”
<definiciónCampo>
  ::= <tipo> <declaraciónSimple>
      { “,” <declaraciónSimple> } “;”
<declaraciónSimple>
  ::= <nombreCampo> [ “=” <expresión> ]
```

Registros

- Dominio de valores
  - Producto cartesiano de los dominios de valores de los campos
- Representación externa
  - Listas (solo en la inicialización)
    - `Fecha hoy = {15, 11, 2019};`
    - `Nif rey = {15, 'S'};`

Registros

- Operadores:
  - "." : operador de selección de campo
    - `hoy.dia = 27;`
    - `hoy.dia++;`
    - `cout << rey.dni << "-" << rey.letra;`
    - `cumplesClase[0].mes = 8;`
  - Asignación
- No son operadores disponibles:
  - Comparación
  - Lectura de teclado o escritura en la pantalla

Selección de un campo de un registro

Para trabajar de forma individualizada con cada uno de los campos de un registro se ha de seleccionar el nombre del campo mediante el operador de selección «`.`» al que precede el nombre del registro y al que sigue el nombre del campo. Ejemplos:

- `reg1.nombre_campo_3`
- `reg2.nombre_campo_1`
- `T[i + 1].nombre_campo_2`

Metodología de trabajo con registros en módulos

- Fichero de **interfaz** del módulo
  - Definición del tipo registro
  - Declaraciones de funciones adicionales para trabajar con el tipo
- Fichero de **implementación** del módulo
  - Definiciones de las funciones declaradas en el fichero de interfaz
  - Definiciones de otras funciones auxiliares

Metodología de trabajo con registros en módulos

Cuando se define un nuevo tipo de dato se recomienda diseñar una colección de funciones básicas para facilitar el trabajo con los datos del nuevo tipo [...].

Desde un punto de vista metodológico conviene programar la definición de cada nuevo tipo y sus funciones básicas en un nuevo módulo. De esta forma se facilita la reutilización del nuevo tipo en todos los programas en los que sea necesario. La definición del nuevo módulo consta de dos ficheros:

- Fichero de **interfaz** del módulo. Contiene la definición del nuevo tipo de dato y los prototipos de las funciones básicas de carácter público que se ponen a disposición de los desarrolladores de otros módulos o programas.
- Fichero de **implementación** del módulo con el código de sus funciones básicas y, en su caso, con la definición de los elementos privados auxiliares que fueran necesarios.

Fuente: **Javier Martínez.** «Capítulo 11. Estructuración agregada de datos». *Curso de Programación 1*. 2017

Ejemplo. Representación de un número de identificación fiscal (NIF)

Para representar el número de identificación fiscal (NIF) de una persona se ha definido el tipo de datos `Nif` como un registro con dos campos. El campo `dni` corresponde a un entero que representa el número del documento nacional de identidad de la persona y el campo `letra` que representa la letra mayúscula asociada, por motivos de seguridad, al número de DNI anterior.

Observar que la definición de la estructura del tipo `Nif` viene precedida por un comentario que explica qué es lo que representa un dato de este tipo y, cada uno de sus dos campos, viene acompañado por un comentario que precisa el significado de la información almacenada en él.

Siempre que se defina un nuevo tipo de dato debe procederse de un modos similar, documentando adecuadamente el tipo definido y explicando los detalles de su representación interna.

[...]

En el caso del tipo `Nif` se ha optado por definir [dos] funciones básicas [...]:

- La función `esValido(unNifAValidar)` permite comprobar si la letra del dato `unNif` es la que corresponde a su número de DNI.
- [La función `mostrar(unNifAEscribir)` que permite escribir el dato `unNifAEscribir` en la pantalla, con un formato como «01234567-L».]

Fuente: **Javier Martínez.** «Capítulo 11. Estructuración agregada de datos». *Curso de Programación 1*. 2017

Número de identificación fiscal. Interfaz

```
/*
 * Definición del tipo de dato Nif que representa la
 * información del NIF (Número de Identificación
 * Fiscal) de una persona.
 */
struct Nif {
    int dni;        // número del DNI
    char letra;     // letra asociada al DNI anterior
};

...
```

Número de identificación fiscal. Interfaz

```
...

/*
 * Pre:  ---
 * Post: Ha devuelto «true» si y solo si
 *       «nifAValidar» define un NIF válido,
 *       es decir, su letra es la que le corresponde
 *       a su DNI.
 */
bool esValido(const Nif unNifAValidar);

...
```

Número de identificación fiscal. Interfaz

```
...

/*
 * Pre:  El valor del parámetro
 *       «nifAEscribir» representa un NIF
 *       válido.
 * Post: Ha escrito «nifAEscribir» en la
 *       pantalla, con un formato como
 *       «01234567-L».
 */
void mostrar(const Nif nifAEscribir);
```

Número de identificación fiscal. Implementación

La implementación de las funciones básicas especificadas anteriormente, en el fichero de interfaz, se muestra a continuación. La función `letra(dni)` es una función auxiliar invisible desde el exterior del módulo[, al igual que tampoco lo son las constantes `NUM_LETRAS` y `TABLA_NIF`].

Número de identificación fiscal. Implementación

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

const int NUM_LETRAS = 23;
const char TABLA_NIF[NUM_LETRAS]
        = { 'T', 'R', 'W', 'A', 'G', 'M',
            'Y', 'F', 'P', 'D', 'X', 'B',
            'N', 'J', 'Z', 'S', 'Q', 'V',
            'H', 'L', 'C', 'K', 'E' };
```

Número de identificación fiscal. Implementación

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

const int NUM_LETRAS = 23;
const char TABLA_NIF[NUM_LETRAS + 1]
        = "TRWAGMYFPDXBNJZSQVHLCKE";
```

Número de identificación fiscal.
Implementación

```
/*
 * Pre:  ---
 * Post: Ha devuelto la letra del número
 *       de identificación fiscal que
 *       corresponde a un número de
 *       documento nacional de identidad
 *       igual a «dni».
 */
char letra(int dni) {
    return TABLA_NIF[dni % NUM_LETRAS];
}
```

Número de identificación fiscal.
Implementación

```
/*
 * Pre:  ---
 * Post: Ha devuelto «true» si y solo si
 *       «nifAValidar» define un NIF
 *       válido, es decir, su letra es la
 *       que le corresponde a su DNI.
 */
bool esValido(const Nif nifAValidar) {
    return letra(nifAValidar.dni)
           == nifAValidar.letra;
}
```

Número de identificación fiscal. Implementación

```
/*
 * Pre:  El valor del parámetro «nifAEscribir» representa
 *       un NIF válido.
 * Post: Ha escrito «nifAEscribir» en pantalla, con un
 *       formato como «01234567-L». También ha modificado
 *       el carácter de relleno que utiliza el manipulador
 *       «setw», estableciendo el espacio en blanco.
 */
void mostrar(const Nif nifAEscribir) {
    cout << setfill('0');
    cout << setw(8) << nifAEscribir.dni << "-"
         << nifAEscribir.letra;
    cout << setfill(' ');
}
```

Número de identificación fiscal.
Ejemplo de uso

```
Nif n;
n.dni = 1234567;
n.letra = 'L';
if (esValido(n)) {
    mostrar(n);
    cout << endl;
}
```

Número de identificación fiscal.
Ejemplo de uso

```
Nif n = {1234567, 'L'};
if (esValido(n)) {
    mostrar(n);
    cout << endl;
}
```

Horas. Fichero de interfaz horas.h

```
/*
 * Definición del tipo de dato Hora
 * que representa la información de
 * una hora como hora minutos y
 * segundos.
 */
struct Hora {
    int horas, minutos, segundos;
};
```

Horas. Fichero de interfaz horas.h

```
/*
 * Pre:  Los valores de los campos horas,
 *       minutos y segundos del parámetro
 *       «unaHora» son nulos o positivos.
 * Post: Ha modificado el valor de «unaHora»
 *       para que, representando la misma
 *       cantidad de tiempo que al principio,
 *       los campos minutos y segundos
 *       estén en el intervalo [0, 60).
 */
void ajustar(Hora& unaHora);
```

Horas. Fichero de interfaz horas.h

```
/*
 * Pre:  Los valores de los campos horas,
 *       minutos y segundos del parámetro
 *       «unaHora» son nulos o positivos.
 * Post: Ha devuelto la hora correspondiente a
 *       «unaHora», después de que haya
 *       transcurrido un tiempo de «segundos»
 *       segundos.
 */
Hora sumarSegundos(const Hora unaHora,
                   const int segundos);
```

Horas. Fichero de interfaz horas.h

```
/*
 * Pre:  Los valores de los campos horas, minutos y
 *       segundos de los parámetros «horaAnterior» y
 *       «horaPosterior» son nulos o positivos y
 *       «horaAnterior» es cronológicamente anterior
 *       a «horaPosterior».
 * Post: Ha devuelto la cantidad de tiempo
 *       transcurrida entre «horaAnterior» y
 *       «horaPosterior», expresada en segundos.
 */
int calcularTiempoTranscurrido(const Hora horaAnterior,
                               const Hora horaPosterior);
```

Horas. Fichero de interfaz horas.h

```
/*
 * Pre:  ---
 * Post: Ha escrito el valor de
 *       «unaHora» en pantalla, con
 *       el formato "27:59:59" o
 *       "03:04:05".
 */
void mostrar(const Hora unaHora);
```

Horas.
Fichero de implementación horas.cpp

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

Horas.
Fichero de implementación horas.cpp

```
/*
 * Pre:  Los valores de los campos horas, minutos y segundos del
 *       parámetro «unaHora» son nulos o positivos.
 * Post: Ha modificado el valor de «unaHora» para que, representando
 *       la misma cantidad de tiempo que al principio, los campos
 *       minutos y segundos estén en el intervalo [0, 60).
 */
void ajustar(Hora& unaHora) {
    unaHora.minutos += unaHora.segundos / 60;
    unaHora.segundos = unaHora.segundos % 60;

    unaHora.horas += unaHora.minutos / 60;
    unaHora.minutos = unaHora.minutos % 60;
}
```

Horas.
Fichero de implementación horas.cpp

```
/*
 * Pre:  Los valores de los campos horas, minutos y segundos del
 *       parámetro «unaHora» son nulos o positivos.
 * Post: Ha devuelto la hora correspondiente a «unaHora»,
 *       después de que haya transcurrido un tiempo de
 *       «segundos» segundos.
 */
Hora sumarSegundos(const Hora unaHora, const int segundos) {
    Hora resultado = unaHora;
    resultado.segundos += segundos;
    ajustar(resultado);
    return resultado;
}
```

Horas.
Fichero de implementación horas.cpp

```
/*
 * Pre:  Los valores de los campos horas, minutos y segundos de
 *       los parámetros «horaAnterior» y «horaPosterior» son
 *       nulos o positivos y «horaAnterior» es cronológicamente
 *       anterior a «horaPosterior».
 * Post: Ha devuelto la cantidad de tiempo transcurrida entre
 *       «horaAnterior» y «horaPosterior», expresada en
 *       segundos.
 */
int calcularTiempoTranscurrido(const Hora horaAnterior,
                               const Hora horaPosterior) {
    return segundosTotales(horaPosterior)
         - segundosTotales(horaAnterior);
}
```

Horas.
Fichero de implementación horas.cpp

```
/*
 * Pre:  Los valores de los campos horas, minutos y
 *       segundos del parámetro «unaHora» son nulos
 *       o positivos.
 * Post: Ha devuelto la cantidad de tiempo
 *       correspondiente a «h» expresada en
 *       segundos.
 */
int segundosTotales(const Hora h) {
    return h.horas * 3600 + h.minutos * 60
         + h.segundos;
}
```

Horas.
Fichero de implementación horas.cpp

```
/*
 * Pre:  ---
 * Post: Ha escrito el valor de «unaHora» en pantalla, con
 *       el formato "27:59:59" o "03:04:05".
 */
void mostrar(const Hora unaHora) {
    cout << setfill('0');
    cout << setw(2) << unaHora.horas << ":"
         << setw(2) << unaHora.minutos << ":"
         << setw(2) << unaHora.segundos;
    cout << setfill(' ');
}
```

Horas. Ejemplo de uso

```
Hora h1 = {0, 106, 39};
ajustar(h1);
mostrar(h1);
cout << endl;

Hora h2 = sumarSegundos(h1, 3600);
mostrar(h2);
cout << endl;

cout << calcularTiempoTranscurrido(h1, h2)
     << endl;
```

Personas

- Gestionar información relativa a personas:
  - Nombre
  - Apellidos
  - NIF
  - Fecha de nacimiento
  - Estado civil (casado, no casado)
  - Sexo

Representación de personas

A continuación se presentan los ficheros de interfaz e implementación del módulo `persona` en el que se define el tipo `Persona` y un conjunto de funciones básicas para trabajar con los datos de ese tipo.

En el siguiente capítulo se trabajará en la resolución de diversos problemas planteados sobre tablas cuyos elementos son datos del tipo `Persona`.

Para representar la información de una persona, se ha definido el tipo de dato `Persona` como un registro que agrupa sus datos más relevantes: su nombre y apellidos, su [NIF], su fecha de nacimiento, su sexo (hombre o mujer) y su estado civil (soltero o casado).

Para trabajar con datos de tipo \texttt{Ciudadano} se ha optado por definir [dos] funciones básicas [...]:

- [La función `nombreCompleto(p, cadena)`, que facilita el nombre completo de la persona `p`.
- La función `mostrar(p)`, que escribe en la pantalla los datos de la persona `p`.]

Persona. Interfaz

```
#include "nif.h"
#include "fecha.h"

/*
 * Longitudes máximas del nombre y
 * los apellidos de una persona
 */
const int MAX_LONG_NOMBRE = 24;
const int MAX_LONG_APELLIDOS = 24;
```

Persona. Interfaz

```
/*
 * Definición del tipo de dato Persona que
 * representa la información relevante de una
 * persona: nombre y apellidos, número de
 * identificación fiscal, fecha de nacimiento,
 * estado civil y sexo
 */
struct Persona {
      char nombre[MAX_LONG_NOMBRE];
      char apellidos[MAX_LONG_APELLIDOS];
      Nif nif;
      Fecha nacimiento;
      bool estaCasado;
      bool esMujer;
};
```

Persona. Interfaz

```
/*
 * Pre:  ---
 * Post: Ha asignado a «cadena» el nombre
 *       completo de la persona «p».
 */
void nombreCompleto(const Persona p,
                    char cadena[]);

/*
 * Pre:  ---
 * Post: Ha mostrado los datos de la
 *       persona «p» en la pantalla.
 */
void mostrar(const Persona p);
```

Persona. Interfaz

```
/*
 * Pre:  ---
 * Post: Ha devuelto «true» si y
 *       solo si la fecha de
 *       nacimiento del «persona1»
 *       es estrictamente anterior a
 *       la fecha de nacimiento del
 *       «persona2».
 */
bool esMayorQue(const Persona persona1,
                const Persona persona2);
```

Persona. Implementación

```
#include <cstring>
#include <iostream>
#include "persona.h"
using namespace std;

/*
 * Pre:  ---
 * Post: Ha asignado a «cadena» el nombre completo de la
 *       persona «p».
 */
void nombreCompleto(const Persona p, char cadena[]) {
    strcpy(cadena, p.nombre);
    strcat(cadena, " ");
    strcat(cadena, p.apellidos);
}
```

Persona. Implementación

```
/*
 * Pre:  ---
 * Post: Ha mostrado los datos de la persona «p»
 *       en la pantalla.
 */
void mostrar(const Persona p) {
    char marcaGenero = 'o';
    if (p.esMujer) {
        marcaGenero = 'a';
    }

    ...
}
```

Persona. Implementación

```
void mostrar(const Persona p) {
    ...
    char cadenaNombreCompleto[MAX_LONG_NOMBRE
                              + MAX_LONG_APELLIDOS];
    nombreCompleto(p, cadenaNombreCompleto);
    cout << "Persona: " << cadenaNombreCompleto
         << endl;
    cout << "NIF: "; mostrar(p.nif); cout << endl;
    cout << "Nacid" << marcaGenero << " el ";
    mostrar(p.nacimiento);
    cout << endl;
    if (p.estaCasado) {
        cout << "Casad" << marcaGenero << endl;
    }
    else {
        cout << "Solter" << marcaGenero << endl;
    }
}
```

Persona. Implementación

```
/*
 * Pre:  ---
 * Post: Ha devuelto «true» si y solo si la
 *       fecha de nacimiento del «ciudadano1»
 *       es estrictamente anterior a la fecha de
 *       nacimiento del «ciudadano2».
 */
bool esMayorQue(const Persona persona1,
                const Persona persona2) {
    return esAnterior(persona1.nacimiento,
                      persona2.nacimiento);
}
```

Persona. Ejemplos de utilización

```
Persona rey;
strcpy(rey.nombre, "Felipe");
strcpy(rey.apellidos, "Borbón Grecia");
rey.nif.dni = 15;
rey.nif.letra = 'S';
rey.nacimiento.dia = 30;
rey.nacimiento.mes = 1;
rey.nacimiento.agno = 1968;
rey.esMujer = false;
rey.estaCasado = true;
mostrar(rey); cout << endl;
```

Persona. Ejemplos de utilización

```
Persona reinaEmerita
        = { "Sofia", "Grecia Dinamarca",
            {11, 'B'}, {2, 11, 1938},
            true, true
          };
mostrar(reinaEmerita);
cout << endl;
```

Persona. Ejemplos de utilización

```
Persona reinaEmerita
        = { "Sofia",          // nombre
            "Grecia Dinamarca",  // apellidos
            {11, 'B'},        //  NIF
            {2, 11, 1938}, // fecha nacimiento
            true,             // esMujer
            true              // estaCasada
          };
mostrar(reinaEmerita);
cout << endl;
```

Persona. Ejemplos de utilización

```
Persona princesa = {"Leonor",
                    "Borbón Ortiz"};
princesa.nif = {16, 'Q'};
princesa.nacimiento = {19, 6, 2014};
princesa.esMujer = true;
princesa.estaCasado = false;
mostrar(princesa);
cout << endl;
```

Registros

- Permiten a los programadores trabajar con los datos del nuevo tipo conociendo cómo se ha representado el nuevo tipo.
- Esto significa que el efecto de cualquier cambio en la representación del tipo puede extenderse **a todo** el código que haga uso del tipo.

Registros

- ¿Qué pasaría en el código de un proyecto grande si cambiamos la forma de representar personas?

Registro Persona

```
struct Persona {
      char nombre[MAX_LONG_NOMBRE];
      char apellidos[MAX_LONG_APELLIDOS];
      Nif nif;
      Fecha nacimiento;
      bool estaCasado;
      bool esMujer;
};
```

Registro Persona. Cambio 1

```
struct Persona {
      char nombre[MAX_LONG_NOMBRE];
      char primerApellido[MAX_LONG_APELLIDOS];
      char segundoApellido[MAX_LONG_APELLIDOS];
      Nif nif;
      Fecha nacimiento;
      bool estaCasado;
      bool esMujer;
};
```

Registro Persona. Cambio 2

```
struct Persona {
      char nombre[MAX_LONG_NOMBRE];
      char primerApellido[MAX_LONG_APELLIDOS];
      char segundoApellido[MAX_LONG_APELLIDOS];
      Nif nif;
      Fecha nacimiento;
      int agnosMatrimonio;
      bool esMujer;
};
```

Registro Persona. Cambio 3

```
struct Persona {
      char nombre[MAX_LONG_NOMBRE];
      char primerApellido[MAX_LONG_APELLIDOS];
      char segundoApellido[MAX_LONG_APELLIDOS];
      Nif nif;
      Fecha nacimiento, matrimonio;
      int agnosMatrimonio;
      bool esMujer;
};
```

Clases y objetos en C++

- Permiten representar la misma información que los registros.
- *Encapsulamiento:* permiten de forma cómoda ocultar sus campos o *atributos* fuera del módulo en el que se definen.
- *Comportamiento:* requieren de la definición de un mayor número de funciones *(métodos)* para gestionar la información que contienen.
- Facilitan el cambio en la representación de un tipo de datos, minimizando el impacto en el código externo al módulo afectado.

¿Cómo se puede estudiar este tema?

- Repasando estas transparencias
- Trabajando con el código de estas transparencias
  - https://github.com/prog1-eina/tema-11-registros
- Leyendo
  - «Data structures». *Cplusplus.com*. 2000–2017
    - Excepto la sección «Pointers to structures».
    - http://www.cplusplus.com/doc/tutorial/structures/
- Trabajando con los problemas de las clases del 21 y 28 de noviembre