# Capítulo 3
# Estructuración de los datos

## 1. Programación estructurada

Los lenguajes de programación empezaron muy pronto a agrupar instrucciones reutilizables, conocidos como funciones. Las variables siguieron el ejemplo, aunque un poco más tarde.

La matriz se puede utilizar para procesar determinados algoritmos, siempre que los datos a procesar sean de un tipo uniforme (`char`, `int`...). Cuando los datos a procesar contienen diferentes tipos de información, es necesario utilizar varias matrices o una única que utilice un tipo `void*`. Hay que reconocer que esta solución no es recomendable.

En su lugar, definimos estructuras que agrupan varias variables llamadas campos. Estas variables existen en tantas copias como quiera, cada copia se llama instancia.

El lenguaje C++ tiene varias formas compuestas:

– Las estructuras y las uniones, basadas en la C.
– Las clases, que se tratarán en el capítulo dedicado a la programación orientada a objetos.

## 1.1 Estructuras

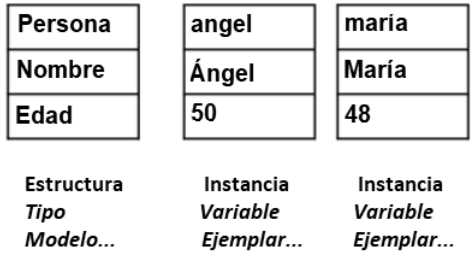
Las estructuras de C++, al igual que las estructuras de C, definen nuevos tipos de datos. El nombre dado a la estructura genera un tipo de datos:

```
struct Persona
{
  char nombre[50];
  int edad;
} ;
```

A partir de esta estructura `Persona`, ahora vamos a crear variables, siguiendo la sintaxis habitual de declaración que asocia un tipo y un nombre:

```
Persona angel, maria;
```

`angel` y `maria` son dos variables de tipo `Persona`. Como son tipos no primitivos (`char`, `int`...), decimos que son instancias de la estructura `Persona`. El término instancia nos recuerda que el nombre y la edad son características propias de cada persona.




Para acceder a los campos de una instancia se utiliza una notación especial:

```
angel.edad = 50; // edad de ángel
printf("%s",maria.nombre); // nombre de maría
```

Esta notación vincula el campo a su instancia.

## 1.2 Construir una estructura

Una estructura puede contener un número ilimitado de campos. Para los lectores que se inician en este tipo de programación y están acostumbrados a las bases de datos, resulta útil comparar una estructura con una tabla de una base de datos.

| C++ | SQL |
|---|---|
| estructura | tabla |
| campo | campo / columna |
| instancia | registro |

Por supuesto, cada campo de la estructura es de un tipo determinado. Puede ser de tipo primitivo (`char`, `int`...) o de tipo estructura. También se pueden obtener construcciones interesantes por composición:

```
struct Direccion
{
  char *direccion1, *direccion2;
  int codigo_postal;
  char* ciudad;
} ;

struct Cliente
{
  char*nombre;
  Direccion direccion;
} ;
```

El operador punto se utiliza como puntero para llegar al campo deseado:

```
Cliente cli;
cli.nombre = "Los molinos de viento";
cli.direccion.ciudad = "Pau";
```

Por último, es posible definir estructuras autorreferenciales, es decir, estructuras en las que uno de los campos es un puntero a una instancia de la misma estructura.

Esto permite construir estructuras dinámicas como listas, árboles y grafos:

```
struct Lista
{
 char* elemento;
 Lista* conjunto;
} ;
```

El compilador no tiene ninguna dificultad para prever esta construcción: conoce el tamaño de un puntero, generalmente 4 bytes, por lo que, para él, el tamaño de la estructura es perfectamente calculable.

## 1.3 Instanciar estructuras

Hay varias formas de instanciar una estructura. Como hemos visto, una estructura genera un nuevo tipo de datos, por lo que la sintaxis clásica para declarar variables funciona muy bien:

```
Persona mateo;
```

## 1.4 Instanciar durante la definición

La sintaxis para declarar una estructura nos reserva una sorpresa; es posible definir instancias inmediatamente después de declarar la estructura:

```
struct Persona
{
  char*nombre;
  int edad;
} angel,maria ;
```

En nuestro caso, `angel` y `maria` son dos instancias de la estructura `Persona`. Por supuesto, es posible utilizar otros modos de instanciación más adelante.

## 1.5 Instanciar reservando memoria

Por último, la instanciación actúa como la asignación de un espacio segmentado para almacenar los campos de la nueva copia. La función `malloc()`, que asigna bytes, hace precisamente eso:

```
Persona* sergio = (Persona*) malloc(sizeof(Persona))
```

Como la función `malloc()` devuelve un puntero a `void`, realizamos un transtyping (`cast`) al tipo `Persona*` para afinar cada lado de la igualdad. El operador `sizeof()` determina el tamaño de la estructura, en bytes.

Para acceder a los campos se utiliza el operador `->`, que sustituye al punto:

```
sergio->edad = 37;
```

Para reservar varias instancias consecutivas –una matriz– hay que multiplicar el tamaño de la estructura por el número de elementos a reservar:

```
Persona* personal = (Persona*) malloc(sizeof(Persona)*5)
```

Para acceder a un campo de una instancia, combinamos la notación anterior con la utilizada para las matrices:

```
personal[0]->nombre = "Pablo";
personal[0]->edad = 41;
personal[1]->nombre = "Blanca";
personal[1]->edad = 27;
```

La función `malloc()` se declara en la cabecera `<memory.h>`, que se debe incluir si es necesario. Este tipo de instanciación ya funcionaba en el lenguaje C, pero el operador `new`, introducido en C++, va más allá.

## 1.6 Instanciación con el nuevo operador

El operador `new` simplifica la sintaxis, ya que devuelve un puntero correspondiente al tipo asignado:

```
Persona*jose = new Persona;
```

No es necesario transtyping, ya que el operador new aplicado a la estructura Persona devuelve un puntero (una dirección) de tipo `Persona*`.

Para reservar una matriz, basta con añadir el número de elementos entre corchetes:

```
Persona*empleados = nueva Persona[10];
```

También en este caso se simplifica la sintaxis, ya que no es necesario especificar el tamaño de cada instancia. El operador `new` lo tiene en cuenta directamente, sabiendo que se aplica a un tipo concreto, en este caso `Persona`.

Simplificar la escritura no es la única ventaja del operador `new`. Si la estructura tiene un constructor (véase el capítulo Programación orientada a objetos sobre las clases), se le llama cuando se instancia la estructura mediante el operador `new`, mientras que la función `malloc()` se limita a reservar memoria. El papel de un constructor, una función "interna" a la estructura, es inicializar los campos de la nueva instancia. Más adelante veremos en detalle cómo funciona.

## 1.7 Punteros y estructuras

Independientemente de cómo se instancie la estructura, mediante `malloc()` o mediante `new`, se accede al campo utilizando la notación de flecha en lugar de la notación de punto, reservada esta última para acceder a una instancia por valor.

El operador & aplicado a una instancia, tiene el mismo significado que para cualquier otra variable, es decir, su dirección.

Si se combina este operador con el acceso a un campo, se puede obtener la dirección de este campo para una instancia determinada. La siguiente tabla resume estos métodos de acceso. Para leerla, consideremos las líneas siguientes:

```
Persona angel;
Persona* daniel = new Persona;
Persona* personal = new Persona[10];
```

| | |
|---|---|
| `angel.edad` | El campo de `edad` relativo a `angel`. |
| `daniel->edad` | El campo `edad` se refiere a `daniel`, que es un puntero. |
| `&angel` | Dirección de `Juan`.<br>Permite escribir `Persona*angel_prime=&angel`. |
| `&angel.edad` | La dirección del campo de `edad` para la instancia angel. |
| `&daniel` | La dirección del puntero `daniel`, que no es la de la instancia. |
| `&daniel->edad` | La dirección del campo de `edad` para la instancia de `daniel`. |
| `personal[2] ->edad` | La edad de la persona en el índice 2 de la matriz (es decir, la tercera casilla a partir de 0). |
| `persona[2]` | La dirección de la persona en el índice 2 de la matriz (es decir, la tercera casilla a partir de 0). |
| `&persona[2] ->edad` | Dirección del campo de `edad` de la persona en el índice 2 de la tabla (es decir, la tercera casilla a partir de 0). |

Observamos que no se ha añadido nada nuevo. Las notaciones siguen siendo las mismas.

## 1.8 Organización de la programación

Cuando se crea una estructura, no es raro verla definida en un archivo de cabecera `.h` que lleva su nombre. Posteriormente, todos los módulos `cpp` que contengan funciones que vayan a utilizar esta estructura, deben incluir el archivo utilizando la directiva `#include`.

### 1.8.1 Inclusión mediante #include

Tomemos el caso de nuestra estructura `Persona`, que se definirá en el archivo `persona.h`:

```
// Archivo: persona.h
struct Persona
{
  char nombre[50];
  int edad;
} ;
```

Cada módulo con extensión `.cpp` que lo utilice debe incluir a su vez este archivo, teniendo en cuenta que se compila por separado del resto de archivos:

```
#include "persona.h"

int main()
{
Persona angel;
  Angel.edad=30;
}
```

### 1.8.2 Protección contra inclusiones múltiples

El sistema de cabeceras da buenos resultados, pero a veces ciertos archivos `.h` se incluyen varias veces, lo que lleva a múltiples declaraciones del tipo `Persona`, lo que penaliza el trabajo del compilador.

Hay dos formas de solucionar este problema. En primer lugar, podemos utilizar una directiva de compilación `#ifndef` seguida de `#define`:

```
#ifndef _Persona
#define _Persona

// Archivo : persona.h
struct Persona
{
  char nombre[50];
  int edad;
} ;
#endif
```

La segunda técnica, más sencilla, consiste en utilizar una directiva específica del compilador, `#pragma once`. Esta directiva, que se coloca al inicio del archivo de cabecera, asegura que el contenido no se incluya accidentalmente dos veces. Si esto ocurre, el compilador recibirá una versión que no contiene la misma definición dos veces, por lo que no obtendremos un error.

La primera técnica puede parecer menos directa, pero es más portable, ya que las directivas `#pragma` (por pragmáticas) dependen de cada compilador. Por supuesto, es probable que se encuentre con ambas en un programa C++ de terceros.

## 1.9 Uniones

Una unión es una estructura especial en la que los campos se solapan. Esta construcción especial permite comprimir los datos y ahorrar considerablemente. Cuando se escribe un campo para una instancia, se sobrescriben los demás, ya que todos los campos tienen la misma dirección. Por lo tanto, el tamaño de la unión corresponde al tamaño del campo más grande.

Las uniones tienen dos tipos de aplicación. En primer lugar, se pueden utilizar para segmentar una estructura de diferentes maneras. Un ejemplo es la estructura `direcciones`, que alberga una unión diseñada para representar diferentes formatos de direcciones de red. Dependiendo de la naturaleza de la red - IP, Apple Talk, SPX - las direcciones se representan de diferentes maneras.