Capítulo 2
De C a C++

1. Programación estructurada

Los lenguajes de programación comenzaron muy temprano a ensamblar instrucciones en forma de grupos reutilizables, las funciones. Las variables naturalmente tomaron el mismo camino, aunque un poco más tarde.

El array permite tratar ciertos algoritmos, siempre que los datos a procesar sean de tipo uniforme (`char`, `int`, etc.). Cuando los datos a procesar contienen información de diferente naturaleza, es necesario utilizar varios arrays o incluso, un solo array utilizando un tipo 'todo vale' `void*`. Hay que admitirlo, esta solución se debe evitar.

En su lugar, definimos estructuras que agrupan varias variables llamadas campos. Estas variables existen en tantas copias como se desee, y cada copia toma el nombre de la instancia.

El lenguaje C++ conoce varias formas compuestas:

– las estructuras y las uniones, dispuestas desde C;

– las clases, que se tratarán en el próximo capítulo.

1.1 Estructuras

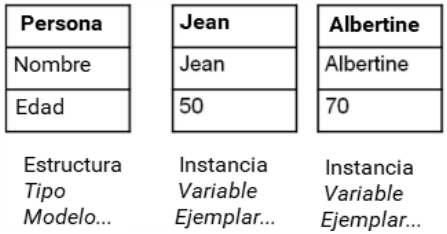
Las estructuras de C++, como las de C, definen nuevos tipos de datos. El nombre que se le da a la estructura genera un tipo de datos:

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

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

```
Persona jean, albertine;
```

`jean` y `albertine` son dos variables de tipo `Persona`. Como se trata de un tipo no primitivo (`char`, `int`, etc.), decimos que se trata de instancias de la estructura `Persona`. El término instancia recuerda que el nombre y la edad son características propias de cada persona.



Usamos una notación particular para alcanzar los campos de una instancia:

```
jean.edad = 50;  // la edad de jean
printf("%s",albertine.nombre); // el nombre de albertine
```

Esta notación vincula el campo a su instancia.

1.1.1 Constitución de una estructura

Una estructura puede contener un número ilimitado de campos. Para el lector que es nuevo en este tipo de programación y que está acostumbrado a las bases de datos, es útil comparar una estructura con una tabla en una base de datos.

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

Cada campo de la estructura es, por supuesto, de un tipo particular. Puede ser de tipo primitivo (`char`, `int`,etc.) o de un tipo estructura. También podemos obtener interesantes construcciones, por composición:

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

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

Se utiliza el operador punto como controlador para llegar hasta el campo deseado:

```
Cliente cli;
cli.nombre = "Los molinos unidos ";
cli.direccion.ciudad = "Villanueva";
```

Finalmente, es posible definir estructuras auto referenciadas, es decir, estructuras de las que uno de los campos es un puntero a una instancia de la misma estructura.

Esto le permite construir estructuras dinámicas, como listas, árboles y gráficos:

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

El compilador no tiene dificultades en considerar esta construcción: conoce muy bien el tamaño de un puntero, generalmente 4 bytes, por lo que para él el tamaño de la estructura es perfectamente calculable.

1.1.2 Instanciación de 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 maria;
```

Instanciación durante la definición

La sintaxis para declarar una estructura reserva una sorpresa: es posible definir instancias inmediatamente después de la declaración del tipo:

```
struct Persona
{
  char*nombre;
  int edad;
} jean,albertine ;
```

En nuestro caso, `jean` y `albertine` son dos instancias de la estructura `Persona`. Por supuesto, es posible a continuación utilizar otros modos de instanciación.

Instanciación mediante reserva de memoria

Finalmente, la instanciación actúa como la asignación de un espacio segmentado para organizar los campos de la nueva copia. La función `malloc()` que asigna bytes, hace exactamente esta tarea:

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

La función `malloc()` devuelve un puntero a `void`, y realizamos un transtipado (`cast`) hacia el tipo `Persona*` con el objetivo de hacer concordar cada lado de la igualdad. El operador `sizeof()` determina el tamaño de la estructura, en bytes.

Entonces se accede a los campos por medio del operador `->` que sustituye al punto:

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

Para reservar varias instancias consecutivas (un array), multiplique 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 los arrays:

```
personal[0]->nombre = "Georgette";
personal[0]->edad = 41;
personal[1]->nombre = "Amandine";
personal[1]->edad = 27;
```

La función `malloc()` se declara en el encabezado **`<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.

1.1.3 Instanciación con el operador new

De hecho, el operador `new` simplifica la sintaxis, porque devuelve un puntero correspondiente al tipo asignado:

```
Persona*josette = new Persona;
```

No se necesita ningún cast porque el operador `new` aplicado a la estructura `Persona` devuelve un puntero (una dirección) de tipo `Persona*`.

Para reservar un array, simplemente agregue el número de elementos entre paréntesis:

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

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

La simplificación de la escritura no es el único avance del operador `new`. Si la estructura tiene un constructor (ver el capítulo de clases sobre este tema), este se llama cuando se instancia la estructura mediante el operador `new`, mientras que la función `malloc()` se contenta con reservar memoria. El rol de un constructor, una función "interna" de la estructura, es inicializar los campos de la nueva instancia. Volveremos para tratar en detalle cómo funciona.

1.1.4 Punteros y estructuras

Independientemente de cómo se haya instanciado la estructura, con `malloc()` o con `new`, se accede al campo usando la notación de flecha en lugar del punto, este último está reservado para el acceso a una instancia por valor.

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

Si este operador se combina con el acceso a un campo, podemos obtener la dirección de este campo para una instancia en particular. La siguiente tabla resume estos métodos de acceso. Para leerlo, consideramos las siguientes líneas:

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

`juan.edad`	El campo `edad` está relacionado con `juan`.
`daniel->edad`	El campo `edad` está relacionado con `daniel`, siendo este último un puntero.
`&juan`	La dirección de juan. Permite escribir `Persona*juan_prime=&juan`.
`&juan.edad`	La dirección del campo `edad` para la instancia `juan`.

`&daniel`	La dirección del puntero `daniel`, que no es el de la instancia.
`&daniel->edad`	La dirección del campo `edad` para la instancia `daniel`.
`personal[2]->edad`	La edad de la persona con índice 2 del array (es decir, la tercera celda partiendo de 0).
`persona[2]`	La dirección de la persona con índice 2 del array (es decir, la tercera celda partiendo de 0).
`&persona[2]->edad`	La dirección del campo `edad` para la persona con índice 2 del array (es decir, la tercera celda partiendo de 0).

Observamos que no ha aparecido nada nuevo. Las notaciones se mantienen consistentes.

1.1.5 Organización de la programación

Cuando se crea una estructura, no es raro verla definida en un archivo de encabezado `.h` con su nombre. Posteriormente, todos los módulos `cpp` que contienen funciones que utilizarán esta estructura deben incluir el archivo a través de la directiva `#include`.

Inclusión con ayuda de #include

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

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