1. Definición de tipos

En C++ existe una palabra reservada (*keyword*) con la cual podemos otorgar un *alias* a un tipo de dato llamada **typedef**. Esta se utiliza para simplificar o clarificar el código.

Ejemplo sin usar **typedef**

```
int vida_jugador;
int defensa_escudo;

void defender(int poder_arma) {
  if(poder_arma > defensa_escudo) {
    vida_jugador -= (poder_arma - defensa_escudo);
  }
}
```

En el ejemplo, observamos que los puntos de vida o energía son enteros. Veamos cómo quedaría con una definición de tipo nueva:

Ejemplo utilizando **typedef**

```
// Se declara "energia_t" como un alias para "int"
typedef int energia_t;

energia_t vida_jugador;
energia_t defensa_escudo;

void defender(energia_t poder_arma) {
  if(poder_arma > defensa_escudo) {
    vida_jugador -= (poder_arma - defensa_escudo);
  }
}
```

Aquí observamos la sintaxis de **typedef**, donde en primer lugar se escribe esta *keyword* y luego el tipo de dato a reemplazar. Se finaliza la declaración con el nuevo nombre para las variables de este tipo. Es usual, pero no obligatorio, que se termine la definición con un **'_t'**, para que el programador sepa que es una definición de tipo.

En este ejemplo es posible señalar dos ventajas. Por un lado, documenta el código de una forma más clara y por el otro, si en una futura implementación los puntos de energía necesitan convertirse en otro tipo de dato (por ejemplo **double**), alcanza con cambiar la definición del tipo de dato.

Este recurso es especialmente útil cuando el tipo de dato original es complejo o riguroso de escribir. Pero especialmente se utiliza para darle nombre a registros, más conocidos en C/C++ como **estructuras de datos**.

Es posible también utilizarlo para declarar arreglos de datos, pero no es una práctica muy usual:

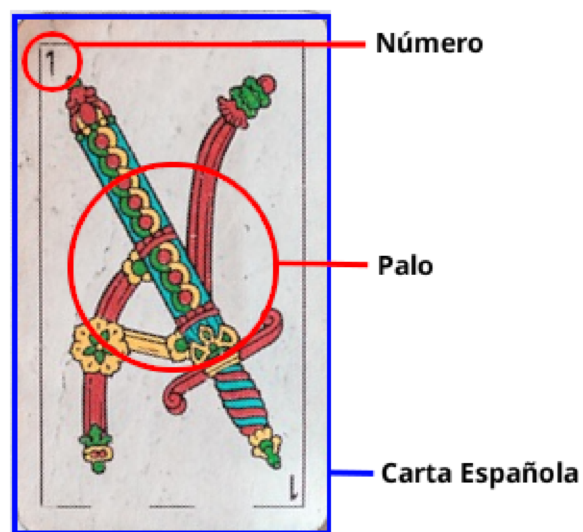
Matías S. Ávalos© CC BY NC SA 4.0

```
typedef double arr10Num_t[10];

arr10Num_t notas; // eq-> double notas[10];
```

2. Estructura de datos

Los registros son una agrupación de datos que conforman una sola entidad. Es decir, son el modelo de datos de algo concreto. Por ejemplo, imagine que se quisiera programar un juego que utilice la baraja española. Cada carta tiene dos datos que la componen: el *palo* y su *número*.



Esto podría codificarse con arreglos independientes (uno para el nombre del palo y otro para los números), pero sería bastante engorroso y difícil de mantener o extender su uso. Es por ello que podríamos agrupar estos dos datos en lo que se conoce en una estructura:

```
struct carta_espanyola_t {
  string palo;
  int numero;
};
```

A cada variable dentro de una estructura se lo conoce como **miembro**. Note que se ha utilizado el nombre `carta_espanyola_t`, ya que en `C++` hay un `typedef` implícito. Es decir que es equivalente al siguiente código:

```
typedef struct {
  string palo;
  int numero;
} carta_espanyola_t;
```


Matías S. Ávalos© CC BY-NC-SA 4.0

2.1. Declarar una estructura

Para poder declarar una variable llamada `sota_basto` de esta estructura se debe hacer –como se haría normalmente– de la siguiente manera:

```
// declaración de la variable:
carta_espanyola_t sota_basto;
```

2.2. Inicialización, asignación y lectura

Podemos inicializar la estructura de dos maneras diferentes. La primera es dando los valores a los miembros directamente y el otro nombrando el miembro de la estructura al que queremos asignarle un valor, siempre encerrado entre llaves '{}':

```
// Método 1: dando simplemente valores:
carta_espanyola_t sota_basto = {"Basto", 10};

// Método 2: nombrando el miembro correspondiente:
carta_espanyola_t sota_basto = {.palo = "Basto", .numero = 10};
```

En ambos casos, es necesario respetar imperiosamente el orden en que fueron declarados los miembros de la estructura. El querer **declarar el número antes que el palo dará un error de compilación**. Aquí se optará siempre por utilizar el segundo método, el cual clarifica el código y es considerado una mejor práctica. En caso de omitir miembros en la incialización estos tomaran un valor `0` o equivalente dependiendo el tipo de dato del miembro.

Para poder acceder y/o modificar un miembro de una variable del tipo estructura se debe utilizar el operador '.' (punto) al igual que en la inicialización nombrada. Tanto para asignar nuevos valores, como para mostrar los mismos.

```
carta_espanyola_t miCarta = {.palo = "Espada", .numero = 1};

cout << miCarta.palo << " - " << miCarta.numero << '\n';

miCarta.palo   = "Basto";  // se modifica el palo
miCarta.numero = 4;        // y el número...

cout << miCarta.palo << " - " << miCarta.numero << '\n';
```

La salida generada por este fragmento de código sería:

```
Espada - 1
Basto - 4
```


Matías S. Ávalos© CC BY NC SA 4.0

2.3. Funciones y estructuras

Las estructuras pueden llegar a contener grandes registros con muchos datos en ellos. Es por ello que siempre se utilizarán referencias como parámetro en las funciones que necesiten operar con los mismos. Esto es debido a una cuestión de *performance* en el manejo de la memoria:

```
void mostrar_carta(carta_espanyola_t &c) {
  cout << c.numero << " de " << c.palo << '\n';
}
```

Al pasar una referencia, es posible modificar el contenido o valor de los miembros de la estructura dada.

Programa 1: Funciones con estructuras

```
1  #include <iostream>
2  using namespace std;
3
4  // Se declara el registro:
5  struct carta_espanyola_t {
6    string palo;
7    int numero;
8  };
9
10 // Se declaran las funciones
11 void mostrar_carta(carta_espanyola_t &c);
12 void cambiar_carta(carta_espanyola_t &c, string palo, int num);
13
14 int main(){
15   carta_espanyola_t miCarta = {.palo = "Espada", .numero = 1};
16   mostrar_carta(miCarta);
17
18   cambiar_carta(miCarta, "Basto", 4);
19   mostrar_carta(miCarta);
20 }
21 // Implementaciones:
22 void mostrar_carta(carta_espanyola_t &c) {
23   cout << c.numero << " de " << c.palo << '\n';
24 }
25
26 void cambiar_carta(carta_espanyola_t &c, string palo, int num) {
27   c.palo = palo;
28   c.numero = num;
29 }
```


Matías S. Ávalos© CC BY-NC-SA 4.0

3. Enumeradores

Los enumeradores son recursos que ayudan a mantener los límites y ayudar al programador a no cometer errores. En el ejemplo del Programa 1, solo existen 4 palos posibles: Basto, Copa, Espada y Oro. Por lo tanto limitar la variable a esos 4 únicos valores es algo deseable. Esto se hace asignando un valor numérico que representa dichos estados (de allí el nombre de *enumerar*). Esto en C++ se hace de la siguiente manera:

```
// BASTO = 0, COPA = 1, etc...
enum palo_t {BASTO, COPA, ESPADA, ORO};
```

Nuevamente, aquí hay un `typedef` implícito. Si no se asignan valores específicos, automáticamente al primer valor constante se le asignará el entero `0` y subsecuentemente el `1`, `2`, etc. En cambio si se asigna un primer valor, los demás se asignarán a partir de este:

```
// BASTO = 1, COPA = 2, etc...
enum palo_t {BASTO = 1, COPA, ESPADA, ORO};
```

Finalmente, es posible asignar un valor totalmente diferente a cada constante:

```
enum palo_t {
  BASTO  = 10,
  COPA   = 20,
  ESPADA = 30,
  ORO    = 40
};
```

La decisión dependerá de las facilidades que otorgue este mecanismo. Se reescribirá el Programa 1 utilizando un enumerador:

Programa 2: Uso de enumeradores

```
1  #include <iostream>
2  using namespace std;
3
4  // BASTO = 0, COPA = 1, etc...
5  enum palo_t {BASTO, COPA, ESPADA, ORO};
6
7  // Se declara el registro:
8  struct carta_espanyola_t {
9    palo_t palo; // palo solo puede ser BASTO, COPA, ESPADA u ORO
10   int numero;
11 };
12
13 // Se declaran las funciones
14 string palo_a_str(palo_t p);
15 void mostrar_carta(carta_espanyola_t &c);
16 void cambiar_carta(carta_espanyola_t &c, palo_t p, int num);
17
18 int main(){
```


Matías S. Ávalos© CC BY NC SA 4.0

```
19    carta_espanyola_t miCarta = {.palo = ESPADA, .numero = 1};
20    mostrar_carta(miCarta);
21
22    cambiar_carta(miCarta, BASTO, 4);
23    mostrar_carta(miCarta);
24  }
25
26  string palo_a_str(palo_t p) {
27    const string PALOS[4] = {"Basto", "Copa", "Espada", "Oro"};
28    return PALOS[p]; // se utiliza el palo como índice (0,1,2 ó 3)
29  }
30
31  void mostrar_carta(carta_espanyola_t &c) {
32    cout << c.numero << " de " << palo_a_str(c.palo) << '\n';
33  }
34
35  void cambiar_carta(carta_espanyola_t &c, palo_t p, int num) {
36    c.palo = p;
37    c.numero = num;
38  }
```

Si tiene dudas sobre los valores de un `enum`, puede utilizar el '`::`' para asegurarse de estar utilizando un valor correcto:

```
miCarta.palo = palo_t::ESPADA;
// ...
cambiar_carta(miCarta, palo_t::BASTO, 4);
```

4. Arreglo de Registros

Una de las ventaja más evidente es el poder agrupar los datos en un solo arreglo de datos. Esto hace que no necesitemos tener dos arreglos independientes para datos relacionados. Es parecido a tener una tabla de datos, donde cada elemento del arreglo es una fila y cada miembro de la estructura una columna.

baraja		
índice	**palo**	**número**
0	BASTO	1
1	BASTO	2
...	...	...
47	ORO	12

Cada registro (fila), es una `carta_espanyola_t`, y `baraja` es el arreglo de 48 cartas. Observe una posible implementación de esto modificado el Programa 2:

Programa 3: Arreglo de cartas

```
1  #include <algorithm> // shuffle
2  #include <iostream>  // cout
```


Matías S. Ávalos© CC BY NC SA 4.0

```
3  #include <random>    // random_device()
4  using namespace std;
5  // BASTO = 0, COPA = 1, etc...
6  enum palo_t { BASTO, COPA, ESPADA, ORO };
7  // Se declara el registro:
8  struct carta_espanyola_t {
9    palo_t palo; // palo solo puede ser BASTO, COPA, ESPADA u ORO
10   int numero;
11 };
12 // Se declaran las funciones
13 string palo_a_str(palo_t p);
14 void inicializar_baraja(carta_espanyola_t baraja[]);
15 void mostrar_carta(carta_espanyola_t &c);
16 void cambiar_carta(carta_espanyola_t &c, palo_t p, int num);
17
18 int main() {
19   // Se declara y carga el arreglo con las cartas:
20   const int CANT_CARTAS = 48;
21   carta_espanyola_t baraja[CANT_CARTAS];
22   inicializar_baraja(baraja);
23
24   // Se muestran ordenadas:
25   cout << "Baraja creada:\n";
26   for (int i = 0; i < CANT_CARTAS; i++) {
27     mostrar_carta(baraja[i]);
28   }
29
30   // Se mezclan las cartas:
31   random_device rand;
32   shuffle(baraja, baraja + CANT_CARTAS, rand);
33
34   // Se muestran las cartas mezcladas:
35   cout << "\n\n" "Baraja mezcalda:\n";
36   for (int i = 0; i < CANT_CARTAS; i++) {
37     mostrar_carta(baraja[i]);
38   }
39 }
40
41 void inicializar_baraja(carta_espanyola_t baraja[]) {
42   for (int p = 0; p < 4; p++) {     // 4 palos
43     for (int n = 0; n < 12; n++) { // 12 cartas por palo
44       // Se *castea p* de int a palo_t (0->BASTO, 1->COPA, etc)
45       baraja[(12 * p) + n].palo   = palo_t(p);
46       baraja[(12 * p) + n].numero = n + 1;
47     }
48   }
49 }
50
51 string palo_a_str(palo_t p) {
52   const string PALOS[4] = {"Basto", "Copa", "Espada", "Oro"};
53   return PALOS[p]; // se utiliza el palo como índice (0,1,2 ó 3)
```


Matías S. Ávalos© CC BY NC SA 4.0

```
54 }
55
56 void mostrar_carta(carta_espanyola_t &c) {
57   cout << c.numero << " de " << palo_a_str(c.palo) << '\n';
58 }
59
60 void cambiar_carta(carta_espanyola_t &c, palo_t p, int num) {
61   c.palo = p;
62   c.numero = num;
63 }
```

En el ejemplo anterior, se "llena" el arreglo **baraja** invocando, en la *línea 22* a la función **inicializar_baraja()**, cargando ordenadamente por palo y número, y cuya implementación se encuentra entre las *líneas 41 y 49*. Luego, es mostrada de esta forma, recorriendo las 48 cartas de la baraja española (líneas 26 a 28). Se utiliza la función **shuffle()** para mezclar la baraja (*línea 32*) y luego mostrar cómo quedó (*líneas 36 y 37*).

Puede ver cómo es posible acceder a los miembros de la estructura dentro del arreglo utilizando primero **'[]'** para seleccionar el elemento del arreglo y luego el operador punto **'.'** para seleccionar el miembro en las *líneas 45 y 46*.

4.1. Arreglos dentro de registros

Una estructura podría tener como miembro un arreglo, incluso un arreglo de registros:

```
struct cancion_t {
  string titulo;
  string artista;
  double duracion;
};

// Cantidad máxima de canciones en un album
const int MAX_TRACKS_ALBUM = 30;

struct album_t {
  string titulo;
  cancion_t track[MAX_TRACKS_ALBUM];
  double precio;
};
```

En este ejemplo, observamos que un álbum musical, contiene un arreglo de registro del tipo canción. Y su utilización sería de la siguiente manera:

```
album_t mas_vendido;
mas_vendido.precio = 125.59;
mas_vendido.titulo = "Thriller"; // nombre del album
mas_vendido.track[3].duracion = 5.96;
mas_vendido.track[3].artista = "Michael Jackson";
mas_vendido.track[3].titulo = "Thriller"; // nombre canción
```


Matías S. Ávalos© CC BY NC SA 4.0

5. Ejercitación

Codifique los siguientes programas utilizando arreglos de registros y funciones en `C++`.

1. Pida ingresar nombre, apellido y DNI de 10 personas. Muestre las 10 personas con sus datos completos ordenados por su apellido en el siguiente formato:

```
<Apellido>, <Nombre>
DNI: <XXXXXXXX>
```

2. Cree un tipo de dato para almacenar números complejos que contenga parte real e imaginaria. Desarrolle dos funciones que puedan mostrarlos en su forma binómica ($a + bi$) y polar ($|m| \pm A^o$).

3. Cree el tipo de dato para almacenar punto en el plano (x,y). Luego, ingrese los puntos `A`, `B`, `C` y `D`. Muestre la suma de los lados de la figura `ABCD`.

4. Cree un tipo de dato para almacenar punto en el espacio (x,y,z) e ingrese 3 puntos. Calcule el perímetro del triangulo formado por los mismos.

5. Cree un tipo dato para una bicicleta que contenga marca, modelo, rodado, precio y cantidad en *stock* (puede ser `0`). Ingrese 10 bicicletas y liste solo las que tengan stock.

6. Ingrese 10 tipos de yerba mate, constituidos por marca, intensidad (suave, mediana y fuerte) y precio. Muéstrelas agrupadas por intensidad.

7. Cree el tipo de dato "libro" que contenga, nombre, isbn y autor. Este último, a su vez contiene nombre, apellido, fecha de nacimiento y nacionalidad. Ingrese los datos de 5 libros y muéstrelos agrupados por nombre del autor.

8. Cree el tipo para jugadores de fútbol que contenga apellido, valor en el mercado (en millones de euros), edad, cantidad de partidos jugados, promedio de goles por partido. Ingrese 10 jugadores y muéstrelos ordenados por valor en el mercado.

9. Ingrese 5 capitales del mundo con su nombre, país, cantidad de habitantes y metros cuadrados. Muéstrelas ordenadas de forma descendente por $habitantes/m^2$

10. Cree un tipo de dato celular constituido por marca, modelo, pulgadas de la pantalla, cantidad de núcleos, RAM y precio. Ingrese 10 celulares y muéstrelos agrupados por marca y ordenados por precio.

Matías S. Ávalos© CC BY NC SA 4.0