Los archivos binarios, como su nombre lo indica, no almacenan texto sino **bytes de información** *en crudo*. Infinidad de formatos utilizan este tipo de información como serían imágenes en diversos formatos como bmp, jpeg, png, etc. como así también son los archivos de música (wav, mp3), video (mkv, mp4, vid, mpeg), incluso los ejecutables generados al compilar nuestro código fuente.

En este caso se utilizará como una forma "fácil" de *serializar* y *deserializar* los datos de una estructura o un arreglo de estructuras para luego poder recuperar los mismos sin tener que preocuparse en el formato o que los datos sean legibles fuera de nuestro software.

1. String vs arreglo de caracteres

Como se ha mencionado anteriormente, en `C` no existe el tipo de dato `string`, simplemente se utilizan arreglos de caracteres (`char[]`). Dada la naturaleza de los *strings* en `C++`, los cuales son de una longitud variable, esto representa una dificultad para almacenar datos estructurados en un archivo binario, ya que para hacerlo de manera eficiente es necesario que todas las estructuras posean un mismo tamaño. Es por ello, que para lograr *serializar* y *deserializar* los datos se optara por hacerlo en arreglos de caracteres fijos.

Para ello, cuando sea necesario utilizar `string`, estos serán reemplazados. Por ejemplo, en la siguiente estructura:

```
struct empleado_t {
  char nombres[70];      // en lugar de string nombres;
  char apellidos[40];    // en lugar de string apellidos;
  int dni;
  int legajo;
  bool es_externo;
  double salario;
};
```

En lo cotidiano, se puede seguir utilizando el dato `string`, pero antes de guardarlos en la estructura debemos *copiarlo* al arreglo de caracteres utilizando la función interna `copy()`, la cual recibe como argumentos el arreglo y la cantidad de caracteres a copiar:

```
1 empleado_t e; // Estructura sin inicializar...
2 // Lectura del string:
3 string nombres = leer_texto("Ingrese nombre(s): ");
4 // Copia de "string" a "arreglo de caracteres"...
5 nombres.copy(e.nombres, nombres.length());
6 // Se pone la terminación del string al final !!
7 e.nombres[nombres.length()] = '\0';
```

La *línea 7* en el anterior fragmento de código es muy importante, ya que `copy()` solo ha copiado el texto que contenía el *string* sin incluir la terminación del mismo, es por ello que al final esta debe ser agregada a mano.

Matías S. Ávalos© CC BY NC SA 4.0

2. Escritura de un archivo binario

A diferencia de los archivos de texto, los binarios deben abrirse agregando el argumento `ios::binary` al momento de abrirlos, lo cual aplica tanto a `ofstream` como a `ifstream`.

```
ofstream arch_escritura;
arch_escritura.open("empleados.data", ios::binary);
```

Para poder escribir datos en crudo (**bytes**) debemos utilizar el tipo de dato `char`. Esta práctica es heredada de muchos años atrás (de `C`) cuando no existían otras maneras u otros tipos de datos más adecuados. Las función estándar `write()` recibe la posición de memoria inicial de la variable que deseamos leer y su tamaño en bytes.

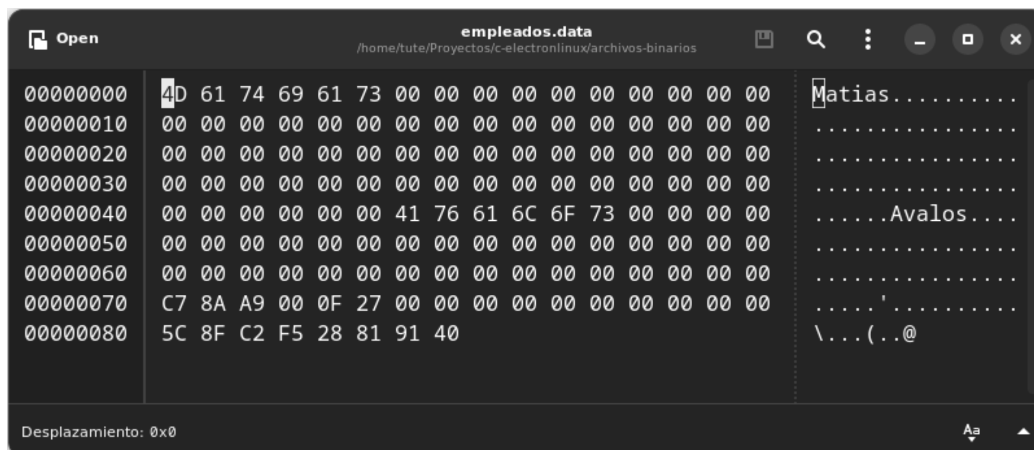
```
empleado_t empleado1 = {
                         .nombres     = "Matias",
                         .apellidos   = "Avalos",
                         .dni         = 11111111,
                         .legajo      = 9999,
                         .es_externo  = false,
                         .salario     = 1120.29
                        };

arch_escritura.write((char *) &empleado1, sizeof(empleado_t));
arch_escritura.close(); // !! Siempre es necesario
```

Como se observa en la *línea 10*, a la dirección de memoria de la variable (obtenida a través del operador `&`) se la *castea* a `char *`, que es el tipo de dato del parámetro de `write()`, a este tipo de datos con el operador `*` (asterisco) se los denomina **punteros**. No se explicará con mucha más profundidad, **es simplemente un tipo de dato que almacena direcciones de memoria**. Y el segundo parámetro recibirá el tamaño en bytes –como se anticipó– con ayuda del operador `sizeof()` sobre el tipo de dato.

Finalmente, en la *línea 11* **se cierra el archivo**, ya que de no hacerlo, es posible que no se guarden los cambios en el mismo o queden *truncos* (perdida de información).

En el archivo `'empleados.dat'` quedará almacenado algo parecido a lo que se ve en la imagen, la cual fue abierta con un software especial para lectura de archivos binarios.



Matías S. Ávalos© CC BY NC SA 4.0

3. Leer de un archivo binario

Una vez *serializado* el archivo binario, se podrá recuperar su información siempre y cuando se utilicen las mismas condiciones, es decir, la misma estructura: con las mismas cantidades de variables en el mismo orden.

Para hacerlo utilizaremos la función `read()` que es análoga a la función `write()` pero utilizando un `ifstream`.

```
ifstream arch_lectura;
arch_lectura.open("empleados.data", ios::binary);

if(arch_lectura.good()) {
  // Se lee y guarda en 'e':
  empleado_t e;
  arch_lectura.read((char *) &e, sizeof(empleado_t));
  arch_lectura.close();
  // Se muestran los datos leidos:
  cout << "Empleado Leg.Nro: " << e.legajo << '\n'
       << e.apellidos << ", " << e.nombres << '\n'
       << "DNI: " << e.dni << '\n'
       << "Sueldo: $ " << e.sueldo << '\n';

  if(e.es_externo) {
    cout << "Empleado de contratación externa.\n";
  } else {
    cout << "Empleado de planta.\n";
  }
}
```

Luego de leer los datos y almacenarlos en la variable `'e'`, simplemente se muestran los datos de la estructura generando una salida parecida a esta:

```
Empleado Leg.Nro: 1111
Avalos, Matias
DNI: 11111111
Sueldo: $ 1129.29
Empleado de planta.
```

Además, existen funciones que nos indican y ayudan a movilizarnos dentro del archivo binario. Para ello se valen del cursor de obtención: *get pointer*.

Función	Descripción	Parámetros/retorno
`seekg()`	Mover el *get pointer*	**offset:** Cantidad de bytes a desplazarnos. **position:** (*Opcional*) Posición de referencia del desplazamiento. Si no se indica es de la posición actual del cursor (`ios::cur`). **retorno:** `void`.
`tellg()`	Obtener posición del *get pointer*	**retorno:** Posición del *get pointer*.

Matías S. Ávalos© CC BY NC SA 4.0

Por ejemplo esto se puede utilizar para saber el tamaño en bytes del archivo:

```
ifstream archivo;
archivo.open("empleados.dat", ios::binary);
// nos desplazamos 0 bytes desde el final del archivo:
archivo.seekg(0, ios::end);
// Pedimos el cursor de obtención (get-pointer):
int tamanyo_archivo = archivo.tellg();

cout << "'empleados.dat' pesa " << tamanyo_archivo << "bytes.\n";

// Volver a la posición inicial del archivo:
archivo.seekg(0, ios:beg);
```

Como se observa en este ejemplo, existen ayudas como `ios::beg` e `ios::end` para referenciar el desplazamiento desde el principio o desde el final del archivo. Así, cuando se solicita desplazarse *0 bytes* tomando de referencia el final del archivo, estamos posicionando el cursor justamente allí. Lo mismo ocurre al volver al principio del archivo al final del ejemplo.

4. Acceso secuencial

Al igual que con los archivos de texto, al escribir o leer, el cursor avanza a medida que se efectúan las acciones. Es decir que lo datos son accedidos o escritos de manera secuencial. En el Programa 1, se observa como acceder o escribir un arreglo de empleados.

Programa 1: Serialización y Deserialización de datos

```
1  #include <iostream>  // cout, cin, cerr
2  #include <algorithm> // swap
3  #include <fstream>   // ifstream, ofstream
4
5  using namespace std;
6
7  struct empleado_t {
8    char nombres[70];
9    char apellidos[40];
10   int dni;
11   int legajo;
12   bool es_externo;
13   double salario;
14 };
15
16 /****  Funciones genéricas de lectura de datos ****/
17 string leer_texto(string msj);
18 int leer_entero(string msj);
19 int leer_entero_entre(string msj, int vmin, int vmax);
20 double leer_numero(string msj);
21 bool leer_booleano(string opcion_verdadera, string opcion_falsa);
22
```


Matías S. Ávalos© CC BY NC SA 4.0

```
23 /**** Funciones manejo de empleados ****/
24 // Mostrar e ingresar empleados:
25 void mostrar_empleado(empleado_t &e);
26 empleado_t ingresar_empleado();
27 // Guardar y recuperar empleados del archivo binario:
28 int deserializar_empleados(empleado_t empleados[], int max_size);
29 void serializar_empleados(empleado_t empleados[], int cant);
30
31 int main() {
32   // Declaración de arreglo de empleados
33   const int MAX_EMPLEADOS = 50;
34   empleado_t empleados[MAX_EMPLEADOS];
35
36   // Recuperar empleados guardados (de-serialización)
37   int cant_emp = deserializar_empleados(empleados,MAX_EMPLEADOS);
38   if(cant_emp < MAX_EMPLEADOS) {
39     // Ingresar nuevos empleados:
40     int empleados_nuevos = leer_entero_entre("Empleados a
     ingresar: ", 0, MAX_EMPLEADOS-cant_emp);
41
42     for(int i = 0; i < empleados_nuevos; i++) {
43       empleados[cant_emp] = ingresar_empleado();
44       cant_emp++;
45     }
46   }
47
48   // Mostrar empleados cargados:
49   for(int i = 0; i < cant_emp; i++) {
50     mostrar_empleado(empleados[i]);
51   }
52
53   // Guardar empleados (serialización)
54   serializar_empleados(empleados, cant_emp);
55 }
56
57 void serializar_empleados(empleado_t empleados[], int cant) {
58   // Archivo de datos
59   ofstream dataf;
60   dataf.open("empleados.dat", ios::binary);
61
62   if (dataf.good()) {
63     // Escribir los elementos del arreglo empleados en el archivo
64     for(int i=0; i < cant; i++) {
65       dataf.write((char *) &empleados[i], sizeof(empleado_t));
66     }
67     // ¡IMPORTANTE! No olvidar cerrar el archivo:
68     dataf.close();
69   } else {
70     cerr << "¡¡NO PUDO GUARDARSE LA INFO EN 'empleados.dat'!!\n";
71   }
72 }
```


Matías S. Ávalos© CC BY-NC-SA 4.0

```
73  int deserializar_empleados(empleado_t empleados[], int max_emp){
74    ifstream dataf;
75    dataf.open("empleados.dat", ios::binary);
76
77    int cant_e = 0;
78    if(dataf.good()) {
79      cout << "\nRecuperando datos de los empleados...\n\n";
80      // Obtener la longitud/tamaño del archivo.
81      dataf.seekg(0, ios::end);
82      int tam_archivo = dataf.tellg();
83      dataf.seekg(0, ios::beg); // cursor al princpio del archivo
84      // Mientras no sea el fin del archivo o fin del arreglo
85      while(dataf.tellg() < tam_archivo and (cant_e < max_emp)) {
86        dataf.read((char *)&empleados[cant_e], sizeof(empleado_t));
87        cant_e++;
88      }
89      dataf.close();
90      cout << cant_e << " empleados recuperados...\n\n";
91    } else {
92      cout << "\nNo hay datos de empleados...\n\n";
93    }
94    return cant_e;
95  }
96
97  empleado_t ingresar_empleado() {
98    empleado_t e;
99    cout << "\n*** Ingrese nuevo empleado ***\n";
100   // Leemos un string y lo copiamos al arreglo de caracteres:
101   string nombres = leer_texto("Nombre(s): ");
102   nombres.copy(e.nombres, nombres.length());
103   e.nombres[nombres.length()] = '\0';
104
105   string apellidos = leer_texto("Apellido(s): ");
106   apellidos.copy(e.apellidos, apellidos.length());
107   e.apellidos[apellidos.length()] = '\0';
108
109   e.dni        = leer_entero_entre("DNI: ", 1000000, 99999999);
110   e.legajo     = leer_entero_entre("Legajo: ", 1000, 9999);
111   e.es_externo = leer_booleano("Contratado externo.", "Empleado
       de planta.");
112   e.salario    = leer_numero("Salario: ");
113
114   cout << "\n*** Empleado cargado ***\n";
115   return e;
116 }
117
118 void mostrar_empleado(empleado_t &e) {
119   cout << "\n*** Empleado Leg. Nro " <<  e.legajo << " ***\n\n"
120        << e.apellidos << ", " << e.nombres
121        << "\nDNI: " << e.dni
122        << "\nSalario: $" << e.salario << '\n';
```


Matías S. Ávalos© CC BY-NC-SA 4.0

```
123    if(e.es_externo) {
124      cout << "Contratado externamente.\n\n";
125    } else {
126      cout << "Empleado de planta.\n\n";
127    }
128    cout << "******************************\n";
129  }
130  /* Implementación de las funciones de lectura de datos.... */
131  // ...
```

Se observa que en la función `serializar_empleados()` (*líneas 57 a 72*), se declara una variable del tipo `ofstream`, y luego de abrir el archivo (*línea 60*) se recorre el arreglo de empleados, con la cantidad indicada en el parámetro `cant`, llamando a la función `write()` pasando la dirección de memoria secuencialmente (*líneas 64 a 66*), es decir, de cada elemento del arreglo. Por último, se cierra el archivo para asegurar la correcta escritura en el mismo (*línea 68*).

Por otro lado, en la función `deserializar_empleados()`, que es la primera en ser invocada (*línea 37*) e implementada en las *líneas 74 a 96*, podemos observar que se declara una variable del tipo `ifstream` para abrir y obtener los datos del archivo '`empleados.dat`' y devuelve la cantidad de empleados encontrados dentro del mismo. Para ello, se toma el arreglo `empleados` y se los recorre almacenando en él los datos encontrados en el archivo (*líneas 86 a 89*). Esto ocurre mientras no se llegue al final del archivo consultando la posición del cursor obtenido previamente (*líneas 82 a 83*) o, bien, si se alcanzó el máximo tamaño del vector. En caso de no existir el archivo, se mostrará la leyenda *"No hay datos de empleados..."*, pero si existiera, devolvería la cantidad de empleados presentes en el mismo.

5. Acceso aleatorio

A través de las funciones `seekg()` y `tellg()`, se observo que es posible desplazar el cursor de obtención (*get pointer*) en el archivo cuando este es de lectura (`ifstream`), pero además también podemos mover el cursor de escritura (*put pointer*) con las funciones `seekp()` y `tellp()` con archivos de escritura (`ofstream`). De esta forma podemos tomar aleatoriamente un dato del archivo y/o modificarlo.

Utilizando otro programa que toma los mismos datos del archivo '`empleados.dat`' del Programa 1, se mostrará como hacer otro programa que modifique el dato del salario de un empleado tomado aleatoriamente.

Programa 2: Modificación aleatoria

```
1  #include <iostream>
2  #include <algorithm>
3  #include <fstream>
4  using namespace std;
5
6  struct empleado_t { /* atributos de  empleado_t */ };
7  // Entrada de datos
8  int leer_entero(string msj);
```


Matías S. Ávalos© CC BY-NC-SA 4.0

```
9  int leer_entero_entre(string msj, int vmin, int vmax);
10 double leer_numero(string msj);
11 // Lectura y modificación de empleados
12 void mostrar_empleado(empleado_t &e);
13 int obtener_cantidad_empleados();
14 empleado_t obtener_empleado(int pos_empleado);
15 void modificar_empleado(empleado_t &e, int pos_empleado);
16
17 int main() {
18   // Obtener cantidad de empleados en el archivo:
19   int empleados = obtener_cantidad_empleados();
20   cout << "\nEmpleados en 'empleados.dat': " << empleados <<'\n';
21   // Obtener empleado a modificar salario:
22   int  n_empleado = leer_entero_entre("Empleado a modificar: ",
      1, empleados);
23   empleado_t e = obtener_empleado(n_empleado-1);
24   // Se muestra el empleado:
25   mostrar_empleado(e);
26   // Se modifica el valor del salario
27   e.salario = leer_numero("Ingrese nuevo salario: ");
28   // Se escribe el empleado modificado en el archivo
29   modificar_empleado(e, n_empleado-1);
30 }
31
32 int obtener_cantidad_empleados() {
33   ifstream dataf;
34   dataf.open("empleados.dat", ios::binary);
35   int cant_empleados = 0;
36   if(dataf.good()) {
37     dataf.seekg(0, ios::end);
38     // [bytes totales] / [bytes que ocupa cada empleado]
39     cant_empleados = dataf.tellg() / sizeof(empleado_t);
40     dataf.close();
41   }
42   return cant_empleados;
43 }
44
45 empleado_t obtener_empleado(int pos_empleado) {
46   ifstream dataf;
47   dataf.open("empleados.dat", ios::binary);
48   empleado_t e;
49   e.legajo = -1;
50   if(dataf.good()) {
51     dataf.seekg(sizeof(empleado_t)*pos_empleado, ios::beg);
52     dataf.read((char *)&e,sizeof(empleado_t));
53     dataf.close();
54   } else {
55     cerr << "\nError de lectura.\n";
56   }
57   return e;
58 }
```


Matías S. Ávalos© CC BY NC SA 4.0

```
59
60 void modificar_empleado(empleado_t &e, int pos_empleado) {
61   ofstream dataf;
62   // AGREGAR 'ios::in' para no truncar el archivo existente!!
63   dataf.open("empleados.dat", ios::in | ios::binary);
64   if(dataf.good()) {
65     dataf.seekp(sizeof(empleado_t)*pos_empleado, ios::beg);
66     dataf.write((char *)&e, sizeof(empleado_t));
67     dataf.close();
68     cout << "\nEmpleado modificado...\n";
69   }
70 }
71 // Demás implementaciones de funciones...
```

En este nuevo programa, la obtención de la cantidad de empleados se calcula en la función `obtener_cantidad_empleados()` utilizando el tamaño del archivo y dividiéndolo por lo que ocupa un empleado en el mismo (*línea 39*).

En la función `obtener_empleado()` el archivo es accedido aleatoriamente, ubicando el cursosr con la función `seekg()` en la posición indicada por el parámetro `pos_empleado`. Esto se logra multiplicando lo que ocupa un empleado en el archivo por la posición (desplazamiento), tomando el cursor desde el principio del archivo (*línea 51*).

Luego de modificarse el empleado (*línea 27*) se pasa por referencia a la función `modificar_empleado()` junto a la posición donde se encontraba originalmente (*línea 29*). En la implementación de la función (*líneas 60 a 70*), el archivo es abierto pasando dos atributos diferentes en el modo `ios::in` y `ios::binary` (*línea 63*). Si bien se declara un `ofstream` debemos aplicar el atributo `ios::in` para habilitar también la lectura, de esta forma evitamos que se sobre escriban los datos existentes en el archivo como normalmente ocurriría. Luego, se mueve el cursor de escritura al igual que se hizo en `obtener_empleado()`, pero utilizando `seekp()`.

6. Ejercitación

Codifique en C++ los siguiente programas utilizando arreglos, registros y archivos binarios.

1. Desarrolle un software que administre hasta 100 estudiantes, cuyos datos serán almacenados en un archivo binario `'estudiantes.db'` y de los cuales se quiere registrar: nombres, apellidos, legajo de 6 dígitos, año, división y promedio general. Los mismos deben ser almacenados en orden alfabético y en cada ejecución permitir agregar, quitar estudiantes y listarlos de mejor a peor promedio general. Verifique que no haya dos legajos iguales.

2. Agregue al software anterior la capacidad de mostrar a los estudiantes ordenados por apellido y legajo. También la posibilidad de modificar los datos de un estudiante buscado por su legajo (este dato no puede ser modificado).

3. Implemente un software que utilizando las funciones de acceso aleatorio (sin utilizar un arreglo) levante los datos de `'estudiantes.db'` y reordene el contenido del archivo por un criterio a elección: apellidos, legajo o promedio.

Matías S. Ávalos© CC BY NC SA 4.0