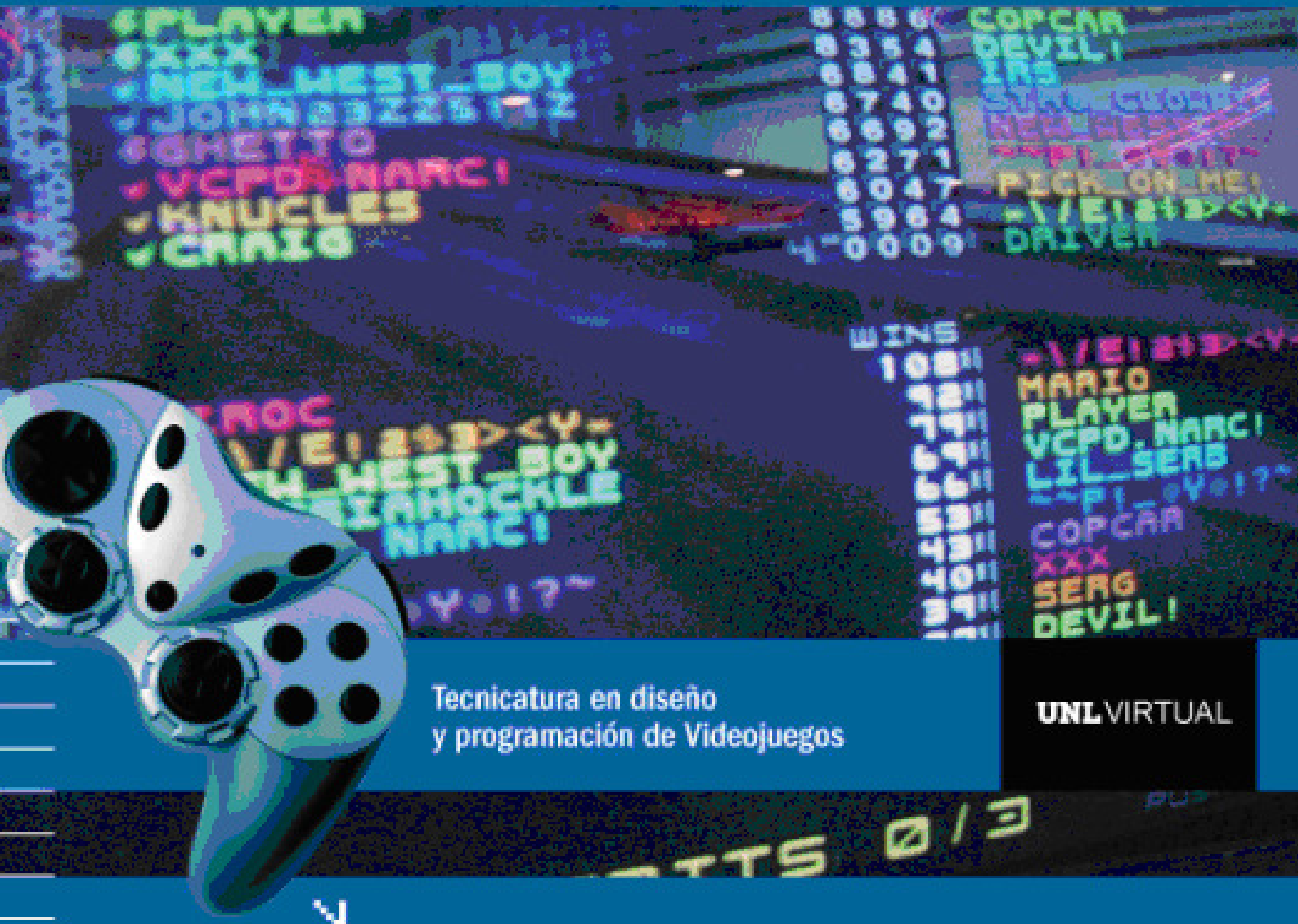
UNIVERSIDAD NACIONAL DEL LITORAL
FACULTAD DE INGENIERÍA Y CIENCIAS HÍDRICAS



Tecnicatura en diseño
y programación de Videojuegos

UNLVIRTUAL

Introducción a la programación

Unidad Temática Número 6

Cstrings, Punteros y Archivos

Objetivo: Introducir el tipo de datos string de C. ver los fundamentos de Punteros. Ver el manejo de Archivos de C++. Trabajar tanto con Archivos de texto como archivos binarios

Temas: Cstring. Punteros. Referencia y Desreferencia. Operaciones con Punteros. Pasaje de Punteros a Funciones. Punteros y Arreglos. Entrada y Salida de cstring. Funciones de Biblioteca. Archivos. Operaciones sobre archivos. Uso de archivos en videojuegos

UNIDAD 6

Cstrings, Punteros y Archivos

Introducción

Muchos de los datos que empleamos en programas deben almacenarse usando esta estructura: apellidos, nombres, direcciones, denominaciones, códigos, etc.; todos estos ejemplos corresponden a cadenas de caracteres o strings.

C++ posee una biblioteca estándar heredada de C; en ella encontraremos varias funciones para procesar cadenas de caracteres o strings al estilo de lenguaje C. También existe en C++ una biblioteca más moderna donde se almacena una clase de objetos strings con métodos para operarlos en forma más directa y sencilla. Entonces cabe plantear una serie de preguntas para aclarar la coexistencia de 2 elementos que parecen superponerse en el lenguaje de programación C++.

¿Cuántos tipos de strings existen en C++?
En C++ coexisten 2 tipos de Strings: los strings heredados de C que se operan como cadenas de caracteres finalizados con el caracter '/0' que denominaremos **cstrings** y los **objetos strings** que se crean a partir de la clase de igual nombre.

¿Por qué emplear cstrings del lenguaje C en C++?
Una de las razones es la de mantener la compatibilidad de código C en C++. También ciertas operaciones de C++ con cstrings no pueden realizarse a través de objetos de la clase string y deben operarse como cstrings. Es el caso de la representación de los nombres de archivos físicos, la cual debe hacerse siempre a través de un cstring.

¿Qué es un cstring?
Las cadenas de caracteres o cstrings constituyen unidades de información, formadas por secuencias de caracteres que se procesan en C/C++ como arreglos de caracteres que finalizan con el carácter nulo ('\0').

Punteros y cstrings
C++ dispone para operar cstrings de varias funciones predefinidas (heredadas del lenguaje C). La mayoría de estas funciones operan con parámetros csrtings y devuelven strings a través de punteros. Por este motivo estudiaremos algunos elementos básicos de punteros en C++ para comprender mejor las funciones con cstrings.

Punteros en C/C++

Un puntero es una variable que representa un valor numérico correspondiente a la ubicación física de un elemento determinado del programa. Ese valor numérico asignado a un puntero también es conocido como *dirección de memoria* del elemento. Dicho elemento puede constituir en C++:

- Un dato simple
- Una estructura de datos
- Una función
- Una variable de cualquier tipo
- Un objeto

Es decir que podemos emplear punteros para referenciar datos o estructuras más complejas y también administrar bloques de memoria asignados dinámicamente. Su empleo hace más eficiente la administración de recursos del programa y su ejecución.

Muchas funciones predefinidas de C++ emplean punteros como argumentos e inclusive devuelven punteros. Para operar con punteros C++ dispone de los operadores **&** y ***** .

Una de las ventajas más interesantes del uso de punteros es la posibilidad de crear variables dinámicas; esto es, variables que pueden crearse y destruirse dentro de su ámbito, lo cual permite optimizar el uso de recursos disponibles en un programa.

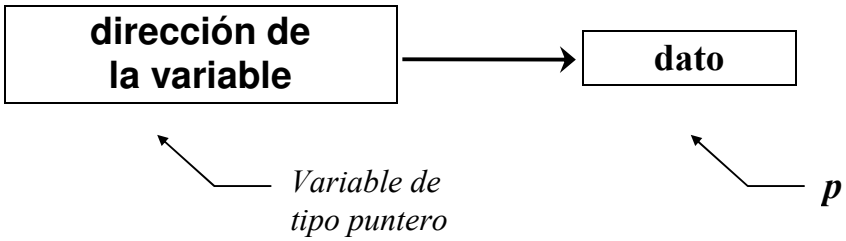
Definición de Punteros

Para declarar un puntero, C++ emplea el operador * anteponiéndolo a la variable puntero. Además se debe indicar el tipo de dato al que apuntará esta variable.

```
tipo  *variable_puntero;
```

Ejemplo:

```
int *p; // se declara a x como variable puntero a un entero
```



En el ejemplo anterior, la variable puntero **p** contiene la dirección donde se almacenará un dato de tipo entero, pero ***p** es una expresión nemotécnica que corresponde a un entero; por lo tanto podemos emplear ***p** en cualquier proposición o expresión de C++ que requiera la presencia de un entero.

En otras palabras: ***p** es una expresión que representa un valor entero almacenado en la dirección de memoria contenida en **p**.

Relación entre los operadores de referencia & y de desreferencia *

Como vimos en el epígrafe anterior, es posible declarar una variable puntero a través del operador * y asignarle la dirección de memoria de algún elemento del programa.

Pero si declaramos una variable simple cualquiera: ¿Cómo determinamos su dirección?. En C++ es posible conocer la dirección de cualquier variable a través del *operador dirección* que representamos con: **&**. Ese dato solo podremos asignarlo a una variable que admita direcciones, es decir a una variable puntero:

```
int  z=8;  // declaramos e inicializamos con 8 una variable entera z
int *p;  // declaramos un puntero p a un entero
p= &z;  // asignamos la dirección de z al puntero p
cout<< *p <<endl;  // informamos 8, el dato apuntado por p
cout<< p;  // informamos la dirección de memoria contenida en p
```

Nota: la asignación de la dirección de **z** a **p** fue posible pues **z** es entero y **p** fue declarada como puntero a un entero.

En resumen: el operador **&** seguido de un operando permite obtener la dirección de dicho operando; por el contrario, el operador * permite obtener el dato ubicado en la dirección asignada a una variable puntero. De allí que se denomine a estos operadores:

- **&:** operador dirección o referencia
- ***:** operador indirección o desreferencia;

La constante NULL

Si una variable puntero no ha sido inicializada apunta a una dirección aleatoria. Si la lógica del programa requiere que una variable puntero no apunte a ningún dato ni elemento del programa podemos asignarle la constante **NULL.**

```
float *q = NULL;
cout << q;   //devuelve la dirección nula 00000000
```

El objeto de definir con **NULL** a una variable puntero tiene que ver con controles que suelen ser útiles en algoritmos que emplean estructuras de datos con punteros.

Nota: la mayoría de los compiladores C++ devuelven un entero en base hexadecimal de 8 cifras para definir una dirección de memoria.

Operaciones con Punteros

Las variables puntero pueden ser operadas aritméticamente; esto nos permite referenciar una nueva dirección de memoria.

La Aritmética de Punteros es similar a la aritmética usada para todas las variables ordinales, después de todos las direcciones de memoria tienen un orden de menor a mayor.

Esa nueva dirección después de la operación dependerá del tipo de dato al que apunta la variable puntero. Por ejemplo:

```
int *p;
```

p+=3; /*La nueva dirección de **p** supera a la anterior en 12 bytes, pues al sumar 3 estamos desplazándonos la cantidad de bytes correspondientes a 3 enteros (12 bytes) */

```
double *r;
```

r+=2; /*La nueva dirección de **r** supera a la anterior en 16 bytes pues al incrementar en 2 su dirección, la estamos desplazando la cantidad de bytes correspondientes a 2 datos de tipo double (16 bytes)*/

Ampliando estos conceptos podemos resumir las siguientes operaciones como válidas para las variables de tipo puntero:

a) Se puede asignar a una variable puntero la dirección de una variable no puntero.

```
float x, *p;
.....
p=&x;
```

b) A una variable puntero puede asignarse el contenido de otra variable puntero si ambas variables son compatibles (ambos punteros apuntan al mismo tipo de dato).

```
int *u, *v;
.....
u=v;
```

Nota: estamos asignando el contenido de v que es una dirección a un número entero.

c) A un puntero es posible asignarle el valor NULL (el puntero no apunta a dirección de memoria alguna).

```
int *p;
p=NULL;    //Dirección nula: 00000000
```

d) Es posible sumar o restar una cantidad entera **n** a una variable puntero. La nueva dirección de memoria obtenida difiere en una cantidad de bytes dada por: **n** por el tamaño del tipo apuntado por el puntero.

```
int *p;
.....
p+=4; //la dirección de p se ha incrementado 16 bytes
p-=1; //La dirección de p se decrementó en 4 bytes.
```

e) Es posible comparar dos variables puntero.

u<v u>=v u==v u!=v
u==NULL

f) Una variable puntero puede ser asignada con la dirección de una variable creada dinámicamente a través del operador **new**.

```
float *q; // se declara q como puntero a un float
q= new float; /* se asigna a q la dirección de una
nueva variable */
*q=4.1513; //se almacena un float en la dirección de q
```

new asigna un espacio de memoria para contener a un float y devuelve la dirección donde reservo ese espacio de memoria.

Paso de punteros a una función

Uno de los objetivos de pasar punteros a una función es poder modificar los parámetros empleados en llamada, es decir, permite efectuar un pasaje por referencia. Otro objetivo es la eficiencia en términos de ejecución y de recursos: al pasar direcciones evitamos crear nuevas variables locales en la función y efectuar las copias (asignaciones) de parámetros, lo cual permite lograr una ejecución más rápida del programa.

Al pasar un puntero como parámetro de una función se pasa la dirección del parámetro; cualquier cambio que se efectúe en los datos ubicados en esa dirección, se reflejarán en el bloque de llamada y en la propia función.

Esto es radicalmente diferente al pasaje de parámetros por valor donde las direcciones de los argumentos de llamada son diferentes a las direcciones de los argumentos formales.

En el ejemplo siguiente se ingresan 2 datos enteros en 2 variables x e y, y mediante el pasaje de las direcciones de estas 2 variables podemos intercambiar sus valores:

```
#include <iostream>
using namespace std;

void intercambio(int *p,int *q);

int main(int argc, char *argv[]) {
  int x,y;
  cout<<"Primer dato:\n x=" ; cin>>x;
  cout<<"Segundo dato:\n y=" ; cin>>y;
  intercambio(&x,&y);
  cout<<"\n\nDespues de llamar a la funcion intercambio(x,y);"
cout<<"\n x="<<x<<"  y="<<y;;
  return 0;
}
void intercambio(int *p,int *q)
{ int aux=*p; *p= *q; *q=aux;
}
```

La salida correspondiente será:

```
  Primer dato:
  x= 34
```

```
Segundo dato:
y= 99;
Despues de llamar a la funcion intercambio(x,y);
x= 99  y= 34
```

En este caso los parámetros actuales x e y se han modificado pues en la función se ha trabajado con sus direcciones.

En el caso de pasaje de un parámetro de tipo array a una función debemos considerar que el nombre de un arreglo representa a la dirección del primer elemento del arreglo. Por esto, no es necesario emplear el símbolo & en la llamada a una función con un parámetro actual de tipo arreglo.

```
int   x[6]={5,9,12,45,41,11};

func(x); /*Llamada a una función empleando como parámetro la
dirección de inicio del arreglo x, es decir: &x[0] */
```

Punteros y arreglos lineales

En C++ los punteros y los arreglos están íntimamente relacionados. Las declaraciones de arreglos que hemos estudiado pueden plantearse a través de punteros logrando mayor eficiencia en la ejecución del programa.

Como dijimos, el nombre de un arreglo representa la dirección del primer elemento del arreglo, es decir, el nombre es un puntero al inicio de esa estructura.

```
int   x[6]={5,9,12,45,41,11};
```

Esto significa que en el arreglo lineal **x** del ejemplo, la dirección de su primer componente puede ser referenciada con el propio identificador del arreglo **x**, o también con **&x[0]**.

Para referenciar al segundo elemento del arreglo podemos hacerlo de 2 formas equivalentes: **&x[1]** y **x+1.**

```
cout << x+1 << endl;     /* obtenemos como salida la
dirección del elemento 1 del arreglo. Por ejemplo: 0x000ffee
*/

cout << &x[1] << endl;  /* otra forma de obtener como salida
la dirección del elemento 1 del arreglo: 0x000ffee   */
```

O sea que, la dirección del elemento i-ésimo de un arreglo **x** será **x+i-1** o bien **&x[i-1]**.

Para expresar el contenido del elemento i-ésimo de un arreglo, podemos emplear también dos formas: **x[i-1]** y ***(x+i-1)**

Observemos el siguiente programa C++ donde se obtienen y muestran en la salida las direcciones de memoria de cada componente del arreglo **x** :

```
#include <iostream>

using namespace std;

int main(int argc, char *argv[]) {
     int x[]={12,13,14,15,16};
     for (int i=0; i<5; i++)
     cout<<&x[i]<<endl;
```

```
    return 0;
}
```

La salida que se obtiene es similar a la siguiente:

```
0x22ff50
0x22ff54
Ox22ff58
0x22ff5c
0x22ff60
```

Obsérvese que las direcciones de memoria de los 6 componentes del arreglo **x** (en base hexadecimal) saltan de 4 en 4. Esto es debido a que cada elemento del arreglo es de tipo **int** y requiere 4 bytes de memoria. Si el arreglo fuera de elementos de tipo **float** las direcciones también saltarán de 4 en 4 (cada número de tipo **float** ocupa 4 bytes). Para el tipo **double** el salto sería de 8 bytes.

Obsérvese a continuación 4 maneras distintas de asignar el sexto y último elemento del arreglo **x** al segundo elemento de dicho arreglo:

```
x[1] = x[5] ;
x[1] = *(x+5) ;
*(x+1) = x[5]  ;
*(x+1) = *(x+5) ;
```

Entonces, de acuerdo con el concepto de que el nombre de un arreglo representa a la dirección de su primer elemento, podemos declarar al arreglo lineal **x** de la manera siguiente

```
int *x;
```

La diferencia con la declaración **int x[6]={5,9,12,45,41,11}**; se basa en que ***x** no reserva un espacio de memoria para todos los elementos del arreglo. Es necesario definir un bloque de memoria para alojar la estructura. Esto es posible si usamos las funciones de biblioteca de C: **malloc()** y **sizeof()**.

```
x = (int *) malloc(6*sizeof(int));
```

La expresión anterior asigna a **x** un puntero a un bloque de memoria cuya cantidad de bytes está indicada en el argumento (24 bytes en el ejempo). La fusión de tipos **(int *)** es necesaria pues la función **malloc()** es de tipo void. De este modo reservamos la cantidad de bytes requeridas en un bloque único de memoria para almacenar el arreglo de 6 elementos enteros.

Importante: si en la declaración de un arreglo se incluye la inicialización entre llaves de los elementos del mismo, se lo tiene que declarar como arreglo y no se puede emplear el operador *.

Caracteres

Hemos visto el tipo char para definir variables de tipo carácter. Una constante carácter en C++ se representa por cualquiera de los 256 símbolos del código ASCII encerrados entre apóstrofos.

```
char v=´A´;
```

En el ejemplo anterior se asigna el carácter 65 (la letra 'A') del código ASCII a la variable **v**. Para C++ una constante carácter es un entero (el valor ASCII de dicho carácter), por ello es posible realizar operaciones con variables de tipo char, no admitidas en otros lenguajes de programación. Obsérvense los ejemplos siguientes considerando que v tiene asignado 'A':

```
v++;  //asigna 'B' (ASCII 66) a la variable v
```

También son caracteres muchos símbolos que no tienen una representación gráfica definida: salto de línea, tabulación, escape, etc.

```
char s=´\n´;//asigna un salto de línea (ASCII 10)a s
```

C++ dispone de algunas funciones para operar con datos de tipo char.

Constantes *cstrings*

Las constantes de tipo cstring en C++ se expresan encerrando la secuencia de caracteres entre comillas. Son ejemplos de cstrings:

```
"Alvez Julio"    "----------"
"San Martín 3328-3000 Santa Fe"  "0342-4565551"
```

Un constante cstring puede tener un único caracter. No debe confundirse esta constante como de tipo char, las cuales se representan encerradas por apóstrofos.

"A" es una constante cstring

'A' es una constante char

Las constantes cstrings pueden emplearse en los objetos de flujo de salida cout y pueden incluirse en ellas caracteres especiales para manipular la salida

```
cout<<"Programar en C++\n\nes interesante."
```

Obtendrá como salida:

Programar en C++

es interesante.

Declaración e inicialización de *cstrings*

Para declarar cstrings en C++ se debe utilizar un arreglo de caracteres o un puntero a caracter:

```
char nombre[]= "Martinelli Julián";
char *direc= "San Martín 3328-3000 Santa Fe";
```

En el caso de que una función utilice como parámetros un cstring, se debe plantear en su prototipo el parámetro formal de tipo cstring como puntero a char usando el operador * obligatoriamente.

```
void muestra_cadena(char *s)
{ cout<<s<<'\n';// muestra la cadena s
}
```

Es importante conocer algunos aspectos básicos de punteros y su relación con arreglos para manejar cstrings.

Al declarar una variable cstrings se debe usar la sintaxis de la declaración de arreglos de caracteres en C++:

```
char  palabra[12]={´I´,´n´,´t´,´e´,´r´,´n´,´e´,´t´};
```

aunque esa declaración es poco usual, pues es posible asignarle a la variable cstring el valor inicial completo:

```
char  frase[20]="Ciberespacio";
```

Estas dos declaraciones e inicializaciones han asignado la información de ambos cstrings de la forma siguiente.

Palabra[0]	'I'
Palabra[1]	'n'
Palabra[2]	't'
Palabra[3]	'e'
Palabra[4]	'r'
Palabra[5]	'n'
Palabra[6]	'e'
Palabra[7]	't'
Palabra[8]	'\0'

frase[0]	'C'
frase[1]	'i'
frase[2]	'b'
frase[3]	'e'
frase[4]	'r'
frase[5]	'e'
frase[6]	's'
frase[7]	'p'
frase[8]	'a'
frase[9]	'c'
frase[10]	'i'
frase[11]	'o'
frase[12]	'\0'

Las cadenas de caracteres en C++ finalizan con el carácter nulo '\0' (ASCII 0). Este caracter actúa como señal o bandera para indicar el fin de la secuencia de caracteres que forman el cstring. Por lo tanto se debe especificar siempre un carácter más (como mínimo) de la cantidad que posee la cadena *visible* que se desea almacenar.

¿Qué ocurre con lo elementos declarados y no definidos en la inicialización del cstring?

Usualmente la dimensión propuesta en la declaración de un cstring supera a la cantidad de elementos asignados. Por ejemplo en la declaración:

```
char  palabra[12]="Internet"
```

se establecen 12 caracteres para `palabra[]` y solo se emplean 8; el resto de los elementos dimensionados se asignan con el caracter nulo, o poseen valores al azar (dependiendo del compilador).

No se puede inicializar un cstring fuera de la declaración

C++ no admite ciertas operaciones con cstrings que involucren a la entidad completa, como es el caso de la asignación:

```
char  frase[20];//declara un cstring de 20 char
```

```
frase="Ciberespacio"; //error!
```

Pero veremos que existen funciones que permiten realizar fácilmente estas tareas.

Entrada/Salida de cstrings

El ingreso de datos para variables cstrings se puede realizar a través del flujo **cin,** pero debemos considerar que este comando considera a ciertos caracteres como delimitadores o separadores. Por ejemplo, el carácter espacio en blanco ' ', la tabulación, el carácter de fin de línea y el de fin de archivo.

Si pretendemos leer con **cin** un cstring que incluya el carácter ' ' (espacio en blanco), el dato a asignar será truncado en el lugar donde se encuentre dicho espacio. Por ello C++ dispone de las funciones **gets**( ) y **puts**( ) cuyos prototipos están definidos en la biblioteca **stdio.h .**

Las función **gets**( ) admiten el espacio en la lectura de un cstring.

gets( ): acepta un cstring desde el dispositivo estándar de entrada hasta encontrar un carácter de fin de línea y lo asigna a la variable indicada como argumento.

puts( ): envía al dispositivo de salida estándar la variable cstring indicada como argumento y reemplaza al carácter nulo con un carácter de nueva línea ('\n').

Recordemos que el flujo **cout** a diferencia de **puts**, no agrega un carácter de próxima línea y permite enviar a salida cualquier expresión además de cstrings.

```
char  u[18],v[18];
//Se ingresa por teclado "Espacio digital" en la variable v
cin >> v;
```

Espacio Digital <ENTER>

pero solo se asigna "*Espacio"* a v

```
cout << v; // se obtiene como salida:
```

Espacio

Si se vuelve a ingresar por teclado el cstring "*Espacio digital"* en la variable u

```
gets(u);
```

Espacio Digital <ENTER>

// Se asigna "*Espacio digital*" a u

puts(u);/* se obtiene "*Espacio digital*" como salida y se produce un avance de línea */.

Espacio Digital

Una función importante para operar el ingreso de cstrings a una variable mediante la lectura desde algún dispositivo es **cin.getline( )**. Esta función admite 3 parámetros: la variable de tipo arreglote caracteres donde alojar el cstring, la cantidad de caracteres a asignar y un parámetro de tipo char que hace de delimitador de la cadena.

```
cin.getline (char *var,int largo, char limite='\n');
```

por defecto el delimitador de la cadena a asignar a *var* es el carácter de avance de línea. Por ejemplo:

```
char  u[18];
cin.getline( u, 18);
```

La primer línea del código anterior declara un cstring **u**; la segunda lee desde la consola una línea de texto de hasta 18 caracteres. Si la entrada fuera de menor longitud lee hasta el carácter de avance de línea. Si en cambio empleamos:

```
char  u[18];
cin.getline(u,18,'+');
cout<<u<<endl;
```

E ingresamos por consola:

```
Programar en C++ es interesante
```

La salida debido al objeto `cout` será:

```
Programar en C
```

porque **`cin.getline(u,18,'+')`** cortará la entrada antes del caracter delimitador '+'.

Cómo definir un arreglo de cstrings

Una de las maneras de plantear en C++ un arreglo de cstrings es definir una matriz de caracteres: el primer índice nos ubica el carácter inicial de cada cstring':

```
char z[5][18]={"Santa Fe","Entre Ríos","Córdoba","Salta",
"Chaco"};
for (int i=0; i<5; i++)
     puts( z[i] ); //muestra un cstring por línea
puts z[2]; //obtiene "Córdoba" en la salida
```

Funciones de biblioteca para manejar cstrings

En la biblioteca **string.h** se encuentran varias funciones predefinidas del lenguaje C que pueden emplearse en todo programa C++ para operar cstrings. Veamos algunas de ellas. Observará que se emplea el tipo **size_t** para declarar enteros; este tipo se halla definido en la biblioteca <stdef.h> incluido en la biblioteca <string.h>, y corresponde a entero sin signo: **unsigned long**

También observará la presencia de la etiqueta **const** en algunos parámetros cstring; esto indica que la cadena de caracteres no será modificada en la función.

`int strcmp(const char *c1, const char *c2)`

Compara las cadenas y devuelve un entero menor que cero si **`c1`** está antes alfabéticamente que **`c2`** , devuelve cero si las cadenas son iguales; y devuelve un entero mayor que cero si **`c2`** está antes.

```
     char a[25]; char b[25];
     cout<< "Ingrese la primer cadena de caracteres:";
```

```
gets(a);
cout<< "Ingrese la segunda cadena de caracteres:";
gets(b);
if (strcmp(a,b)==0)
      cout<<"Ambas cadenas coinciden";
else  if (strcmp(a,b)<0)
         cout<<a<<" esta antes que "<<b;
      else
         cout<<b<<" esta antes que "<<a;
```

---

int strncmp(const char *c1, const char *c2, size_t n)

Compara los primeros **n** caracteres de las cadenas empleadas como parámetros. Devuelve un entero menor que cero si los primeros n caracteres de **c1** forman una subcadena que está antes alfabéticamente que los primeros n caracteres de **c2** , devuelve cero si las subcadenas son iguales; y devuelve un entero mayor que cero si la subcadena de **c2** está antes que la de **c1**.

```
if strncmp(a,b,8)==0
    cout<<"Tienen los primeros 8 caracteres iguales\n";
  else
    cout<<"Los primeros 8 caracteres no coinciden\n";
```

---

char *strcat(char *c1,const char *c2)

Concatena (agrega) el cstring indicado como segundo parámetro (**c2**) al final de la variable cstring indicado como primer parámetro (**c1**). La función devuelve el contenido de **c1**.

```
char a[20]="Internet";
char b[20]=" es la red de redes";
strcat(a,b);  //Concatena en a ambos cstrings
puts(a);  //muestra: Internet es la red de redes
```

---

char *strncat(char *c1,const char *c2, size_t n)

Concatena (agrega) los **n** primeros caracteres de **c2** a la variable **c1**. La función devuelve la dirección de **c1**. El tipo **size_t** representa un entero positivo.

```
char x[20]="Internet";
char y[20]=" es la red de redes";
strncat(x,y,10);
//agrega a x los primeros 10 caracteres de y
puts(x);  //muestra: Internet es la red
```

---

size_t strlen(const char *c)

Devuelve un entero con la longitud actual de la cadena argumento (sin incluir el carácter de terminación nulo).

```
char x[20]="Internet";
cout<<"La cantidad de caracteres del cstring ";
cout<<x<<" es:";
cout<<strlen(x);  //muestra el número 8
```

---

char *strstr(const char *c1, const char *c2)

Devuelve un puntero al inicio de la primer ocurrencia de c2 en c1. Si no encuentra c1 devuelve la dirección nula NULL.

```
char x[20]="Internet es la red de redes";
char y[20]="la red";
cout<<strstr(x,y)  //muestra: la red de redes
```

---

char *strchr(const char *c1,char c)

Devuelve un puntero al inicio de la primer ocurrencia del carácter c en c1. Devuelve NULL si no ocurre.

```
char x[20]="Internet es la red de redes";
puts(strchr(x, 'd'));  //muestra: d de redes
```

---

char *strlwr(char *c)
char *strupr(char *c)

Ambas funciones convierten la cadena argumento en minúsculas y mayúsculas respectivamente.

```
char x[20]="Internet";
cout<< strlwr(x);  //muestra: internet
puts(strupr(x));  //muestra: INTERNET
```

También podemos emplear las funciones **tolower(carácter), toupper(carácter)** pero solo aceptan un caracter como argumento y no un cstring. Estas 2 funciones se hallan prototipadas en el archivo **ctype.h** y convierten el carácter argumento a minúscula y mayúscula respectivamente.

---

char *strcpy(char *c1, const char *c2)

Esta función copia los caracteres de **c2** a **c1** y devuelve un puntero a c1. Podemos emplear esta función para asignar una variable cstring a otra.

```
char x[20]="Internet es la red de redes";
char y[20]="la red";
strcpy(x,y);
puts(x);  //muestra: la red
```

---

char *strncpy(char *c1, const char *c2, size_t n)

Esta función copia **n** caracteres de **c2** a **c1** y devuelve un puntero a **c1**. Si n es menor que la longitud de la cadena destino (**c1**), esta conserva el resto de los caracteres iniciales; si n es mayor que **c1** su longitud aumentará para dar cabida a los n caracteres, pero debe agregarse el carácter nulo **'\0'** para completar el cstring.

```
char x[25]="Pedro es programador";
char y[25]="Pablo es ingeniero";
strncpy(x,y,5); /* copia los 5 primeros caracteres
de y en x */
puts(x);  //muestra: Pablo es programador
```

Conversión de cstrings a otros tipos

Es posible emplear cstrings para capturar información que debe ser procesada en formato numérico. Por lo tanto es útil disponer de medios para convertir una cadena de tipo cstring a alguno de los tipos numéricos empleados en C++. Por ejemplo la cadena “124” no puede ser tratada numéricamente pues se trata de un cstring formado por los caracteres ‘1’, ‘2’ y ‘4’. C++ provee funciones para convertir cstrings y obtener (si es posible) el valor numérico correspondiente.

- **atoi(s):** devuelve un **int** correspondiente al cstring s
- **atol(s):** devuelve un **long** correspondiente al cstring s
- **atof(s):** devuelve un **float** correspondiente al cstring s

Si la conversión no fuera exitosa, estas funciones retornan cero.
Estas funciones se hallan en la librería **stdlib.h** pero muchos compiladores C++ reconocen sus funciones sin necesidad de incluirla.
Ejemplo:

```
char s[4];
cout<<"Ingrese el anio:";
gets(s);
int anio= atoi(s);
while (anio==0)
 {cout<<"Ud. debe ingresar un numero para el anio:";
  gets(s);
  anio= atoi(s);
 }
```

Esta forma de ingresar información nos permite evitar errores en tiempo de ejecución causados por la lectura de datos.en nuestro programa.

Archivos

Los Archivos en C/C++ se pueden manejar de dos formas diferentes:

Como texto o como binario.

Los archivos de texto son aquellos que contienen cadenas de caracteres.

Los archivos binarios son los que contienen bytes.

Aunque en el fondo un caracter es un byte (al menos en la codificación ASCII) la diferencia no está tanto en lo que contiene el archivo sino en la forma de tratarlo.

Por ejemplo, si decimos que x.txt es un archivo de texto y lo abrimos como tal, las funciones de C/C++ esperaran que haya líneas de caracteres separadas por un retorno de carro ('\n').

En cambio si abrimos x.txt como archivo binario, no esperará líneas de caracteres sino que tomará al archivo como un gran vector de bytes, que ira leyendo de a x cantidad de bytes según le vayamos pidiendo.

Archivos en C++

Vamos a explicar como se acceden a los archivos en C++, similarmente a string, existen dos formas diferentes una para C y otra para C++, ambas formas conviven, pero a diferencia de los cstring que se siguen usando en casos especiales, para archivos es lo mismo usar una u otra.

Como es lo mismo usar una u otra, el conjunto de C++ es mucho más versátil y simple, por ese motivo vamos a ver ese.

Para declarar un archivo, hay que declarar una variable del tipo stream con alguno de los siguientes:

ifstream cuando se quiere abrir un archivo para lectura
ofstream cuando se quiere abrir un archivo para escritura
fstream cuando se quiere hacer ambas cosas a la vez con un archivo

por ejemplo:

```
ifstream archlect;
ofstream archescr;
fstream archivo;
```

el segundo paso es indicarle cual es el archivo al que queremos acceder, para eso se usa open

por ejemplo:

archlect.open(“x.txt”);
También se puede hacer estas dos cosas de un solo paso al declarar el stream:

ifstream archlect(“a.txt”);
ofstream archescr(“b.txt”);
fstream archivo(“c.txt”);

Al abrir el archivo el nombre es obligatorio, sino no sabría el programa a cual nos estamos refiriendo, la cadena del nombre debe ser un cstring, y puede tener el path absoluto como relativo.

Nota: el path es la ubicación del archivo con respecto a la raíz del disco (en cuyo caso es absoluto) o a la posición donde esta el programa que se está ejecutando (en cuyo caso es relativo).
Ver: http://es.wikipedia.org/wiki/Ruta_%28inform%C3%A1tica%29

Además de informarle el nombre, podemos informar otras características de cómo queremos que abra el archivo, para ello se usan los siguientes indicadores o banderas:

ios::app añade al final del archivo (siempre se agrega al final)
ios::ate añade al final del archivo (se agrega al final, pero si uno cambia el punto para agregar sigue en el nuevo lugar, con app ignora el cambio)
ios::binary abre el archivo en forma binaria (en forma de texto se abre por defecto)
ios::in abre el archivo para lectura
ios::out abre el archivo para escritura
ios::trunc abre el archivo y si existe lo borra y comienza desde cero.

Por ejemplo si queremos abrir el archivo b.bmp en binario para lectura podemos:

ifstream archlect;

archlect.open(“b.bmp”, ios::binary);

o

ifstream archlect(“b.bmp”, ios::binary);

Estas banderas se pueden combinar con el operador binario OR (|) por ejemplo:

fstream arch(“b.bmp”, ios::in | ios::binary);

Ahora ya podemos operar sobre el archivo.

Una vez finalizada la operación sobre el mismo hay que cerrar el archivo con close por ejemplo:

```
arch.close();
```

Esta acción nos libera arch para ser usada para otro archivo (a través de un nuevo open).

Operaciones sobre Archivos

Habíamos dicho que dependiendo de la forma en que abrimos el archivo (binario o texto) era la forma que podíamos operar sobre él. Veamos cada una de ellas:

Archivos de texto

Las operaciones sobre archivos de textos son las mismas que para cin y cout, >> y <<. Con la diferencia que en vez de usar las palabras reservadas cin y cout usamos nuestra variable.

Por ejemplo para escribir a un archivo:

```
ofstream archescr(“b.txt”);

archescr<<”Esta es una línea de texto”<<endl;
archescr<<”Esta es otra línea de texto”<<endl;
```

También se pueden escribir variables que son automáticamente convertidas a texto por ejemplo:

```
int x=10;
ofstream archescr(“b.txt”);

archescr<<”el resultado de restar:”<<x<<” menos 5 es:”<<x-5;
```

Funciona exactamente igual a cout solo que en vez de salir por pantalla se escribe en el archivo. Es más, cout y cin son casos particulares de stream.

Entonces para de leer desde un archivo se puede usar >> con las mismas reglas de cin:

```
char s[200];
int a;

ifstream archlect(“a.txt”);

archlect>>s; //pasa el string a s hasta que encuentra un separador como espacio
archlect>>a; //introduce un numero en a (si es que hay uno en el lugar que esta leyendo)
```

En definitiva, funciona como si se le tecleara al cin todo el archivo de texto.

Por supuesto, también funciona con getline()

Archlect.getline(s,200);

Archivos Binarios

Los archivos binarios son secuencia de bytes, lo que signifiquen esos bytes es problema de nosotros, por ejemplo si suponemos que cada byte es un char estamos tratando al archivo como si fuera de texto.

Ya vimos como abrir un archivo binario:

fstream arch("b.bmp", ios::binary);

Para escribir en el archivo se usa el método write cuyos parámetros son la dirección de memoria donde están los bytes y la cantidad de los mismos por ejemplo si queremos escribir un entero:

int n=35;

arch.write(&n,sizeof(n));

Como ven la dirección se toma como si fuera un puntero, y para la cantidad (aunque sabemos que un int tiene 4 bytes) usamos una función llamada sizeof que devuelve cuantos bytes ocupa el argumento de la misma.

Para hacer lo inverso, o sea, leer desde un archivo se usa read:

int n;

arch.read(&n,sizeof(n));

cout<<n; //mostrara en pantalla 35 (si es el mismo archivo)

Las lecturas y escrituras se hacen en forma secuencial, es decir, si escribes:

```
int n=35;
int f=9;

arch.write(&n,sizeof(n));
arch.write(&f,sizeof(f));
```

Los primeros cuatro bytes del archivo tendrán la representación en bytes del int 35 y los segundos cuatro bytes tendrán la representación del int 9.

Lo mismo con la operación inversa:

```
arch.write(&n,sizeof(n));
arch.write(&f,sizeof(f));
```

asignará los primeros cuatro bytes a n y los segundos cuatro bytes a f recuperando los valores con los que fueron grabados.

Lo importante es que si queremos obtener los mismos valores que teníamos al guardar, debemos leer en el mismo orden en que escribimos.

Moverse dentro de los archivos. Acceso Aleatorio.

Nos podemos mover dentro de los archivos para escribir o leer porciones del archivo sin necesidad de leer o escribir las porciones anteriores.

Para esto se usan cuatro métodos, dos para escribir y dos para leer, y de esas dos una es para ubicar donde estamos y otra es para posicionarnos en donde queramos, veámoslas:

Para leer desde un lugar específico de un archivo:

Para esto se usa seekg la cual tiene dos parámetros para pasarle, el primero es la cantidad de bytes y el segundo es desde donde los cuenta. Existen 3 lugares desde donde contar indicados por los siguientes modificadores:

```
ios::beg  desde el comienzo del archivo
ios::cur  desde la posición actual
ios::end  desde el final del archivo
```

Por ejemplo, si queremos leer un entero a partir del décimo byte podemos hacer:

```
fstream arch(“b.bmp”, ios::binary);

int n;

arch.seekg(10,ios::beg);
arch.read(&n,sizeof(n));
```

La versión para escribir en cualquier lugar del archivo es seekp que funciona exactamente igual solo que para escritura, en nuestro ejemplo escribimos un entero de valor 20 a partir del décimo byte:

```
fstream arch(“b.bmp”, ios::binary);

int n=20;

arch.seekp(10,ios::beg);
```

```
arch.write(&n,sizeof(n));
```

Como se estarán imaginando se puede leer y escribir en lugares diferentes del mismo archivo a la vez (ya que son métodos diferentes que manejan posiciones diferentes) siempre y cuando lo hayamos abierto para lectura y escritura a la vez. También podemos movernos en forma negativa (hay que tener cuidado de no terminar en la posición -1 del archivo), es decir podemos movernos 10 bytes antes del final haciendo arch.seekg(-10,ios::end);

Los otros dos métodos sirven para el caso que queramos recordar donde estuvimos escribiendo, estas nos devuelven la posición desde el inicio:

```
unsigned long p,u;//nos aseguramos un tamaño que entre por si es muy grande el archivo

ofstream arch("b.bmp", ios::binary);

int n=20;

arch.seekp(10,ios::beg);
p=arch.tellp();
arch.write(&n,sizeof(n));
u=arch.tellp();

cout<<"posición antes de escribir"<<p<<"  posición después de escribir"<<u;
```

Como tenemos dos métodos para posicionarnos en el archivo, uno para lectura y otro para escritura, tenemos dos métodos para conocer la posición donde se va a escribir (tellp) y donde se va a leer (tellg).

Más métodos

Ya hemos visto open, close, write, read, seekg, seekp, tellg y tellp, veamos algunos más:

put(char) sirve para stream de salida, envía un carácter.

getline(char *, int n, char c='\n') lee una cadena desde el archivo hasta que alcanza los n caracteres o encuentra el carácter c, sino se especifica c este será el final de línea ('\n').

eof() verifica si se llegó al final del archivo, devuelve 0 (falso) sino se llegó.

clear() cada vez que se produce una condición de error, el error permanece hasta que se usa clear, por ejemplo, si llegamos al final del archivo, eof() nos devolverá verdadero, si usamos seekg() por ejemplo para volver dentro del archivo eof() seguirá dando verdadero hasta que se use clear().

fail() indica que ha ocurrido un error en el flujo de caracteres, pero que los caracteres no se han perdido.

bad() indica que ha ocurrido un error que ocasiona la pérdida de datos.

good() indica que no hay error de ningún tipo (eof, fail o bad).

flush() sirve para vaciar el buffer hacia el archivo, esto ocurre porque normalmente la escritura en archivo se hace de forma asincrónica, es decir, en el momento que se envía un dato no se guarda, normalmente se espera un tiempo prudencial por si llegan más datos y guardarlos a todos de una sola vez, esto es así porque la grabación es una operación lenta que incluye el movimiento de partes móviles tanto para uno como para 100 datos, por lo tanto en vez de mover todo 100 veces se previere esperar un poco por los 100 datos y mover todo una vez.

Más modificadores pueden encontrarlos en Wikibooks:

http://es.wikibooks.org/wiki/Programaci%C3%B3n_en_C%2B%2B/Streams

Y esto es lo básico para manejar archivos.

¿Como usamos archivos?

Ahora lo interesante que es lo que podemos hacer con los archivos.

Archivos de texto

Con los archivos de texto podemos hacer lo mismo que hacemos con la pantalla y el teclado. En el caso de los videojuegos se suelen usar para guardar valores de inicialización para poder alterar ciertos valores dentro de los mismos sin necesidad de recompilar el videojuego.

La razón de usar archivos de texto es porque nos interesa que quien lo modifique (o sea un humano) no tenga problemas en leerlo, un posible archivo de inicialización para nuestro ejemplo de antiaéreo sería:

----Inicio de Archivo----

#Inicialización programa antiaéreo versión 1.0b

#aviones indica la cantidad de aviones que comienzan en el primer nivel
aviones=3

#frecavi indica la probabilidad por décima de segundo de aparecer un avión
# 20 significa que aparecerá un avión cada 2 segundos en promedio
frecavi=20

#frecabom indica la probabilidad por décima de segundo que avión suelta una bomba
# 10 significa que lanza una por segundo
Frecabom=10

----Fin de Archivo----

Como ven solo hay que leer línea por línea y buscar las palabras claves y obtener sus valores, si encontramos un # ignoramos lo que viene después porque es un comentario o aclaración (por supuesto esto es a gusto del programador dejar poner aclaraciones).

Básicamente se usan archivos de texto en los juegos si queremos que puedan ser fácilmente modificados por otras personas.

Archivos Binarios

Los archivos binarios son todos los archivos (incluso los de texto) es la forma básica de abrir un archivo, después de todo, en el fondo son una serie larga de bytes.
Con archivos binarios en los videojuegos podemos hacer frecuentemente dos cosas: salvar o cargar posiciones en el juego para que en otra oportunidad se siga jugando desde donde estábamos y la otra cosa es obtener datos que necesita el programa para funcionar y que no están en el fuente del mismo como ser música, dibujos, animaciones, etc.

¿Cómo salvar posiciones en un videojuego?

Salvar una posición es salvar todos los valores de las variables necesarias para jugar en ese momento del juego, se guardan en un orden con write y obviamente se deben obtener en el mismo orden con read. Las variables pueden ser unas pocas, o muchas, pueden ser simples o muy complejas.

Por ejemplo, en la batalla naval se deberían guardar, las matrices con los mapas, y al menos a quien le toca tirar la próxima vez. Son pocas variables pero tienen muchos datos.

Y esto nos lleva a:

¿Cómo hacemos para guardar struct y arreglos?

Bueno, una primera aproximación sería usar bucles for para guardar y leer elemento por elemento de un arreglo. Pero write y read escriben y leen de a bloque por lo que se puede escribir y leer un arreglo de un solo paso, por ejemplo:

int a[10];

```
ofstream arch("b.bmp", ios::binary);

arch.write(a,10*sizeof(int));

arch.close();
```

Como ven solo usamos un write, el primer parámetro del write, como recordaran, es la dirección de memoria donde se encuentra el bloque a escribir, si mal no recuerdan de arreglos y punteros, el nombre del arreglo sin corchetes es la dirección al primer elemento del mismo y es por eso que en este caso no usamos el operador &.

El segundo parámetro indica la cantidad de bytes a guardar desde la dirección de memoria dada. Para ello calculamos la cantidad de elementos (10) por el tamaño de cada uno de ellos (sizeof(int)) y con eso obtenemos el tamaño de todo el arreglo.

Con los struct sucede algo parecido, aquí la ayuda nos la da sizeof() ya que esta función también nos devuelve el tamaño ocupado por un struct, por ejemplo podemos hacer:

```
struct ficha{
   char nombre[30];
   int edad;
}

struct ficha f;

ofstream arch("b.bmp", ios::binary);

arch.write(&f,sizeof(f));

arch.close();
```

Como ven, al darnos siazeof() el tamaño del struct es muy sencillo guardarlos o leerlos, en el caso que fuera un arreglo de struct sería similar a nuestro primer ejemplo de arreglos:

```
struct ficha{
   char nombre[30];
   int edad;
}

struct ficha f[10];

ofstream arch("b.bmp", ios::binary);

arch.write(f,10*sizeof(f));

arch.close();
```

Con esto ya pueden guardar y leer cualquier tipo de estructura y arreglo que se les presente.

Usar datos externos al programa

Es muy común en los videojuegos tener sonido e imágenes, sin embargo esos sonidos y esas imágenes a menos que sena muy simples no están incorporadas a los archivos fuentes del programa.

Además este tipo de datos son generados normalmente por programas externos y por personas expertas en su rama, es decir, la música la elaboran los músicos, probablemente con instrumentos, y los gráficos los elaboren los artistas que pueden usar tanto una computadora como una hoja de papel. Sea como sea estos elementos terminan siendo archivos en algún formato conocido tanto de sonido como de imágenes.

Para ampliar sobre formatos de archivos pueden consultar la Wikipedia:

http://es.wikipedia.org/wiki/Formato_de_archivo_inform%C3%A1tico

Aunque existen bibliotecas que abstraen el manejo de archivos comunes, es interesante ver como se arman internamente los archivos de datos, para eso vamos a ver el tipo de archivo bmp que usa el paint, no es de por si el mejor formato de archivo pero es uno de los más simples para ver:

http://es.wikipedia.org/wiki/BMP

Como ven, los primeros dos bytes sirven para identificar el tipo de archivo independientemente de su nombre, como son dos char que deben tener la cadena BM podríamos leerlos como tales:

```
ifstream arch(“b.bmp”, ios::binary);

char bm[3];

arch.read(bm,2); //solo leemos dos bytes

bm[2]=0; //colocamos el terminador nulo a la cadena para poder comparala

if(strcmp(bm,”BM”)==0) es un archivo bmp;
```

Los siguientes cuatro bytes es el tamaño del archivo, obviamente un entero.

```
int tamarch;

arch.read(tamarch,sizeof(int));
```

Con esto obtenemos el tamaño del archivo que también podemos comparar con el real para confirmar que realmente estamos en archivo bmp (podría ser casualidad el BM del principio) o que no está dañado.

Los siguientes son cuatro bytes que están reservados, los leemos solo para avanzar al siguiente dato:

```
int reservado;

arch.read(reservado,sizeof(int));
```

Luego tenemos el inicio de los datos que es un entero de cuatro bytes:

```
int inicio;

arch.read(inicio,sizeof(int));
```

Con este valor ya podríamos ir directamente a tomar los datos haciendo seek(inicio,ios::beg), pero nos falta aún algunos valores más para poder interpretarlos correctamente como por ejemplo el ancho y el alto de la imagen para poder distinguir donde termina la primera línea y comienza la segundo.

Con esto ya es suficiente como ejemplo, todos los archivos de formato tiene normalmente un pedazo del mismo donde se encuentran los valores para armar e interpretar correctamente los datos, este pedazo de archivo se conoce normalmente como cabecera, ya que comúnmente se encuentra al comienzo del archivo.

Obviamente uno puede generar su propio formato si quisiera que los datos de imágenes y sonido no se encuentren fácilmente disponibles para cualquiera.

Bibliografía

Ing. Horacio Loyarte. Cstrings [UNL-FICH] [2008]