

## CURSO DE C C++

-----

## Introducción:

Este tutorial o curso pretende ser un rápido repaso a las cuestiones más importantes del lenguaje C y una introducción al C++. No se requieren grandes conocimientos para seguirlo, si bien serán necesarios unos conocimientos mínimos. No es un curso avanzado ni pretende serlo, sino más bien un curso básico.

La idea de poner esto aquí, surgió a raíz de un curso que hice, y en el que tome las notas directamente en el PC. Es decir, se trata a fin de cuentas de unos apuntes, que después he ido completando y corrigiendo pero sin complicar la cosa.

Como se podrá comprobar los ejemplos son sencillos y siempre están referidos al C/C++ estándar. Como podéis ver, en ningún momento se habla de ningún paquete o entorno en especial, si bien comentaré que todos los ejemplos se probaron en un entorno Linux. He intentado compilar algunos en entorno Visual Studio y he obtenido errores debido a librerías inexistentes, quizás es que no tuviera bien configurado el compilador.

El presente tutorial aun está en fase de desarrollo, lo estoy retocando, corrigiendo y ampliando, por lo que si tienes alguna información que quieras aportar sobre algún tema, y colaborar a hacer una cosa un poco mejor con mucho gusto lo incorporaré a estos apuntes, si bien no me gustaría que fueran demasiado extensos, pues ya existen por la red tutoriales mucho más completos que este. También podéis hacerme alguna corrección si veis algo que no sea correcto. Cualquier colaboración será bien recibida (me la podéis enviar por correo).

El formato final del curso aun no lo tengo decidido, de momento lo pongo en formato PDF, por que es lo que más comodo me resulta a partir de un archivo de texto plano. Como podéis observar no está muy presentable, los temas no están separados, las líneas no están controladas para facilitar la lectura, etc. pero puede ir sirviendo como punto de partida.

## Tipos de datos:

Los tipos de datos pueden ser simples o complejos (registros, estructuras o uniones). Los tipos de datos complejos se construyen a partir de los sencillos. Los tipos sencillos en C++ son:

- char: carácter
- int: número entero
- float: número real
- double: número real largo
- bool: booleano (true o false). True es un valor diferente de 0 y false es 0. El valor de true, puede llevar signo, lo cual según el caso puede aportar alguna información extra (cuando se comprueba como entero, no como booleano).

Además pueden ser signed o unsigned, según tengan o no signo. En caso de tener signo el valor de un tipo estará en el intervalo  $[-a, a-1]$ , siendo  $a=2^{(n-1)}$ , siendo n el número de bits que ocupa el tipo en cuestión. En caso de no tener signo el intervalo será  $[0, 2a-1]$ .

También pueden ser en algunos casos short o long. Un tipo short tiene un rango de valores posibles menor que el tipo normal, y un tipo long tiene un rango de valores posibles mayor. Se pueden hacer combinaciones entre short, long y signed y unsigned en función de las necesidades del programa.

El tamaño de los tipos es dependiente de la plataforma, tipo de máquina y sistema operativo, sobre la que se esté trabajando, si bien siempre serán múltiplos del byte (8bits).

## Ejemplo:

```
// Ejemplo del tamaño de los tipos de datos
#include <iostream> // Compilar con #g++ <nombre_archivo>
using namespace std; // No haría falta ponerlo si hubiéramos usado #include <iostream.h>

int main()
{
    cout << "Tamaño de char: " << sizeof(char) << endl;
    cout << "Tamaño de int: " << sizeof(int) << endl;
    cout << "Tamaño de short int: " << sizeof(short int) << endl;
    cout << "Tamaño de long int: " << sizeof(long int) << endl;
    cout << "Tamaño de double: " << sizeof(double) << endl;
    cout << "Tamaño de long double: " << sizeof(long double) << endl;
    cout << "Tamaño de float: " << sizeof(float) << endl;
}
```

## Salida que produce:

```
Tamaño de char: 1
Tamaño de int: 4
Tamaño de short int: 2
Tamaño de long int: 4
Tamaño de double: 8
Tamaño de long double: 12
Tamaño de float: 4
```

## Operadores aritméticos:

```

+: Suma
-: Resta
/: División
*: Multiplicación
%: Módulo o Resto de la división

```

El resultado de una operación aritmética será del mismo tipo que los valores que se han tratado, pudiendo perderse resolución o exactitud en el cálculo debido al tipo utilizado. No es conveniente mezclar tipos a la hora de realizar operaciones, en su lugar convendrá hacer instanciaciones o castings de uno de los tipos para operar con valores del mismo tipo. El casting se hace anteponiendo el tipo encerrado entre paréntesis a la variable que se quiere tratar. Por ejemplo:

```

int i;
double a, r;
a=(double)i*r; // i se trata durante la operación como un doble
i=(int)a/int(r); // a y r se tratan como enteros y su resultado se asigna a i

```

. Posiblemente si se hubiera hecho  $i=a/r$  el resultado sería diferente, a pesar de ser entero, pues  $i$  se ha declarado como entero.

Operadores booleanos:

Se utilizan para comparar expresiones, dando como resultado normalmente true o false si se cumplen o no (más concretamente un valor distinto de cero o cero).

```

< Menor que
> Mayor que
<= Menor o igual que
>= Mayor o igual que
== Igual a. Ojo con el ==, se suele confundir mucho con la asignación =

```

```

v=v+5 es equivalente a v+=5
v=v*2 es equivalente a v*=2

```

Operadores de incremento/decremento: ++, --

```

i=i+1; se usa habitualmente como ++i o i++, según sea con pre o postincremento
i=i-1; se usa habitualmente como --i o i--, según sea con pre o postdecremento

```

```

int x=10;
int y=x++;
El valor de y será 10 y x pasara a valer 11.
int x=10;
int y=++x;
El valor de y será 11 y x pasara a valer 11.

```

Operadores Lógicos &&, ||

```

&&: Y lógico
||: O lógico
!: Negación

```

Operadores a nivel de Bit &, |

```

&: Y o multiplicación a nivel de bit
|: O o suma a nivel de bit
^: Or exclusivo a nivel de bit
~: Or exclusivo a nivel de bit

```

Sentencias de Control:

Las palabras reservadas siempre van en minúsculas. Los grupos de sentencias se agrupan entre llaves {}, que equivalen al BEGIN END de Pascal. Toda sentencia, llamada a función, etc. finaliza con un ";" excepto las sentencias condicionales e iterativas como veremos a continuación. Suelen ser muy comunes errores al compilar debidos a la ausencia del ";". Un ";" detrás de una sentencia iterativa puede dar lugar a bucles infinitos.

Condicionales: Son sentencias según las cuales se ejecutará una u otra parte de un código dependiendo de las condiciones que se cumplan.

-if then ... else: Se ejecuta una u otra acción dependiendo de si se cumple o no una o varias condiciones

```

if (condición(es))                Se ejecuta si se cumple la condición
{
...
}
else                                Se ejecuta si no se cumple la condición
{
...
}

```

-switch: es equivalente a un if else anidado. Se llama case en otros lenguajes. Los case son puntos de entrada, por lo que para finalizar habrá que utilizar puntos de ruptura o salida mediante break.

```

switch (expresión)                La expresión ha de devolver un valor entero (int, char)
{
case valor1: ...
break; Finaliza la ejecución del bloque y no continúa con la siguiente instrucción.
case valor2: ...
break;
default: ...
}

```

}

Los caracteres deben ir encerrados entre comillas simples ' '. Los caracteres con válidos por que a fin de cuentas son enteros. Si se usara comillas dobles "" estaríamos hablando de una cadena de caracteres o array.

-Operador ternario ?: es una abreviación de la sentencia if. Solo trabaja con una condición, sobre una única variable y solo se puede ejecutar una sentencia en cada caso.

```
(condición)?casoafirmativo:casonegativo;
if (n<10)
    x=n+1;
else
    x=n-1;
Equivale a:
x=(n<10)?n+1:n-1
```

Iterativas o Bucles: Son sentencias o grupos de sentencias que se cumplirán mientras se cumpla una o varias condiciones (o no se cumplan).

-while: Mientras la condición sea verdadera se ejecuta lo contenido en el bucle.

```
while (condición)
{
...;
}
```

-do ... while: es una evolución de la sentencia anterior.

```
do
{
...;
continue;
}
```

```
while (condición);
```

La diferencia con el caso anterior es el número mínimo de veces que se ejecuta cada flujo de sentencias, en el primer caso puede suceder que no se ejecute ninguna vez, mientras que en el segundo al menos se ejecuta una vez el contenido del bucle. Esto es debido a como se realiza la comprobación de la condición, al principio o al final del flujo.

-for: realiza el flujo de sentencias un número finito de veces, en función de la variación de los contadores.

```
for (inicio contadores; condición de iteración o final contadores; zona de incremento de contadores)
{
...;
}
```

La condición de iteración es idéntica a la de un while.

```
for (i=0;i<10;i++)
{
cout << i;
}
```

Las sentencias dentro de la zona de contadores, van separadas por comas.

La zona de inicialización es optativa, si no se especifica, se tomará el valor que tengan los contadores en el momento en que se entra en el bucle.

La zona de incremento también es optativa, pues se puede especificar dentro del flujo de sentencias del bucle.

```
i=0;; // Esto podría ir dentro del for, en la zona de inicio de contadores
```

```
for (;i<10;)
{
cout << i;
i++; // Esto podría ir dentro del for, en la zona de incremento de contadores
}
```

```
for (i=0,j=6;i<j;i++,j--) // En una array saca el contenido de los extremos al centro
```

```
cout << numeros[i] << numeros[j] ;
```

```
for (i=0;i<10;i++)
{
cout << i;
}
```

equivale a las dos sentencias siguientes:

```
for (i=0;i<10, cout << i ; i++);
for (i=0;i<10; cout << i++ );
```

El bucle:

```
for(;;);
```

sería un bucle infinito, del que no se saldría nunca salvo finalizando el programa desde fuera (Ctrl-Alt-Sup, Ctrl-C, etc.), debido por un lado a las condiciones de los contadores, en este caso ninguna, y al ";" existente al final.

Salto: Son sentencias mediante las cuales se altera el flujo normal del programa saltando a otra parte del mismo. Generalmente estas sentencias se insertan para ejecutar un nuevo código por q



```

1465341783
1465341783
Violación de segmento

```

Arrays multidimensionales:

```

tipo nombre[tamaño_fila][tamaño_columna];
int matriz[3][3];

```

Para recorrer los elementos de un array se suelen usar los bucles for, exigiendo un anidamiento por cada índice.

En memoria estos arrays se almacenan de forma lineal, ubicándose por líneas (linea1, linea2, linea3, ...)

No es obligatorio definir el número de filas, pero si el número de columnas.

```

int matriz[][2]={{1,3},{7,9},{12,24}};           // 1  3
                                                    // 7  9
                                                    // 12 24
char nombres[][6]={"Ana","Luis","Pedro"};       // Se reserva un espacio de más para el '\0'
cout << nombres[1] << endl;                       // mostraría "Luis"

```

Punteros:

Un puntero es una variable que contiene una dirección de memoria. Hay que especificar el tipo de valor que hay en dicha posición de memoria, para conocer el tamaño de la región a la que se refiere, porque una dirección de memoria como tal es un byte, y en la variable puede almacenarse un doble, una cadena de caracteres, etc.

Los punteros se definen de siguiente modo:

```

tipo *nombre;           // En el tipo es donde se especifica el tamaño de la dirección de memoria con la que se trabajará.
int *ptr1;
char *ptr2;

```

Para conocer la dirección de memoria de una variable se antepone el símbolo & al nombre de la variable.

```

char *ptr;
cout << "La dirección de memoria es " << ptr << endl;
cout << "El contenido de ptr es " << *ptr << endl;

```

Ejemplo 1: Punteros

```

#include <iostream>
using namespace std;
int main()
{
    int i;
    int *ptr;
    int **ptrptr;
    i=1000;
    ptr=&i;
    ptrptr=&ptr;
    cout << "dirección de memoria de ptr: " << ptr << endl;
    cout << "valor de ptr: " << *ptr << endl;
    cout << "valor de &ptr: " << &ptr << endl;
    cout << "dirección de memoria de i: " << &i << endl;
    cout << "valor de i: " << i << endl;
    cout << "dirección de memoria de ptrptr: " << ptrptr << endl;
    cout << "valor de *ptrptr: " << *ptrptr << endl;
    cout << "valor de **ptrptr: " << **ptrptr << endl;
}

```

Salida que produce:

```

dirección de memoria de ptr: 0xbfffe734
valor de ptr: 1000
valor de &ptr: 0xbfffe730
dirección de memoria de i: 0xbfffe734
valor de i: 1000
dirección de memoria de ptrptr: 0xbfffe730
valor de *ptrptr: 0xbfffe734
valor de **ptrptr: 1000

```

Ejemplo 2: Punteros y Arrays

```

#include <iostream>
using namespace std;
int main()
{
    int i;
    int numeros[]={2,4,6,8,10};
    cout << "numeros: " << numeros << endl;
}

```

```

        cout << "&numeros[0]: " << &numeros[0] << endl;
        cout << "*numeros: " << *numeros << endl;
        for (i=0;i<5;i++)
            cout << "En la dirección " << &numeros[i] << " hay un " << numeros[i]
] << endl;
    }

```

Salida que produce:

```

numeros: 0xbffffeb80
&numeros[0]: 0xbffffeb80
*numeros: 2
En la dirección 0xbffffeb80 hay un 2
En la dirección 0xbffffeb84 hay un 4
En la dirección 0xbffffeb88 hay un 6
En la dirección 0xbffffeb8c hay un 8
En la dirección 0xbffffeb90 hay un 10

```

Una observación es que `numeros+1` es equivalente a `numeros[1]` y a `numeros+sizeof(int)`, pues en realidad lo que se está haciendo es acceder a la siguiente dirección de memoria a la dirección de memoria base

```
cout << numeros+1 << "hay un 4" << endl;
```

Funciones:

Las funciones son partes de código que tienen entidad propia y que realizan una tarea determinada que puede ser repetitiva a lo largo del programa y que ayudan a clarificar la estructura del programa. Existen tres formas de trabajar con funciones:

- La función está implementada en el mismo archivo en que se utiliza, antes del cuerpo del programa principal.

```

#include
#define
int sumar(int a, int b)
{
    return a+b;
}

main ()
{
    ...
    cout << sumar(76,4);
    ...
}

```

- La función está definida al principio e implementada al final en el mismo archivo en que se utiliza,

```

#include
#define
int sumar(int, int);
main ()
{
    ...
    cout << sumar(76,4);
    ...
}
int sumar(int a, int b)
{
    return a+b;
}

```

- La función se define e implementa en un archivo independiente. Para ello, el archivo en que se defina deberá ir incluido en la zona de includes.

```

--MiPrograma.cc
#include
#include "MisFunciones.h"
#define
int sumar(int, int);

main ()
{
    cout << sumar(76,4);
}

--MisFunciones.h:
int sumar(int, int); // Podría ir el código de implementación a continuación, pero no se suele hacer.

--MisFunciones.cc
#include
#include "MisFunciones.h"
#define
int sumar(int a, int b)

```

```

    {
    return a+b;
    }

```

Las funciones pueden ser macros o inline, de modo que en tiempo de compilación la llamada a la función se sustituye por el código de la función si se puede. Esto evita la pérdida de velocidad en muchos casos, pero hace que se genere más código. Para que una función pueda ser inline no ha de contener bucles ni ha de llamar a otra función inline. Las funciones inline se definen de la siguiente forma:

```
inline tipo función (parámetros)
```

Si la función no es void ha de existir un return al final de la función y ha de ser del mismo tipo que el con el que se ha declarado la función. En caso de que la función sea void se puede hacer un return sin devolver nada.

Esto se compilaría del siguiente modo:

```

g++ -c -o MiPrograma MiPrograma.cc           # Se compila el programa
g++ -c -o MisFunciones MisFunciones.cc      # Se compilan las funciones
g++ -o Salida MiPrograma MisFunciones      # Se linkan las funciones y el programa (enlazan)

```

Funciones Recursivas:

Son funciones normales, como las vistas hasta ahora, con la salvedad de que son funciones que se llaman a sí mismas desde la propia función. Su utilización dota al programa de elegancia y reduce el código escrito, si bien reduce su rendimiento. La recursividad se podría sustituir por un bucle en muchas ocasiones. En toda función recursiva debe existir un punto de salida, para evitar caer en un bucle infinito que llegaría a bloquear la memoria del ordenador. Un típico ejemplo de función recursiva es el cálculo de un factorial:

```

int Factorial (int i)
{
    if (i>1)
        return i*Factorial(i-1);           // Mientras el valor pasado sea mayor que 1 se sigue
    llamando a la función
    else
        return 1;                          // Cuando el valor sea 1 o menor devolver 1. Este es
    el punto de salida de la función
}

```

Makefile:

El archivo makefile se emplea para compilar todo un proyecto. En él se incluyen las líneas necesarias para realizar la compilación y enlace de todos los archivos. El proyecto se construye ejecutando el comando make, automáticamente el buscará el archivo makefile, en caso de no existir habrá que especificar el archivo equivalente al makefile:

```
make -f <archivo>
```

El formato de un makefile viene a ser el siguiente:

```

etiqueta1: fichero dependencia
    g++ ...

etiqueta2: fichero dependencia
    g++ ...

...

```

Los comentarios se pueden poner comenzando con el símbolo #, como en cualquier script de UNIX.

Ejemplo:

```

#Archivo makefile de prueba
#Generación del ejecutable
ejecutable: Misfunciones Miprograma
    g++ -o ejecutable Misfunciones Miprograma

#Generación de los archivos objeto
Misfunciones.o: Misfunciones.h Misfunciones.c
    g++ -c -o Misfunciones Misfunciones.cc
Miprograma.o: Misfunciones.h Misfunciones.cc Miprograma.cc
    g++ -c -o Miprograma Miprograma.cc

Test:
    ./ejecutable

```

Este archivo se ejecutaría con make si se llamara Makefile.

Para ejecutar solo una etiqueta del makefile se haría del modo:

```
make -f <archivo> <etiqueta>
```

El comando make comprueba si se ha modificado algún archivo desde la última compilación, y solo realiza los pasos afectados por las modificaciones realizadas desde entonces.

Funciones con argumentos por defecto:

Son funciones que si no se especifica un argumento, se asocia un valor por defecto, no es lo mismo que funciones con número variable de argumentos. Estas funciones se definen del siguiente modo:

```
int dividir(int a, int b=1);
```

Si no se especifica el parámetro b, se tomará el valor por defecto 1. Los valores por defecto siempre han de ir a la derecha (al final de los parámetros) y nunca por delante de un parámetro obligatorio.

Dentro del mismo ámbito en C no se pueden tener funciones con el mismo nombre, por problemas del compilador, si en diferentes ámbitos. En C++ si se pueden tener dos funciones con el mismo nombre, lo cual se llama sobrecarga. Pero existen unas reglas para realizar la sobrecarga.

Para poder sobrecargar las funciones se han de cumplir algunas de las condiciones siguientes:

- Tener diferente número de argumentos.
- Tener el mismo número de argumentos pero al menos han de diferenciarse en el tipo de un argumento.
- El tipo que devuelven las funciones es diferente, pero con mucho cuidado, pues puede darse el caso de llamadas ambiguas, pues puede darse el caso de no conocer a cual de las dos funciones se llama.

Las siguientes funciones serían validas dentro del mismo ámbito.

```
int sumar(int, int);
int sumar(char, int);
int sumar(int, char[]);
int sumar(int, int, int);
char[] sumar(char[], char[]);
char[] sumar(int, int); // Esta puede dar ambigüedad con la primera función.
```

Las variables a una función se pueden pasar por valor o por referencia.

Cuando una variable se pasa por valor, en realidad se pasa una copia de la misma, y cualquier modificación que la función haga sobre ella no tendrá efecto sobre el valor de la variable en el programa o función desde el que se realiza la llamada.

```
void doblar(int a);
```

y se llamaría con:

```
doblar(valor);
```

Cuando una variable se pasa por referencia, si se modifica el contenido de la misma. En C esto se realiza pasando a la función la dirección de memoria. Existe dos formas de hacer esto en C++, una usando la forma tradicional de C y otra introducida por C++:

```
void doblarReferenciaC(int * a);
void doblarReferenciaCpp(int & a); // El & indica que se hace por referencia, cambia respecto a C
```

Para evitar la modificación de la estructura pasada, durante la compilación, se podría hacer mediante una referencia a una constante:

```
void doblarReferenciaCpp(const int & a); // Si se detecta algo del tipo a= falla en compilación
```

las llamadas serían:

```
doblarReferenciaC(&valor);
doblarReferenciaCpp(valor); // Ojo, aquí no se pasa la dirección de memoria
```

Los arrays siempre se pasan por referencia. Se recomienda que las grandes estructuras se pasen por referencia, para evitar problemas de memoria (ahorrar memoria). Solo se recomienda pasar por valor variables de los tipos nativos o simples. Este efecto de la memoria se hace más importante aun en el caso de funciones recursivas, dentro de las cuales se realiza una llamada a la misma función.

Ejemplo: Función que recibe dos punteros e intercambia sus valores en C y en C++.

```
void intercambiarC(int **ptr1, int **ptr2) // La llamada sería intercambiarC(&ptr1, &ptr2);
{
    int *aux; //Valor auxiliar para el intercambio
    aux=*ptr1;
    *ptr1=*ptr2;
    *ptr2=aux;
}

void intercambiarCPP(int *&ptr1, int *&ptr2) // La llamada sería intercambiarC(ptr1, ptr2);
```

```

{
int *aux;          //Valor auxiliar para el intercambio
aux=ptr1;
ptr1=ptr2;
ptr2=aux;
}

```

Cast:

Las variables nunca cambian de tipo, pero a veces puede ser necesario hacer un casting para obtener un resultado de un tipo a partir de valores de otro. Dos enteros divididos dan un entero, pero muchas veces es necesario obtener el real, para ello es lo que se usa el casting, y se hace anteponiendo el tipo que se desea obtener delante de la expresión:

```
(tipo) expresión;
```

```

int a=2, b=3
cout << b/a << endl;          // Da 1
cout << (double)b/(double)a << endl; // Da 1.5      (3.0/2.0=1.5), también sería valido
(double)(b/a)

```

```

Conversión implícita: double d=7;
Conversión explícita: int i=(int)6.3;

```

Memoria Dinámica:

Hasta ahora hemos trabajado con variables de dimensión fija. Existen también variables de dimensión variable, que se redimensionan en tiempo de ejecución. Esto es lo que se conoce como memoria dinámica. Inicialmente serán un puntero, y durante la ejecución se le reservará la memoria necesaria, se le asignarán valores, etc. y se liberará. Antes de comenzar a trabajar con memoria reservada, hay que comprobar que realmente se ha obtenido.

En C, las funciones para reservar y liberar memoria están incluidas en `stdlib.h`, y son:

```

malloc      pide al SO bytes libres de memoria
calloc     pide al SO bytes libres de memoria y los inicializa a 0
realloc    redimensiona un espacio de memoria
free       libera una memoria que ya no es necesaria

```

En C++ para reservar memoria se usan los operadores `new` para reservar memoria y `delete` para liberarla.

```

variable = new tipo [numero de elementos];
delete [] variable;

```

Ejemplo:

```

int *numeros
numeros=(int *)malloc(elementos*sizeof(int));
if (numeros==0)          // equivale a if((numeros=(int *)malloc(elementos*sizeof(int)))==NULL)
L)
    exit(-1);          // Salir si no se ha conseguido reservar la memoria
....
free(numeros);        // Si se intenta liberar un puntero nulo da un error

```

Ejemplo (versión C):

```

#include <iostream>
using namespace std;
#include <stdlib.h>
bool esPar(int a);

```

```

int main()
{
    int i, cuantos;
    int *numeros;

    cout << "Cuantos numeros deseas en el array? " ;
    cin >> cuantos;
    if ((numeros=(int *)malloc(cuantos*sizeof(int)))==NULL)
        exit(-1);
    for (i=0;i<cuantos;i++)
        cin >> numeros[i];
    for (i=0;i<cuantos;i++)
    {
        if (esPar(numeros[i]))
            cout << numeros[i] << " es PAR\n";
        else
            cout << numeros[i] << " es IMPAR\n";
    }
    free(numeros);
}

```

```
bool esPar(int a)
{
    return !(a%2);
}
```

Ejemplo (versión C++):

```
#include <iostream>
using namespace std;
#include <stdlib.h>
bool esPar(int a);

int main()
{
    int i, cuantos;
    int *numeros;

    cout << "Cuantos numeros deseas en el array? " ;
    cin >> cuantos;
    if ((numeros=new int [cuantos])==NULL)                // En lugar de mallo

        exit(-1);
    for (i=0;i<cuantos;i++)
        cin >> numeros[i];
    for (i=0;i<cuantos;i++)
        {
            if (esPar(numeros[i]))
                cout << numeros[i] << " es PAR\n";
            else
                cout << numeros[i] << " es IMPAR\n";
        }
    delete []numeros;                                    // En lugar de free
}

bool esPar(int a)
{
    return !(a%2);
}
```

La salida por pantalla en ambos casos sería la misma y del tipo:

```
Cuantos numeros deseas en el array? 6
76 34 56 67 89 23
76 es PAR
34 es PAR
56 es PAR
67 es IMPAR
89 es IMPAR
23 es IMPAR
```

Si se omite el free o delete, al finalizar el programa, la memoria ocupada se liberará si reservada en el es segmento del programa, pero no en el caso de que se hubiera reservado fuera. Como es una cosa que no se puede controlar, pues es el SO quien controla donde ubicar dicha memoria, hay que liberarla, para que después no se produzcan cuelgues o bloqueos del sistema.

Como regla general, todo new, ha de ir seguido por un delete.

Si se ponen dos delete relativos a la misma variable o zona de memoria sin haber sido asignada por segunda vez, el programa dará un error en ejecución. Después de un delete o un free es conveniente, pero no obligatorio poner la variable apuntado a NULL, es decir, asignarla NULL.

El objeto cin tiene un método que toma una línea de caracteres hasta que se introduce un carácter determinado, que por defecto es el retorno de carro '\n':

```
cin.getline(buffer, longitud, carácter);
cin.getline(buffer, longitud);                // Por defecto toma '\n'
```

Funciones de cadenas de caracteres, incluidas en la librería <string.h>

- strlen(cadena): Devuelve la longitud de la cadena descontando el carácter NULL del final
- strcpy(char \*destino, char \*origen): Copia la cadena origen en la cadena destino. El destino tendrá el tamaño necesario para poder contener el contenido de origen.
- strcat(destino, origen): Concatena la cadena origen a la cadena destino y lo deja en ella.
- strcmp(s1, s2): Compara las dos cadenas, devolviendo <0 si s1<s2, =0 si s1=s2 y >0 cuando s1>s2, entendiendo la comparación en orden alfabético.

Ejemplo con cadenas (ejemplo09-Cadenas.cc):

```
#include <iostream>
```

```

using namespace std;
#include <string.h>

int main()
{
    char buffer[255];
    char *nombre;
    char *apellidos;
    char *domicilio;
    cout << "Nombre: ";
    cin.getline(buffer,254);
    nombre=new char [strlen(buffer)+1]; // 1 más para el carácter '\0'
    strcpy(nombre,buffer);
    cout << "Apellidos: " ;
    cin.getline(buffer,254);
    apellidos=new char [strlen(buffer)+1]; // 1 más para el carácter '\0'
    strcpy(apellidos,buffer);
    cout << "Domicilio: " ;
    cin.getline(buffer,254);
    domicilio=new char [strlen(buffer)+1]; // 1 más para el carácter '\0'
    strcpy(domicilio,buffer);
    cout << "Nombre: " << nombre << endl;
    cout << "Apellidos: " << apellidos << endl;
    cout << "Domicilio: " << domicilio << endl;
    delete [] nombre;
    delete [] apellidos;
    delete [] domicilio;
}

```

Quando queremos trabajar con una array variable de cadenas variable, tendremos una matriz de dos dimensiones. Habrá que reservar espacio en primer lugar para el array de cadenas (cuantas cadenas va a contener) y después habrá que reservar la memoria para cada una de las cadenas contenidas. El array de cadenas, en realidad lo que contiene son punteros a las cadenas que se definirán después.

Ejemplo:

```

char **nombres;
cout << "Cuantos nombres?";
cin >> cuantos;

nombres=new char * [cuantos];
for (i=0;i<cuantos;i++)
{
    cout << "Introduzca el nombre " << i << ": ";
    cin.getline(buffer,254);
    nombres[i]=new char[strlen(buffer)+1];
    strcpy(nombres,buffer);
}

// ...;
for (i=0;i<cuantos;i++)
{
    delete [] nombres[i];
}
delete [] nombres;
}

```

Estructuras y Uniones:

Las estructuras o registros son un tipo de datos complejos formados por agrupación de tipos mas sencillos. Al final de la definición de una estructura, después de la llave hay que poner el ";"

```

struct TPersona
{
    char nombre[40];
    int edad;
};

struct TPersona personal; // Para una persona
struct TPersona personas[10]; // Para 10 personas (Array)
personal.edad=10;
strcpy(personal.nombre, "Luis");

```

Una función se podría definir y usar como hasta ahora:  
imprimir personal;

donde la función imprimir seria del tipo:

```

void imprimir(struct TPersona persona)
{
    cout << "Nombre: " << persona.nombre;
    cout << "Edad: " << persona.edad;
}

```

```
}

```

Una estructura se puede inicializar en el momento de la declaración

```
struct TPersona persona2={"Luis", 22}
```

También se pueden generar listas variables de registros y trabajar dinámicamente con ellas:

```
struct TPersona *personas;

personas=new struct TPersona[N] // Donde N se define dinámicamente
...
delete [] personas;
```

También se pueden usar punteros a estructuras:

```
struct TPersona *ptrpersona

ptrpersona=new struct TPersona;
ptrpersona->edad=22; // Es equivalente a *(ptrpersona).edad
strcpy(ptrpersona->nombre,"Luis"); // Es equivalente a *(ptrpersona).nombre
```

Una estructura puede ser tan compleja como se desee, pudiendo contener dentro de ella otras estructuras más sencillas, lo cual favorece la reutilización de tipos o estructuras más sencillas:

```
struct TEmpleado
{
    struct TPersona DatosPersonales;
    char Departamento[10]
};

struct TEmpleado empleado1

empleado1.DatosPersonales.edad=23;
strcpy(empleado1.DatosPersonales.nombre,"Luis");
strcpy(empleado1.Departamento,"Logistica");
```

La función `cin.ignore()` se emplea para ignorar caracteres leídos como por ejemplo el retorno de carro.

Las uniones son iguales a las estructuras excepto en el espacio de memoria que ocupan. En una unión por ejemplo podríamos trabajar con personas o con empleados, a priori sin saber que trabajaremos. Se reservará siempre espacio de memoria para la estructura de mayor tamaño, pero dependiendo de la forma de ejecución se utilizará una u otra estructura.

```
union datos
{
    struct TPersona persona;
    struct TEmpleado empleado
};

union datos candidatos[10]; // Se reservan 10 espacios de la estructura de mayor
tamaño entre TPersona y TEmpleado
```

Si trabajamos con personas usaremos `candidatos[i].nombre` y si lo hacemos con empleados con `candidatos[i].datospersonales.nombre`. Si el array lleva estructuras de los dos tipos intercaladas habrá que llevar un control de ello para que no se produzcan resultados inesperados, que no errores, pues es el programa en principio no tiene por que fallar.

## ORIENTACION A OBJETOS

La POO es un modelo de programación en la que se emplean objetos para la solución de problemas. En POO no se habla de variables, si no de objetos, tampoco se habla de tipos, sino de clases. Las funciones o procedimientos en POO se denominan métodos. Los métodos pueden implementarse utilizando programación en C o aprovechando las ventajas que aporta el C++.

En principio cualquier programa es susceptible de ser realizado con POO, si bien hay que resaltar que un programa realizado en C siempre será más rápido que uno realizado en C++, pero también será más complicado de leer y modificar en el futuro. En los programas realizados en C++ se puede modificar una clase sin que halla que retocar el programa, aunque si habrá que recompilarlo. En POO el programador no necesita saber como están implementadas las diferentes clases, sino únicamente deberá saber como trabajar con ellas, que métodos hay accesibles y como se devuelven los datos que interesen para cada caso.

La POO se caracteriza por:

- Abstracción: Con el fin de evitar llamadas complejas a funciones se utilizan llamadas sencillas a acciones propias del objeto que se quiere utilizar.
- Encapsulación: Solo se hace publico lo que se desea que sea utilizable, permaneciendo el resto privado, con el fin de facilitar el entendimiento y evitar que se pueda manipular cosas que afectarían a lo publico.
- Reutilización: Para no tener que volver a crear objetos ya creados.

-Modularización: Hay que diseñar las aplicaciones en módulos de modo que estos sean independientes entre sí, de modo que una modificación de un módulo sea independiente de los demás.

Una clase es un tipo abstracto de datos (TAD) con herencia. Un TAD es una estructura de datos con unas operaciones asociadas. Así pues una clase es una estructura de datos junto con unas operaciones asociadas a ellas en las que hay una herencia de otras clases de rango superior.

La nomenclatura que se empleara en POO será del tipo:

```

clase objeto;
clase *objeto;

objeto.atributo;
objeto->atributo;

```

Como se puede apreciar la nomenclatura es similar a la usada al trabajar con estructuras o registros. Las llamadas a métodos o funciones propias de una clase se realizará de forma similar a como se referencian los atributos:

```

objeto.metodo();
objeto->metodo();

```

Habrán partes públicas, privadas y protegidas. Las partes públicas serán accesibles desde fuera de la clase, las partes privadas solo serán accesibles desde la clase y las partes protegidas serán accesibles desde la propia clase y su inmediata derivada.

Se recomienda que las estructuras de datos (variables internas que constituyen el objeto) se an siempre privadas, y que sean manipuladas a través de los métodos o funciones diseñadas a tal efecto.

Los métodos generalmente son públicos, aunque pueden existir métodos privados que solo se utilicen a nivel interno dentro de la clase. Los métodos pueden ser de tres tipos:

-Métodos de Construcción/destrucción de objetos. Construyen o reservan espacio y lo liberan para un objeto.

-Métodos de Acceso o de tipo Set o Get. Establecen o recuperan propiedades del objeto

-Métodos de Comportamiento. El resto de procedimientos.

Una clase se define de la siguiente forma (con ";" al final de las llaves)

```

class NOMBRE_CLASE
{
public:
    ...
    ...

private:
    ...
    ...

protected:
    ...
    ...

public:
    ...
    ...
};

```

Orden de definición de una clase, lo cual se indica en el archivo de cabecera clase.h:

1.-Definición de Estructuras de Datos o Atributos

2.-Funcionalidad, operaciones o métodos

a.- Constructores

b.- Destrucción (se denominan como el constructor anteponiendo ~ al nombre del constructor)

c.- Métodos de Acceso (setter's y getter's)

d.- Métodos de Comportamiento.

La implementación de los métodos se realizaría en el archivo clase.cpp

Generalmente, los programas orientados a objetos suelen ser más lentos que los programas desarrollados directamente en C.

Con this hacemos referencia al objeto sobre el cual se está trabajando, es un puntero genérico, tiene sentido en tiempo de ejecución. Un ejemplo de cuando puede ser útil es:

```

void SetX(int x){
    this->x=x;
}
equivale a
void setX(int a){
    x=a;
}

```

Ejemplo de la clase Punto en 2 dimensiones con diferentes utilizaciones de this y usando funciones i

nline para no perjudicar la velocidad de ejecución.

Punto2D.h

```
class Punto2D
{
private:
    int X, Y;
public:
    Punto2D(); //Constructor por defecto
    Punto2D(int a, int b); //Constructor con argumentos
    Punto2D(const Punto2D &p); //Constructor copia
    inline void setX(int x);
    inline void setY(int y);
    inline int getX() const; //Para que funcione el constructor c

    inline int getY() const;
    void moverABS(int x, int y);
    void moverREL(int dx, int dy);
    Punto2D clonar();
    void imprimir();
    ~Punto2d();
};
```

opia

Punto2D.cpp

```
#include "Punto2D.h"
#include <iostream>
using namespace std;

Punto2D::Punto2D() //Constructor por defecto
{
    setX=0;
    setY=0;
}
Punto2D::Punto2D(int a, int b) //Constructor con argumentos
{
    setX=a;
    setY=b;
}
Punto2D::Punto2D(const Punto2D &p); //Constructor copia
{
    setX(p.getX());
    setY(p.getY());
}
Punto2D::~Punto2D(); //Destructor
{
    cout << "destruyendo ... ";
    imprimir();
}
void Punto2D::setX(int x)
{
    this->X=x;
}
void Punto2D::setY(int y)
{
    this->Y=y;
}
int Punto2D::getX() const
{
    return this->X;
}
int Punto2D::getY() const
{
    return this->Y;
}
void Punto2D::moverABS(int x, int y)
{
    this->X=x;
    this->Y=y;
}
void Punto2D::moverREL(int dx, int dy)
{
    // Para que sea más fácil modificar en el futuro, au
    setX(getX()+dx); // this->X+=dx;
    setY(getY()+dy); // this->Y+=dy;
}
Punto2D Punto2D::clonar()
{
    Punto2D pAux;
```

nque es mas lento

```

        pAux.setX(this->getX());
        pAux.setY(this->Y);
        return pAux;
    }

    void Punto2D::imprimir()
    {
        cout << "X=" << getX();          // También sería valido
this->getX()
        cout << "Y=" << this->Y << endl;
    }

```

```

TestPunto2D.cc
#include "Punto2D.h"

```

```

int main()
{
    Punto2D *A, *B;
    A=new Punto2D(1,1);
    B=new Punto2D();
    A->imprimir();
    A->setX(4);
    A->setY(3);
    A->imprimir();
    *B=A->clonar();
    A->moverABS(2,2);
    B->moverREL(1,-1);
    B->imprimir();
    A->imprimir();
    Punto2D p2(*B);          //Constructor copia
    p2.imprimir();
    delete A;
    delete B;
}

```

La salida por pantalla sería:

```

X=1 Y=1
X=4 Y=3
Destruyendo ... X=4 Y=3          //Debido a que borra el auxiliar usado en cl
onar
X=5 Y=2
X=2 Y=2
X=5 Y=2
Destruyendo ... X=2 Y=2
Destruyendo ... X=5 Y=2
Destruyendo ... X=5 Y=2

```

Constructores:

Un constructor inicializa el estado de un objeto, no lo crea. Los constructores se llaman al definir el objeto. Las características de un constructor se pueden resumir en:

- Inicializa un objeto
- No devuelve nada, ni siquiera void
- Su nombre coincide con el nombre de la clase
- Se pueden sobrecargar, en función del número de argumentos. Existen constructores:
  - Por defecto, es el que se utiliza cuando se define un objeto. Los valores que toman los atributos no suelen tener ningún sentido.
  - En función del número de argumentos, en este caso los atributos toman ya valores con algún sentido.
  - Constructor de copia, inicializa un objeto en base a otro objeto
- Toda clase tiene un constructor. Si no se define uno, el sistema crea uno, llamado por omisión.
- Al constructor se le llama cuando se define el objeto o cuando se realiza un new.

Ejemplo de como llamar al constructor:

Por defecto o sin argumentos:

Estático:

```

Punto2D p1;
Punto2D puntos[2];
Punto2D p1=Punto2D();

```

Dinámico:

```

Punto2D *p1=new Punto2D();

```

Con argumentos:

Estático:

```

Punto2D p1(7,28);
Punto2D puntos[]={Punto2D(7,28), Punto2D()}
Punto2D p1=Punto2D(7,28);

```

Dinámico:

```

                Punto2D *p1=new Punto2D(7,28);
Copia:
    Estático:
        Punto2D p2(p1);
        Punto2D p2=Punto2D(p1);
    Dinámico
        Punto2D *p2=new Punto2D(p1);

```

El constructor copia solo tiene sentido cuando los atributos son estáticos.

Siempre que haya un new en un constructor hay que implementar necesariamente un constructor copia, para que objetos diferentes no apunten a las mismas zonas de memoria. Siempre que haya que implementar el constructor copia, habrá que modificar (redefinir) el operador asignación.

#### Destructores:

Los destructores son la última operación que realiza cualquier objeto. Entre sus características destacan:

- No devuelven nada, ni siquiera void.
- Se denominan como la clase anteponiendo "~" p.ej. ~Punto2D()
- Los destructores no se pueden llamar, son siempre llamados por el sistema al finalizar la ejecución del método, programa o se hace un delete del objeto
- Los destructores deberían ser siempre virtuales, por cuestiones de herencia.
- Los destructores tienen por misión liberar la memoria dinámica utilizada por el objeto. Por cada new que se haga en el constructor hay que hacer un delete en el destructor.

Un destructor se puede utilizar para almacenar datos sobre la ejecución, grabar a disco, cerrar un archivo, cerrar una conexión, etc.

Cuando un objeto contiene otro objeto, este puede estarlo de dos formas, por asociación (en cuyo caso se hace asignación en el constructor) o por composición (en cuyo caso se hace un new en el constructor).

Ver ejemplo del Rectángulo.

Se puede sobrecargar cualquier operador (+, -, \*, /, new, delete, <=, ==, =, [], (), etc.), excepto el operador de ámbito ":", los operadores de selección de miembro "." y "->", el operador condicional "?:", el operador de tamaño "sizeof" y el de tipo "typeid".

#### HERENCIA:

La herencia es una especialización de una clase. Una clase general se utiliza para construir una clase más concreta. Por ejemplo, un empleado es una persona, por tanto la clase empleado se pondrá entre otras cosas de la clase persona. Según esto, la clase empleado hereda de persona, y por tanto todo lo implementado para persona está implementado para empleado, y tan solo hay que implementar todo lo que sea nuevo para la clase empleado.

```

class empleado : public persona // Se podría hacer private, pero los métodos no serán
    {
    }

```

La herencia no solo puede ser por los atributos, sino que puede ser por modificación del comportamiento.

Los métodos de la clase padre son heredados por la clase hija y pueden ser utilizados como propios. La parte privada de la clase padre no es accesible para la clase hija directamente, solo son accesibles las partes públicas y protected. Lo declarado como privado solo es accesible desde la clase en que se declara.

La palabra virtual se emplea para indicar a la clase de que se debe buscar en las heredadas antes de buscar en la clase padre por si se hubiera redefinido un método. En caso de no haberse redefinido se utilizará el método de la clase padre. Esto es útil en el caso de polimorfismo.

#### POLIMORFISMO:

El polimorfismo se basa en definir un puntero a la clase padre y luego poder usar con el objetos de clases hijas. Por ejemplo:

```

Punto2D *puntos[2];
puntos[0]=new Punto3D(1,1,1);
puntos[1]=new Punto2D(0,0);
for (i=0;i<2;i++)
    puntos[i]->imprimir();

```

Para cada se utilizará la función imprimir adecuada.

Para ver un ejemplo de todo esto ver el ejemplo Punto3D.

Ejemplo de como escribir en un archivo (se necesita la librería fstream):

```

#include <iostream> // Incluye la definición de endl
#include <fstream> // Para tratar archivos
using namespace std;

```

```

int main()
{
    int n=10;
    ofstream f("datos.dat", ios::out | ios::app); // Abre el fichero para escritura y si existe añade las cosas al final.
    f << "Hola " << n << endl;
    f.flush();
}

```

ios.out e ios::app son dos atributos públicos de la clase ios, y además son constantes, o un atributo estático, o atributo de clase. Este tipo de atributos se definen del siguiente modo:

```

class clase
{
    static tipo variable; // static const tipo variable; para que además fuera constante y no se pudiera modificar
    ...;
};

```

y el calor se le da en la implementación de la clase ( en el .cpp)

```

tipo clase::variable = VALOR; // se define fuera de cualquier método, por ejemplo, justo después de los includes

```

La palabra static indica que esa variable es común para todas las clases, y que todas acceden a la misma posición de memoria. Un método estático solo puede consultar atributos estáticos.

Entrada/Salida:

Para las operaciones de E/S se utilizan las librerías iostream, ifstream, ofstream.

En el ejemplo en papel se incluye un programa que divide un archivo en archivos de 10 bytes.

Formateo de salida: Ver programa de Entrada-Salida.

```

#include <iostream>
using namespace std;

int main()
{
    double ventas[]={5189.0987, 65.45, 789.45, 1452.36, 5412, 456.8};
    char * ciudades[]{"Madrid", "Ciudad Real", "Sevilla", "Barcelona", "Toledo", "Lugo"};

    int i;
    //const int formato=ios::fixed | ios::left; // Constante que fija la salida a una alineación a la izquierda

    cout.flags(ios::fixed | ios::left);
    for (i=0;i<sizeof(ventas)/sizeof(double);i++)
    {
        cout.width(20); // Ancho de cada campo
        cout.fill('.'); // Rellenar espacios con '.'
        cout << ciudades[i];
        cout.unsetf(ios::left); // Quitar alineación izquierda
        cout.width(10); // Ancho de cada campo
        cout.precision(2); // Sacar 12 decimales
        cout << ventas[i] << endl;
        cout.setf(ios::left); // Se vuelve a poner alineación izquierda, equivale a cout.flags(formato)
    }
}

```

Producirá una salida del tipo:

```

Madrid.....5189.10
Ciudad Real.....65.45
Sevilla.....789.45
Barcelona.....1452.36
Toledo.....5412.00
Lugo.....456.80

```

Concurrencia:

Antes de nada debemos recordar que un proceso es un programa en ejecución. En un sistema monoproceso, solo existe un proceso en ejecución en cada instante, y no comienza otro proceso hasta que no finalice el anterior. Esto como es evidente presenta problemas de rendimiento, pues el sistema puede estar haciendo otras operaciones mientras el proceso por ejemplo esta leyendo de disco, o mostrando cualquier cosa en pantalla. Para aprovechar estos espacios de tiempo en que el sistema se encuentra ocupado en tareas que no necesitan de la CPU apareció el concepto de multiproceso. En un sistema multiproceso un proceso ocupa la CPU durante un intervalo de tiempo, transcurrido el cual,

libera la CPU pasando a la lista de procesos en cola y `pasando el primer proceso de la cola a ocupar la CPU. Un proceso puede estar en tres estados:

- En ejecución: se ejecutan sus instrucciones en la CPU
- En espera: está a la espera de que se libere la CPU para pasar a ejecutarse
- Dormido: está a la espera de que suceda algún evento para pasar a la cola de espera y posteriormente continuar ejecutandose.

En un sistema monoprocesador, con una sola CPU (por ejemplo un PC), en un instante determinado solo hay un proceso en ejecución, pero los demás procesos pueden estar en la cola de espera o dormidos. En un sistema multiprocesador (varias CPU) si pueden existir varios procesos ejecutandose simultaneamente. Por eso no hay que confundir el concepto de multiproceso con el de multiprocesador.

El que un sistema sea monoproceso o multiproceso lo determina el sistema operativo. MSDOS era un monoproceso, en cambio Linux y las versiones modernas de Windows son de tipo multiproceso. El que un sistema sea multiprocesador o no, en cambio vendrá limitado por el número de CPU que tenga el sistema, además de usar un sistema operativo que sea multiproceso.

La concurrencia es la ejecución simultanea de procesos de modo que interaccionen entre si, existiendo una interdependencia que hay que controlar para evitar interbloqueos. En un proceso por lotes o batch no existe concurrencia, pues un proceso comienza cuando ha finalizado el anterior. Concurrencia se da cuando los procesos se ejecutan de forma paralela, tengan o no que ver entre si. En caso de que tenga algo que ver existirán puntos en los que uno u otro proceso esperará por otro, o utilizará los resultados de otro. La concurrencia suele ir asociada al concepto de multiproceso.

Para lanzar procesos desde un programa se utiliza la función fork, que realiza una copia del programa en la memoria y sigue la ejecución del mismo por puntos diferentes, dando lugar a un proceso padre (el que se ejecuto inicialmente) y un proceso hijo, generado con la llamada a fork.

La llamada a fork crea un proceso hijo a partir del punto en que se invoca y devuelve un entero que es 0 cuando se habla del proceso hijo, un entero mayor que 0 cuando es el proceso padre, que es el pid del mismo. Si el valor es menor que cero es que se ha producido un error.

Ejemplo:

```
#include <unistd.h>                                /* Biblioteca para utilizar fork() */
/
#include <time.h>                                   /* Biblioteca para usar time()*/
#include <iostream>
using namespace std;

int main (int argc, char **argv) {
    int i;
    /*****
    Lanza un proceso hijo
    Imprimen la iteración actual y esperar un tiempo máximo de 1.5 sg
    El proceso padre no espera a que termine el hijo
    *****/
    switch (fork()) {
        case 0: /* Hijo */
            for (i = 0; i < 20; i++) {
                cout << "Hijo " << i << endl;
                /*Esperamos como máximo 1 segundo y medio */
                int valor = (int) (1500000.0*rand()/(RAND_MAX+1.0));
                usleep (valor);
            }
            break;

        case -1:
            cout << "Error: no pude lanzar al hijo" << endl;
            break;

        default: /* Padre */
            /*Regeneramos la semilla de los numeros aleatorios*/
            srand(time(NULL));
            for (i = 0; i < 20; i++) {
                /* Esperamos como máximo un segundo y medio */
                int valor = (int) (1500000.0*rand()/(RAND_MAX+1.0));
                cout << "Padre " << i << endl;
                usleep(valor);
            }
    }
}
```

Salida por pantalla:

```
Hijo 0
Padre 0
Padre 1
Padre 2
Hijo 1
Padre 3
```





```

stdin -> 0
stdout -> 1
stderr -> 2

```

Para evitar tener varias copias del programa en memoria, como sucede en estos casos, se pueden usar las ipc que es otra forma de concurrencia.

#### IPC (Inter Process Communication)

Es un mecanismo de concurrencia, que se puede implementar mediante:

- Colas de mensajes, enviando con send y recibiendo con receive. Son de tipo FIFO
- Semáforos, tienen varios estados y permiten acceder o no a una determinada zona
- Memoria Compartida, consiste en usar las mismas variables, de modo que sean compartidas. Para ello será necesario que cada proceso tenga punteros a dicha zona de memoria.

Para ello será necesario que cada proceso tenga punteros a dicha zona de memoria.

Para trabajar con colas de mensajes, hay que:

- Crear la cola
- Enviar Mensajes
- Recibir Mensajes
- Destruir la cola, aunque a veces puede desearse que la cola siga activa para un futuro.

uro.

Ejemplo:

Cola.h

```

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define FICHERO "/sender.c"
#define TIPO_MENSAJE 1
#define TIPO_CONTADOR 2
#define TIPO_FIN 3

```

```

struct mensaje {
    long tipo;
    char buf[255];
};

```

```

struct contador {
    long tipo;
    long valor;
};

```

Sender.c

```

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
/*#include <stdio.h>*/
#include <iostream>
using namespace std;
#include "colas.h"

int main (int argc, char ** argv)
{
    key_t key = ftok (FICHERO, 0);           // Se obtiene una clave para la cola
, para identificarla, pero aun no esta creada.
    struct mensaje msg;
    struct contador cont;
    struct msqid_ds info;
    int rc;

    int q_id = msgget (key, IPC_CREAT | 0666); // Obtener la cola de mensajes, crearla si hace falta con permisos rw para todos
    if (q_id < 0) {                             // Si no se ha podido crear,
        mostrar el error. Por defecto la cola se genera con
        cerr << "msgget";                       // permiso de lectura únicamente.
        exit (-1);
    }

    cout << "Cola creada" << endl;

    msg.tipo = TIPO_MENSAJE;                    // A cada estructura se le indica el tipo de mensaje
    cont.tipo = TIPO_CONTADOR;

```

```

        for (cont.valor = 0; cin.getline(msg.buf, sizeof (msg.buf)); cont.valor++)
        {
//Mientras se lea del teclado hasta el Ctrl-D
            rc = msgsnd (q_id, (struct msgbuf *)&msg, sizeof (msg.buf), 0);
// Se envía el mensaje, para ello se hace un cast
            if (rc < 0) {
// del mensaje a la estructura necesaria para la
                cerr << "msgsnd 1";
// cola y se indica el tamaño de dicho mensaje.
                break;
// Si hay un error indicarlo y finalizar.
            }

            rc = msgsnd (q_id, (struct msgbuf *)&cont, sizeof (cont.valor), 1);
// Se envía un contador a la cola del mismo modo
            if (rc < 0) {
// que antes. Si hay error indicarlo y finalizar.
                cerr << "msgsnd 2";
                break;
            }
        } // Finaliza con Ctrl-D y se
finaliza el envío.

        cont.tipo = TIPO_FIN;
        rc = msgsnd (q_id, (struct msgbuf *)&cont, sizeof (cont.valor), 1);
        if (rc < 0) {
            cerr << "msgsnd 3";
        }

        rc = msgctl(q_id, IPC_RMID, &info); // Destruir la cola
        if (rc < 0) {
            cerr << "msgctl IPC_RMID";
            exit (-1);
        }
    }
}

```

## Reciever.c

```

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <iostream>
using namespace std;
#include "colas.h"

int main (int argc, char ** argv)
{
    key_t key = ftok (FICHERO, 0);
    struct mensaje msg;
    int rc, bandera;
    long tipo = 0;

    if (argc >= 2) {
        tipo = atol (argv[1]);
    }
    cout << "Recibiendo mensajes tipo " << tipo << endl;

    int q_id = msgget (key, IPC_CREAT | 0666); // Obtener la cola de mensajes,
crearla si hace falta, */
    if (q_id < 0) {
        cerr << "msgget";
        exit (-1);
    }

    for (bandera = -1; bandera;) {
        rc = msgrcv (q_id, (struct msgbuf *)&msg, sizeof (msg.buf), tipo, 0)
;
        if (rc < 0) {
            cerr <<"msgrcv";
            break;
        }
    }
}

```

```

switch (msg.tipo)
{
case TIPO_MENSAJE:
    cout << "Mensaje: " << msg.buf << endl;
    break;
case TIPO_CONTADOR:
    cout << "Contador " << ((struct contador *)&msg)->val
or << endl;

    break;
case TIPO_FIN:
    cout << "Fin de datos" << endl;
    bandera = 0;
    break;
default:
    cout << "Tipo de mensaje desconocido: " << msg.tipo
<< endl;
}
}
exit (0);
}

```

La cola puede ser creada por el generador o el receptor.

Ejemplo de la salida por pantalla:

```

sender // Se introduce lo siguiente por teclado en la venta
na del sender Cola creada
hola que tal
muy bien

recieve // En otra ventana al estar activa se produce la sal
ida Recibiendo mensajes tipo 0
Mensaje: hola que tal
Contador 0
Mensaje: muy bien
Contador 1

```

Las funciones msgget y msgrcv envían y reciben en colas  
Las funciones semget y semrcv envían y reciben en semáforos  
Las funciones shmget y shmrcv envían y reciben en colas

Los semáforos se utilizan para controlar el acceso a una zona de memoria por parte de difere  
ntes procesos. Existen tres diferentes estados:

```

Ocupado
Libre
Acceso

```

Ejemplo:

Comun.h

```

#define TAM_PILA 10
#define FICHERO "/home/mario/src/test/concurrencia"

#define NUM_SEM 3
#define SEM_OCUPADO 0
#define SEM_LIBRE 1
#define SEM_ACCESO 2

struct pila {
    int indice;
    int elementos[10];
};

```

Consumidor.c

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include "comun.h"

int main (int argc, char **argv) {
    struct pila * buffer;
    int shmid, semid, rc, valor;
    struct sembuf oper[2];
    struct timespec interval, remain;

```

```

key_t key = ftok (FICHERO, 0);

/* Crear u obtener el segmento de memoria compartida */
shmid = shmget (key, sizeof (struct pila), IPC_CREAT | 0600);
if (shmid < 0) {
    perror ("shmget");
    exit (-1);
}

/* Mapear el segmento de memoria a nuestro puntero */
buffer = (struct pila *)shmat (shmid, NULL, 0);
if (buffer == NULL) {
    perror ("shmat");
    exit (-1);
}

/* Obtener el conjunto de semáforos */
semid = semget (key, NUM_SEM, IPC_CREAT | 0600);
if (semid < 0) {
    perror ("semget");
    exit (-1);
}

/* Bucle sin fin */
for (;;) {
    /* Obtenemos los dos semáforos de la cola */
    oper[0].sem_num = SEM_ACCESO;
    oper[0].sem_op = -1; /* Bloquear si hay alguien */
    oper[0].sem_flg = 0; /* Queremos que espere y no queremos Undo */

    oper[1].sem_num = SEM_OCUPADO;
    oper[1].sem_op = -1; /* Esperar a que haya datos */
    oper[1].sem_flg = 0; /* Queremos que espere y no queremos Undo */

    semop (semid, oper, 2);

    valor = buffer->elementos [--buffer->indice];
    /*
     * Con esto no necesitamos el semáforo SEM_ACCESO siempre y cuando haya solo
     * un productor y un consumidor
     */
    buffer->elementos [semctl (semid, SEM_OCUPADO, SEM_GETVAL)]
    /*
    printf ("Consumidor: %d\n", valor);

    interval.tv_sec = 0;
    interval.tv_nsec = ((double)random())/RAND_MAX*1000000000;
    nanosleep (&interval, &remain);

    oper[0].sem_num = SEM_LIBRE;
    oper[0].sem_op = 1; /* Incrementar el valor de libre */
    oper[0].sem_flg = 0; /* Queremos que espere y no queremos Undo */

    oper[1].sem_num = SEM_ACCESO;
    oper[1].sem_op = 1; /* Permitir el acceso */
    oper[1].sem_flg = 0; /* Queremos que espere y no queremos Undo */

    semop (semid, oper, 2);
}
}

```

Productor.c

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include "comun.h"

int main (int argc, char **argv) {
    struct pila * buffer;
    int shmid, semid, rc, valor, contador;
    struct sembuf oper[2];
    struct timespec interval, remain;
    /*
    union semun sem_val;

```

```

*/

key_t key = ftok (FICHERO, 0);

/* Crear el segmento de memoria compartida */
shm_id = shmget (key, sizeof (struct pila), IPC_CREAT | 0600);
if (shm_id < 0) {
    perror ("shmget");
    exit (-1);
}

/* Mapear el segmento de memoria a nuestro puntero */
buffer = (struct pila *)shmat (shm_id, NULL, 0);
if (buffer == NULL) {
    perror ("shmat");
    exit (-1);
}

buffer->indice = 0;
contador = 0;

/* Obtener el conjunto de semáforos */
sem_id = semget (key, NUM_SEM, IPC_CREAT | 0600);
if (sem_id < 0) {
    perror ("semget");
    exit (-1);
}

/* Inicializar los semáforos */
semctl(sem_id, SEM_OCUPADO, SETVAL, 0);
semctl(sem_id, SEM_LIBRE, SETVAL, sizeof (buffer->elementos)/sizeof (buffer->elemen
tos[0]));
semctl(sem_id, SEM_ACCESO, SETVAL, 1);

/* Bucle sin fin */
for (;;) {
    /* Obtenemos los dos semáforos de la cola */
    oper[0].sem_num = SEM_ACCESO;
    oper[0].sem_op = -1; /* Bloquear si hay alguien */
    oper[0].sem_flg = 0; /* Queremos que espere y no queremos Undo */

    oper[1].sem_num = SEM_LIBRE;
    oper[1].sem_op = -1; /* Esperar a que haya un hueco */
    oper[1].sem_flg = 0; /* Queremos que espere y no queremos Undo */

    semop (sem_id, oper, 2);

    valor = (int)random();
    buffer->elementos [buffer->indice++] = valor;
    /*
        Con esto no necesitamos el semáforo SEM_ACCESO siempre y cuando solo haya
        un productor y un consumidor

        buffer->elementos [semctl (sem_id, SEM_OCUPADO, SEM_GETVAL)]
    */
    printf ("Productor [%d]: %d\n", contador++, valor);

    interval.tv_sec = 0;
    interval.tv_nsec = ((double)random())/RAND_MAX*1000000000;
    nanosleep (&interval, &remain);

    oper[0].sem_num = SEM_OCUPADO;
    oper[0].sem_op = 1; /* Incrementar el valor de ocupado */
    oper[0].sem_flg = 0; /* Queremos que espere y no queremos Undo */

    oper[1].sem_num = SEM_ACCESO;
    oper[1].sem_op = 1; /* Permitir el acceso */
    oper[1].sem_flg = 0; /* Queremos que espere y no queremos Undo */

    semop (sem_id, oper, 2);
}
}

```

## REDES:

A continuación veremos un ejemplo en el que un servidor esta escuchando y esperando conexión es a través de un puerto (1234). Cuando un cliente se conecta a la máquina en que esta activo el servidor y al puerto en cuestión, el servidor muestra un mensaje y cierra la conexión.

Cliente:

```

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <iostream>
using namespace std;

int main (int argc, char ** argv) {
    int sockfd, rc, received;
    struct sockaddr_in addr;
    char buf[255];

    sockfd = socket (PF_INET, SOCK_STREAM, 0);           // Se crea el socket

    addr.sin_family = AF_INET;
    addr.sin_port = htons (atoi (argv[2]));           // Puerto al que se conectar
a
    addr.sin_addr.s_addr = inet_addr (argv[1]);         // Dirección IP a la que hay
que conectar

    memset (&(addr.sin_zero), '\0', sizeof (addr.sin_zero));

    rc = connect(sockfd, (struct sockaddr *)&addr, sizeof (addr));           // Conectar
a la IP y puerto dado

    for (;;) {
        received = recv(sockfd, (void *)buf, sizeof (buf)-1, 0);           //recibir mi
entras haya datos
        if (received == 0) {
            break;
        }
        buf[received] = '\0';
        cout << buf << endl;           // muestra los datos recibidos
    }           // no cierra la conexion en este cas
o por que lo hace el servidor
}

```

Servidor:

```

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>
#include <string.h>

int main (int argc, char **argv) {
    int serv_sockfd, sockfd, rc;
    struct sockaddr_in addr, client_addr;
    int client_addr_len, indice, longitud;
    char * mensaje = "hola caracola\n\n";

    serv_sockfd = socket (PF_INET, SOCK_STREAM, 0);           // Se crea y abre el socket
que se utilizara

    addr.sin_family = AF_INET;           // Preparar la dirección IP local
    addr.sin_port = htons (1234);         // Familia de protocolo con que se va a trab
ajar en este caso TCP/IP
    addr.sin_addr.s_addr = htonl (INADDR_ANY);           // Puerto htons convierte un numero decimal
al formato que sea entendible

    memset (&(addr.sin_zero), '\0', sizeof (addr.sin_zero));

    /* Conectar el socket a la dirección local y preparar para escuchar */
    rc = bind (serv_sockfd, (struct sockaddr*)&addr, sizeof (addr));           // Se conect
a
    rc = listen(serv_sockfd, 10);           // Permanece
a la escucha hasta un máximo de 10 clientes

    longitud = strlen (mensaje);

    for (;;) {           // El programa se queda parado hasta que se provoca una seña
l a ese puerto, lo cual se consigue con accept
        sockfd = accept(serv_sockfd, (struct sockaddr *)&client_addr,(socklen_t
*)&client_addr_len);
        for (indice = 0; indice < longitud; ) {           // Una vez existe un
a conexion se envían datos
            indice += send (sockfd, mensaje+indice, longitud-indice, 0);
        }
        rc = close (sockfd);           // se cierra la conexion
    }
}

```

Salida:

Se arranca el servidor desde una ventana:

Desde otra se puede hacer un telnet:

```
telnet 1.1.1.81 1234
Trying 1.1.1.81...
Connected to 1.1.1.81.
Escape character is '^]'.
hola caracola
```

Connection closed by foreign host.

O bien ejecutar el cliente:

```
cliente 1.1.1.81 1234
hola caracola
```