# Tema 3: C++ Parte II

**Arrays vs punteros, Consideraciones Prácticas**

Existe una equivalencia casi total entre arrays y punteros

1→ Declaramos un puntero del mismo tipo que los elementos del array, y que apunta al primer elemento del array.
2 →Reservamos memoria para todos los elementos del array. Los elementos de un array se almacenan internamente en el ordenador en posiciones consecutivas de la memoria.
3 →el nombre de un array es un **puntero constante**, no podemos hacer que apunte a otra dirección de memoria.
4 → El compilador asocia una zona de memoria para los elementos del array, cosa que no hace para los elementos apuntados por un puntero auténtico.

Ejemplo:

```
int vector[10];
int *puntero;
puntero = vector; /* Equivale a puntero = &vector[0];
esto se lee como "dirección del primer de vector" */
(*puntero)++; /* Equivale a vector[0]++; */
puntero++; /* puntero equivale a &vector[1] */
```

¿Qué hace cada una de estas instrucciones?:
La primera incrementa el contenido de la memoria apuntada por "puntero", que es vector[0].
La segunda incrementa el puntero, esto significa que apuntará a la posición de memoria del siguiente "int", pero no a la siguiente posición de memoria. El puntero no se incrementará en una unidad, como tal vez sería lógico esperar, sino en la longitud de un "int".

```
int main(int argc, char *argv[])
{
Objeto vector[10];
Objeto *puntero;
for(int i=0;i<10;i++) vector[i].nombrar();;
puntero = vector; // Equivale a puntero =
&vector[0];
puntero +=5; //Avanza 5 posiciones en
tamaños de Objeto
cout <<"\n";
for(int i=0;i<5;i++) {
        puntero->nombrar();
        (*puntero).nombrar();
        puntero++;
}
```

Uso del Puntero Generico y el operador de casting

```
#include <iostream>
using namespace std;
int main() {
char cadena[10] = "Hola";
char *c;
int *n;
void *v;
c = cadena; // c apunta a cadena
n = (int *)cadena; // n también apunta a cadena
v = (void *)cadena; // v también
cout << "carácter: " << *c << endl;
cout << "entero: " << *n << endl;
cout << "float: " << *(float *)v << endl;
cin.get();
return 0;
}
```

El resultado será:

```
    carácter: H
    entero: 1634496328
    float: 2.72591e+20
```

**Punteros como parámetros de funciones: referencia y valor**

```
#include <iostream>
using namespace std;
void funcion(int *q);
int main() {
int a;
int *p;
a = 100;
p = &a;
// Llamamos a funcion con un puntero funcion(p);
cout << "Variable a: " << a << endl;
cout << "Variable *p: " << *p << endl;
// Llamada a funcion con la dirección de "a" (constante)
funcion(&a);
cout << "Variable a: " << a << endl;
cout << "Variable *p: " << *p << endl;

}
void funcion(int *q) {
// Cambiamos el valor de la variable apuntada por
// el puntero
*q += 50;
q++;
}
//para hacerlo por referencia

void funcionRef(int *&q) {
// Cambiamos el valor de la variable apuntada por
// el puntero y el puntero
*q += 50;
q++;
}
```

**Arrays como parámetros de funciones**

Se pasa un puntero al primer elemento.

- si sólo pasamos el nombre del array de más de una dimensión no podremos
- acceder a los elementos del array mediante subíndices, ya que la función no tendrá información sobre el tamaño de cada dimensión.→ declarar el parámetro como un array

```
void funcion(int tabla[][10]) {
...
cout << tabla[2][4] << endl;
}
int main() {
     int Tabla[5][10];
     ...
     funcion(Tabla);
     ...
     return 0;
}
```

**Arrays Dinámicos**

los arrays tienen un inconveniente: hay que definir su tamaño y después no puede ser modificado.

Solución : punteros a puntero : `int **tabla;  //Puntero a puntero a int`

```
#include <iostream>
using namespace std;
int main() {
int **tabla;
int n = 134;
int m = 231;
int i;
// Array de punteros a int:
tabla = new int*[n];
// n arrays de m ints
for(i = 0; i < n; i++)
tabla[i] = new int[m];
tabla[21][33] = 123;
cout << tabla[21][33] << endl;
// Liberar memoria:
for(i = 0; i < n; i++) delete[] tabla[i];
delete[] tabla;
cin.get();
return 0;
}
```

**Funciones friend**

Dentro de una clase se pueden declarar funciones como amigas. Las funciones amigas no son métodos de clase sino funciones que se definen realmente fuera del ámbito de la clase, esto es, son accesibles desde nuestro programa como una llamada a función normal, no como una invocación de método.

Las funciones amigas de una clase tienen acceso a todos los atributos y métodos de la clase, incluyendo tanto públicos como privados y protegidos.

Son un detalle excepcional de C++ y violan el principio de encapsulamiento de la programación orientada a objetos. Sin embargo, son útiles en ciertas ocasiones. Sobre todo para determinados casos de sobrecarga operadores.

| Punto |
| --- |
| - int x<br>- int y |
| + Punto( int x=0, int y=0 )<br>+ void pinta()<br>**Friend** Punto sumapuntos<br>( const Punto p1, const Punto p2 ) |

```
#include <cstdlib>
#include <iostream>

using namespace std;

class Punto{
   // función amiga
   friend Punto sumapuntos( const Punto p1, const Punto p2 );
private:
   int x;
   int y;
public:
   Punto( int nx=0, int ny=0 ){ x=nx; y=ny; }
   void pinta(){ cout << "(" << x << "," << y << ")\n"; }
};

Punto sumapuntos( Punto p1, Punto p2 ){
   Punto res;
   /* ¡¡ Acceso a atributos privados !!*/
   res.x = p1.x + p2.x;
   res.y = p1.y + p2.y;
   return res;
}

int main(int argc, char *argv[])
{
   Punto p1(10,10), p2(20,20);
   Punto p3 = sumapuntos( p1,p2 );
   p3.pinta();

   system("PAUSE");
   return EXIT_SUCCESS;
}
```

**Clases friend**

Dentro de una clase se pueden declarar otras clases como amigas. Las clases amigas tendrán acceso a todos los atributos y métodos de la clase, incluyendo tanto públicos como privados y protegidos.

| Punto |
| --- |
| - int x<br>- int y |
| + Punto( int x=0, int y=0 )<br>+ void pinta()<br>**Friend** class Amiga |

| Amiga |
| --- |
| |
| + void ponACero( Punto &p ) |

```
#include <cstdlib>
#include <iostream>

using namespace std;

class Punto{
   friend class Amiga; // clase amiga
private:
   int x;
   int y;
public:
   Punto( int nx=0, int ny=0 ){ x=nx; y=ny; }
   void pinta(){ cout << "(" << x << "," << y << ")\n"; }
};

class Amiga{
public:
   void ponACero( Punto &p ){
      p.x = 0;
      p.y = 0;
   }
};

int main(int argc, char *argv[])
{
   Punto p1(10,10);
   Amiga a;
   a.ponACero( p1 );
   p1.pinta();

   system("PAUSE");
   return EXIT_SUCCESS;
}
```

**Clases friend (caso de referencia cruzada)**

Es posible que tengamos dos clases tales que la primera deba ser amiga de la segunda y la segunda de la primera también. En este caso, tenemos el problema de que no podemos utilizar una clase antes de declararla. La solución consiste en **redirigir la declaración de la clase** que definimos en segundo lugar.

Cuando se redirige una clase, se pueden declarar variables de ese tipo y punteros. Básicamente, podemos declarar los prototipos pero, como aún no se han declarado los atributos y métodos reales de la clase redirigida, no se pueden invocar. La solución es sólo hacer las declaraciones primero y luego definir los métodos.

| Amiga1 |
| --- |
| - int privada |
| + void modificaAmiga2<br>( Amiga2 &a2, int val )<br>+ void pinta()<br>**Friend** class Amiga2 |

| Amiga2 |
| --- |
| - int privada |
| + void modificaAmiga1<br>( Amiga1 &a1, int val )<br>+ void pinta()<br>**Friend** class Amiga1 |

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

class Amiga2; // Redirección de la declaración

class Amiga1{
   friend class Amiga2;
private:
   int privada;
public:
   void modificaAmiga2( Amiga2 &a2, int val );
   /* Aquí no podemos definir el método porque aún no hemos declarado
que la clase Amiga2 tiene un atributo int privada... */
   void pinta(){ cout << privada << "\n"; }
};

class Amiga2{
   friend class Amiga1;
private:
   int privada;
public:
   void modificaAmiga1( Amiga1 &a1, int val );
/* Aquí sí podríamos definir porque ya hemos declarado Amiga1 */
   void pinta(){ cout << privada << "\n"; }
};

/* Ahora sí que podemos definir el método porque ya hemos declarado
que la clase Amiga2 tiene un atributo int privada... */
void Amiga1::modificaAmiga2( Amiga2 &a2, int val ){
   a2.privada = val;
}
void Amiga2::modificaAmiga1( Amiga1 &a1, int val ){
   a1.privada = val;
}

int main(int argc, char *argv[])
{
   Amiga1 a1;
   Amiga2 a2;
   a1.modificaAmiga2(a2,10);
   a2.modificaAmiga1(a1,20);
   a1.pinta();
   a2.pinta();

   system("PAUSE");
   return EXIT_SUCCESS;
}
```

**Sobrecarga de Operadores**

Operadores que tienen varios usos, como por ejemplo *,+,-

C++ podemos definir varias funciones con el mismo nombre, con la única condición de que el número y/o el tipo de los parámetros sean distintos

Si ninguna de las funciones se adapta a los parámetros indicados, se aplicarán las reglas implícitas de conversión de tipos.

No se pueden definir nuevos operadores ni cambiar la aridad.
Se pueden sobrecargar como funciones o como métodos de clase.

En los operadores definidos como métodos miembros, el objeto perteneciente a la clase dónde se define el método es el primer operando siempre, de modo que si queremos que nuestros objetos puedan operar funcionando como operandos en la parte derecha, será necesario hacerlo en una función (posiblemente friend) o en un método de la clase del operando que quede a la izquierda (posiblemente friend)…

Prototipo:

```
<tipo> operator <operador> (<argumentos>);
```

Definición:

```
<tipo> operator <operador> (<argumentos>)
{
<sentencias>;
}
```

```
class complejo
{public:
float a,b;
 complejo( int entera=0, int compleja=0){
           a=entera;
           b=compleja;
           }
};
complejo operator +(complejo a, complejo b) {
complejo temp (a.a+b.a, a.b+b.b);
return temp;
}
int main() {
complejo x(10,32);
complejo y (21,12);
complejo z;
/* Uso del operador sobrecargado + con complejos */
z = x + y;
cout << z.a << "," << z.b << endl;
```

La forma adecuada de implementar el operador es con un parámetro:

```
class complejo
{public:
float a,b;
 complejo( int entera=0, int compleja=0){
           a=entera;
           b=compleja;
           }


complejo operator +(complejo comp) {
         complejo temp;
temp.a+=a+ comp.a;
temp.b+=b+ comp.b;
return temp;
}
};
int main() {
complejo x(10,32);
complejo y (21,12);
complejo z;
/* Uso del operador sobrecargado + con complejos */
z = x + y;
cout << z.a << "," << z.b << endl;
}
```

También hemos usado el operador =, a pesar de que nosotros no lo hemos definido. Esto es porque el compilador crea un operador de asignación por defecto si nosotros no lo hacemos.

**Sobrecarga Operador =**

Si en nuestro programa declaramos dos objetos de tipo Cadena, y los asignamos, estaremos apuntando al mismo objeto cadena en lugar de producir una copia.

```
class Cadena {
public:
        Cadena(char *cad);
        Cadena() : cadena(NULL) {};
        ~Cadena() { delete[] cadena; };
        void Mostrar() const;
private:
        char *cadena;
};
Cadena::Cadena(char *cad) {
        cadena = new char[strlen(cad)+1];
        strcpy(cadena, cad);}
```

```
Cadena &Cadena::operator=(const Cadena &c) {
        if(this != &c) {//si soy yo hacer nada
              delete[] cadena;
              if(c.cadena) {//si no es NULL
                    cadena = new char[strlen(c.cadena)+1];
                    strcpy(cadena, c.cadena);
              }
               else cadena = NULL;
       }
      return *this;
}
```

Esto permite realizar asignaciones del tipo

```
        Cadena c1("hola"), c2;
        c2=c1;
```

**Sobrecarga de operadores unarios**

```
class complejo {
…
complejo operator ++(int sufijo) {// el parámetro no se utiliza, pero lo diferencia del prefijo
   a++;
   b++;
return *this;
}
complejo operator ++() {//prefijo
   ++ a;
   ++b;
return *this;
}

};
int main() {
complejo z(10,32);
z++;
cout << z.a << "," << z.b << endl;
++z;
cout << z.a << "," << z.b << endl;
}
```

Los operadores =, [], () y -> sólo pueden sobrecargarse en el interior de clases.

## Sobrecarga de <<

| Punto |
|---|
| - static int numserie<br>- int *identificador<br>+ int x<br>+ int y |
| + Punto( int x, int y )<br>+ ~Punto()<br>+ int operator==( Punto p )<br>+ Punto operator=( Punto p )<br>**Friend**: ostream& operador<<<br>( ostream &salida,<br>const Punto p ) |

En funciones friend:
Primer parámetro: operando de izquierda.
Segundo parámetro: operando de la derecha (en binarios).
En miembros, a la izquierda va el objeto.
Se usan friend para conmutatividad también.

Cuando una función friend se define fuera de la clase, no se escribe friend ni el resolutor de ámbito. La función no es realmente un método.

Existen casos especiales con el operador de asignación cuando el objeto que se copia se declaró como un puntero. Hay que comprobar si se copia el mismo o si se le asigna un NULL.

```
#include <cstdlib>
#include <iostream>

using namespace std;

class Punto{

friend ostream& operator<<( ostream &salida, const Punto p ){
      // acceso a datos protected y private...
      salida << *p.identificador <<":(" << p.x << "," << p.y << ")";
      return salida;
}

private:
   static int numserie;
   int *identificador;
public:
   int x;
   int y;
   Punto(int nx, int ny){ x=nx; y=ny; identificador=new int(numserie++); }
   ~Punto(){ delete identificador; }
   int operator==( Punto p ){ return (x==p.x)&&(y==p.y); }
   Punto operator=( Punto p )
      { x=p.x, y=p.y; /*no se sobrescribe la serie*/
        return p;}
};
int Punto::numserie = 0;

int main(int argc, char *argv[])
{
   Punto p1(10,15), p2(10,15), p3(0,0);
   cout << p1 << ((p1==p2)?"==":"!=") <<p2 << "\n" ;
   p2.x++;
   cout << p1 << ((p1==p2)?"==":"!=") <<p2 << "\n" ;
   p3=p2=p1;
   cout << p1 << " " << p2 << " " << p3 << "\n";
   cout << p1 << ((p1==p3)?"==":"!=") <<p3 << "\n" ;

   system("PAUSE");
   return EXIT_SUCCESS;
}
```

**Operadores sobrecargables:**

```
+  -  *  /  %  ^  &  |  ~  !  =  <  >  +=  -=  *=  /=  %=  ^=  &=  |=  <<  >>  >>=  <<=  ==  !=  <=
>=  &&  ||  ++  --  ->*  , ->  []  ()  new  new[]  delete  delete[]
```

**Operadores no sobrecargables:**

```
.  .*  ::  ?:  sizeof
```

Con la sobrecarga del operador << es necesario utilizar la funcion friend. Es posible tener comportamiento similar sobrecargando char*

**Conversiones de Tipos**

```
Tiempo T1(12,23);
        unsigned int minutos = 432;
T1 += minutos;
```

Con toda probabilidad no obtendremos el valor deseado, no hemos definido el comportamiento para un casting entero a *Tiempo*.

Para ello hace falta sobrecargar el constructor:

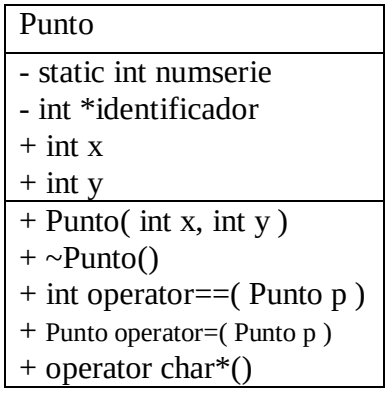
```
Tiempo(unsigned int m) : hora(0), minuto(m) {
        while(minuto >= 60) {
        minuto -= 60;
        hora++;
        }
}
```

Pero si lo que queremos es producir un entero:

```
Tiempo T1(12,23);
int minutos;
minutos = T1;
```

```
class Tiempo {
        ...
        operator int();
...
        operator int() {
            return hora*60+minuto;
        }
```

**Ejemplo 2: Operador de conversión de tipo , igualdad y asignación**

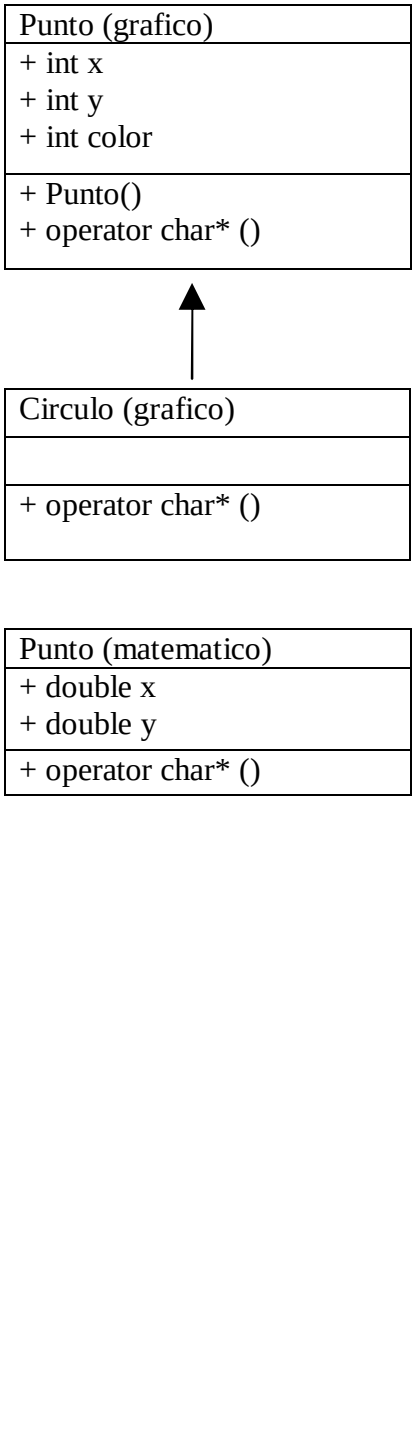



```
#include <cstdlib>
#include <iostream>
#include <stdio.h>
#include <string.h>

using namespace std;

class Punto{

private:
   static int numserie;
   int *identificador;
public:
   int x;
   int y;
   Punto(int nx, int ny){ x=nx; y=ny; identificador=new int(numserie++); }
   ~Punto(){ delete identificador; }
   int operator==( Punto p ){ return (x==p.x)&&(y==p.y); }
   Punto operator=( Punto p ){
          x=p.x, y=p.y;/*no se sobrescribe la serie*/
          return p;}

   operator char*(){
      char salida[30];
      sprintf( salida, "%i:(%i,%i)",*identificador, x, y );
      return strdup( salida );
   }
};
int Punto::numserie = 0;


int main(int argc, char *argv[])
{
   Punto p1(10,15), p2(10,15);
   cout << p1 << ((p1==p2)?"==":"!=") <<p2 << "\n" ;

   system("PAUSE");
   return EXIT_SUCCESS;
}
```

**Espacios con nombre**

```
namespace [<identificador>] {
...
<declaraciones y definiciones>
...
```

```
#include <iostream>
namespace uno {
int x;
}
namespace dos {
int x;
}
using namespace uno;
int main() {
x = 10;
dos::x = 30;
std::nacout << x << ", " << dos::x << std::endl;
std::cin.get();
return 0;
}
```




```
#include <cstdlib>
#include <iostream>

using namespace std;

namespace grafico{

   class Punto{
   public:
      int x;
      int y;
      int color;
      Punto(){ x=0; y=0; }
      operator char*(){ return "Punto gráfico\n"; }
   };
   class Circulo: public Punto{
   public:
      operator char*(){ return "Círculo\n"; }
   };

}

namespace matematico{

   class Punto{
   public:
      double x;
      double y;
      operator char*(){ return "Punto matemático\n"; }
   };
}

using namespace grafico;

int main(int argc, char *argv[])
{
   Punto p1;
   matematico::Punto p2;
   grafico::Circulo c1;

   cout << p1 << p2 << c1;

   system("PAUSE");
   return EXIT_SUCCESS;
}
```

## Funciones miembro constantes

Cuando una función miembro no modifique el valor de la clase, se debe declarar **constante** .

Se pueden crear **objetos constantes**. Los métodos de estos objetos no modifican el estado del objeto.
**Sintaxis**
**const** detrás del nombre y la lista de argumentos del método. Los métodos definidos como **const** están disponibles para los objetos normales.

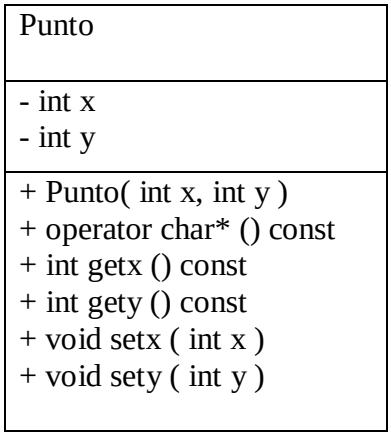
Dentro de un método **const** no se pueden modificar los atributos de clase y sólo se puede llamar a los métodos **const** de la clase.

```
class Ejemplo2 {
public:
        Ejemplo2(int a = 0) : A(a) {}
        void Modifica(int a) { A = a; }
        int Lee() const { return A; }
private:
        int A;
};
int main() {
        Ejemplo2 X(6);
        cout << X.Lee() << endl;
        X.Modifica(2);
        cout << X.Lee() << endl;
}
```

## Valores de retorno constantes

```
class cadena {
public:
     cadena(char *c){cad=c;}
 const char *Leer() {
       return cad; // el compilador no dejará modificarlo
}
private:
        char *cad;
};
int main() {
        char *cadena2;
        cadena Cadena1("hola");
        cout << Cadena1.Leer() << endl; // Legal

        cadena2= Cadena1.Leer() ; // Ilegal
        Cadena1.Leer()[1] = 'O'; // Ilegal
}
```

La declaración de variables constantes evita que se puedan utilizar métodos que no sean constantes.
**Const Punto p(0,0).** // cuando un punto de origen no debe cambiar durante la ejecución.
**Otros ejemplos:**




```
#include <cstdlib>
#include <iostream>
#include <stdio.h>
#include <string.h>

using namespace std;

class Punto{
private:
   int x;
   int y;
public:
   Punto( int nuevax, int nuevay ){
      x = nuevax;
      y = nuevay;
   }
   /* Los métodos de cambio de las coordenadas no
   deben ser invocados por objetos constantes */
   void setx( int nuevax ){ x = nuevax; }
   void sety( int nuevay ){ y = nuevay; }

   /* Los métodos de lectura de las coordenadas sí
   deben ser invocados por objetos constantes */
   int getx() const { return x; }
   int gety() const { return y; }

   /* La conversión también es accesible para constantes */
   operator char*() const {
      char salida[30];
      sprintf( salida, "(%i,%i)",x,y );
      return strdup( salida );
   }
};

int main(int argc, char *argv[])
{
   Punto p1(10,20);
   const Punto p2(5,5);
   cout << p1 << ":" << p2 << "\n";
   p1.setx(40);
   // p2.setx(100);  ¡ no se permite !
   cout << p1 << ":" << p2 << "\n";

   system("PAUSE");
   return EXIT_SUCCESS;
}
```

**const**

| Punto |
| --- |
| - int x<br>- int y |
| + Punto( int x, int y )<br>+ operator char* () const<br>+ int getx () const<br>+ int gety () const<br>+ void setx ( int x )<br>+ void sety ( int y )<br>+ void setx ( int x ) const<br>+ void sety ( int y ) const |

Se pueden hacer dos versiones para un método, una const y la otra no…

```
#include <cstdlib>
#include <iostream>
#include <stdio.h>
#include <string.h>

using namespace std;

class Punto{
private:
   int x;
   int y;
public:
   Punto( int nuevax, int nuevay ){
      x = nuevax;
      y = nuevay;
   }
   /* Los métodos de cambio de las coordenadas no
   deben ser invocados por objetos constantes */

   void setx( int nuevax ){ x = nuevax; }
   void sety( int nuevay ){ y = nuevay; }
   void setx( int nuevax ) const{}
   void sety( int nuevay ) const{}

   /* Los métodos de lectura de las coordenadas sí
   deben ser invocados por objetos constantes */
   int getx() const { return x; }
   int gety() const { return y; }
   /* La conversión también es accesible para constantes */
   operator char*() const {
      char salida[30];
      sprintf( salida, "(%i,%i)",x,y );
      return strdup( salida );
   }
};

int main(int argc, char *argv[])
{
   Punto p1(10,20);
   const Punto p2(5,5);

   cout << p1 << ":" << p2 << "\n";
   p1.setx(40);
   p2.setx(100);  // se permite
   cout << p1 << ":" << p2 << "\n";

   system("PAUSE");
   return EXIT_SUCCESS;
}
```

**Clases proxy**

Existe un problema con las inclusiones de los *.hpp*: las declaraciones dejan ver gran parte de la implementación. Todo lo que no es público se podría ocultar al programador externo. Esto es consistente con el principio de ocultación.

## ClaseImplementacion.hpp

```
#ifndef CLASEIMPLEMENTACION_HPP
#define CLASEIMPLEMENTACION_HPP

class ClaseImplementacion{
private:
   int ocultable;
   int tope;
   void metodoInterno();
protected:
   void metodoInternoSubclases();
public:
   ClaseImplementacion( int tope );
   void apilar( int elemento );
};

#endif
```

## ClaseImplementacion.cpp

```
#include <iostream>

#include "ClaseImplementacion.hpp"

void ClaseImplementacion::metodoInterno(){}
void ClaseImplementacion::metodoInternoSubclases(){}
ClaseImplementacion::ClaseImplementacion( int ntope ){
   tope = ntope;
}
void ClaseImplementacion::apilar( int elemento ){
   std::cout << "Apilando:" << elemento;
}
```

| ClaseImplementacion |
| --- |
| - int ocultable<br>- int tope |
| - metodoInterno()<br># metodoInternoSubclase<br>+ ClaseImplementacion( int tope )<br>+ void apilar( int elemento ) |

## ClaseProxy.hpp

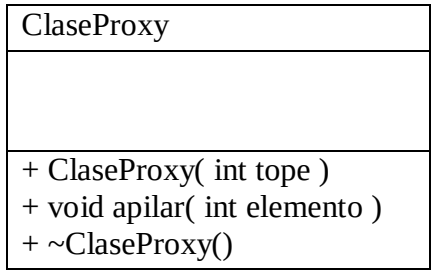
```
#ifndef CLASEPROXY_HPP
#define CLASEPROXY_HPP

class ClaseImplementacion;

class ClaseProxy{
public:
   ClaseProxy( int tope );
   void apilar( int elemento );
   ~ClaseProxy()
private:
   ClaseImplementacion *imp;
};

#endif
```

## ClaseProxy.cpp

```
#include "ClaseProxy.hpp"
#include "ClaseImplementacion.hpp"

ClaseProxy::ClaseProxy( int tope ){
   imp = new ClaseImplementacion( tope );
}
void ClaseProxy::apilar( int elemento ){
   imp->apilar( elemento );
}
ClaseProxy::~ClaseProxy(){ delete imp; }
```

## Prueba.cpp

```
#include <cstdlib>
#include <iostream>
#include "ClaseProxy.hpp"

using namespace std;

int main(int argc, char *argv[])
{
   ClaseProxy c(4);

   c.apilar( 10 );
   cout << "\n";

   system("PAUSE");
   return EXIT_SUCCESS;
}
```




Se redirige la declaración de la clase *ClaseImplementacion*

Se guarda una referencia a *ClaseImplementacion* que se inicializará en el constructor

Sólo *ClaseProxy.cpp* incluye las declaraciones en *ClaseImplementacion.hpp*, con lo que se oculta la declaración real

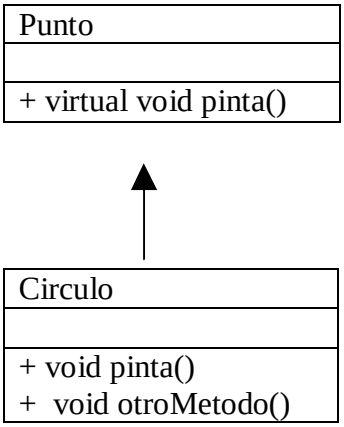
Para compilar la prueba nos basta con tener *ClaseProxy.hpp* y el objeto (o biblioteca) ya compilado de la implementación. Normalmente, ese fichero objeto (o biblioteca) incluirá el código (objeto) tanto de *ClaseImplementacion* como de *ClaseProxy*

**Static y Dinamic cast**

Información de tipo (clase) en tiempo de ejecución: <typeinfo>
typeid y dynamic_cast.

```
static_cast<>
        int a;
        float b;
b = static_cast<float>(a);
```

```
#include <iostream>
using namespace std;
class Tiempo {
public:
        Tiempo(int h=0, int m=0) : hora(h), minuto(m) {}
        void Mostrar();
        operator int() {
                return hora*60+minuto;
        }
private:
        int hora;
        int minuto;
        };
void Tiempo::Mostrar() {
        cout << hora << ":" << minuto << endl;
}
int main() {
        Tiempo Ahora(12,24);
        int minutos;
        Ahora.Mostrar();
        minutos = static_cast<int> (Ahora);
        // minutos = Ahora; // Igualmente legal, pero implícito
        cout << minutos << endl;
}
```




```
#include <cstdlib>
#include <iostream>

using namespace std;

#include <typeinfo>

class Punto{
public:
   virtual void pinta(){ cout << "Punto\n"; }
};

class Circulo:public Punto{
public:
   virtual void pinta(){ cout << "Circulo\n"; }
};

int main(int argc, char *argv[])
{
   Punto **array = new Punto * [2];
   array[0] = new Punto;
   array[1] = new Circulo;

   for (int i=0; i<2; i++){
      // Se pueden hacer comparaciones del tipo:
      // if (typeid( *array[i] )==typeid( Punto ))

      cout << typeid( *array[i] ).name() <<" ";

      if ( Circulo *c = dynamic_cast<Circulo*>(array[i]) ){
         cout << "Molde valido\n";
         c->pinta();
          c->otroMetodo();
      }else
         cout << "Molde NO valido\n";
   }

   system("PAUSE");
   return EXIT_SUCCESS;
}
```

Permite acceder con un puntero base a funciones sólo existentes en la derivada.

## String

### Cadenas con <string>

C++, en su biblioteca estándar, nos provee de la clase string, que resuelve muchos problemas clásicos con las cadenas C.

Puede buscar una referencia completa en Internet.

Por ejempo:

**http://www.msoe.edu/eecs/cese/resources/stl/string.htm**

**http://www.cppreference.com/cppstring/**

La cabecera está en **<string>**.

Existen varios constructores y se definen operadores como la concatenación (+, +=) y las comparaciones ( <, >, ==, != ).

Se pueden transformar en una cadena C normal (método c_str)

Versión C++

```
#include <cstdlib>
#include <iostream>
#include <string>

using namespace std;

int main(int argc, char *argv[])
{
   string cad1 = "Una cadena";
   string *cad2 = new string("Otra cadena");
   cad1 += " y " + *cad2 + "\n";

   cout << cad1;

   delete cad2;

   system("PAUSE");
   return EXIT_SUCCESS;
}
```

Se puede sustituir por:

```
sprintf( tmp,"%s y %s\n" cad1, cad2 );
```

Versión C

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
   char *cad1="Una cadena";
   char *cad2;
   char *tmp;
   cad2 = strdup( "Otra cadena" );

   /* Hay que reservar espacio suficiente!! */
   tmp = (char*)malloc(
            (strlen(cad1)+
            strlen(" y ")+
            strlen(cad2)+
            strlen("\n")+1)*sizeof(char) );

   strcpy( tmp, cad1 );
   strcat( tmp, " y " );
   strcat( tmp, cad2 );
   strcat( tmp, "\n" );

   printf( "%s", tmp );

   free(tmp);
   free(cad2);

   system("PAUSE");
   return EXIT_SUCCESS;
}
```

**Cadenas con <string>**

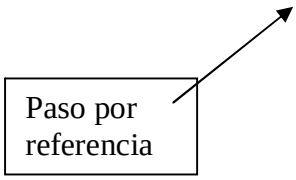
Las cadenas de C++ sobrecargan también el operador [] de modo que se pueden usar como arrays.

```
#include <cstdlib>
#include <iostream>
#include <string>

using namespace std;

int main(int argc, char *argv[])
{
   string strCPP = "Una cadena C++\n";
   char *strC = "Una cadena C\n";
   int tamCPP, tamC;

   tamCPP = strCPP.size();
   tamC = strlen( strC );

   for ( int i=0; i<tamCPP; i++ )
      cout << " " << strCPP[i];
   for ( int i=0; i<tamC; i++ )
      cout << " " << strC[i];

   system("PAUSE");
   return EXIT_SUCCESS;
}
```

```
#include <cstdlib>
#include <iostream>
#include <string>

using namespace std;

void funcion( string str ){
   for ( int i=0; i<str.size(); i++ )
      str[i] = '.';
}

int main(int argc, char *argv[])
{
   string str = "ORIGINAL";
   funcion(str);
   cout << str;

   system("PAUSE");
   return EXIT_SUCCESS;
}
```

**Atención:** las cadenas C++, al contrario que los arrays, son objetos, no punteros, de modo que se pasan por copia cuando son argumentos de funciones:

```
#include <cstdlib>
#include <iostream>
#include <string>

using namespace std;

void f( string str ){
   for ( int i=0; i<str.size(); i++ )
      str[i] = '.';
}

int main(int argc, char *argv[])
{
   string str = "ORIGINAL";
   f( str );
   cout << str;

   system("PAUSE");
   return EXIT_SUCCESS;
}
```




```
#include <cstdlib>
#include <iostream>
#include <string>

using namespace std;

void f( string &str ){
   for ( int i=0; i<str.size(); i++ )
      str[i] = '.';
}

int main(int argc, char *argv[])
{
   string str = "ORIGINAL";
   f( str );
   cout << str;

   system("PAUSE");
   return EXIT_SUCCESS;
}
```

## Cadenas con <string>

| | |
|---|---|
| String constructors | create strings from arrays of characters and other strings |
| String operators | concatenate strings, assign strings, use strings for I/O, compare strings |
| append | append characters and strings onto a string |
| assign | give a string values from strings of characters and other C++ strings |
| at | returns an element at a specific location |
| begin | returns an iterator to the beginning of the string |
| c_str | returns a standard C character array version of the string |
| capacity | returns the number of elements that the string can hold |
| clear | removes all elements from the string |
| compare | compares two strings |
| copy | copies characters from a string into an array |
| data | returns a pointer to the first character of a string |
| empty | true if the string has no elements |
| end | returns an iterator just past the last element of a string |
| erase | removes elements from a string |
| find | find characters in the string |
| find_first_not_of | find first absence of characters |
| find_first_of | find first occurrence of characters |
| find_last_not_of | find last absence of characters |
| find_last_of | find last occurrence of characters |
| getline | read data from an I/O stream into a string |
| insert | insert characters into a string |
| length | returns the length of the string |
| max_size | returns the maximum number of elements that the string can hold |
| push_back | add an element to the end of the string |
| rbegin | returns a reverse_iterator to the end of the string |
| rend | returns a reverse_iterator to the beginning of the string |
| replace | replace characters in the string |
| reserve | sets the minimum capacity of the string |
| resize | change the size of the string |
| rfind | find the last occurrence of a substring |
| size | returns the number of items in the string |
| substr | returns a certain substring |
| swap | swap the contents of this string with another |

- Assigning one string object's value to another string object
- string string_one = "Hello";
- string string_two;
  string_two = string_one;

  Assigning a C++ string literal to a string object
- string string_three;
  string_three = "Goodbye";

- Assigning a single character (char) to a string object
- string string_four;
- char ch = 'A';
- string_four = ch;
  string_four = 'Z';

- two string objects
- string str1 = "Hello ";
- string str2 = "there";
  string str3 = str1 + str2; // "Hello there"

- a string object and a character string literal
- string str1 = "Hello ";
  string str4 = str1 + "there";

- a string object and a single character
- string str5 = "The End";
  string str6 = str5 + '!';

The comparison operators return a **bool** (true/false) value indicating whether the specified relationship exists between the two operands. The operands may be:

- two string objects
- a string object and a character string literal

The extraction operator reads a character string from an input stream and assigns the value to a string object.

```
string str1;
cin >> str1;
```

**Excepciones en C++**

C++ implementa manejo de excepciones. Las excepciones nos sirven para gestionar errores de una forma homogénea. Previamente (en C) no existía un mecanismo definido para el tratamiento de errores. Cada programador o grupo de desarrollo decidía cómo realizar esta tarea.

La idea subyacente es que un procedimiento *lanzará* (o elevará) una excepción de modo que en el contexto superior se puede detectar una situación anormal.

Será responsabilidad del contexto superior el tomar las decisiones apropiadas en vista del error detectado. Será posible también ignorar las excepciones que pueda elevar una determinada función o método.

Las palabras reservadas de C++ en relación con las excepciones son:
**throw**: declara que una función o método lanza una excepción.
**try**: declara un bloque dentro del cual se puede capturar una excepción.
**catch**: declara un bloque de tratamiento de excepción.

**Un primer ejemplo: división por 0**

```
#include <cstdlib>
#include <iostream>
#include <exception>

using namespace std;

double divide( int dividendo, int divisor ){
   if ( divisor == 0 )
      throw exception();
   return (double)dividendo/divisor;
}

int main(int argc, char *argv[])
{
   cout << "Division correcta:" << divide(1,2) <<
endl;
   /* Este programa termina automáticamente */
   cout << "Division por cero:" << divide(1,0) <<
endl;

   system("PAUSE");
   return EXIT_SUCCESS;
}
```

```
#include <cstdlib>
#include <iostream>
#include <exception>

using namespace std;

double divide( int dividendo, int divisor ){
   if ( divisor == 0 )
      throw exception();
   return (double)dividendo/divisor;
}

int main(int argc, char *argv[])
{
   cout << "Division correcta:" << divide(1,2) << endl;
   /* Este programa captura y trata la excepción */
   try{
      cout << "Division por cero:" << divide(1,0) << endl;
   }catch( exception e ){
      cout << "Ocurrió una excepción: " << e.what();
   }
   system("PAUSE");
   return EXIT_SUCCESS;
}
```

Cuando un bloque try se ejecuta sin lanzar excepciones, el flujo de control pasa a la instrucción siguiente al último bloque catch asociado con ese bloque try.

**Excepciones en C++**

Podemos definir nuestras propias excepciones. Las excepciones se definen como clases.

```
#include <cstdlib>
#include <iostream>

using namespace std;

class ExcepcionDivCero{
public:
   ExcepcionDivCero(): mensaje( "Excepción:
división por cero" ) {}
   const char *what() const{ return mensaje; }
private:
   char *mensaje;
};

double divide( int dividendo, int divisor ){
   if ( divisor == 0 )
      throw ExcepcionDivCero();
   return (double)dividendo/divisor;
}

int main(int argc, char *argv[])
{
   cout << "Division correcta:" << divide(1,2) <<
endl;
   /* Este programa sale automáticamente... */
   cout << "Division por cero:" << divide(1,0) <<
endl;

   system("PAUSE");
   return EXIT_SUCCESS;
}
```

```
#include <cstdlib>
#include <iostream>

using namespace std;

class ExcepcionDivCero{
public:
   ExcepcionDivCero(): mensaje( "Excepción: división
por cero" ) {}
   const char *what() const{
      return mensaje;
   }
private:
   char *mensaje;
};

double divide( int dividendo, int divisor ){
   if ( divisor == 0 )
      throw ExcepcionDivCero();
   return (double)dividendo/divisor;
}

int main(int argc, char *argv[])
{
   cout << "Division correcta:" << divide(1,2) << endl;
   /* Este programa captura y trata la excepción */
   try{
      cout << "Division por cero:" << divide(1,0) << endl;
   }catch( ExcepcionDivCero e ){
      cout << "Ocurrió una excepción: " << e.what();
   }
   system("PAUSE");
   return EXIT_SUCCESS;
}
```

**Excepciones en C++**

**Las clases de excepciones pueden heredar de otras clases de excepción o no heredar de nadie. La declaración de la clase básica exception está en la cabecera <exception>.** Aunque no es necesario, se suelen hacer subclases de *exception* o de otras clases de excepción que hayamos definido. Esto se hace porque en los bloques match se capturan las excepciones de la clase que aparece o de cualquier subclase de ella. La herencia debe ser pública.

```
#include <cstdlib>
#include <iostream>
#include <exception>

using namespace std;

class ExcepcionDivCero: public exception{
public:
   ExcepcionDivCero(): mensaje( "Excepción: división por cero" ) {}
   const char *what() const throw(){
      return mensaje;
   }
private:
   char *mensaje;
};

double divide( int dividendo, int divisor ){
   if ( divisor == 0 )
      throw ExcepcionDivCero();
   return (double)dividendo/divisor;
}

int main(int argc, char *argv[])
{
   cout << "Division correcta:" << divide(1,2) << endl;
   /* Este programa captura y trata la excepción */
   try{
      cout << "Division por cero:" << divide(1,0) << endl;
   }catch( exception &e ){
      cout << "Ocurrió una excepción: " << e.what();
   }

   system("PAUSE");
   return EXIT_SUCCESS;
}
```

**Excepciones en C++**

Como es posible que nuestras excepciones no deriven de ninguna clase base, C++ permite una sentencia catch que capture *cualquier clase que se eleve*. La forma de hacer esto es con **catch ( …)**

**Un bloque try puede tener varios bloques catch asociados. Cada bloque catch es un ámbito diferente.**

```
#include <cstdlib>
#include <iostream>
#include <exception>

using namespace std;

class ExcepcionDivCero: public exception{
public:
   ExcepcionDivCero(): mensaje( "Excepción: división por cero" ) {}
   const char *what() const throw(){
      return mensaje;
   }
private:
   char *mensaje;
};

double divide( int dividendo, int divisor ){
   if ( divisor == 0 )
      throw ExcepcionDivCero();
   return (double)dividendo/divisor;
}

int main(int argc, char *argv[])
{
   cout << "Division correcta:" << divide(1,2) << endl;
   /* Este programa captura y trata la excepción */
   try{
      cout << "Division por cero:" << divide(1,0) << endl;
   }catch( ExcepcionDivCero e ){
      cout << "Ocurrió una excepción: " << e.what();
   }catch( exception &e ){
      cout << "Ocurrió una excepción: " << e.what();
   }catch( ... ){
      cout << "Ocurrió una excepción que no hereda de exception";
   }

   system("PAUSE");
   return EXIT_SUCCESS;
}
```

**Excepciones en C++**

Se pueden tener varios **bloques try anidados** de modo que, si no se encuentran manejadores de excepción en un bloque, se pasa a buscarlos al bloque inmediatamente superior. Si se sale de todos los bloques anidados sin encontrar un manejador, el programa terminará (por defecto).

```
#include <cstdlib>
#include <iostream>
#include <exception>

using namespace std;

class ExcepcionDivCero: public exception{
public:
   ExcepcionDivCero(): mensaje( "Excepción: división por cero" ) {}
   const char *what() const throw(){
      return mensaje;
   }
private:
   char *mensaje;
};

double divide( int dividendo, int divisor ){
   if ( divisor == 0 )
      throw ExcepcionDivCero();
   return (double)dividendo/divisor;
}

int main(int argc, char *argv[])
{
   cout << "Division correcta:" << divide(1,2) << endl;
   /* Este programa captura y trata la excepción */
   try{
         /* código susceptible de elevar excepciones */

         try{

               /* Este bloque try lanza una excepcion que
                no manejan sus correspondientes catch*/
              throw exception();

         }catch( ExcepcionDivCero &e ){
               cout << "Ocurrió una excepción: " << e.what();
         }

   }catch( exception e ){
      cout << "Ocurrió una excepción: " << e.what();
   }catch( ... ){
      cout << "Ocurrió una excepción que no hereda de exception";
   }

   system("PAUSE");
   return EXIT_SUCCESS;
}
```

**Excepciones en C++**

Tenga en cuenta que **el bloque try define un ámbito** y los objetos declarados en ese ámbito no están disponibles en el ámbito del bloque catch que, en su caso, se ejecute.

Es posible **relanzar la misma excepción** que se está tratando. Esto se hará en casos en que el tratamiento de la excepción no se haga completamente en un único manejador. Si utilizamos un bloque catch que da nombre a la excepción capturada, podemos volver a lanzarla con *throw <nombre>*; . En el caso de estar en un manejador *catch( … )*, se puede relanzar la excepción con *throw;*

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

class ExcepcionDivCero: public exception{
public:
   ExcepcionDivCero(): mensaje( "Excepción: división por cero" ) {}
   const char *what() const throw(){
      return mensaje;
   }
private:
   char *mensaje;
};

double divide( int dividendo, int divisor ){
   if ( divisor == 0 )
      throw ExcepcionDivCero();
   return (double)dividendo/divisor;
}

int main(int argc, char *argv[])
{
   char *array;
   /* Este programa captura y trata la excepción */
   try{
      /* bloque susceptible de elevar excepciones */
      try{
         /* reserva de memoria */
         array = new char[200];
         /* Este bloque try lanza una excepcion que
            no manejan sus correspondientes catch*/
         throw exception();
      }catch( ... ){
         /* Queremos liberar la memoria, aunque la excepción
            se tratará en otro bloque try, si procede... */
         delete [] array;
         throw;
      }
   }catch( exception e ){
      cout << "Ocurrió una excepción: " << e.what();
   }catch( ... ){
      cout << "Ocurrió una excepción que no hereda de exception";
   }
   system("PAUSE");
   return EXIT_SUCCESS;
}
```

**Excepciones en C++**

El comportamiento por defecto cuando no se encuentra un manejador de excepción es la terminación del programa. Este comportamiento se puede modificar con la función **set_terminate( void(*)() )**.

```
#include <cstdlib>
#include <iostream>
#include <exception>

using namespace std;

double divide( int dividendo, int divisor ){
   if ( divisor == 0 )
      throw exception();
   return (double)dividendo/divisor;
}

void terminar(){
   cout << "Final anormal del programa!!\n";
   system("PAUSE");
   exit(EXIT_SUCCESS);
}

int main(int argc, char *argv[])
{
   /* Este programa no captura la excepción */

   set_terminate( terminar );
   divide(1,0);

   system("PAUSE");
   return EXIT_SUCCESS;
}
```

**Excepciones en C++**

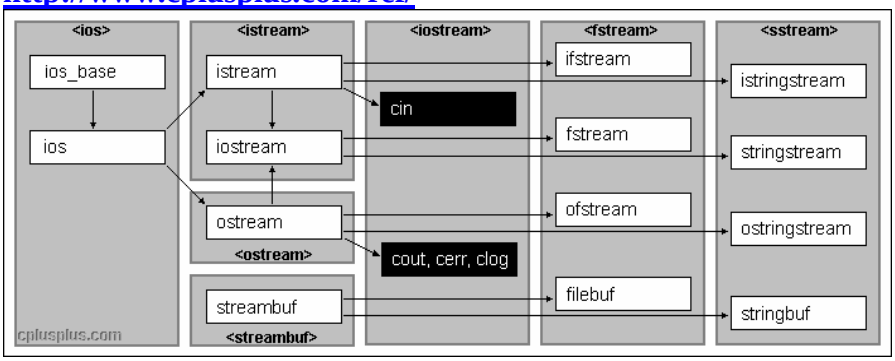
Se puede especificar una lista de excepciones que una función o método puede lanzar. Para ello, se escribe una lista ***throw( [lista clases excepcion] )*** después del nombre y lista de parámetros de la función en cuestión. A la hora de heredar, no es posible sobrescribir un método dándole menores restricciones en cuanto a lanzamiento de excepciones, esto es, un método que sobrescribe a otro puede lanzar, a lo sumo, las mismas excepciones que el método sobrescrito.

```
#include <cstdlib>
#include <iostream>
#include <exception>

using namespace std;

class ExcepcionDivCero: public exception{
public:
   ExcepcionDivCero(): mensaje( "Excepción: división por cero" ) {}
   const char *what() const throw(){
      return mensaje;
   }
private:
   char *mensaje;
};

double divide( int dividendo, int divisor )
      throw (exception,ExcepcionDivCero)
{
   if ( divisor == 0 )
      throw ExcepcionDivCero();
   return (double)dividendo/divisor;
}

int main(int argc, char *argv[])
{
   char *array;
   /* Este programa captura y trata la excepción */
   try{
      /* bloque susceptible de elevar excepciones */
      try{
         /* reserva de memoria */
         array = new char[200];
         /* Este bloque try lanza una excepcion que
            no manejan sus correspondientes catch*/
         throw exception();
      }catch( ... ){
         /* Queremos liberar la memoria, aunque la excepción
            se tratará en otro bloque try, si procede... */
         delete [] array;
         throw;
      }
   }catch( exception e ){
      cout << "Ocurrió una excepción: " << e.what();
   }

   system("PAUSE");
   return EXIT_SUCCESS;
}
```

Con *throw()* se declara que una función o método no lanza excepciones.

**Entrada y salida con iostream**

La biblioteca estándar de C++ nos provee de una útil batería de clases de entrada y salida utilizando flujos. Puede consultar la jerarquía de clases en Internet:
**http://www.cplusplus.com/ref/**




**En iostream tenemos las clases base para flujos de entrada y salida y los flujos predefinidos cout, cin, cerr, clog.**

**sstream** permite utilizar cadenas como flujos**.**

```
#include <cstdlib>
#include <iostream>
#include <sstream>
#include <string>

using namespace std;

int main(int argc, char *argv[])
{
   stringstream flujoCadena;
   char strC1[200], strC2[200];
   int dato;
   string strCPP;
   /* Salida a la cadena */
   flujoCadena << "  Primera línea 1\n  Segunda línea 2\n";
   /* Leer una línea (no ignora los blancos iniciales) */
   flujoCadena.getline(strC1,200);
   /* Con el operador >> se puede leer también a un char * */
   flujoCadena >> strC2;
   /* Leer la siguiente entrada. Descarta los blancos iniciales
      por defecto... */
   flujoCadena >> strCPP;
   /* Se pueden leer más tipos de datos */
   flujoCadena >> dato;

   /* El método .str() nos da un string el contenido del flujo */
   cout << "flujoCadena:\n" << flujoCadena.str() << "\n";
   cout << "strC1:\n" << strC1 << "\n";
   cout << "strC2:\n" << strC2 << "\n";
   cout << "strCPP:\n" << strCPP << "\n";
   cout << "dato:\n" << dato << "\n";

   system("PAUSE");
   return EXIT_SUCCESS;
}
```

Public member functions: (www.cplusplus.com)

| | |
|---|---|
| **stringstream members:** | |
| (constructor) | Construct an object and optionally initizalize string content. |
| rdbuf | Get the stringbuf object associated with the stream. |
| str | Get/set string value. |
| **members inherited from istream:** | |
| operator>> | Performs a formatted input operation (extraction) |
| gcount | Get number of characters extracted by last unformatted input operation |
| get | Extract unformatted data from stream |
| getline | Get a line from stream |
| ignore | Extract and discard characters |
| peek | Peek next character |
| read | Read a block of data |
| readsome | Read a block of data |
| putback | Put the last character back to stream |
| unget | Make last character got from stream available again |
| tellg | Get position of the get pointer |
| seekg | Set position of the get pointer |
| sync | Syncronize stream's buffer with source of characters |
| **members inherited from ostream:** | |
| operator<< | Perform a formatted output operation (insertion). |
| flush | Flush buffer. |
| put | Put a single character into output stream. |
| seekp | Set position of put pointer. |
| tellp | Get position of put pointer. |
| write | Write a sequence of characters. |
| **members inherited from ios:** | |
| operator void * | Convert stream to pointer. |
| operator ! | evaluate stream object. |
| bad | Check if an unrecoverable error has occurred. |
| clear | Set control states. |
| copyfmt | Copy formatting information. |
| eof | Check if End-Of-File has been reached. |
| exceptions | Get/set the exception mask. |
| fail | Check if failure has occurred. |
| fill | Get/set the fill character. |
| good | Check if stream is good for i/o operations. |
| imbue | Imbue locale. |
| narrow | Narrow character. |
| rdbuf | Get/set the associated streambuf object. |
| rdstate | Get control state. |
| setstate | Set control state. |
| tie | Get/set the tied stream. |
| widen | Widen character. |
| **members inherited from ios_base:** | |
| flags | Get/set format flags. |
| getloc | Get current locale. |
| imbue | Imbue locale. |
| iword | Get reference to a long element of the internal extensible array. |
| precision | Get/set floating-point decimal presision. |
| pword | Get reference to a void* element of the internal extensible array. |
| register_callback | Register event callback function. |
| setf | Set some format flags. |

**Entrada y salida con iostream**

**fstream** nos permite utilizar **archivos** como flujos.

Los **modos de apertura** son constantes de **máscara de bit**, de modo que se puede hacer un *or* lógico de ellos para conseguir un modo de apertura combinado.

**Ejemplo en modo texto:**

```
#include <cstdlib>
#include <iostream>
#include <fstream>
#include <string>

using namespace std;

int main(int argc, char *argv[])
{
   fstream archivo;

   /* Abrimos el archivo en modo salida y ponemos a 0
      , es decir, descartamos el contenido actual... */
   archivo.open( "Prueba.txt", fstream::out | fstream::trunc );
   if ( archivo.is_open() ){ // Comprobamos que se abrió correctamente
      archivo << "Hola " << 5 << endl;
      archivo.close();
   }

   /* Abrimos el archivo en modo salida y añadir
      , es decir, mantenemos el contenido actual y nos
      disponemos a añadir al final */
   archivo.open( "Prueba.txt", fstream::out | fstream::app );
   if ( archivo.is_open() ){
      archivo << "Adiós " << 4 << endl;
      archivo.close();
   }

   /* Abrimos el archivo en modo entrada */
   archivo.open( "Prueba.txt", fstream::in );

   if ( archivo.is_open() ){
      string lectura;
      /* Para controlar en fin de archivo correctamente, es necesario
         hacer una lectura antes de comprobar si se ha llegado al fin
         de archivo */
      archivo >> lectura;
      while ( !archivo.eof() ){
         cout << lectura;
         archivo >> lectura;
      }
      archivo.close();
   }

   system("PAUSE");
   return EXIT_SUCCESS;
}
```

**Entrada y salida con iostream (fstream, ejemplo en modo binario con estructuras )**

```
#include <cstdlib>
#include <iostream>
#include <fstream>
#include <string.h>

using namespace std;

struct registro{
   char nombre[21];
   char apellido[21];
   int edad;
};

int main(int argc, char *argv[])
{
   fstream archivo;
   registro r1 = { "José","Pérez",20 };

   /* Abrimos el archivo en modo salida y ponemos a 0
      , es decir, descartamos el contenido actual... */
   archivo.open( "Prueba.bin", fstream::out | fstream::trunc | fstream::binary );
   archivo.write( (char*)&r1, sizeof( registro ) );
   archivo.close();

   /* Otro registro */
   strcpy( r1.nombre, "Ana" );
   strcpy( r1.apellido, "Román" );
   r1.edad = 19;

   /* Abrimos el archivo en modo salida y añadir
      , es decir, mantenemos el contenido actual y nos
      disponemos a añadir al final */
   archivo.open( "Prueba.bin", fstream::out | fstream::app | fstream::binary );
   archivo.write( (char*)&r1, sizeof( registro ) );
   archivo.close();

   /* Abrimos el archivo en modo entrada */
   archivo.open( "Prueba.bin", fstream::in | fstream::binary );

   registro lectura;

   /* Para controlar en fin de archivo correctamente, es necesario
      hacer una lectura antes de comprobar si se ha llegado al fin
      de archivo */
   archivo.read( (char*)&lectura, sizeof( registro ) );
   while (!archivo.eof()){
      cout << lectura.nombre << endl
           << lectura.apellido << endl
           << lectura.edad << ende;
      archivo.read( (char*)&lectura, sizeof( registro ) );
   }

   archivo.close();

   system("PAUSE");
   return EXIT_SUCCESS;
}
```

Se ha omitido la comprobación de apertura correcta ( is_open() ) por razones de espacio.

**Entrada y salida con iostream (fstream, ejemplo en modo binario con clases)**

```
#include <cstdlib>
#include <iostream>
#include <fstream>
#include <string.h>

using namespace std;

class Registro{
private:
   char nombre[21];
   char apellido[21];
   int edad;
public:
   void pinta(){ cout << nombre << " " << apellido << " " << edad << endl; }
   void cambia( char *nom, char *ape, int eda ){
      strcpy( nombre, nom );
      strcpy( apellido, ape );
      edad = eda;
   }
   void almacena( fstream &archivo ){
      archivo.write( (char*)this, sizeof( Registro ) );
   }
   void recupera( fstream &archivo ){
      archivo.read( (char*)this, sizeof( Registro ) );
   }
};

int main(int argc, char *argv[])
{
   fstream archivo;
   Registro r1;
   r1.cambia("José","Pérez",20);

   archivo.open( "Prueba.bin", fstream::out | fstream::trunc | fstream::binary );
   r1.almacena( archivo );
   archivo.close();

   r1.cambia("Ana","Román",19);

   archivo.open( "Prueba.bin", fstream::out | fstream::app | fstream::binary );
   r1.almacena( archivo );
   archivo.close();

   archivo.open( "Prueba.bin", fstream::in | fstream::binary );

   r1.recupera( archivo );
   while (!archivo.eof()){
      r1.pinta();
      r1.recupera( archivo );
   }

   archivo.close();

   system("PAUSE");
   return EXIT_SUCCESS;
}
```

Se ha omitido la comprobación de apertura correcta ( is_open() ) por razones de espacio.

Pasamos el archivo como referencia

# Public member functions: (www.cplusplus.com)

| | |
|---|---|
| **fstream members:** | |
| (constructor) | Construct an object and optionally open a file. |
| rdbuf | Get the filebuf object associated with the stream. |
| is_open | Check if a file has been opened. |
| open | Open a file. |
| close | Close an open file. |
| **members inherited from istream:** | |
| operator>> | Performs a formatted input operation (extraction) |
| gcount | Get number of characters extracted by last unformatted input operation |
| get | Extract unformatted data from stream |
| getline | Get a line from stream |
| ignore | Extract and discard characters |
| peek | Peek next character |
| read | Read a block of data |
| readsome | Read a block of data |
| putback | Put the last character back to stream |
| unget | Make last character got from stream available again |
| tellg | Get position of the get pointer |
| seekg | Set position of the get pointer |
| sync | Syncronize stream's buffer with source of characters |
| **members inherited from ostream:** | |
| operator<< | Perform a formatted output operation (insertion). |
| flush | Flush buffer. |
| put | Put a single character into output stream. |
| seekp | Set position of put pointer. |
| tellp | Get position of put pointer. |
| write | Write a sequence of characters. |
| **members inherited from ios:** | |
| operator void * | Convert stream to pointer. |
| operator ! | evaluate stream object. |
| bad | Check if an unrecoverable error has occurred. |
| clear | Set control states. |
| copyfmt | Copy formatting information. |
| eof | Check if End-Of-File has been reached. |
| exceptions | Get/set the exception mask. |
| fail | Check if failure has occurred. |
| fill | Get/set the fill character. |
| good | Check if stream is good for i/o operations. |
| imbue | Imbue locale. |
| narrow | Narrow character. |
| rdbuf | Get/set the associated streambuf object. |
| rdstate | Get control state. |
| setstate | Set control state. |
| tie | Get/set the tied stream. |
| widen | Widen character. |
| **members inherited from ios_base:** | |
| flags | Get/set format flags. |
| getloc | Get current locale. |
| imbue | Imbue locale. |
| iword | Get reference to a long element of the internal extensible array. |
| precision | Get/set floating-point decimal presision. |

**...**

## Cadenas

```
int main() {
char cadena[128];
    ofstream fs("nombre.txt");
  fs << "Hola, mundo" << endl;
  fs.close();
  ifstream fe("nombre.txt");
  fe.getline(cadena, 128);
  cout << cadena << endl;
}
```

## Binarios

```
int main() {
   char Desde[] = "excepcion.cpp"; // Este fichero
   char Hacia[] = "excepcion.cpy";
   char buffer[1024];
   int leido;
       ifstream fe(Desde, ios::in | ios::binary);
       ofstream fs(Hacia, ios::out | ios::binary);

       do {
         fe.read(buffer, 1024);
         leido = fe.gcount();
         fs.write(buffer, leido);
         } while(leido);
       fe.close();
       fs.close();
       cout <<" fin de copia";

 cin.get();
 return 0;
}
```

**Funciones clave**

**open, close, read ,write , get, getline,  fail, good, exceptions, eof ,**

## Ejemplo2 Texto , Modos de apertura y lectura

```
#include <iostream>
#include <fstream>
#include <string>

using namespace std;

int main(int argc, char *argv[]) {
        fstream archivo;
// archivo en modo salida y descarta el contenido actual
        archivo.open( "Prueba.txt", fstream::out | fstream::trunc );
        if ( archivo.is_open() ){
        archivo << "Hola " << 5 << endl;
        archivo.close();
        }
/* archivo en modo salida y añadir */
        archivo.open( "Prueba.txt", fstream::out | fstream::app );
        if ( archivo.is_open() ){
        archivo << "Adiós " << 4 << endl;
        archivo.close();
}
/* archivo en modo entrada */
        archivo.open( "Prueba.txt", fstream::in );
        if ( archivo.is_open() ){
        string lectura;
/* Para controlar en fin de archivo correctamente, es necesario
hacer una lectura antes de comprobar si se ha llegado al fin
}
}
return EXIT_SUCCESS;
}
archivo >> lectura;
```

| bit | efecto |
|---|---|
| **ios_base::app** | Va al final del stream antes de cada operacion de escritura. |
| **ios_base::ate** | Va al finl del stream en la apertura |
| **ios_base::binary** | Abre el stream en modo binario |
| **ios_base::in** | Permite operaciones de lectura |
| **ios_base::out** | Permite operaciones de escritura |
| **ios_base::trunc** | (truncate) borra el contenido previo del fichero al abrir |

```
#include <cstdlib>
#include <iostream>
#include <fstream>
#include <string.h>

using namespace std;

class Provincias{
     private:
            char nombre[21];
            char capital[21];
            int censo;
     public:
     void pinta(){ cout << nombre << " " << capital << " " << censo << endl; }
      Provincias( char *nom, char *cap, int cen ){
         strcpy( nombre, nom );
         strcpy( capital, cap );
         censo = cen;
         }

};

int main(int argc, char *argv[])
{
  fstream archivo;
  Provincias r1("Huelva","Huelva",150000),r2("Mallorca","Palma de Mallorca",250000);
  archivo.open( "Prueba.bin", fstream::out | fstream::trunc | fstream::binary );
  archivo.write( (char*)&r1, sizeof( Provincias ) );
  archivo.close();
 //Añadir
  archivo.open( "Prueba.bin", fstream::out | fstream::app | fstream::binary );
  archivo.write( (char*)&r2, sizeof( Provincias) );
  archivo.close();
  archivo.open( "Prueba.bin", fstream::in | fstream::binary );
  archivo.read( (char*)&r1, sizeof( Provincias) );

  while (!archivo.eof()){
       r1.pinta();
        archivo.read( (char*)&r1, sizeof( Provincias ) );
  }
  archivo.close();
  system("PAUSE");
  return EXIT_SUCCESS;
}
```

Cuando se graba una clase, el espacio de memoria dinámica reservado por dicha clase no es continuo, pudiendo haber multiples referencias a otros objetos. Es necesario guardar la información de manera que sea posible recuperar y montar la estructura partiendo sólo de la informaciónj disponible en el fichero.

```
using namespace std;

class Cadena {
private:
            char* contenido;
public:

Cadena (const char* cadena){
              contenido=strdup(cadena);
}
bool almacenar(fstream &f){
    int tamanio;
    tamanio=strlen(contenido)+1;
    f.write((char *)(&tamanio),sizeof(int));
    f.write((char *)(contenido),tamanio);
    return(f.good());
}
bool recuperar(fstream &f){
    int tamanio;

    f.read((char *)(&tamanio),sizeof(int));
    delete contenido;
    contenido=new char [tamanio];
    f.read((char *)(contenido),tamanio);
    return(f.good());
}
operator char* (){
        return contenido;
        }
};
int main(int argc, char *argv[])
{
  fstream archivo;
 archivo.open( "Prueba.bin", fstream::out | fstream::trunc | fstream::binary );
   Cadena a("Hola"),b("");
   a.almacenar(archivo);
   cout <<a << endl;
   archivo.close();
   archivo.open( "Prueba.bin", fstream::in | fstream::binary );
   b.recuperar(archivo);
   archivo.close();
   cout <<b << endl;
  system("PAUSE");
  return EXIT_SUCCESS;
}
```

Si se quiere guardar una estructura aún más compleja se hace necesario establecer un orden .

```
class ListaCadenas{
     private:
           Cadena **lista;
           int num,cont;
     public:
     ListaCadenas(int _num){
               num = _num;
               cont =0;
               lista= new Cadena*[num];
               }
     void  add(Cadena &c){
               lista[cont++]=&c;
               }
    bool almacenar(fstream &f){
    int tamanio;
    tamanio=num;
    f.write((char *)(&tamanio),sizeof(int));
    for (int i=0;i<cont;i++)
       lista[i]->almacenar(f);
    return(f.good());
}

bool recuperar(fstream &f){
    int tamanio;

    f.read((char *)(&tamanio),sizeof(int));
    num = tamanio;
    cont = num;
    delete [] lista;
    lista = new Cadena*[num];
    for (int i=0;i<num;i++){
       Cadena *tmp= new Cadena("");
       tmp->recuperar(f);
       lista[i] =tmp;
    }
     return(f.good());
}
operator char*(){
   char rec[800];
      strcpy( rec,"");
       for (int i =0;i<num;i++)
          sprintf(rec,"%s%s\n",rec,(char *)(*lista[i]));
   return strdup(rec);
}};
```

```
int main(int argc, char *argv[])
{
  fstream archivo;
 archivo.open( "Prueba.bin", fstream::out | fstream::trunc |
fstream::binary );
   Cadena a("Hola"),b("adios");
   cout <<a << endl << b << endl ;
   ListaCadenas l(2),l1(2);
   l.add(a);l.add(b);
   l.almacenar(archivo);
   archivo.close();

   archivo.open( "Prueba.bin", fstream::in | fstream::binary
);
   l1.recuperar(archivo);
   archivo.close();

   cout<<"  Contenido de l1\n" << l1 ;
  system("PAUSE");
  return EXIT_SUCCESS;
}
```

Versión con herencia: hay que saber el tipo de lo guardado

```
class Cadena {
private:
        char* contenido;
protected:
        int tipo;
public:
Cadena (const char* cadena){
          contenido=strdup(cadena);
          tipo=1;
}
virtual bool almacenar(fstream &f){
    int tamanio;
    tamanio=strlen(contenido)+1;
    f.write((char *)(&tipo),sizeof(int));
    f.write((char *)(&tamanio),sizeof(int));
    f.write((char *)(contenido),tamanio);
    return(f.good());
}
…

class Cadenita:public Cadena{
     private:
          int valor;
     public:
      Cadenita(const char* c):Cadena(c){tipo=2;}
       bool almacenar( fstream &f ){
            f.write( (char*)this, sizeof( Cadenita ) );
            Cadena::almacenar(f);
            return(f.good());
       }
       bool recuperar( fstream &f ){
           f.read( (char*)this, sizeof( Cadenita) );
           Cadena::recuperar(f);
           return(f.good());
       }
};
```

```
class ListaCadenas{
     private:
          Cadena **lista;
          int num,cont;
     public:
…

bool recuperar(fstream &f){
    int tamanio;

    f.read((char *)(&tamanio),sizeof(int));
    num = tamanio;
    cont = num;
    delete [] lista;
    lista = new Cadena*[num];
    int tipo=0;
    for (int i=0;i<num;i++){
       f.read((char *)(&tipo),sizeof(int));
       if (tipo ==1 )lista[i]= new Cadena("");
       else    lista[i]= new Cadenita("");
       lista[i]->recuperar(f);
  }
     return(f.good());
}
…
};

int main(int argc, char *argv[]){
  fstream archivo;
 archivo.open( "Prueba.bin", fstream::out |
fstream::trunc | fstream::binary );
   Cadena a("Hola"),b("adios");
   Cadenita c("bye");
   cout <<a << endl << b << endl ;
   ListaCadenas l(3),l1(3);
   l.add(a);l.add(b);l.add(c);
   l.almacenar(archivo);
   archivo.close();

   archivo.open( "Prueba.bin", fstream::in |
fstream::binary );
   l1.recuperar(archivo);
   archivo.close();

   cout<<"  Contenido de l1\n" << l1 ;
  system("PAUSE");
  return EXIT_SUCCESS;
}
```

## Excepciones en los ficheros

### ios:: exceptions

La mascara de excepciones esta compuesta por flags (bits) que representan si se emitirá una excepción en el caso de llegar a uno de dichos estados:

- **badbit** (critical error in stream buffer)
- **eofbit** (End-Of-File reached while extracting)
- **failbit** (failure extracting from stream)
- **goodbit** (no error condition, represents the absence of the above bits)

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

int main () {
  ifstream file;
  file.exceptions ( ifstream::eofbit |
ifstream::failbit | ifstream::badbit );
  try {
    file.open ("test.txt");
    while (!file.eof()) file.get();
  }
  catch (ifstream::failure e) {
    cout << "Exception opening/reading file";
  }

  file.close();

  return 0;
}
```

### bool operator ! ( ) (similar a good y fail)

si los flags de excepción están puestos devuelve el estado del stream

```
int main () {
  ifstream is;
  is.open ("test.txt");
  if (!is)
    cerr << "Error abriendo 'test.txt'\n";
  return 0;
}
```

## Genéricos

C++ incluye la posibilidad de definición de funciones y de clases parametrizadas o genéricas. La forma de definirlos es a través de plantillas (templates) y utilizando variables de clase, es decir, variables que pueden tomar como valor un tipo o clase.

## Función genérica

| Punto |
| --- |
| + int x<br>+ int y |
| + Punto( int x, int y )<br>+ operador char*() |

```
#include <cstdlib>
#include <iostream>
#include <stdio.h>
#include <string.h>

using namespace std;

template< class T >
void pinta( T impresion ){
   cout << impresion <<"\n";
}

class Punto{
public:
   int x;
   int y;
   Punto( int nx, int ny ){ x=nx; y=ny; }
   operator char*(){
      char tmp[30];
      sprintf(tmp,"(%i,%i)",x,y);
      return strdup(tmp);
   }
};

int main(int argc, char *argv[])
{
   pinta<int>( 5 );
   pinta<float>( 5.4 );
   pinta<char>('B');
   pinta<Punto>( *(new Punto(2,3)) );

   system("PAUSE");
   return EXIT_SUCCESS;
}
```

**Genéricos**

**Clase genérica:**

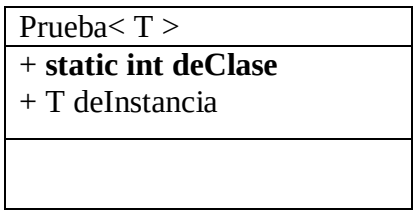
| Punto< T > |
|---|
| + T x<br>+ T y |
| + Punto( T x, T y )<br>+ void pinta() |

| Complejo |
|---|
| + double real<br>+ double imag |
| + Complejo<br>( double r=0, double i=0 )<br>+ operator char* () |

```
#include <cstdlib>
#include <iostream>
#include <stdio.h>
#include <string.h>

using namespace std;

template <class T>
class Punto{
private:
   T x;
   T y;
public:
   Punto( T nx, T ny );
   void pinta();
};

template <class T>
Punto<T>::Punto( T nx, T ny ){
      x=nx;
      y=ny;
   }

template <class T>
void Punto<T>::pinta(){
      cout << "(" << x << "," << y << ")\n";
   }

class Complejo{
public:
   double real;
   double imag;
   Complejo( double r=0, double i=0 ){ real=r; imag=i; }
   operator char*(){
      char tmp[30];
      sprintf(tmp,"[%.2lf,%.2lf]",real,imag);
      return strdup(tmp);
   }
};

int main(int argc, char *argv[])
{
   Punto<int> pi(10,5);
   Punto<char> pc('a','b');
   Punto<double> pd(10.2345,-0.777);
   Punto<Complejo> pcom( *(new Complejo(1,0.45)),
                         *(new Complejo(3.23,45)) );

   pi.pinta(); pc.pinta(); pd.pinta(); pcom.pinta();

   system("PAUSE");
   return EXIT_SUCCESS;
}
```

**Genéricos**

**Clase genérica: variables y métodos de clase (static)**




```
#include <cstdlib>
#include <iostream>

using namespace std;

template< class T >
class Prueba{
public:
   static int deClase;
   T deInstancia;
};
template< class T >
int Prueba<T>::deClase=0;

int main(int argc, char *argv[])
{
   Prueba<int> p1;
   Prueba<int> p2;
   Prueba<float> p3;

   p1.deClase=1;
   p2.deClase=2;
   p3.deClase=3;

   cout << p1.deClase << ","
        << p2.deClase << ","
        << p3.deClase << "\n";

   system("PAUSE");
   return EXIT_SUCCESS;
}
```

**Nota:** Existen casos diferentes cuando se utilizan funciones friend con genéricos que no veremos en este curso.

## Anexo : **Funciones extendidas con punteros**

- **malloc/free** frente a **new/delete**

**Utilización de un bloque de memoria de diferentes maneras:**

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

int main(){

   int enteros[10];
   char *caracteres = (char*)enteros;

   for (int i=0; i<(10*sizeof(int))/sizeof(char); i++){
      caracteres[i]='A'+i;
      cout << caracteres[i];
   }

   for (int i=0; i<10; i++){
            cout << enteros[i] <<" ";
   }

   system("PAUSE");
   exit(EXIT_SUCCESS);
}
```

**Repaso de punteros, arrays y punteros a funciones**

**Punteros a funciones:**

Se declaran:
**<tipo devuelto>(*<nombre>)( [Lista de parámetros] );**

**Ejemplo:** el puntero a función f que se corresponde con una función que devuelve un real y que toma como argumentos un entero y un puntero a char:

```
float (*pf)( int entero, char *puntero );
// no es necesario dar nombre a los parámetros:
float (*pf)( int, char* );
```

El puntero se llama **pf.**

Al igual que con el resto de los punteros, se les hace apuntar a alguna función. C/C++ toma el nombre de la función como la dirección de memoria donde está almacenada. No es necesario utilizar &. Al igual que con los arrays estáticos, & utilizado sobre un nombre de función devuelve la misma dirección.

```
pf = funcion;
```

Podemos utilizar el puntero para invocar a la función a la que apunta.

```
pf( 10, “Hola” );
```

Array de punteros a función:

```
int (*arr[10])( const int, const int );
```

para llamarlas:

```
arr[3]( 4, 67 );
```

**Punteros a funciones: ejemplo con qsort;**

```
#include <cstdlib>
#include <iostream>
#include <sstream>
#include <string.h>

using namespace std;

class Punto{
public:
   int x;
   int y;
   operator char*(){
      stringstream cad;
      cad << "(" << x << "," << y << ")";
      return strdup( (cad.str()).c_str());
   }
};

int menor_a_mayor( const void *e1, const void *e2 ){
   Punto *p1=(Punto*)e1, *p2=(Punto*)e2;
   if ( p1->x == p2->x )
      return ( p1->y - p2->y );
   else
      return ( p1->x - p2->x );
}

int mayor_a_menor( const void *e1, const void *e2 ){
   return - menor_a_mayor( e1, e2 );
}

void muestra_array( Punto array[], int tam ){
   for (int i=0; i<tam; i++){
      cout << array[i] << " , ";
      if (i%7 == 6) cout << endl;
   }
}

int main(int argc, char *argv[])
{
   Punto array[30];
   for ( int i=0; i<30; i++ ){
      array[i].x=rand()%100;
      array[i].y=rand()%100;
   }
   muestra_array( array, 30 );
   cout << "\nDe menor a mayor:\n";
   qsort( array, 30, sizeof(Punto), menor_a_mayor );
   muestra_array( array, 30 );

   cout << "\nDe mayor a menor:\n";
   qsort( array, 30, sizeof(Punto), mayor_a_menor );
   muestra_array( array, 30 );

   system("PAUSE");
   return EXIT_SUCCESS;
}
```

El prototipo de qsort es:

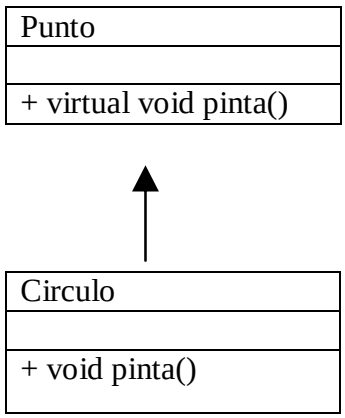
```
void  qsort ( void * base, size_t num,
              size_t width,
             int (*fncompare)(const void *, const void *) );
```

El primer argumento es un puntero al inicio del bloque de memoria que queremos ordenar.
El segundo argumento es el número de elementos que hay en el bloque de memoria.
El tercer argumento es el tamaño en bytes de un elemento de los que queremos ordenar.

El cuarto argumento es un puntero a función. Cambiando este puntero podemos cambiar el comportamiento de la función qsort. Este es un ejemplo de función polimórfica en tiempo de ejecución previa al paradigma de programación orientada a objetos.

**Ejercicio propuesto:**

Programe el comportamiento dinámico de la jerarquía de clases que se muestra sin utilizar objetos (utilice estructuras en su lugar)




El programa de ejemplo puede crear un array de referencias a puntos, llenarlo con referencias a puntos y a círculos y luego recorrer el array utilizando la función pinta correspondiente en cada caso. Sólo se debe utilizar la referencia para realizar la llamada correcta y se debe poder extender con nuevas formas de pintar sin necesidad de cambiar el código del programa que usa esta funcionalidad.