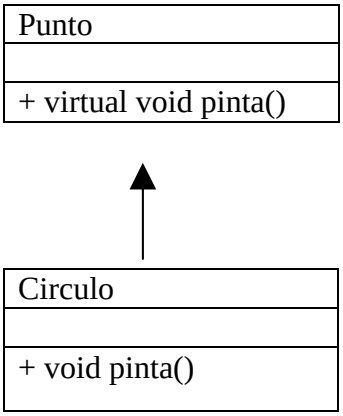
Información de tipo (clase) en tiempo de ejecución: <typeinfo>
typeid y dynamic_cast

static_cast<>
int a;
float b;
b = static_cast<float>(a); **ANSI-ISO incluido siempre**
const_cast<> **ANSI-ISO incluido siempre**
reinterpret_cast<> **ANSI-ISO incluido siempre**
dynamic_cast<> **en <typeinfo>**
typeid **en <typeinfo>**



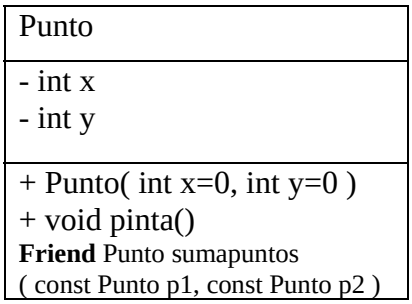

```
#include <cstdlib>
#include <iostream>

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 compraciones 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();
      }else
         cout << "Molde NO valido\n";
   }

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

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.




```
#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 - int y
+ Punto(int x=0, int y=0) + void pinta() **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 (Amiga2 &a2, int val) + void pinta() **Friend** class Amiga2

Amiga2
- int privada
+ void modificaAmiga1 (Amiga1 &a1, int val) + void pinta() **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.

No se pueden definir nuevos operadores ni cambiar la aridad.
Utilización de funciones friend en algunos casos
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)…

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

```
#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

Operador de conversión de tipo

Punto
- static int numserie - int *identificador + int x + int y
+ Punto(int x, int y) + ~Punto() + int operator==(Punto p) + Punto operator=(Punto p) + operator char*()

```
#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;
}
```

Operador ++ y –

Tienen dos tratamientos diferentes: preincremento/decremento, posincremento/decremento

Ciclo
- int cuenta - int maximo
+ Ciclo(int maximo) + operator char*() Friend: int operator++ (Ciclo &c) Friend: int operator++ (Ciclo &c, int inc)

El prefijo es como un operador unario normal. El posfijo supone que el segundo argumento es un 0 (entero)

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

using namespace std;

class Ciclo{
/* Prefijo */
friend int operator++( Ciclo &c ){
   c.cuenta = (++c.cuenta)%c.maximo;
   return c.cuenta;
}
/* Posfijo */
friend int operator++( Ciclo &c, int ){
   int tmp = c.cuenta;
   c.cuenta = (++c.cuenta)%c.maximo;
   return tmp;
}
private:
   int cuenta;
   int maximo;
public:
   Ciclo( int max ){
      cuenta = 0;
      maximo = max;
   }
   operator char*(){
      char salida[30];
      sprintf( salida, "%i",cuenta );
      return strdup( salida ); // Ojo! Aquí hay una fuga de memoria…
   }
};

int main(int argc, char *argv[])
{
   Ciclo c(10);

   for (int i=0; i<20 ;i++){
      cout << ++c <<",";
      cout << c++ << "\n";
   }

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

Operador ++ y –

Tienen dos tratamientos diferentes: preincremento/decremento, posincremento/decremento

Ciclo
- int cuenta - int maximo
+ Ciclo(int maximo) + operator char*() + int operator++() + int operator++(int inc)

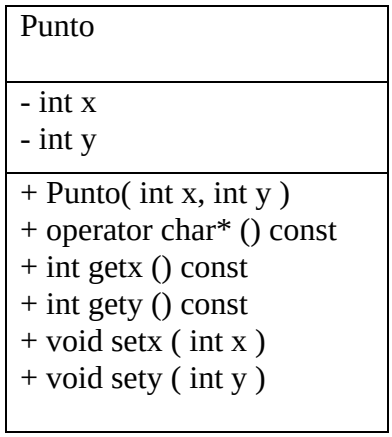
El prefijo es como un operador unario normal. El posfijo supone que el segundo argumento es un 0 (entero)

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

using namespace std;

class Ciclo{
private:
   int cuenta;
   int maximo;
public:
   Ciclo( int max ){
      cuenta = 0;
      maximo = max;
   }
   operator char*(){
      char salida[30];
      sprintf( salida, "%i",cuenta );
      return strdup( salida );
   }
   /* Prefijo */
   int operator++(){
      cuenta = (++cuenta)%maximo;
      return cuenta;
   }
   /* Posfijo */
   int operator++( int incremento ){
      int tmp = cuenta;
      cuenta = (++cuenta)%maximo;
      return tmp;
   }
};

int main(int argc, char *argv[])
{
   Ciclo c(10);

   for (int i=0; i<20 ;i++){
      cout << ++c <<",";
      cout << c++ << "\n";
   }

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

const

Se pueden crear objetos que deben ser constantes. El acceso a los métodos de estos objetos debe estar restringido de modo que sólo se llame a métodos que no modifiquen el estado del objeto. Se utiliza la palabra reservada **const** detrás del nombre y la lista de argumentos del método en cuestión para permitir la invocación de esos métodos por parte de objetos constantes. 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.




```
#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

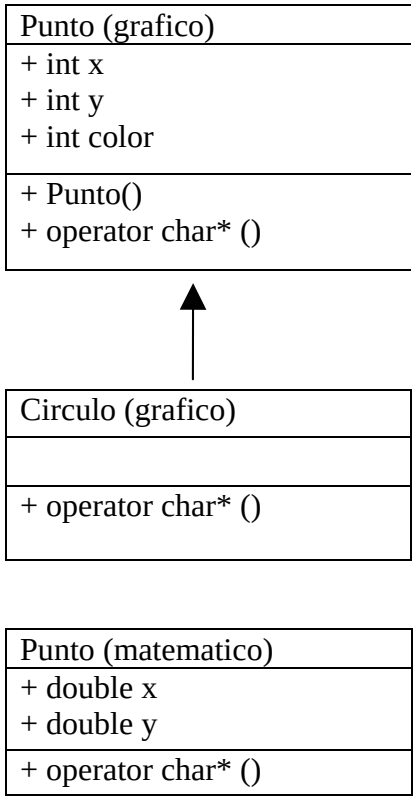
Punto
- int x - int y
+ Punto(int x, int y) + operator char* () const + int getx () const + int gety () const + void setx (int x) + void sety (int y) + void setx (int x) const + void sety (int y) const

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;
}
```

Namespaces

Los espacios de nombres nos permiten encapsular declaraciones (funciones, declaraciones de clases, etc.) relacionadas bajo un mismo ámbito.




```
#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;
}
```

Clases proxy

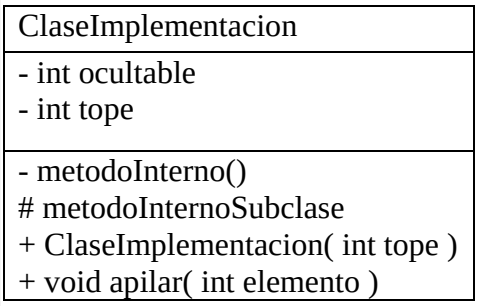
Existe un problema con las inclusiones de los *.hpp*: las declaraciones dejan ver gran parte de la implementación. Todo lo que no es público se podría ocultar al programador externo. Esto es consistente con el principio de ocultación.

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;
}
```



ClaseProxy.hpp

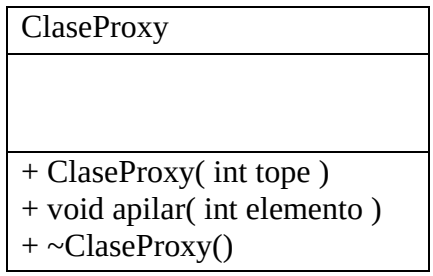
```
#ifndef CLASEPROXY_HPP
#define CLASEPROXY_HPP

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*

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 + int y
+ Punto(int x, int y) + 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 + T y
+ Punto(T x, T y) + void pinta()

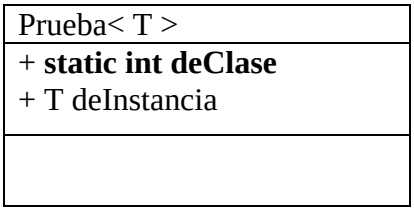
Complejo
+ double real + double imag
+ Complejo (double r=0, double i=0) + operator char* ()

Existen casos diferentes cuando se utilizan funciones friend con genéricos. Los estáticos son por clase concreta distinta…

```
#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.

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();
           delete [] array;
      }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();
         delete [] array;
      }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.

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 ejemplo:

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 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;
}
```

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;
}
```

Paso por referencia

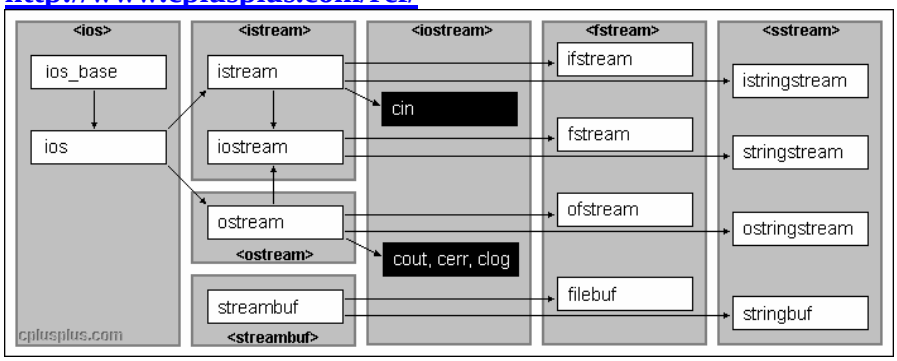
```
#include <cstdlib>
#include <iostream>
#include <string>

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

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.

Salida y entrada formateada: fotocopias.

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.
sync_with_stdio	Activates / Deactivates synchronization with cstdio functions. [static]
unsetf	Clear format flag.
width	Get/set field width.
xalloc	Return a new index for the internal extensible array. [static]

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.
pword	Get reference to a void* element of the internal extensible array.
register_callback	Register event callback function.
setf	Set some format flags.
sync_with_stdio	Activates / Deactivates synchronization with cstdio functions. [static]
unsetf	Clear format flag.
width	Get/set field width.
xalloc	Return a new index for the internal extensible array. [static]

Repaso de punteros, arrays y punteros a funciones

- **La dirección en que se almacena un puntero.**
- **Un puntero es una variable y, como tal, tiene asignada una memoria. Es un tipo de dato que guarda una dirección de memoria.**

Puntero y array:

int *i puede ser un puntero o un array

```
i = new int(5); /* entero */
i = new int[10]; /* un array */
```

para ver el contenido: *
para obtener la dirección en que se almacena una variable: &

Nota: ¿ Qué pasa si se hace & de un array estático ?

```
int estatico[20];
cout << estatico << &estatico;
```

***i = 5 (en el primer caso)**
***i en el segundo caso, el contenido de la primera posición.**

Se puede utilizar el indexado con la suma:

***(i+2) contenido de la tercera posición en el caso del array**
i+2 es equivalente a &i[2] y
***(i+2) es equivalente a i[2].**

Puntero a puntero, array de punteros o array de arrays…

```
int entero = 5;
int *simple = &entero;
int **doble = &simple;

cout << *doble;
cout << **doble;

int *array_punteros[30]; // o
int **array_punteros = new int*[30];

array_punteros[3] = simple;

int **array_de_arrays = new int*[10];
for ( int i=0; i<10; i++ )
        array_de_arrays[i] = new int[5];

array_de_arrays[3][4]=10;
```

Repaso de punteros, arrays y punteros a funciones

- **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);
}
```

Ejemplo: eficiencia en una pantalla

Queremos hacer operaciones gráficas en pantallas. El tamaño de pantalla lo podemos definir al crearla.

Podemos reservar la memoria como un array. Los accesos en arrays son relativamente rápidos porque se basan directamente en instrucciones máquina del microprocesador.

Por simplicidad, consideramos que la información de un punto se almacena en un byte (256 colores).

char *pantalla = new char[ancho*alto];

Para acceder al punto situado en las coordenadas (x,y), podemos hacer:

pantalla[y*ancho + x]

Esta operación es relativamente costosa, ya que para cada acceso a la pantalla estamos realizando tres accesos a memoria, una multiplicación y una suma.

Podemos hacer lo siguiente:

```
char **lineas = new char*[ alto ];  // array de arrays

for ( int indice=0, int inicio=0 ; i < alto ; i++, inicio+=ancho )
        lineas[ indice ] = pantalla + inicio;

/* A partir de este punto tenemos disponible un array de arrays de modo que cada posición es un array con el contenido de la línea que se indexa */
```

Ahora, para acceder al punto (x,y) podemos hacer:

```
lineas[y][x]
```

Con lo que hemos reducido la operación anterior a dos accesos a memoria.

Y podemos realizar operaciones sobre las líneas sin tener que recalcular el inicio de cada una cada vez.

Lo cierto es que esta optimización es poco notable en los procesadores actuales, pero sigue habiendo una razón de legibilidad y comodidad en el código para utilizar esta técnica.

Ejercicio propuesto: programe la clase Pantalla que utilice la técnica anterior. Programe una pantalla de caracteres de modo que se pueda mostrar en pantalla con cout. Sobrecargue el operador de casting (char*) y pruebe a sobrecargar el operador [] para que los puntos se puedan acceder con p[linea][columna].

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 );
```

Repaso de punteros, arrays y punteros a funciones

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:

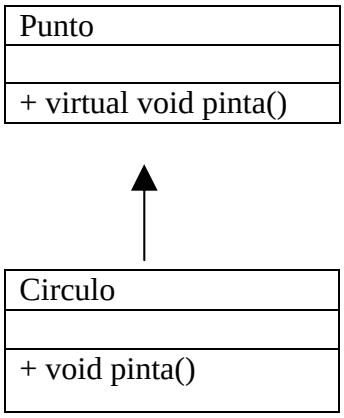
```
void  qsort ( void * base, size_t num,
              size_t width,
             int (*fncompare)(const void *, const void *) );
```

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.

Minimanual para compilar C++ desde la línea de comandos de Linux:

- **man gcc**

Compilar desde la línea de comandos

gcc [-o<archivo ejecutable salida>] [<opciones>] <archivo fuente>

Opción sólo compilar (obtener un .o) : -c

Hacer una librería

ar –q <archivo .a destino> [lista de archivos .o]

Enlazar varios archivos

Basta con poner los .o en la línea de comandos junto con el fuente que se quiere compilar.

gcc fuente.cpp modulo.o modulo2.o librería.a

Enlazar con librerías

En ocasiones es necesario comunicar al compilador las rutas donde están las cabeceras y/o las librerías:

-I<directorio de include>
-L<directorio de librerías>
-l<nombre de librería>

L opción l tiene la peculiaridad de que espera que el archivo se llame:
lib<nombre>.a

de modo que si se escribe
gcc fuente.cpp –lm
se busca la librería libm.a (que, en este caso, es la librería matemática estándar)

gcc fuente.cpp –I./libPila/ –L./libPila/ –lpila

Busca cabeceras en ./libPila/, busca librerías en ./libPila/ y busca la librería libPila.a

Makefiles

- **man make**

  http://www.gnu.org/software/make/manual/make.html
  Un archivo makefile contiene básicamente líneas

  objetivo: dependencias
  comandos

  Ejemplo:

  programa.exe: programa.cpp
  rm programa.o
  gcc –oprograma.exe programa.cpp –lm

  De modo que programa.exe depende de programa.cpp, esto es, si programa.cpp se modificó después que programa.exe, se realizan las operaciones siguientes. Se suelen hacer dependencias más elaboradas. Se ponen varias dependencias en un mismo proyecto de modo que si se ha modificado alguna de las librerías, se recompile. Es posible también poner objetivos que no se corresponden con un archivo e invocarlos explícitamente.

Se queda en el tintero

Acceso directo a memoria
El preprocesador
Biblioteca estándar de C
Biblioteca estándar de plantillas de C++ (STL: Standard template library)
Operaciones a nivel de bits en C

Cosas que se deberían saber:

Cadenas de caracteres en C
Diferencias de sintaxis entre C y C++
Nociones básicas de utilización de la memoria por parte de C/C++

Cosas interesantes:

Utilización de la memoria por parte de C
Modo real y modo protegido: segmentos, desplazamientos y modos de memoria
El proceso de enlazado y formatos de archivos:.a, .o, .exe, ejecutables …
Bibliotecas de enlace dinámico
Utilización de C/C++ en conjunción con otros lenguajes
Programación de manejadores de interrupciones
…

Grandes desventajas de C/C++ frente a Java: Portabilidad, escasez de estándares.