Metodología de la Programación

Grado en Informatica

ÍNDICE

1. Programación Estructurada

- Un programa está compuesto de 1 a n ***funciones*.**
- ***Función (subprograma):*** conjunto de acciones agrupadas bajo un nombre común.
- Las funciones:
  - Se escriben ***una sola vez*** y puede ser usadas ***varias veces en distintas partes*** del programa**.**
  - Hacen que un programa sea ***más corto*, *fácil de entender y corregir* y *disminuye los errores***.

- Ventajas de uso de funciones:
  - **Modularización**.
  - **Ahorro** de memoria **y tiempo** de desarrollo**.**
  - **Independencia** de los datos y **ocultamiento** de la información.

Ejemplo:

```
// declaracion o prototipo
double potencia(int base, int exponente);

...

// definicion
double potencia(int base, int exponente) {
 double resultado;
 resultado = ... ;
 return resultado;
}

...
valor = potencia(2,3); // llamada
valor = potencia(7,2); // llamada
```

Una función es conveniente ***declararla***, para que el compilador sepa que existe y pueda chequear el nº y tipo de los argumentos y del valor de retorno.

En la ***definición*** se indica lo que hace la función, que tipo de valor retorna y cuántos y de que tipo son sus argumentos.

La ***llamada*** se realiza incluyendo su nombre, y sus argumentos (datos pasados a la función).

2. Ámbitos globales, locales y de clase

Variable local

- ***Es declarada dentro de un bloque o función***.
- Sólo puede ser usada dentro del bloque en el que está. No puede usarse fuera.
- Sólo existe durante la ejecución del bloque en el que está: se crea al entrar en el bloque, se destruye al salir.

Variable global

- ***Es declarada fuera de cualquier bloque o función***, al inicio del programa
- Puede ser usada en cualquier parte del programa, y en cualquier función (no hay que pasarla como parámetro).
- Existe durante toda la ejecución del programa.

Una variable local y global pueden tener el mismo nombre (la variable local oculta a la global)

Variable miembro de clase

Variable miembro privada

- ***Es declarada en la parte private: de la clase***.
- No es visible fuera de la clase.
- Sólo la puede usar las funciones miembros (métodos) de la clase.

Variable miembro pública

- ***Es declarada en la parte public: de la clase***.
- Es visible fuera de la clase.
- La puede usar los métodos de la clase y los objetos de dicha clase que haya en el programa.

Una variable miembro puede tener el mismo nombre que una variable local y una global (la variable local oculta a la de clase y a la global, la de clase oculta a la global)

Mediante el **operador de resolución de ámbito ::** podemos acceder a la que nos interese
::vble accede a la global
clase::vble accede a la de clase

Ejemplo:

```
#include <iostream>
using namespace std;
void ver();

class clase {
 int a;
public:
 int aget() { return a; }
 void aset(int i) { a = i; }
 void ver();
};

int a; clase x; // global

void clase::ver(){
 a=2; //de clase
 int a=0; //local
 a++;  //local
 clase::a++; //de clase
 ::a=3; //global
 cout << "A " << ::a;
 cout << " X.a " << clase::a;
 cout << " a " << a << endl;
}
```

```
void ver(){
 a=0; //global
 x.aset(4); //global
 int a=5; //local
 clase x; //local
 a++; x.aset(2); //local
 ::x.aset(::x.aget()+1); ::a++; //global
 cout << "A " << ::a << " X.a " << ::x.aget();
 cout << " a " << a << " x.a " << x.aget() << endl;
}

int main(){
 a=2; x.aset(0); //global
 cout << "A " << a << " X.a " << x.aget() << endl;
 ver();
 cout << "A " << a << " X.a " << x.aget() << endl;
 x.ver();
 system("Pause"); return 0;
}
```

Salida del Programa:

```
A 2  X.a 0
A 1  X.a 5  a 6  x.a 2
A 1  X.a 5
A 3  X.a 3  a 1
```

3. Funciones y Métodos: Interfaces

3.1 Funciones genéricas

```
void funcion(int x, int y, char c) // bien
void funcion(int x, y, char c)     // mal
```

- Sintaxis:

```
tipo nombre([tipo v1, ..., tipo vN]) {
    [declaraciones;]
    sentencias;
    [return valor_devuelto;]
}
```

```
if (cubo(x) > cuadrado(y))
   x = max(y, z) - raiz(9) + 7;
```

- Las funciones:
  - Pueden ser usadas en una expresión o asignación (excepto void).
  - Existen durante todo el programa (tiempo de vida global).
  - Pueden ser utilizadas en todo el programa (alcance global).
  - Pueden ser recursivas y/o pueden llamar a otra función.

3.2 Funciones miembros de una clase: métodos

- Sintaxis:

```
tipo nombre_clase::nombre_funcion([tipo v1, ..., tipo vN]) {
    [declaraciones;]
    sentencias;
    [return valor_devuelto;]
}
```

```
class complejo {
 int real,imag;
public:
 void ini(int r,int i);
 void mostrar();
};
```

- Las funciones miembros o métodos:
  - Tienen acceso directo (no hay que pasarlo como parámetro) a los **miembros privados** de su clase y a los **miembros privados de los objetos de su clase pasados como argumentos**.
  - Pueden ser **públicas** o **privadas**:
    - Las funciones **privadas** sólo son accesibles por los miembros de la clase
    - Las funciones **públicas** son accesibles por los miembros de la clase y por los objetos de dicha clase definidos en el programa.

3.2.1 Funciones miembros constantes (métodos const)

- Indican al compilador que **el método no puede modificar el objeto que lo invoca**
- Sintaxis:

```
tipo nombre_clase::nombre_funcion([tipo v1, …, tipo vN]) const {
    [declaraciones;]
    sentencias; //si alguna modifica algún atributo -> error
    [return valor_devuelto;]
}
```

- Un **método const** no puede modificar los atributos de la clase y **sólo puede llamar a otros métodos const de la clase** (aunque si puede llamar a **métodos no const** de otras clases)
- **Los objetos constantes (objetos const) sólo pueden invocar los métodos const de su clase** (lógico, ya que al ser objetos constantes el compilador impide que puedan ser modificados y por tanto sólo permite que invoque los métodos **const** de su propia clase)
- **Los objetos y variables constantes sólo se pueden inicializar, no modificar**

```
const Punto p(0,0); //un punto de origen no debe cambiar nunca
```

Ejemplo:

Punto
- int x - int y
+ Punto(int a, int b) + operator char* () const + int getx () const + int gety () const + void setx (int n) + void sety (int n) - int min() const + void en(Punto &p) const + void clona(Punto p) + void cambia(Otra &o) const

Otra
- int x
+ void ini(int x=0) + int getx () const

Salida:

(1,2):(5,4):(2,3)
(5,4):(5,4):(4,3)
(4,3):(5,4):(5,4)

void cambia(Otra &o) const

- invoca un método const de su clase
- invoca un método no const de otra clase

El objeto constante p2 solo puede invocar métodos **const**

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

using namespace std;

class Otra {
  int x;
public:
  void ini(int x=0) { this->x=x; }
  int getx() const { return x; }
};

class Punto {
  int x, y;
  int min() const { return x<y?x:y; }
public:
  Punto(int a, int b) { x = a; y = b; }

/*Los métodos de cambio de las coordenadas no
  pueden ser invocados por objetos constantes */
  void setx(int n) { x = n; } //const nunca
  void sety(int n) { y = n; } //const nunca
  void en(Punto &p) const { p.x=x; p.y=y; }
  void clona(Punto p) { x=p.x; y=p.y; } //const nunca
  void cambia(Otra &o) const { int n=min(); o.ini(n); }

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

/*La conversión de tipo si puede ser constante */
  operator char*() const {
    char salida[30];
    sprintf(salida, "(%i,%i)",x,y);
    return strdup( salida );
  }
};

int Punto::gety() const { return y; }

int main(int argc, char *argv[]) {
  Punto p1(1,2),p3(2,3);
  const Punto p2(5,4);
  Otra ot;
  cout << p1 << ":" << p2 << ":" << p3 << "\n";
  p3.setx(4);
//p2.setx(100); //ERROR
  p1.clona(p2);
//p2.clona(p3); //ERROR
  cout << p1 << ":" << p2 << ":" << p3 << "\n";
  p2.en(p3);
  p2.cambia(ot);
  p1.sety(p3.gety()-1);
  p1.setx(ot.getx());
  cout << p1 << ":" << p2 << ":" << p3 << "\n";
//p1.min(); //ERROR min es privado
  system("PAUSE");
  return EXIT_SUCCESS;
}
```

3.2.1 Funciones miembros constantes (sobrecarga)

- **Un método puede sobrecargarse con una versión const y otra no const**
- En tiempo de ejecución el compilador ejecutará una versión u otra dependiendo de si el objeto que invoca el método es un objeto constante o no

Punto
- int x - int y
+ Punto(int a, int b) + operator char* () const + int getx () const + int gety () const + void setx (int n) + void sety (int n) - int min() const + void clona(Punto &p) const + void clona(Punto p) + int proce() const + int proce() + void proce(int n) const

Sobrecarga método clona( )

- void clona(Punto &p) const
- void clona(Punto p)

Sobrecarga método proce( )

- int proce() const
- int proce()
- void proce(int n) const

Salida:

```
(1,2):(5,4):(2,3)
(5,4):(5,4):(5,4)
1,0
10
15
```

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

using namespace std;

class Punto {
  int x, y;
  int min() const { return x<y?x:y; }
public:
  Punto(int a, int b) { x = a; y = b; }

  void setx(int n) { x = n; } //const nunca
  void sety(int n) { y = n; } //const nunca

  void clona(Punto &p) const { p.x=x; p.y=y; }
  void clona(Punto p) { x=p.x; y=p.y; }

  int proce() const;
  int proce();
  void proce(int n) const;
//void fun() const;

  int getx() const { return x; }
  int gety() const { return y; }

  operator char*() const {
    char salida[30];
    sprintf(salida, "(%i,%i)",x,y);
    return strdup( salida );
  }
};

int Punto::proce() const {
  return 0;
}

int Punto::proce() {
  return 1;
}

void Punto::proce(int n) const {
  cout << n << endl;
}
//void Punto::fun() { //no es fun() const!!!; }

int main(int argc, char *argv[]) {
  Punto p1(1,2),p3(2,3);
  const Punto p2(5,4);
  cout << p1 << ":" << p2 << ":" << p3 << "\n";
  p1.setx(5);
//p2.setx(5); //ERROR
  p1.clona(p2);
  p2.clona(p3);
  cout << p1 << ":" << p2 << ":" << p3 << "\n";
  cout << p1.proce() << "," << p2.proce() << "\n";
  p1.proce(10);
  p2.proce(15);
//p1.min(); //ERROR min es privado
  system("PAUSE");
  return EXIT_SUCCESS;
}
```

3.3 Especificador inline para funciones

- Indica al compilador que **sustituya la llamada a una función por su código**
  - La ejecución es más rápida, al no tener que transferir el control.
  - El programa ocupa más espacio al duplicar el código en cada llamada:

Existen 2 formas de definirlas

- Poniendo la palabra inline en la definición (no declaración) de la función

```
void ver(int a); // declaración
...
inline void ver(int a) {
 . . . // definición
}
```

```
x = cuadrado(x) + doble(3);
es sustituido por:
x = x*x + 2*3;
```

- Introduciendo el código de la función en la declaración

```
int doble(int x) { int d=2*x; return d; }
int cuadrado(int a) { return a*a; }
```

Ejemplo:

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

int doble(int n);  // prototipo funcion (declaracion)
int factorial(int n);

class complejo {
 int real,imag; // parte privada
public:
 void ini(int re, int im) {real=re; imag=im;} // inline
 void clon(complejo c) { real=c.real; imag=c.imag; } //inline
 void suma(int re, int im);
 void ver() const;
};

inline void complejo::ver() const {
 cout << real << "+" << imag << "i";
}

void complejo::suma(int re, int im) {
 real += re; imag = imag + im;
}

inline int doble(int n) { return 2*n; }

int factorial(int n) {
 int fact=1;
 for(int i=1; i <= n; i++) fact *= i;
 return(fact);
}
```

Salida del Programa:

```
x: 2+3i y: 1+2i z: 2+3i
c: 2+3i
x: 6+5i y: 12+122i
```

```
int main() {
  complejo x,y,z;
  z.ini(2,3); y.ini(1,2); x.clon(z);
//x es 2+3i, y es 1+2i, z es 2+3i
  const complejo c=z; //ERROR si no inicializo
  int n = doble(6)-1; // n = 11
  cout << "\nx: "; x.ver();
  cout << " y: "; y.ver();
  cout << " z: "; z.ver();
  cout << "\nc: "; c.ver();
  x.suma(4,2);
//c.clon(y); //ERROR c no puede cambiar
  y.suma(n,factorial(5));
  cout << "\nx: "; x.ver();
  cout << " y: "; y.ver();
  system("Pause");
  return 0;
}
```

Nota:

- Hay 2 funciones genéricas doble y factorial: doble() está definida como inline.
- Hay una clase complejo con varias funciones miembros: ini(), clon() y ver() son inline.
- La función miembro clon() accede directamente a los **miembros privados** de su argumento implícito (el objeto que invoca la llamada) y a los **miembros privados del objeto pasado como argumento**, al ser dicho argumento un objeto de su misma clase**.**
- El método **ver()** es constante y es el único método que puede invocar el **objeto const c**
- El objeto **c es const**: sólo puede ser inicializado y no puede invocar ningún método que modifique el objeto

3.4 Funciones amigas (friend)

- Una función amiga (***friend***) de una clase es una función que no ***es miembro de la clase***, pero ***tiene permiso para acceder a sus elementos privados*** a través de un objeto de dicha clase por medio de los operadores punto (.) y flecha (->), sin tener que hacerlo a través de la interfaz pública de dicha clase.
- Una función amiga de una clase puede ser una **función miembro de otra clase** o puede ser una **función no miembro** que no pertenece a ninguna clase.
- Una función puede ser amiga de cuantas clases se quiera.
- Para declarar una **función** o **método** amiga de otra clase, es necesario incluir su prototipo, precedido por la palabra **friend**, en la clase para la que va a ser amiga (es indiferente ponerla como amiga en la parte pública o privada: su accesibilidad la determina la zona donde está definida en la clase de la que es miembro)
- Las funciones amigas violan el principio de encapsulamiento de la programación orientada a objetos, pero es la clase la que decide quiénes son sus amigos (ninguna función se puede autodeclarar como amigo de una clase sin que la propia clase tenga conocimiento).

Punto
- int x - int y
+ Punto(int x=0, int y=0) + void pinta() const **friend** Punto suma (Punto p1, const Punto &p2) **friend** void General::inc(Punto p, int x)

General
- int n
+ void inc(Punto p, int x) + void pinta() const

Salida:
(30, 15) (32)

Punto suma(Punto, const Punto &p2)
p2 referencia constante : - No hace una copia - Al ser constante el compilador prohibe que se modifique

```
#include <iostream>
using namespace std;
class Punto; //declaracion anticipada para que
class General {
 int n;
public:
 void inc(Punto p, int x); //no de error aqui
 void pinta() const { cout <<"(" << n  << ")\n"; }
};
class Punto {
 friend Punto suma(Punto p1, const Punto &p2);
 friend void General::inc(Punto p, int x);
private:
 int x, y;
public:
 Punto( int nx=0, int ny=0 ){ x=nx; y=ny; }
 void pinta() const {cout <<"("<< x <<","<< y << ")\n"; }
};
//inc hay que definirlo aquí tras definir Punto
void General::inc(Punto p, int x) {
 n=p.x+x; //Acceso a atributos privados de p!!
};
Punto suma(Punto p1, const Punto &p2 ) {
 Punto res;
 res.x = p1.x + p2.x; //Acceso atributos privados
 res.y = p1.y + p2.y;
 return res;
}
int main(int argc, char *argv[]) {
 Punto p1(10,5), p2(20,10), p3 = suma(p1, p2);
 General g;
 p3.pinta();
 g.inc(p3, 2);
 g.pinta();
 system("PAUSE");
 return EXIT_SUCCESS;
}
```

- Las funciones amigas no son estrictamente necesarias (todo lo que se puede conseguir con funciones amigas se puede conseguir con funciones no amigas que utilicen la interfaz pública de la clase). Su uso se justifica sólo por razones de eficiencia

Con funciones amigas

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

class Punto; //declaracion anticipada para que

class General {
 int n;
public:
 void inc(Punto p, int x); //no de error
 void pinta() { cout << n << endl; }
};

class Punto {
 friend Punto suma(Punto p1, const Punto &p2);
 friend void General::inc(Punto p, int x);
private:
 int x, y;
public:
 Punto(int nx=0, int ny=0) {x=nx; y=ny;}
 void pinta(){ cout << "(" << x << ","
                   << y << ")\n"; }
};

//inc hay que definirlo tras definir Punto
void General::inc(Punto p, int x) {
 n=p.x+x; //Acceso a parte privada de p!!
};

Punto suma(Punto p1, const Punto &p2 ) {
 Punto res;
 res.x = p1.x + p2.x; //acceso a
 res.y = p1.y + p2.y; //parte privada
 return res;
}

int main(int argc, char *argv[]) {
 Punto p1(10,5), p2(20,10);
 Punto p3 = suma(p1, p2);
 General g;
 p3.pinta();
 g.inc(p3, 2);
 g.pinta();
 system("PAUSE");
 return EXIT_SUCCESS;
}
```

Sin funciones amigas

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

class Punto {
private:
 int x, y;
public:
 Punto(int nx=0, int ny=0) { x=nx; y=ny; }
 void pinta() { cout << "(" << x << ","
                    << y << ")\n"; }
 int getx() const { return x; }
 int gety() const { return y; }
};

class General {
 int n;
public:
 void inc(Punto p, int x) { n=p.getx()+x; }
 void pinta() { cout << n << endl; }
};

Punto suma(Punto p1, const Punto &p2 ) {
 Punto res(p1.getx() + p2.getx(),
           p1.gety() + p2.gety());
 return res;
}

int main(int argc, char *argv[]) {
 Punto p1(10,5), p2(20,10);
 Punto p3 = suma(p1, p2);
 General g;
 p3.pinta();
 g.inc(p3, 2);
 g.pinta();
 system("PAUSE");
 return EXIT_SUCCESS;
}
```

Nota:

- En la función suma, p2 es una referencia **constante**, por tanto solo puede invocar métodos **constantes**, es por ello por lo que getx() y gety() son declarados métodos **constantes**
  - tipo clase::método(parámetros) **const** método constante: el compilador impide que modifique los atributos de la clase
  - const tipo &p referencia constante: - el compilador impide que se modifique el objeto p - no se pasa una copia del objeto sino el objeto

3.5 Clases amigas (friend)

- Una clase puede declararse como amiga de otra/s clase/s.
- Cuando una clase A se declara amiga de otra B, todas las funciones miembros de la clase A pasan a ser amigas de la clase B.
- Es la forma más rápida de declarar todas las funciones miembros de una clase como amigas de otra.

```
class Cualquiera {
 . . .
 friend class Amiga;
};
```

Las funciones miembro de la clase ***Amiga*** pueden acceder a la parte privada de la clase ***Cualquiera***, pero los métodos de la clase ***Cualquiera*** no pueden acceder a la parte privada de la clase ***Amiga***.

Punto
- int x - int y
+ Punto(int x=0, int y=0) + void pinta() **friend** class Amiga

General
+ void inc(Punto p, int x) + void ponACero(Punto &p)

Salida:

(10, 5)
(0, 0)
0
-2

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

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

class General {
 int n;
public:
 void ponACero(Punto &p) {
  p.x = 0;  //Acceso a parte privada de Punto
  p.y = 0;
  n = 0;
 }
 void inc(Punto p, int x) {
  n = p.x + x; //Acceso a parte privada de Punto
 }
 void pinta() { cout << n << endl; }
};

int main(int argc, char *argv[]) {
 Punto p1(10,5);
 General g;
 p1.pinta();
 g.ponACero(p1);
 p1.pinta();
 g.pinta();
 g.inc(p1, -2);
 g.pinta();

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

3.5 Clases amigas (friend) (caso de referencia cruzada)

- Si se quiere que dos clases tengan acceso mutuo a los miembros privados de la otra clase, cada una debe declararse como amiga de la otra de forma recíproca.
- **Problema**: no podemos utilizar una clase antes de declararla.
- **Solución**: ***declarar anticipadamente*** (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 **friend** int suma(Amiga1 a1, Amiga2 a2)

Amiga2
- int privada
+ void modificaAmiga1 (Amiga1 &a1, int val) + void pinta() **friend** class Amiga1 **friend** int suma(Amiga1 a1, Amiga2 a2)

Salida:

20
10
suma = 30

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

class Amiga2; // declaración anticipada

class Amiga1 {
    friend class Amiga2;
    friend int suma(Amiga1 a1, Amiga2 a2);
private:
    int privada;
public:
    void modificaAmiga2(Amiga2 &a2, int val);
    /* Aquí no podemos definir el método porque aún
el compilador no ha llegado a leer la definición de
la clase Amiga2 y no sabe que atributos tiene... */
    void pinta() { cout << privada << "\n"; }
};

class Amiga2 {
    friend class Amiga1;
    friend int suma(Amiga1 a1, Amiga2 a2);
private:
    int privada;
public:
    void modificaAmiga1(Amiga1 &a1, int val) {
/* Aquí si podemos porque ya hemos declarado Amiga1 */
     a1.privada = val;
    }
    void pinta(){ cout << privada << "\n"; }
};

int suma(Amiga1 a1, Amiga2 a2 ) {
  return (a1.privada+a2.privada);
}

/* Ahora sí que podemos definir el método porque ya
hemos declarado los atributos de la clase Amiga2 */

void Amiga1::modificaAmiga2(Amiga2 &a2, int val) {
    a2.privada = val;
}

int main(int argc, char *argv[]) {
    Amiga1 am1;
    Amiga2 am2;
    am1.modificaAmiga2(am2,10);
    am2.modificaAmiga1(am1,20);
    am1.pinta();  am2.pinta();
    cout << "suma = " << suma(am1, am2) << "\n";
    system("PAUSE");   return EXIT_SUCCESS;
}
```

4. Parámetros por Valor y por Referencia

- Los ***parámetros*** son los valores que se le pasan a la función al ser llamada. Cada parámetro se comporta dentro de la función como una variable local**.**
- Los parámetros escritos en la sentencia de llamada a la función se llaman ***parámetros reales o argumentos***.
- Los que aparecen en la descripción de la función se llaman ***parámetros formales***.
- Los parámetros formales son sustituidos por los reales (argumentos) en el orden de llamada. Al emparejarse éstos deben coincidir en número y en tipo, excepto las funciones con parámetros formales con valores por defecto.

4.1 Parámetros por Valor y por Referencia

Parámetros por valor o copia	**Parámetros por referencia**
▪ ***Los cambios producidos dentro de la función no afectan a la variable real usada como argumento.***	▪ ***Los cambios producidos dentro de la función afectan a la variable con la que se realiza la llamada.***
▪ En la llamada se pasa una copia del valor del argumento.	▪ En la llamada se transfiere la propia variable.
▪ El parámetro real puede ser una constante, variable o expresión del tipo indicado en el parámetro formal.	▪ El parámetro real sólo puede ser una variable, ya que se trasfiere ésta.
	▪ Se indica poniendo el carácter & delante del nombre del parámetro.

Ejemplo:

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

class objeto; //declaracion anticipada
void permutar(objeto &a, objeto &b); //ref
void cambiamal(objeto a, objeto b);  //valor

class objeto {
 int n;
public:
 int get() { return n; }
 void set(int i) { n = i; }
};

void permutar(objeto &a, objeto &b) {
  int aux=a.get();
  a.set(b.get());
  b.set(aux);
}

void cambiamal(objeto a, objeto b) {
  int aux=a.get();
  a.set(b.get());
  b.set(aux);
}
```

```
int main(){
 objeto x,y;
 x.set(5);
 y.set(7);
 cambiamal(x, y);
 cout << "\nx: " << x.get()
      << " y: " << y.get();
 permutar(x, y);
 cout << "\nx: " << x.get()
      << " y: " << y.get();
 system("Pause"); return 0;
}
```

Salida del Programa:

x: 5 y: 7
x: 7 y: 5

4.2 Variables de tipo Referencia

- Es un alias o sinónimo para una variable o un objeto.
- Se declaran con el operador & y deben ser inicializadas a otra variable

```
int i=2;
int& jref;     // declaración no válida
int& iref = i; // declaración válida
iref = 6;      // es lo mismo que poner i=6
```

- Una función puede retornar una variable tipo referencia

```
int& maxref(int& a, int& b) {
 if (a >= b)  return a;
 else  return b; //con & devuelve b, no una copia de b
}               //sin & devuelve una copia de b, no b
```

- Al devolver una referencia, la llamada a la función puede estar a la izquierda del operador de asignación:

```
maxref(i, j) = 0; // asigna un 0 al parametro mayor
```

- **Peligro al retornar referencias**: el retorno puede estar indefinido si no programamos bien

```
int& malprogramado(int n) {
 return n; //retorna n que al ser local es destruido al terminar
}
```

4.3 Referencias constantes

Cuando la variable que queremos pasar como parámetro por valor a una función **ocupa mucha memoria, se suele pasar** dicho parámetro **por referencia por motivos de eficiencia**, ya que al pasarlo por valor el compilador tiene que realizar una copia del parámetro y al pasarlo por referencia, mediante un puntero o un alias (referencia), no se pierde tiempo en hacer una copia, sino que se pasa directamente el parámetro en cuestión.

Para evitar que accidentalmente se modifique el parámetro (al ser pasada por referencia, en vez de por valor) dentro de la función, es conveniente declarar el parámetro constante (**const).**

- El especificador **const** se puede utilizar tanto con variables (y referencias) como con punteros.
- Una **variable const** o un **puntero a una variable const** no puede cambiar su valor durante toda la ejecución del programa (**sólo puede ser inicializado**).
- Un **objeto const** o un **puntero a un objeto const** sólo puede invocar **métodos const** y no puede cambiar durante toda la ejecución del programa (**sólo puede ser inicializado**).

Recuerda:

Si por motivos de eficiencia, en vez de pasar un objeto por valor lo pasamos por referencia (para evitar que el compilador tenga que realizar una copia del parámetro)

&&

por motivos de seguridad, para evitar que el parámetro pueda ser modificado (cuando se pasa por valor no importa ya que es una copia) **la referencia la declaramos constante**

↓

debemos etiquetar como constantes aquellos métodos que lo sean, ya que si no, **no podrán ser invocados por el parámetro por referencia contante**

Ejemplo: Supongamos una clase que tiene una gran cantidad de datos…

uso habitual (menos eficiente)

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

class General {
private:
 int x[1000], n;

public:
 General(int n, int ini, int inc);
 int getn() { return n; }
 int getx(int i) { return x[i]; }
 int setx(int i, int v) { x[i]=v; }
 void ver() { for(int i=0; i<n; i++)
                cout << x[i] << " "; }
 bool compara(General g);
};

bool General::compara(General g) {
 bool v = n==g.n;
 for (int i=0; v && i<n; i++)
   v = x[i]==g.x[i];
 return v; //g.n=0 no importa,es copia
}

General::General(int n, int ini, int inc) {
  this->n=n;
  for(int j=0,i=ini; j<n; i+=inc, j++)
    x[j]=i;
}

General suma(General p1, General p2) {
 int a=p1.getn(), b=p2.getn();
 int n=a<b?a:b;
 General res(n, 0, 0);
 for(int i=0;i<n;i++)
  res.setx(i, p1.getx(i)+p2.getx(i));
 return res;
}

int main(int argc, char *argv[]) {
 General a(5,-4,2), b(3,2,5), c=suma(a,b);
 c = suma(a, b);
 cout << "a: "; a.ver(); cout << endl;
 cout << "b: "; b.ver(); cout << endl;
 cout << "c: "; c.ver(); cout << endl;
 if (!a.compara(b)) cout << "a<>b\n";
 a=b;
 cout << "a: "; a.ver(); cout << endl;
 cout << "b: "; b.ver(); cout << endl;
 if (a.compara(b)) cout << "a = b\n";
 system("PAUSE");
 return EXIT_SUCCESS;
}
```

uso de referencias constantes (más eficiente)

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

class General {
private:
 int x[1000], n;

public:
 General(int n, int ini, int inc);
 int getn() const { return n; }
 int getx(int i) const { return x[i]; }
 int setx(int i, int v) { x[i]=v; }
 void ver() const { for(int i=0; i<n; i++)
                      cout << x[i] << " ";}
 bool compara(const General &g);
};

bool General::compara(const General &g) {
 bool v = n==g.n;
 for (int i=0; v && i<n; i++)
   v = x[i]==g.x[i];
 return v; //g.n=0 no permitido, es const
}

General::General(int n, int ini, int inc) {
  this->n=n;
  for(int j=0,i=ini; j<n; i+=inc, j++)
    x[j]=i;
}

General suma(const General &p1, const General &p2){
 int a=p1.getn(), b=p2.getn();
 int n=a<b?a:b;
 General res(n, 0, 0);
 for(int i=0;i<n;i++)
  res.setx(i, p1.getx(i)+p2.getx(i));
 return res;
}

int main(int argc, char *argv[]) {
 General a(5,-4,2), b(3,2,5), c=suma(a,b);
 c = suma(a, b);
 cout << "a: "; a.ver(); cout << endl;
 cout << "b: "; b.ver(); cout << endl;
 cout << "c: "; c.ver(); cout << endl;
 if (!a.compara(b)) cout << "a<>b\n";
 a=b;
 cout << "a: "; a.ver(); cout << endl;
 cout << "b: "; b.ver(); cout << endl;
 if (a.compara(b)) cout << "a = b\n";
 system("PAUSE");
 return EXIT_SUCCESS;
}
```

```
Salida:
a: -4 -2  0  2  4
b:  2  7 12
c: -2  5 12
a <> b
a:  2  7 12
b:  2  7 12
a = b
```

- suma( ) al usar referencias constantes 'obliga' a etiquetar getn() y getx() como const
- suma() y compara() son más eficientes usando referencias constantes
- aunque no es obligatorio, **ver() se etiqueta const ya que no modifica la clase**
- en izquierdo no importa (es copia), en derecho si importa (es original) pero no está permitido

4.4 Valores de retorno constantes. Consideraciones

- **Una referencia (o un puntero) a un atributo privado** permite el **acceso** y/o modificación **de éste desde fuera de la clase!!!!**.
- Esto permite "ahorrarnos" métodos ya que la referencia devuelta la podemos utilizar como consulta y también como modificación.

Clase con métodos sin referencias devueltas	Clase equivalente con referencias devueltas

`#include <iostream>`
`using namespace std;`
 	
`class Punto {`
`  int x, y;`
`public:`
`  Punto(int a, int b) { x=a; y=b; }`
`  int getx() { return x; }`
`  int gety() { return y; }`
`  void set(int a, int b) { x=a; y=b; }`
`};`
 	
`int main(int argc, char *argv[]) {`
`  Punto x(3,2);`
`  cout <<x.getx()<<","<<x.gety()<<endl;`
`  x.set(5,7);`
`  cout <<x.getx()<<","<<x.gety()<<endl;`
`  system("PAUSE"); return EXIT_SUCCESS;`
`}`
	`#include <iostream>`
	`using namespace std;`
	
	`class Punto {`
	`  int x, y;`
	`public:`
	`  Punto(int a, int b) { x=a; y=b; }`
	`  int& refx() { return x; }`
	`  int& refy() { return y; }`
	
	`};`
	
	`int main(int argc, char *argv[]) {`
	`  Punto x(3,2);`
	`  cout <<x.refx()<<","<<x.refy()<<endl;`
	`  x.refx()=5; x.refy()=7;`
	`  cout <<x.refx()<<","<<x.refy()<<endl;`
	`  system("PAUSE"); return EXIT_SUCCESS;`
	`}`

- **Peligro**: Supongamos que y debe ser siempre positivo, si no, no se debe modificar

Clase robusta	Clase no robusta (fallo de seguridad)

`class Punto {`
`  int x, y;`
`public:`
`  Punto(int a, int b) { x=a; y=b; }`
`  int getx() { return x; }`
`  int gety() { return y; }`
`  void set(int a, int b) {if (b>0){x=a;y=b;}}`
`};`
 	
`int main(int argc, char *argv[]) {`
`  Punto x(3,2);`
`  cout <<x.getx()<<","<<x.gety()<<endl;`
`  x.set(5,-7); //no modifica al ser negativo`
`  cout <<x.getx()<<","<<x.gety()<<endl;`
`  system("PAUSE"); return EXIT_SUCCESS;`
`}`
	`class Punto {`
	`  int x, y;`
	`public:`
	`  Punto(int a, int b) { x=a; y=b; }`
	`  int& refx() { return x; }`
	`  int& refy() { return y; }`
	`  void set(int a, int b) {if (b>0){x=a;y=b;}}`
	`};`
	
	`int main(int argc, char *argv[]) {`
	`  Punto x(3,2);`
	`  cout <<x.refx()<<","<<x.refy()<<endl;`
	`  x.refx()=5; x.refy()=-7; //negativo!!!`
	`  cout <<x.refx()<<","<<x.refy()<<endl;`
	`  system("PAUSE"); return EXIT_SUCCESS;`
	`}`

- **Solución**: no retornar referencias o punteros desde un método cuando haya restricciones.

Si queremos devolver referencias o punteros por motivos de eficiencia (para ahorramos hacer una copia si lo devuelto ocupa mucha memoria) **o porque no hay más remedio** (para devolver una cadena de caracteres hay que devolver un puntero a dicha cadena) **y queremos evitar que desde fuera de la clase se pueda modificar el atributo devuelto debemos devolverlo como una referencia constante o como un puntero constante**

```
struct enorme {
 int T[1000];
 float f;
};
class General {
  char *cad; //puntero a cadena caracteres
  enorme x; //ocupa mucha memoria
public:
  const char *getcad() { return cad; }
  const enorme& refx() { return x; }
  enorme getx() { return x; } //ineficiente
};
```

En el ej. el método refx( ) es mas eficiente que getx( ) ya que evita tener que copiar x que ocupa mucha memoria

4.4 Valores de retorno constantes. Consideraciones

- Cuando un método devuelve una referencia de un atributo privado o un puntero de un atributo privado (bien por eficiencia, para evitar hacer una copia, o bien porque no hay mas remedio) y queramos evitar que se utilice esa referencia devuelta para modificar el atributo debemos etiquetar esa referencia como **const** (referencia constante).

problema

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

class cadena {
  char *cad;
  int x[5], y;
public:
  cadena(int ini, int incr, char
cad[]);
  int *leer() { return x; }
  int &leer0() { return x[0]; }
  void ver();
  char *leercad() { return cad; }
  int &refy() { return y; }
  int *puny() { return &y; }
};

void cadena::ver() {
  cout << cad << ":";
  for(int i=0;i<5;i++)
    cout << x[i];
  cout << " -> " << y << endl;
}

cadena::cadena(int ini, int incr, char cad[]){
  y=0;
  for(int j=0,i=ini; j<5; j++,i+=incr) {
    x[j]=i;
    y+=x[j];
  }
  this->cad = new char[strlen(cad)+1];
  strcpy(this->cad, cad);
}

int main(int argc, char *argv[]) {
 char *c;
 int *pn;
 cadena x(1,1,"cadena x");
 x.ver();
 c=x.leercad();
 strcpy(c,"CADENA X"); //x.cad="CADENA X"
 x.leer()[2]=0; //x.x[2]=0
 x.refy()++; //x.y++
 *(x.puny())+=2; //x.y+=2
 x.ver();
 int &n=x.refy(); n=4; //x.y=4
 pn=x.puny(); (*pn)--; //x.y--
 n=n*2; //x.y=x.y*2
 x.leer0()=8; //x.x[0]=8
 int *p=&(x.leer0()); p[1]=9; //x.x[1]=9
 x.ver();
 cout << x.leercad() << x.refy() << "\n";
 system("PAUSE"); return EXIT_SUCCESS;
}
```

solución

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

class cadena {
  char *cad;
  int x[5], y;
public:
  cadena(int ini, int incr, char cad[]);
  const int *leer() { return x; }
  const int &leer0() { return x[0]; }
  void ver();
  const char *leercad() { return cad; }
  const int &refy() { return y; }
  const int *puny() { return &y; }
};

int main(int argc, char *argv[]) {
 char *c;
 int *pn;
 cadena x(1,1,"cadena x");
 x.ver();
 c=x.leercad(); //ERROR no ok: uso indebido
 strcpy(c,"CADENA X"); //x.cad="CADENA X"
 x.leer()[2]=0; //x.x[2]=0 ERROR
 x.refy()++; //x.y++ ERROR no ok: uso indebid

 *(x.puny())+=2; //x.y+=2 ERROR
 x.ver();
 int &n=x.refy(); n=4; //x.y=4 ERROR
 pn=x.puny(); (*pn)--; //x.y-- ERROR
 n++; //x.y++ ERROR
 x.leer0()=8; //x.x[0]=8 ERROR
 int *p=&(x.leer0()); p[1]=9; //ERROR
 x.ver();
 cout << x.leercad() << x.refy() << "\n";//ok
 system("PAUSE"); return EXIT_SUCCESS;
}
```

Con el puntero char *c puedo modificar el atributo privado char *cad!!!

Con el puntero int *pn puedo modificar el atributo privado int y!!!

Con la referencia int &n puedo modificar el atributo privado int y!!!

Con los métodos leer(), leer0() puedo modificar la tabla privada int x[5]!!!

Salida:

```
cadena x:12345 -> 15
CADENA X:12045 -> 18
CADENA X:89045 -> 6
CADENA X6
```

5. Sobrecarga de Métodos y Operadores

- **C++ permite sobrecargar funciones, métodos y operadores**, es decir, permite que existan varias funciones, métodos y operadores con el mismo nombre
- **Ventajas**:
  - Ayuda a reducir la complejidad de un programa, ya que permite usar el mismo nombre y/o operador para operaciones relacionadas, (más intuitivo).
  - Permite que funciones con el mismo nombre hagan cosas diferentes (polimorfismo)
  - Permite utilizar los mismos operadores (y realizar las mismas operaciones) con los nuevos tipos de datos (clases) que creemos (siempre que redefinamos el operador)

5.1 Sobrecarga de funciones

- C++ permite definir varias funciones distintas con un mismo nombre, **siempre y cuando** difieran en el número y/o el tipo de sus argumentos
- El número y/o tipo de los argumentos de la llamada indicará la función a usar.

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

Si las funciones sólo difieren en el tipo de datos que devuelven no se pueden sobrecargar:

```
int funcion(int x);
float funcion(int x);
int main() {
   . . .          //ambigüedad
   funcion(2); //a cual llama?
}
```

Si sólo difieren en el tipo de sus argumentos, puede ser ambiguo, aunque la función no sea ambigua:

```
float dividir(float x);
double dividir(double x);
int main() {
  . . .          //ambigüedad
  dividir(15); //15 double o float?
}
```

Ejemplos:

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

int vabs(int n);
long vabs(long n);
double vabs(double n);

int main() {
 long b=-50000;
 cout << "\n|-6|: " << vabs(-6);
 cout << "\n|-50000|: " << vabs(b);
 cout << "\n|-6.7|: " << vabs(-6.7);
 system("Pause"); return 0;
}

int vabs(int n) {
 return(n < 0 ? –n : n);
}

long vabs(long n) {
 if (n < 0)  return (-n);
 else  return n;
}

double vabs(double n) {
 if (n < 0)  n = -n;
 return n;
}
```

Pantallla:

|-6|: 6
|-50000|: 50000
|-6.7|: 6.7

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

void copiar(char copia[20],
            const char original[20]) {
  strcpy(copia, original);
}

void copiar(char copia[20],
            const char original[20],
            int n) {
  strncpy(copia, original, n);
}

int main() {
  char cad_a[20], cad_b[20];
  copiar(cad_a, "Buen Dia");
  copiar(cad_b, "Buen Dia", 4);
  cad_b[4] = '\0';
  cout << cad_a << " y " << cad_b;
  system("Pause");
  return 0;
}
```

Pantalla:

Buen Dia y Buen

5. Sobrecarga de Métodos y Operadores

5.2 Sobrecarga de funciones miembros (métodos)

- C++ permite definir varias funciones miembros (métodos) con un mismo nombre, **siempre y cuando** difieran en el número y/o el tipo de sus argumentos
- El número y/o tipo de los argumentos de la llamada indicará la función miembro a usar.
- También permite tener dos versiones para un método, una **const** y la otra no, es decir, **permite tener dos funciones miembros identicas** (mismo nombre, número y tipo de argumentos) **siempre y cuando** una sea una **función miembro constante y la otra no**
- El objeto que realiza la llamada determinará la función miembro a usar:
  - si el objeto que realiza la llamada es un **objeto constante** (**const**) se **ejecutará el método constante** (**const**)
  - si el **objeto** que invoca el método **no es const, ejecutará el método no const**

Ejemplo:

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

using namespace std;

class Punto {
 int x, y;
public:
 Punto(int a, int b) { x = a; y = b; }
 Punto() { x = 0; y = 0; }

 void set(int a, int b) { x=a; y=b; }

 int pr() const { return 0; }
 int pr() { return 1; }
 Punto suma(int n) const;
 Punto suma(int a, int b) const;

 int getx() const { return x; }
 int gety() const { return y; }

 operator char*() const {
    char salida[30];
    sprintf(salida, "(%i,%i)",x,y);
    return strdup( salida );
  }
};

Punto Punto::suma(int n) const {
  Punto p;
  p.set(x+n,y);
  return p;
}

Punto Punto::suma(int a, int b) const {
  return Punto(x+a, y+b);
}
```

```
int main(int argc, char *argv[]) {
  Punto p1(1,2), p2;
  const Punto p3(5,4);
  cout << p1 <<":"<< p2 <<":"<< p3 <<"\n";
  cout << p1.pr() << "\n";
  cout << p3.pr() << "\n";
  p1.set(5,p1.gety());
//p3.set(5,2);  //ERROR p3 es const y set no
  p1=p1.suma(10);
  p2=p3.suma(1,2);
//p3=p3.suma(10); //ERROR p3 es const y = no
  cout << p1 <<":"<< p2 <<":"<< p3 <<"\n";
  system("PAUSE");
  return EXIT_SUCCESS;
}
```

Pantallla:

(1,2):(0,0):(5,4)
1
0
(15,2):(6,6):(5,4)

5. Sobrecarga de Métodos y Operadores

5.3 Valores por defecto en los argumentos (argumentos implícitos)

- Son asignados cuando **se omiten** los parámetros en la **llamada**.
- El **valor por defecto se pone** junto al parámetro **tras el signo =**
- Los **argumentos implícitos**:
  - **deben estar al final** de la lista
  - **Si se omite uno en la llamada, deben omitirse los que sigan**
  - Los valores por defecto **deben ser constantes** o **variables globales**.
  - **sólo deben indicarse una vez**: en el prototipo de la función o método.
  - **Es una forma de sobrecarga**

Ejemplo:

```
float f(int x, int y = 1,
               int z = 0);
...
int main() {
 float v;
 v = f(10);       // f(10, 1, 0)
 v = f(10, 7);    // f(10, 7, 0)
 v = f(10, 2, 3);
 ...
}
```

Pantallla:

3,2
0,0
1,0
5,2
4,2
5,2
7,5
9,4

Ejemplos:

```
...
int perimetro(int b, int h) {
  return(b * 2 + h * 2);
}
int perimetro(int b) {
  return(b * 4);
}
int main() {
  perimetro(6, 4);
  perimetro(8);
  ...
}
```

```
// versión argumentos implícitos
int perimetro(int b, int h = 0) {
  if (h == 0)
    h = b;
  return (b * 2 + h * 2);
}
```

```
// versión argumentos implícitos
class Punto {
 int x,y;
public:
 Punto(int a=0, int b=0) { x = a; y = b; }
 Punto add(Punto p, Punto q=0);
 void inc23(Punto p=Punto(2,3))
    { x+=p.x; y+=p.y; }
 void ver() { cout << x <<","<< y <<endl; }
};
Punto Punto::add(Punto p, Punto q) {
 x=x+p.x+q.x; y+=p.y+q.y; return (*this);
}
```

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

using namespace std;

class Punto {
  int x,y;
public:
  Punto() { x = 0; y = 0; }
  Punto(int a) { x = a; y = 0; }
  Punto(int a, int b) { x = a; y = b; }
  Punto add(Punto p) {
    x+=p.x; y+=p.y; return (*this);
  }
  Punto add(Punto p, Punto q);
  void inc23() { x+=2; y+=3; }
  void inc23(Punto p) { x+=p.x; y+=p.y; }
  void ver() { cout << x <<","<< y <<endl; }
};

Punto Punto::add(Punto p, Punto q) {
  x=x+p.x+q.x; y+=p.y+q.y; return (*this);
}

int main() {
  Punto x(3,2), y, z=1; //z=Punto(1);
  x.ver(); y.ver(); z.ver();
  y.add(x, z);
  x=z.add(y);
  x.ver(); y.ver(); z.ver();
  x.inc23(); y.inc23(z);
  x.ver(); y.ver();
  system("PAUSE"); return EXIT_SUCCESS;
}
```

Nota:

- La inicialización z=1 y la asignación del valor por defecto del parámetro q=0 es posible porque:
  1. existe un constructor con 1 parámetro
  2. El compilador realiza un casting int --> Punto invocando el constructor con 1 argumento

5. Sobrecarga de Métodos y Operadores

5.4 Sobrecarga de operadores

- **Los operadores** de C++ (excepto . :: .* ?) **pueden ser sobrecargados** (overloaded), de forma que se puedan utilizar con los objetos de las nuevas clases creadas de la misma forma que se usan con los tipos predefinidos (redefiniendo su comportamiento para adaptarlo a las nuevas clases sobre las que queremos que actúe).
- Cuando se sobrecarga un operador:
  - **No se puede cambiar** su **precedencia** y **asociatividad**.
  - **No puede modificarse el número de operandos** que tiene.
  - Al menos **uno de los operandos debe ser de la clase que lo sobrecarga**.
  - **No se permite definir nuevos operadores**
- **En una clase, un mismo operador puede estar sobrecargado varias veces siempre y cuando difiera en el tipo de sus argumentos**. Dependiendo del tipo de objeto/s que tenga como operando/s, el operador puede actuar de un modo distinto.
- El tipo del argumento/s en la llamada determinará el método a usar con ese operador.
- Un **operador** puede sobrecargarse mediante:
  - Un **método** de la clase (**preferible**, por si usamos herencia).
  - Una función (por eficiencia dicha función suele declararse amiga **friend** de la clase)

  siempre y cuando el operando izquierdo sea de la clase para la que se sobrecarga
- Cuando el **operando izquierdo no es un objeto de la clase** para la que se sobrecarga, dicho operador **únicamente puede sobrecargarse mediante una función** (o mediante un método de la clase del operando que quede a la izquierda).

Al sobrecargar un operador …

- **Relacional o lógico**: debe devolver un entero o booleano (**bool**) que indique verdad o falso
- **Asignación: operator=()** debe devolver el objeto para permitir **cadenas de asignaciones**

  o1 = o2 = o3;
- **Aritmético**: debe devolver un objeto de la clase, para permitir expresiones aritmética complejas y/o para que el resultado se pueda asignar a otro objeto
  - **operator+()** debe devolver un objeto **para permitir cadenas de +: (idem con - * /)**

    o3=o1+o2+o1; o3=-o1; o3=o1-o2-o3; o3=o1*o2*o3; o3=o1/o2/o3; c=a+b-c*d/a;
  - **operator++()** debe devolver **el objeto que incrementa para poder hacer:**

    obj2 = obj1++; obj2 = ++obj1;
    - **Si ++ precede al operando (++op),** se usa la función **operator++().** Idem con --
    - **Si ++ sigue al operando (op++),** se usa la función **operator++(int notused).**
- **El operando derecho puede ser de otra clase**:

  claseA operator+ (int y); claseA operator+ (claseA x); claseA operator+(claseB y);
  claseA operator+ (int x, claseA y); claseA operator+ (claseB x, claseA y);
- **Si un operador devuelve algo que pueda ser destino, debe retornar una referencia**

  Ej: char & cadena::operator[ ] (int i); ← ***permite hacer*** → cadena x("Cola"); x[3]='t';

  char & cadena::set(int i); ← ***permite hacer*** → cadena x("Cola"); x.set(3)='t';

5. Sobrecarga de Métodos y Operadores

5.4.1 Sobrecarga de operadores mediante una función miembro (método)

- El primer operando debe ser un objeto de esa clase (constituye el **argumento implícito**)
- Sintaxis:

```
tipo nombre_clase::operator#([tipo operador_derecho]) {
    [declaraciones;]
    sentencias;
    return valor_devuelto;
}
```

donde # representa el operador a sobrecargar

- Si una función miembro (método) sobrecarga un operador:
  - **binario:** el operando izq se pasa implícitamente y el derecho se pasa como argumento
  - **unario:** el operando se pasa implícitamente (no hay parámetros).
- Dada una clase para la que se ha sobrecargado un operador # determinado, la expresión

a # b (operador binario)	**#b** (operador unario)
se interpreta como	se interpreta como
a.operator#(b):	**b.operator#( ):**

5.4.2 Sobrecarga de operadores mediante una función (no miembro)

- Es la única forma de realizar la sobrecarga de un operador cuando el primer operando no es un objeto de la clase para la que se sobrecarga
- Por motivos de eficiencia dicha función se suele declarar amiga (**friend**) de la clase, pero no es obligatorio que lo sea
- No permite sobrecargar **el operador de asignación =** (tampoco se permite la sobrecarga de los operadores . :: .* ?)
- Sintaxis:

```
tipo operator#(tipo operador_izq [, tipo operador_dcho]) {
    [declaraciones;]
    sentencias;
    return valor_devuelto;
}
```

donde # representa el operador a sobrecargar (+, -, *, /, >, <, !=, ==, etc.)

- Si una función sobrecarga un operador:
  - **binario:** hay que pasar explícitamente el operando izquierdo y derecho.
  - **unario:** hay que pasar explícitamente el operando.
- Dada una clase para la que se ha sobrecargado un operador # determinado, la expresión

a # b (operador binario)	**#b** (operador unario)
se interpreta como	se interpreta como
operator#(a, b):	**operator#(b):**

Ejemplo:

Implementar una clase fracción que permita operar con fracciones

Las operaciones que debe ser capaz de hacer son:

- **Sumar y restar fracciones y enteros.** Ej: x=2/3+4/2-5/2, x=2/3+7, x=7+2/3, x=2/3-7-3/2
- **Cambiar el signo de una fracción(-):** Ej: x=-2/3, x=-4 (x se convierte en -4/1)
- **Incrementar el numerador (prefijo y posfijo):** Ej: Si y=2/3 x=++y hace que x=3/3 y=3/3
  Ej: Si y=2/3 x=y++ hace que x=2/3 y=3/3
- **Comparar 2 fracciones (> y ==):** Ej: Si x=4/6 y=2/3 x==y es true x>y es false

Implementa un programa que genere la siguiente salida a partir de a(3,2), b(3), c(2,3) y d:

```
           a     b       c       d
           3/2   3/1     2/3     0/1
a=a+a      6/2   3/1     2/3     0/1
b=a-c      6/2   14/6    2/3     0/1
a=-b       -14/6 14/6    2/3     0/1
d=1+a-c+b  -14/6 14/6    2/3     2/6
a=++b      15/6  15/6    2/3     2/6
a=c++      2/3   15/6    3/3     2/6
a=b+5      45/6  15/6    3/3     2/6
c=5+b      45/6  15/6    45/6    2/6
a=a>b?-d:b -2/6  15/6    45/6    2/6
a y c distintos
```

Solución:

Sobrecarga con función miembro (métodos)

```
#include <iostream> //cin, cout
using namespace std;

int mcm(int a, int b) {
 int comun, mayor;
 if (a<0) a=-a;
 if (b<0) b=-b;
 mayor = a>b ? a: b;
 for (comun=mayor;   ; comun+=mayor)
   if (comun%a==0 && comun%b==0)
       return comun;
}

class frac {
 int n, d;
public:
 frac() { n=0; d=1; }
 frac(int n, int d=1) { set(n,d); }
 int getn() { return n; }
 int getd() { return d; }
 void set(int n, int d=1);
 frac operator+(frac q);
 frac operator-(frac q);
 frac operator-();
 frac operator++();
 frac operator++(int flag);
 int operator==(frac q);
 bool operator>(frac q);
//operator+(int,fraq) no lo declaro amiga
 void ver();
};
```

Sobrecarga con función (friend)

```
#include <iostream> //cin, cout
using namespace std;

int mcm(int a, int b) {
 int comun, mayor;
 if (a<0) a=-a;
 if (b<0) b=-b;
 mayor = a>b ? a: b;
 for (comun=mayor;   ; comun+=mayor)
   if (comun%a==0 && comun%b==0)
       return comun;
}

class frac {
 int n, d;
public:
 frac() { n=0; d=1; }
 frac(int n, int d=1) { set(n,d); }
 int getn() { return n; }
 int getd() { return d; }
 void set(int n, int d=1);
 friend frac operator+(frac p, frac q);
 friend frac operator-(frac p, frac q);
 friend frac operator-(frac p);
 friend frac operator++(frac &p);
 friend frac operator++(frac &p, int flag);
 friend int operator==(frac p, frac q);
 friend bool operator>(frac p, frac q);
//friend frac operator+(int n, frac q);
 void ver();
};
```

```
void frac::set(int n, int d) {
 if (d==0) exit(1);
 this->n=n; this->d=d;
}

frac frac::operator+(frac q) {
 frac suma;
 int comun = mcm(d, q.d);
 suma.d = comun;
 suma.n = n*comun/d + q.n*comun/q.d;
 return suma;
}

frac frac::operator-(frac q) {
 frac menosq(-q.n, q.d);
 return (*this)+menosq;
}

frac frac::operator-() {
 return frac(-n, d);
}

frac frac::operator++() { // ++obj
 n++;
 return *this; //devuelve una copia del
}              //objeto que invoca el ++

frac frac::operator++(int flag) {//obj++
 frac copia(*this);
 n++;
 return copia;
}

frac operator+(int n, frac q) {
 frac suma(n*q.getd()+q.getn(), q.getd());
 return suma;
}

int frac::operator==(frac q) {
 if (n*q.d==d*q.n)
  return 1;
 return 0;
}

bool frac::operator>(frac q) {
 return (n*q.d>d*q.n);
}

void frac::ver() { //signo en numerador
 if (d>0) cout << n << "/" << d << "\t";
 else cout << -n << "/" << -d << "\t";
}

int main() {
 frac a(3,2),b(3),T[]={frac(2,3),frac()};
 cout << "\n\t  a\tb\tc\td\n\t  ";
 a.ver();b.ver();T[0].ver();T[1].ver();//(*)
 a=a+a;       cout << "\na=a+a\t  ";//(*)
 b=a-T[0];    cout <<"\nb=a-c\t  "; //(*)
 a=-b;        cout <<"\na=-b\t  ";  //(*)
 T[1]=1+a-T[0]+b; cout <<"\nd=1+a-c+b ";//(*)
 a=++b;       cout <<"\na=++b\t  "; //(*)
 a=T[0]++;    cout <<"\na=c++\t  "; //(*)
 a=b+5;/*(1)*/cout << "\na=b+5\t  "; //(*)
 T[0]=5+b;    cout <<"\nc=5+b\t  "; //(*)
 a=a>b?-T[1]:b; cout <<"\na=a>b?-d:b"; //(*)
 if (a==T[0]) cout << "\na y c iguales\n";
 else         cout << "\na y c distintos\n";
 system("PAUSE"); return EXIT_SUCCESS;
}
```

```
void frac::set(int n, int d) {
 if (d==0) exit(1);
 this->n=n; this->d=d;
}

frac operator+(frac p, frac q) {
 frac suma;
 int comun = mcm(p.d, q.d);
 suma.d = comun;
 suma.n = p.n*comun/p.d + q.n*comun/q.d;
 return suma;
}

frac operator-(frac p, frac q) {
 frac menosq(-q.n, q.d);
 return p+menosq;
}

frac operator-(frac p) {
 return frac(-p.n, p.d);
}

frac operator++(frac &p) { // ++obj
 p.n++;
 return p;
}

frac operator++(frac &p, int flag) {
 frac copia(p);
 p.n++;
 return copia;
}

/*frac operator+(int n, frac q) {
 frac suma(n*q.d+q.n, q.d);
 return suma;
}*/

int operator==(frac p, frac q) {
 if (p.n*q.d==p.d*q.n)
  return 1;
 return 0;
}

bool operator>(frac p, frac q) {
 return (p.n*q.d>p.d*q.n);
}

void frac::ver() { //signo en numerador
 if (d>0) cout << n << "/" << d << "\t";
 else cout << -n << "/" << -d << "\t";
}

int main() {
 frac a(3,2),b(3),T[]={frac(2,3),frac()};
 cout << "\n\t  a\tb\tc\td\n\t  ";
 a.ver();b.ver();T[0].ver();T[1].ver();//(*)
 a=a+a;       cout << "\na=a+a\t  ";//(*)
 b=a-T[0];    cout <<"\nb=a-c\t  "; //(*)
 a=-b;        cout <<"\na=-b\t  ";  //(*)
 T[1]=1+a-T[0]+b; cout <<"\nd=1+a-c+b ";//(*)
 a=++b;       cout <<"\na=++b\t  "; //(*)
 a=T[0]++;    cout <<"\na=c++\t  "; //(*)
 a=b+5;/*(1)*/cout << "\na=b+5\t  "; //(*)
 T[0]=5+b;    cout <<"\nc=5+b\t  "; //(*)
 a=a>b?-T[1]:b; cout <<"\na=a>b?-d:b"; //(*)
 if (a==T[0]) cout << "\na y c iguales\n";
 else         cout << "\na y c distintos\n";
 system("PAUSE"); return EXIT_SUCCESS;
}
```

Observaciones:

- La suma de un entero con una fracción **(int+frac)** no la podemos codificar con una función miembro ya que el operando izquierdo (el implícito) no es un objeto **frac** sino un **int**, por tanto hay que codificarlo con una función no miembro (**friend** o no, según queramos).

  ```
  frac operator+(int n, frac q);
  ```

- La suma de una fracción con un entero **(frac+int)** no la hemos codificado (hemos codificado **(frac+frac)** y **(int+frac)**). ¿Cómo es posible que funcione?...

  … porque **el compilador hace un casting implícito convirtiendo el entero en una fracción**, gracias a que **existe un constructor que admite un parámetro.**

  ```
  a=b+5; //el compilador lo convierte en a=b+frac(5);
  ```

  Si no hubiera un constructor con un parámetro o la suma de x/y + z tuviera que realizar una operación diferente a x/y + z/1 entonces tendríamos que sobrecarga la operación.

  `frac frac::operator+(int n);` o bien `frac operator+(frac p, int n);`

- Pero entonces… ¿el compilador no puede hacer un **casting implícito** con la suma de un entero con una fracción **(int+frac)**, al igual que hace con la suma de un **(frac+int)**?

  ```
  a=5+b; //el compilador lo convierte en a=frac(5)+b;
  ```

  - La respuesta es **sí**, por eso en la versión en la que hemos sobrecargado los operadores con funciones amigas (parte derecha), la sobrecarga de **(int+frac)** no está implementada (se indica como habría que hacerlo, pero está comentada entre /* y */)

  - En la sobrecarga de operadores con funciones miembros (parte izquierda), la sobrecarga de **(int+frac) SI** está implementada ya que es **obligatoria (al implementarse con una función miembro el operando izquierdo a la fuerza debe ser un objeto de la clase)**.

- Para **mejorar la eficiencia los parámetros** por valor **podríamos haberlo pasado mediante referencias constantes** (así no se tiene que hacer copia de los parámetros)

```
class frac {
 int n, d;
public:
 frac() { n=0; d=1; }
 frac(int n, int d=1) { set(n,d); }
 int getn() const { return n; }
 int getd() const { return d; }
 void set(int n, int d=1);
 frac operator+(const frac &q);
 frac operator-(const frac &q);
 frac operator-();
 frac operator++();
 frac operator++(int flag);
 int operator==(const frac &q);
 bool operator>(const frac &q);
//operator+(int, const frac &q)
 void ver();
};
```

```
class frac {
 int n, d;
public:
 frac() { n=0; d=1; }
 frac(int n, int d=1) { set(n,d); }
 int getn() const { return n; }
 int getd() const { return d; }
 void set(int n, int d=1);
 friend frac operator+(const frac &p, const frac &q);
 friend frac operator-(const frac &p, const frac &q);
 friend frac operator-(const frac &p);
 friend frac operator++(frac &p);
 friend frac operator++(frac &p, int flag);
 friend int operator==(const frac &p, const frac &q);
 friend bool operator>(const frac &p, const frac &q);
//friend frac operator+(int n, const frac &q);
 void ver();
};
```

- A pesar de no sobrecargar el operador = lo hemos podido usar porque el compilador (en caso de no existir sobrecarga) crea un operador de asignación por defecto que hace una copia binaria (sólo lo tendríamos que redefinir cuando la copia binaria no fuera lo correcto)

Observaciones: (continuación)

- Para **mejorar la eficiencia** aún más los operadores sobrecargados que modifican el propio objeto podíamos haberlo devuelto mediante referencias (devolvería el propio objeto en lugar de una copia del objeto). En nuestro ejemplo el único operador que devuelve el propio objeto modificado es el operator++ prefijo (++x)

```
class frac {
 int n, d;
public:
 ...
 frac& operator++();
 frac operator++(int flag);
};
```

```
class frac {
 int n, d;
public:
 ...
 friend frac& operator++(frac &p);
 friend frac operator++(frac &p, int flag);
};
```

No podemos aplicarlo al ++ postfijo ya que devuelve copia que es un objeto local

```
frac& frac::operator++() { //++obj
 n++;
 return *this;
}

frac frac::operator++(int flag){ //obj++
 frac copia(*this);
 n++;
 return copia;
}
```

```
frac& operator++(frac &p) { // ++obj
 p.n++;
 return p;
}

frac operator++(frac &p, int flag) {
 frac copia(p);
 p.n++;
 return copia;
}
```

- La sobrecarga de operadores se podía hacer sin funciones amigas (**friend**). La modificación de los atributos privados se haría a través de la interfaz pública de la clase.

```
class frac; //declaracion anticipada

frac operator+(const frac &p, const frac &q);
frac operator-(const frac &p, const frac &q);
frac operator-(const frac &p);
frac operator++(frac &p);
frac operator++(frac &p, int flag);
int operator==( const frac &p, const frac &q);
bool operator>( const frac &p, const frac &q);
//frac operator+(int n, const frac &q);

class frac {
 int n, d;
public:
 frac() { n=0; d=1; }
 frac(int n, int d=1) { set(n,d); }
 int getn() const { return n; }
 int getd() const { return d; }
 void set(int n, int d=1);
 void ver();
};

void frac::set(int n, int d) {
 if (d==0) exit(1);
 this->n=n; this->d=d;
}

frac operator+(const frac &p, const frac &q) {
 frac suma;
 int comun = mcm(p.getd(), q.getd());
 suma.set(p.getn()*comun/p.getd()+
          q.getn()*comun/q.getd(), comun);
 return suma;
}

frac operator-(const frac &p) {
 return frac(-p.getn(), p.getd());
}
```

```
frac operator-(const frac &p, const frac &q) {
 frac menosq(-q.getn(), q.getd());
 return p+menosq;
}

frac operator++(frac &p) { // ++obj
 p.set(p.getn()+1,p.getd());
 return p;
}

frac operator++(frac &p, int flag) {
 frac copia(p);
 p.set(p.getn()+1,p.getd());
 return copia;
}

/*frac operator+(int n, const frac &q) {
 frac suma(n*q.getd()+q.getn(), q.getd());
 return suma;
}*/

int operator==(const frac &p, const frac &q) {
 if (p.getn()*q.getd()==p.getd()*q.getn())
  return 1;
 return 0;
}

bool operator>(const frac &p, const frac &q) {
 return (p.getn()*q.getd()>p.getd()*q.getn());
}

void frac::ver() { //signo en numerador
 if (d>0) cout << n << "/" << d << "\t";
 else cout << -n << "/" << -d << "\t";
}
```

5. Sobrecarga de Métodos y Operadores

5.5 Sobrecarga de ciertos operadores

5.5.1 Sobrecarga del operador de asignación =

- Es el único operador que sobrecarga por defecto el compilador, en caso que el programador no lo sobrecargue
- Sólo se puede sobrecargar mediante una función miembro.
- Sintaxis: (permite cadenas de asignaciones y máxima eficiencia)

```
clase & clase::operator=(const clase& operador_derecho);
```

- void operator=(…) permite hacer a=b, pero no encadenar asignaciones a=b=c;
- clase operator=(…) { … return *this; } permite encadenar asignaciones a=b=c; Devuelve copia del objeto
  (x=y=z).metodo(); → x=y=z; copiax.metodo(); (x=y=z)++ → x=y=z; copiax++;
- clase& operator=(…) { … return *this; } permite encadenar asignaciones a=b=c; Devuelve objeto original
  (x=y=z).metodo(); → x=y=z; x.metodo(); (x=y=z)++ → x=y=z; x++;
- Por eficiencia retorno por referencia y parámetros referencias constantes → clase& operator=(const clase &);

- La sobrecarga por defecto que proporciona el compilador hace una copia binaria de los datos:

```
clase & clase::operator=(const clase& operador_derecho) {
    *this=operador_derecho; //llamada recursiva ERROR
    sentencias_que_hacen_una_copia_binaria;
    return *this; //con & devuelve el propio objeto implicito
}                 //sin & devuelve una copia del objeto implicito
```

  Si en la asignación no queremos hacer exactamente una copia binaria de los datos, entonces debemos sobrecargar el operador = y codificar lo que queremos hacer.

- El operador de asignación = lo podemos sobrecargar con otros objetos o tipos de datos:

```
clase & clase::operator=(tipo operador_derecho);
clase & clase::operator=(const otraClase &operador_derecho);
```

  El compilador no sobrecarga por defecto el operador = para estos casos, es el programador el que lo tiene que hacer explícitamente.

5.5.2 Sobrecarga del operador += (o del operador -=, operador/=, operador*=)

- Se puede sobrecargar mediante una función miembro (método) o una función no miembro.
- Sintaxis: (permite cadena de += y al ser por referencia máxima eficiencia al no hacer copia)

```
clase & clase::operator+=(const clase &operador_derecho);
clase & operator+=(clase &op_izq, const clase &op_derecho);
```

- El operador de asignación+ = lo podemos sobrecargar con otros objetos o tipos de datos:

```
clase & clase::operator+=(tipo operador_derecho);
clase & clase::operator+=(const otraClase &operador_derecho);
```

```
clase & operator+=(clase& op_izq, tipo operador_derecho);
clase & operator+=(clase& op_izq, const otraClase& op_derecho);
```

- El compilador no sobrecarga por defecto el operador +=. Para poder usarlo con objetos de nuevas clases creadas, el programador las tiene que sobrecargar explícitamente.

Sobrecarga Operador =

Ejemplo: sobrecargar el operador = y += para poder usarlo en una clase cadena, de forma que permita copiar una cadena en otra utilizando dicho operador

```
#include <iostream> //cin, cout
using namespace std;

class cadena {
 char *s;
 int nchar;
public:
 cadena(char*); // constructor general
 ~cadena() { delete [] s; }
 int getnchar() const { return nchar; }
 const char * gets() { return s; }
 void sets(char *);
 void ver();
 cadena& operator= (const cadena& cd);
 cadena& operator+= (const cadena& cd);
};

cadena::cadena(char* c) {
 nchar = strlen(c);
 s = new char[nchar+1];
 strcpy(s, c);
}

void cadena::sets(char* c) {
 delete [] s;
 nchar = strlen(c);
 s = new char[nchar+1];
 strcpy(s, c);
}

void cadena::ver() {
 cout << nchar << "," << s << endl;
}
```

```
cadena& cadena::operator=(const cadena& cd){
 if (this != &cd) { //por si es cd=cd
  nchar = cd.nchar; //para ello cd debe
  delete [] s;      //ser por referencia &
  s = new char[nchar + 1];
  strcpy(s, cd.s);
 }
 return *this;
}

cadena& cadena::operator+=(const cadena& cd){
 char *aux=s;
 nchar += cd.nchar;
 s = new char[nchar + 1];
 strcpy(s, aux);
 strcat(s, cd.s);
 delete [] aux;
 return *this;
}

int main() {
 cadena c1("toro");
 cadena c2("cobra");
 cout << "c1:"; c1.ver();
 cout << "c2:"; c2.ver();
 c2=c1;
 cout << "c2:"; c2.ver();
 c1.sets("sentado");
 cout << "c1:"; c1.ver();
 cout << "c2:"; c2.ver();
 c2+=c1;
 cout << "c1:"; c1.ver();
 cout << "c2:"; c2.ver();
 system("PAUSE"); return EXIT_SUCCESS;
}
```

Pantalla:

```
c1:4,toro
c2:5,cobra
c2:4,toro
c1:7,sentado
c2:4,toro
c1:7,sentado
c2:11,torosentado
```

Si no sobrecargamos el operador =, el compilador al realizar la asignación

c2=c1

haría una copia binaria del objeto c1 en c2, con lo que ambos objetos apuntarían a la misma dirección de memoria (el valor de s es el mismo en ambos), en lugar de producir una copia.

Como consecuencia de ello, ambos compartirían la misma cadena, de forma que los cambios provocados en uno de ellos afectarían al otro, al apuntar ambos a la misma zona de memoria.

Si no sobrecargmos el operador += el compilador daría un error al encontrar la instrucción

c2+=c1

ya que dicho operador por defecto sólo se puede utilizar con los tipos básico de C++ (int, float, char) y para poderlo usar con las nuevas clases creadas es necesario sobrecargarlo.

Pruebe a eliminar la sobrecarga del operador = y vea lo que ocurre
A continuación haga lo mismo eliminando la sobrecarga del operador +=

Nota: la sentencia if en la sobrecarga de operador de asignación evita la asignación de un objeto a sí mismo (cd=cd) ya que en ese caso, al liberar la memoria del 1er operando, el 2º (que es el mismo) la pierde también, con lo que habría un error. Para ello **el operando derecho hay que pasarlo por referencia** para pasar el propio objeto, de forma que detecte que ambos objetos **this** (dir objeto izquierdo) y **&cd** (dir del objeto derecho) son el mismo. Si se pasa por valor o copia, &cd devuelve la dir de la copia que no es la misma que la del objeto original.

5. Sobrecarga de Métodos y Operadores

5.5.3 Sobrecarga del operador [ ]

- Se puede sobrecargar mediante una función miembro o una función no miembro.
- Sintaxis: (al devolver una referencia permite modificar lo que se devuelve)

```
tipo1 & clase::operator[](tipo2 i); // miembro
tipo1 & operator[](const clase &op_izq, tipo2 i); //no miembro
```

siendo **tipo1** el tipo indexado y **tipo2** el índice (puede ser cualquier tipo, lo normal es **int**)

y donde & es opcional, dependiendo de si nos interesa o no que devuelva una referencia a un objeto **tipo1** o que devuelva un objeto **tipo1**.

- Si devuelve una referencia, entonces el operador [ ] **podrá utilizarse** a la derecha, pero también a la izquierda de una asignación, con el peligro que eso conlleva si la clase tiene alguna restricción respecto a los valores de sus atributos (véase ejemplo apartado *4.4)*
- Al devolver una referencia permite modificar lo que se devuelve

```
clase x; tipo1 y,z; int i;
//tipo2 es int
...
x[2]=y; //x.operator[](2)=y
z=x[i];
```

```
clase x; tipo1 y,z; char *cad;
//tipo2 es char *
...
x["ana"]=y; //x.operator[]("ana")=y
z=x[cad];
```

Si no queremos permitir que operator[ ] pueda aparecer en la parte izquierda de una asignación, haremos que no devuelve una referencia o que la referencia sea **const**.

Ejemplo: sobrecargar el operador [ ] para poder usarlo en una clase cadena, de forma que permita acceder y/modificar los caracteres indexándolos a partir del 1

```
#include <iostream> //cin, cout
using namespace std;
class cadena {
 char *s;
 int nchar;
public:
 cadena(char*); // constructor general
 ~cadena() { delete [] s; }
 int getnchar() const { return nchar; }
 char& operator[](int i);
 const char * gets() { return s; }
};

cadena::cadena(char* c) {
 nchar = strlen(c);
 s = new char[nchar+1];
 strcpy(s, c);
}

char& cadena::operator[](int i) {
 if (i <= nchar && i>=1)
  return s[i-1];
 else {
  cout << "ERROR\n";
  system("PAUSE");
  exit(1); //aborta el programa
 }
}
```

```
int main() {
 cadena cad("casa");
 char h;
 cout << cad.gets() << " tiene " <<
         cad.getnchar() << " letras\n";
 cad[3] = 'n'; //cad.operator[](3)='n';
 cout << "Introduzca una letra: ";
 cin >> cad[2];//cin>>cad.operator[](2);
 cout << cad.gets() << endl;
 cout << "la 1ª letra es " << cad[1] << endl;
 h = cad[4]; //h=cad.operator[](4);
 cout << "la 4ª letra es " << h << endl;
 cad[9] = 'k'; //ERROR 9 excede cad
 system("PAUSE"); return EXIT_SUCCESS;
}
```

Pantalla:

casa tiene 4 letras
Introduzca una letra: e
cena
la primera letra es c
la cuarta letra es a
ERROR

5. Sobrecarga de Métodos y Operadores

5.5.4 Sobrecarga de los operadores de inserción << y de extracción>>

- Los operadores de **inserción (<<)** y **extracción (>>)** en los flujos (streams) de E/S **sólo se pueden sobrecargar mediante una función no miembro**.
- Sintaxis: (al devolver una referencia permite modificar lo que se devuelve)

```
friend std::istream& operator>>(std::istream& s, clase &o);
friend std::ostream& operator<<(std::ostream& s, const clase &o);
```

donde **istream** es el *flujo* de entrada y **ostream** el *stream* de salida y lo gris es opcional

Estos **flujos** funcionan como cintas transportadoras que entran (>>) o salen (<<) del programa. Se recibe una **referencia al flujo** como 1er argumento, se añade o se retira de él **la variable que se desee**, y se devuelve siempre como valor de retorno **una referencia al flujo** (stream) modificado.

- istream y ostream se devuelven por referencia para permitir cadenas de << y de >>

o1 << o2 << o3; o1 >> o2 >> o3;

- ambas se suelen declarar amigas de la clase para la que se sobrecarga, para poder acceder a su parte privada, aunque no es obligatorio
- en operator<< el objeto de la clase para la que se sobrecarga se suede pasar por **referencia** (por motivos de eficiencia) **constante** (por seguridad), aunque no es obligatorio.

El C++no tiene instrucciones de E/S, pero si tiene librerías que manejan **streams** (flujos). La librería <**iostream**> proporciona una serie de clases que permiten la E/S por teclado (**istream**) como por pantalla (**ostream**).

El C++ tiene predefinidos los objetos streams **cin** y **cout**:

- **cin** es un objeto de la clase **istream** que se encarga de la entrada por teclado
- **cout** es un objeto **ostream** que maneja la salida por pantalla.
- << y >> son operadores que están sobrecargados para las clases de streams

Ejemplo: sobrecarga de >> con funcion amiga y << con funcion no amiga

```
#include <iostream> //cin, cout

using namespace std;

class frac {
 int n, d;
public:
 frac() { n=0; d=1; }
 frac(int x, int y) { n=x; d=y; }
 int getn() const { return n; }
 int getd() const { return d; }
 void set(int n, int d=1);
 friend istream& operator>>(istream& s,
                            frac &p);
};

istream& operator>>(istream& s, frac &p) {
 cout << "numerador: ";
 s >> p.n; //s actúa a modo de cin
 do {
  cout << "denominador: ";
  s >> p.d; //s actúa a modo de cin
 } while (p.d==0);
 return s;
}
```

```
ostream& operator<<(ostream &s,
                    const frac &p) {
 if (p.getd()>0)
   s << p.getn() << "/" << p.getd();
 else //s actúa a modo de cout
   s << -p.getn() << "/" << -p.getd();
 return s;
}

int main() {
 frac a(3,2),b;
 cout << a << " , " << b << endl;
 cout << "Introduce 2 fracciones \n";
 cin >> a >> b;
 cout << a << " , " << b << endl;
 system("PAUSE"); return EXIT_SUCCESS;
}
```

Facil de programar:

- en operator>>, s actúa como cin
- en operator<<, s actúa como cout

Si operator>> no fuera amiga:

```
istream& operator>>
(istream& s, fracc &p) {
  int n, d;
  cout << "numerador: ";
  s >> n;
  do {
    cout << "denominador: ";
    s >> d;
  } while (d==0);
  p.set(n,d);
  return s;
}
```

Pantalla:

```
3/2 , 0/1
Introduce 2 fracciones
numerador: 2
denominador: 3
numerador: 2
denominador: 0
denominador: 5
2/3 , 2/5
```

5. Sobrecarga de Métodos y Operadores

5.5.5 Operadores de conversión de tipos vs constructores de conversión

- Cuando definimos clases (son nuevos tipos) el compilador no define ninguna forma de conversión (automática o no) con otras clases o tipos estándar, a menos que el programador especifique de forma explícita cómo deben ocurrir dichas conversiones.
- **Estas conversiones pueder ser realizadas de 2 formas,** (en un sentido u otro) **mediante:**
  - **contructores de conversión:** permiten convertir el objeto o el tipo estándar pasado como parámetro, en un objeto de la clase que lo sobrecarga *(sentido desde)*
  - **operadores de conversión:** permiten convertir un objeto de la clase que lo sobrecarga, a un objeto de otra clase o a un tipo estándar *(sentido hacia*)
- **Una clase puede tener tantos constructores y operadores de conversión como desee**.

5.5.5.1 Sobrecarga de constructores de conversión

- **Un constructor de conversión** es un constructor que tiene un solo argumento (objeto de otra clase o un tipo estándar) desde el cual se convierte a un objeto de la clase.
- Sintaxis: (la 1ª convierte otraClase en clase, la 2ª convierte tipo_estandar en clase)

```
clase::clase(const otraClase &o); //conversión desde otra clase
clase::clase(tipo_estandar o);    //conversión desde un tipo
```

donde **const &** es opcional (**&** por motivos de eficiencia, **const** por motivos de seguridad)

5.5.5.2 Sobrecarga de operadores de conversión de tipos

- **Los operadores de conversión (cast)** convierten un objeto de la clase donde está definido en un objeto de otra clase o en un tipo.
- **Sólo se pueden sobrecargar mediante una función miembro sin argumentos ni valor de retorno, ni siquiera void** (esto no significa que no devuelvan nada, sino que no hay que indicarlo al ser el tipo de retorno el propio tipo al que se quiere convertir).
- Sintaxis: (la 1ª convierte clase en otraClase, la 2ª convierte clase en tipo_estandar)

```
clase::operator otraClase();     //convierte clase en otraClase
clase::operator tipo_estandar(); //conversión de clase a tipo
```

Ejemplo:

```
#include <iostream>
using namespace std;
class tiempo {
 int hora, minuto;
public:
 tiempo(int h, int m);
 tiempo(int m);
 operator int() {return hora*60+minuto;}
 friend ostream& operator<<(ostream &s,
};                      const tiempo &t);
tiempo::tiempo(int h, int m) {
 hora=h+m/60;
 minuto=m%60;
}
tiempo::tiempo(int m) {
 hora=0; minuto=m;
 while(minuto >= 60) {
  minuto -= 60;
  hora++;
 }
}
```

Pantalla:

```
127 , 1:30
0:30 , 90
120
2:0 , 120
```

```
ostream& operator<<(ostream &s, const tiempo &t) {
  s << t.hora << ":" << t.minuto;
  return s; //s actúa a modo de cout
}

int main() {
  tiempo a(2,7),b(90),c(0,0);
  int m;
  cout << (int)a << " , " << b << endl;
//a+=15; //ERROR += no esta sobrecargado
  a=15; //a=tiempo(15)
  a=a+15; //a=tiempo((int)a+15)
  cout << a << " , " << (int)b << endl;
  m = a; //m=(int)a;
  m+= b; //m+=(int)b;
  cout << m << endl;
  m = c = a+b;
//c=tiempo((int)a+(int)b);  m=(int)c;
  cout << c << " , " << m << endl;
  system("PAUSE"); return EXIT_SUCCESS;
}
```

5. Sobrecarga de Métodos y Operadores

5.5.5 Operadores de conversión de tipos vs constructores de conversión

Ejemplo 2: conversión de tipos y clases vs constructores de conversión

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

class reloj {
 int horas, minutos;
public:
 reloj(int h, int m);
 reloj operator+(int m);
 int geth() const { return horas; }
 int getm() const { return minutos; }
};

reloj::reloj(int h, int m) {
 minutos=m%60;
 horas=(h+m/60)%24;
}

reloj reloj::operator+(int m) {
 reloj suma(horas,minutos+m);
 return suma;
}

class tiempo {
 int hora, minuto;
public:
 tiempo(int h, int m);
 tiempo(int m);
 tiempo(const reloj &r);
 operator int();
 operator float();
 operator reloj();
 tiempo operator-(tiempo t);
 friend ostream& operator<<(ostream &s,
                    const tiempo &t);
};

tiempo::tiempo(int h, int m) {
 hora=h+m/60;
 minuto=m%60;
}

tiempo::tiempo(int m) {
 hora=m/60;
 minuto=m%60;
}

tiempo::tiempo(const reloj &r) {
 hora=r.geth();
 minuto=r.getm();
}

tiempo::operator int() {
 return hora*60+minuto;
}

tiempo::operator float() {
 return hora+(float)minuto/60;
}

tiempo::operator reloj() {
 reloj aux(hora,minuto);
 return aux;
}

tiempo tiempo::operator-(tiempo t) {
   tiempo aux(hora-t.hora, minuto-t.minuto);
   return aux;
}

ostream& operator<<(ostream &s, const tiempo &t) {
   s << t.hora << ":" << t.minuto;
   return s; //s actúa a modo de cout
}

int main() {
 reloj x(25,62), y(0,0);
 tiempo a(25,62), b(90), c(x);
 int minus;
 y=b;          //y=(reloj)b;
 y=y+60*24+1; //y=y.operator+(1441);
 cout << x.geth() << "h" << x.getm() << "m\n";
 cout << y.geth() << "h" << y.getm() << "m\n";
 cout << a << "(" << (int)a << ")," << (float)b;
 a=15;         //a=tiempo(15);
//a=a+15;      //ERROR ambiguo, puede ser
               //a=tiempo((int)a+15)
               //a=tiempo((float)a+15)
  a=(int)a+15;
//a+=15;       //ERROR += no sobrecargado
  cout << a << ", " << b << endl;
  minus = a;  //minus=(int)a
//minus=a+b;   //ERROR ambiguo, puede ser
               //minus=(int)a+(int)b
               //minus=(float)a+(int)b
               //minus=(int)a+(float)b
               //minus=(float)a+(float)b
  minus = (int)a + (int)b;
//minus+= b;   //ERROR ambiguo, puede ser
               //minus+=(int)b
               //minus+=(float)b
  cout << minus << endl;
  minus = c = a - b;
//c = a.operator-(b); minus = (int)c;
  cout << minus << ", " << c << endl;
  c = minus = a - b;
//minus=(int)a.operator-(b); c=tiempo(minus)
  cout << minus << ", " << c << endl;
  c = x+2;
//c=tiempo(x.operator+(2));
  cout << (int)c << ", " << c << endl;
  system("PAUSE"); return EXIT_SUCCESS;
}
```

Pantalla:

```
2h2m
1h31m
26:2(1562), 1.5
0:30, 1:30
120
-60, -1:0
-60, -1:0
124. 2:4
```

Pruebe a eliminar la sobrecarga del operador – en la clase tiempo y vea lo que ocurre
Modifica el main para que la resta a – b se pueda realizar sin necesidad de sobrecargar el –
Solución: minus = c = (int)a – (int) b; //asi se elimina la ambigüedad (se restan 2 enteros y casting de int a tiempo)

6. Constructores y Destructores

- **Una clase debe tener al menos un constructor (puede tener varios) y un único destructor**

Constructor	Destructor
▪ **Es una función miembro que tiene el mismo nombre que la clase en la que está definido y no tiene tipo devuelto (ni void).**	▪ **Idem, pero el nombre viene precedido por el carácter ~ (Alt+126) y no tiene tipo devuelto (ni siquiera void).**
▪ Es llamado automáticamente cada vez que se crea un objeto de esa clase (si el objeto es global: al empezar el programa, si es local: al alcanzar su declaración dentro del bloque donde está definido, si es dinámico, al crearlo con **new**).	▪ Es llamado automáticamente cuando el objeto va a dejar de existir (si el objeto es global: al terminar el programa, si es local: al salir del bloque donde está definido, si es un objeto dinámico **new**, al destruirlo con **delete**).
▪ Puede tener argumentos (**con o sin valor por defecto**) y puede estar sobrecargado.	▪ No tiene argumentos y no puede estar sobrecargado (es siempre único).
▪ Se usa para inicializar los valores de las variables miembros de la clase.	▪ Se usa para liberar memoria dinámica que se haya podido reservar durante la vida del objeto, para terminar asuntos pendientes, cerrar ficheros, etc.

- Si el programador no define ningún constructor para una clase, el compilador proporciona el siguiente **constructor de oficio** (default constructor) sin argumentos (que no hace nada):

```
clase::clase() { //constructor de oficio, sin argumentos
  //no hace nada (lo crea el compilador si no definimos ninguno)
}
```

- Lo mismo ocurre con el destructor. Si el programador no define ningún destructor para una clase, el compilador proporciona el siguiente **destructor de oficio** (no hace nada):

```
clase::~clase() { //destructor de oficio
  //no hace nada (lo crea el compilador si no definimos ninguno)
}
```

6.1 Construcción, destrucción y tiempo de vida

- Para **objetos locales**, el constructor se invoca cada vez que se crea el objeto, y su destructor al terminar la función o bloque en el que está.
- Para **objetos globales**, el contructor se invoca una sola vez, al inicio del programa, y el destructor al terminar el programa
- Para **objetos dinámicos**, el constructor se invoca cuando se crea con **new** y el destructor cuando se destruye con **delete**.
- **Los destructores son llamados en orden inverso a los constructores.**

```
Orden de creación y destrucción de los objetos:

1 Creación de x, y       7 Destrucción de p
2 Creación de a          8 Creación de b
3 Creación de b          9 Destrucción de b
4 Destrucción de b      10 Destrucción de d, c
5 Creación de p         11 Destrucción de a
6 Creación de c, d      12 Destrucción de y, x
```

```
class frac {
 . . .
public:
 frac(); // constructor
 ~frac(); // destructor
}

frac x,y; //objeto global

void proceso() {
 frac b; // objeto local
 . . .
}

int main() {
 frac a,*p; //objeto local
 proceso();
 p = new frac();
 frac c,d; // objeto local
 delete p;
 proceso();
 . . .
}
```

6. Constructores y Destructores

6.2 Inicializadores

- **Permiten inicializar variables miembros fuera del cuerpo del constructor.**
- **Sólo lo pueden usar los constructores**, no lo pueden usar otras funciones miembros.
- **Sólo puede aparecer en la definición de la función miembro**, no en su declaración.
- Los inicializadores se introducen separados por comas, tras el carácter dos puntos (:), justo antes de abrir las llaves del cuerpo del constructor
- Sintaxis:

```
clase::clase([parámetros]):atributo1(valor1),atributo2(valor2)… {
 [sentencias;]            //constructor con inicializadores
}
```

donde **atributoN** son atributos de la clase y **valorN** los valores que queremos asignarles

lo anterior es equivalente a:

```
clase::clase([parámetros]) {
 atributo1 = valor1;      //constructor sin inicializadores
 atributo2 = valor2;
 [sentencias;]
}
```

- Los inicializadores son más eficientes que las sentencias de asignación.
- **Son necesarios** cuando en una clase definimos variables miembros (atributos) que son referencias (**&**) o constantes (**const**), ya que deben inicializarse, al no poder modificar sus valores una vez construidos (no podemos usar sentencias de asignación).

Ejemplo: clase que tiene atributos constantes y atributos referencia

```
#include <iostream>
using namespace std;
class cla {
 int a,b;
 int &ref;
 const float fijo;
public:
 cla():a(0),b(0),ref(a),fijo(9.8) { }
 cla(int x, int y, float v)
   :a(x),b(y),ref(a),fijo(v) { }
 void set(int x, int y) { a=x; b=y; }
};
int main() {
 cla a, b(2, 3, 0.5);
 b.set(6,3);
 system("PAUSE");return EXIT_SUCCESS;
}
```

Lo anterior es equivalente a:

```
class cla {
 int a,b;
 int &ref;
 const float fijo;
public:
 cla():ref(a),fijo(9.8) { a=0; b=0; }
 cla(int x, int y, float v);
 void set(int x, int y) { a=x; b=y; }
};

cla::cla(int x, int y, float v):ref(a), fijo(v) {
 a=x;
 b=y;
}
```

- Los atributos **ref** y **fijo** hay que inicializarlos obligatoriamente mediante inicializadores: no puede asignarse en el cuerpo de la función ni inicializarlas al declararlas.

```
class cla {
 int a,b, &ref=a; //ERROR
 const float fijo=9.8; //ERROR
public:
 cla();
 cla(int x, int y, float v);
 void set(int x, int y): a(x), b(y) { }
};   //ERROR set no es un constructor
```

```
class cla {
 int a,b, &ref;
 const float fijo;
public:
 cla() { a=0; b=0; ref=a; fijo=9.8; } //ERROR
 cla(int x, int y, float v) {
   a=x; b=y; ref=a; fijo=v; } //ERROR
};
```

6. Constructores y Destructores

6.3 Constructor por defecto

- **Constructor por defecto es aquel que no tiene parámetros** o que si los tiene, todos sus argumentos tienen asignado un valor por defecto**, es decir, aquel constructor que puede ser llamado sin tener que pasarle ningún argumento.**
- **Es necesario si se quiere poder crear un objeto de la forma:**

```
clase objeto; //se invoca el constructor sin argumentos
```

y también cuando se quiere crear un vector de objetos:

```
clase array[10]; //se invoca el constructor sin argumentos para
                 //cada objeto del array
```

ya que no es posible pasar argumentos propios para cada uno de los objetos del array para su inicialización (debe haber un constructor sin argumentos, que es el que se invoca)

6.4 Constructor de oficio

- **Constructor por defecto sin argumentos que crea automáticamente el compilador en caso que el programador no defina ningún constructor para la clase.**

```
clase::clase() { //constructor de oficio, sin argumentos
  //no hace nada (lo crea el compilador si no definimos ninguno)
}
```

- **Si el programador define algún** constructor para la clase (con o sin argumentos), **el compilador no crea el constructor de oficio.**

  Si el usuario define una clase con un/os constructor/es con argumentos y ningún constructor por defecto entonces no podrá crear un vector de objetos (ni un objeto sin indicar el valor de sus atributos).

Ejemplo:

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

class hour {
 int hora, mint;
public:
 hour(int h, int m):hora(h+m/60),mint(m%60){ }
 hour(int m):hora(m/60), mint(m%60){ }
 hour():hora(0) { mint=0; }
 friend ostream& operator<<(ostream &s,
                      const hour &t);
};

ostream& operator<<(ostream &s, const hour &t) {
  s << t.hora << ":" << t.mint;
  return s; //s actúa a modo de cout
}

int main() {
  hour a(2,7);//hour a=hour(2,7);
  hour b(90); //hour b=hour(90);
  hour c;     //hour c=hour();
  hour t[3];  //hour t[3]={hour(),hour(),hour()};
  for(int i=0,j=30; i<3; i++,j+=20)
    t[i]=hour(i+1,j);
  cout << a << "," << b << "," << c << endl;
  cout << t[0] << "," << t[1] << "," << t[2];
  system("PAUSE"); return EXIT_SUCCESS;
}
```

Pantalla:

2:7 , 1:30 , 0:0
1:30 , 2:50 , 4:10

Si no tuviera el constructor por defecto

```
class hour {
 int hora, mint;
public:
 hour(int h, int m);
 hour(int m);
 hour(); //contructor por defecto
};
```

El compilador no lo genera, al tener la clase otros constructores, por lo que:

```
int main() {
 hour a(2,7), b(90);
 hour c;     //ERROR
 hour t[3];  //ERROR
 ...
}
```

Para que no diera error habría que poner:

```
int main() {
 hour a(2,7), b(90);
 hour c(0);
 hour t[3]={hour(0),hour(0),hour(0)};
 ...
}
```

6.5 Constructor de copia

- Cuando se crea un objeto inicializándolo a partir de otro objeto de la misma clase se llama a un constructor especial llamado **constructor de copia** (**copy constructor**).
- El **constructor de copia** (que crea un nuevo objeto a partir de uno existente de su misma clase) **tiene un único argumento**: una referencia constante a un objeto de su clase.
- Sintaxis:

```
clase::clase(const clase &objeto); //constructor de copia
```

- **Si el programador no define un constructor de copia** (aunque haya definido otros constructores con o sin argumentos)**, el compilador crea un constructor de copia de oficio**, el cual realiza una **copia bit a bit** de las variables miembro del objeto original pasado como parámetro, al objeto creado que lo invoca.

```
clase::clase(const clase &objeto) {
 sentencias_que_hacen_copia_binaria;  //constructor de copia
}
```

- **Si el comportamiento del constructor de copia de oficio no es el deseado el programador deberá definir uno propio.**
- Cuando no hay punteros implicados, el constructor de copia de oficio funciona bien. Sin embargo, cuando se utilizan punteros, el constructor de copia de oficio al copiar todos los atributos del objeto que se pasa por referencia al objeto actual binariamente hará que tanto el atributo puntero del objeto copiador como del objeto copia apunten al mismo sitio.
- **El constructor de copia se invoca en los siguientes casos:**
  - ☐ **cuando se declara un objeto inicializándolo a partir de otro objeto de la misma clase.**
  - ☐ **cuando a una función se le pasan objetos por valor.**
  - ☐ **cuando una función devuelve un objeto como valor de retorno.**

  En esos casos hay que crear copias del objeto y para ello se usa el constructor de copia.

Ejemplo 1:

```
clase x, y;  //constructor por defecto (se crea los objetos x y)
clase a = x; //constructor de copia (se crea el objeto a)
clase b(x);  //constructor de copia (se crea el objeto b)
y = x;       //operador de asignación (se modifica y)
```

En la 2ª sentencia se crea un objeto ***a*** inicializando sus variables miembros con los mismos valores que tienen en ***x***. La 3ª sentencia es una forma sintáctica equivalente a la 2ª: ***b*** se inicializa con los valores de ***x***. La 4ª sentencia es una asignación (no se crea un nuevo objeto sino que se trabaja con objetos ya creados)

Las sentencias 2ª y 3ª anteriores se ejecutan aunque no se haya definido en la clase ningún constructor de copia, ya que en ese caso, el compilador crea uno de oficio.

Ejemplo 2:

```
class clase {
  ...
public:
 clase proc(clase a, clase &b);
};
clase clase::proc(clase a, clase &b) {
 clase aux; //se invoca constructor por defecto
  ...
 return aux;//se invoca constructor de copia
}           //para devolver una copia de aux
int main() {
 clase x,y,c,z;
 c = x.proc(y,z);//invoca constructor copia
}                //ya que y se pasa por valor
```

En el ***main()*** al invocar el objeto ***x*** el método ***proc()*** pasa el objeto ***y*** por valor y ***z*** por referencia, por lo que para copiar el objeto ***y*** en el parámetro ***a*** del método se invoca el contructor de copia.

Al terminar el método ***proc()*** debe devolver una copia del objeto local ***aux***, por lo que de nuevo se invoca el constructor de copia para asignar al objeto ***c*** del ***main()*** una copia de ***aux***.

6.6 Necesidad de escribir un constructor de copia y un operador =

- Como norma general, cuando una clase trabaje con memoria dinámica por tener una variable miembro tipo puntero, se deberá definir las siguientes funciones miembros:
  - **Un constructor** (que reserve memoria e inicialice el resto de variables miembros).
    Si pretendemos crear arrays de objetos la clase debe tener un constructor por defecto.
  - **Un constructor de copia** (que evite que al pasar un objeto por copia o cree un objeto a partir de otro, sus variables mimbros tipo puntero apunten a la misma zona de memoria).
  - **Un destructor** (que libere la memoria dinámica).
  - **Un operador de asignación = sobrecargado** (que evite que la variable miembro tipo puntero del objeto asignado apunte al mismo lugar que la del objeto del que se copia).
- Como los constructores, destructores y operador de asignación = sobrecargado no se heredan, una clase derivada de una clase base debe también implementar estos métodos.

Ejemplo: Elimine solo operator=. Elimine solo constructor copia. Elimine ambos. Vea lo que ocurre

```
#include <iostream> //cin, cout
using namespace std;

class cad {
 char* s;
 int n;
public:
 cad() { s=new char[1]; strcpy(s,""); n=0; }
 cad(const char*); // c. general
 cad(const cad&); // c. de copia
 ~cad() { delete [] s; }
 void setcad(const char*);
 cad& operator=(const cad &);
 cad operator+(const cad &) const;
 bool operator==(cad) const;
 friend ostream& operator<<(ostream&, const cad&);
 void ver() const { cout << n << "," << s; }
};

cad::cad(const char* c) {
 n = strlen(c);
 s = new char[n + 1];
 strcpy(s, c);
}

cad::cad(const cad& cd) {
 n = cd.n;
 s = new char[n+1];
 strcpy(s, cd.s);
}

void cad::setcad(const char* c) {
 n = strlen(c);
 delete [] s;
 s = new char[n+1];
 strcpy(s, c);
}

cad& cad::operator=(const cad &cd) {
 if(this != &cd) {
  n = cd.n;
  delete [] s;
  s=new char[n+1];
  strcpy(s, cd.s);
 }
 return *this;
}
```

```
cad cad::operator+(const cad& b) const {
 cad c;
 c.n = n + b.n;
 c.s = new char[c.n + 1];
 strcpy(c.s, s);
 strcat(c.s, b.s);
 return c;
}

bool cad::operator==(cad b) const {
 if (n != b.n) return false;
 return (strcmp(s,b.s)==0);
}

ostream& operator<<(ostream& o, const cad& c) {
 o << c.s;
 return o;
}

int main() {
 cad c1;  c1 = "Blanca";
 cad c2("Paloma");
 cad c3 = c2;
 c2.setcad("Avestruz");
 c1.ver(); c2.ver(); c3.ver(); cout<<endl;
 cad c4(c2);
 c2.ver(); c4.ver(); cout << endl;
 if (c2 == c4)
  cout << c2 << " == " << c4 << endl;
 (c3 = c2) = c1; //c3=c2; c3=c1;
 c3 = c2 = c1;   //c2=c1; c3=c2;
 c1.setcad("Gris");
 cout << c1 <<","<< c2 <<","<< c3 << endl;
 c2 = c2;
 c2 = c1;
 c2 = c1 + "Rojo";
 c3 = c1 + c1;
 cad T[3];
 T[0]=c1; T[1]=c2; T[2]=c3;
 for(int i=0; i<3; i++)
  cout << "T[" << i << "] objeto "
       << (long)&T[i] <<":" << T[i] << endl;
 system("PAUSE"); return EXIT_SUCCESS;
}
```

Pantalla:

```
6,Blanca 8,Avestruz 6,Paloma
8,Avestruz 8,Avestruz
Avestruz == Avestruz
Gris,Blanca,Blanca
T[0] obj 2686672: Gris
T[1] obj 2686680: GrisRojo
T[2] obj 2686688: GrisGris
```

Para entender lo que ocurre vamos a añadir mensajes a las funciones miembros:

```
#include <iostream> //cin, cout
using namespace std;

class cad {
 int n;
 char* s;
public:
 cad();            //constructor por defecto
 cad(const char*); //constructor general
 cad(const cad&);  //constructor de copia
 ~cad();           //destructor
 void setcad(const char*);
 cad& operator=(const cad &);
 cad operator+(const cad &) const;
 bool operator==(cad) const;
 friend ostream& operator<<(ostream&, const cad&);
 void ver() const;
};

cad::cad() {
 s = new char[1]; strcpy(s, ""); n = 0;
 cout << " Constructor por defecto: crea "
  << (long)this << " " << (long)s << endl;
}

cad::cad(const char* c) {
 n=strlen(c); s=new char[n+1]; strcpy(s,c);
 cout << " C. general: crea " << (long)this
  << " " << (long)s << " desde " << c <<"\n";
}

cad::cad(const cad& cd) {
 n=cd.n; s=new char[n+1]; strcpy(s,cd.s);
 cout << " C. copia: crea " << (long)this
  <<" "<<(long)s<<" desde "<<(long)&cd<< endl;
}

cad::~cad() { // destructor
 delete [] s;
 cout << " Destructor de " << (long)this << endl;
}

void cad::setcad(const char* c) {
 n = strlen(c);  delete [] s;
 s = new char[n + 1]; strcpy(s, c);
 cout << " setcad: " << (long)this << " "
      << (long)s << endl;
}

cad& cad::operator=(const cad &cd) {
 if(this != &cd) {
  n = cd.n;  delete [] s;
  s=new char[n+1];  strcpy(s, cd.s);
  cout << " Operador asignacion: " <<
  (long)this << " = " << (long)&cd << endl;
 }
 return *this;
}
```

```
cad cad::operator+(const cad& b) const {
 cout << " ";
 cout << (long)this << ".operator+("
      << (long)&b << ")\n";
 cad c;
 c.n = n + b.n;
 c.s = new char[c.n + 1];
 strcpy(c.s, s);
 strcat(c.s, b.s);
 return c;
}

bool cad::operator==(cad b) const {
 cout << " ";
 cout << (long)this << ".operator==("
      << (long)&b << ")\n";
 if (n != b.n) return false;
 return (strcmp(s,b.s)==0);
}

ostream& operator<<(ostream& o, const cad& c) {
 o << c.s;
 return o;
}

void cad::ver() const {
 cout << (long)this << " n= " << n
      << " s="<< (long)s << s << endl;
}

int main() {
{ //no es un error, es intencionado
 cad c1;  c1 = "Blanca";
 cad c2("Paloma");
 cad c3 = c2;
 c2.setcad("Avestruz");
 c1.ver(); c2.ver(); c3.ver();
 cad c4(c2);
 c2.ver(); c4.ver();
 if (c2 == c4)
  cout << c2 << " == " << c4 << endl;
 (c3 = c2) = c1; //c3=c2; c3=c1;
 c3 = c2 = c1;   //c2=c1; c3=c2;
 c1.setcad("Gris");
 cout << c1 <<","<< c2 <<","<< c3 << endl;
 c2 = c2;
 c2 = c1;
 c2 = c1 + "Rojo";
 c3 = c1 + c1;
 cad T[3];
 T[0]=c1; T[1]=c2; T[2]=c3;
 for(int i=0; i<3; i++)
  cout << "T[" << i << "] objeto "
       << (long)&T[i] <<":"<< T[i] << endl;
} //para ver mensajes del destructor
 system("PAUSE"); return EXIT_SUCCESS;
}
```

Elimina la sobrecarga del operador de asignación = y vea lo que ocurre.

Elimina el constructor copia y vea lo que ocurre.

Elimina ambos (operador de asignación y contructor copia) y vea lo que ocurre.

En la sobrecarga del operador + pasa el parámetro por valor en vez de por referencia constante y vea lo que ocurre: `cad cad::operator+(cad b) const;`

Ejercicio: Ejecuta el siguiente programa e intenta comprender a qué son debidos los errores que se producen al ejecutarlo. Corrigelos

Sobrecarga de <<

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.

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 {
  static int numserie;
  int *identificador;
  int x,y;
public:
  Punto(int nx, int ny) { x=nx; y=ny; identificador=new int(numserie++); }
  ~Punto() { delete identificador; }
  int getx() { return x; }
  int gety() { return y; }
  void setxy(int x, int y) { this->x=x; Punto::y=y; }
  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;
  }
  friend ostream& operator<<( ostream &salida, const Punto p );
};

int Punto::numserie=0;

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

Punto::operator char*() { //en vez de operator<<
  char salida[30];
  sprintf( salida, "%i:(%i,%i)",*identificador, x, y );
  return strdup( salida );
}

int main(int argc, char *argv[]) {
  Punto p1(10,15), p2(10,15), p3(0,0);
  cout << p1 << " -- " << p2 << " -- " << p3 << endl;
  cout << p1 << " -- " << p2 << " -- " << p3 << endl;
  cout << p1 << ((p1==p2)?" == ":" != ") << p2 << "\n" ;
  p2.setxy(p2.getx(), p2.gety()+1);
  cout << p1 << ((p1==p2)?" == ":" != ") << p2 << "\n" ;
  p3=p2=p1=p1;
  cout << p1 << " -- " << p2 << " -- " << p3 << endl;
  cout << p1 << ((p1==p3)?" == ":" != ") << p3 << "\n" ;

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

Pantalla:

```
0:(10,15) -- 1:(10,15) -- 2:(0,0)
0:(10,15) -- 5383304:(10,15) -- 5383408:(0,0)
5383424:(10,15) == 5375160:(10,15)
5374344:(10,15) != 5375160:(10,16)
5374344:(10,15) -- 5375160:(10,15) -- 5383408:(10,15)
La ultima línea no se ejecuta… el programa aborta…
```

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*** en vez de << (no recomendado: memoria no se libera)

Solucion:
Los errores son debidos a que hay memoria dinámica (la variable miembro identificador es de tipo puntero) y no se ha definido un constructor de copia, por lo que el que genera de oficio el compilador hace una copia binaria de los datos → cuando se pasa por valor la copia apunta a la misma zona de memoria y al destruirse la copia y ejecutarse el destructor se libera la memoria del original.
En la sobrecarga de los operadores ==, = y << se pasa por copia, por lo que al ejecutar cualquiera de dichos operadores se producen errores.

Sol1: definir constructor de copia (recomendable)

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

class Punto {
  static int numserie;
  int *identificador;
  int x,y;
public:
  Punto(int nx, int ny) { x=nx; y=ny;
                          identificador=new int(numserie++); }
  Punto(const Punto &p) { x=p.x; y=p.y;
                          identificador=new int(*(p.identificador)); }
  ~Punto() { delete identificador; }
  int getx() { return x; }
  int gety() { return y; }
  void setxy(int x, int y) { this->x=x; Punto::y=y; }
  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;
  }
  friend ostream& operator<<(ostream &s, const Punto p );
};

int Punto::numserie=0;

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

int main(int argc, char *argv[]) {
  Punto p1(10,15), p2(10,15), p3(0,0);
  cout << p1 << " -- " << p2 << " -- " << p3 << endl;
  cout << p1 << " -- " << p2 << " -- " << p3 << endl;
  cout << p1 << ((p1==p2)?" == ":" != ") << p2 << "\n" ;
  p2.setxy(p2.getx(), p2.gety()+1);
  cout << p1 << ((p1==p2)?" == ":" != ") << p2 << "\n" ;
  p3=p2=p1=p1;
  cout << p1 << " -- " << p2 << " -- " << p3 << endl;
  cout << p1 << ((p1==p3)?" == ":" != ") << p3 << "\n" ;

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

Sol2: pasar todo por referencia

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

class Punto {
  static int numserie;
  int *identificador;
  int x,y;
public:
  Punto(int nx, int ny) { x=nx; y=ny;
                          identificador=new int(numserie++); }
  ~Punto() { delete identificador; }
  int getx() { return x; }
  int gety() { return y; }
  void setxy(int x, int y) { this->x=x; Punto::y=y; }
  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;
  }
  friend ostream& operator<<(ostream &s, const Punto &p );
};

int Punto::numserie=0;

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

int main(int argc, char *argv[]) {
  Punto p1(10,15), p2(10,15), p3(0,0);
  cout << p1 << " -- " << p2 << " -- " << p3 << endl;
  cout << p1 << " -- " << p2 << " -- " << p3 << endl;
  cout << p1 << ((p1==p2)?" == ":" != ") << p2 << "\n" ;
  p2.setxy(p2.getx(), p2.gety()+1);
  cout << p1 << ((p1==p2)?" == ":" != ") << p2 << "\n" ;
  p3=p2=p1=p1;
  cout << p1 << " -- " << p2 << " -- " << p3 << endl;
  cout << p1 << ((p1==p3)?" == ":" != ") << p3 << "\n" ;

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

Pantalla:

```
0:(10,15) -- 1:(10,15) -- 2:(0,0)
0:(10,15) -- 1:(10,15) -- 2:(0,0)
0:(10,15) == 1:(10,15)
0:(10,15) != 1:(10,16)
0:(10,15) -- 1:(10,15) -- 2:(10,15)
0:(10,15) == 2:(10,15)
Presione una tecla para continuar . . .
```

Solucion:

Los errores son debidos a que hay memoria dinámica (la variable miembro identificador es de tipo puntero) y no se ha definido un constructor de copia, por lo que el que genera de oficio el compilador hace una copia binaria de los datos → cuando se pasa por valor la copia apunta a la misma zona de memoria y al destruirse la copia y ejecutarse el destructor se libera la memoria del original.
En la sobrecarga de los operadores ==, = y << se pasa por copia, por lo que al ejecutar cualquiera de dichos operadores se producen errores.

Sol1: definir constructor de copia (recomendable)

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

class Punto {
  static int numserie;
  int *identificador;
  int x,y;
public:
  Punto(int nx, int ny) { x=nx; y=ny;
                          identificador=new int(numserie++); }
  Punto(const Punto &p) { x=p.x; y=p.y;
                          identificador=new int(*(p.identificador)); }
  ~Punto() { delete identificador; }
  int getx() { return x; }
  int gety() { return y; }
  void setxy(int x, int y) { this->x=x; Punto::y=y; }
  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;
  }
  friend ostream& operator<<(ostream &s, const Punto p );
};

int Punto::numserie=0;

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

int main(int argc, char *argv[]) {
  Punto p1(10,15), p2(10,15), p3(0,0);
  cout << p1 << " -- " << p2 << " -- " << p3 << endl;
  cout << p1 << " -- " << p2 << " -- " << p3 << endl;
  cout << p1 << ((p1==p2)?" == ":" != ") << p2 << "\n" ;
  p2.setxy(p2.getx(), p2.gety()+1);
  cout << p1 << ((p1==p2)?" == ":" != ") << p2 << "\n" ;
  p3=p2=p1=p1;
  cout << p1 << " -- " << p2 << " -- " << p3 << endl;
  cout << p1 << ((p1==p3)?" == ":" != ") << p3 << "\n" ;

  system("PAUSE
  return EXIT_SU
}
```

Pantalla:

```
0:(10,15) -- 1:(10,15) -- 2:(0,0)
0:(10,15) -- 1:(10,15) -- 2:(0,0)
0:(10,15) == 1:(10,15)
0:(10,15) != 1:(10,16)
0:(10,15) -- 1:(10,15) -- 2:(10,15)
0:(10,15) == 2:(10,15)
Presione una tecla para continuar . . .
```

Sol: constructor de copia y paso por referencia const (mejor)

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

class Punto {
  static int numserie;
  int *identificador;
  int x,y;
public:
  Punto(int nx, int ny) { x=nx; y=ny;
                          identificador=new int(numserie++); }
  Punto(const Punto &p) { x=p.x; y=p.y;
                          identificador=new int(*(p.identificador)); }
  ~Punto() { delete identificador; }
  int getx() { return x; }
  int gety() { return y; }
  void setxy(int x, int y) { this->x=x; Punto::y=y; }
  int operator==( const Punto &p ) { return (x==p.x)&&(y==p.y); }
  Punto& operator=(const Punto &p ) {
    if (this != &p) { //por si es p=p
      x=p.x, y=p.y;  /*no se sobrescribe la serie*/
    }
    return *this;
  }
  friend ostream& operator<<(ostream &s, const Punto &p );
};

int Punto::numserie=0;

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

int main(int argc, char *argv[]) {
  Punto p1(10,15), p2(10,15), p3(0,0);
  cout << p1 << " -- " << p2 << " -- " << p3 << endl;
  cout << p1 << " -- " << p2 << " -- " << p3 << endl;
  cout << p1 << ((p1==p2)?" == ":" != ") << p2 << "\n" ;
  p2.setxy(p2.getx(), p2.gety()+1);
  cout << p1 << ((p1==p2)?" == ":" != ") << p2 << "\n" ;
  p3=p2=p1=p1;
  cout << p1 << " -- " << p2 << " -- " << p3 << endl;
  cout << p1 << ((p1==p3)?" == ":" != ") << p3 << "\n" ;

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