> ***Regla***
>
> Para prevenir a los usuarios de la clase de crear un objeto sin parámetros, se puede: 1) omitir el constructor por defecto; o bien, 2) hacer el constructor privado.

EJEMPLO 2.6. La clase `EquipoSonido` se define con tres constructores; un constructor por defecto, otro con un argumento de tipo cadena y el tercero con tres argumentos.

```
class EquipoSonido
{
private:
    int potencia, voltios;
    string marca;
public:
    EquipoSonido()    // constructor por defecto
    {
        marca = "Sin marca"; potencia = voltios = 0;
    }
    EquipoSonido(string mt)
    {
        marca = mt; potencia = voltios = 0;
    }
    EquipoSonido(string mt, int p, int v)
    {
        marca = mt;
        potencia = p;
        voltios = v;
    }
};
```


La instanciación de un objeto `EquipoSonido` puede hacerse llamando a cualquier constructor. A continuación se crean tres objetos:

```
EquipoSonido rt;    // constructor por defecto
EquipoSonido ft("POLASIT");
EquipoSonido gt("PARTOLA", 35, 220);
```

2.3.3. Array de objetos

Los objetos se pueden estructurar como un array. Cuando se crea un array de objetos éstos se inicializan llamando al constructor sin argumentos. Por consiguiente, siempre que se prevea organizar los objetos en un array, la clase debe tener un constructor que pueda llamarse sin parámetros.

> ***Precaución***
>
> Tenga cuidado con la escritura de una clase con sólo un constructor con argumentos. Si se omite un constructor que pueda llamarse sin argumento no será posible crear un array de objetos.

EJEMPLO 2.7. Se crean arrays de objetos de tipo `Complejo` y `EquipoSonido`.

```
Complejo zz[10];  // crea 10 objetos, cada uno se inicializa a 0,1
EquipoSonido* pq; // declaración de un puntero
int n;
cout << "Número de equipos: "; cin >> n;
pq = new EquipoSonido[n];
```

2.3.4. Constructor de copia

Este tipo de constructor se activa cuando al crear un objeto se inicializa con otro objeto de la misma clase. Por ejemplo:


```
Complejo z1(1, -3);      // z1 se inicializa con el constructor
Complejo z2 = z1;        /* z2 se inicializa con z1, actúa el
                            constructor de copia                */
```

También se llama al constructor de copia cuando se pasa un objeto por valor a una función, o bien cuando la función devuelve un objeto. Por ejemplo:

```
extern Complejo resultado(Complejo d);
```

para llamar a esta función se pasa un parámetro de tipo `Complejo`, un objeto. En esta transferencia se llama al constructor de copia. Lo mismo ocurre cuando la misma función devuelve (`return`) el resultado, un objeto de la clase `Complejo`.

El constructor de copia es una función miembro de la clase, su prototipo:

```
NombreClase(const NombreClase& origen);
```

el argumento `origen` es el objeto copiado, `z1` en el primer ejemplo. La definición del constructor de copia para la clase `Complejo`:


```
Complejo(const Complejo& origen)
{
    x = origen.x;
    y = origen.y;
}
```

No es siempre necesario definir el constructor de copia, por defecto se realiza una copia miembro a miembro. Sin embargo, cuando la clase tenga atributos (punteros) que representen memoria dinámica, *buffer dinámico*, sí debe definirse, para realizar una *copia segura*, reservando memoria y copiando en esa memoria o *buffer* los elementos.

2.3.5. Asignación de objetos

El operador de asignación, =, se puede utilizar con objetos, de igual forma que con datos de tipo simple. Por ejemplo:

```
Racional r1(1, 3);
Racional r2(2, 5);
r2 = r1;          // r2 toma los datos del objeto r1
```

Por defecto, la asignación se realiza miembro a miembro. El numerado de `r2` toma el valor del numerador de `r1` y el denominador de `r2` el valor del denominador de `r1`.

C++ permite cambiar la forma de asignar objetos de una clase. Para ello se implementa una función miembro de la clase especial (se denomina sobrecarga del operador `=`) con este prototipo:

```
nombreClase& operator = (const nombreClase&);
```

A continuación se implementa esta función en la clase `Persona`:

```
class Persona
{
private:
     int edad;
     string nom, apell;
     string dni;
public:
     // sobrecarga del operador de asignación
     Persona& operator = (const Persona& p)
     {
          if (this == &p) return *this; // es el mismo objeto
          edad = p.edad;
          nom = p.nom;
          apell = p.apell;
          dni = p.dni;
          return *this;
     }
```


En esta definición se especifica que no se tome acción si el objeto que se asigna es él mismo: `if (this == &p)`.

En la mayoría de las clases no es necesario definir el operador de asignación ya que, por defecto, se realiza una asignación miembro a miembro. Sin embargo, cuando la clase tenga miembros de tipo puntero (memoria dinámica, *buffer dinámico*) sí debe definirse, para que la asignación sea *segura*, reservando memoria y asignando a esa memoria o *buffer* los elementos.

EJEMPLO 2.8. La clase `Libro` se define con un array dinámico de páginas (clase `Pagina`). El constructor establece el nombre del autor , el número de páginas y crea el arr ay. Además, se escribe el constructor de copia y la sobrecarga del oper ador de asignación.

```
// archivo Libro.h

class Libro
{
private:
     int numPags, inx;
     string autor;
     Pagina* pag;
     // ...
```

```
public:
    Libro(string a, int n);      // constructor
    Libro(const Libro& cl);      // constructor de copia
    Libro& operator = (const Libro& al);  // operador de asignación
    // ... funciones miembro
};

// archivo Libro.cpp

Libro::Libro(string a, int n)
{
    autor = a;
    inx = 0;
    numPags = n;
    pag = new Pagina[numPags];
}
// constructor de copia
Libro::Libro(const Libro& cl)
{
    autor = cl.a;
    inx = cl.inx;
    numPags = cl.numPags;
    pag = new Pagina[numPags];  // copia segura
    for (int i = 0; i < inx; i++)
       pag[i] = cl.pag[i];
}
// operador de asignación
Libro& operator = (const Libro& al)
{
    if (this == &p) return *this;
    autor = al.a;
    inx = al.inx;
    numPags = al.numPags;
    pag = new Pagina[numPags];  // copia segura
    for (int i = 0; i < inx; i++)
       pag[i] = al.pag[i];
    return *this;
}
```


2.3.6. Destructor

Un objeto se libera, se destruye, cuando se sale del ámbito de definición. También, un objeto creado dinámicamente, con el operador `new`, se libera al aplicar el operador `delete` al puntero que lo referencia. Por ejemplo:

```
Punto* p = new Punto(1,2);
if (...)
{
    Punto p1(2, 1);
    Complejo z1(8, -9);
}   // los objetos p1 y z1 se destruyen
delete p;
```

El destructor tiene el mismo nombre que clase, precedido de una tilde (~). Cuando se define, no se puede especificar un valor de retorno, ni argumentos:

```
~NombreClase()
{
     ;
}
```

El destructor es necesario implementarlo cuando el objeto contenga memoria reserva dinámicamente.

EJEMPLO 2.9. Se declara la clase `Equipo` con dos atributos de tipo puntero, un constructor con valores por defecto y el destructor.

El constructor define una array de `n` objetos `Jugador` con el operador `new`. El destructor libera la memoria reservada.

```
class Equipo
{
private:
     Jugador* jg;
     int numJug;
     int actual;
public:
     Equipo(int n = 12)
     {
          jg = new Jugador[n];
          numJug = n;  actual = 0;
     }
     ~Equipo()     // destructor
     {
          if (jg != 0) delete [] jg;
     }
};
```

2.4. AUTOREFERENCIA DEL OBJETO: `THIS`

`this` es un puntero al objeto que envía un *mensaje*, o simplemente, un puntero al objeto que llama a una función miembro de la clase (ésta no debe ser `static`). Este puntero no se define, internamente se define:

```
const NombreClase* this;
```

por consiguiente, no puede modificarse. Las variables y funciones de las clase están referenciados, implícitamente, por `this`. Por ejemplo, la siguiente clase:

```
class Triangulo
{
private:
     double base, altura;
```