Programación Orientada a Objetos en C++

Jorge M. Finochietto

UNC-CONICET

2010

Outline

General

Programación Orientada a Objetos (POO)

Es un *paradigma* (modelo de trabajo) de programación que agrupa los datos y los procedimientos para manejarlos en una única entidad: *el objeto*.

Objeto

Un objeto es una unidad que engloba en sí mismo variables y funciones necesarios para el tratamiento de esos datos.

Cada programa es un objeto, que a su vez está formado de objetos que se relacionan entre ellos.

Definiciones

Método

Función perteneciente a un determinado objeto.

Atributo

Variable perteneciente a un determinado objeto.

Los objetos se comunican e interrelacionan entre si a través del acceso a sus atributos y del llamado a sus métodos.

Definiciones

Clase

Una clase se puede considerar como un patrón para construir objetos.

Interfaz

Es la parte del objeto que es visible para el resto de los objetos. Es decir, el conjunto de métodos y atributos que dispone un objeto para comunicarse con él.

- Un objeto es sólo una instancia (un ejemplar) de una clase determinada.
- Las clases tienen partes públicas y partes privadas. Generalmente, llamaremos a la parte pública de una clase su interfaz.

Definiciones

Herencia

Capacidad de crear nuevas clases basándose en clases base previamente definidas, de las que se aprovechan ciertos datos y métodos, se desechan otros y se añaden nuevos.

Jerarquía

Orden de subordinación de un sistema clases.

Polimorfismo

Propiedad según la cual un mismo objeto puede considerarse como perteneciente a distintas clases.

- La herencia siempre se produce en un mismo sentido, creando estructuras de clases en forma de árbol.
- Es posible retroceder hasta una o varias clases base.

Outline

Declaración

Sintáxis

```
class <identificador de clase> [<lista de clases base>] {
<lista de miembros>
};
```

- La lista de clases base se usa para derivar clases.
- La lista de miembros comprende una lista de funciones y datos.

Ejemplo

```
class punto {
  private:
   // Atributos de la clase "punto"
   int a, b;
  public:
   // Métodos de la clase "punto"
   void Lee(int &a2, int &b2);
   void Guarda(int a2, int b2);
};

// Sigue definición de métodos de la clase "punto"
```

Acceso

Sintáxis

```
class <identificador de clase> {
  public:
   <lista de miembros>
  private:
   <lista de miembros>
  protected:
   <lista de miembros>
};
```

- Acceso privado (private)
  - Sólo son accesibles por los propios miembros de la clase, pero no desde funciones externas o desde funciones de clases derivadas.
  - Acceso por defecto.
- Acceso público (public)
  - Cualquier miembro público de una clase es accesible desde cualquier parte donde sea accesible el propio objeto.
- Acceso protegido (protected)
  - Con respecto a las funciones externas, es equivalente al acceso privado, pero con respecto a las clases derivadas se comporta como público.

Métodos (Funciones Miembro)

- La lista de métodos puede ser
  - simplemente una declaración de prototipos,
  - o pueden ser también definiciones.
- Cuando se definen fuera se debe usar el operador de ámbito "::"
- Las funciones definidas de este modo serán tratadas como *inline* (se inserta el código cada vez que son llamadas)

Ejemplo

```
class punto {
  .....
  public:
   // Métodos de la clase "punto"
   void Lee(int &a2, int &b2);
   void Guarda(int a2, int b2) {
      a = a2;
      b = b2;
   }
};

void punto::Lee(int &a2, int &b2) {
   a2 = a;
   b2 = b;
}
```

Ejemplo

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

class punto {
  private:
   // Atributos de la clase "punto"
   int a, b;
  public:
   // Métodos de la clase "punto"
   void Lee(int &a2, int &b2);
   void Guarda(int a2, int b2) {
      a = a2;
      b = b2;
   }
};
void punto::Lee(int &a2, int &b2) {
   a2 = a;
   b2 = b;
}

int main() {
  punto par1;
  int x, y;
  par1.Guarda(12, 32);
  par1.Lee(x, y);
  cout << "Valor de par1.a: " << x << endl;
  cout << "Valor de par1.b: " << y << endl;
  return 0;
}
```

El puntero this

- Para cada objeto declarado de una clase se mantiene una copia de sus datos, pero todos comparten la misma copia de las funciones de esa clase.
  - Esto ahorra memoria y hace que los programas ejecutables sean más compactos, pero plantea un problema.
- Cada función de una clase puede hacer referencia a los datos de un objeto, modificarlos o leerlos, usando el puntero especial llamado `this` que apunta al mismo objeto.

Ejemplo

```
   void punto::Guarda(int a2, int b2) {
   this->a = a2;
   this->b = b2;
}
```

Ejemplo

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

class clase {
  public:
   clase() {}
   void EresTu(clase& c) {
      if(&c == this) cout << "Sí, soy yo." << endl;
      else cout << "No, no soy yo." << endl;
   }
};

int main() {
   clase c1, c2;

   c1.EresTu(c2);
   c1.EresTu(c1);

   return 0;
}
```

Outline

Constructores

- Son métodos especiales que sirven para inicializar un objeto de una determinada clase al mismo tiempo que se declara.
- Son especiales por varios motivos:
  - Tienen el mismo nombre que la clase a la que pertenecen.
  - No tienen tipo de retorno, y por lo tanto no retornan ningún valor.
  - No pueden ser heredados.
  - Deben ser públicos, no tendría ningún sentido declarar un constructor como privado, ya que siempre se usan desde el exterior de la clase, ni tampoco como protegido, ya que no puede ser heredado.

Sintáxis

```
class <identificador de clase> {
  public:
     <identificador de clase>(<lista de parámetros>) [: <lista de constructores>] {
         <código del constructor>
     }
     ...
}
```

Constructores

Ejemplo

```
class punto {
  .....
  public:
   // Métodos de la clase "punto"
   // Constructor
   punto(int a2, int b2);
   void Lee(int &a2, int &b2);
   void Guarda(int a2, int b2) {
      a = a2;
      b = b2;
   }
};

punto::punto(int a2, int b2) {
   a = a2;
   b = b2;
}
```

- Si no definimos un contructor el compilador creará uno por defecto, sin parámetros, que no hará absolutamente nada.
- Si definimos un constructor que requiere argumentos, es obligatorio suministrarlos.
- Será llamado siempre que se declare un objeto de esa clase.

Ejemplo

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

class punto {
  .....
  public:
   // Métodos de la clase "punto"
   // Constructor
   punto(int a2, int b2);
   .....
};

punto::punto(int a2, int b2) {
   a = a2;
   b = b2;
}
......

int main() {
  punto par1(12, 32);
  int x, y;

  par1.Lee(x, y);
  cout << "Valor de par1.a: " << x << endl;
  cout << "Valor de par1.b: " << y << endl;
  return 0;
}
```

Inicialización de objetos

- Se pueden inicializar los datos miembros de los objetos en los constructores invocando los constructores de ellos.
- En C++ incluso las variables de tipos básicos como int, char o float son objetos.
- Cada inicializador consiste en el nombre de la variable miembro a inicializar, seguida de la expresión que se usará para inicializarla entre paréntesis.
- Los inicializadores se añadirán a continuación de los parámetros del constructor, antes del cuerpo del constructor y separado del paréntesis por dos puntos ":"

Ejemplo

```
punto::punto(int a2, int b2) : a(a2), b(b2) {}
```

- Es preferible usar la inicialización siempre que sea posible en lugar de asignaciones.

Sobrecarga de constructores

- Los constructores son funciones, también pueden definirse varios constructores para cada clase (sobrecarga).
- Como en las funciones, no pueden declararse varios constructores con el mismo número y el mismo tipo de argumentos

Ejemplo

```
class punto {
  public:
   // Constructores
   punto(int a2, int b2) : a(a2), b(b2) {}
   punto() : a(0), b(0) {}
   // Funciones miembro de la clase "punto"
   void Lee(int &a2, int &b2);
   void Guarda(int a2, int b2);
  private:
   // Datos miembro de la clase "punto"
   int a, b;
};
```

Argumentos por defecto

- Pueden asignarse valores por defecto a los argumentos del constructor.
- De este modo reducimos el número de constructores necesarios.

Ejemplo

```
class punto {
  public:
   // Constructor
   punto(int a2=0, int b2=0) : a(a2), b(b2) {}
   // Funciones miembro de la clase "punto"
   void Lee(int &a2, int &b2);
   void Guarda(int a2, int b2);
  private:
   // Datos miembro de la clase "punto"
   int a, b;
};
```

Asignación de objetos

- La asignación está permitida entre objetos existentes.
- El compilador crea un operador de asignación por defecto.
- Éste copia los valores de todos los atributos de un objeto al otro.

Ejemplo

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

int main() {
   punto par1(12, 32), par2;
   int x, y;

   par2 = par1;
   par2.Lee(x, y);
   cout << "Valor de par2.a: " << x << endl;
   cout << "Valor de par2.b: " << y << endl;

   return 0;
}
```

Constructor copia

- Crea un objeto que no existe a partir de otro objeto existente.
- Sólo tienen un argumento, que es una referencia a un objeto de su misma clase.
- El compilador crea un constructor copia por defecto que copia los valores de todos los atributos de un objeto al otro.

Uso del cosntructor copia

Caso explícito

```
int main() {
   punto par1(12, 32)
   punto par2(par1);
}
```

Caso implícito con mismo objeto

```
int main() {
   punto par1(12, 32)
   // Se usa porque el objeto par2 se inicializa al mismo tiempo que se declara.
   punto par2=par1;
}
```

Caso implícito con objeto diferente

```
int main() {
   punto par1(12, 32)
   // El compilador intenta crear el objeto par2 usando el constructor copia sobre el objeto 14.
   // Como 14 no es un objeto de la clase punto, el compilador usa el constructor de punto
   // con el valor 14, y después usa el constructor copia.
   punto par2=14;
}
```

Copia superficial vs. Copia profuna

- Tanto la asignación como el constructor copia por defecto hacen una *copia superficial*.
  - Sólo se copian los miembros pero no la memoria a que estos apuntan (para el caso de punteros)
  - Por lo tanto, al copiarse punteros ambos objetos terminan apuntando a la misma área de memoria.
- La *copia profunda* implica además hacer una copia de los bloques de memoria hacia donde apunten punetros miembros del objeto.
- Para implementar la copia profunda, pueden definirse por el usuario tanto el operador de asignación como el constructor copia.

Destructores

- Son métodos especiales que sirven para eliminar un objeto de una determinada clase.
- Realizan procesos necesarios cuando un objeto termina su ámbito temporal:
  - liberan la memoria dinámica utilizada por dicho objeto,
  - liberan recursos usados, como ficheros, dispositivos, etc.
- Tienen algunas características especiales:
  - Tienen el mismo nombre que la clase a la que pertenecen, pero tienen el símbolo $\sim$ delante
  - No tienen tipo de retorno, y por lo tanto no retornan ningún valor.
  - No tienen parámetros.
  - No pueden ser heredados.
  - Deben ser públicos.
  - No pueden ser sobrecargados, lo cual es lógico, puesto que no tienen valor de retorno ni parámetros, no hay posibilidad de sobrecarga.

Destructores

- Cuando se define un destructor para una clase, éste es llamado automáticamente cuando se abandona el ámbito en el que fue definido.
- Esto es así salvo cuando el objeto fue creado dinámicamente con el operador `new`, ya que en ese caso, cuando es necesario eliminarlo, hay que hacerlo explícitamente usando el operador `delete`.

Ejemplo: Clase

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

class cadena {
  public:
   cadena();         // Constructor por defecto
   cadena(const char *c); // Constructor desde cadena c
   cadena(int n);    // Constructor de cadena de n caracteres
   cadena(const cadena &);   // Constructor copia
   ~cadena();        // Destructor

   void Asignar(const char *dest);
   char *Leer(char *c);
  private:
   char *cad;        // Puntero a char: cadena de caracteres
};
...
```

Ejemplo: Definición de constructores y destructores

```
...
cadena::cadena() : cad(NULL) {}

cadena::cadena(const char *c) {
   cad = new char[strlen(c)+1];// Reserva memoria para cadena
   strcpy(cad, c);             // Almacena la cadena
}

cadena::cadena(int n) {
   cad = new char[n+1]; // Reserva memoria para n caracteres
   cad[0] = 0;          // Cadena vacía
}

cadena::cadena(const cadena &Cad) {
   // Reservamos memoria para la nueva y la almacenamos
   cad = new char[strlen(Cad.cad)+1];
   // Reserva memoria para cadena
   strcpy(cad, Cad.cad);             // Almacena la cadena
}

cadena::~cadena() {
   delete[] cad;        // Libera la memoria reservada a cad
}
...
```

Ejemplo: Funciones

```
...
...
void cadena::Asignar(const char *dest) {
   // Eliminamos la cadena actual:
   delete[] cad;
   // Reservamos memoria para la nueva y la almacenamos
   cad = new char[strlen(dest)+1];
   // Reserva memoria para la cadena
   strcpy(cad, dest);              // Almacena la cadena
}

char *cadena::Leer(char *c) {
   strcpy(c, cad);
   return c;
}
...
```

Ejemplo: Principal

```
...

int main() {
   cadena Cadena1("Cadena de prueba");
   cadena Cadena2(Cadena1);   // Cadena2 es copia de Cadena1
   cadena *Cadena3;           // Cadena3 es un puntero
   char c[256];

   // Modificamos Cadena1:
   Cadena1.Asignar("Otra cadena diferente");
   // Creamos Cadena3:
   Cadena3 = new cadena("Cadena de prueba nº 3");

   // Ver resultados
   cout << "Cadena 1: " << Cadena1.Leer(c) << endl;
   cout << "Cadena 2: " << Cadena2.Leer(c) << endl;
   cout << "Cadena 3: " << Cadena3->Leer(c) << endl;

   delete Cadena3;  // Destruir Cadena3.
   // Cadena1 y Cadena2 se destruyen automáticamente

   return 0;
}
```

Outline

Concepto de amistad (friend)

- Los miembros privados de una clase no son accesibles para funciones y clases exteriores a dicha clase.
- En ciertas ocasiones, necesitaremos tener acceso a determinados miembros de un objeto de una clase desde otros objetos de clases diferentes, pero sin perder ese encapsulamiento para el resto del programa, es decir, manteniendo esos miembros como privados.
- El modificador friend puede aplicarse a clases o funciones para inhibir el sistema de protección.
- Las relaciones de amistad entre clases suponen lo siguiente
  - La amistad no puede transferirse, si A es amigo de B, y B es amigo de C, esto no implica que A sea amigo de C.
  - La amistad no puede heredarse. Si A es amigo de B, y C es una clase derivada de B, A no es amigo de C.
  - La amistad no es simétrica. Si A es amigo de B, B no tiene por qué ser amigo de A.

Funciones amigas externas

Ejemplo

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

class A {
  public:
    A(int i=0) : a(i) {}
    void Ver() { cout << a << endl; }
  private:
    int a;
    friend void Ver(A); // "Ver" es amiga de la clase A
};

void Ver(A Xa) {
   // La función Ver puede acceder a miembros privados
   // de la clase A, ya que ha sido declarada "amiga" de A
   cout << Xa.a << endl;
}

int main() {
   A Na(10);

   Ver(Na);  // Ver el valor de Na.a
   Na.Ver(); // Equivalente a la anterior

   return 0;
}
```

Funciones amigas con otras clases

Ejemplo

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

class A; // Declaración previa (forward)

class B {
   public:
    B(int i=0) : b(i) {}
    void Ver() { cout << b << endl; }
    bool EsMayor(A Xa);  // Compara b con a
   private:
    int b;
};

class A {
   public:
    A(int i=0) : a(i) {}
    void Ver() { cout << a << endl; }
   private:
    // Función amiga tiene acceso
    // a miembros privados de la clase A
    friend bool B::EsMayor(A Xa);
    int a;
};

bool B::EsMayor(A Xa) {
   return b > Xa.a;
}
...
```

Funciones amigas con otras clases

Ejemplo

```
...
int main() {
   A Na(10);
   B Nb(12);

   Na.Ver();
   Nb.Ver();
   if(Nb.EsMayor(Na)) cout << "Nb es mayor que Na" << endl;
   else cout << "Nb no es mayor que Na" << endl;

   return 0;
}
```

Clases amigas

Ejemplo

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

/* Clase para elemento de lista enlazada */
class Elemento {
  public:
   Elemento(int t);                              /* Constructor */
   int Tipo() { return tipo;}                   /* Obtener tipo */
  private:                                            /* Datos: */
   int tipo;                                        /* Tipo */
   Elemento *sig;                          /* Siguiente elemento */
  friend class Lista;                       /* Amistad con lista */
};
```

Clases amigas

Ejemplo

```
...
/* Clase para lista enlazada de números*/
class Lista {
  public:
   Lista() : Cabeza(NULL) {}                    /* Constructor */
                                                /* Lista vacía */
   ~Lista() { LiberarLista(); }                  /* Destructor */
   void Nuevo(int tipo);                     /* Insertar figura */
   Elemento *Primero()              /* Obtener primer elemento */
   { return Cabeza; }
   /* Obtener el siguiente elemento a p */
   Elemento *Siguiente(Elemento *p) {
      if(p) return p->sig; else return p;};
   /* Si p no es NULL */
   /* Averiguar si la lista está vacía */
   bool EstaVacio() { return Cabeza == NULL;}

  private:
   Elemento *Cabeza;             /* Puntero al primer elemento */
   void LiberarLista(); /* Función privada para borrar lista */
};
```

Clases amigas

Ejemplo

```
/* Constructor */
Elemento::Elemento(int t) : tipo(t), sig(NULL) {}
  /* Asignar datos desde lista de parámetros */

/* Añadir nuevo elemento al principio de la lista */
void Lista::Nuevo(int tipo) {
   Elemento *p;

   p = new Elemento(tipo);  /* Nuevo elemento */
   p->sig = Cabeza;
   Cabeza = p;
}

/* Borra todos los elementos de la lista */
void Lista::LiberarLista() {
   Elemento *p;

   while(Cabeza) {
      p = Cabeza;
      Cabeza = p->sig;
      delete p;
   }
}
```

Clases amigas

Ejemplo

```
...
int main() {
   Lista miLista;
   Elemento *e;

   // Insertamos varios valores en la lista
   miLista.Nuevo(4);
   miLista.Nuevo(2);
   miLista.Nuevo(1);

   // Y los mostramos en pantalla:
   e = miLista.Primero();
   while(e) {
      cout << e->Tipo() << " ,";
      e = miLista.Siguiente(e);
   }
   cout << endl;

   return 0;
}
```

Outline

Funciones en línea (inline)

- Cuando las definiciones de funciones miembro son muy pequeñas, ser interesante declararlas como `inline`.
- En estos casos el código generado para la función cuando el programa se compila, se inserta en el punto donde se invoca a la función, en lugar de hacerlo en otro lugar y hacer una llamada.
  - El código de estas funciones se ejecuta más rápidamente,
  - peroel programa ejecutable final puede ser mucho más grande.
- Hay dos maneras de declarar una función como inline.
  - En forma implícita
  - En forma explícita

Funciones en línea implícitas

Ejemplo con función inline implícita

```
class Ejemplo {
  public:
   Ejemplo(int a = 0) : A(a) {}

  private:
   int A;
};
```

Ejemplo sin función inline

```
class Ejemplo {
  public:
   Ejemplo(int a = 0);

  private:
   int A;
};

Ejemplo::Ejemplo(int a) : A(a) {}
```

Funciones en línea explícitas

- Para hacerlo en forma explícita se debe usar la palabra reservada `inline`

Ejemplo

```
class Ejemplo {
  public:
   Ejemplo(int a = 0);

  private:
   int A;
};

inline Ejemplo::Ejemplo(int a) : A(a) {}
```

Funciones constantes

- Cuando una función miembro no deba modificar el valor de ningún dato de la clase, podemos y debemos declararla como constante.
- Esto evitará que la función intente modificar los datos del objeto.
- Proporciona ciertos mecanismos necesarios para mantener la protección de los datos.
- Se utiliza la palabra reservada `const`

Funciones constantes

Ejemplo

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

class Ejemplo2 {
  public:
   Ejemplo2(int a = 0) : A(a) {}
   void Modifica(int a) { A = a; }
   int Lee() const { return A; }

  private:
   int A;
};

int main() {
   Ejemplo2 X(6);

   cout << X.Lee() << endl;
   X.Modifica(2);
   cout << X.Lee() << endl;

   return 0;
}
```

- Si intentamos modificar algo de la clase, el compilador no lo permitirá.

Valores de retorno constantes

- Permite proteger el valor y contenido de un puntero para que el mismo no se modificado.

Ejemplo

```
class cadena {
   public:
      cadena();         // Constructor por defecto
      cadena(const char *c); // Constructor desde cadena c
      cadena(int n); // Constructor para cadena de n caracteres
      cadena(const cadena &);   // Constructor copia
      ~cadena();        // Destructor

      void Asignar(const char *dest);
      char *Leer(char *c) {
         strcpy(c, cad);
         return c;
      }
   private:
      char *cad;        // Puntero a char: cadena de caracteres
};
```

Valores de retorno constantes

- La función ”Leer”, verás que devuelve un puntero a la cadena que pasamos como parámetro, después de copiar el valor de `cad` en esa cadena.
- Si nos limitáramos a devolver ese parámetro, el programa podría modificar la cadena almacenada a pesar de ser `cad` un miembro privado.
- Para evitar eso podemos declarar el valor de retorno de la función ”Leerçomo constante

```
   ...
   const char *Leer() { return cad; }
   ...

int main() {
   cadena Cadena1("hola");

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

Outline

Clases base y clases derivadas

- Cada nueva clase obtenida mediante herencia se conoce como *clase derivada*.
- Las clases a partir de las cuales se deriva, *clases base*.
- Cada clase derivada puede usarse como clase base para obtener una nueva clase derivada.
- Cada clase derivada puede serlo de una o más clases base (derivación múltiple).
- Puede crearse una jerarquía de clases tan compleja como sea necesario.

Derivación de clases

Sintáxis

```
class <clase_derivada> :
   [public|private] <base1> [,[public|private] <base2>] {};
```

- Para cada clase base podemos definir dos tipos de acceso:
  - `public`: los miembros heredados de la clase base conservan el tipo de acceso con que fueron declarados en ella.
  - `private`: todos los miembros heredados de la clase base pasan a ser miembros privados en la clase derivada.
- Si no se especifica ninguno de los dos, por defecto se asume que es `private`.

Ejemplo

```
// Clase base Persona:
class Persona {
  public:
   Persona(char *n, int e);
   const char *LeerNombre(char *n) const;
   int LeerEdad() const;
   void CambiarNombre(const char *n);
   void CambiarEdad(int e);

  protected:
   char nombre[40];
   int edad;
};

// Clase derivada Empleado:
class Empleado : public Persona {
  public:
   Empleado(char *n, int e, float s);
   float LeerSalario() const;
   void CambiarSalario(const float s);

  protected:
   float salarioAnual;
};
```

- El acceso `protected` nos permite que los datos sean inaccesibles desde el exterior de las clases, pero a la vez, permite que sean accesibles desde las clases derivadas.

Constructores de clases derivadas

- Cuando se crea un objeto de una clase derivada, primero se invoca al constructor de la clase o clases base y a continuación al constructor de la clase derivada.
- Si la clase base es a su vez una clase derivada, el proceso se repite recursivamente.
- Si no hemos definido los constructores de las clases, se usan los constructores por defecto que crea el compilador.

Ejemplo

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

class ClaseA {
  public:
   ClaseA() : datoA(10) {
      cout << "Constructor de A" << endl;
   }
   int LeerA() const { return datoA; }

  protected:
   int datoA;
};

class ClaseB : public ClaseA {
  public:
   ClaseB() : datoB(20) {
      cout << "Constructor de B" << endl;
   }
   int LeerB() const { return datoB; }

  protected:
   int datoB;
};

int main() {
   ClaseB objeto;

   cout << "a = " << objeto.LeerA()
        << ", b = " << objeto.LeerB() << endl;
   return 0;
}
```

Inicialización de clases base en constructores

- Para inicializar las clases base usando parámetros desde el constructor de una clase derivada se utiliza el mismo método que para con los datos miembro.
- Las llamadas a los constructores deben escribirse antes de las inicializaciones de los parámetros.

Sintáxis

```
<clase_derivada>(<lista_de_parámetros>) :
   <clase_base>(<lista_de_parámetros>) {}
```

Ejemplo

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

class ClaseA {
  public:
   ClaseA(int a) : datoA(a) {
      cout << "Constructor de A" << endl;
   }
   int LeerA() const { return datoA; }

  protected:
   int datoA;
};

class ClaseB : public ClaseA {
  public:
   ClaseB(int a, int b) : ClaseA(a), datoB(b) { // (1)
      cout << "Constructor de B" << endl;
   }
   int LeerB() const { return datoB; }

  protected:
   int datoB;
};

int main() {
   ClaseB objeto(5,15);

   cout << "a = " << objeto.LeerA() << ", b = "
        << objeto.LeerB() << endl;
   return 0;
}
```

Inicialización de objetos miembros de clases

- Cuando una clase tiene miembros objetos de otras clases, esos miembros pueden inicializarse se procede del mismo modo que con cualquier dato miembro
- Se añade el nombre del objeto junto con sus parámetros a la lista de inicializaciones del constructor.

Ejemplo

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

class ClaseA {
  public:
   ClaseA(int a) : datoA(a) {
      cout << "Constructor de A" << endl;
   }
   int LeerA() const { return datoA; }

  protected:
   int datoA;
};

class ClaseB {
  public:
   ClaseB(int a, int b) : cA(a), datoB(b) {
      cout << "Constructor de B" << endl;
   }
   int LeerB() const { return datoB; }
   int LeerA() const { return cA.LeerA(); }

  protected:
   int datoB;
   ClaseA cA;
};

int main() {
   ClaseB objeto(5,15);

   cout << "a = " << objeto.LeerA() << ", b = "
        << objeto.LeerB() << endl;
   return 0;
}
```

Destructores de clases derivadas

- Cuando se destruye un objeto de una clase derivada, primero se invoca al destructor de la clase derivada
- Si existen objetos miembro a continuación se invoca a sus destructores.
- Finalmente, se llama al destructor de la clase o clases base.
- Si la clase base es a su vez una clase derivada, el proceso se repite recursivamente.

Ejemplo

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

class ClaseA {
  public:
   ClaseA() : datoA(10) {
      cout << "Constructor de A" << endl;
   }
   ~ClaseA() { cout << "Destructor de A" << endl; }
   int LeerA() const { return datoA; }

  protected:
   int datoA;
};

class ClaseB : public ClaseA {
  public:
   ClaseB() : datoB(20) {
      cout << "Constructor de B" << endl;
   }
   ~ClaseB() { cout << "Destructor de B" << endl; }
   int LeerB() const { return datoB; }

  protected:
   int datoB;
};

int main() {
   ClaseB objeto;

   cout << "a = " << objeto.LeerA() << ", b = "
        << objeto.LeerB() << endl;
   return 0;
}
```

Outline

Redefinición de funciones en clases derivadas

- En una clase derivada se puede definir una función que ya existía en la clase base, esto se conoce como superposición o redefinición de una función.
- La redefinición de la función en la clase derivada oculta la definición previa en la clase base.
- En caso necesario, es posible acceder a la función oculta de la clase base mediante su nombre completo:
  `<objeto>.<clase_base>::<método>;`

Ejemplo

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

class ClaseA {
  public:
   ClaseA() : datoA(10) {}
   int LeerA() const { return datoA; }
   void Mostrar() {
      cout << "a = " << datoA << endl;
   }
  protected:
   int datoA;
};

class ClaseB : public ClaseA {
  public:
   ClaseB() : datoB(20) {}
   int LeerB() const { return datoB; }
   void Mostrar() {
      cout << "a = " << datoA << ", b = "
           << datoB << endl;
   }
  protected:
   int datoB;
};

int main() {
   ClaseB objeto;

   objeto.Mostrar();
   objeto.ClaseA::Mostrar();

   return 0;
}
```

Superposición y sobrecarga

- Cuando se superpone una función, se ocultan todas las funciones con el mismo nombre en la clase base.
- No es posible acceder a ninguna de las funciones superpuestas de la clase base, aunque tengan distintos valores de retorno o distinto número o tipo de parámetros.

Ejemplo

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

class ClaseA {
  public:
   void Incrementar() { cout << "Suma 1" << endl; }
   void Incrementar(int n) { cout << "Suma " << n << endl; }
};

class ClaseB : public ClaseA {
  public:
   void Incrementar() { cout << "Suma 2" << endl; }
};

int main() {
   ClaseB objeto;

   objeto.Incrementar();
//   objeto.Incrementar(10); No es posible hacer este llamado
   objeto.ClaseA::Incrementar();
   objeto.ClaseA::Incrementar(10);

   return 0;
}
```

Polimorfismo

- Nos permite acceder a objetos de una clase derivada usando un puntero a la clase base.
- Sólo podremos acceder a datos y funciones que existan en la clase base, los datos y funciones propias de los objetos de clases derivadas serán inaccesibles.

Ejemplo

```
class Persona {
  public:
   Persona(char *n) { strcpy(nombre, n); }
   void VerNombre() { cout << nombre << endl; }
  protected:
   char nombre[30];
};

class Empleado : public Persona {
  public:
   Empleado(char *n) : Persona(n) {}
   void VerNombre() {
      cout << "Emp: " << nombre << endl;
   }
};

class Estudiante : public Persona {
  public:
   Estudiante(char *n) : Persona(n) {}
   void VerNombre() {
      cout << "Est: " << nombre << endl;
   }
};

int main() {
   Persona *Pepito = new Estudiante("Jose");
   Persona *Carlos = new Empleado("Carlos");

   Carlos->VerNombre();
   Pepito->VerNombre();
   delete Pepito;
   delete Carlos;
   return 0;
}
```

Funciones Virtuales

- Pemiten que se invoque cuando una función que se superpone en la clase derivada, se llame a la de la clase derivada.
- Cuando en una clase declaramos una función como virtual, y la superponemos en alguna clase derivada, al invocarla usando un puntero de la clase base, se ejecutará la versión de la clase derivada.
- Sintaxis: `virtual <tipo> <nombre_función>(<lista_parámetros>) [{}];`

Ejemplo

```
class Persona {
  public:
   Persona(char *n) { strcpy(nombre, n); }
   virtual void VerNombre() {
      cout << nombre << endl;
   }
  protected:
   char nombre[30];
};
```