Programación orientada a objetos

El Lenguaje C++

Sumario

- Nuevas incorporaciones al C
- Evolución hacia la programación orientada a objetos
- El lenguaje C++
  - Clases y objetos
    - Constructores y destructores
    - Clases compuestas
  - Herencia
    - Re-definición de miembros
    - Herencia múltiple
    - Constructores y destructores en clases derivadas
  - Poliformismo
    - Sobrecarga de funciones y operadores
    - Ligadura dinámica: funciones virtuales
  - Genericidad
    - Funciones genéricas
    - Clases genéricas
  - Entrada/Salida
  - Manejo de excepciones

- Espacio de utilización de una variable
  - se puede declarar variables en cualquier parte del programa.

```
void f(void)
{ int i;
  i=1;
  int q=5;
    { int j;
      j=1;
    }
  i=1;
  q=6;
}
```

  - variables globales

```
int v;

void f1(void)
{...
}

int x;
void f2(void)
{ ...
}
```

- Acceso a variables ocultas

```
int x;

void f(void)
{
    int x;

    x=1;
    ::x=2; //modificamos la x externa
}
```

- Referencias
  - Nombre alternativo para una variable u objeto

```
int i=1;
int &x=i;
```

  - Las variables x e i se refieren al mismo espacio de memoria.
  - Se tienen que inicializar siempre.
  - Se utilizan en pase de parámetros a funciones donde se quiera modificar su contenido.

- Memoria dinámica: new y delete
  - new tipo[tamaño];
  - delete [] puntero;

```
char *p;
char *q;

p=new char[20]; //p=malloc(20);
q=new char;

delete []p; //free(p);
delete q;
```

- Declaraciones de funciones
  - En C++ sólo se permite declaración ANSI.
  - Parámetros con valor por defecto:

```
void f(char a='a')
{
printf("El valor de a es %c\n",a);
}
```

  - Llamadas:

```
f();   → El valor de a es a
f('F') → El valor de a es F
```

- Si hay varios parámetros, si el parámetro k-1 está definido por defecto, el parámetro k también tiene que estarlo.

```
void f(int i,int x=2,int y=2);
void f(int x=2,int y);  //incorrecto
```

- Pase de parámetros por referencia

```
int x=2;
f(x); ...

void f(int &x) { x++; }
```

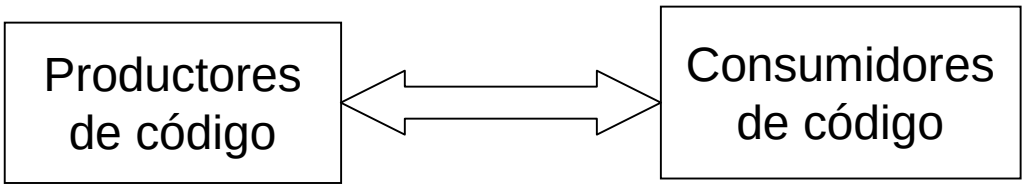
- Operadores E/S:
  - operador extracción >>
  - operador inserción <<
  - Stream entrada estándar: cin
  - Stream salida estándar: cout

```
int edades[10];
for(i=0; i<10; i++)
  { cout << "Edad persona num "<< i <<":";
    cin >> edad[i];
    cout << "\n";
  }
```

La evolución hacia la programación orientada a objetos

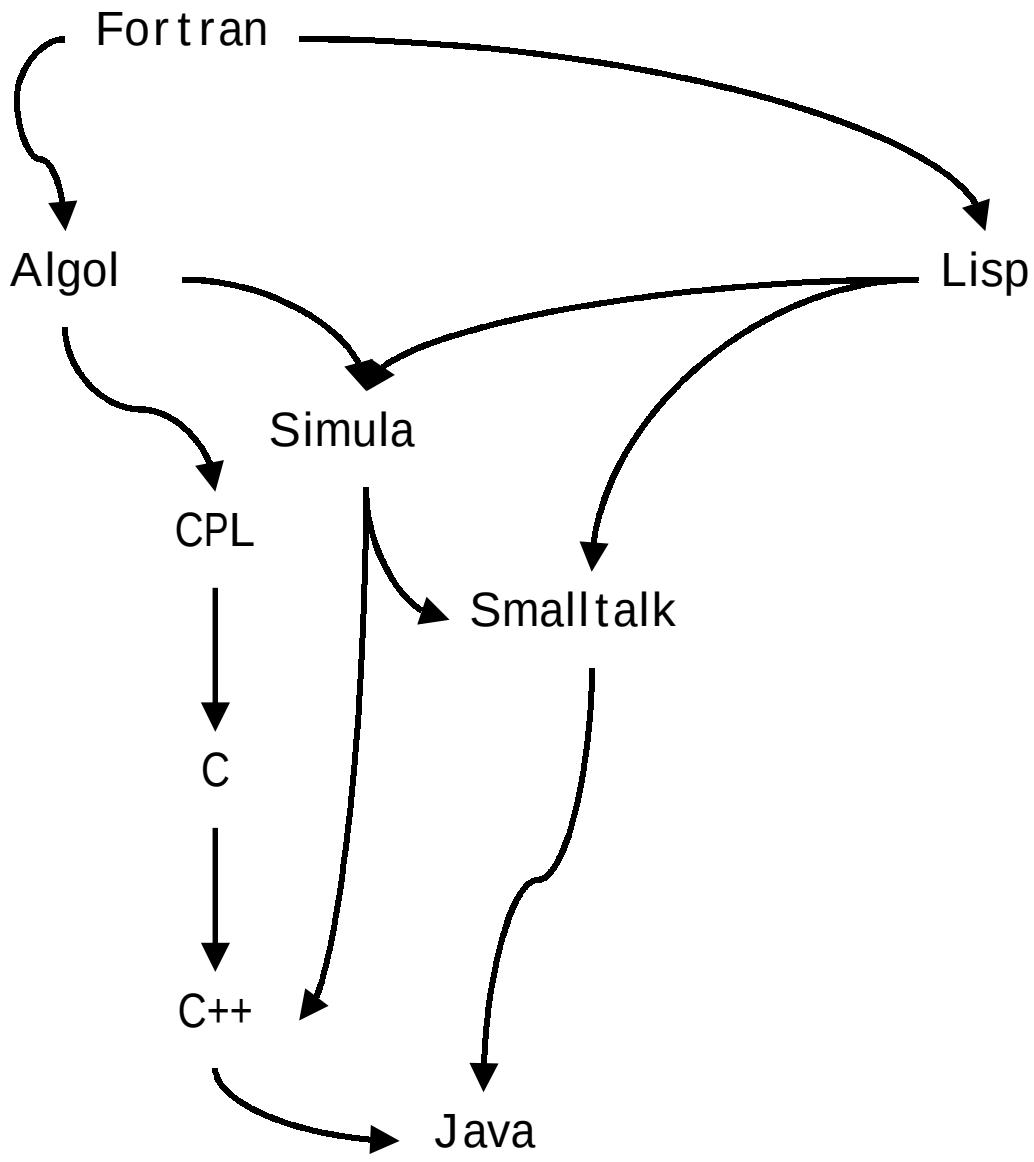
1. La crisis del software.
2. Fases de la evolución
   1. Los procedimientos
   2. Los módulos
   3. Los objetos

- Tradicionalmente, la programación ha sido un proceso artesanal.
- Aumento de la complejidad y tamaño de los proyectos (crisis del SW).
  - Metodología insostenible.
  - Facilitar intercambio del software:
    - Entre programadores de un mismo proyecto.
    - Diferentes proyectos (ej. compra de librerias).



  - Abstracción: acercarse a las ideas y huir de las cuestiones técnicas.
  - Re-utilización del código:
    - Ahorro de tiempo y costes.
    - Fiabilidad.
  - Aplicación de metodologías que mejoren y faciliten el desarrollo del software (Ingeniería del Software).

- Necesidades (motivaciones):
  - programas de gran talla
  - estructuración de masa de conocimientos



Evolución hacia la P.O.O

1. Procedimientos
2. Módulos (Programación modular)
3. Objetos (Programación orientada a objetos)

Objetivos de la evolución:

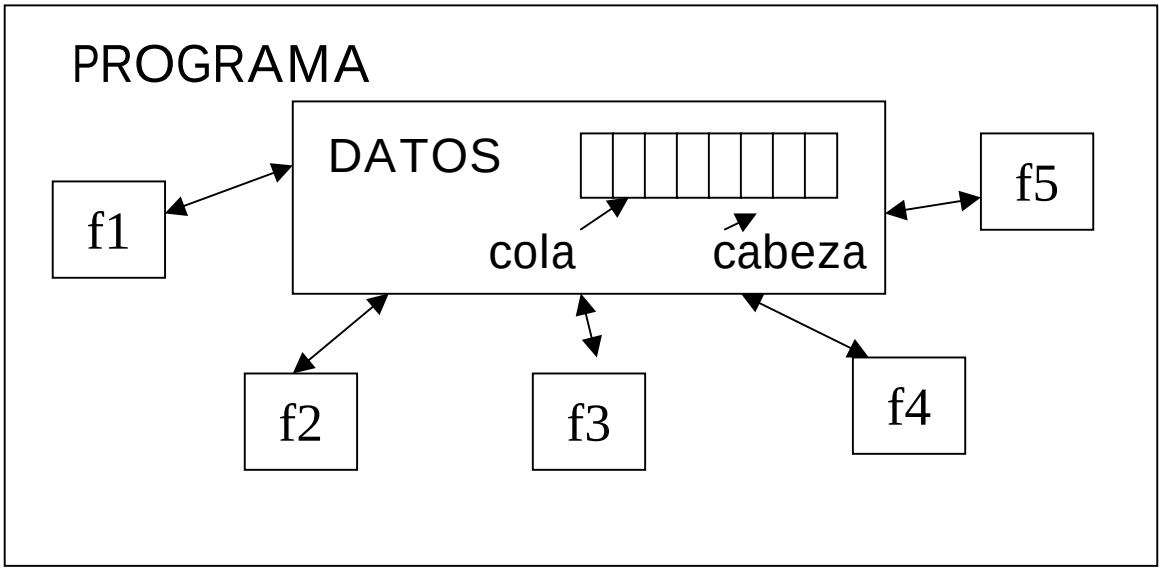
- Menor interconexión entre las partes de un programa.
- Ocultación de la información.
- Abstracción:
  - que los componentes del S.W. sean los más cercanos posibles a las ideas que componen el problema).
- Re-utilización del código.

Todos estos objetivos están relacionados, pero la piedra angular es la ocultación de la información.

Programación con procedimientos

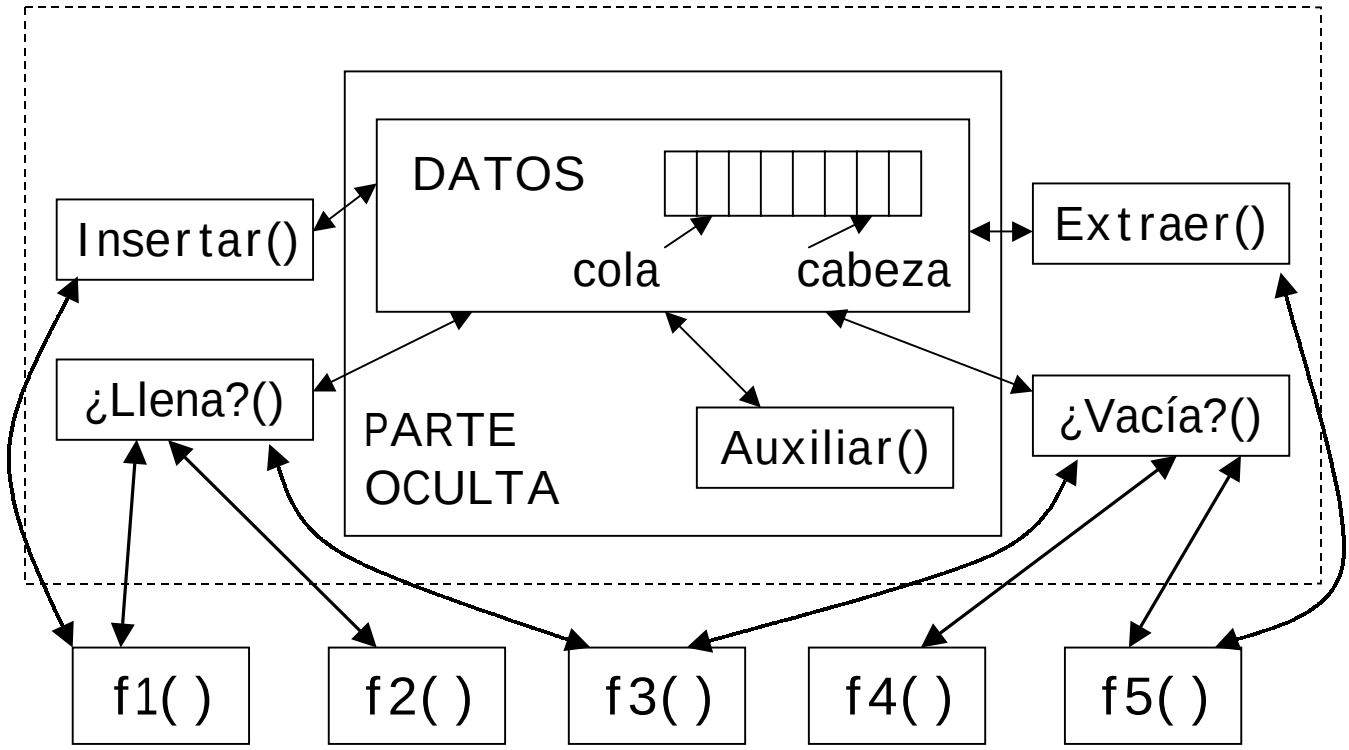
- El programa se divide en dos partes:
  - Algoritmos
  - Datos
- La parte algorítmica se estructura en procedimientos.
- Beneficios:
  - Abstracción de operaciones.
  - Extensión del lenguaje.
  - Re-utilización del código.
- Inconvenientes:
  - No existe una clara estructuración de los datos que los relacione con los procedimientos que los manejan.
  - Indefensión ante varios problemas:
    - No se garantiza el correcto estado de las estructuras de información.
    - No hay independencia de representación.

Ejemplo: manejo FIFO



- No hay nada que impida que un programador haga un mal uso de los datos. Ejemplo:
  - confundir el uso de cola y cabeza.
- No es posible garantizar que la cola se encuentra en un estado correcto.
- ¿Que sucede si un día queremos cambiar la estructura de los datos.?
  - Por ejemplo: cambiar el vector de la cola por una lista enlazada.
    - **Tendría que re-escribir el programa.**
- Realmente,¿qué parte del código es re-utilizable?.

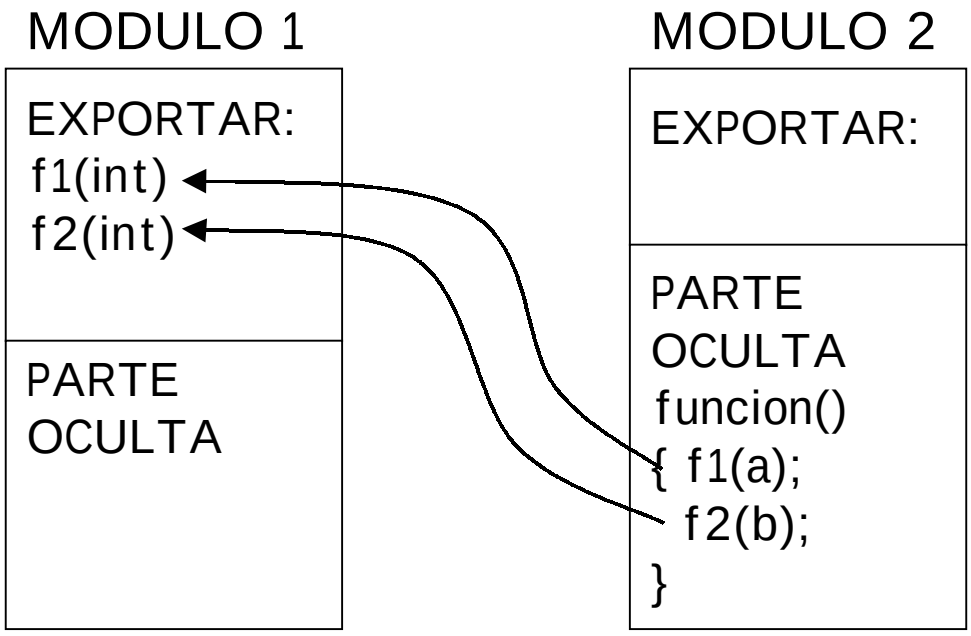
Ocultación de la información



- Cierta parte de los datos y funciones sólo será accesible desde ciertas funciones.
- Si el interfaz está bien definido, el estado de la cola siempre es correcto.
- Si se cambia la estructura interna de los datos sólo hay que cambiar las funciones del interfaz.
- Re-utilización del código: se aprovecha
  - la estructura de datos,
  - las funciones que la manejan.
- Ocultación de la información = Encapsulamiento de los datos

Formas de ocultación de la información

- Módulos (Programación modular)
  - Las funciones (procedimientos) y datos se agrupan en diferentes ficheros (módulos).
  - Para cada módulo se definen los procedimientos y datos que se exportan.
  - Sólo se puede acceder desde fuera del módulo a aquella parte que ha sido exportada.



- Objetos: programación orientada a objetos

Ejemplo: MODULA-2

- Un módulo se compone de dos partes:
  - definición → Interfaz
  - implantación → El código que sustenta tanto el interfaz, como a la parte oculta.

```
DEFINITION MODULE COLA;
   EXPORT QUALIFIED insertar, extraer,
                    llena, vacia;
   PROCEDURE insertar(x:INTEGER): BOOLEAN;
   PROCEDURE extraer ( ): INTEGER;
   PROCEDURE llena ( ): BOOLEAN;
   PROCEDURE vacia ( ): BOOLEAN;
END cola;

IMPLEMENTATION MODULE COLA;
       CONST max = 100;
       VAR A     : ARRAY[0..max] OF INTEGER;
           cola  : CARDINAL;
           cabeza: CARDINAL;

   PROCEDURE insertar(x:INTEGER):BOOLEAN
   BEGIN
   IF llena THEN RETURN FALSE;
   ...
   END insertar;
```

```
    PROCEDURE extraer ( ) : INTEGER
    BEGIN
    ...
    END extraer;

    ...

BEGIN
cola:=0;
cabeza:=0;
END cola.
```

```
MODULE Programa;
    FROM cola IMPORT insertar, extraer,
                     llena, vacia;

    VAR n : INTEGER;

BEGIN
    ...
    insertar(n);
    ...
END
END Programa.
```

- Aportaciones de la modularidad
  - Descomposición de un problema complejo en módulos más simples.
  - Re-utilización de los módulos para componer nuevo S.W.
  - Independencia de la implantación.
  - Ocultación de la información.

- Limitaciones de la modularidad
  - Sólo puedo tener una cola en mi programa.

- Los objetos son entidades que agrupan datos y procedimientos que operan sobre esos datos.
  - modulo ≈ fichero
  - objeto ≈ variable
- Los objetos se caracterizan por tres propiedades:
  - Estado
  - Operaciones
  - Identidad dentro del programa
- Una clase es la descripción de un conjunto de objetos similares.
- Los objetos son instancias de una clase.
  - Por ejemplo: podemos definir la clase matriz y a partir de ella los objetos matriz1 y matriz2.

Clases y objetos

---

1. Clases y objetos
2. Mensajes
3. El concepto de objeto
4. Declaración de clases en C++
5. Constructores y destructores
6. Clases compuestas
7. Expansión inline
8. Clases y funciones friend
9. Calificación de variables miembro
10. Variables y funciones de clase
11. La palabra reservada this
12. Vectores y punteros a objetos

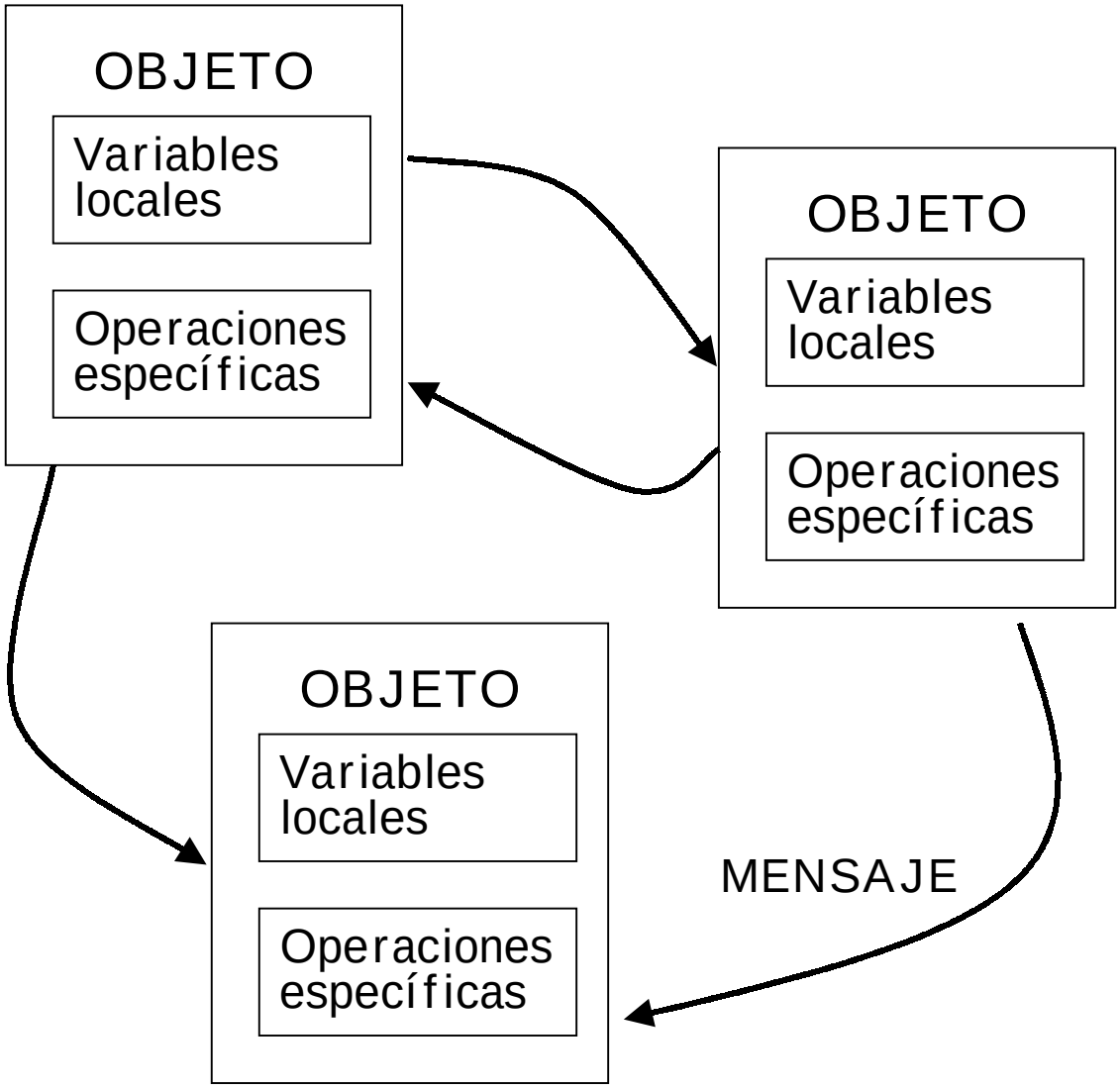
- **CLASE**:

  Abstracción de un conjunto de objetos caracterizados por las mismas propiedades y los mismos comportamientos.

- **OBJETO**:

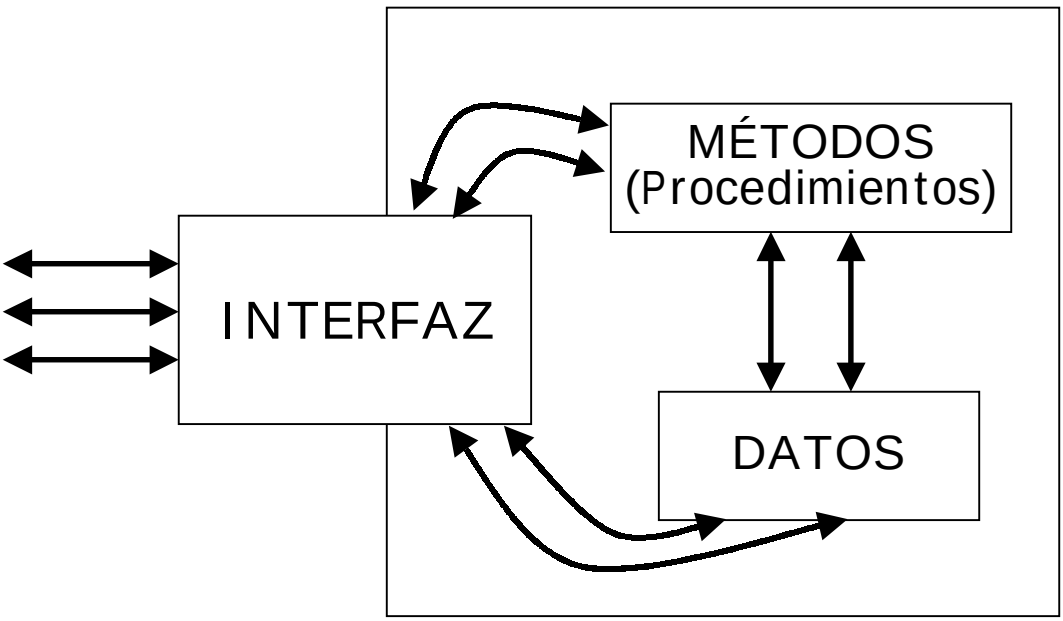
  Abstracción (máquina abstracta) capaz de responder a una petición de servicio mensaje.

- Un objeto es una instancia de una clase.
- La clase contiene su comportamiento y sus propiedades.
- El objeto puede crearse dinámicamente en el transcurso de la ejecución de un programa.

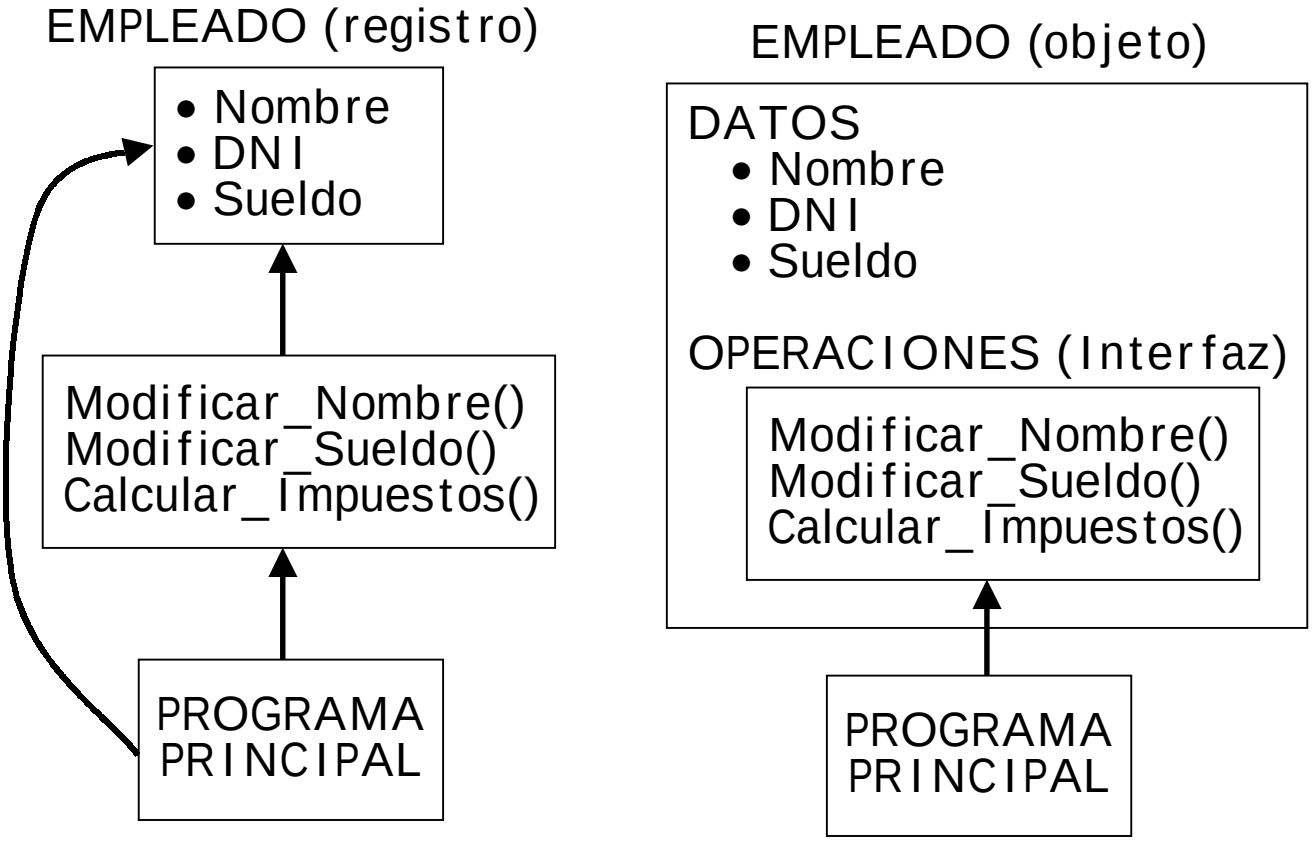


- El MENSAJE es el único medio de comunicación entre los objetos.
- Consiste en una petición explícita de una operación que debe realizar el objeto (puede ser asimilado a una llamada de procedimiento)

- Objeto: Entidad de programación con componentes de dos tipos:
  - Estado:
    - datos
  - Comportamiento:
    - Procedimientos que manipulan los datos con exclusividad.
- Cada objeto posee un interfaz que contiene los servicios que oferta al exterior.



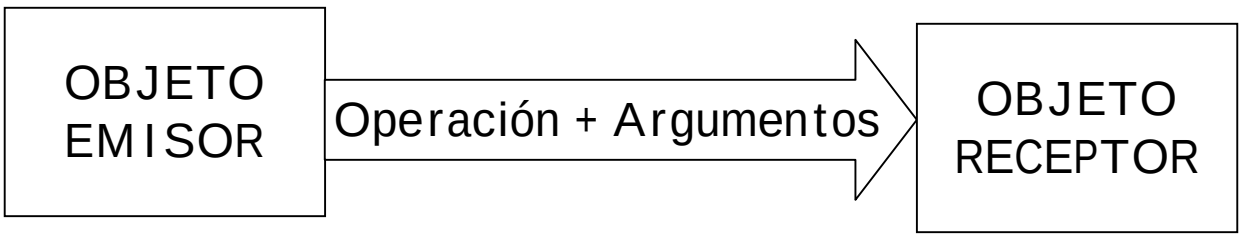
- Acceso convencional *vs* O.O.



- Comunicación entre objetos:

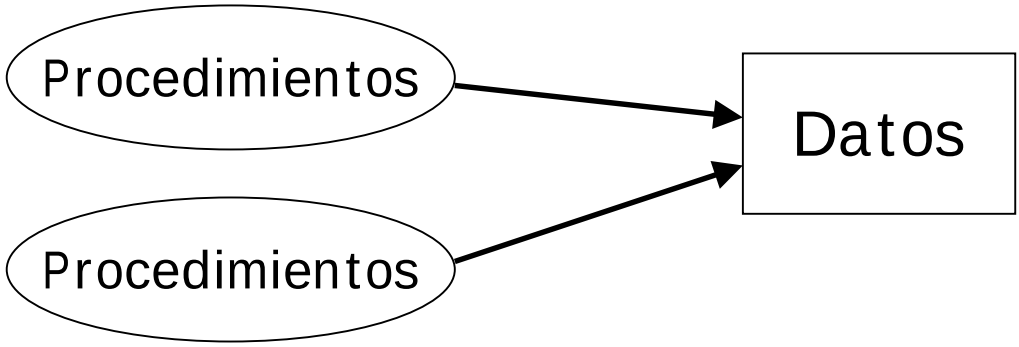
  Un objeto pide un servicio a otro por medio de un mensaje:

  destinatario + operación + argumentos



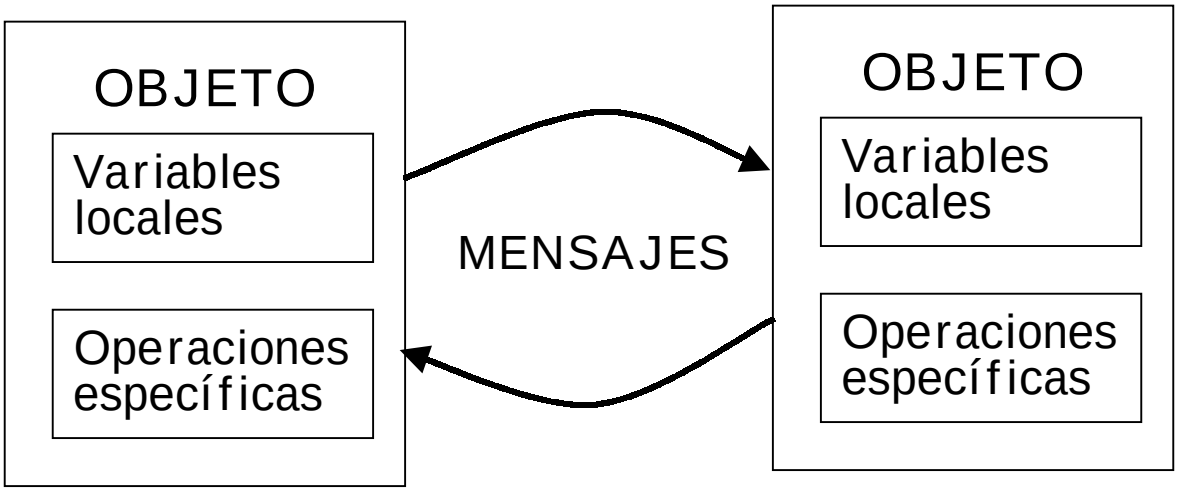
Globalidad *vs* Localidad

- Programación modular



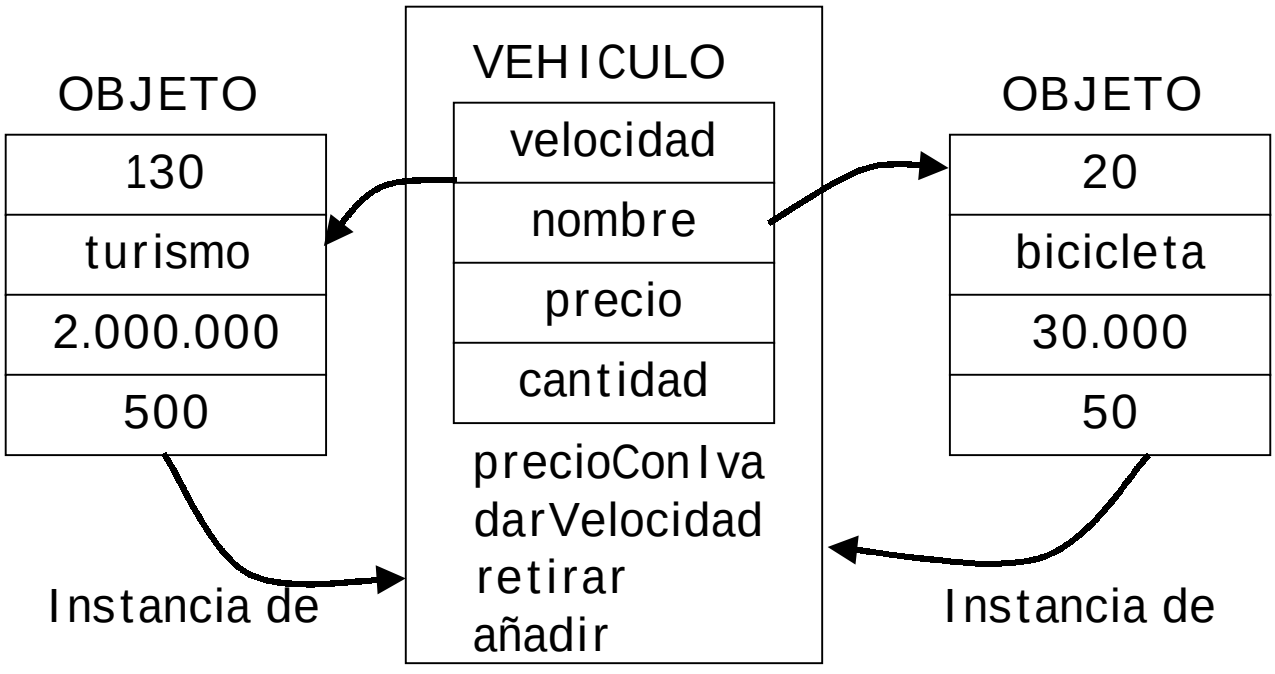
- Aquí los datos son globales, son accesibles desde todos los procedimientos.

- Programación por objetos:



- Aquí los datos son locales a cada objeto:
  - Sólo son accesibles para los procedimientos (operaciones) definidos en el objeto.

- CLASE
  - vehiculo
- CAMPOS
  - velocidad
  - nombre
  - precio
  - cantidad
- METODOS
  - precioConIVA:
    - devolver (1,16*precio)
  - darVelocidad:
    - devolver (velocidad)
  - retirar(c):
    - cantidad ← cantidad - c
  - añadir (c):
    - cantidad ← cantidad + c



- Declaración de una clase “Persona”.

```
class Persona
{ private:
        char nombre [30];
        int edad;
  public:
        void ModificarNombre(char *n);
        void ModificarEdad(int e);
        int SaberEdad(void);
};
```

- Definición de las funciones

```
void Persona::ModificarNombre(char * n)
{ strcpy (nombre , n ); }

void Persona::ModificarEdad(int e)
{ edad = e; }

int Persona::SaberEdad(void)
{ return edad; }
```

- Instanciación

```
void main(void)
{ Persona P; //P es una instancia

P.ModificarEdad(5);
}
```

- Tres formas de definir una clase C++:
  - Mediante la palabra struct:
    - Por defecto todos los miembros son públicos.

```
struct Cuadrado
{ double CalcularArea() ;
  void LeerDatos(double Lado1,
  double Lado2);
private:
 double Lado1 ;
 double Lado 2 ;
};
```

  - Mediante la palabra union:
    - por defecto los miembros son públicos
    - los datos comparten espacio de memoria.

```
union Nombre_Persona
{ void MuestraNombreCompleto();
  void MuestraApellido();
  void MuestraNombre();
private :
  char Nombre_Completo[30];
  char Nombre_y_Apellido[2][15];
};
```

- Mediante la palabra class:
  - los miembros son privados por defecto.
  - Es la forma usual de declarar clases
  - struct y union se suelen reservar para un uso similar al que tienen en C .

```
class Vehiculo
{ int Numero_Ruedas;
  int Numero_Ocupantes;
public :
  void MostrarNumeroOcupantes();
};
```

Programa ejemplo (pila)

```
// Programa ejemplo de definicion de clases
// Se define la clase pila, con dos funciones
// publicas pop y push .
// El programa principal permite rellenar y
// extraer caracteres de la pila

#include<iostream.h>

void MostrarOpciones(void ); //Prototipo función

class pila  //Clase de objetos de pila
{ char buf[10];  //Buffer de la pila
  int n;         //Elemento actual
  void errorllena(void); //Prototipo error llena
  void errorvacia(void); //Prototipo error vacia

public:
  void ini(void);    //Inicializa pila
  void push(char a); //Prototipo poner caracter
  char pop(void);    //Prototipo extraer caracter
};

void pila::errorllena(void)
{ cout << ”##ERROR. LA PILA ESTA LLENA/n”;
}

void pila::errorvacia ( void )
{ cout << “## ERROR.LA PILA ESTA VACIA/n”;
}
```

```
void pila::ini(void)
{ n = 0;
}
void pila::push(char a)
{ if ( n < 10 )
    buf[n++] = a;
  else
    errorrllena();
}

char pila::pop(void)
{ if ( n<=0 )
    { errorvacia () ;
      return ‘’ ;
    }
  else
    return buf[--n];
}


//Funcion para mostrar menu opciones
void MostrarOpciones(void)
{
cout << “\n A. Añadir\n”;
cout << “ E. Extraer\n”;
cout << “ F. Fin \n\n”;
cout << “ Opcion: “;
}
```

```
void main(void)
{ char c;
  pila p;
  int fin=0;

 p.ini();
 while(!fin)
 { MostrarOpciones();
   cin >> c;

   switch(c)
    { case ‘a’:
      case ‘A’: cout << “Escribe una letra: “;
                cin >> c;
                p.push(c);
                break;
      case ‘e’:
      case ‘E’: cout <<”La letra extraida es ”;
                cout << p.pop();
                cout << ”\n”;
                break;
      case ‘f’:
      case ‘F’: cout << “\n\nFin”;
                fin=1;
                break;
    }
  }
}
```

- Destructor
  - Función miembro de la clase que se ejecuta al final de la vida de cada objeto.

```
class A
{ ...

public:
  ~A(void);
};

A::~A(void)
{ cout << " Este objeto se ha muerto";
}
```

- Constructor
  - Función miembro de la clase que se ejecuta cuando se crea un objeto.
  - Podemos tener varios constructores en una clase, pero han de diferir en los parámetros.

```
class A
{ int dato1,dato2;
public :
  A(void);
  A(int d1);
  A(int d1,char d2);
};

A::A(void)            //constructor C1
{ dato1=dato2=0; }

A::A(int d1)          //constructor C2
{ dato1=dato2=d1; }

A::A(int d1,char d2) //constructor C3
{ dato1=d1; dato2=d2; }
```

- Utilización de constructores:
  - Sin parámetros:
    - Crea un objeto llamando a C1.
      ```
      A a;
      ```
  - Con un parámetro:
    - Crea un objeto llamando a C2.
      ```
      A a(5);  ó  A a=5;
      ```
  - Con más de un parámetro:
    - Crea un objeto llamando a C3.
      ```
      A a(5,3);
      ```

- Constructores por defecto:

```
class B
{ int dato1;

public :
  void f(int d);
};

...
B b1;    // constructor sin parámetros
B b2=b1; // constructor copia
```

  - Constructor copia:
    - Crea un objeto b2 donde los datos de b2 serán iguales a los de b1.
    - Aunque no se defina el constructor copia, éste existe para que se puedan pasar objetos como parámetros por valor.
  - Constructor sin parámetros:
    - Aunque no se defina, éste existe para que se puedan definir objetos de la clase (no ejecuta nada).

- Desaparición de constructores por defecto:
  - Constructor sin parámetros:
    - Desaparece cuando definimos cualquier otro constructor.

```
class B
{ int dato1;
public :
  B(int x);
};

B::B(int x) { dato1=x; }

void main(void)
{ B b1=6; // se llama al constructor
  B b2; // Incorrecto!!. El constructor
        // sin parámetros no existe en
        // esta clase.
}
```

- Constructor copia:
  - Desaparece cuando definimos un nuevo constructor copia.
  - Siempre hay un constructor copia, y solo uno, en cada clase.

```
class B
{ int *vector;
public :
  B(int x); // constructor
  B(B &b);  // constructor copia
};

...
B::B(B &b)
{ tam=b.tam;
   vector=new int[tam];
   memcpy(vector, b.vector,tam*sizeof(int));
}

void main(void)
{ B b1=6;  // se llama al constructor
  B b2=b1; // se llama al constructor
           // copia que se ha definido.
}
```

```
class  X
{ int c1;
  float c2;
public:
  X(void) { c1=0; c2=0 }
  X(X &ob) { c1=ob.c1; c2=ob.c2; }
};
```

Equivale a:
`X(void): c1(0),c2(0) {}`

Constructor copia
Si no lo defino es como si estuviera, y su acción es precisamente esa, copiar campo a campo.

- Utilización
  - Explícita:

```
X ob1;
X  ob2=ob1; //aquí se ejecuta el
            //constructor copia.
```

  - Implícita:

```
void f1(void)
{ x ob1;
  x ob2=f2(ob1);
}

X f2 (X par)
{ return par; }
```

Aquí se ejecuta 3 veces el constructor copia:
- Para construir ob2
- Para construir par
- Para construir el objeto temporal que se devuelve con `return par`

- Una clase donde algún dato miembro es un objeto de otra clase.
- ¿Cuál es el orden de construcción?

```
class contador
{ long cont;

public :
  contador(void)   { cont=0; }
  contador(long n) { cont=n; }
  void reset(void) { cont=0; }
  void incr(void) { cont ++; }
  long get_cont(void) { return cont; }
};


class cronometradores   //clase compuesta
{ contador crono1;
  contador crono2;

public:
   cronometradores() { }
   cronometradores(int x,int y) :
               crono1(x), crono2(y) {}
};
```

- Evitar llamadas a funciones.
- El compilador simula el pase de parámetros.

```
void main(void)
{ int x,y;
  int z ;
...
z = Calculo(x,y);
cout << Calculo(z,x);
}
```

El compilador lo convierte en ...

```
void main(void)
{ int x,y;
  int z ;
...
z=(((x/100)+(y/300))%5);
cout<<(((z/100 )+(x/300))%5;
}
```

```
inline int Calculo(int a,int b)
{ return ( ( (a/100)+ (b/300) ) %5 );
}
```

- Funciones *friend*:
  - Una función ajena a la clase pueda acceder a la parte privada de la clase.

```
class A
{ int x;

public:
  friend int funcion(A a);
};
```

```
int funcion(A a)
{ return(a.x/2); // puede acceder a la
                 // parte privada
}
```

- Clases *friend*:
  - Todos los métodos de una clase puedan acceder a la parte privada de otra clase.

```
class A
{ int x;

public:
   friend class Amiga;
};
```

```
class Amiga
{ A a;

public:
  void ModificaX()
            { a.x=5; }
};
```

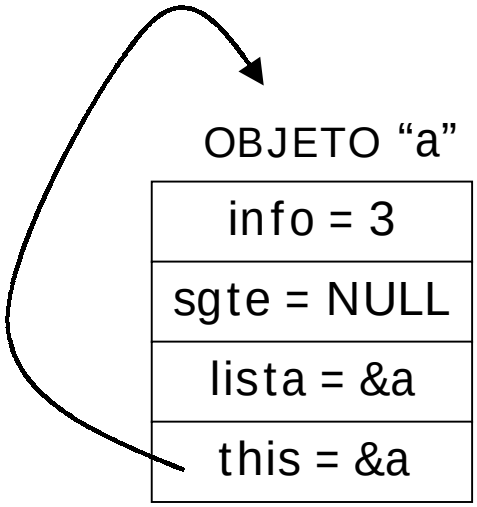
Calificación de variables miembro

- Distinguir entre:
  - variables miembro de una clase
  - otro tipo de variables.
- Operador resolución de ámbito "::"

```
class X
{ int m;

public:
  void Setm(int);
  void Getm(void) { cout << m; }
};


void X::Setm(int m)
{  X::m = m;  // distinguir entre el
              // parámetro m del miembro
              // m de la clase X
}

void main (void)
{ X x;

x.Setm(5);
x.Getm();
}
```

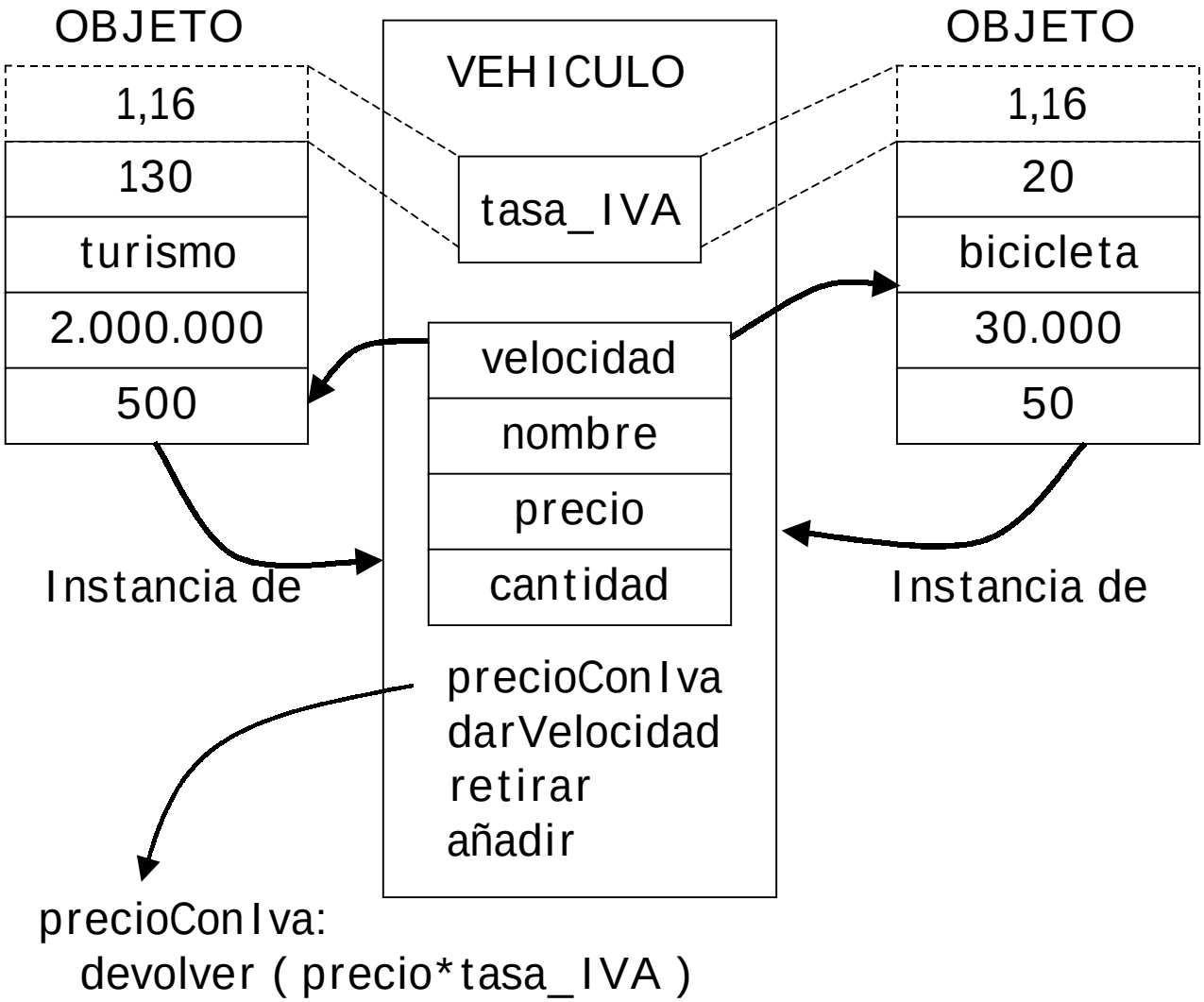
La palabra reservada *this*

- *this* en C++ representa un puntero al objeto que recibe la llamada.
- Todos los objetos de todas las clases tienen *this* como dato miembro.




```
class nodo
{ int info;
  nodo *sgte;
  static nodo *lista;
// nodo *this;
//...
};

nodo * nodo::lista=NULL;

void main(void)
{ nodo a(3);

//...
}
```

- Variables de clase:
  - conocimiento común a todos los objetos (instancias) de una clase.
- Variables de instancia:
  - conocimiento propio de cada objeto (instancia) de la clase.



Variables de clase e instancia

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

class Persona
{ static char Nclass[10]; //variable de clase
  char Ninstancia[10]; //variable de instancia

public:
  void SetNClass(int NuevoValor)
               { strcpy(Nclass,Nuevonombre); }

  const char *GetNClass(void)
               { return Nclass; }
};

char Persona::Nclass[10]="kk";
                              //inicialización
void main(void)
{ Persona P,Q;

cout << P.GetNClass(); // salida "kk"

P.SetNClass("Persona");
cout << P.GetNClass(); // salida "Persona"

Q.SetNclass("Humanos");
cout<<P.GettNClass() ; // salida "Humanos"
}
```

- Funciones de clase:
  - Actúan sobre variables de clase.
  - Sólo pueden actuar sobre las variables de instancia de la propia clase cuando:
    - Pertenecen a un objeto pasado como parámetro.
    - Pertenecen a un objeto declarado dentro de la propia función.

```
#include <iostream.h>

class nodo
{ int info;
  nodo *sgte;
  static nodo *lista;

public:
  nodo(int n );
  void imprimir();
  static void imprimir_lista();
};

nodo::nodo(int n)
{ info = n;
  sgte = lista;
  lista = this;
}
```

```
void nodo::imprimir(void)
{ cout << info << endl;
}

void nodo::imprimir_lista(void)
{
for(nodo *p=lista; p!=NULL; p=p->sgte)
  p->imprimir();
}


nodo * nodo::lista=NULL; //inicialización


void main(void)
{ nodo a(3);
  nodo b(2);
  nodo c(1);

c.imprimir_lista(); //misma salida que
nodo::imprimir_lista();
}
```

Inicialización de vectores

- No es posible ejecutar un constructor para todos los elementos de un vector.

```
X a(5);     //variable simple
X vector[10](5); //incorrecto
```

- Si es posible ejecutarlo para cada uno de los elementos del vector.

```
X vector[10]={X(5),X(6),...};
```

- Esto es similar a:

```
X vector[10]={5,6,...};
```

- Para constructores de más parámetros

```
X vector[10]={X(5,2),X(3,4),...};
```

- Al salir del ámbito de utilización del vector:
  - se ejecutará el destructor para cada objeto del vector.

- Declaración de punteros a objetos de una clase:

```
class X
{ //...

public:
  X(void) { cout << “el constructor\n”; }
};

//...

X *a; // a es un puntero a objetos de X
```

- Utilización del operador *new* y *delete*:
  - Al ejecutarse el operador *new*, se llama al constructor de la clase automáticamente (*malloc* no lo hace).
  - *delete* llama al destructor (*free* no).

```
X *a;
a = new X;  // se ejecuta el constructor
delete a;   // se ejecuta el destructor
```

Ejemplo: vectores y punteros a objetos

```
#include <iostream.h>

class Numero
{ int n ;

public:
  Numero(int valor=10) { n = valor; }
  ~Numero(void) { cout<< “Destructor\n”; }
  void PrintNumero() { cout<< n <<“\n”; }
};

void main (void)
{ Numero N[10];// No es posible pasarle
               // argumento al constructor
for(int i=0; i<10; i++)
  N[i].PrintNumero();

Numero *NN = new Numero;  // al hacer el
          // new se ejecuta el constructor

NN->PrintNumero();
delete NN;  // el destructor se ejecuta
            // automaticamente.
}
// Al acabar el ámbito del vector N, se
// ejecutan los diez destructores
```