Laboratorio de Programación de Sistemas
Guiones de las Sesiones (3ª edición)

B. Casas R.M. Jiménez C. Martínez S. Pérez

5 de octubre de 2006

Concepto de *TAD* (Tipo Abstracto de Datos)

- Un TAD define un conjunto de valores y de funciones sobre dichos valores.

Concepto de *TAD* (Tipo Abstracto de Datos)

- Un TAD define un conjunto de valores y de funciones sobre dichos valores.
- El calificativo abstracto responde al hecho de que la definición (o especificación) de los valores y comportamiento de las funciones es independiente de cualquier implementación.

Concepto de *TAD* (Tipo Abstracto de Datos)

- Un TAD define un conjunto de valores y de funciones sobre dichos valores.
- El calificativo abstracto responde al hecho de que la definición (o especificación) de los valores y comportamiento de las funciones es independiente de cualquier implementación.
- El comportamiento de las funciones puede especificarse mediante algún formalismo (por ejemplo, ecuaciones) o notación informal.

Concepto de *TAD* (Tipo Abstracto de Datos)

- Un TAD define un conjunto de valores y de funciones sobre dichos valores.
- El calificativo abstracto responde al hecho de que la definición (o especificación) de los valores y comportamiento de las funciones es independiente de cualquier implementación.
- El comportamiento de las funciones puede especificarse mediante algún formalismo (por ejemplo, ecuaciones) o notación informal.
- Una especificación debe ser siempre precisa y completa.

Concepto de *TAD* (Tipo Abstracto de Datos)

Ejemplo de un TAD y su especificación

```
TAD CELDA genero celda ops
    cero:  → celda
    crear: entero → celda
    leer: celda → entero
    escribir: celda entero → celda
fops
```

Concepto de *TAD* (Tipo Abstracto de Datos)

- La operación cero devuelve una celda con valor 0.

Concepto de *TAD* (Tipo Abstracto de Datos)

- La operación cero devuelve una celda con valor 0.
- La operación crear devuelve una celda cuyo valor es el entero dado.

Concepto de *TAD* (Tipo Abstracto de Datos)

- La operación cero devuelve una celda con valor 0.
- La operación crear devuelve una celda cuyo valor es el entero dado.
- La operación leer consulta el valor de la celda.

Concepto de *TAD* (Tipo Abstracto de Datos)

- La operación cero devuelve una celda con valor 0.
- La operación crear devuelve una celda cuyo valor es el entero dado.
- La operación leer consulta el valor de la celda.
- La operación escribir modifica el valor de la celda por el entero dado.

Concepto de *Clase* y de *Objeto*

- Una clase es una representación de un TAD.

Concepto de *Clase* y de *Objeto*

- Una clase es una representación de un TAD.
- La diferencia entre una clase y un TAD es que mientras un TAD define un conjunto de valores y de funciones sobre dichos valores, una clase define un conjunto de objetos y operaciones sobre los objetos.

Concepto de *Clase* y de *Objeto*

- Una clase es una representación de un TAD.
- La diferencia entre una clase y un TAD es que mientras un TAD define un conjunto de valores y de funciones sobre dichos valores, una clase define un conjunto de objetos y operaciones sobre los objetos.
- Un objeto es un instancia concreta de una clase.

Concepto de *Clase* y de *Objeto*

- Una clase es una representación de un TAD.
- La diferencia entre una clase y un TAD es que mientras un TAD define un conjunto de valores y de funciones sobre dichos valores, una clase define un conjunto de objetos y operaciones sobre los objetos.
- Un objeto es un instancia concreta de una clase.
- Un objeto almacena la representación de un valor de un TAD. Es por tanto una "caja" capaz de guardar la representación de un valor abstracto. La clase a la que pertenece un objeto define los valores permitidos y las operaciones que se pueden aplicar sobre el objeto.

Concepto de *Clase* y de *Objeto*

- En la definición de una clase hay dos tipos de componentes: los atributos y los métodos.

Concepto de *Clase* y de *Objeto*

- En la definición de una clase hay dos tipos de componentes: los atributos y los métodos.
- Los métodos son las operaciones que se permiten aplicar sobre los objetos de la clase.

Concepto de *Clase* y de *Objeto*

- En la definición de una clase hay dos tipos de componentes: los atributos y los métodos.
- Los métodos son las operaciones que se permiten aplicar sobre los objetos de la clase.
- Cada atributo almacena una parte de la información contenida en un objeto. Todos los objetos de una clase poseen los mismos atributos, y difieren potencialmente en el valor de los atributos. Los atributos representan al valor abstracto de ese objeto, también denominado estado.

Concepto de *Clase* y de *Objeto*

- Las diferentes instancias de una clase (objetos) se identifican generalmente por un nombre o identificador propio. Si `X` es el nombre de una clase, la declaración

```
X mi_obj;
```

dice que `mi_obj` es un objeto de la clase `X`, `mi_obj` es el identificador del objeto.

Concepto de *Clase* y de *Objeto*

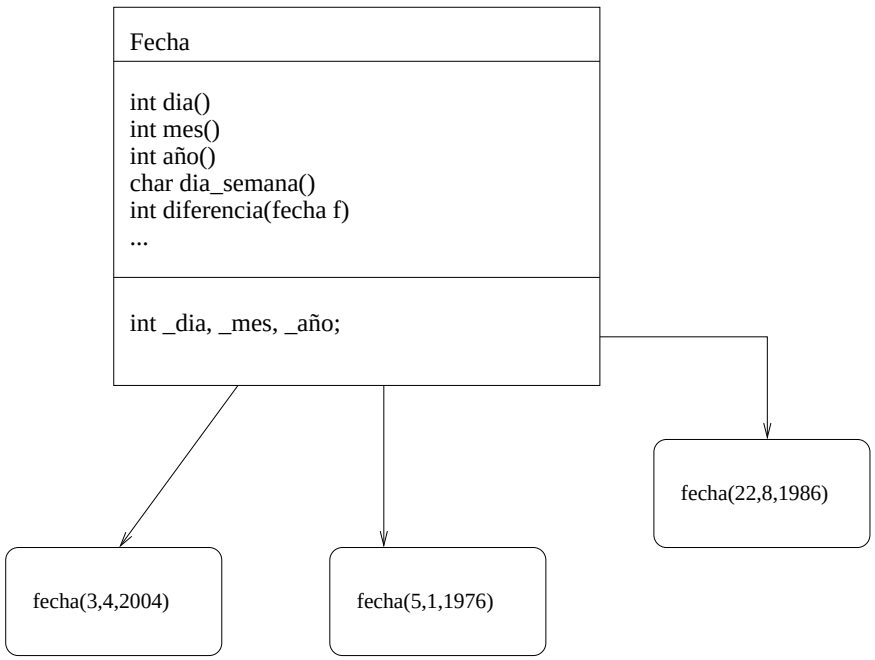
- Las diferentes instancias de una clase (objetos) se identifican generalmente por un nombre o identificador propio. Si `X` es el nombre de una clase, la declaración

```
X mi_obj;
```

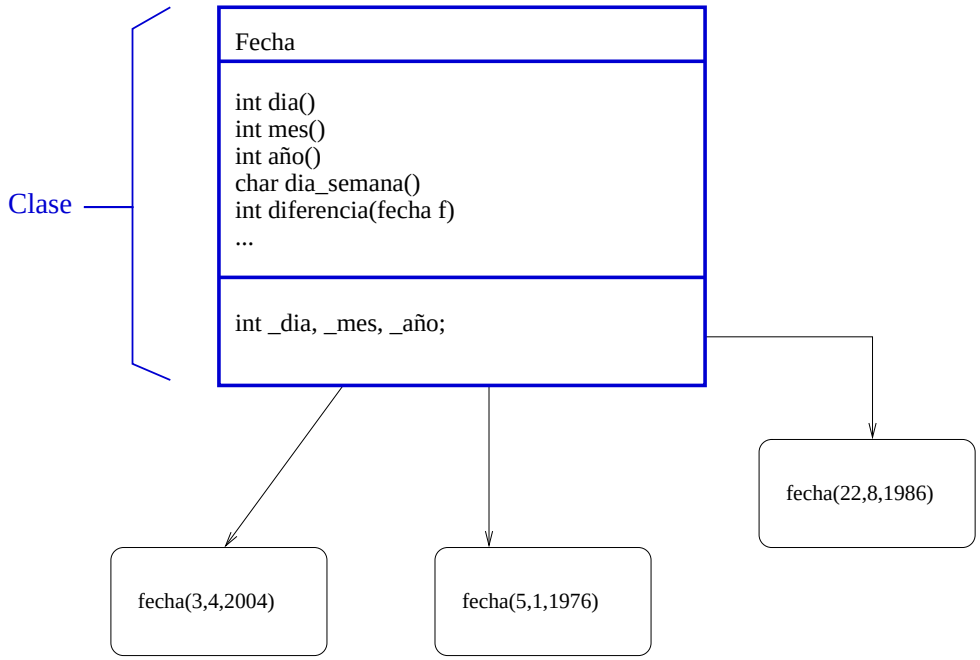
  dice que `mi_obj` es un objeto de la clase `X`, `mi_obj` es el identificador del objeto.

- Habitualmente sólo se permite cambiar el estado de un objeto mediante sus métodos.

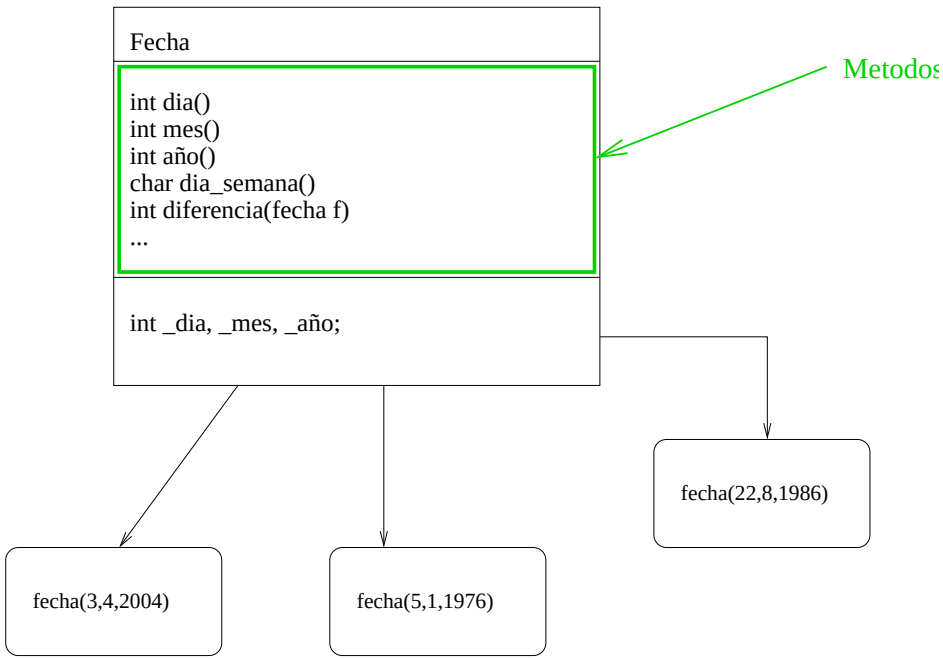
Concepto de *Clase* y de *Objeto*



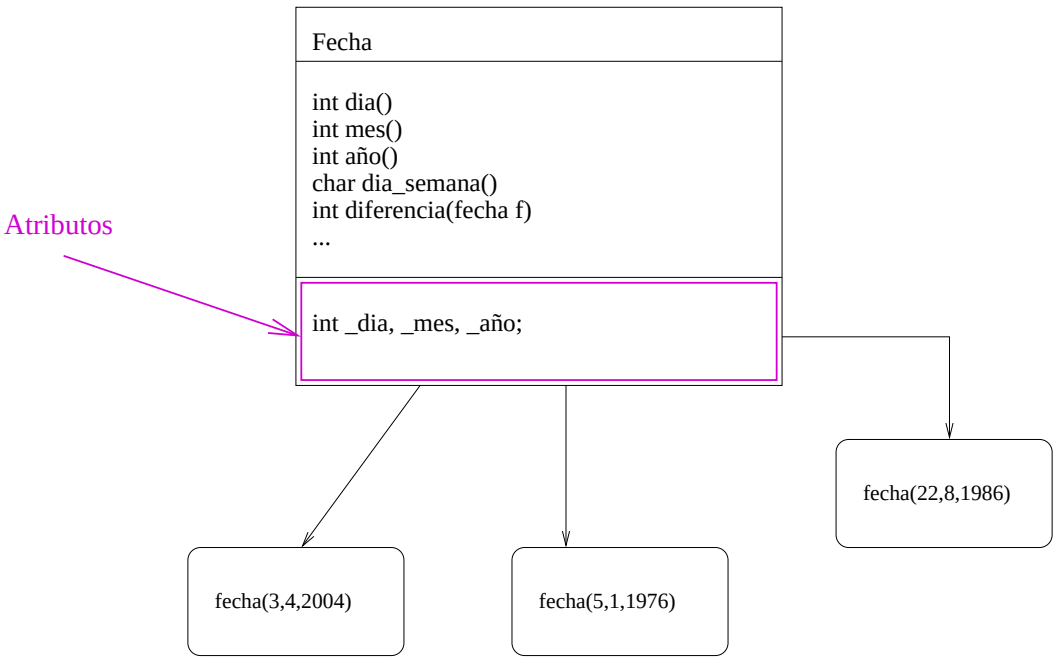
Concepto de *Clase* y de *Objeto*



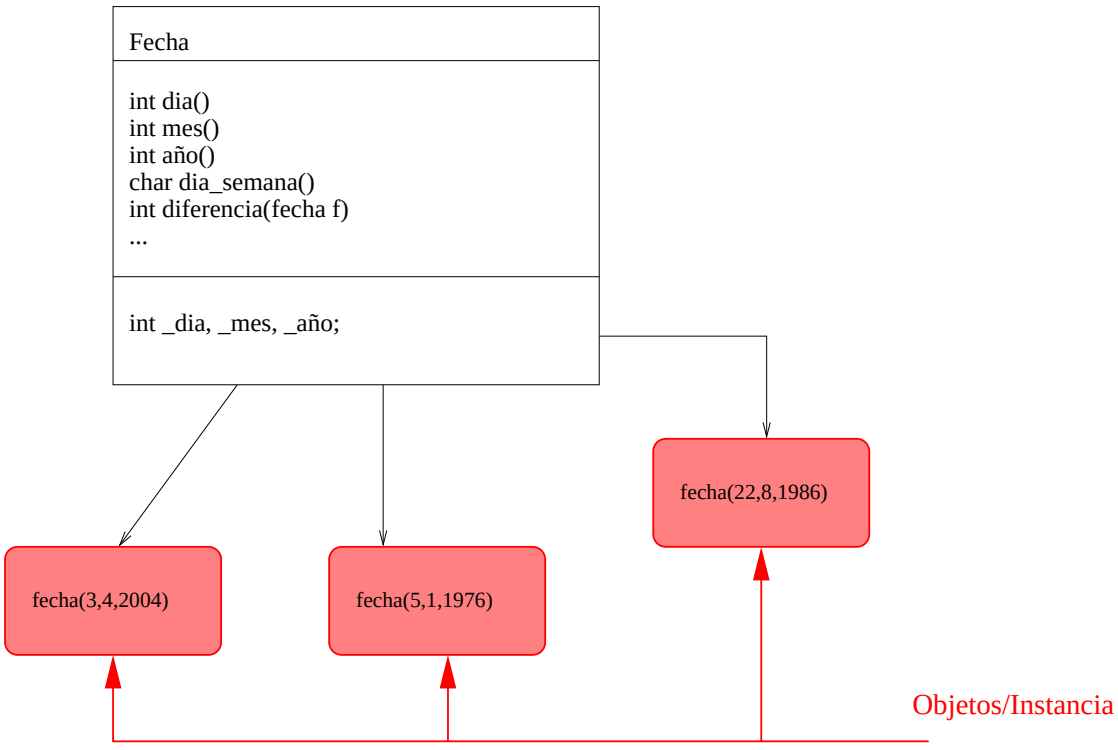
Concepto de *Clase* y de *Objeto*



Concepto de *Clase* y de *Objeto*



Concepto de *Clase* y de *Objeto*



Concepto de *Clase* y de *Objeto*

Ejemplo de Definición de Clase

```
class IntCell {
   public :
      IntCell()  { storedValue = 0; }

      IntCell(int initialValue)
      { storedValue = initialValue; }

      int read()  { return storedValue; }

      void write(int x)  { storedValue = x; }
   private :
      int storedValue;
};
```

Concepto de *Clase* y de *Objeto*

La clase `IntCell` es una representación del TAD `CELDA`.

- La definición de `IntCell` está formada por

Concepto de *Clase* y de *Objeto*

La clase `IntCell` es una representación del TAD `CELDA`.

- La definición de `IntCell` está formada por
  - Un atributo `storedValue` de tipo entero

Concepto de *Clase* y de *Objeto*

La clase `IntCell` es una representación del TAD `CELDA`.

- La definición de `IntCell` está formada por
  - Un atributo `storedValue` de tipo entero
  - Cuatro métodos: `read`, `write` y los otros dos son constructores.

Concepto de *Clase* y de *Objeto*

La clase `IntCell` es una representación del TAD `CELDA`.

- La definición de `IntCell` está formada por
  - Un atributo `storedValue` de tipo entero
  - Cuatro métodos: `read`, `write` y los otros dos son constructores.
- El cuerpo o body de la clase tiene dos partes, bajo las etiquetas `public` y `private`.

Componentes Públicos y Privados

- Las etiquetas `public` y `private` determinan la visibilidad de los componentes de la clase. En el ejemplo, todo, excepto el atributo `storedValue`, es público, mientras que `storedValue` es privado.

Componentes Públicos y Privados

- Las etiquetas `public` y `private` determinan la visibilidad de los componentes de la clase. En el ejemplo, todo, excepto el atributo `storedValue`, es público, mientras que `storedValue` es privado.
- Un componente que es público puede ser accedido por cualquier método de cualquier clase.

Componentes Públicos y Privados

- Las etiquetas `public` y `private` determinan la visibilidad de los componentes de la clase. En el ejemplo, todo, excepto el atributo `storedValue`, es público, mientras que `storedValue` es privado.
- Un componente que es público puede ser accedido por cualquier método de cualquier clase.
- Un componente privado sólo puede ser accedido desde la misma clase.

Componentes Públicos y Privados

- Las etiquetas `public` y `private` determinan la visibilidad de los componentes de la clase. En el ejemplo, todo, excepto el atributo `storedValue`, es público, mientras que `storedValue` es privado.
- Un componente que es público puede ser accedido por cualquier método de cualquier clase.
- Un componente privado sólo puede ser accedido desde la misma clase.
- Normalmente los atributos son declarados como privados. Los métodos de uso general se declaran como públicos.

Componentes Públicos y Privados

- Mediante la declaración privada de los atributos, se garantiza que se hace buen uso de los objetos, manteniendo la coherencia de la información.

Componentes Públicos y Privados

- Mediante la declaración privada de los atributos, se garantiza que se hace buen uso de los objetos, manteniendo la coherencia de la información.
- Al usuario le es suficiente con saber cómo comunicarse con un objeto, pero no tiene porqué conocer el funcionamiento interno del mismo.

Componentes Públicos y Privados

- Mediante la declaración privada de los atributos, se garantiza que se hace buen uso de los objetos, manteniendo la coherencia de la información.
- Al usuario le es suficiente con saber cómo comunicarse con un objeto, pero no tiene porqué conocer el funcionamiento interno del mismo.
- Podemos cambiar la representación interna de un objeto, sin que estos cambios afecten a otras partes del programa que utilizen el objeto. Esto es así porque el acceso al objeto se realiza a través de los métodos públicos y bastará que éstos sigan cumpliendo su especificación.

Métodos

- Un método constructor describe cómo se construye un objeto (instancia).

Métodos

- Un método constructor describe cómo se construye un objeto (instancia).
- Un método que examina, pero que no cambia, el estado de su objeto asociado es un consultor.

Métodos

- Un método constructor describe cómo se construye un objeto (instancia).
- Un método que examina, pero que no cambia, el estado de su objeto asociado es un consultor.
- Por el contrario, un método que cambia el estado de su objeto asociado es un modificador.

Métodos

- Un método constructor describe cómo se construye un objeto (instancia).
- Un método que examina, pero que no cambia, el estado de su objeto asociado es un consultor.
- Por el contrario, un método que cambia el estado de su objeto asociado es un modificador.
- En la clase `IntCell` que se presenta a continuación se han introducido cuatro cambios respecto a la definición dada anteriormente.

Otros Conceptos

```
class IntCell {
  public :
     explicit IntCell(int initialValue = 0)
     : storedValue(initialValue) {}
     int read() const
     { return storedValue; }
     void write(int x)
     { storedValue = x; }
  private :
     int storedValue;
};
```

Otros Conceptos

```
class IntCell {
  public :
     explicit IntCell(int initialValue = 0)
     : storedValue(initialValue) {}
     int read() const
     { return storedValue; }
     void write(int x)
     { storedValue = x; }
  private :
     int storedValue;
};
```

Otros Conceptos

A) Parámetros por defecto: el constructor `IntCell` es un ejemplo de la utilización de parámetros por defecto.

- Si no se especifica ningún parámetro el constructor `IntCell` utilizará el valor 0 para inicializar `storedValue`.

```
IntCell m; // crea un objeto IntCell llamado m
           // y lo inicializa a 0
```

- En caso contrario el parámetro `initialValue` se utilizará para inicializar.

```
IntCell m(7); // crea un objeto IntCell llamado m
              // y lo inicializa a 7
```

Otros Conceptos

```
class IntCell {
  public :
     explicit IntCell(int initialValue = 0)
     : storedValue(initialValue) {}
     int read() const
     { return storedValue; }
     void write(int x)
     { storedValue = x; }
  private :
     int storedValue;
};
```

Otros Conceptos

```
class IntCell {
  public :
     explicit IntCell(int initialValue = 0)
     : storedValue(initialValue) {}
     int read() const
     { return storedValue; }
     void write(int x)
     { storedValue = x; }
  private :
     int storedValue;
};
```

Otros Conceptos

B) Lista de inicialización: Se utiliza para inicializar los atributos. La utilización de listas de inicialización en lugar de sentencias de asignación ahorra tiempo si son clases que tienen inicializaciones complejas. Hay situaciones en las que es imprescindible usar listas de inicialización (para más detalles consultad la *Guía de Programación*).

Otros Conceptos

```
class IntCell {
  public :
     explicit IntCell(int initialValue = 0)
     : storedValue(initialValue) {}
     int read() const
     { return storedValue; }
     void write(int x)
     { storedValue = x; }
  private :
     int storedValue;
};
```

Otros Conceptos

```
class IntCell {
  public :
     explicit IntCell(int initialValue = 0)
     : storedValue(initialValue) {}
     int read() const
     { return storedValue; }
     void write(int x)
     { storedValue = x; }
  private :
     int storedValue;
};
```

Otros Conceptos

C) Constructor explícito: El constructor `IntCell` es explícito, lo que significa que no se pueden aplicar ciertas reglas que permitirían conversiones de tipo indeseables.

```
IntCell obj;   // obj es una IntCell obj = 37;
               // tipo no corresponde
```

- La asignación no debería funcionar, dado que la expresión de la parte derecha, no es una `IntCell`. Se debería utilizar el método `write` de la clase para asignarle el valor 37 a `obj`.
- Si `IntCell` se ha declarado como explícito el compilador dará error ya que los tipos no coinciden. Pero si no se hubiera declarado como explícito entonces el compilador no protestaría y la asignación (usando una conversión) se haría.

Otros Conceptos

```
class IntCell {
  public :
     explicit IntCell(int initialValue = 0)
     : storedValue(initialValue) {}
     int read() const
     { return storedValue; }
     void write(int x)
     { storedValue = x; }
  private :
     int storedValue;
};
```

Otros Conceptos

```
class IntCell {
  public :
     explicit IntCell(int initialValue = 0)
     : storedValue(initialValue) {}
     int read() const
     { return storedValue; }
     void write(int x)
     { storedValue = x; }
  private :
     int storedValue;
};
```

Otros Conceptos

D) Método Consultor:

- Es posible marcar cada método como consultor o modificador. Hacer este marcaje es una parte importante del proceso de diseño y comporta consecuencias semánticas.
- Por defecto todos los métodos son modificadores. Para marcar un método como consultor se debe añadir la palabra `const` en la cabecera del método.
- Si un método es etiquetado como consultor pero su implementación cambia el valor de algún atributo el compilador generará un error.
- En la clase `IntCell`, el método `read` es claramente un consultor y el método `write` un modificador.

Definición e Implementación

- El usuario de una clase no necesita saber cómo está implementada la clase. Basta que sepa cuáles son los métodos públicos disponibles, qué parámetros tienen, para qué sirven. Por eso es frecuente separar la parte de definición de la implementación.

Definición e Implementación

- El usuario de una clase no necesita saber cómo está implementada la clase. Basta que sepa cuáles son los métodos públicos disponibles, qué parámetros tienen, para qué sirven. Por eso es frecuente separar la parte de definición de la implementación.
- Para ello se escribe la definición de una clase en un fichero con extensión `.hpp` (header). El fichero `IntCell.hpp` se muestra en la siguiente página.

Definición e Implementación

- El usuario de una clase no necesita saber cómo está implementada la clase. Basta que sepa cuáles son los métodos públicos disponibles, qué parámetros tienen, para qué sirven. Por eso es frecuente separar la parte de definición de la implementación.
- Para ello se escribe la definición de una clase en un fichero con extensión `.hpp` (header). El fichero `IntCell.hpp` se muestra en la siguiente página.
- La separación entre definición e implementación tiene la ventaja adicional de que permite compilar por separado el fichero que contiene la implementación de los métodos de una clase y los ficheros que usan a la clase. Si cambiamos la implementación de un cierto método, no será necesario recompilar el código que usa a ese método.

Definición e Implementación

IntCell.hpp

```
#ifndef _IntCell_H_
#define _IntCell_H_

class IntCell {
  public :
     explicit IntCell(int initialValue = 0);
     int read() const;
     void write(int x);

  private :
     int storedValue;
};
#endif
```

Definición e Implementación

- En C++ sólo se puede definir una cosa una vez. Imaginad que `IntCell` es usada para definir una clase `A` y otra clase `B` y que tenemos un programa que usa a esas dos clases.

A.hpp

```
#include "IntCell.hpp"
...
```

B.hpp

```
#include "IntCell.hpp"
...
```

prog.cpp

```
#include "A.hpp"
#include "B.hpp"
...
```

Definición e Implementación

- Entonces nuestro programa contendría dos veces la definición de `IntCell`. Para evitarlo se usa la condicional (`#ifndef` ... `#endif`) del preprocesador de C++; la primera vez sí se incluye la definición de `IntCell` porque el símbolo `_IntCell_H_` no existe, pero la segunda vez **no** se incluye porque `_IntCell_H_` queda definido tras la primera inclusión.

Definición e Implementación

- Para conseguir una ocultación de la representación de la clase es preciso utilizar técnicas más sofisticadas (se explican en la *Guía de Programación*).
- La implementación de una clase será almacenada en un fichero con extensión `.cpp`.

IntCell.cpp

```
#include "IntCell.hpp"

IntCell::IntCell(int initialValue) :
         storedValue(initialValue) {}
int IntCell::read() const { return storedValue; }
void IntCell::write(int x) { storedValue = x; }
```

Definición e Implementación

- Se ha de identificar la clase a la que pertenece cada método. Para ello se utiliza el operador de resolución `::` (scoping resolution).

Definición e Implementación

- Se ha de identificar la clase a la que pertenece cada método. Para ello se utiliza el operador de resolución `::` (scoping resolution).
- La cabecera de cada método del fichero `.cpp` debe coincidir con la del fichero de definición, excepto que los parámetros por defecto están especificados en el fichero `.hpp` solamente y son omitidos en la implementación.

Uso de Clases

TestIntCell.cpp

```
#include <iostream>
#include "IntCell.hpp"

int main() {
    IntCell m;
    m.write(5);
    cout << "Valor: " << m.read() << endl;
    return 0; // el programa termina con exito
}
```

Uso de Clases

- La ejecución de un programa empieza siempre invocando a la función `main()`. Desde ésta se puede invocar a otras funciones y métodos declarados en los ficheros incluidos (mediante `#include`).

Uso de Clases

- La ejecución de un programa empieza siempre invocando a la función `main()`. Desde ésta se puede invocar a otras funciones y métodos declarados en los ficheros incluidos (mediante `#include`).
- En el ejemplo se crea un objeto m de tipo `IntCell` inicializado a 0, se modifica su contenido a 5, lo consulta y lo imprime.

Uso de Clases

- La ejecución de un programa empieza siempre invocando a la función `main()`. Desde ésta se puede invocar a otras funciones y métodos declarados en los ficheros incluidos (mediante `#include`).
- En el ejemplo se crea un objeto m de tipo `IntCell` inicializado a 0, se modifica su contenido a 5, lo consulta y lo imprime.
- Para aplicar un método a un objeto, sea para consultarlo, sea para modificarlo, se utiliza el operador de selección. El operador de selección nos permite acceder a un atributo o activar un método de un objeto. En el ejemplo activamos los métodos `write` y `read` del objeto `m` mediante `m.write(5)` y `m.read()`, respectivamente.

Uso de Clases

- C++ provee la librería estandar de entrada/salida llamada `iostream` con los siguientes objetos de tipo `stream` y operadores sobre dichos objetos:

Uso de Clases

- C++ provee la librería estandar de entrada/salida llamada `iostream` con los siguientes objetos de tipo `stream` y operadores sobre dichos objetos:
  - `cout`: salida estándar, normalmente pantalla

Uso de Clases

- C++ provee la librería estandar de entrada/salida llamada `iostream` con los siguientes objetos de tipo `stream` y operadores sobre dichos objetos:
  - `cout`: salida estándar, normalmente pantalla
  - `cin`: entrada estándar, normalmente teclado

Uso de Clases

- C++ provee la librería estandar de entrada/salida llamada `iostream` con los siguientes objetos de tipo `stream` y operadores sobre dichos objetos:
  - `cout`: salida estándar, normalmente pantalla
  - `cin`: entrada estándar, normalmente teclado
  - `cerr`: error estándar, normalmente pantalla

Uso de Clases

- C++ provee la librería estandar de entrada/salida llamada `iostream` con los siguientes objetos de tipo `stream` y operadores sobre dichos objetos:
  - `cout`: salida estándar, normalmente pantalla
  - `cin`: entrada estándar, normalmente teclado
  - `cerr`: error estándar, normalmente pantalla
  - <<: escribir en el `stream` de salida

Uso de Clases

- C++ provee la librería estandar de entrada/salida llamada `iostream` con los siguientes objetos de tipo `stream` y operadores sobre dichos objetos:
  - `cout`: salida estándar, normalmente pantalla
  - `cin`: entrada estándar, normalmente teclado
  - `cerr`: error estándar, normalmente pantalla
  - <<: escribir en el `stream` de salida
  - >>: leer del `stream` de entrada

Uso de Clases

- C++ provee la librería estandar de entrada/salida llamada `iostream` con los siguientes objetos de tipo `stream` y operadores sobre dichos objetos:
  - `cout`: salida estándar, normalmente pantalla
  - `cin`: entrada estándar, normalmente teclado
  - `cerr`: error estándar, normalmente pantalla
  - <<: escribir en el `stream` de salida
  - >>: leer del `stream` de entrada
- El identificador `endl` añade un salto de línea al `stream` e imprime el `stream`.

Sobrecarga de Funciones

- Es una buena idea dar nombres diferentes a funciones que realizan diferentes tareas. Cuando algunas funciones realizan conceptualmente la misma tarea en objetos de diferentes tipos, es conveniente asociarles el mismo nombre.

Sobrecarga de Funciones

- Es una buena idea dar nombres diferentes a funciones que realizan diferentes tareas. Cuando algunas funciones realizan conceptualmente la misma tarea en objetos de diferentes tipos, es conveniente asociarles el mismo nombre.
- El uso del mismo nombre para funciones sobre diferentes tipos se denomina sobrecarga.

Sobrecarga de Funciones

- Es una buena idea dar nombres diferentes a funciones que realizan diferentes tareas. Cuando algunas funciones realizan conceptualmente la misma tarea en objetos de diferentes tipos, es conveniente asociarles el mismo nombre.
- El uso del mismo nombre para funciones sobre diferentes tipos se denomina sobrecarga.
- Por ejemplo, existe un único nombre para la suma $+$, aunque puede ser utilizada para sumar enteros, o reales.

Sobrecarga de Funciones

- Es una buena idea dar nombres diferentes a funciones que realizan diferentes tareas. Cuando algunas funciones realizan conceptualmente la misma tarea en objetos de diferentes tipos, es conveniente asociarles el mismo nombre.
- El uso del mismo nombre para funciones sobre diferentes tipos se denomina sobrecarga.
- Por ejemplo, existe un único nombre para la suma $+$, aunque puede ser utilizada para sumar enteros, o reales.
- Desde el punto de vista del compilador lo único que tienen en común las funciones sobrecargadas es el nombre. Cuando una función `f` es llamada, el compilador debe descubrir cuál de las funciones con el nombre `f` ha sido invocada.

Sobrecarga de Funciones

- Para averiguar qué función es la que se corresponde mejor con los argumentos actuales existen una serie de criterios que se aplican en serie:

Sobrecarga de Funciones

- Para averiguar qué función es la que se corresponde mejor con los argumentos actuales existen una serie de criterios que se aplican en serie:
  A) Correspondencia exacta: se da cuando no hay conversiones o bien son conversiones triviales: por ejemplo, del tipo T al "tipo" const T (ver la sesión **??**)

Sobrecarga de Funciones

- Para averiguar qué función es la que se corresponde mejor con los argumentos actuales existen una serie de criterios que se aplican en serie:
  A) Correspondencia exacta: se da cuando no hay conversiones o bien son conversiones triviales: por ejemplo, del tipo T al "tipo" const T (ver la sesión **??**)
  B) Correspondencia utilizando promociones: por ejemplo, `bool` a `int`, `char` a `int`, `float` a `double`.

Sobrecarga de Funciones

- Para averiguar qué función es la que se corresponde mejor con los argumentos actuales existen una serie de criterios que se aplican en serie:
  A) Correspondencia exacta: se da cuando no hay conversiones o bien son conversiones triviales: por ejemplo, del tipo T al "tipo" const T (ver la sesión **??**)
  B) Correspondencia utilizando promociones: por ejemplo, `bool` a `int`, `char` a `int`, `float` a `double`.
  C) Correspondencia utilizando conversiones estándar: por ejemplo, `int` a `double`, `double` a `int`.

Sobrecarga de Funciones

- Para averiguar qué función es la que se corresponde mejor con los argumentos actuales existen una serie de criterios que se aplican en serie:
  A) Correspondencia exacta: se da cuando no hay conversiones o bien son conversiones triviales: por ejemplo, del tipo T al "tipo" const T (ver la sesión **??**)
  B) Correspondencia utilizando promociones: por ejemplo, `bool` a `int`, `char` a `int`, `float` a `double`.
  C) Correspondencia utilizando conversiones estándar: por ejemplo, `int` a `double`, `double` a `int`.
  D) Correspondencia utilizando conversiones explícitas definidas por el usuario.

Sobrecarga de Funciones

- Para averiguar qué función es la que se corresponde mejor con los argumentos actuales existen una serie de criterios que se aplican en serie:
  A) Correspondencia exacta: se da cuando no hay conversiones o bien son conversiones triviales: por ejemplo, del tipo T al "tipo" const T (ver la sesión **??**)
  B) Correspondencia utilizando promociones: por ejemplo, `bool` a `int`, `char` a `int`, `float` a `double`.
  C) Correspondencia utilizando conversiones estándar: por ejemplo, `int` a `double`, `double` a `int`.
  D) Correspondencia utilizando conversiones explícitas definidas por el usuario.
  E) Correspondencia utilizando elipsis ... en la declaración de la función.

Sobrecarga de Funciones

- Para averiguar qué función es la que se corresponde mejor con los argumentos actuales existen una serie de criterios que se aplican en serie:
  A) Correspondencia exacta: se da cuando no hay conversiones o bien son conversiones triviales: por ejemplo, del tipo T al "tipo" const T (ver la sesión **??**)
  B) Correspondencia utilizando promociones: por ejemplo, `bool` a `int`, `char` a `int`, `float` a `double`.
  C) Correspondencia utilizando conversiones estándar: por ejemplo, `int` a `double`, `double` a `int`.
  D) Correspondencia utilizando conversiones explícitas definidas por el usuario.
  E) Correspondencia utilizando elipsis ... en la declaración de la función.
- Si se encuentran dos correspondencias de la misma prioridad, la llamada es rechazada por ambigua.

Sobrecarga de Funciones

Ejemplo de Sobrecarga

```
void print(int); void print(double); void print(long); void
print(char);

void h(char c, int i, short s, float f); {
   print(c);   //(A) llamada a print(char)
   print(i);   //(A) llamada a print(int)
   print(s);   //(B) llamada a print(int)
   print(f);   //(B) llamada a print(double)

   print('a'); //(A) llamada a print(char)
   print(49);  //(A) llamada a print(int)
   print(0);   //(A) llamada a print(int)
}
```

Sobrecarga de Funciones

- La razón para distinguir entre conversiones y promociones es que son preferibles promociones seguras (tales como `char` a `int`) que conversiones inseguras (como por ejemplo `int` a `char`).

Sobrecarga de Funciones

- La razón para distinguir entre conversiones y promociones es que son preferibles promociones seguras (tales como `char` a `int`) que conversiones inseguras (como por ejemplo `int` a `char`).
- La resolución de sobrecarga es independiente del orden de la declaración de las funciones consideradas.

Sobrecarga de Funciones

- La razón para distinguir entre conversiones y promociones es que son preferibles promociones seguras (tales como `char` a `int`) que conversiones inseguras (como por ejemplo `int` a `char`).
- La resolución de sobrecarga es independiente del orden de la declaración de las funciones consideradas.
- El tipo de resultado de la función no se utiliza en la resolución de la sobrecarga.

Sobrecarga de Funciones

- La llamada a `r.f` (donde `r` es un objeto de la clase `X`) invoca al método `f` del objeto `r`, aunque haya métodos o funciones llamadas `f` en otras clases o externas a cualquier clase. Si la clase `X` definiese varios métodos `f` se aplican las reglas vistas anteriormente.

Sobrecarga de Funciones

- La llamada a `r.f` (donde `r` es un objeto de la clase `X`) invoca al método `f` del objeto `r`, aunque haya métodos o funciones llamadas `f` en otras clases o externas a cualquier clase. Si la clase `X` definiese varios métodos `f` se aplican las reglas vistas anteriormente.
- La sobrecarga se basa en una serie de reglas relativamente complicadas. Conviene usar la sobrecarga con prudencia.

Sobrecarga de Funciones

- La alternativa a la sobrecarga consiste en definir varias funciones (que realizan la misma tarea) con nombres distintos:

```
void print_int(int);
void print_char(char);

void g(int i, char c, double d);
{
    print_int(i);  //ok
    print_char(c); //ok

    print_int(c);  //char -> int
    print_char(i); //peligro! int -> char
    print_int(d);  //peligro! double -> int
}
```

Sobrecarga de Funciones

- Si comparamos con la función sobrecargada `print`, en este segundo caso tendremos que recordar los diferentes nombres y también cuando debemos utilizar cada uno de ellos.

Sobrecarga de Funciones

- Si comparamos con la función sobrecargada `print`, en este segundo caso tendremos que recordar los diferentes nombres y también cuando debemos utilizar cada uno de ellos.
- Esto puede resultar tedioso y nos hace fracasar en el intento de hacer programación genérica. Dado que no usamos sobrecarga, algunas conversiones aplicadas a los argumentos son estándar y éstas pueden producir errores.

Sobrecarga de Funciones

- Si comparamos con la función sobrecargada `print`, en este segundo caso tendremos que recordar los diferentes nombres y también cuando debemos utilizar cada uno de ellos.
- Esto puede resultar tedioso y nos hace fracasar en el intento de hacer programación genérica. Dado que no usamos sobrecarga, algunas conversiones aplicadas a los argumentos son estándar y éstas pueden producir errores.
- Hay que tener cuidado con la sobrecarga dentro de una misma clase y la sobrecarga de funciones externas a clases. En cambio, puede ser útil el hecho de que haya métodos con el mismo nombre en clases distintas y eso no origina problemas; por ejemplo: `print`, `+`, ...

Sobrecarga de Operadores

- Los operadores, al igual que las funciones, pueden ser sobrecargados (overloaded).

Sobrecarga de Operadores

- Los operadores, al igual que las funciones, pueden ser sobrecargados (overloaded).
- La sobrecarga de operadores quiere decir que se pueden redefinir algunos de los operadores existentes para que actúen de la manera definida por el programador.

Sobrecarga de Operadores

- Los operadores, al igual que las funciones, pueden ser sobrecargados (overloaded).
- La sobrecarga de operadores quiere decir que se pueden redefinir algunos de los operadores existentes para que actúen de la manera definida por el programador.
- Todos los operadores aritméticos/lógicos se pueden sobrecargar.

Sobrecarga de Operadores

Operadores que se pueden sobrecargar

`+`	Suma	`++`	Incremento
`-`	Resta	`--`	Decremento
`*`	Multiplicación	`[]`	Indexación
`/`	División	`&`	Dirección (unario)
`%`	Módulo	`*`	Dereferencia (unario)
`<=`	Menor o igual que	`->`	Selección por puntero
`>=`	Mayor o igual que	`||`	Disyunción (OR lógico)
`>`	Mayor que	`&&`	Conjunción (AND lógico)
`<`	Menor que	`!`	Negación (NOT lógico)
`==`	Igual	`<<`	Shift izq.
`!=`	Diferente	`>>`	Shift der.
`=`	Asignación		

Sobrecarga de Operadores

- Algunos operadores **no** se pueden sobrecargar:

Sobrecarga de Operadores

- Algunos operadores **no** se pueden sobrecargar:
  - el operador de selección (`.`)

Sobrecarga de Operadores

- Algunos operadores **no** se pueden sobrecargar:
  - el operador de selección (`.`)
  - el `if` aritmético (`?:`)

Sobrecarga de Operadores

- Algunos operadores **no** se pueden sobrecargar:
  - el operador de selección (`.`)
  - el `if` aritmético (`?:`)
  - el operador `sizeof`

Sobrecarga de Operadores

- Algunos operadores **no** se pueden sobrecargar:
  - el operador de selección (`.`)
  - el `if` aritmético (`?:`)
  - el operador `sizeof`
  - el operador de resolución de ámbito (`::`)

Sobrecarga de Operadores

- El objetivo es simplificar al máximo el código del usuario, a cambio de complicar algo la definición de las clases. Una clase que disponga de operadores sobrecargados es una clase más compleja de definir, pero más sencilla e intuitiva de utilizar.

Sobrecarga de Operadores

- El objetivo es simplificar al máximo el código del usuario, a cambio de complicar algo la definición de las clases. Una clase que disponga de operadores sobrecargados es una clase más compleja de definir, pero más sencilla e intuitiva de utilizar.
- La sobrecarga de operadores tiene limitaciones:

Sobrecarga de Operadores

- El objetivo es simplificar al máximo el código del usuario, a cambio de complicar algo la definición de las clases. Una clase que disponga de operadores sobrecargados es una clase más compleja de definir, pero más sencilla e intuitiva de utilizar.
- La sobrecarga de operadores tiene limitaciones:
  - Se puede modificar la definición de un operador pero no su sintaxis, es decir, el número de operandos sobre los que actúa, la precedencia y la asociatividad.

Sobrecarga de Operadores

- El objetivo es simplificar al máximo el código del usuario, a cambio de complicar algo la definición de las clases. Una clase que disponga de operadores sobrecargados es una clase más compleja de definir, pero más sencilla e intuitiva de utilizar.
- La sobrecarga de operadores tiene limitaciones:
  - Se puede modificar la definición de un operador pero no su sintaxis, es decir, el número de operandos sobre los que actúa, la precedencia y la asociatividad.
  - Si el operador es un método de una clase, es necesario que el operando de la izquierda sea un objeto de la clase en la que se ha definido el operador sobrecargado.

Sobrecarga de Operadores

- El objetivo es simplificar al máximo el código del usuario, a cambio de complicar algo la definición de las clases. Una clase que disponga de operadores sobrecargados es una clase más compleja de definir, pero más sencilla e intuitiva de utilizar.
- La sobrecarga de operadores tiene limitaciones:
  - Se puede modificar la definición de un operador pero no su sintaxis, es decir, el número de operandos sobre los que actúa, la precedencia y la asociatividad.
  - Si el operador es un método de una clase, es necesario que el operando de la izquierda sea un objeto de la clase en la que se ha definido el operador sobrecargado.
- La redefinición de los operadores se hace anteponiendo la palabra `operator` al símbolo del operador.

Sobrecarga de Operadores

complex.hpp

```
class complex {
   public:
      complex(double r, double i);
      complex operator+(complex otro) const;
      complex operator*(complex otro) const;
      ...
   private:
      double _re, _im;
};
```

complex.cpp

```
complex::complex(double r, double i) : _re(r), _im(i) {}
...
```

Sobrecarga de Operadores

- Un complejo está representado por un par de números reales en coma flotante de doble precisión.
- Los operadores `complex::operator+` y `complex ::operator*` se implementan en el fichero `.cpp` correspondiente.
- Ejemplo de uso:

```
#include "complex.hpp"
void f() {
   complex a = complex (1, 3.1); // a = 1 + 3.1 i
   complex b = complex (1.2, 2); // b = 1.2 + 2 i
   complex c = b;
   a = b + c;
   c = a * b + complex(1, 2);    // c = a * b + (1 + 2 i)
}
```

- El compilador transforma las expresiones tales como `b + c` en `b.operator+(c)`

Sobrecarga de Operadores

complex.cpp

```
... complex complex::operator+(complex otro) const {
   complex suma;
   suma._re = _re + otro._re;
   suma._im = _im + otro._im;
   return suma;
}
```

Notad que cuando en el código previo se pone `_re` o `_im`, sin más, se refiere a los atributos del objeto al que se aplica el método. El objeto sobre el que se aplica el método es un parámetro implícito de la operación.

Atributos y Métodos de Clase

- Hemos visto que los atributos y métodos de una clase están asociados a los objetos de la clase. Cada objeto tiene los mismos atributos y cada método opera sobre el objeto al que está asociado.

Atributos y Métodos de Clase

- Hemos visto que los atributos y métodos de una clase están asociados a los objetos de la clase. Cada objeto tiene los mismos atributos y cada método opera sobre el objeto al que está asociado.
- Pero también podemos tener atributos y métodos de clase. No están asociados a ningún objeto de la clase.

Atributos y Métodos de Clase

- Hemos visto que los atributos y métodos de una clase están asociados a los objetos de la clase. Cada objeto tiene los mismos atributos y cada método opera sobre el objeto al que está asociado.
- Pero también podemos tener atributos y métodos de clase. No están asociados a ningún objeto de la clase.
- Si un atributo es de clase entonces sólo existe una vez y su valor es "compartido" por todos los objetos de la clase.

Atributos y Métodos de Clase

- Hemos visto que los atributos y métodos de una clase están asociados a los objetos de la clase. Cada objeto tiene los mismos atributos y cada método opera sobre el objeto al que está asociado.
- Pero también podemos tener atributos y métodos de clase. No están asociados a ningún objeto de la clase.
- Si un atributo es de clase entonces sólo existe una vez y su valor es "compartido" por todos los objetos de la clase.
- Los métodos de clase sólo pueden actuar sobre los atributos de clase o sus parámetros explícitos.

Atributos y Métodos de Clase

- Los atributos y métodos de clase pueden ser públicos o privados, y las reglas de visibilidad son idénticas: por ejemplo, un atributo de clase privado sólo podrá ser accedido por los métodos (de clase o de objeto, públicos o privados) de la clase y sólo por éstos.
- Un atributo o método de clase se distingue porque se antepone la palabra reservada `static` en su definición:

```
class X {
   public :
      static int x;           // atributo de clase publico
      static void f(...);  // metodo de clase publico
   private :
      static char y;          // atributo de clase privado
      static void g(...);  // metodo de clase privado
};
```

Atributos y Métodos de Clase

- Para usar un método de clase público se usa la sintaxis: `X::f(...)`, es decir, se pone el nombre de la clase y el operador de resolución, y no se usa el operador de selección ya que no hay ningún objeto al que esté asociado el método. De manera parecida para inicializar o modificar un atributo público de clase se utiliza el nombre de la clase y el operador `::` antes del identificador.

Atributos y Métodos de Clase

- Para usar un método de clase público se usa la sintaxis: `X::f(...)`, es decir, se pone el nombre de la clase y el operador de resolución, y no se usa el operador de selección ya que no hay ningún objeto al que esté asociado el método. De manera parecida para inicializar o modificar un atributo público de clase se utiliza el nombre de la clase y el operador `::` antes del identificador.
- Los atributos de clase son básicamente variables o constantes globales de la clase, y los métodos de clase permiten consultar o modificar dichas variables.

Atributos y Métodos de Clase

- Es frecuente utilizar atributos de clase públicos para definir constantes "globales" (para lo cual se usa el cualificador `const`):

```
class X {
  public:
     static const int MAXSIZE;
     static const char MARCA_FIN;
  ...
};
```

```
...
if (n >= X::MAXSIZE) ...
while (c != X::MARCA_FIN) {
 ...
}
```

Atributos y Métodos de Clase

- Los atributos de clase públicos deben inicializarse externamente a cualquier función (se puede hacer por ejemplo al inicio del fichero `.cpp` que implementa la clase). Si el atributo es de tipo entero, carácter o similar se puede hacer la inicialización dentro de la definición de la clase.

Atributos y Métodos de Clase

- Los atributos de clase públicos deben inicializarse externamente a cualquier función (se puede hacer por ejemplo al inicio del fichero `.cpp` que implementa la clase). Si el atributo es de tipo entero, carácter o similar se puede hacer la inicialización dentro de la definición de la clase.
- Los métodos de clase privados pueden tener otros usos, ya que aunque no están asociados a un objeto, sí tienen acceso a la parte privada de un objeto de su clase, si se les pasa el objeto como parámetro explícito (ver la siguiente sección sobre operaciones privadas).

Operaciones Privadas

Distinguiremos tres tipos de operaciones privadas:

- Métodos privados: Permiten modificar o consultar el objeto al cual están asociados. Pero sólo pueden ser utilizados por otros métodos (públicos o privados) de la clase.

Operaciones Privadas

Distinguiremos tres tipos de operaciones privadas:

- Métodos privados: Permiten modificar o consultar el objeto al cual están asociados. Pero sólo pueden ser utilizados por otros métodos (públicos o privados) de la clase.
- Métodos privados de la clase: Están asociados a una clase, pero no a los objetos de la clase. Tienen acceso a la representación de la clase y pueden manipular un objeto si lo reciben como parámetro explícito.

Operaciones Privadas

Distinguiremos tres tipos de operaciones privadas:

- Métodos privados: Permiten modificar o consultar el objeto al cual están asociados. Pero sólo pueden ser utilizados por otros métodos (públicos o privados) de la clase.
- Métodos privados de la clase: Están asociados a una clase, pero no a los objetos de la clase. Tienen acceso a la representación de la clase y pueden manipular un objeto si lo reciben como parámetro explícito.
- Funciones externas: No están asociadas a ningún objeto o clase y no tienen acceso a la parte privada de ninguna clase. Son "privadas" ya que nada más pueden ser usadas en la unidad de compilación en la que se definen e implementan.

Operaciones Privadas

Tomemos la clase `complex` como ejemplo. Podríamos necesitar un método privado que calcule el complejo conjugado de uno dado. El método privado correspondiente sería:

complex.hpp

```
class complex {
 ...
private:
   double _re, _im;
   complex conjugado() const;
};
```

complex.cpp

```
complex complex::conjugado() const {
   return complex(_re, -_im);
}
```

Operaciones Privadas

También podemos resolver el problema mediante un método privado de clase, que necesariamente habrá de recibir el número complejo cuyo conjugado se quiere calcular como parámetro:

complex.hpp

```
class complex {
 ...
private:
   double _re, _im;
   static complex conjugado(complex z);
};
```

complex.cpp

```
complex complex::conjugado(complex z) {
   return complex(z._re, -z._im);
}
```

Operaciones Privadas

Para usarlo y, puesto que no está asociado a ningún objeto, escribiremos por ejemplo:

```
... complex y = complex::conjugado(z); ...
```

aunque podemos omitir el prefijo `complex::` ya que sólo se puede usar este método desde dentro de un método de la clase `complex`.

Operaciones Privadas

Las funciones externas no son accesibles fuera de la unidad de compilación donde se definen, y por ello podemos decir que son "privadas". Por ejemplo, para calcular el cociente de dos números complejos $x = a + b \cdot \mathrm{i}$ y $y = c + d \cdot \mathrm{i}$ es necesario calcular $c^2 + d^2$. El trabajo lo puede hacer una función "privada externa":

complex.cpp

```
static double denom(double c,
                    double d); // declaracion
...
complex complex::operator/(complex y) {
   double d = denom(y._re, y._im);
   double res_re = (_re * y._re + _im * y._im) / d;
   double res_im = (_im * y._re - _re * y._im) / d;
   return complex(res_re, res_im);
}
...
double denom(double c, double d) { // definicion
   return c * c + d * d;
}
```

Punteros

En tiempo de ejecución, la declaración de una variable de cierto tipo implica la reserva de un espacio de memoria para valores de dicho tipo, que empieza en una dirección concreta.

1. El operador & sobre una variable devuelve la dirección donde empieza el espacio reservado para dicha variable.

Punteros

En tiempo de ejecución, la declaración de una variable de cierto tipo implica la reserva de un espacio de memoria para valores de dicho tipo, que empieza en una dirección concreta.

1. El operador `&` sobre una variable devuelve la dirección donde empieza el espacio reservado para dicha variable.
2. La dirección de una variable de tipo `T` es de tipo `T*`.

Punteros

En tiempo de ejecución, la declaración de una variable de cierto tipo implica la reserva de un espacio de memoria para valores de dicho tipo, que empieza en una dirección concreta.

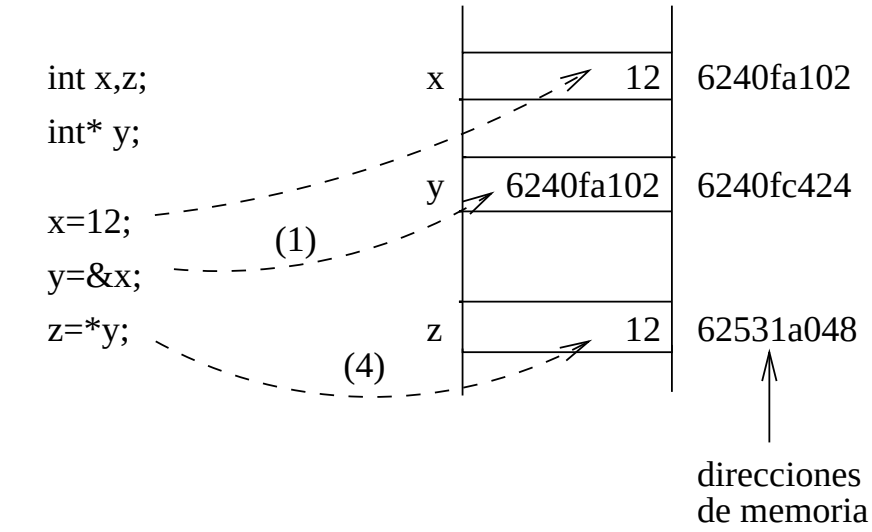
1. El operador & sobre una variable devuelve la dirección donde empieza el espacio reservado para dicha variable.
2. La dirección de una variable de tipo `T` es de tipo `T*`.
3. Una variable de tipo `T*` se denomina puntero a `T` y puede contener direcciones de variables de tipo `T`.

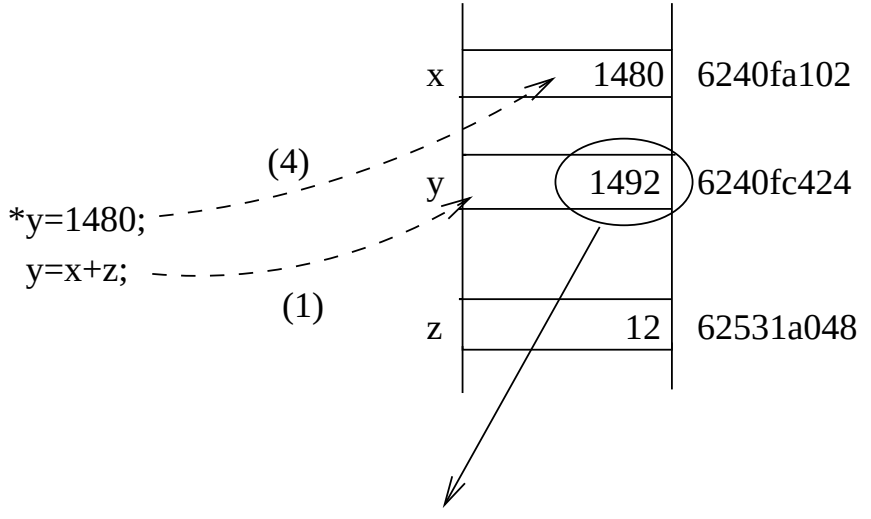
Punteros

En tiempo de ejecución, la declaración de una variable de cierto tipo implica la reserva de un espacio de memoria para valores de dicho tipo, que empieza en una dirección concreta.

1. El operador & sobre una variable devuelve la dirección donde empieza el espacio reservado para dicha variable.
2. La dirección de una variable de tipo `T` es de tipo `T*`.
3. Una variable de tipo `T*` se denomina puntero a `T` y puede contener direcciones de variables de tipo `T`.
4. El operador * aplicado a un puntero permite acceder al valor apuntado.

Punteros





VIOLACION DE ESPACIO DE MEMORIA

Ejemplo de manipulación de punteros

Punteros

- Un puntero que no apunta a ningún sitio se denomina puntero nulo y su valor es 0 (≡ `NULL`). Si `p` es un puntero nulo entonces `*p` no está definido. En general, provoca un error de ejecución inmediato (tipo *Segmentation fault*), pero dependerá del sistema. En cualquier caso es un error grave.

```
char* p; char c = 'b'; *p = 'a';
          // ERROR! p no esta inicializado,
          // puede apuntar a cualquier sitio!
p = 0;    // p es nulo
p = &c;  // p apunta a c
*p = 0;   // c contiene caracter con codigo ASCII 0
```

Punteros

- Se pueden declarar punteros genéricos del tipo `void*`. Estos pueden apuntar a cualquier cosa:

```
void* g = &x;
...
g = &y;
```

Conviene evitar su uso, ya que son una fuente de errores muy difíciles de corregir. Su uso debe quedar restringido a funciones de muy bajo nivel.

Referencias

- Una referencia es un nombre alternativo a una variable ya declarada, es decir, un alias.
- Una referencia de una variable de tipo `T` es de tipo `T&`
- Toda referencia ha de ser inicializada en su declaración

```
char a; char& b = a;
```

excepto cuando la referencia es un parámetro de la función: en ese caso se inicializa al invocarse la función y efectuarse el paso de parámetros.

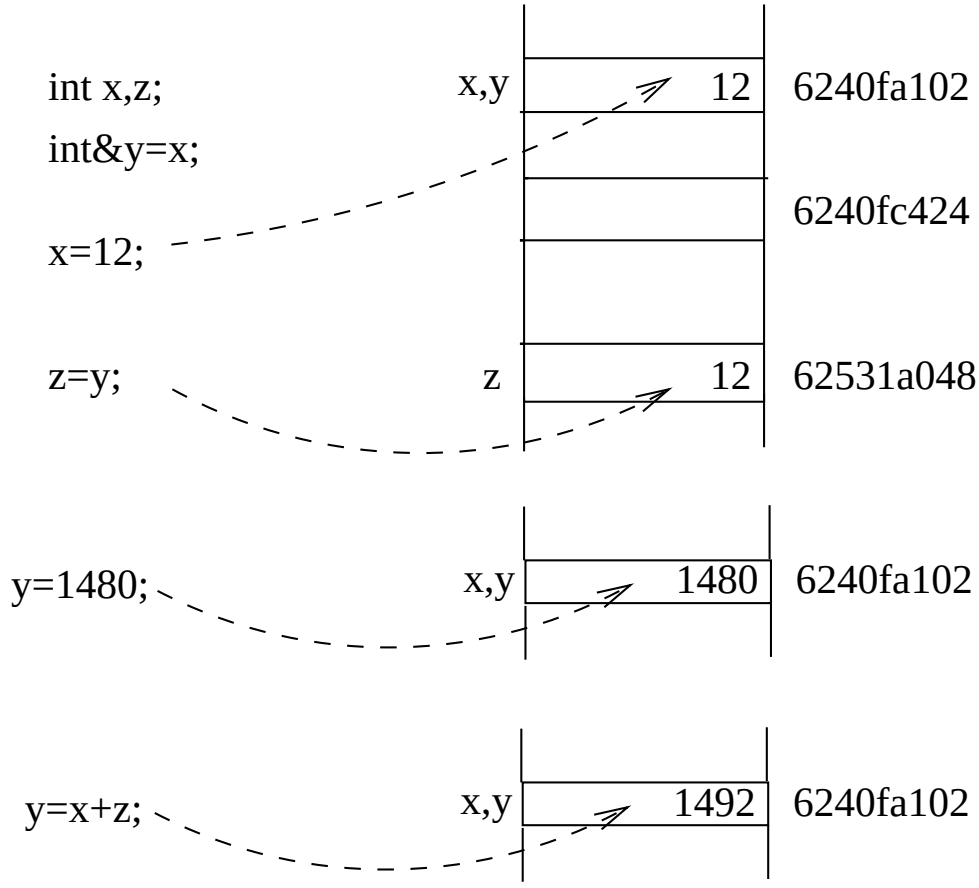
Referencias

- El valor de una referencia no se puede cambiar. Sólo se puede modificar el valor al que se refiere

```
a = 'b'; b = 'c'; // a vale 'c'
```

- La dirección de una referencia es la de la variable a la que se refiere.
- Las referencias están implementadas mediante punteros, pero no disponen de las operaciones sobre punteros.

Referencias



Ejemplo de manipulación de referencias

Paso de Parámetros

- Los de salida y entrada/salida se especifican mediante el uso de referencias, como sigue:
  (paso de parámetro por referencia)

Paso de Parámetros

- Los de salida y entrada/salida se especifican mediante el uso de referencias, como sigue:
  (paso de parámetro por referencia)
  - Parámetro formal: `T&` *`identificador`*

Paso de Parámetros

- Los de salida y entrada/salida se especifican mediante el uso de referencias, como sigue:
  (paso de parámetro por referencia)
  - Parámetro formal: `T&` *`identificador`*
  - Parámetro actual: *variable*

Paso de Parámetros

- Los de salida y entrada/salida se especifican mediante el uso de referencias, como sigue:
  (paso de parámetro por referencia)
  - Parámetro formal: `T&` *`identificador`*
  - Parámetro actual: *variable*
- Los parámetros de entrada pueden ser especificados de las siguientes maneras:

Paso de Parámetros

- Los de salida y entrada/salida se especifican mediante el uso de referencias, como sigue:
  (paso de parámetro por referencia)
  - Parámetro formal: `T&` *`identificador`*
  - Parámetro actual: *variable*
- Los parámetros de entrada pueden ser especificados de las siguientes maneras:
  1. El parámetro formal se comporta como una variable local de la función, inicializada con el valor del parámetro actual (se usa el constructor por copia del tipo `T`):
     (paso de parámetro por valor)

Paso de Parámetros

- Los de salida y entrada/salida se especifican mediante el uso de referencias, como sigue:
  (paso de parámetro por referencia)
  - Parámetro formal: `T&` *`identificador`*
  - Parámetro actual: *variable*
- Los parámetros de entrada pueden ser especificados de las siguientes maneras:
  1. El parámetro formal se comporta como una variable local de la función, inicializada con el valor del parámetro actual (se usa el constructor por copia del tipo `T`):
     (paso de parámetro por valor)
     - Parámetro formal: `T` *`identificador`*

Paso de Parámetros

- Los de salida y entrada/salida se especifican mediante el uso de referencias, como sigue:
  (paso de parámetro por referencia)
  - Parámetro formal: `T&` *`identificador`*
  - Parámetro actual: *variable*
- Los parámetros de entrada pueden ser especificados de las siguientes maneras:
  1. El parámetro formal se comporta como una variable local de la función, inicializada con el valor del parámetro actual (se usa el constructor por copia del tipo `T`):
     (paso de parámetro por valor)
     - Parámetro formal: `T` *`identificador`*
     - Parámetro actual: *expresión*

Paso de Parámetros

2. El parámetro formal se comporta como una constante local de la función, inicializada con el valor del parámetro actual (se usa el constructor por copia del tipo `T`):
   (paso de parámetro por valor constante)

Paso de Parámetros

2. El parámetro formal se comporta como una constante local de la función, inicializada con el valor del parámetro actual (se usa el constructor por copia del tipo `T`):
   (paso de parámetro por valor constante)
   - Parámetro formal: `const T` *`identificador`*

Paso de Parámetros

2. El parámetro formal se comporta como una constante local de la función, inicializada con el valor del parámetro actual (se usa el constructor por copia del tipo `T`):
   (paso de parámetro por valor constante)
   - Parámetro formal: `const T` *`identificador`*
   - Parámetro actual: *expresión*

Paso de Parámetros

2. El parámetro formal se comporta como una constante local de la función, inicializada con el valor del parámetro actual (se usa el constructor por copia del tipo `T`):
   (paso de parámetro por valor constante)
   - Parámetro formal: `const T` *`identificador`*
   - Parámetro actual: *expresión*
3. El parámetro formal es una referencia al parámetro actual (no se hace copia) pero el parámetro formal no se puede modificar:
   (paso de parámetro por referencia constante)

Paso de Parámetros

2. El parámetro formal se comporta como una constante local de la función, inicializada con el valor del parámetro actual (se usa el constructor por copia del tipo `T`):
   (paso de parámetro por valor constante)
   - Parámetro formal: `const T` *`identificador`*
   - Parámetro actual: *expresión*
3. El parámetro formal es una referencia al parámetro actual (no se hace copia) pero el parámetro formal no se puede modificar:
   (paso de parámetro por referencia constante)
   - Parámetro formal: `const T&` *`identificador`*

Paso de Parámetros

2. El parámetro formal se comporta como una constante local de la función, inicializada con el valor del parámetro actual (se usa el constructor por copia del tipo `T`):
   (paso de parámetro por valor constante)
   - Parámetro formal: `const T` *`identificador`*
   - Parámetro actual: *expresión*
3. El parámetro formal es una referencia al parámetro actual (no se hace copia) pero el parámetro formal no se puede modificar:
   (paso de parámetro por referencia constante)
   - Parámetro formal: `const T&` *`identificador`*
   - Parámetro actual: *variable*

Paso de Parámetros

- El paso de parámetros por valor (`T x`, `const T x`) se recomienda para los parámetros de entrada de tipos elementales predefinidos (`int`, `char`, `bool`, ...).

Paso de Parámetros

- El paso de parámetros por valor (`T x`, `const T x`) se recomienda para los parámetros de entrada de tipos elementales predefinidos (`int`, `char`, `bool`, ...).
- En cambio, para parámetros de entrada que sean objetos "complicados" es preferible el paso por referencia constante (`const T& x`), ya que se evita hacer copias costosas.

Paso de Parámetros

- El paso de parámetros por valor (`T x`, `const T x`) se recomienda para los parámetros de entrada de tipos elementales predefinidos (`int`, `char`, `bool`, ...).
- En cambio, para parámetros de entrada que sean objetos "complicados" es preferible el paso por referencia constante (`const T& x`), ya que se evita hacer copias costosas.
- El calificativo `const` permite que el compilador detecte errores en los que se modifica inadvertidamente un parámetro de entrada; no sería un problema si usamos paso por valor (`T x`), ya que trabajamos con una copia, pero es un problema muy serio si usamos paso por referencia, pues la función trabaja con el parámetro actual (simplemente le da un nombre o alias para referirse a él).

Paso de Parámetros

Parámetros de entrada/salida usando referencias

```
void ff(int& a, int& b) {
    if (a > 3) { a += 2; }   // ≡ a = a + 2
    else       { a -= 3; }   // ≡ a = a - 3
    b = b + 5 * a;
}

void f() {
    int x, y;
    cin >> x >> y;

    // p.e. x = 4, y = 7
    ff(x, y);
    // x = 6, y = 37
}
```

Paso de Parámetros

- Los parámetros de salida o entrada/salida pueden ser simulados mediante el uso de punteros al estilo de C (no recomendado).
- En el ejemplo siguiente el calificativo `const` indica que el puntero **no** puede modificarse (pero el objeto apuntado sí puede ser modificado).

Parámetros de entrada/salida usando punteros

```
void ff(int* const a,int* const b) {
   if (*a > 3) { *a += 2; }
   else        { *a -= 3; }
   *b = *b + 5 * (*a);
}

void f() {
   int x, y;
   cin >> x >> y;

   // p.e. x = 4, y = 7
   ff(&x,&y);
   // x = 6, y = 37
}
```

Paso de Parámetros

Parámetros de salida

```
void ff(int& a, int& b) {
   cin >> a >> b;

   if (a > 3) { a += 2; }
   else       { a -= 3; }
   b = b + 5 * a;
}

void f() {
   int x, y;

   // los valores de ‘x’ e ‘y’ no son
   // relevantes para ‘ff’
   ff(x, y);
   // ‘x’ e ‘y’ se han modificado
}
```

Paso de Parámetros

Parámetros de entrada

```
void escribe_f(int a, const int b,
               const racional& r) {
   // 'b' y 'r' no pueden modificarse aqui
   cout << a + r.num() << endl;
   a = a + 2;
   cout << b - a + r.denom() << endl;
}

void f() {
   int x, y;
   racional r(1, 7);

   cin >> x >> y;

   // p.e. x = 5, y = 2 , r = 1/7
   // el 3er parametro tiene que ser objeto
   escribe_f(x + 3,  2 * y, r);
   // 'x', 'y', 'r' no han cambiado su valor
}
```

Devolución de Resultados

Una función puede devolver sus resultados por copia, por referencia o por referencia constante, i.e.,

tipo funcion(*lista_parámetros_formales*)

donde *tipo* es el nombre de un tipo (`T` o `void`), una referencia (`T&`), o una referencia constante (`const T&`).

Devolución de Resultados

```
int  f1(int x) { return x; }
int  f2(int& x) { return x; }
int& f3(int& x) { return x; }
int& f4(int x)  { return x; } ← MAL!!

void f() {
   int w = 10;
   int  y = f1(w);
   // genera 3 copias del valor de 'w':
   // w → x, x → resultado, resultado → y
   int z = f2(w);
   // el parametro actual ha de ser una variable!
   // genera 2 copias: w ≡ x → resultado, resultado → z
   int u = f3(w);
   // genera 1 copia: w ≡ x ≡ resultado → u
   ...
}
```

Devolución de Resultados

Una función no debe devolver un puntero o una referencia a una variable local o a un parámetro puesto que son destruidos al acabar la ejecución. El compilador detecta la mayor parte de errores como los que mostramos a continuación si activamos los *flags* adecuados. No obstante, a veces no son detectados, de manera que el error se produce en tiempo de ejecución.

```
float* calcula(int x, const float y) {
   float z = 0.0;
   while (x > 0) {
      z = z + y / x;
      x--;
   }
   return &z;      ERROR !!
}

float& calcula(int x, const float y) {
   float z = 0.0;
   while (x > 0) {
      z = z + y / x;
      x--;
   }
   return z;       ERROR !!
}
```

Arrays

- C++ tiene un constructor de vectores (*arrays*) predefinido que básicamente funciona igual que en C o Java.
- Para declarar un vector de n elementos del tipo `T` escribiremos:

```
T identificador[n];
```

- Los componentes de un vector de n elementos se indexan de 0 a $n-1$. En C++ no se hace comprobación de rango ni estática ni en tiempo de ejecución; acceder a `A[i]`, si `i` < 0 ó `i` $\geq n$, puede provocar un error inmediato o tener consecuencias todavía más nefastas.

Arrays

- Los arrays (y los punteros) suelen emplearse como mecanismos de bajo nivel para implementar clases. Estos detalles de implementación quedan ocultos al usuario, que manejará clases seguras (se realizan las comprobaciones que hagan falta internamente) y no tendrá que manipular arrays directamente.
- Cuando se crea un array con

```
T identificador[n];
```

el valor n ha de ser constante o una expresión calculable en tiempo de compilación. Para crear un array *dinámico* se ha de emplear una técnica diferente (veáse la sesión **??**).

Arrays

- La conexión entre arrays y punteros en C++ es profunda: un array es de hecho un puntero constante al primer componente del vector: `p` ≡ `& p[0]`. Y en general, `p + i` ≡ `& p[i]`. La única diferencia entre un puntero y un array es que un array no puede ser "modificado":

```
int p[10]; int z[10]; int* q = p;
for (int i = 0; i < 10; i++)
    p[i] = (i + 1) * 10;
cout << *q << endl;       // imprime 10
cout << p[2] << endl;     // imprime 30
cout << *(q + 2) << endl; // imprime 30
q[2] = 33;
cout << *(p + 2) << endl; // imprime 33
q = z;                    // ok
p = z;  // ERROR! la direccion guardada en p no
        // puede ser modificada
```

Arrays

- Puesto que un array es un puntero, se puede pasar como parámetro eficientemente, pero convendrá usar el calificador `const` para impedir que se hagan modificaciones accidentales si se trata de un parámetro de entrada.
- Se puede pasar una referencia a un array, de modo que las reglas de paso de parámetros sean las mismas que en los restantes casos; pero nunca se puede hacer paso por valor, sólo por referencia o por referencia constante.
- El tipo de un array de `T`'s es, cualquiera que sea su tamaño, `T* const`, o equivalentemente, `T[]`.
- No hay arrays multidimensionales: para matrices, por ejemplo, se habrá de usar un array de arrays:

```
double matriz[10][10];
```

Arrays

```
double producto_escalar(const double v[],
                        const double w[], int n) {
   double s = 0.0;
   for (int i = 0; i < n; i++)
      s = s + v[i] * w[i];
   return s;
}

double eval_pol(const double *const coef, int n,
                double x) {
   double s = 0.0;
   for (int i = n - 1; i >= 0; i--)
      s = s * x + coef[i];
   return s;
}

void main(void) {
   double v[10], w[10], x[20], y[20];
   double pesc = producto_escalar(v, w, 10);
   pesc = pesc + producto_escalar(x, y, 20);
   // queremos que 'v' represente a x^3 - 2x + 3
   v[0] = 3; v[1] = -2; v[2] = 0; v[3] = 1;
   // y evaluar el polinomio en x = sqrt(2)
   double res = eval_pol(v, 4, sqrt(2.0));
}
```

Memoria Dinámica

- Permite la reserva y liberación de espacio de memoria para datos a lo largo de la ejecución del programa.

Memoria Dinámica

- Permite la reserva y liberación de espacio de memoria para datos a lo largo de la ejecución del programa.
  - La reserva de memoria se realiza mediante las operaciones `new` y `new[`n`]`; ésta última reserva un array de n objetos.

Memoria Dinámica

- Permite la reserva y liberación de espacio de memoria para datos a lo largo de la ejecución del programa.
  - La reserva de memoria se realiza mediante las operaciones `new` y `new[`n`]`; ésta última reserva un array de n objetos.
  - Para liberar memoria se utilizan `delete` y `delete[]`.

Memoria Dinámica

- Permite la reserva y liberación de espacio de memoria para datos a lo largo de la ejecución del programa.
  - La reserva de memoria se realiza mediante las operaciones `new` y `new[`n`]`; ésta última reserva un array de n objetos.
  - Para liberar memoria se utilizan `delete` y `delete[]`.
- Los objetos creados en memoria dinámica son anónimos; no podemos acceder a ellos mediante un nombre, tenemos que hacerlo a través de punteros.

Memoria Dinámica

- La memoria dinámica permite un uso eficiente del espacio de memoria y utilizar sólo la cantidad necesaria.

Memoria Dinámica

- La memoria dinámica permite un uso eficiente del espacio de memoria y utilizar sólo la cantidad necesaria.
- El uso de memoria dinámica puede conllevar la simplificación del código en algunos problemas y la complicación del mismo en otros. Por lo tanto, existe un compromiso entre el uso de memoria estática y de memoria dinámica.

Memoria Dinámica

Obtención y liberación de memoria dinámica para tipos predefinidos

```
int* pi =  new int;
// reserva memoria para un entero al que apunta pi

char* pc =  new char [10];
// reserva memoria para tabla de 10 caracteres (0..9)
// 'pc' apunta a la componente 0 (pc ≡ &pc[0])
// pc[i] ≡ *(pc+i) accede a la i-esima componente

...

 delete pi;   // libera la memoria del entero
              // apuntado por 'pi'; el puntero
              // conserva su valor, pero ahora
              // apunta a una zona de memoria
              // disponible para reservar
 delete[]  pc; // libera la memoria de la tabla
              // de caracteres apuntada por 'pc'
```

Constructor por Copia

- Un constructor por copia **inicializa** objetos con copias de otros objetos de la misma clase.

Constructor por Copia

- Un constructor por copia **inicializa** objetos con copias de otros objetos de la misma clase.
- Toda clase tiene un constructor por copia. Si no lo programamos, el compilador proporciona uno "de oficio" que se limita a copiar uno a uno los atributos.

Constructor por Copia

- Un constructor por copia **inicializa** objetos con copias de otros objetos de la misma clase.
- Toda clase tiene un constructor por copia. Si no lo programamos, el compilador proporciona uno "de oficio" que se limita a copiar uno a uno los atributos.
- En general, el constructor por copia "de oficio" nos sirve, **excepto** si el objeto tiene uno o más atributos que son punteros a memoria dinámica o un array.

Constructor por Copia

- Un constructor por copia **inicializa** objetos con copias de otros objetos de la misma clase.
- Toda clase tiene un constructor por copia. Si no lo programamos, el compilador proporciona uno "de oficio" que se limita a copiar uno a uno los atributos.
- En general, el constructor por copia "de oficio" nos sirve, **excepto** si el objeto tiene uno o más atributos que son punteros a memoria dinámica o un array.
- El constructor por copia se invoca automáticamente al hacer paso de parámetros por valor, al hacer retornos por valor o en declaraciones tales como

Sintaxis equivalentes

```
vector v1 = v2; vector v1(v2);
```

Constructor por Copia

Ejemplo de Constructor por Copia

```
class vector {
  public:
     vector(int ss = 8);
     vector(const vector& V); // por copia
     ...
  private:
     int* t; // puntero a la tabla de enteros
     int s;  // tamaño de la tabla
};

vector::vector(int ss) : s (ss), t (new int[s]) {}

vector::vector(const vector& V) : s (V.s), t (new int[s]) {
   for (int i = 0; i < s; i++)
      t[i] = V.t[i]; // se inicializa la tabla 't' con 'V.t'
}

void f() {
   vector v1(10);  // vector de 10 enteros
   ...
   vector v2 = v1; // vector 'v2' es copia de 'v1'
}
```

Destructor

- Los destructores proporcionan un mecanismo automático que garantiza la destrucción de los objetos.

Destructor

- Los destructores proporcionan un mecanismo automático que garantiza la destrucción de los objetos.
- Se declaran como métodos que no devuelven resultados, con el nombre de la clase precedido del caracter $\sim$. No tienen parámetros.

Destructor

- Los destructores proporcionan un mecanismo automático que garantiza la destrucción de los objetos.
- Se declaran como métodos que no devuelven resultados, con el nombre de la clase precedido del caracter $\sim$. No tienen parámetros.
- Toda clase tiene un único destructor. Si no programamos el destructor, el compilador proporciona uno "de oficio".

Destructor

- Los destructores proporcionan un mecanismo automático que garantiza la destrucción de los objetos.
- Se declaran como métodos que no devuelven resultados, con el nombre de la clase precedido del caracter $\sim$. No tienen parámetros.
- Toda clase tiene un único destructor. Si no programamos el destructor, el compilador proporciona uno "de oficio".
- En general, el destructor "de oficio" nos sirve excepto si el objeto tiene uno o más atributos que son punteros a memoria dinámica.

Destructor

- Los destructores proporcionan un mecanismo automático que garantiza la destrucción de los objetos.
- Se declaran como métodos que no devuelven resultados, con el nombre de la clase precedido del caracter $\sim$. No tienen parámetros.
- Toda clase tiene un único destructor. Si no programamos el destructor, el compilador proporciona uno "de oficio".
- En general, el destructor "de oficio" nos sirve excepto si el objeto tiene uno o más atributos que son punteros a memoria dinámica.
- Un destructor es invocado **sólo** por el sistema, justo en el momento en que el bloque donde se declaró el objeto termina. Nunca se llamará explicitamente al destructor.

Destructor

Ejemplo de Destructor

```
class vector {
  public:
     ...
     ~vector(); // destructor
     ...
  private:
     int s;
     int* t;
};

vector::~vector() { delete[] t; }

void f() {
   if ( ... ) {
      vector v1;
      ...
   } // destruccion de ‘v1’ al salir del bloque ‘if’
   vector v2;
   ...
}   // destruccion de ‘v2’ al salir del bloque de
    // la funcion ‘f’
```

Asignación

- Inicializar es diferente que asignar.

Asignación

- Inicializar es diferente que asignar.
- Toda clase tiene definida la asignación. Si no la programamos, el compilador proporciona el operador de asignación "de oficio".

Asignación

- Inicializar es diferente que asignar.
- Toda clase tiene definida la asignación. Si no la programamos, el compilador proporciona el operador de asignación "de oficio".
- En general, la asignación de oficio nos sirve, excepto si el objeto tiene uno o más atributos que son punteros a memoria dinámica.

Asignación

- Inicializar es diferente que asignar.
- Toda clase tiene definida la asignación. Si no la programamos, el compilador proporciona el operador de asignación "de oficio".
- En general, la asignación de oficio nos sirve, excepto si el objeto tiene uno o más atributos que son punteros a memoria dinámica.
- Podemos redefinir la asignación para que `a = b` tenga el efecto que deseamos.

Asignación

Ejemplo de Asignación

```
class vector {
  public:
     ...
     vector& operator=(const vector& V);
     ...
  private:
     int s;
     int* t;
};

vector& vector::operator=(const vector& V) {
   if (&V != this) {      // no hacer nada
      if (s != V.s) {     // si no son del mismo tamaño
         delete[] t;      // la tabla 't' del objeto
         s = V.s;         // al que se le hace la
         t = new int[s]; // asignacion no sirve
      }
      for (int i=0; i<s; i++)
         t[i] = V.t[i];
   }
   return *this;
}
```

El componente `this`

- `this` es una palabra reservada de C++ y es un puntero al objeto que invoca el método.

El componente `this`

- `this` es una palabra reservada de C++ y es un puntero al objeto que invoca el método.
- Es típico en C++ que la asignación devuelva la referencia al objeto modificado de manera que, p.e., a = b = 0 tenga sentido.

El componente `this`

- `this` es una palabra reservada de C++ y es un puntero al objeto que invoca el método.
- Es típico en C++ que la asignación devuelva la referencia al objeto modificado de manera que, p.e., a = b = 0 tenga sentido.

```
a = b = 0 ≡ a = (b = 0) ≡ a.operator=(b.operator=(0))
```

El componente `this`

- `this` es una palabra reservada de C++ y es un puntero al objeto que invoca el método.
- Es típico en C++ que la asignación devuelva la referencia al objeto modificado de manera que, p.e., a = b = 0 tenga sentido.

```
a = b = 0 ≡ a = (b = 0) ≡ a.operator=(b.operator=(0))
```

- La implementación del ejemplo no es robusta porque asume que `new` no va a fallar en darle memoria.

Reserva de Memoria para Objetos

Ejemplo

```
class vector {
  public:
     vector(int ss = 8); // vector de ‘ss’ enteros
     vector(const vector& T); // por copia
     ...
};

vector* pv = new vector(12);
// memoria para vector de 12 enteros apuntada por ‘pv’
vector* pv1 = new vector;
// memoria para vector de 8 enteros apuntada por ‘pv1’
vector* ptv1 = new vector[10];
// memoria para tabla de 10 vectores apuntada por ‘ptv1’
vector* pv2 = new vector(*pv1);
// memoria para vector copia de *pv1 apuntada por ‘pv2’
vector* ptv2 = new vector[10](*pv1);
// memoria para una tabla de 10 vectores que son copias de *pv1 y apuntada por ‘ptv2’
```

- Cuando se declara una tabla de objetos el sistema invoca el constructor para cada uno de ellos.

Liberación de Memoria de Objetos

Ejemplo

```
class vector {
  public:
     vector(int ss = 8);      // vector de ‘ss’ enteros
     vector(const vector& T); // por copia
     ~vector();               // destructor
     ...
};

// teniendo los objetos de la transparencia anterior
...

delete pv;
delete pv1; delete[] ptv1;
delete pv2; delete[] ptv2;
// se destruye la memoria asignada a los objetos apuntados
```

- Cuando se ejecuta `delete` sobre un puntero a un objeto el sistema invoca al destructor del objeto al que apunta para liberar su memoria.

Ejemplo: pila de enteros

Fichero stack.hpp

```
#ifndef _STACK_H
#define _STACK_H

class stack { public:
   stack();                  // constructor
   stack(const stack& S); // constructor por copia
   ~stack();                 // destructor
   stack& operator=(const stack& S); // redef. asign.

   void push(int x);
   void pop();
   int top() const;
   bool is_empty() const;

   ...
```

Ejemplo: pila de enteros

Fichero `stack.hpp` (cont.)

```
private:

   struct node {  // definicion de tipo privado
      node* next; // puntero a 'node'
      int info;
   };

   node* cim; // la pila consiste en un puntero
              // al nodo de la cima

   // metodo privado de clase para liberar la memoria;
   // libera la cadena de nodos que se inicia en el
   // nodo apuntado por 'p'

   static void borra_pila(node* p);

   // metodo privado de clase para realizar copias; copia toda la cadena
   // de nodos a partir del nodo apuntado por 'origen' y devuelve un puntero
   // al nodo inicial de la copia; la palabra reservada 'const' indica que
   // no se puede modificar el valor apuntado por el puntero 'origen'

   static node* copia_pila(const node* origen);

};
#endif
```

Ejemplo: pila de enteros

Fichero `stack.cpp`

```
#include "stack.hpp"

// metodos privados de clase
void stack::borra_pila(node* p) {
   if (p == NULL) return; // si es pila vacia no hace nada
   borra_pila(p -> next); // p->next equivale a (*p).next
                          // libera la memoria a partir  del siguiente
   delete p; // libera la memoria del objeto apuntado por ‘p’
}

// es necesario poner ’stack::node*’ como tipo del resultado
// porque ’node’ se define dentro de la clase
stack::node* stack::copia_pila
                   (const node* origen) {
   if (origen == NULL) return NULL;
   node* destino = new node;
   // reserva memoria para un nodo al que apunta ‘destino’
   destino -> info = origen -> info;
   destino -> next = copia_pila(origen -> next);
   // copia el resto de la cadena
   return destino; // devuelve el puntero a la copia
}
```

Ejemplo: pila de enteros

Fichero stack.cpp (cont.)

```
// metodos publicos

stack::stack(): cim(NULL) {}

stack::stack(const stack& S) {
   cim = copia_pila(S.cim);
   // genera una copia de la pila apuntada por 'S.cim'
}

stack::~stack() {
   borra_pila(cim);
   // libera la memoria de la pila apuntada por 'cim'
}

stack& stack::operator=(const stack& S) {
   if (this != &S) {    // no hacer autoasignacion
      node* aux = copia_pila(S.cim);
      // 'aux' apunta a una copia de 'S.cim'
      borra_pila(cim);
      cim = aux;
   }
   return *this;
   // devuelve la referencia a la pila que invoca el metodo
}
```

Ejemplo: pila de enteros

Fichero stack.cpp (cont.)

```
void stack::push(int x) {
   node* n = new node;
   // reserva memoria para un node al que apunta 'n'
   n -> info = x;
   n -> next = cim; // conecta el nuevo nodo con el
   cim = n;         // primer nodo de la pila y hace que
                    // este nuevo nodo sea la cima
} // añade el valor 'x' a la pila para la que se invoca el metodo

void stack::pop() {
   node* n = cim;
   if (cim != NULL) {
      cim = cim -> next; // 'cim' apunta a su siguiente
      delete n;          // libera la memoria del node
                         // desapilado (apuntado por 'n')
   }
   // falta tratar el error de pila vacia
}

int stack::top() const {
   if (cim != NULL) return cim -> info;
   // falta tratar el error de pila vacia
}

bool stack::is_empty() const {
   return cim == NULL;
}
```

Ejemplo: pila de enteros

Programa ejemplo

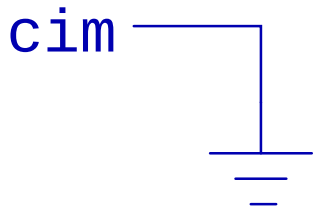
```
#include <iostream>
#include "stack.hpp"

void main(void) {
   int el;
   stack p;

   cin >> el;
   while (el != 0) { // mientras lleguen enteros ≠ 0
                     // por el canal estandar de entrada:
      p.push(el);    // apilarlos
      cin >> el;
   }
   stack q = p;           // inicializacion por copia
   while(!q.is_empty()) { // hace un palindromo en 'p'
      p.push(q.top());
      q.pop();
   }
   while (!p.is_empty()) {    // imprime la pila 'p'
      cout << p.top() << ' '; // (al tiempo que la
      p.pop();                // vacia)
   }
   cout << endl;
}
```

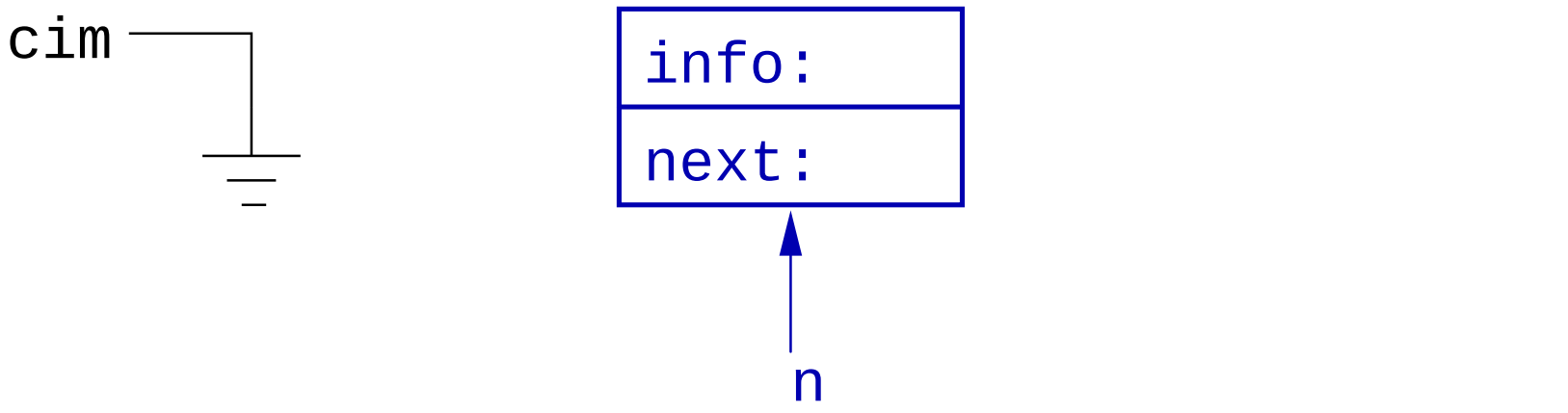
Ejemplo de Push

```
{                                        void stack::push(int x) {
...                                         node* n = new node;
 pila.push(x);                              n -> info = x;
 pila.push(y);                              n -> next = cim;
...                                         cim = n;
}                                        }
```



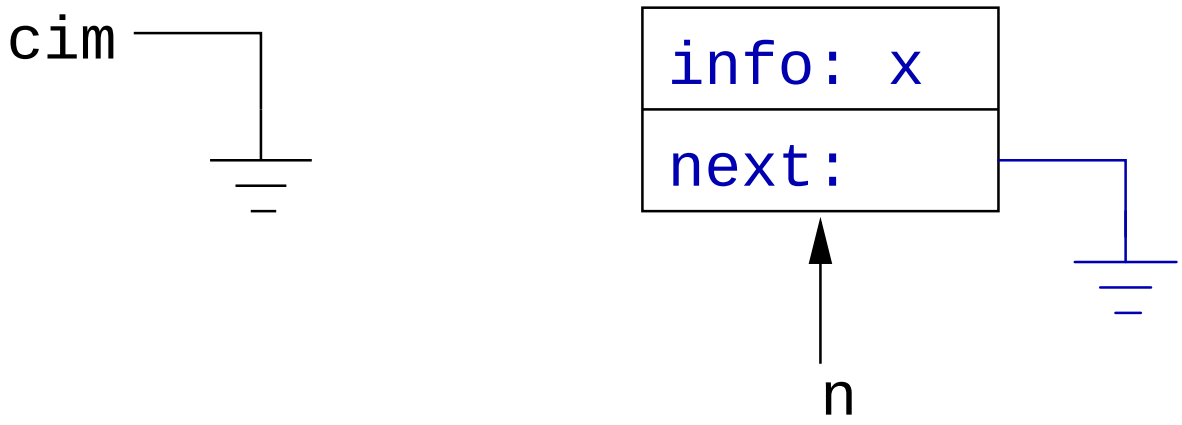
Ejemplo de Push

```
{                                    void stack::push(int x) {
...                                     node* n = new node;
 pila.push(x);                          n -> info = x;
 pila.push(y);                          n -> next = cim;
...                                     cim = n;
}                                    }
```



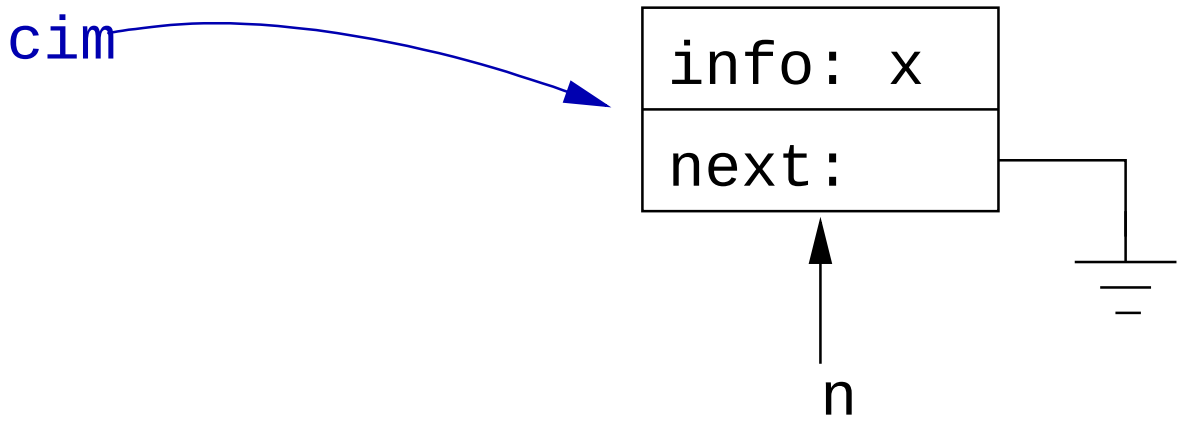
Ejemplo de Push

```
{                                    void stack::push(int x) {
...                                     node* n = new node;
 pila.push(x);                          n -> info = x;
 pila.push(y);                          n -> next = cim;
...                                     cim = n;
}                                    }
```



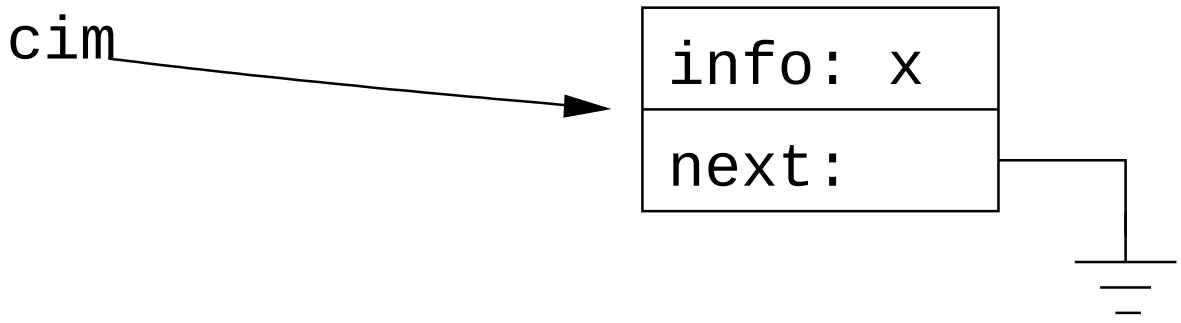
Ejemplo de Push

```
{                                    void stack::push(int x) {
...                                     node* n = new node;
 pila.push(x);                          n -> info = x;
 pila.push(y);                          n -> next = cim;
...                                     cim = n;
}                                    }
```



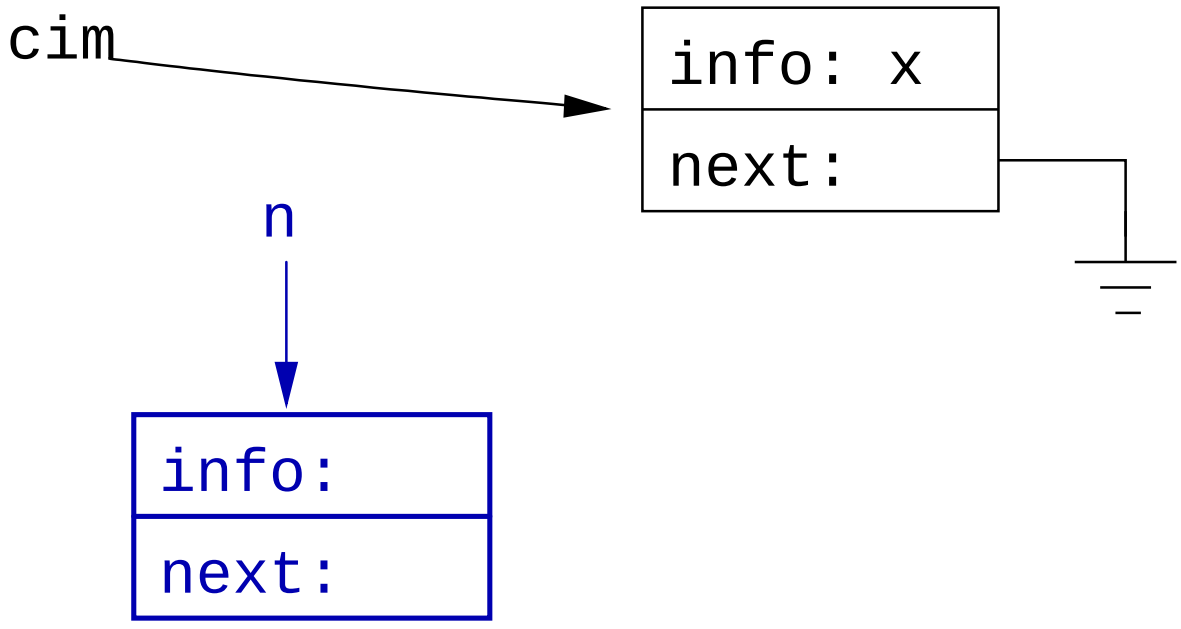
Ejemplo de Push

```
{                                    void stack::push(int x) {
...                                     node* n = new node;
 pila.push(x);                          n -> info = x;
 pila.push(y);                          n -> next = cim;
...                                     cim = n;
}                                    }
```



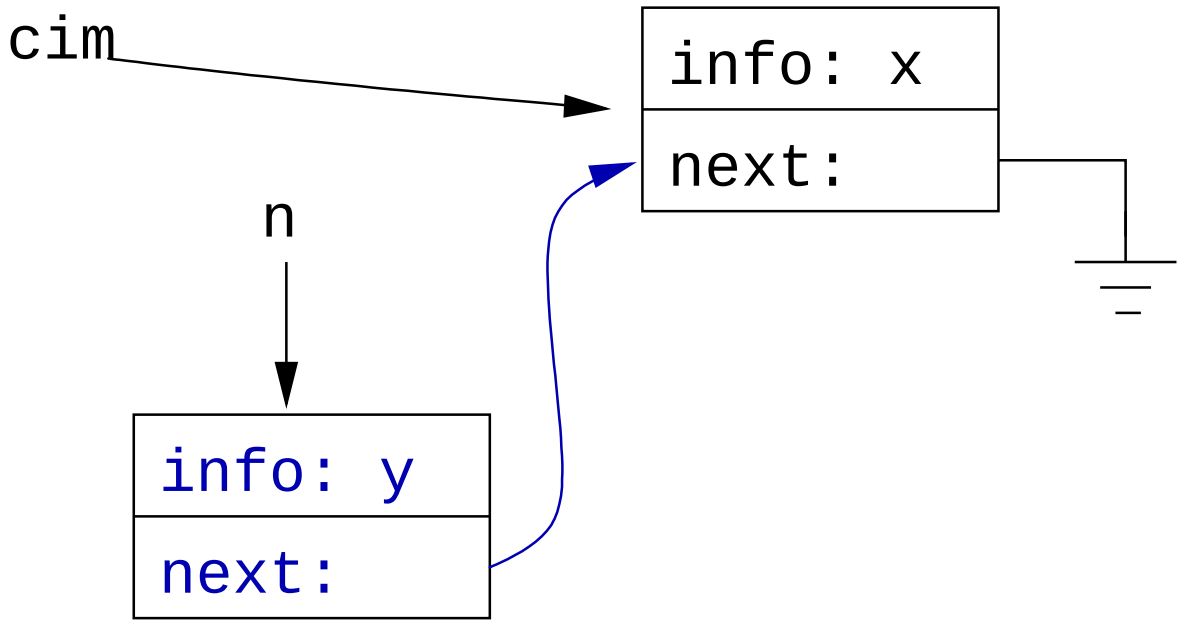
Ejemplo de Push

```
{                                    void stack::push(int x) {
...                                     node* n = new node;
 pila.push(x);                          n -> info = x;
 pila.push(y);                          n -> next = cim;
...                                     cim = n;
}                                    }
```



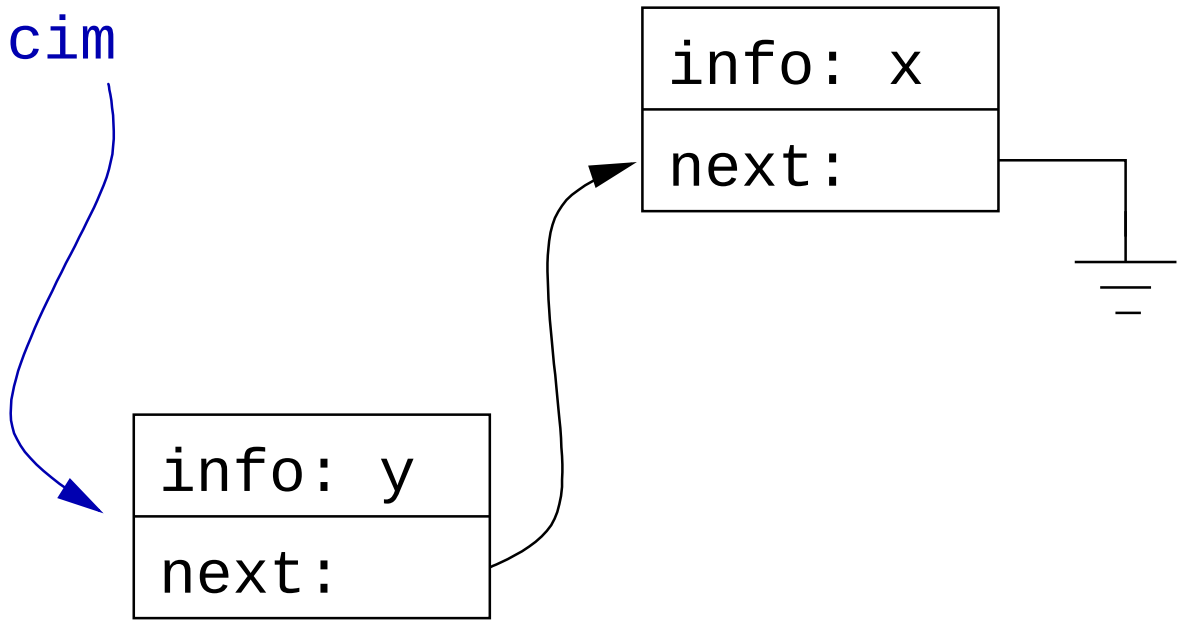
Ejemplo de Push

```
{                                    void stack::push(int x) {
...                                     node* n = new node;
 pila.push(x);                          n -> info = x;
 pila.push(y);                          n -> next = cim;
...                                     cim = n;
}                                    }
```



Ejemplo de Push

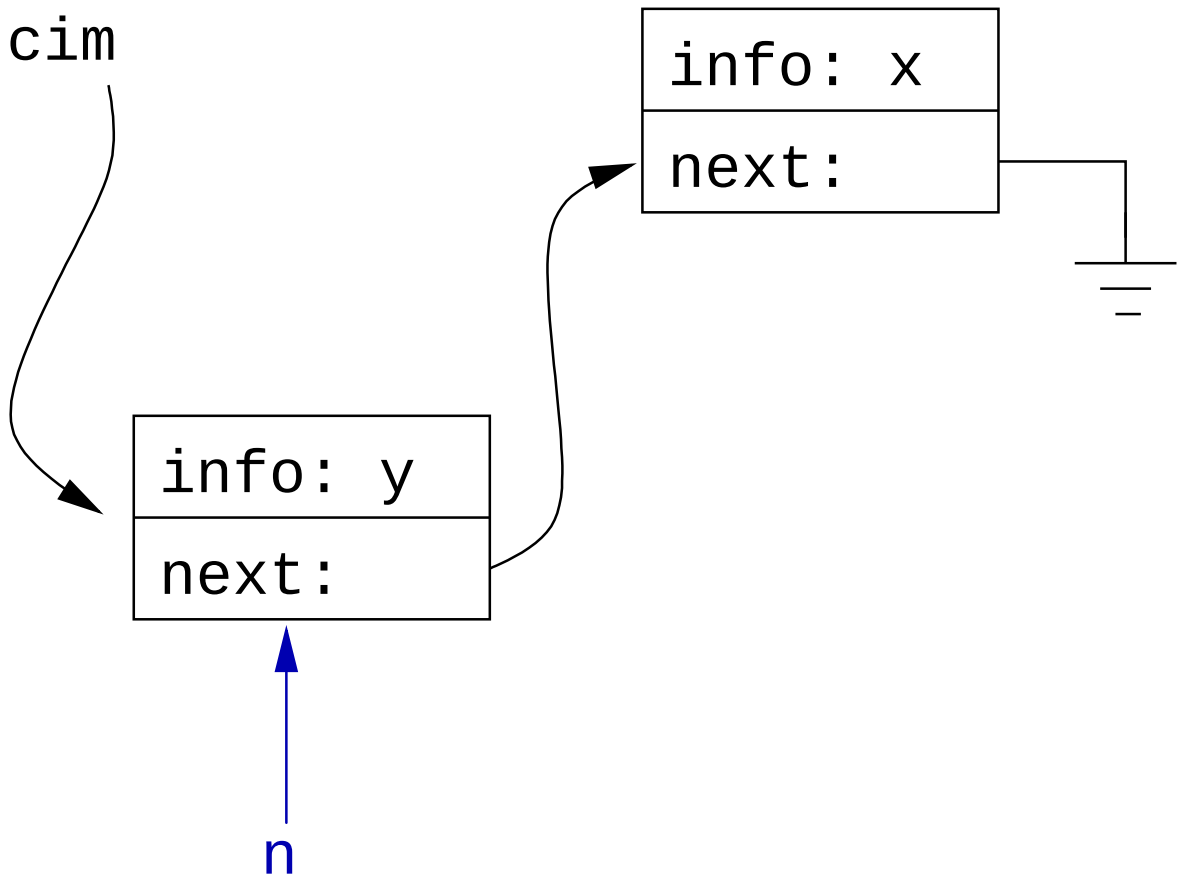
```
{                                    void stack::push(int x) {
...                                     node* n = new node;
 pila.push(x);                          n -> info = x;
 pila.push(y);                          n -> next = cim;
...                                     cim = n;
}                                    }
```



Ejemplo de Pop

```
{
...
 pila.pop();
 pila.pop();
...
}
```

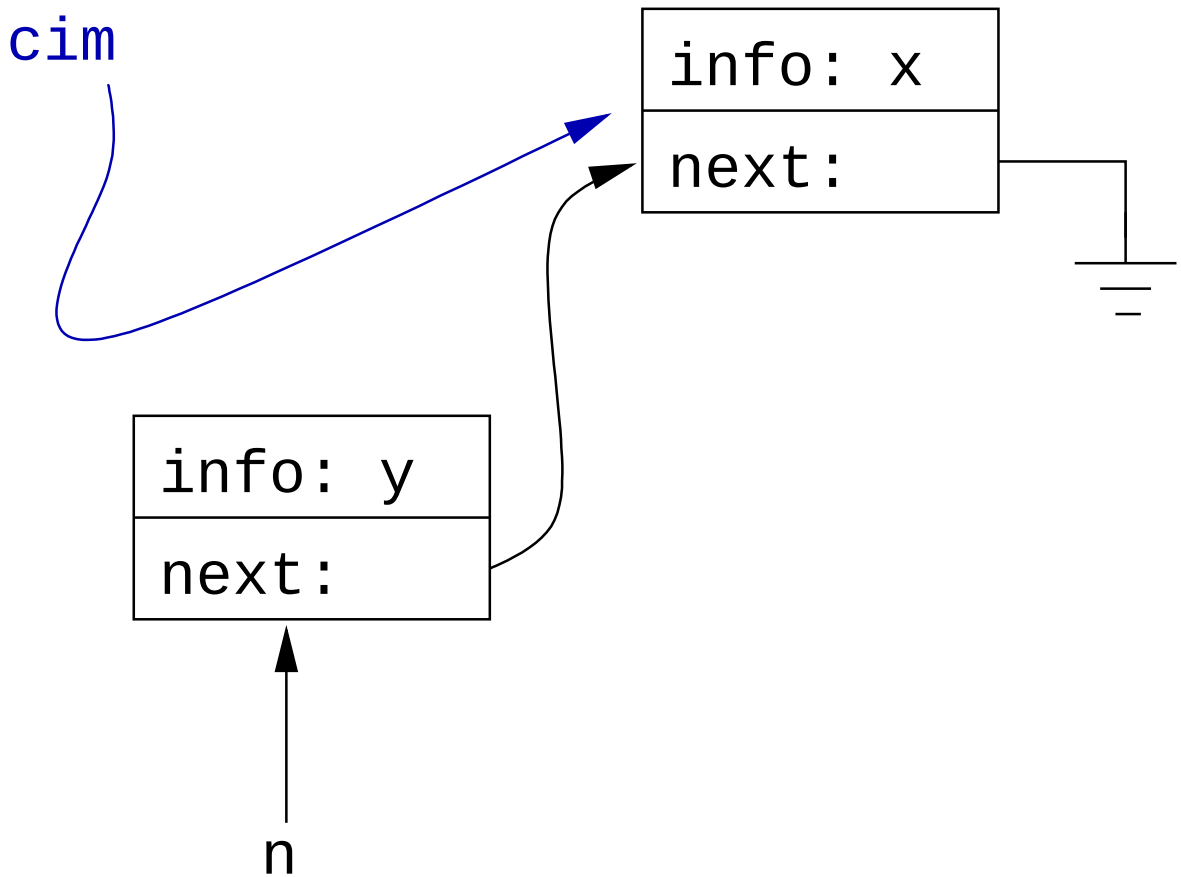
```
void stack::pop() {
   node* n = cim;
   if (cim != NULL) {
      cim = cim -> next;
      delete n;
   }
}
```



Ejemplo de Pop

```
{
...
 pila.pop();
 pila.pop();
...
}
```

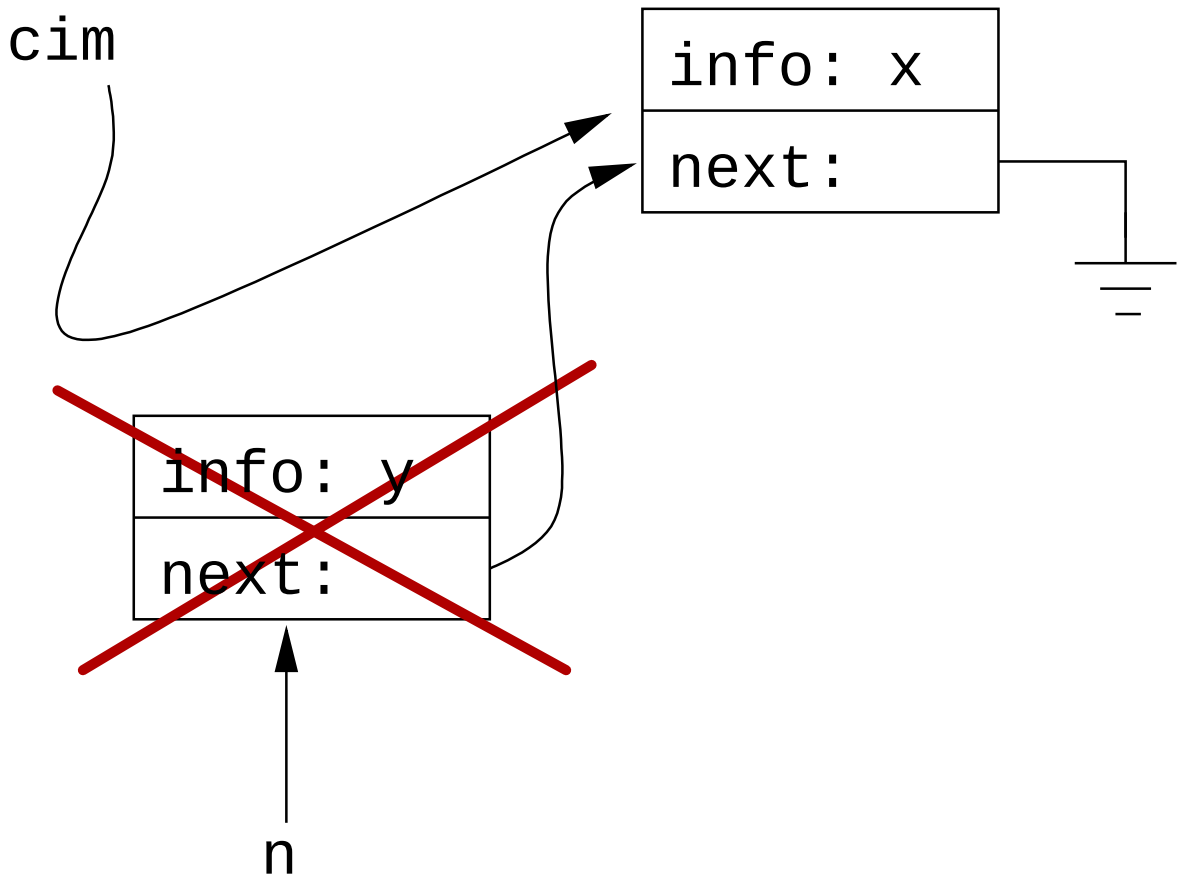
```
void stack::pop() {
   node* n = cim;
   if (cim != NULL) {
      cim = cim -> next;
      delete n;
   }
}
```



Ejemplo de Pop

```
{
...
 pila.pop();
 pila.pop();
...
}
```

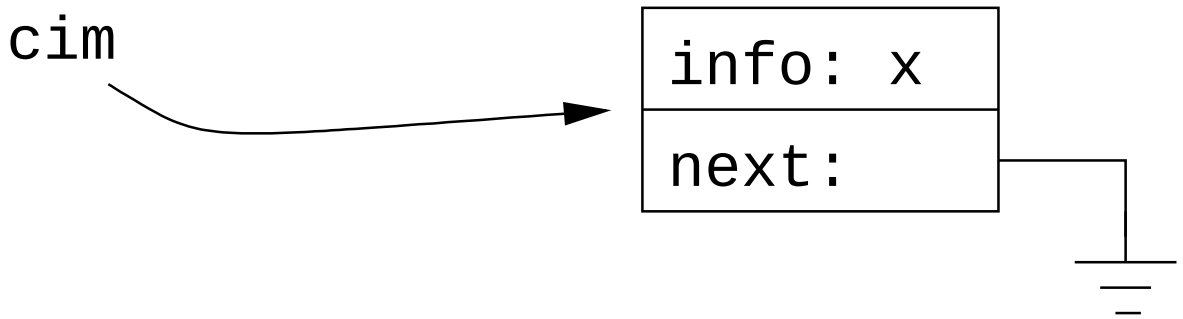
```
void stack::pop() {
   node* n = cim;
   if (cim != NULL) {
      cim = cim -> next;
      delete n;
   }
}
```



Ejemplo de Pop

```
{
...
 pila.pop();
 pila.pop();
...
}
```

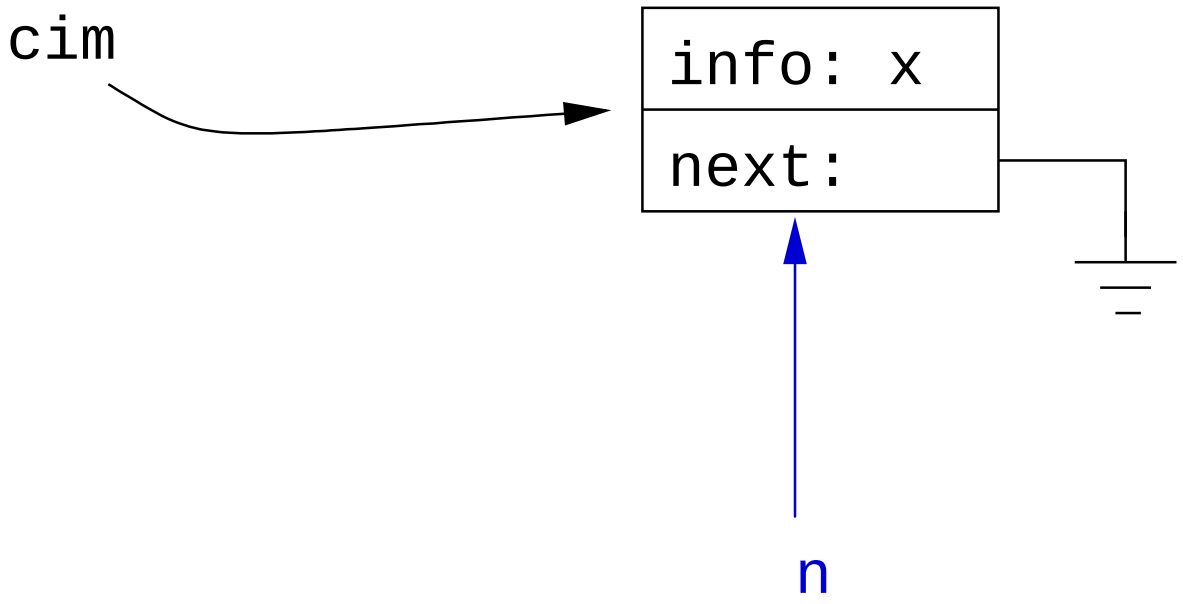
```
void stack::pop() {
   node* n = cim;
   if (cim != NULL) {
      cim = cim -> next;
      delete n;
   }
}
```



Ejemplo de Pop

```
{
...
 pila.pop();
 pila.pop();
...
}
```

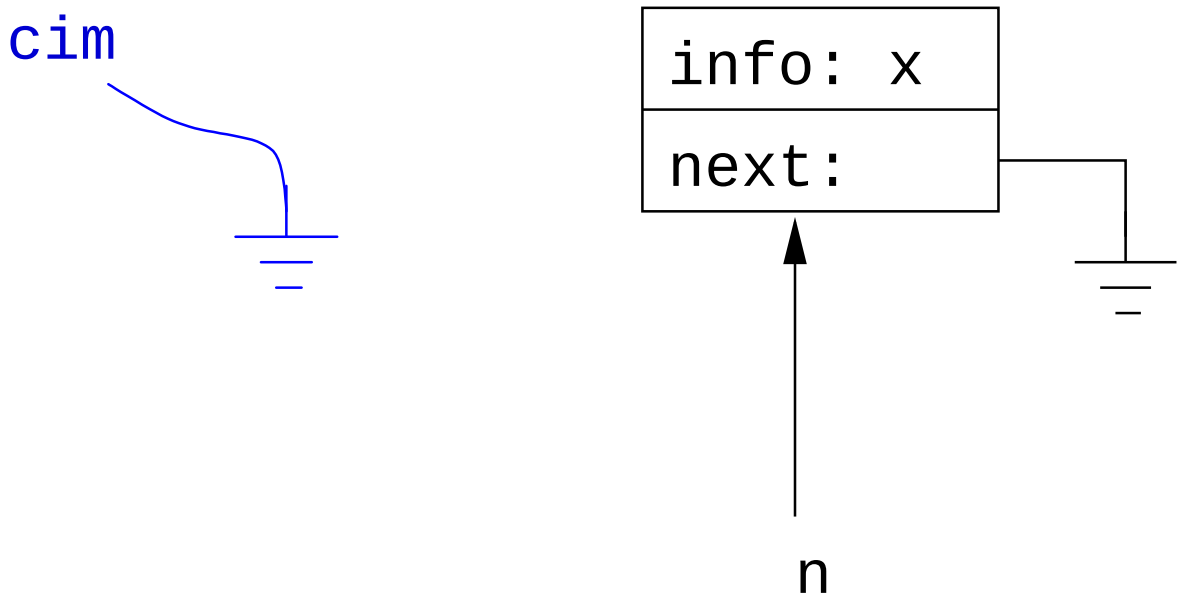
```
void stack::pop() {
   node* n = cim;
   if (cim != NULL) {
      cim = cim -> next;
      delete n;
   }
}
```



Ejemplo de Pop

```
{
...
 pila.pop();
 pila.pop();
...
}
```

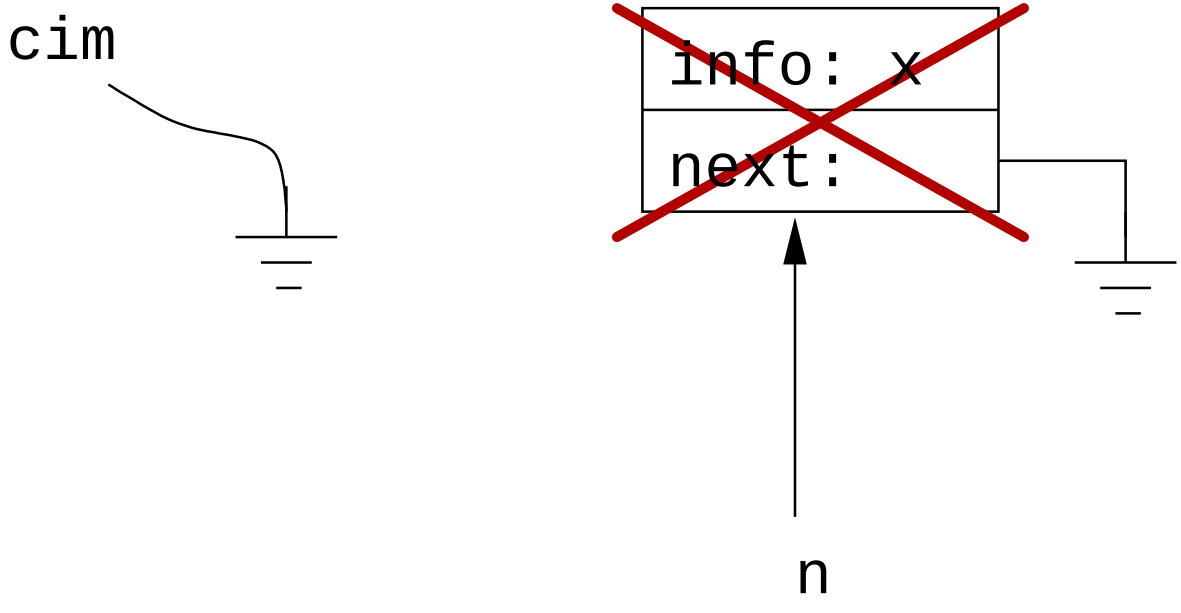
```
void stack::pop() {
   node* n = cim;
   if (cim != NULL) {
      cim = cim -> next;
      delete n;
   }
}
```



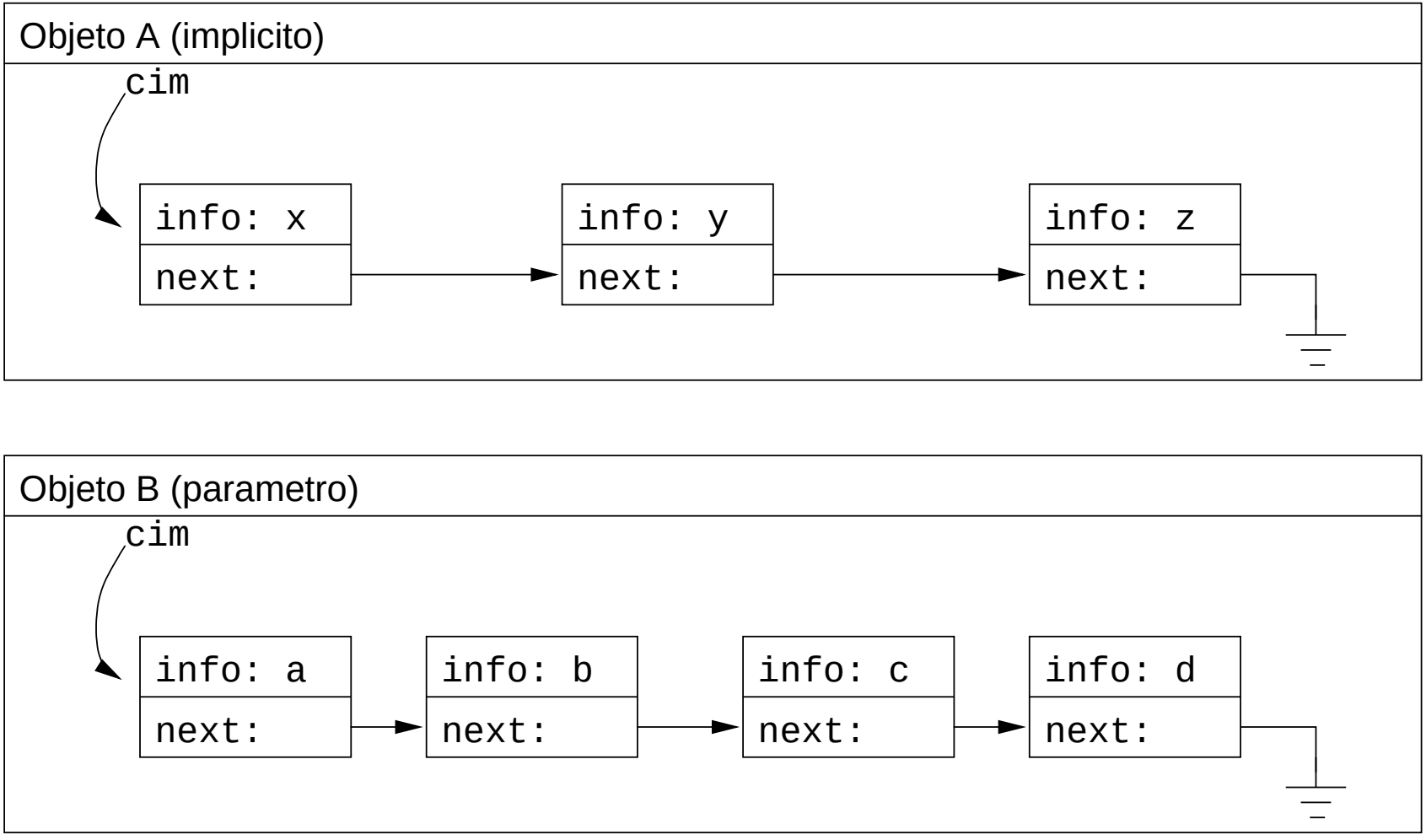
Ejemplo de Pop

```
{
...
 pila.pop();
 pila.pop();
...
}
```

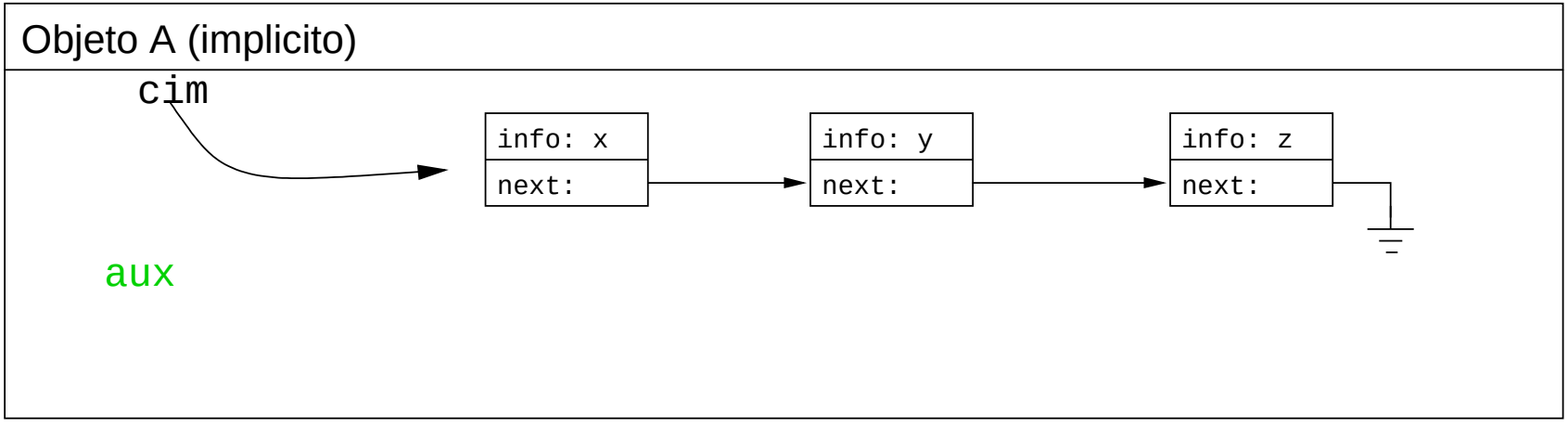
```
void stack::pop() {
   node* n = cim;
   if (cim != NULL) {
      cim = cim -> next;
      delete n;
   }
}
```

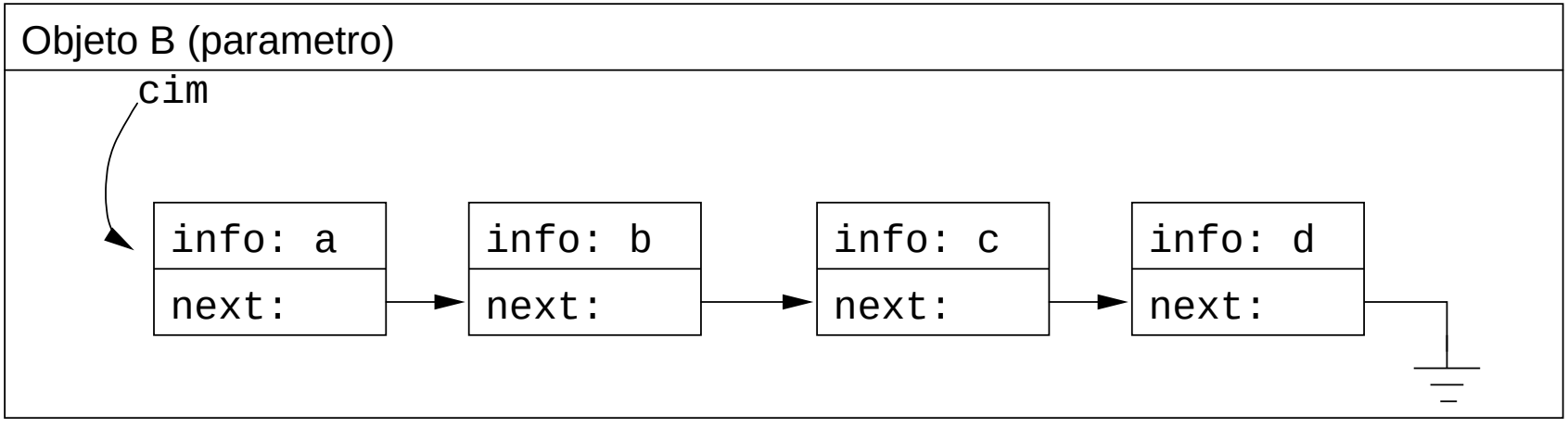


Ejemplo de Asignación. A = B

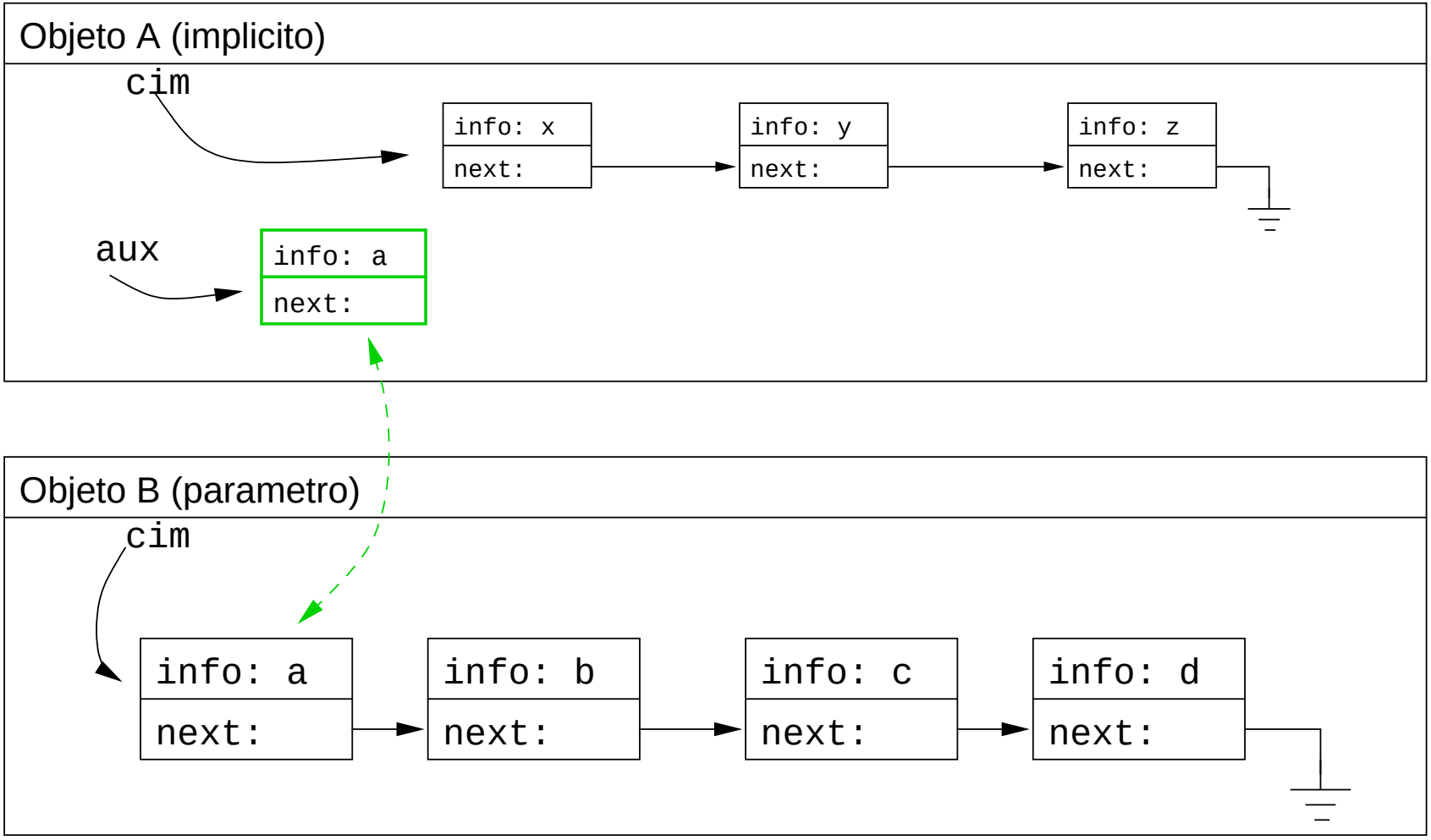


Ejemplo de Asignación. A = B

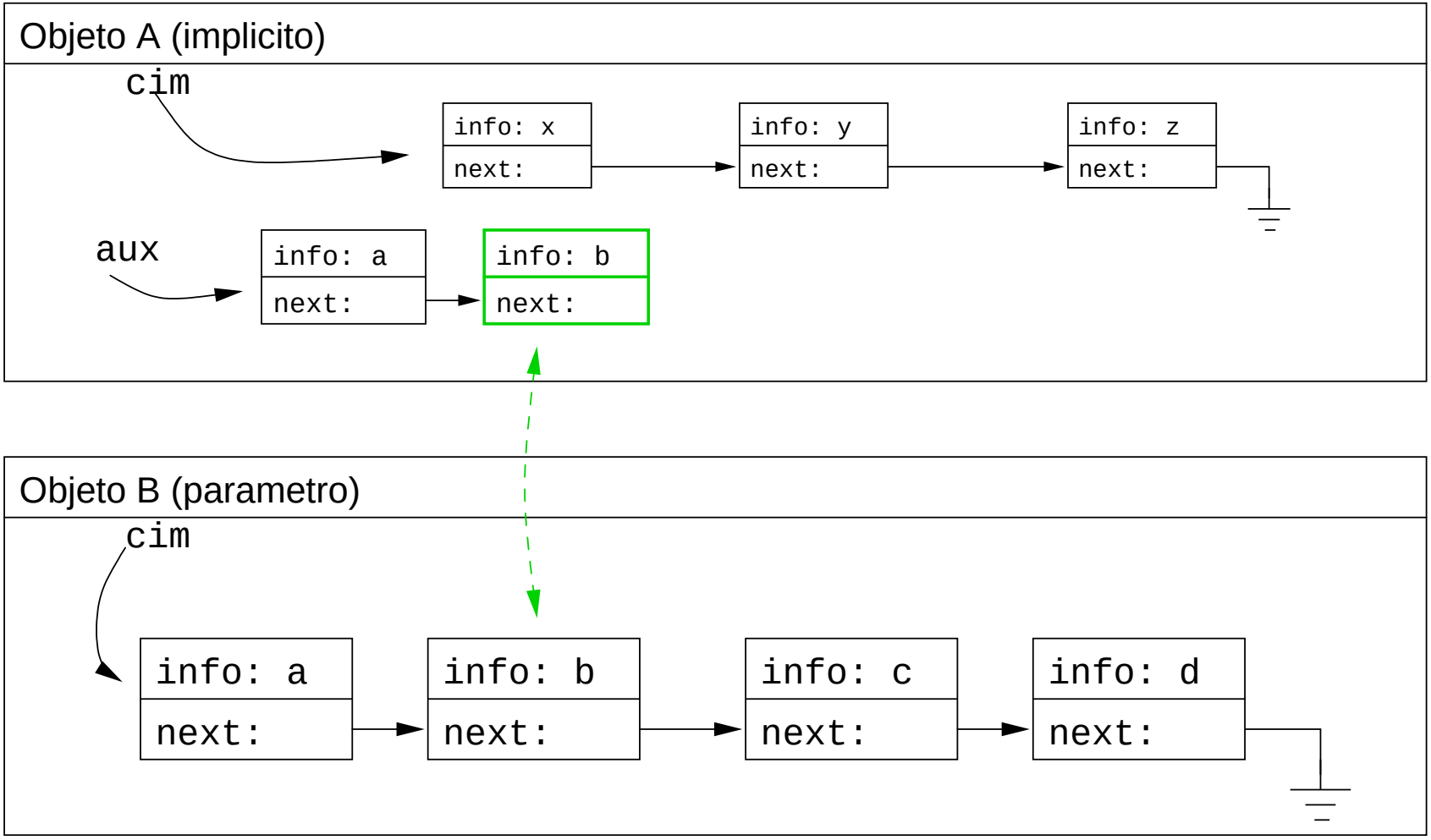




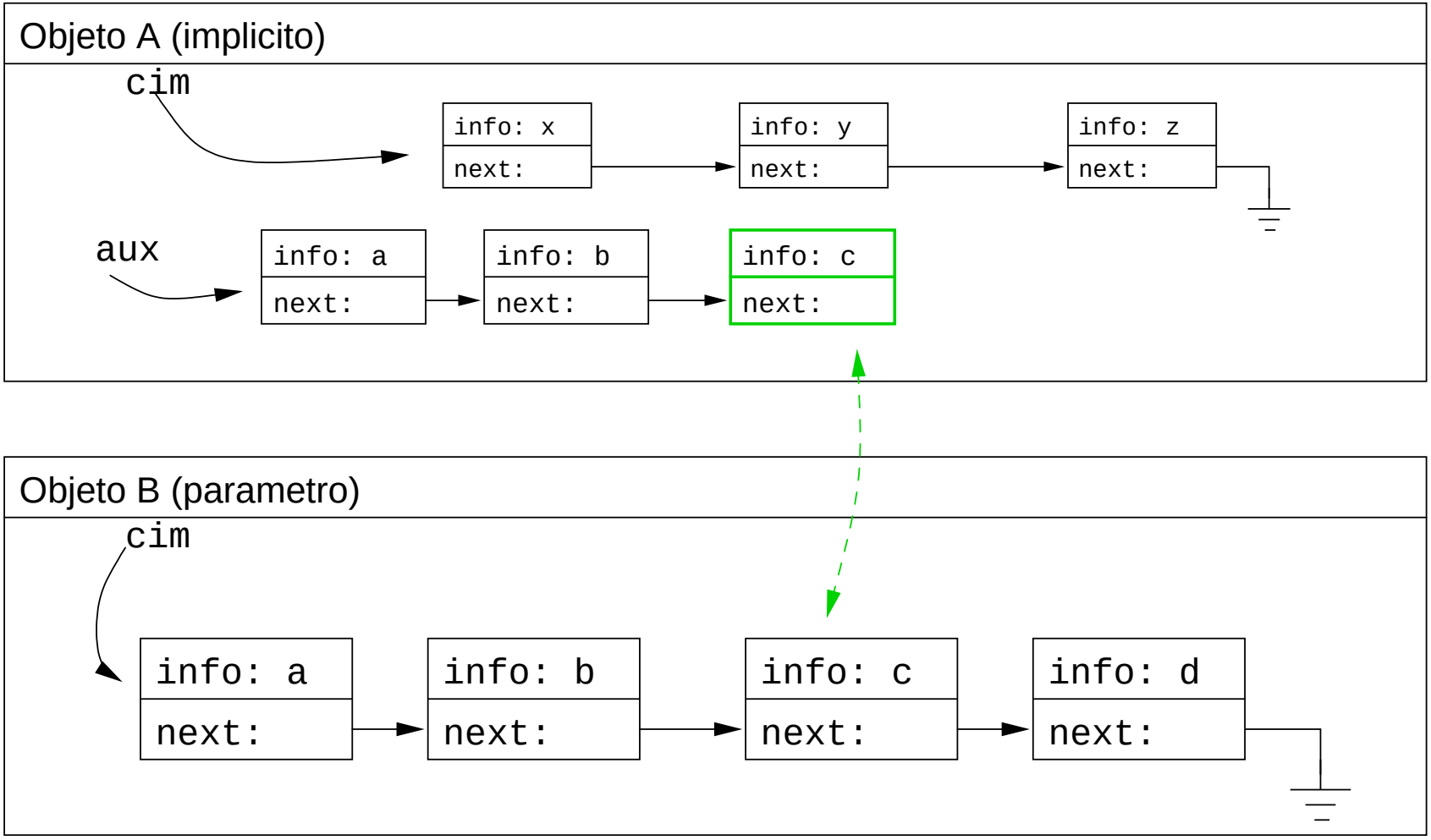
Ejemplo de Asignación. A = B



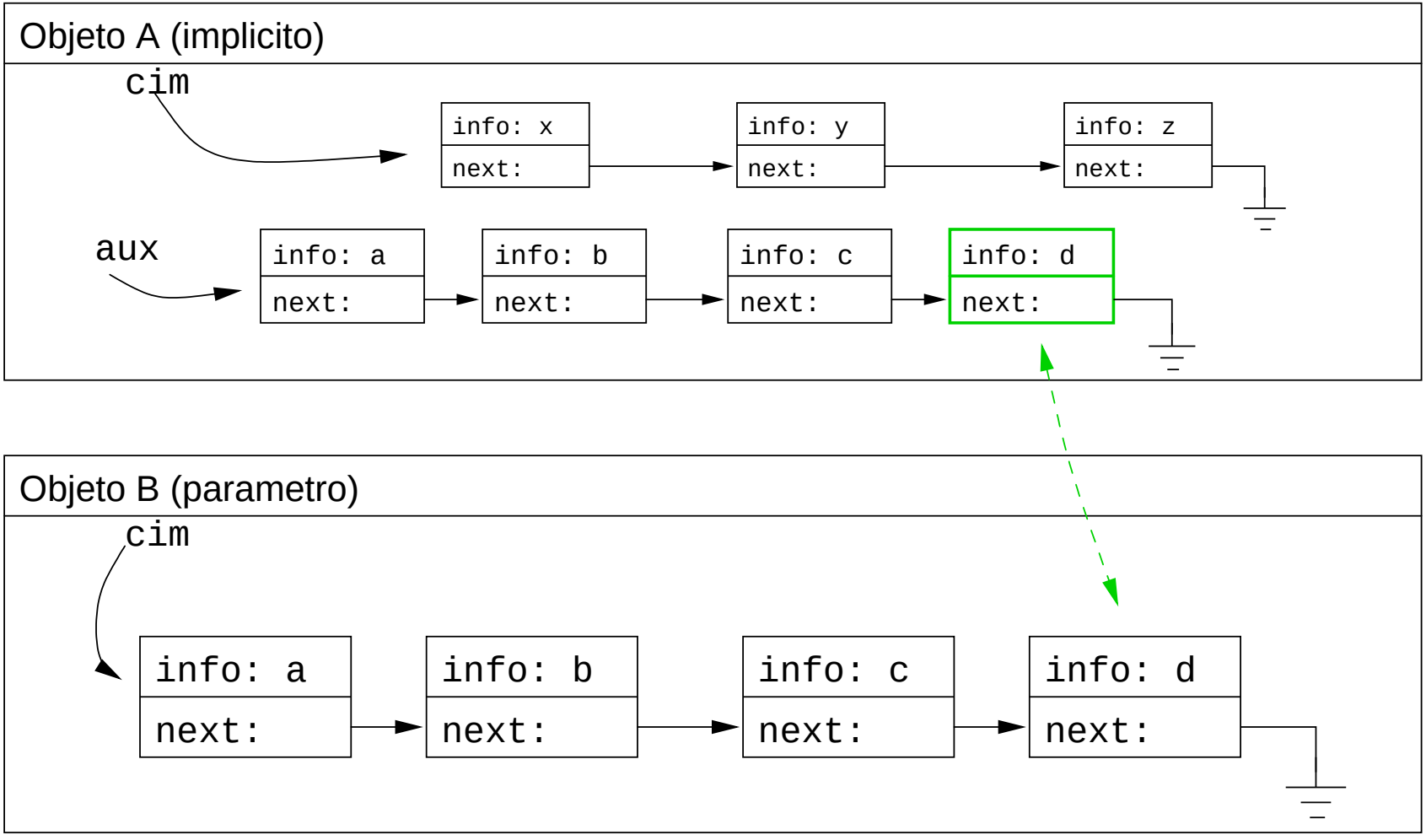
Ejemplo de Asignación. A = B



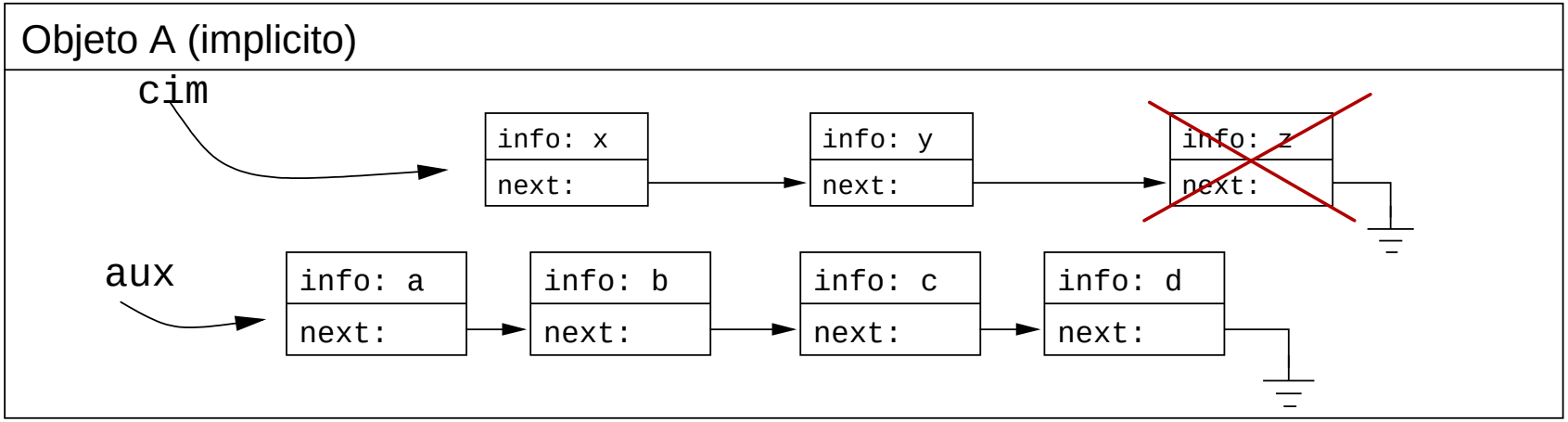
Ejemplo de Asignación. A = B

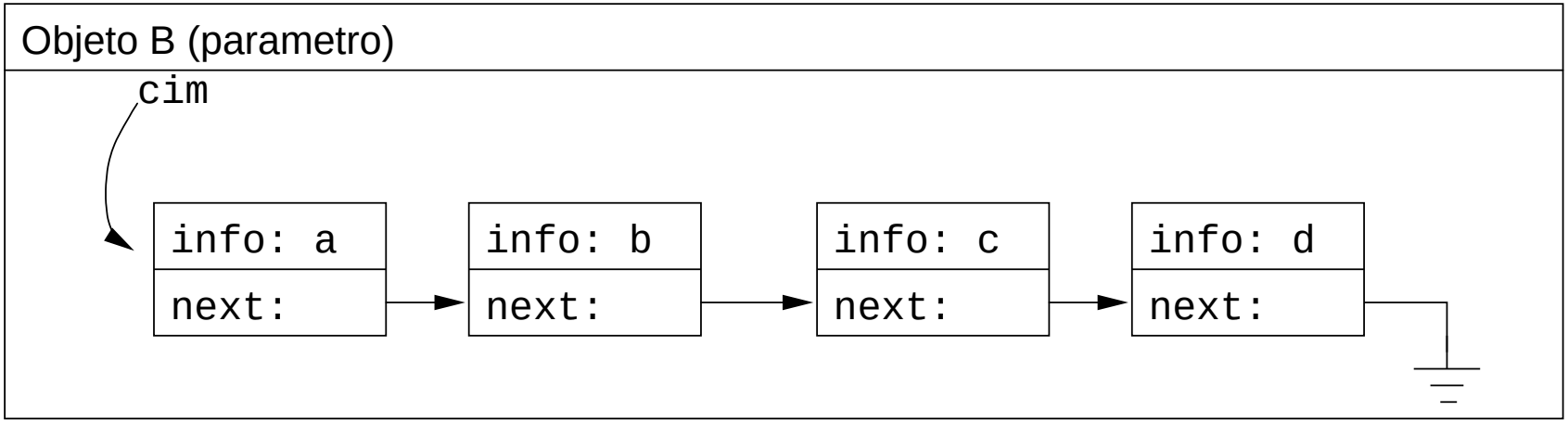


Ejemplo de Asignación. A = B

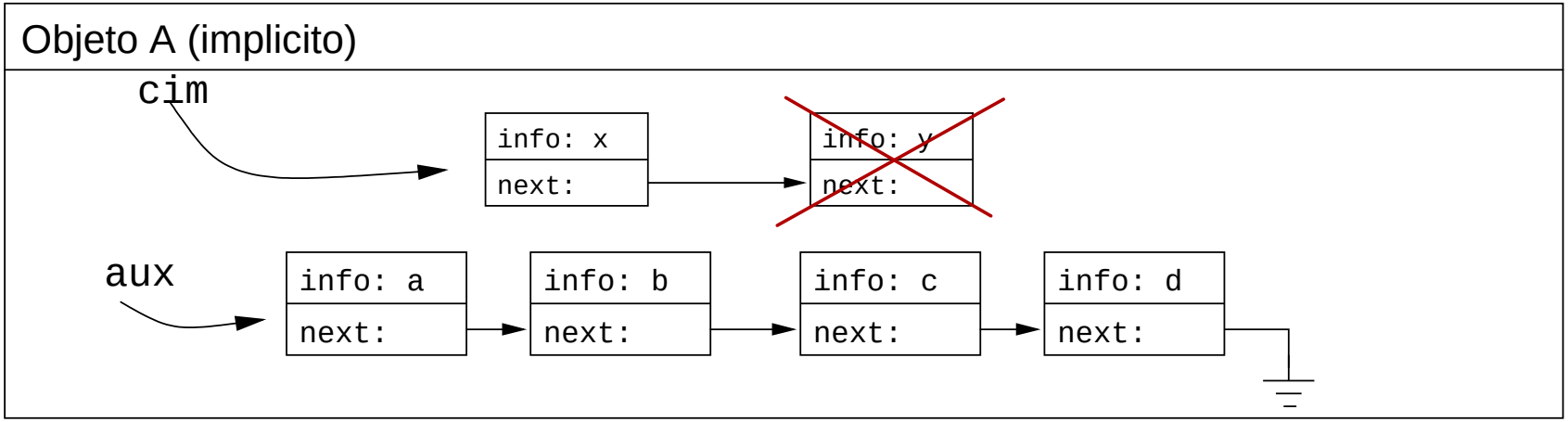


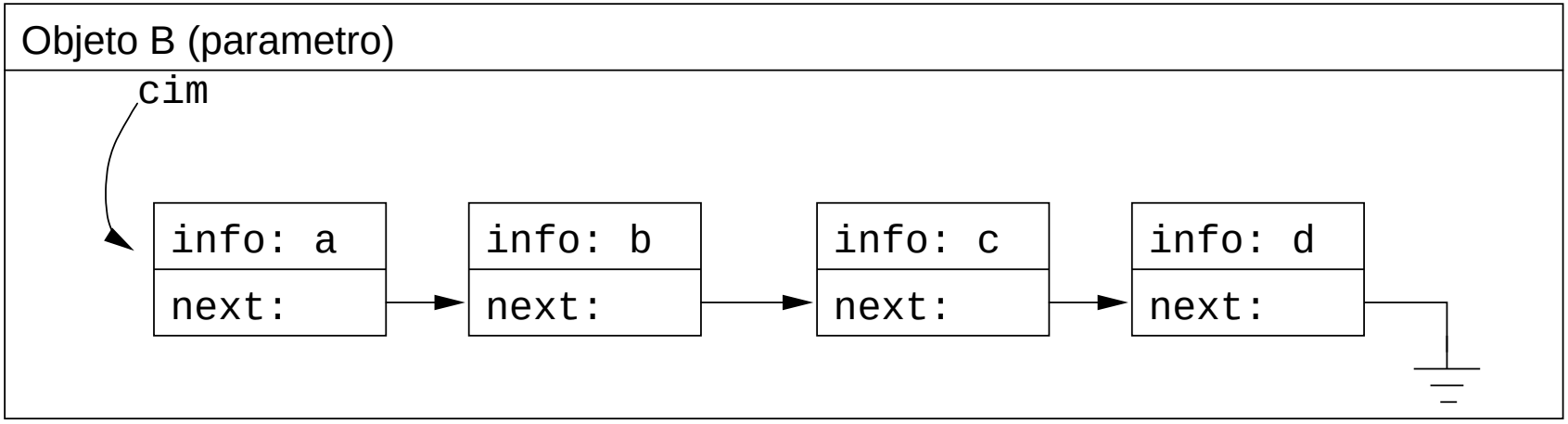
Ejemplo de Asignación. A = B



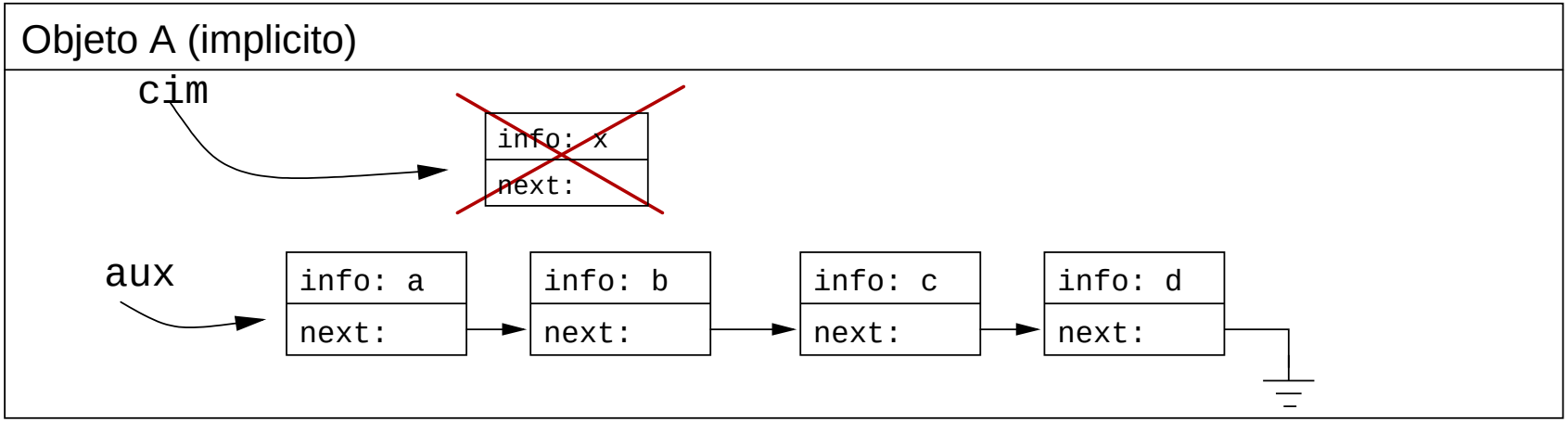


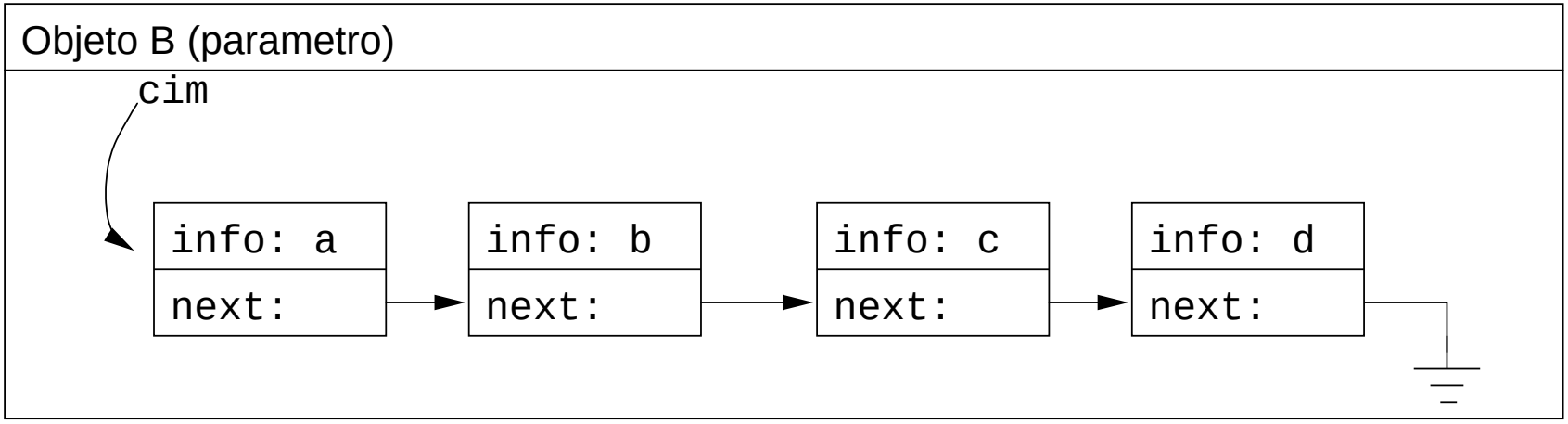
Ejemplo de Asignación. A = B



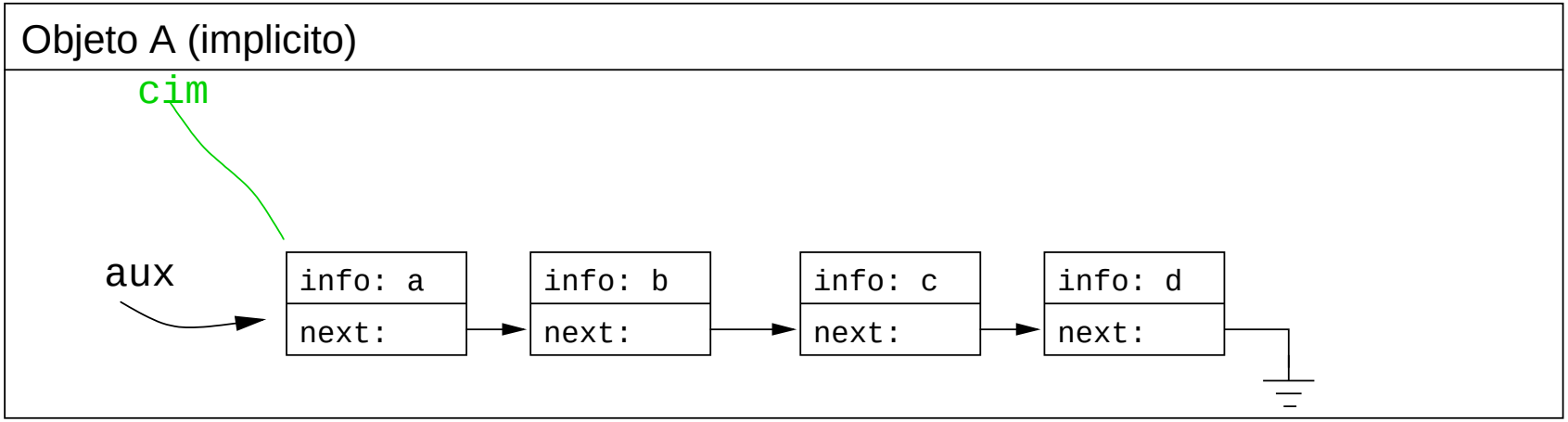


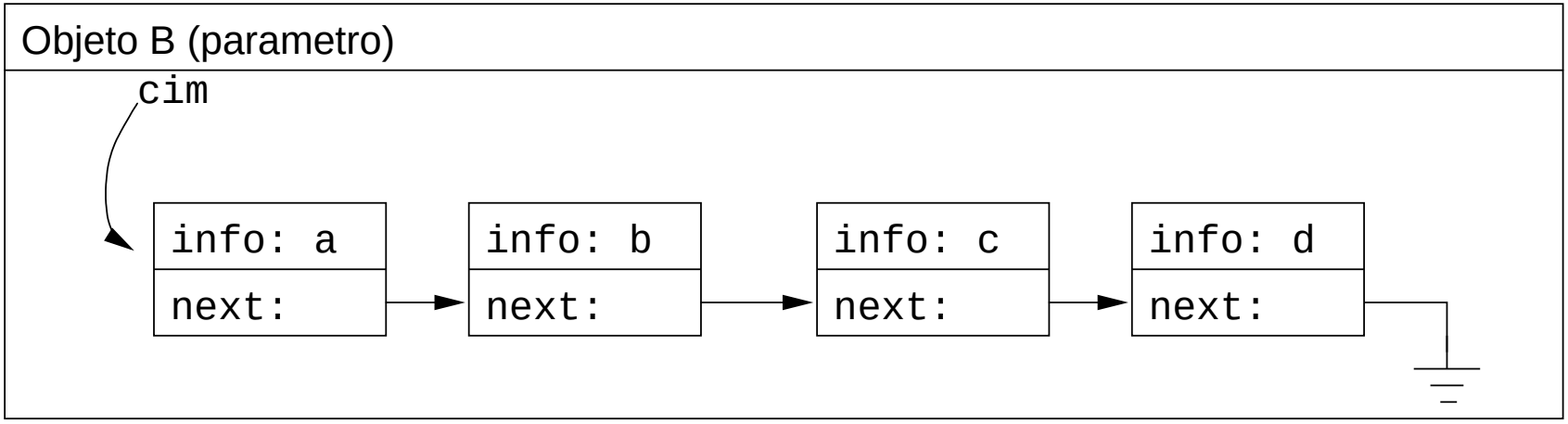
Ejemplo de Asignación. A = B





Ejemplo de Asignación. A = B





Tratamiento de Excepciones

- Las excepciones son anomalías que tienen lugar durante la ejecución de un programa y que impiden que continúe su ejecución normal.

Tratamiento de Excepciones

- Las excepciones son anomalías que tienen lugar durante la ejecución de un programa y que impiden que continúe su ejecución normal.
- Estas anomalías pueden ser tanto errores del usuario (abrir un fichero inexistente, etc.) como errores de lógica (una división por cero, un acceso a una posición inexistente de un vector, etc.) o bien errores del sistema o del *hardware*.

Tratamiento de Excepciones

- Las excepciones son anomalías que tienen lugar durante la ejecución de un programa y que impiden que continúe su ejecución normal.
- Estas anomalías pueden ser tanto errores del usuario (abrir un fichero inexistente, etc.) como errores de lógica (una división por cero, un acceso a una posición inexistente de un vector, etc.) o bien errores del sistema o del *hardware*.
- Hay varias formas de tratar las excepciones:

Tratamiento de Excepciones

- Las excepciones son anomalías que tienen lugar durante la ejecución de un programa y que impiden que continúe su ejecución normal.
- Estas anomalías pueden ser tanto errores del usuario (abrir un fichero inexistente, etc.) como errores de lógica (una división por cero, un acceso a una posición inexistente de un vector, etc.) o bien errores del sistema o del *hardware*.
- Hay varias formas de tratar las excepciones:
  - (Ignorarlas.)

Tratamiento de Excepciones

- Las excepciones son anomalías que tienen lugar durante la ejecución de un programa y que impiden que continúe su ejecución normal.
- Estas anomalías pueden ser tanto errores del usuario (abrir un fichero inexistente, etc.) como errores de lógica (una división por cero, un acceso a una posición inexistente de un vector, etc.) o bien errores del sistema o del *hardware*.
- Hay varias formas de tratar las excepciones:
  - (Ignorarlas.)
  - Abortar el programa.

Tratamiento de Excepciones

- Las excepciones son anomalías que tienen lugar durante la ejecución de un programa y que impiden que continúe su ejecución normal.
- Estas anomalías pueden ser tanto errores del usuario (abrir un fichero inexistente, etc.) como errores de lógica (una división por cero, un acceso a una posición inexistente de un vector, etc.) o bien errores del sistema o del *hardware*.
- Hay varias formas de tratar las excepciones:
  - (Ignorarlas.)
  - Abortar el programa.
  - Usar códigos de error.

Tratamiento de Excepciones

- Las excepciones son anomalías que tienen lugar durante la ejecución de un programa y que impiden que continúe su ejecución normal.
- Estas anomalías pueden ser tanto errores del usuario (abrir un fichero inexistente, etc.) como errores de lógica (una división por cero, un acceso a una posición inexistente de un vector, etc.) o bien errores del sistema o del *hardware*.
- Hay varias formas de tratar las excepciones:
  - (Ignorarlas.)
  - Abortar el programa.
  - Usar códigos de error.
  - Usar aserciones.

Tratamiento de Excepciones

- Las excepciones son anomalías que tienen lugar durante la ejecución de un programa y que impiden que continúe su ejecución normal.
- Estas anomalías pueden ser tanto errores del usuario (abrir un fichero inexistente, etc.) como errores de lógica (una división por cero, un acceso a una posición inexistente de un vector, etc.) o bien errores del sistema o del *hardware*.
- Hay varias formas de tratar las excepciones:
  - (Ignorarlas.)
  - Abortar el programa.
  - Usar códigos de error.
  - Usar aserciones.
  - ⋮

Tratamiento de Excepciones

- Las excepciones son anomalías que tienen lugar durante la ejecución de un programa y que impiden que continúe su ejecución normal.
- Estas anomalías pueden ser tanto errores del usuario (abrir un fichero inexistente, etc.) como errores de lógica (una división por cero, un acceso a una posición inexistente de un vector, etc.) o bien errores del sistema o del *hardware*.
- Hay varias formas de tratar las excepciones:
  - (Ignorarlas.)
  - Abortar el programa.
  - Usar códigos de error.
  - Usar aserciones.
  - ⋮
  - Usar el mecanismo de tratamiento de excepciones del lenguaje C++

Abortar el programa

- Terminación immediata del programa mediante `exit()` o `abort()` (declaradas en la librería standard `<stdlib>`).

```
#include <stdlib>
...
if (n <= 0) {
   cerr << "Mensaje de error" << endl;
   exit(1);
};
```

- Este tratamiento de errores como mínimo permite evitar la obtención de resultados incorrectos.

Códigos de Error

- Consisten en interrumpir la ejecución de una función para delegar el problema (mediante un código de error, posiblemente con información adicional) a la función que la ha invocado, ésta puede volver a delegar el problema, y así sucesivamente.

```
int f() {
   int res, ret;
   res = f1();
   if (res == EXITO) {
      res = f2();
      if (res == EXITO) {
         ret = EXITO;
      } else ret = ERROR2;
   } else ret = ERROR1;
   return ret;
}
```

- Los códigos de error obligan a extender la interfaz de las funciones para poder pasar información sobre el error a las funciones que las han invocado.

Aserciones

- La instrucción `assert()` (de hecho es una *macro*) comprueba una condición dada, interrumpiendo la ejecución del programa e imprimiendo un mensaje de error si la condición no se cumple.

```
assert(n > 0); // si n <= 0 llama abort()
```

Aserciones

- La instrucción `assert()` (de hecho es una *macro*) comprueba una condición dada, interrumpiendo la ejecución del programa e imprimiendo un mensaje de error si la condición no se cumple.

```
assert(n > 0); // si n <= 0 llama abort()
```

- Las aserciones dan escasa información sobre el tipo de error que se ha producido:

Aserciones

- La instrucción `assert()` (de hecho es una *macro*) comprueba una condición dada, interrumpiendo la ejecución del programa e imprimiendo un mensaje de error si la condición no se cumple.

```
assert(n > 0); // si n <= 0 llama abort()
```

- Las aserciones dan escasa información sobre el tipo de error que se ha producido:
  - No permiten corregir las situaciones de error

Aserciones

- La instrucción `assert()` (de hecho es una *macro*) comprueba una condición dada, interrumpiendo la ejecución del programa e imprimiendo un mensaje de error si la condición no se cumple.

```
assert(n > 0); // si n <= 0 llama abort()
```

- Las aserciones dan escasa información sobre el tipo de error que se ha producido:
  - No permiten corregir las situaciones de error
  - No liberan los recursos compartidos

Aserciones

- La instrucción `assert()` (de hecho es una *macro*) comprueba una condición dada, interrumpiendo la ejecución del programa e imprimiendo un mensaje de error si la condición no se cumple.

```
assert(n > 0); // si n <= 0 llama abort()
```

- Las aserciones dan escasa información sobre el tipo de error que se ha producido:
  - No permiten corregir las situaciones de error
  - No liberan los recursos compartidos
  - Abortan la ejecución del programa

Aserciones

- La instrucción `assert()` (de hecho es una *macro*) comprueba una condición dada, interrumpiendo la ejecución del programa e imprimiendo un mensaje de error si la condición no se cumple.

```
assert(n > 0); // si n <= 0 llama abort()
```

- Las aserciones dan escasa información sobre el tipo de error que se ha producido:
  - No permiten corregir las situaciones de error
  - No liberan los recursos compartidos
  - Abortan la ejecución del programa
- Sin embargo, como mínimo permiten evitar la obtención de resultados incorrectos.

Aserciones

- La instrucción `assert()` (de hecho es una *macro*) comprueba una condición dada, interrumpiendo la ejecución del programa e imprimiendo un mensaje de error si la condición no se cumple.

```
assert(n > 0); // si n <= 0 llama abort()
```

- Las aserciones dan escasa información sobre el tipo de error que se ha producido:
  - No permiten corregir las situaciones de error
  - No liberan los recursos compartidos
  - Abortan la ejecución del programa
- Sin embargo, como mínimo permiten evitar la obtención de resultados incorrectos.
- En general se utilizan para la comprobación de precondiciones y postcondiciones.

Excepciones en C++

El lenguaje C++ incorpora un mecanismo unificado de tratamiento de excepciones. Permite tratar las excepciones como objetos de una clase determinada.

- Un bloque `try { ...}` contiene instrucciones que pueden ocasionar la generación de una excepción que se quiere controlar.

Excepciones en C++

El lenguaje C++ incorpora un mecanismo unificado de tratamiento de excepciones. Permite tratar las excepciones como objetos de una clase determinada.

- Un bloque `try { ...}` contiene instrucciones que pueden ocasionar la generación de una excepción que se quiere controlar.
- La instrucción `throw` *`expresión`* genera una excepción al *lanzar* un objecto de la clase de la expresión dada. Esta instrucción se tiene que ejecutar:

Excepciones en C++

El lenguaje C++ incorpora un mecanismo unificado de tratamiento de excepciones. Permite tratar las excepciones como objetos de una clase determinada.

- Un bloque `try { ...}` contiene instrucciones que pueden ocasionar la generación de una excepción que se quiere controlar.
- La instrucción `throw` *`expresión`* genera una excepción al *lanzar* un objecto de la clase de la expresión dada. Esta instrucción se tiene que ejecutar:
  - Dentro de un bloque `try`,

Excepciones en C++

El lenguaje C++ incorpora un mecanismo unificado de tratamiento de excepciones. Permite tratar las excepciones como objetos de una clase determinada.

- Un bloque `try { ...}` contiene instrucciones que pueden ocasionar la generación de una excepción que se quiere controlar.
- La instrucción `throw` *`expresión`* genera una excepción al *lanzar* un objecto de la clase de la expresión dada. Esta instrucción se tiene que ejecutar:
  - Dentro de un bloque `try`,
  - O bien dentro de una función que haya sido invocada dentro de un bloque `try`

Excepciones en C++

El lenguaje C++ incorpora un mecanismo unificado de tratamiento de excepciones. Permite tratar las excepciones como objetos de una clase determinada.

- Un bloque `try { ...}` contiene instrucciones que pueden ocasionar la generación de una excepción que se quiere controlar.
- La instrucción `throw` *`expresión`* genera una excepción al *lanzar* un objecto de la clase de la expresión dada. Esta instrucción se tiene que ejecutar:
  - Dentro de un bloque `try`,
  - O bien dentro de una función que haya sido invocada dentro de un bloque `try`
  - O dentro de una función que invoca a una función que ha sido invocada dentro de un bloque `try`, etc.

Excepciones en C++

- Un bloque `catch ( `*tipo*` ) { ...}` realiza el tratamiento de las excepciones del mismo tipo que el del tipo dado.

Excepciones en C++

- Un bloque `catch ( `*tipo*` ) { ...}` realiza el tratamiento de las excepciones del mismo tipo que el del tipo dado.
  - Este bloque tiene que ir inmediatamente a continuación de un bloque `try`.

Excepciones en C++

- Un bloque `catch ( tipo ) { ...}` realiza el tratamiento de las excepciones del mismo tipo que el del tipo dado.
  - Este bloque tiene que ir inmediatamente a continuación de un bloque `try`.
  - Si entre paréntesis ponemos `(...)` entonces se captura cualquier excepción, sea del tipo que sea.

Excepciones en C++

Esquema de tratamiento con excepciones

```
int main() {
   ...
   try {
      ... // codigo que puede generar excepciones
          // de la clase mi_error y de otras
   }
   catch (mi_error e) {
      ... // codigo de tratamiento para
          // objetos de la clase mi_error;
          // dentro de este bloque la excepcion
          // capturada se conoce como 'e'
   }
   catch (std::range_error) {
      cerr << "fuera de rango" << endl;
   }
   catch (std::bad_alloc) {
      cerr << "no hay memoria disponible" << endl;
   }
   catch (...) {
      ... // para cualquier otra excepcion
   }
}
```

Excepciones en C++

- El mecanismo de excepciones de C++ es muy conveniente ya que permite dividir el programa en secciones separadas para tratar las situaciones normales y las excepcionales. De hecho, el mecanismo consta de dos componentes principales:

Excepciones en C++

- El mecanismo de excepciones de C++ es muy conveniente ya que permite dividir el programa en secciones separadas para tratar las situaciones normales y las excepcionales. De hecho, el mecanismo consta de dos componentes principales:
  - La detección y generación de una excepción, y

Excepciones en C++

- El mecanismo de excepciones de C++ es muy conveniente ya que permite dividir el programa en secciones separadas para tratar las situaciones normales y las excepcionales. De hecho, el mecanismo consta de dos componentes principales:
  - La detección y generación de una excepción, y
  - El tratamiento, propiamente dicho, de la excepción.

Excepciones en C++

- El mecanismo de excepciones de C++ es muy conveniente ya que permite dividir el programa en secciones separadas para tratar las situaciones normales y las excepcionales. De hecho, el mecanismo consta de dos componentes principales:
  - La detección y generación de una excepción, y
  - El tratamiento, propiamente dicho, de la excepción.
- La ejecución normal del programa se suspende en el momento en que se genera una excepción.

Excepciones en C++

- El mecanismo de excepciones de C++ es muy conveniente ya que permite dividir el programa en secciones separadas para tratar las situaciones normales y las excepcionales. De hecho, el mecanismo consta de dos componentes principales:
  - La detección y generación de una excepción, y
  - El tratamiento, propiamente dicho, de la excepción.
- La ejecución normal del programa se suspende en el momento en que se genera una excepción.
- El tratamiento de excepciones se ubica en puntos concretos del programa (en las cláusulas `catch`).

Excepciones en C++

- El mecanismo de excepciones de C++ es muy conveniente ya que permite dividir el programa en secciones separadas para tratar las situaciones normales y las excepcionales. De hecho, el mecanismo consta de dos componentes principales:
  - La detección y generación de una excepción, y
  - El tratamiento, propiamente dicho, de la excepción.
- La ejecución normal del programa se suspende en el momento en que se genera una excepción.
- El tratamiento de excepciones se ubica en puntos concretos del programa (en las cláusulas `catch`).
- La ejecución del programa no vuelve a la parte del programa que ha generado la excepción, sino que continúa con la parte del programa que trata la excepción.

Excepciones en C++

- Por ejemplo, la siguiente clase (sencilla) permite lanzar objetos para las excepciones relacionadas con la división por cero.

```
#include <string>

class ZeroDivide {
public:
   ZeroDivide() {
      message = "Error: division por cero";
   }
   void print() {
      cout << message << endl;
   }

private:
   string message;
};
```

Excepciones en C++

- Las excepciones puede ser lanzadas por cualquier función.

```
float quotient(int n1, int n2) {
   if (n2 == 0) throw ZeroDivide();
   // genera la excepcion y termina la ejecucion de la funcion
   return (float) n1 / n2;
}
```

- El tratamiento de excepciones se realiza dentro de un bloque `catch`.

```
int main() {
   int n1, n2;
   ...
   try {
      float res = quotient(n1, n2);
      ...
   }
   catch (ZeroDivide error) {
      error.print();
} }
```

Excepciones en C++

- La declaración de una función puede incluir una *especificación de excepción*:

```
void f(int x);
// puede generar o propagar cualquier excepcion

void f(int x) throw();
// no genera o propaga ninguna excepcion

void f(int x) throw(X, Y, Z);
// puede generar o propagar excepciones de los tipos
// X, Y y Z
```

- En este último caso la función podría ser:

```
void f(int x) throw(X, Y, Z) {
   if (...) throw X();
   if (...) throw Y();
   if (...) throw Z();
   ... }
```

Excepciones en C++

- Una excepción puede ser capturada por un `catch`, tratada y relanzada para ser recogida por un bloque `try` más externo.

```
void f(int x) {
   try {
      ...
   }
   catch (error e) {
      ...     // tratar (parcialmente) el error e
      throw;  //relanzar el error
   }
}

int main() {
   try {
      ...
      f(x);
      ...
   }
   catch (error e) { ...}
}
```

Excepciones en C++

- La ventaja más evidente del mecanismo de excepciones de C++ es que nos libera de la necesidad de tener que incluir código para propagar las excepciones, contrariamente a lo que ocurre con las soluciones basadas en códigos de error.

```
void f(int x) {
   ...
   g(y); // si 'g' lanza una excepcion, 'f' no hace nada
         // con ella, se limita a propagarla;
         // la ejecucion de 'f' termina inmediatamente
   ...
}
```

Excepciones en C++

- El siguiente ejemplo muestra lo que **no** debe hacerse, ya que es completamente redundante:

```
void f(int x) {
   ...
   try {
      g(y);
   }
   catch(error e) {  //
      throw;          // ¡Redundante!
   }                  //
   ...
}
```

Plantillas para Funciones

- Función para calcular el máximo de dos enteros:

```
int max(int a, int b) {
   return a > b ? a : b;
}
```

- Función para calcular el máximo de dos reales:

```
double max(double a, double b) {
   return a > b ? a : b;
}
```

- Función para calcular el máximo de dos caracteres:

```
char max(char a, char b) {
   return a > b ? a : b;
}
```

Plantillas para Funciones

- Si tenemos la clase `racional`,

```
class racional {
   ...
   static bool operator>(racional a, racional b);
};
```

- también haremos:

```
racional max(racional a, racional b) {
   return a > b ? a : b;
}
```

Plantillas para Funciones

- En general, para calcular el máximo de cualquier par de datos de tipo `T`:

```
T max(T a, T b) {
   return a > b ? a : b;
}
```

- o mejor (para evitar copias de los objetos en el paso de parámetros)

```
T max(const T& a, const T& b) {
   return a > b ? a : b;
}
```

- C++ proporciona un mecanismo para describir funciones o clases genéricas: las plantillas (*templates*).

Plantillas para Funciones

Plantilla para calcular el máximo de cualquier par de datos de tipo T:

```
template <typename T>
T max(const T& a, const T& b) {
   return a > b ? a : b;
}

void f() {
   double d1 = 3.5, d2 = 4.2;
   cout << max(d1, d2);

   int i1 = 3, i2 = 1;
   cout << max(i1, i2) + 1;

   racional r1(10, 20), r2(1, 4);
   racional r = max(r1, r2);
}
```

La función genérica `max` se puede aplicar sobre cualquier tipo que tenga definido el operador >.

InsertionSort genérico

```
template <typename T>
void insertion_sort(T a[], int n) {
   for (int j = 1; j < n; j++) {
      T key = a[j];
      int i = j-1;
      while (i >= 0 && a[i] > key) {
         a[i+1] = a[i];
         i--;
      }
      a[i+1] = key;
} }
```

Para poder aplicar esta plantilla sobre un tipo T, es necesario que T tenga disponibles las operaciones siguientes:

- Constructor por copia: `T (const T&)`
- Asignación: `T& operator=(const T&)`
- Comparación: `bool operator>(const T&, const T&)`

QuickSort genérico

```
template <typename T>
void quicksort(T a[], int p, int r) {
   if (p < r) {
      int q = partition(a, p, r);
      quicksort(a, p, q);
      quicksort(a, q + 1, r);
   }
}
```

Para poder aplicar esta plantilla sobre un tipo `T`, es necesario que `T` tenga disponibles las operaciones siguientes:

- Constructor por copia: `T (const T&)`
- Asignación: `T& operator=(const T&)`
- Comparación: `bool operator>(const T&, const T&)`

QuickSort genérico

```
template <typename T>
void swap(T& x, T& y) {
   T z = x; x = y; y = z;
}

template <typename T>
int partition(T a[], int p, int r) {
   T x = a[p];
   int i = p - 1, j = r + 1;
   while (i < j) {
      do j--; while (a[j] > x);
      do i++; while (x > a[i]);
      if (i < j)
         swap(a[i], a[j]);
   }
   return j;
}
```

Plantillas para la clase Pareja

Implementación

```
template <typename T1, typename T2>
class Pareja {
  public:
     Pareja(const T1&, const T2&);
     T1 primero() const;
     T2 segundo() const;

  private:
     T1 v1;
     T2 v2;
};

template <typename T1, typename T2>
T1 Pareja<T1, T2>::primero() const {
   return v1;
}

template <typename T1, typename T2>
T2 Pareja<T1, T2>::segundo() const {
   return v2;
}
```

Plantillas para la clase Pareja

Ejemplo de Uso

```
void f() {
   Pareja<int, double> par(22, 3.1416);

   int i = par.primero();     // 'i' vale 22
   double d = par.segundo();  // 'd' vale 3.1416

   int j = par.segundo();     // Error!

   Pareja<char*, int> par2("hola", 4);
   Pareja<racional, int> par3(racional(3,2),1);
   typedef Pareja<double, double> punto2d;
}
```

La declaración `typedef T identif` indica que `identif` es un alias o sinónimo del tipo `T`. Por ejemplo, arriba se define `punto2d` como un sinónimo para el tipo `Pareja<double, double>`.

Pila genérica

Especificación del TAD

```
TAD PILA<T>
genero Pila
ops
    crea: enter → Pila
    vacia: Pila → bool
    cima: Pila → T
    apilar: Pila T → Pila
    desapilar: Pila → Pila
fops
```

Pila genérica

Interfaz (Pila.hpp):

```
class ErrorPilaVacia { };
class ErrorPilaLlena { };

template <typename T>
class Pila {
  public:
     Pila(int tam);
     Pila(const Pila<T>& p);
     ~Pila();
     Pila<T>& operator=(const Pila<T>& p);

     T cima() const throw(ErrorPilaVacia);

     // se apila mediante el operador <<
     Pila<T>& operator<<(const T& x) throw(ErrorPilaLlena);

     // el operador >> hace cima + desapilar
     Pila<T>& operator>>(T& x) throw(ErrorPilaVacia);

     bool vacia() const;
...
```

Pila genérica

Interfaz (Pila.hpp) (continuación):

```
...
  private:
     int s;      // tamaño maximo
     int n;      // numero de elementos
     T* t;       // tabla de elementos

     // metodos privados
     void apilar(const T& x);
     void desapilar();
     static void copia_pila(T* tdest,T* torig, int n);
};
#include "Pila.t"
```

Pila genérica

Ejemplo de uso:

```
void f() {
   Pila<int> p(15);
   p << 3 << 4 << 5; // apilamos 3, 4 y 5, por este orden
   int a, b, c;
   p >> a >> b >> c; // y luego se desapilan
   // a = 5, b = 4, c = 3

   Pila<double> p2(3);
   try {
      p2 << 3.1416 << 9.81 << 3.1416
      << 9.81 << 3.1416 << 9.81;
   }
   catch (ErrorPilaLlena e) {
      cerr << "Te has pasado!\n" << flush;
   }

   double x;
   Pila<double> p3 = p2;
   while (!p3.vacia()) {
      p3 >> x;
      cout << x << endl;
}  }
```

Pila genérica

Implementación (Pila.t):

```
template <typename T>
Pila<T>::Pila(int tam)
: s(tam), n(0), t(new T[s]) { }

template <typename T>
Pila<T>::Pila(const Pila<T>& p)
: s(p.s), n(p.n), t(NULL) {
   t = new T[s];
   try {
      copia_pila(t, p.t, p.n);
   }
   catch (...) {  // si algo ha ido mal con la copia
      delete[] t;  // destruye la tabla t asociada a
      throw;       // a la nueva pila y relanza la excep.
   }
}

template <typename T>
Pila<T>::~Pila() {
   delete[] t;
}
```

Pila genérica

Implementación (Pila.t) (continuación):

```
template <typename T>
Pila<T>& Pila<T>::operator=(const Pila<T>& p) {
   if (&p != this) {
      T* aux = new T[p.s];
      try {
         copia_pila(aux, p.t, p.n);
      }
      catch (...) {
         delete[] aux;
         throw;
      }
      delete[] t;
      t = aux; s = p.s; n = p.n;
   }
   return *this;
}

template <typename T>
void Pila<T>::copia_pila(T* tdest, T* torig, int n) {
   for (int i = 0; i < n; i++)
      tdest[i] = torig[i];
}
```

Pila genérica

Implementación (Pila.t) (continuación):

```
template <typename T>
void Pila<T>::apilar(const T& x) {
   if (n == s) throw ErrorPilaLlena();
   t[n] = x;
   n++;
}

template <typename T>
void Pila<T>::desapilar() {
   if (vacia()) throw ErrorPilaVacia();
   n--;
}

template <typename T>
bool Pila<T>::vacia() const {
   return n == 0;
}
```

Pila genérica

Implementación (Pila.t) (continuación):

```
template <typename T>
T Pila<T>::cima() const
                throw (ErrorPilaVacia) {
   if (vacia()) throw ErrorPilaVacia();
   return t[n-1];
}

template <typename T>
Pila<T>& operator<<(const T& x)
                  throw (ErrorPilaLlena) {
   apilar(x);
   return *this;
}

template <typename T>
Pila<T>& operator>>(T& x)
                  throw (ErrorPilaVacia) {
   x = cima();
   desapilar();
   return *this;
}
```

Pila genérica

- Cuando se desapila un objeto sólo se decrementa el índice n, y el objeto sigue estando almacenado en el componente n del `array`, sin ser destruído. La destrucción tendrá lugar si posteriormente se apilan nuevos objetos que lo "sobreescriban" o cuando se destruya la pila.

Pila genérica

- Cuando se desapila un objeto sólo se decrementa el índice n, y el objeto sigue estando almacenado en el componente n del `array`, sin ser destruído. La destrucción tendrá lugar si posteriormente se apilan nuevos objetos que lo "sobreescriban" o cuando se destruya la pila.
- La clase `T` debe contar con el operador de asignación: se usa en `apilar` (cuando hacemos `t[n] = x`), en `cima`, en la constructora por copia de `Pila`, en la redefinición de la asignación de la clase `Pila`, etc.

Pila genérica

- El método destructor `~T()` se invoca al destruir el `array` de objetos en `~Pila()`.

Pila genérica

- El método destructor `~T()` se invoca al destruir el `array` de objetos en `~Pila()`.
- La clase `T` debe tener una constructora por defecto: para reclamar a memoria dinámica una tabla de `T` de tamaño `s`, tanto en la constructora `Pila(int tam)` como en la constructora por copia `Pila(const Pila<T>& p)`, se utiliza `new[]`, que invoca a la constructora por defecto de la clase `T` para inicializar los componentes del `array`.

Factores a tener en cuenta

- Es recomendable dar un nuevo nombre a los tipos genéricos instanciados que se utilizan frecuentemente:

```
typedef Taula<double,100> Taula100Reals;
Taula100Reals t;
```

- Se pueden *especializar* las plantillas, por ejemplo, dar una implementación específica para `Pila<int>` diferente de la general para `Pila<T>`.
- Las plantillas pueden recibir valores por defecto:

```
template <typename T, int size = 100> ...
```

- C++ resuelve la generecidad en tiempo de compilación. El compilador duplica el código tantas veces como sea necesario.

Factores a tener en cuenta

- El código de las plantillas es tan eficiente como si se definiesen tantas clases como instanciaciones del tipo genérico se usen.

Factores a tener en cuenta

- El código de las plantillas es tan eficiente como si se definiesen tantas clases como instanciaciones del tipo genérico se usen.
- Si se utilizan muchas plantillas con tipos diferentes crece la cantidad de código ejecutable generado.

Factores a tener en cuenta

- El código de las plantillas es tan eficiente como si se definiesen tantas clases como instanciaciones del tipo genérico se usen.
- Si se utilizan muchas plantillas con tipos diferentes crece la cantidad de código ejecutable generado.
- Puede suceder que se duplique inútilmente el código de una clase genérica instanciada para un mismo tipo. Ejemplo: se usan `Pila<int>`'s en dos módulos independientes y compilados por separado, que luego forman parte del mismo programa.

Introducción a la Librería Estándar de C++

- La librería estándar de C++ ofrece una gran variedad de clases y funciones útiles.

Introducción a la Librería Estándar de C++

- La librería estándar de C++ ofrece una gran variedad de clases y funciones útiles.
- Entre ellas hemos de destacar los *streams* para entrada/salida, y los *strings*.

Introducción a la Librería Estándar de C++

- La librería estándar de C++ ofrece una gran variedad de clases y funciones útiles.
- Entre ellas hemos de destacar los *streams* para entrada/salida, y los *strings*.
- Además incluye la llamada STL (*Standard Template Library*) que define diversas clases genéricas denominadas *contenedores* y algoritmos sobre éstas.

Introducción a la Librería Estándar de C++

- La librería estándar de C++ ofrece una gran variedad de clases y funciones útiles.
- Entre ellas hemos de destacar los *streams* para entrada/salida, y los *strings*.
- Además incluye la llamada STL (*Standard Template Library*) que define diversas clases genéricas denominadas *contenedores* y algoritmos sobre éstas.
- A continuación veremos algunos de los contenedores de la STL.

Standard Template Library (STL)

- Vectores: `vector<T>`.

Standard Template Library (STL)

- Vectores: `vector<T>`.
  - secuencia de tamaño variable y dinámico

Standard Template Library (STL)

- Vectores: `vector<T>`.
  - secuencia de tamaño variable y dinámico
  - acceso aleatorio

Standard Template Library (STL)

- Vectores: `vector<T>`.
  - secuencia de tamaño variable y dinámico
  - acceso aleatorio
  - inserciones y borrados por el final en tiempo constante

Standard Template Library (STL)

- Vectores: `vector<T>`.
  - secuencia de tamaño variable y dinámico
  - acceso aleatorio
  - inserciones y borrados por el final en tiempo constante
- Deques: `deque<T>`

Standard Template Library (STL)

- Vectores: `vector<T>`.
  - secuencia de tamaño variable y dinámico
  - acceso aleatorio
  - inserciones y borrados por el final en tiempo constante
- Deques: `deque<T>`
  - secuencia de tamaño variable y dinámico

Standard Template Library (STL)

- Vectores: `vector<T>`.
  - secuencia de tamaño variable y dinámico
  - acceso aleatorio
  - inserciones y borrados por el final en tiempo constante
- Deques: `deque<T>`
  - secuencia de tamaño variable y dinámico
  - acceso aleatorio

Standard Template Library (STL)

- Vectores: `vector<T>`.
  - secuencia de tamaño variable y dinámico
  - acceso aleatorio
  - inserciones y borrados por el final en tiempo constante
- Deques: `deque<T>`
  - secuencia de tamaño variable y dinámico
  - acceso aleatorio
  - inserciones y borrados por el principio y por el final en tiempo constante

Standard Template Library (STL)

- Listas: `list<T>`

Standard Template Library (STL)

- Listas: `list<T>`
  - secuencia de tamaño variable y dinámico

Standard Template Library (STL)

- Listas: `list<T>`
  - secuencia de tamaño variable y dinámico
  - acceso secuencial

Standard Template Library (STL)

- Listas: `list<T>`
  - secuencia de tamaño variable y dinámico
  - acceso secuencial
  - inserciones y borrados en cualquier posición designada por un iterador en tiempo constante

Standard Template Library (STL)

- Listas: `list<T>`
  - secuencia de tamaño variable y dinámico
  - acceso secuencial
  - inserciones y borrados en cualquier posición designada por un iterador en tiempo constante
- Conjuntos: `set<K>`

Standard Template Library (STL)

- Listas: `list<T>`
  - secuencia de tamaño variable y dinámico
  - acceso secuencial
  - inserciones y borrados en cualquier posición designada por un iterador en tiempo constante
- Conjuntos: `set<K>`
  - claves únicas

Standard Template Library (STL)

- Listas: `list<T>`
  - secuencia de tamaño variable y dinámico
  - acceso secuencial
  - inserciones y borrados en cualquier posición designada por un iterador en tiempo constante
- Conjuntos: `set<K>`
  - claves únicas
  - recuperación rápida de las claves ($O(\log n)$)

Standard Template Library (STL)

- Listas: `list<T>`
  - secuencia de tamaño variable y dinámico
  - acceso secuencial
  - inserciones y borrados en cualquier posición designada por un iterador en tiempo constante
- Conjuntos: `set<K>`
  - claves únicas
  - recuperación rápida de las claves ($O(\log n)$)
- Arrays asociativos: `map<K,I>`

Standard Template Library (STL)

- Listas: `list<T>`
  - secuencia de tamaño variable y dinámico
  - acceso secuencial
  - inserciones y borrados en cualquier posición designada por un iterador en tiempo constante
- Conjuntos: `set<K>`
  - claves únicas
  - recuperación rápida de las claves ($O(\log n)$)
- Arrays asociativos: `map<K,I>`
  - claves únicas de tipo `K` con información asociada de tipo `I`

Standard Template Library (STL)

- Listas: `list<T>`
  - secuencia de tamaño variable y dinámico
  - acceso secuencial
  - inserciones y borrados en cualquier posición designada por un iterador en tiempo constante
- Conjuntos: `set<K>`
  - claves únicas
  - recuperación rápida de las claves ($O(\log n)$)
- Arrays asociativos: `map<K,I>`
  - claves únicas de tipo `K` con información asociada de tipo `I`
  - recuperación rápida de la información asociada a las claves ($O(\log n)$)

La clase `string`

- Un *string* es una secuencia de caracteres. La clase `string` ofrece operaciones para acceso indexado, copia, comparación, concatenación, búsqueda y extracción de subcadenas, lectura y escritura, etc.
- Se puede utilizar una cadena constante, encerrada entre comillas dobles, como valor del tipo:

```
string s = "hola"; string t;
t = s + " y adios";
```

- Un carácter (`char`) también puede emplearse como un valor del tipo, excepto para inicializar:

```
string s = 'h'; // error!
string t = w;   // uso del constructor por copia
t = t + '*';    // ok!
```

- Los operadores + y += son de concatenación. También puede usarse el método append.

La clase `string`

- Los operadores ==, !=, <, <=, etc. de comparación entre strings están definidos, con los significados habituales (basados en el orden derivado del tipo `char`).
- Se puede acceder al carácter i-ésimo de un string `s` mediante `s[i]` o `s.at(i)`. La primera no verifica el rango y la segunda sí, pero la primera es más eficiente. Los caracteres de un string `s` se indexan de 0 a `s.length()-1`. Pero los strings **no** son arrays de caracteres.
- Los operadores >> y << para entrada y salida están definidos.

```
string nombre;
cout << "Como te llamas? " << endl;
cin >> nombre;
cout << "Hola " + nombre + '!' << endl;
```

La clase `vector`

- Un objeto de la clase `vector` es conceptualmente similar a un array, pero evita algunas de las desventajas de los arrays y ofrece funcionalidades adicionales.

La clase `vector`

- Un objeto de la clase `vector` es conceptualmente similar a un array, pero evita algunas de las desventajas de los arrays y ofrece funcionalidades adicionales.
- El tamaño inicial de un vector se puede indicar en el momento de creación, pero puede aumentar si se añaden elementos por el final, p.e. mediante el método `push_back`.

La clase `vector`

- Un objeto de la clase `vector` es conceptualmente similar a un array, pero evita algunas de las desventajas de los arrays y ofrece funcionalidades adicionales.
- El tamaño inicial de un vector se puede indicar en el momento de creación, pero puede aumentar si se añaden elementos por el final, p.e. mediante el método `push_back`.
- La clase ofrece métodos de acceso por índice: bien mediante el operador de indexación tradicional `[]`, bien mediante el método `at` que chequea el rango y lanza una excepción en su caso.

La clase `vector`

- Como clase de contenedor estándar ofrece un repertorio de operaciones muy extenso, incluyendo algunas que no son típicas de los vectores (p.e. inserción y borrado de elementos en posiciones intermedias).
- El número de elementos del vector es accesible a través del método `size`.
- La clase de los elementos debe tener definida la construcción por copia, la asignación y el operador de igualdad; la semántica de estas operaciones debe ser coherente:

```
T a = b; // por copia
c = b;   // asignacion
bool ok = (a == c); // debe ser 'true'
bool ok = (a == b); // debe ser 'true'
```

La clase vector

Ejemplo de uso:

```
#include <vector>
#include <iostream>
#include "client.hpp"

int main() {
   vector<int> v;
   vector<client> clients;
   vector<vector<double> > matriz;
   for (int i = 0; i < 10; i++)
      v.push_back(i);
   for (int i = 0; i < v.size(); i++)
      cout << v[i] << " ";
   try {
      for (int i = 0; i < 100; i++)
         cout << v.at(i) << " ";
         // lanza una excepcion si no existe!
      cout << endl;
   }
   catch (...)
      cerr << "Fuera de rango!" << endl;
}
```

La clase `list`

- La clase `list` ofrece las operaciones usuales de listas, incluyendo recorridos en ambas direcciones, inserciones y borrados en puntos arbitrarios, etc.

La clase `list`

- La clase `list` ofrece las operaciones usuales de listas, incluyendo recorridos en ambas direcciones, inserciones y borrados en puntos arbitrarios, etc.
- La flexibilidad y eficiencia de esta clase reside en las clases auxiliares de *iteradores*. Un *iterador* es un "apuntador" a un elemento de la lista y se usa para consultar el elemento apuntado, eliminarlo, insertar un nuevo elemento, etc.

La clase `list`

- La clase `list` ofrece las operaciones usuales de listas, incluyendo recorridos en ambas direcciones, inserciones y borrados en puntos arbitrarios, etc.
- La flexibilidad y eficiencia de esta clase reside en las clases auxiliares de *iteradores*. Un *iterador* es un "apuntador" a un elemento de la lista y se usa para consultar el elemento apuntado, eliminarlo, insertar un nuevo elemento, etc.
- Hay iteradores inversos (`reverse_iterator` y `reverse_const_iterator`) que operan justo al revés que los normales: en términos abstractos trabajan sobre la lista invertida.

La clase `list`

- La clase `list` tiene las operaciones usuales de inserción y borrado por el final (`push_back`, `pop_back`) y por el principio (`push_front`, `pop_front`).

La clase `list`

- La clase `list` tiene las operaciones usuales de inserción y borrado por el final (`push_back`, `pop_back`) y por el principio (`push_front`, `pop_front`).
- El método `begin` devuelve un iterador al primer elemento de la lista. El método `end` devuelve un iterador a un elemento (virtual) posterior al último elemento de la lista.

La clase `list`

- La clase `list` tiene las operaciones usuales de inserción y borrado por el final (`push_back`, `pop_back`) y por el principio (`push_front`, `pop_front`).
- El método `begin` devuelve un iterador al primer elemento de la lista. El método `end` devuelve un iterador a un elemento (virtual) posterior al último elemento de la lista.
- Los iteradores pueden ser asignados y comparados (`==`, `!=`). Los operadores de incremento y decremento desplazan al iterador desde un elemento a su sucesor (antecesor). El operador de dereferencia `*` aplicado a un iterador permite acceder a su contenido. Pero un iterador **no** es un puntero, aunque en muchos aspectos se comporte de modo similar.

La clase `list`

- El método `insert` recibe como parámetros un iterador y un elemento e inserta el elemento justo delante del elemento apuntado por el iterador.

La clase `list`

- El método `insert` recibe como parámetros un iterador y un elemento e inserta el elemento justo delante del elemento apuntado por el iterador.
- El método `remove` elimina el elemento al que apunta el iterador dado.

La clase `list`

- El método `insert` recibe como parámetros un iterador y un elemento e inserta el elemento justo delante del elemento apuntado por el iterador.
- El método `remove` elimina el elemento al que apunta el iterador dado.
- El método `sort` ordena la sublista o subvector delimitado por dos iteradores dados. Si no se dan, sus valores por defecto son `begin` y `end`, es decir, `sort` ordena la lista o vector entero.

La clase `list`

Ejemplo de uso:

```
#include <list>
#include <string>
#include <iostream>

int main() {
   list<string> L;
   string s;
   while (cin >> s) { // cin >> s retorna 0 (≡ falso)
                      // si se termina el input
      L.push_back(s);
   }
   L.sort();          // ordena L
   list<string>::iterator it = L.begin();
   while (it != L.end()) {
      cout << *it << " ";
      it++;
   }
}
```

Herencia

- El mecanismo de herencia permite modelizar la relación es-un entre clases. Por ejemplo, un `coche` es un `vehículo` y puede ser conveniente reflejar dicha relación en el diseño de un programa.

Herencia

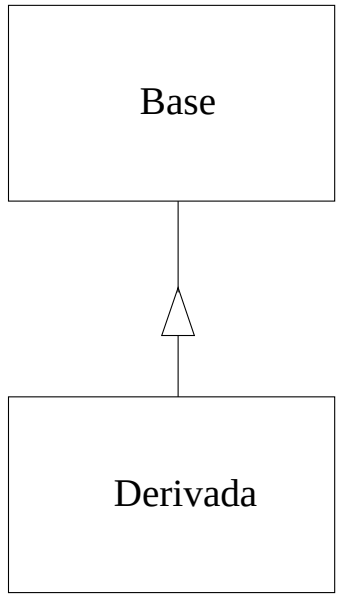
- El mecanismo de herencia permite modelizar la relación es-un entre clases. Por ejemplo, un `coche` es un `vehículo` y puede ser conveniente reflejar dicha relación en el diseño de un programa.
- La herencia es el concepto más característico de la programación orientada a objetos.

Herencia

- El mecanismo de herencia permite modelizar la relación es-un entre clases. Por ejemplo, un `coche` es un `vehículo` y puede ser conveniente reflejar dicha relación en el diseño de un programa.
- La herencia es el concepto más característico de la programación orientada a objetos.
- Si una clase `D` hereda de otra `B`, se dice que la clase `D` es derivada de `B`, que `D` es una subclase de `B` o que `B` es una superclase de `D`.

Herencia

La relación de herencia se representa gráficamente así:



Herencia

- La única forma de herencia que aqí consideraremos es lo que en C++ se denomina herencia pública.

```
class D : public B {
  ...
};
```

- La clase derivada `D` tendrá todos los atributos de `B` y hereda también todos sus métodos, excepto los constructores y los operadores de asignación.
- La clase derivada `D` puede redefinir cualquiera de los métodos (*overriding*).

Herencia

```
class B { public:
 ...
 void f() { cout << "B::f()" << endl; }
 void g() { cout << "B::g()" << endl; }
 ...
};

class D : public B { public:
 void g() { cout << "D::g()" << endl; }
};

int main() {
    D d;

    d.f(); // imprime B::f()
    d.g(); // imprime D::g()
}
```

Herencia

- Los métodos de la clase `D` no tienen acceso a los atributos y métodos privados de su clase base.
- Un método de la clase `D` puede invocar un método público que se aplique sobre la "porción" de `B` de su objeto; para ello hay que utilizar el operador de resolución de ámbito:

```
class D : public B { public:
 void g() {
      B::g();
      cout << "D::g()" << endl;
 }
};

int main() {
   D d;

   d.g(); // imprime B::g() y en
          // la siguiente línea D::g()
}
```

Constructores y destructor

- Los constructores de la clase derivada pueden invocar a constructores de la clase base. Si no se explicita, se invoca de manera implícita al constructor por defecto de B. Si no existe entonces es obligatorio invocar al constructor apropiado en la lista de inicialización.

```
class B { public:
  B(int n) { n_ = n; }
  ...
private:
  int n_;
}; class D : public B { public:
  D() : B(100), a_(0), b_(10) { ... }
  ...
private:
  int a_, b_
};
```

Constructores y destructor

- Cuando se construye un `D`, primero se construye la parte `B` heredada, luego se inicializan los atributos propios de `D` y finalmente se ejecuta el cuerpo de la constructora invocada.
- Es erróneo acceder a los atributos de `B` (incluso si no fueran privados):

```
class D : public B { public: D() : n_(100), a_(0), b_(10) {
... }
      ↑ ERROR!!
... }
```

Constructores y destructor

- La destructora de `D` actúa a la inversa: aplica el código de la destructora, después invoca a las destructoras de los atributos y finalmente a la destructora de `B`.
- Un objeto de la clase `B` puede inicializarse o se le puede asignar un `D` (porque un `D` es-un `B`); pero sólo se usa la porción de `B` que hay en el objeto de la clase `D` (*slicing*):

```
class B { ... private:
  int b_;
}; class D : public B { ... private:
  int d_;
}; ... D d; B b = d; // sólo se copia d.b_ en b.b_
```

Acceso protegido

- La palabra reservada `protected` da acceso a atributos y métodos de una clase a sus clases derivadas, y a ninguna otra.
- Un atributo o método protegido no es accesible salvo para la parte heredada de un objeto.

```
class B { private:
   int x_;
protected:
   int y_;
   int f();
... } class D : public B {
  ...
  int g(const B& b) {
      x_ = 10;   // ERROR
      y_ = 10;   // OK
      b.y_ = 10; // ERROR
      B::f();     // OK
      b.f();      // ERROR
  }
  ...
};
```

Polimorfismo dinámico

- Un puntero a la clase base `B` puede apuntar a un objeto de una clase derivada `D`. También puede inicializarse una referencia a un `B` con un `D`. En este caso no se produce *slicing*.
- Cuando se invoca un método en un objeto a través de un puntero o referencia, se invoca el método que corresponda en la clase de base (vinculación estática, static binding):

```
class B { public: ...
        int f();
        ...
} class D : public B { public: ...
         int f(); // redefine B::f()
        ...
}; ... B* pb = new B; D* pd = new D; B* p = new D; pb->f();
// B::f() pd->f();         // D::f() p->f();          // B::f()
```

Polimorfismo dinámico

- Para que el método a invocar pueda determinarse en función de la clase a la que apunta un apuntador dicho método debe definirse como virtual. En otros lenguajes de OOP la vinculación dinámica es el comportamiento por defecto.

```
class B { public: ...
        virtual int f();
        ...
} class D : public B { public: ...
         int f(); // redefine B::f()
        ...
}; ... B* p = new D; p->f();          // D::f()
```

Polimorfismo dinámico

- La destructora de una clase base debe definirse como método virtual. Incluso si no ha de hacer nada, debemos evitar que se defina la destructora de oficio, ya que no sería virtual.
- Si `~B()` no fuera virtual y hemos creado un objeto de la clase `D` al que apunta un puntero `B*`, al destruir el objeto a traves del puntero invocaríamos a `~B()` en vez de a `~D()`:

```
class B { public: ...
        ~B();
        ...
} class D : public B { public: ...
         ~D();
        ...
}; ... B* p = new D; ... delete p;      // se usa ~B() sobre el
objeto D
```

Clases abstractas

- Un método se dice virtual puro si no se hereda una implementación. Esto nos permite crear clases abstractas, es decir, clases de las cuales se hereda nada más las interfícies de los métodos.

Clases abstractas

- Un método se dice virtual puro si no se hereda una implementación. Esto nos permite crear clases abstractas, es decir, clases de las cuales se hereda nada más las interfícies de los métodos.
- En general, se recomienda crear clases abstractas de base (en Java se llaman `interfaces`), clases concretas que heredan de las clases abstractas y programas escritos en términos de las clases abstractas, es decir, de interfícies.

Clases abstractas

- Un método se dice virtual puro si no se hereda una implementación. Esto nos permite crear clases abstractas, es decir, clases de las cuales se hereda nada más las interfícies de los métodos.
- En general, se recomienda crear clases abstractas de base (en Java se llaman `interfaces`), clases concretas que heredan de las clases abstractas y programas escritos en términos de las clases abstractas, es decir, de interfícies.
- En C++, una clase es abstracta si ofrece uno o más métodos virtuales puros; también puede ofrecer métodos virtuales (no puros) y métodos no virtuales.

Clases abstractas

- Un método se dice virtual puro si no se hereda una implementación. Esto nos permite crear clases abstractas, es decir, clases de las cuales se hereda nada más las interfícies de los métodos.
- En general, se recomienda crear clases abstractas de base (en Java se llaman `interfaces`), clases concretas que heredan de las clases abstractas y programas escritos en términos de las clases abstractas, es decir, de interfícies.
- En C++, una clase es abstracta si ofrece uno o más métodos virtuales puros; también puede ofrecer métodos virtuales (no puros) y métodos no virtuales.
- No se pueden instanciar objetos de una clase abstracta.

Clases abstractas

- Si una clase `D` hereda un método virtual puro tendrá que implementarlo, o bien heredarlo tal cual y ser también abstracta.

```
class AbstractStack { public:
    virtual void push(int x) = 0; // =0 indica que es puro
    virtual void pop() = 0;
    virtual int top() = 0;
    ...
    virtual ~AbstractStack() ;
};

class ArrayStack : public AbstractStack { public:
    void push(int x);
    ...
};
```

Clases abstractas

```
int sum_and_clear(AbstractStack* S) {
    int sum = 0;
    while (!S -> empty()) {
        sum += S -> top();
        S -> pop();
    }
    return sum;
}

... AbstractStack* s = new ArrayStack; ... int result =
sum_and_clear(s);
```

Compilación separada y montaje

- El compilador GNU `gcc-x.xx.x` (instalado en el LCFIB) permite compilar programas en C y en C++. Para compilar programas en C++ los ficheros fuente deben tener extensión `.cpp` y utilizarse el comando `g++`.

Compilación separada y montaje

- El compilador GNU `gcc-`*x.xx.x* (instalado en el LCFIB) permite compilar programas en C y en C++. Para compilar programas en C++ los ficheros fuente deben tener extensión `.cpp` y utilizarse el comando `g++`.
- Se puede consultar un resumen de la documentación con el comando `man gcc`

Compilación separada y montaje

- El compilador GNU `gcc-`*x.xx.x* (instalado en el LCFIB) permite compilar programas en C y en C++. Para compilar programas en C++ los ficheros fuente deben tener extensión `.cpp` y utilizarse el comando g++.
- Se puede consultar un resumen de la documentación con el comando `man gcc`
- El comportamiento del comando g++ se puede controlar mediante diversos *flags*, como es habitual en Unix. Para averiguar los principales *flags* admitidos por g++ se usa el *flag* `--help`

Compilación separada y montaje

- Para compilar por separado un fichero fuente se usa el comando (las letras en cursiva indican un argumento)

```
% g++ -c nom_fich.cpp
```

el cual produce un fichero objeto *nom_fich*.o.

- Para montar ("linkar") varios ficheros objeto simplemente se ponen sus nombres tras el comando g++, seguidos unos de otros y no importando el orden:

```
% g++ nom_fich1.o nom_fich2.o  ...
```

- El fichero ejecutable por defecto se llama `a.out`. Si se quiere que el ejecutable tenga un nombre diferente entonces se debe usar el *flag* -o:

```
% g++ -o nom_ejecutable f1.o f2.o ...
```

Compilación separada y montaje

- También se puede compilar y linkar varios ficheros en una única línea de comando:

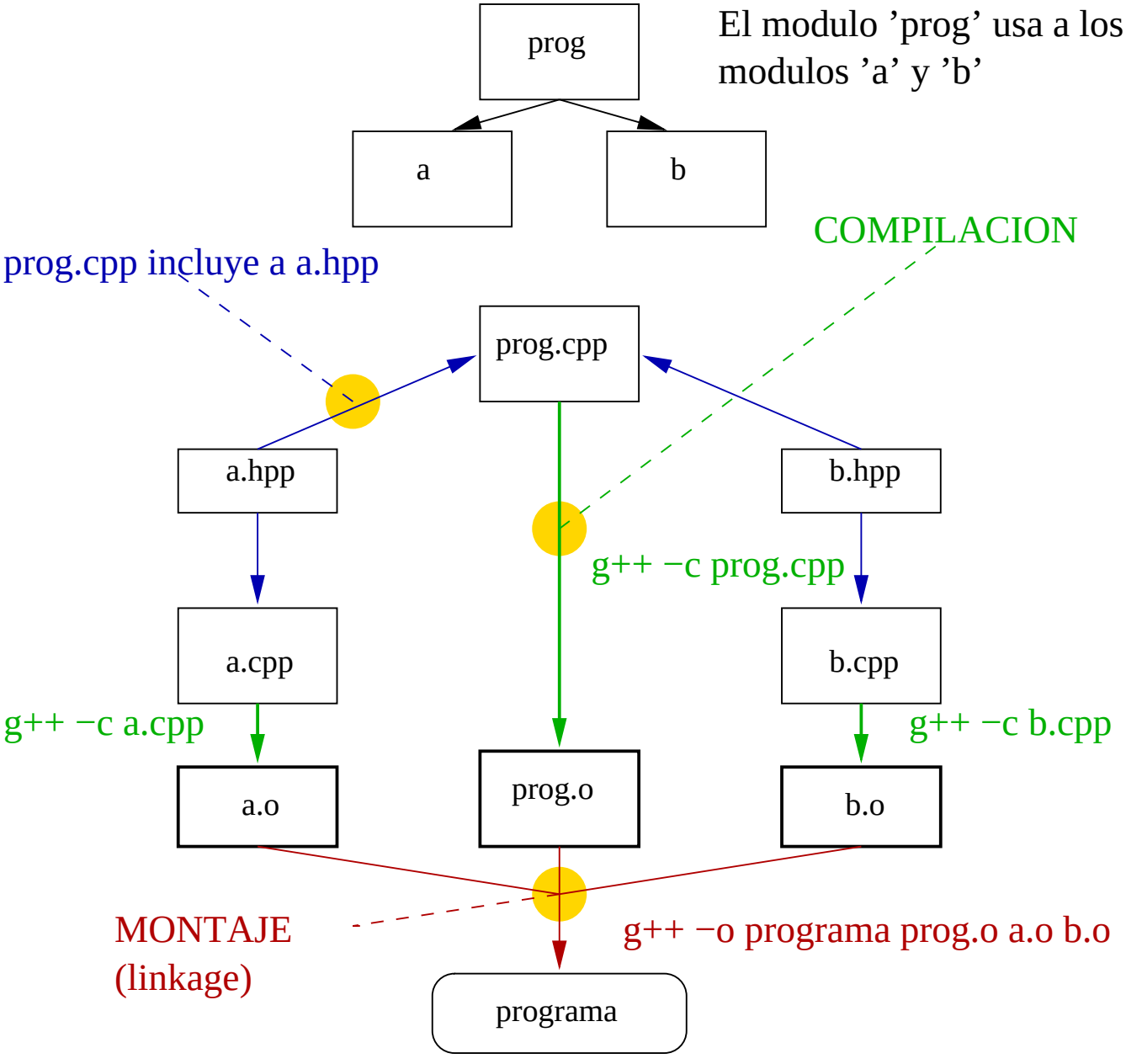
```
% g++ -o nom_ejecutable f1.cpp f2.o ...
```

Por ejemplo,

```
% g++ -o prog cola.o pr.cpp pila.o
```

genera un fichero ejecutable llamado `prog`, para lo cual se compila `pr.cpp` y el fichero objeto resultante se linka con los ficheros objeto `pila.o` y `cola.o`. No se conserva el fichero intermedio `pr.o`.

Compilación separada y montaje



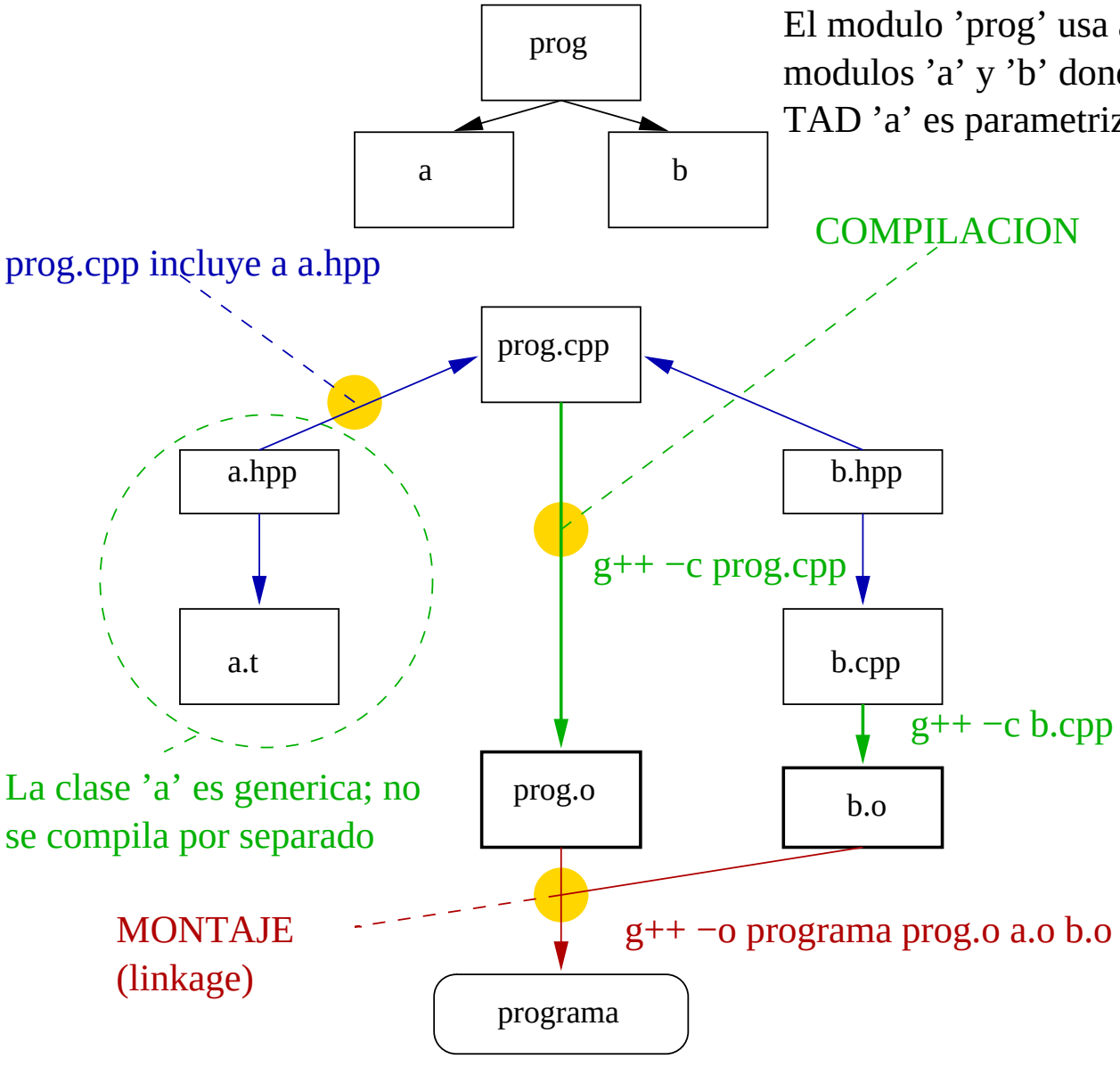
Compilación separada y montaje

- Los módulos que definen clases o funciones genéricas no se compilan por separado. Una forma conveniente de organizar el código consiste en escribir el fichero de cabecera `.hpp` con la declaración de la clase y por otro lado, las definiciones en un fichero aparte al que, por convenio, le daremos la extensión `.t`. El fichero `.hpp` incluye al fichero `.t`. Si un programa usa a la clase genérica X entonces deberá incluir el fichero `X.hpp` (e indirectamente a `X.t`).

Compilación separada y montaje

- Los módulos que definen clases o funciones genéricas no se compilan por separado. Una forma conveniente de organizar el código consiste en escribir el fichero de cabecera `.hpp` con la declaración de la clase y por otro lado, las definiciones en un fichero aparte al que, por convenio, le daremos la extensión `.t`. El fichero `.hpp` incluye al fichero `.t`. Si un programa usa a la clase genérica X entonces deberá incluir el fichero `X.hpp` (e indirectamente a `X.t`).
- Se puede obtener una comprobación sintáctica de la clase genérica mediante la inclusión del `.hpp` (y por tanto del fichero `.t`) en un fichero `.cpp`. Este fichero solamente debe tener la línea de inclusión.

Compilación separada y montaje

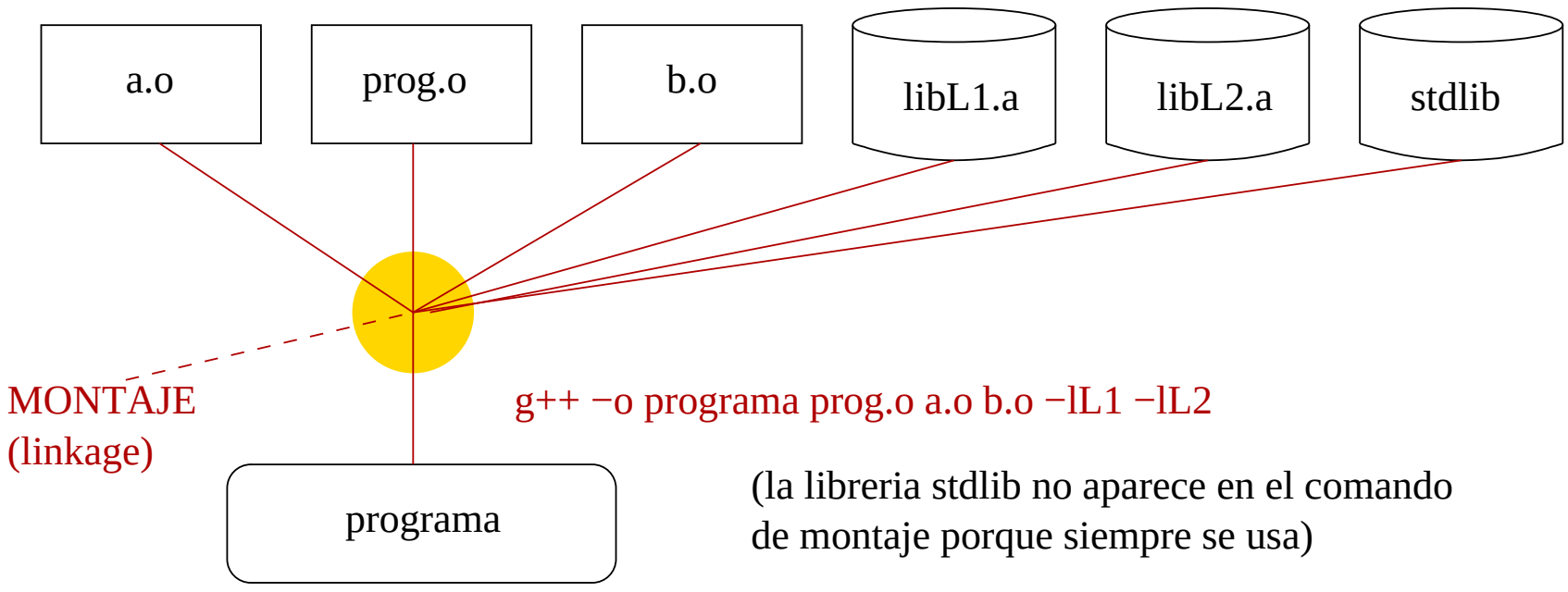


Compilación separada y montaje

- Muchos programas sólo emplean clases, métodos y funciones definidos en la librería estándar (p.e. `string`, `list`, `iostream`, ...).
  El montador (*linker*) emplea siempre (por defecto) dicha librería.
- Si queremos usar otra librería se habrá de indicar explícitamente en el comando de montaje, mediante el *flag* `-l`. En Unix el convenio es llamar a las librerías `lib`*xxx* con extensión `.a` o `.so`. Después del flag `-l` se pone la parte variable del nombre, es decir *xxx*. Por ejemplo, para usar las librerias `libL1` y `libL2` escribiremos el comando

```
% g++ -o nom_ejecutable f1.o f2.o ... -lL1 -lL2
```

Compilación separada y montaje



(la libreria stdlib no aparece en el comando de montaje porque siempre se usa)

Compilación separada y montaje

- Si necesitamos utilizar ficheros de cabecera (*headers*) que no se encuentran en el mismo directorio donde estamos compilando o en un sudirectorio estándar de inclusión (p.e. `/usr/include`) debemos usar uno o más *flags* `-I`, tantos como subdirectorios queramos añadir a la lista de subdirectorios de inclusión. Por ejemplo:

```
% g++ -c -I /usr/users/fred/my_headers f1.cpp
```

- Para especificar un camino en el flag `-I` puede usarse el camino absoluto o el camino relativo. Si el comando del ejemplo anterior se estuviera haciendo en el subdirectorio `/usr/users/wilma/pract` podríamos haber escrito:

```
% g++ -c -I ../../fred/my_headers f1.cpp
```

Compilación separada y montaje

- Un problema similar al anterior se da si necesitamos usar una librería que no está en el directorio en curso o en un subdirectorio estándar. Se usa el *flag* `-L` para indicar el camino. Supongamos que queremos usar `libL1.a` que se encuentra en `/usr/users/ps`. Entonces escribiremos:

```
% g++ -L /usr/users/ps -o pr pr.cpp f1.cpp f2.o -lL1
```

- Para conseguir que el compilador verifique instrucciones dudosas y alerte del máximo número posible fuentes de error debe utilizarse el *flag* `-Wall`:

```
% g++ -c -Wall nom_fich.cpp
```

Compilación separada y montaje

- Si queremos emplear un *debugger* se habrá de poner el *flag* -g al compilar o crear el ejecutable a partir de los ficheros fuente:

```
% g++ -g -c -Wall fich.cpp% g++ -g -o
nom_ejecutable -Wall f1.cpp f2.cpp ...
```

Librerías externas

- Hasta ahora se ha visto que la compilación produce ficheros objeto *nom_fich*.o que se utilizan para montar el ejecutable final. También se ha comentado que el compilador utiliza librerías externas para montar ese ejecutable.

Librerías externas

- Hasta ahora se ha visto que la compilación produce ficheros objeto *nom_fich*.o que se utilizan para montar el ejecutable final. También se ha comentado que el compilador utiliza librerías externas para montar ese ejecutable.
- Estas librerías son generalmente del sistema, pero también se pueden crear unas propias y utilizarlas para futuros programas (o bien distribuirlas para su uso).

Librerías externas

- Existen dos tipos de librerías externas: estáticas y dinámicas. Las estáticas se adjuntan al ejecutable en el momento del enlazado contribuyendo con su código. Es decir, el fichero ejecutable contiene todo el código necesario y eventualmente podríamos prescindir de los ficheros de las librerías sin problema

Librerías externas

- Existen dos tipos de librerías externas: estáticas y dinámicas. Las estáticas se adjuntan al ejecutable en el momento del enlazado contribuyendo con su código. Es decir, el fichero ejecutable contiene todo el código necesario y eventualmente podríamos prescindir de los ficheros de las librerías sin problema
- Por el contrario, el ejecutable se enlaza a las librerías dinámicas mediante una referencia, de modo que parte la correcta ejecución del programa es dependiente de la existencia de los ficheros de la librería. Estos ficheros han de estar ubicados en unos directorios específicos, listados en la variable de entorno `LD_LIBRARY_PATH`

Librerías externas estáticas

- Para formar una librería estática, previamente hemos de compilarla y posteriormente formarla mediante el comando `ar` que es un archivador (al estilo de `tar` o `jar`):
  1. Compilación:

```
% g++ -c -Wall fich1.cpp fich2.cpp
```

  2. Creación de la librería:

```
% ar -r nom_lib fich1.o fich2.o
```

  3. Esto produce un fichero `.a` que contiene los ficheros objeto de la librería.

Librerías externas dinámicas

- El caso de las librerías dinámicas o compartidas, la compilación es algo más compleja. Es necesario solicitar código independiente de la posición (position-independent code (PIC)) mediante el modificador -fpic
  1. Compilación:

```
% g++ -c -Wall -fpic fich1.cpp fich2.cpp
```

  2. Creación de la librería en Linux:

```
% g++ -o nom_lib -G -fpic fich1.o fich2.o
```

  3. Creación de la librería en Solaris:

```
% g++ -o nom_lib -fpic -shared -Ur fich1.o fich2.o
```

  4. Esto produce un fichero *nom_lib*.so que contiene los ficheros objeto de la librería. La extensión de estos ficheros indica una librería dinámica: **s**hared **o**bject.

La ubicación de las librerías dinámicas

- Los ejecutables enlazados con librerías dinámicas dependen de la existencia de los ficheros `.so` correspondientes.
- Para saber qué librerías dinámicas utiliza un determinado ejecutable y si éstas librerías se encuentran disponibles existe el comando `ldd`:

```
% ldd ejecutable
```

- El resultado nos muestra la lista de librerías que se requieren y si se encuentran o no disponibles. En caso de que alguna no se encuentre, el programa puede no funcionar correctamente.

La ubicación de las librerías dinámicas

- Los directorios donde están ubicados los ficheros de librerías dinámicas o compartidas han de estar incluidos en la variable de entorno `LD_LIBRARY_PATH`.
- Para añadir un directorio a la lista de directorios de esta variable:

```
% setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:nuevo_dir
```

- Es importante recalcar que los ejecutables y las librerías han de ser compatibles en cuanto a arquitectura (Intel, Sparc, etc) como a sistema operativo (Unix, GNU/Linux, etc) como a versión del compilador utilizado.

Compilación separada y montaje

- Tener que escribir un comando de compilación y montaje en el que intervienen varios módulos, librerías, *flags*, . . . acaba convirtiéndose en una tarea pesada y propensa al error. Una opción consiste en definir un alias en el fichero `~/.tcshrc` (el nombre dependerá del *shell* que se esté usando). Por ejemplo, si ponemos

```
alias compila 'g++ -c -I dir1 \!*'
```

  a partir de ese momento (mejor dicho, al hacer *login* la siguiente vez) podremos escribir:

```
% compila pr.cpp
```

- Otra opción es definir un *makefile*.

Make

- `make` es un programa de Unix que simplifica notablemente el trabajo de compilación y montaje. Además, instruyendo adecuadamente a `make`, éste se limitará a recompilar los módulos que realmente haga falta.
- El programa `make` utiliza un fichero llamado `Makefile` que se debe encontrar en el mismo directorio donde se ejecuta el `make`. Tiene un argumento, el *target*, que es lo que queremos que se construya. Si se escribe el comando `make` a secas, el *target* es el primero que aparece en el `Makefile`.
- Ilustramos su funcionamiento mediante un ejemplo concreto, correspondiente (más o menos) al diagrama de módulos de la figura 1.

Ejemplo de Makefile

- El fichero `Makefile` que debe residir en el mismo directorio que los ficheros `.cpp`.
- Supondremos que los ficheros `.hpp` están en el subdirectorio `/usr/users/yo/mis_include`.
- Para construir `prog` supondremos que además se necesita la librería `/usr/users/yo/mis_libs/libL1.a`.

Ejemplo de Makefile

Makefile

```
CC = g++
INCL = /usr/users/yo/mis_include
COMPILE = $(CC) -c -Wall -I $(INCL)
LIBS = /usr/users/yo/mis_libs
LINK = $(CC) -L $(LIBS)
OBJS = prog.o a.o b.o
prog: $(OBJS) $(LIBS)/libL1.a
    $(LINK) -o prog $(OBJS) -lL1
prog.o: prog.cpp $(INCL)/a.hpp $(INCL)/b.hpp
    $(COMPILE) prog.cpp
a.o: a.cpp $(INCL)/a.hpp
    $(COMPILE) a.cpp
b.o: b.cpp $(INCL)/b.hpp
    $(COMPILE) b.cpp
```

Sintaxis de Makefile

- Las primeras seis líneas definen variables. Se pueden tener tantas como se quieran. Si `X` es una variable, se puede acceder a su valor mediante `$(X)`. Luego vienen una serie de bloques de la forma:

```
target: dependencias
        comando1
        comando2
        ...
```

- Cada línea en la que escribimos un comando debe necesariamente empezar con un tabulador.

Sintaxis de Makefile

- Las dependencias son listas que dicen que elementos intervienen directamente en la construcción del *target* (y están sujetas a cambio).
- Por ejemplo, `prog` depende de los ficheros objeto `prog.o`, `a.o`, `b.o` y también de `libL1.a`. Eso mismo indican la línea de dependencias. En el comando de abajo se indica como obtener `prog` a partir de las dependencias.
- Lo mismo con `prog.o` y los demás. Por ejemplo, `prog.o` **no** depende de `a.cpp` ni `b.cpp`, pero sí de sus especificaciones (`a.hpp` y `b.hpp`).

Sintaxis de Makefile

- Se pueden hacer **muchas** cosas mediante `make`. No sólo facilitar el proceso de compilación y montaje. Aquí sólo hemos arañado la superficie y dado una visión simplificada. Por ejemplo, si añadimos al final del `Makefile` las líneas

```
clean:
	rm -f *.o ; rm prog
```

  bastará escribir `make clean` para borrar todos los ficheros objetos y el ejecutable.

Ejecución

- Para ejecutar un programa basta con escribir el nombre del ejecutable correspondiente como respuesta al *prompt* de Unix.

```
% nom_ejecutable
```

- Si el programa lee y escribe por los canales estándar de entrada (`cin`) y salida (`cout`) se puede redirigir la entrada para que provenga de un fichero y no del teclado, y la salida para que escriba en un fichero en lugar de en la pantalla.

Ejecución

- Para ello se utilizan los operadores de rediccionamiento (< para la entrada, > para salida) seguidos del nombre del fichero correspondiente. Se puede redireccionar sólo la entrada, sólo la salida, o ambas:

```
% nom_ejecutable < fichero_entrada > fichero_salida
```

- Se puede conectar la salida estándar de un programa con la entrada estándar de otro programa mediante una *pipe*. El operador correspondiente es la barra vertical |.

```
% nom_ejecutable1 | nom_ejecutable2
```

Ejecución

- Al combinar dos programas mediante una *pipe* el resultado se comporta como un único programa al cual se le puede redirigir la entrada, la salida, conectarlo con otra *pipe*, etc. Por ejemplo, `more` muestra su entrada por pantalla página a página esperando una orden del usuario para saltar de una página a la siguiente. Si tenemos un programa `prog` y queremos ver su salida poco a poco podemos escribir:

```
% prog | more < entrada.in
```

Ejecución

- Muchos programas imprimen sus mensajes de error por el canal estándar de error (`cerr`) que normalmente está asociado a la pantalla. Si utilizamos el operador > los mensajes siguen apareciendo por pantalla. Para redirigir el `cerr` se usa el operador >& seguido de un nombre de fichero. Por ejemplo,

```
% g++ -c pr.cpp >& errores.out
```

  genera el fichero `errores.out` con un listado de todos los errores de compilación.

Testing o Fase de Comprobación

- Se denomina *testing* al proceso de ejecución de un programa para intentar encontrar errores.

Testing o Fase de Comprobación

- Se denomina *testing* al proceso de ejecución de un programa para intentar encontrar errores.
- El testing *no* demuestra la correción de un programa, pero puede demostar su incorrección, es decir, la presencia de uno o más errores.

Testing o Fase de Comprobación

- Se denomina *testing* al proceso de ejecución de un programa para intentar encontrar errores.
- El testing *no* demuestra la correción de un programa, pero puede demostar su incorrección, es decir, la presencia de uno o más errores.
- Un *juego de pruebas* está formado por un conjunto de datos de entrada (pueden ser válidos o no) y por los resultados esperados.

Testing o Fase de Comprobación

- Se denomina *testing* al proceso de ejecución de un programa para intentar encontrar errores.
- El testing *no* demuestra la correción de un programa, pero puede demostar su incorrección, es decir, la presencia de uno o más errores.
- Un *juego de pruebas* está formado por un conjunto de datos de entrada (pueden ser válidos o no) y por los resultados esperados.
- Un buen juego de pruebas es aquél que tiene una probabilidad alta de encontrar un error.

Testing o Fase de Comprobación

- Se denomina *testing* al proceso de ejecución de un programa para intentar encontrar errores.
- El testing *no* demuestra la correción de un programa, pero puede demostar su incorrección, es decir, la presencia de uno o más errores.
- Un *juego de pruebas* está formado por un conjunto de datos de entrada (pueden ser válidos o no) y por los resultados esperados.
- Un buen juego de pruebas es aquél que tiene una probabilidad alta de encontrar un error.
- El testing resultará más objetivo si lo llevan a cabo personas que no participaron en la implementación del programa.

Testing o Fase de Comprobación

- Se denomina *testing* al proceso de ejecución de un programa para intentar encontrar errores.
- El testing *no* demuestra la correción de un programa, pero puede demostar su incorrección, es decir, la presencia de uno o más errores.
- Un *juego de pruebas* está formado por un conjunto de datos de entrada (pueden ser válidos o no) y por los resultados esperados.
- Un buen juego de pruebas es aquél que tiene una probabilidad alta de encontrar un error.
- El testing resultará más objetivo si lo llevan a cabo personas que no participaron en la implementación del programa.
- Existen diversas estrategias de testing de módulos, dependiendo de la forma en que se combinen los módulos para su comprobación e implementación.

Estrategias

1. *Big bang*: consiste en una comprobación global de todo el programa al mismo tiempo.

Estrategias

1. *Big bang*: consiste en una comprobación global de todo el programa al mismo tiempo.
   - Inconvenientes

Estrategias

1. *Big bang*: consiste en una comprobación global de todo el programa al mismo tiempo.
    - Inconvenientes
        - Difícil localizar los errores dentro del programa. Difícil determinar el tipo de error: interno a un módulo, de interfície o de interacción entre los módulos.

Estrategias

1. *Big bang*: consiste en una comprobación global de todo el programa al mismo tiempo.
    - Inconvenientes
        - Difícil localizar los errores dentro del programa. Difícil determinar el tipo de error: interno a un módulo, de interfície o de interacción entre los módulos.
        - Los errores pueden estar en la interfície de los módulos, es decir, las llamadas a las funciones de un módulo y su definición no se corresponden o bien no están definidas.

Estrategias

1. *Big bang*: consiste en una comprobación global de todo el programa al mismo tiempo.
    - Inconvenientes
        - Difícil localizar los errores dentro del programa. Difícil determinar el tipo de error: interno a un módulo, de interfície o de interacción entre los módulos.
        - Los errores pueden estar en la interfície de los módulos, es decir, las llamadas a las funciones de un módulo y su definición no se corresponden o bien no están definidas.
        - No se puede testear hasta tener la implementación de todos los módulos.

Estrategias

2. Comprobación separada: consiste en una comprobación separada de cada módulo seguida de una comprobación global de todo el programa. Se usan un *driver* y los *stubs* necesarios para probar cada módulo por separado (ver la definición de *driver* y *stub* más adelante).

Estrategias

2. Comprobación separada: consiste en una comprobación separada de cada módulo seguida de una comprobación global de todo el programa. Se usan un *driver* y los *stubs* necesarios para probar cada módulo por separado (ver la definición de *driver* y *stub* más adelante).
   - Ventajas

Estrategias

2. Comprobación separada: consiste en una comprobación separada de cada módulo seguida de una comprobación global de todo el programa. Se usan un *driver* y los *stubs* necesarios para probar cada módulo por separado (ver la definición de *driver* y *stub* más adelante).
   - Ventajas
     - Facilidad para detectar errores que afecten a un sólo módulo.

Estrategias

2. Comprobación separada: consiste en una comprobación separada de cada módulo seguida de una comprobación global de todo el programa. Se usan un *driver* y los *stubs* necesarios para probar cada módulo por separado (ver la definición de *driver* y *stub* más adelante).
   - Ventajas
     - Facilidad para detectar errores que afecten a un sólo módulo.
     - Se puede probar un módulo aunque no esté implementado el resto.

Estrategias

2. Comprobación separada: consiste en una comprobación separada de cada módulo seguida de una comprobación global de todo el programa. Se usan un *driver* y los *stubs* necesarios para probar cada módulo por separado (ver la definición de *driver* y *stub* más adelante).
   - Ventajas
     - Facilidad para detectar errores que afecten a un sólo módulo.
     - Se puede probar un módulo aunque no esté implementado el resto.
   - Inconvenientes

Estrategias

2. Comprobación separada: consiste en una comprobación separada de cada módulo seguida de una comprobación global de todo el programa. Se usan un *driver* y los *stubs* necesarios para probar cada módulo por separado (ver la definición de *driver* y *stub* más adelante).
   - Ventajas
     - Facilidad para detectar errores que afecten a un sólo módulo.
     - Se puede probar un módulo aunque no esté implementado el resto.
   - Inconvenientes
     - Los errores de interfície y los errores que son de interacción entre los módulos no se pueden detectar hasta el final.

Estrategias

3. Comprobación incremental: consiste en ir añadiendo y comprobando los módulos uno a uno.

Estrategias

3. Comprobación incremental: consiste en ir añadiendo y comprobando los módulos uno a uno.
   - Ventajas

Estrategias

3. Comprobación incremental: consiste en ir añadiendo y comprobando los módulos uno a uno.
   - Ventajas
     - Facilidad para detectar errores que afecten a un sólo módulo.

Estrategias

3. Comprobación incremental: consiste en ir añadiendo y comprobando los módulos uno a uno.
    - Ventajas
        - Facilidad para detectar errores que afecten a un sólo módulo.
        - Para cada módulo que se añade se testean sus interfícies y sus interacciones con los módulos presentes hasta el momento.

Estrategias

3. Comprobación incremental: consiste en ir añadiendo y comprobando los módulos uno a uno.
    - Ventajas
        - Facilidad para detectar errores que afecten a un sólo módulo.
        - Para cada módulo que se añade se testean sus interfícies y sus interacciones con los módulos presentes hasta el momento.
        - Los juegos de pruebas son más efectivos dado que es más probable que se detecten los errores si se utiliza esta estrategia en lugar de las anteriores.

Estrategias

3. Comprobación incremental: consiste en ir añadiendo y comprobando los módulos uno a uno.
    - Ventajas
        - Facilidad para detectar errores que afecten a un sólo módulo.
        - Para cada módulo que se añade se testean sus interfícies y sus interacciones con los módulos presentes hasta el momento.
        - Los juegos de pruebas son más efectivos dado que es más probable que se detecten los errores si se utiliza esta estrategia en lugar de las anteriores.
        - Los módulos ya presentes se siguen comprobando.

Estrategias

3. Comprobación incremental: consiste en ir añadiendo y comprobando los módulos uno a uno.
   - Ventajas
     - Facilidad para detectar errores que afecten a un sólo módulo.
     - Para cada módulo que se añade se testean sus interfícies y sus interacciones con los módulos presentes hasta el momento.
     - Los juegos de pruebas son más efectivos dado que es más probable que se detecten los errores si se utiliza esta estrategia en lugar de las anteriores.
     - Los módulos ya presentes se siguen comprobando.
   - Existen dos formas de realizar la comprobación incremental:

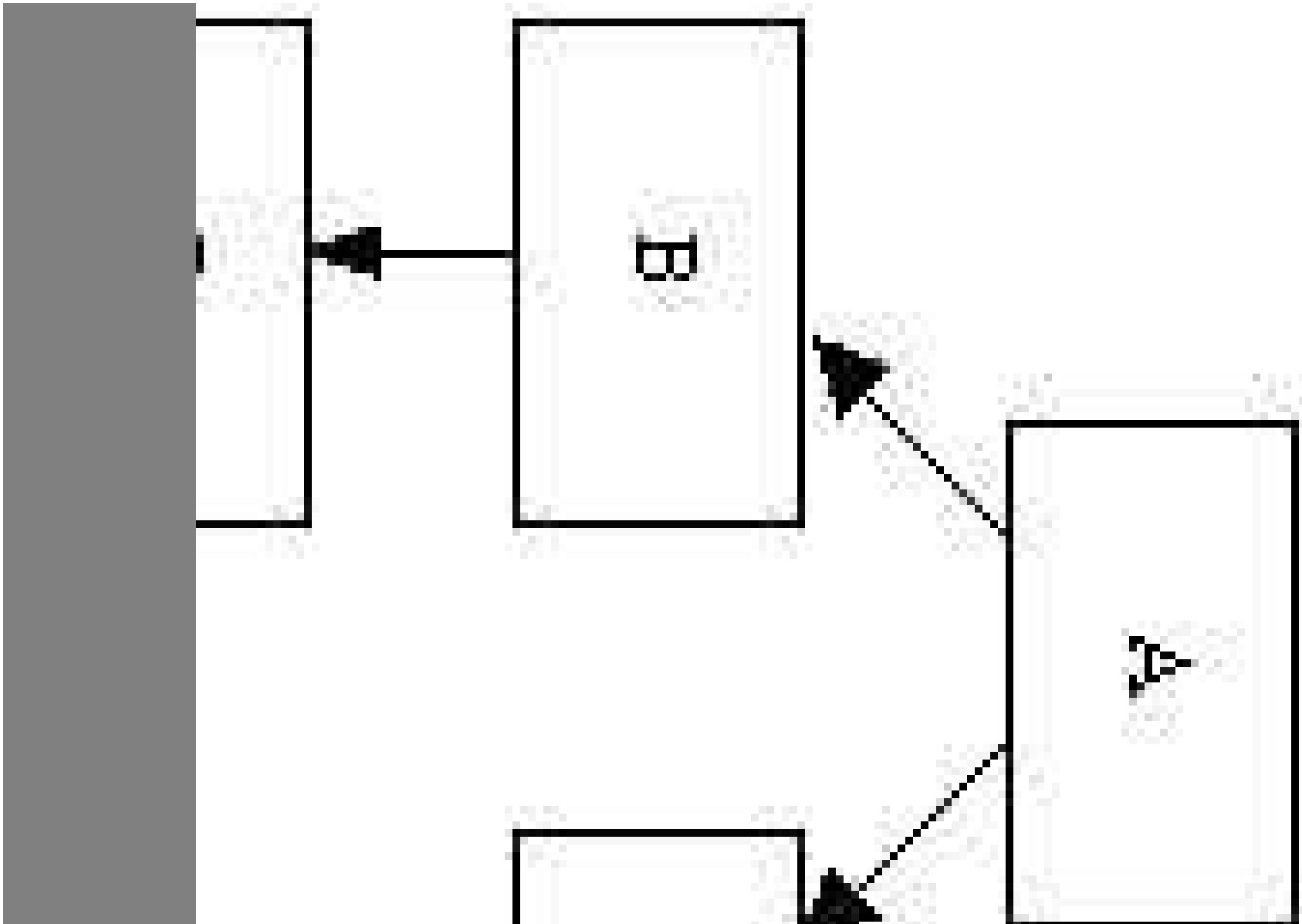
Estrategias

3. Comprobación incremental: consiste en ir añadiendo y comprobando los módulos uno a uno.
   - Ventajas
     - Facilidad para detectar errores que afecten a un sólo módulo.
     - Para cada módulo que se añade se testean sus interfícies y sus interacciones con los módulos presentes hasta el momento.
     - Los juegos de pruebas son más efectivos dado que es más probable que se detecten los errores si se utiliza esta estrategia en lugar de las anteriores.
     - Los módulos ya presentes se siguen comprobando.
   - Existen dos formas de realizar la comprobación incremental:
     - Descendente o *Top Down*

Estrategias

3. Comprobación incremental: consiste en ir añadiendo y comprobando los módulos uno a uno.
   - Ventajas
     - Facilidad para detectar errores que afecten a un sólo módulo.
     - Para cada módulo que se añade se testean sus interfícies y sus interacciones con los módulos presentes hasta el momento.
     - Los juegos de pruebas son más efectivos dado que es más probable que se detecten los errores si se utiliza esta estrategia en lugar de las anteriores.
     - Los módulos ya presentes se siguen comprobando.
   - Existen dos formas de realizar la comprobación incremental:
     - Descendente o *Top Down*
     - Ascendente o *Bottom Up*

Estrategias



Estrategia Top-Down

La estrategia de *comprobación incremental descendente* o *top-down* consiste en:

1. Comprobar los módulos superiores de la descomposición modular del programa mediante *stubs* (módulos auxiliares que simulan la funcionalidad de los módulos reales, con la misma interfície).

Estrategia Top-Down

La estrategia de *comprobación incremental descendente* o *top-down* consiste en:

1. Comprobar los módulos superiores de la descomposición modular del programa mediante *stubs* (módulos auxiliares que simulan la funcionalidad de los módulos reales, con la misma interfície).
2. Reemplazar un *stub* por el módulo verdadero y repetir este proceso con el resto de módulos hasta completar toda la descomposición modular.

Estrategia Top-Down

- Por ejemplo dado el diseño de la figura anterior, la secuencia de comprobaciones sería la siguiente:

Estrategia Top-Down

- Por ejemplo dado el diseño de la figura anterior, la secuencia de comprobaciones sería la siguiente:
  1. A partir del módulo real A y de los *stubs* para los módulos B y C, crear un juego de pruebas para A y comprobar el módulo A.

Estrategia Top-Down

- Por ejemplo dado el diseño de la figura anterior, la secuencia de comprobaciones sería la siguiente:
  1. A partir del módulo real A y de los *stubs* para los módulos B y C, crear un juego de pruebas para A y comprobar el módulo A.
  2. Sustituir el *stub* para el módulo C por el real y crear un juego de pruebas para C y comprobarlo. Pasar de nuevo los juegos de pruebas para A (con el módulo C real y el *stub* de B).

Estrategia Top-Down

- Por ejemplo dado el diseño de la figura anterior, la secuencia de comprobaciones sería la siguiente:
  1. A partir del módulo real A y de los *stubs* para los módulos B y C, crear un juego de pruebas para A y comprobar el módulo A.
  2. Sustituir el *stub* para el módulo C por el real y crear un juego de pruebas para C y comprobarlo. Pasar de nuevo los juegos de pruebas para A (con el módulo C real y el *stub* de B).
  3. Sustituir el *stub* para el módulo B por el real, crear un *stub* para el módulo D y crear un juego de pruebas para B y comprobarlo. Seguir pasando los juegos de pruebas de los módulos A y C.

Estrategia Top-Down

- Por ejemplo dado el diseño de la figura anterior, la secuencia de comprobaciones sería la siguiente:
  1. A partir del módulo real A y de los *stubs* para los módulos B y C, crear un juego de pruebas para A y comprobar el módulo A.
  2. Sustituir el *stub* para el módulo C por el real y crear un juego de pruebas para C y comprobarlo. Pasar de nuevo los juegos de pruebas para A (con el módulo C real y el *stub* de B).
  3. Sustituir el *stub* para el módulo B por el real, crear un *stub* para el módulo D y crear un juego de pruebas para B y comprobarlo. Seguir pasando los juegos de pruebas de los módulos A y C.
  4. Sustituir el *stub* para el módulo D por el real y crear un juego de pruebas para D y comprobarlo. Seguir pasando los juegos de pruebas de los módulos A, C y B.

Estrategia Top-Down

- Inconvenientes

Estrategia Top-Down

- Inconvenientes
  - Dependencia entre los juegos de pruebas y los resultados dados por los *stubs*.

Estrategia Top-Down

- Inconvenientes
  - Dependencia entre los juegos de pruebas y los resultados dados por los *stubs*.
  - Los *stubs* pueden introducir errores.

Estrategia Top-Down

- Inconvenientes
  - Dependencia entre los juegos de pruebas y los resultados dados por los *stubs*.
  - Los *stubs* pueden introducir errores.
- Ventajas

Estrategia Top-Down

- Inconvenientes
  - Dependencia entre los juegos de pruebas y los resultados dados por los *stubs*.
  - Los *stubs* pueden introducir errores.
- Ventajas
  - Permite encontrar errores en las primeras abstracciones del proceso de diseño.

Estrategia Top-Down

- Inconvenientes
  - Dependencia entre los juegos de pruebas y los resultados dados por los *stubs*.
  - Los *stubs* pueden introducir errores.
- Ventajas
  - Permite encontrar errores en las primeras abstracciones del proceso de diseño.
  - Incrementa la portabilidad, al obtener versiones parciales útiles del programa.

Estrategia Bottom-Up

La estrategia de *comprobación incremental ascendente* o *bottom-up* consiste en:

1. Comprobar los módulos inferiores de la descomposición modular del programa mediante *drivers* (módulos auxiliares que permiten llamar a los módulos reales, con la interfície correcta).

Estrategia Bottom-Up

La estrategia de *comprobación incremental ascendente* o *bottom-up* consiste en:

1. Comprobar los módulos inferiores de la descomposición modular del programa mediante *drivers* (módulos auxiliares que permiten llamar a los módulos reales, con la interfície correcta).
2. Cuando todos los módulos llamados por un módulo dado ya han sido comprobados entonces se comprueba dicho módulo y así sucesivamente hasta llegar a los módulos superiores.

Estrategia Bottom-Up

- Dado el diseño de la figura anterior, la secuencia de comprobaciones en este caso sería la siguiente:

Estrategia Bottom-Up

- Dado el diseño de la figura anterior, la secuencia de comprobaciones en este caso sería la siguiente:
  1. Comprobación del módulo D con el *driver* B.

Estrategia Bottom-Up

- Dado el diseño de la figura anterior, la secuencia de comprobaciones en este caso sería la siguiente:
  1. Comprobación del módulo D con el *driver* B.
  2. Sustituir el *driver* B por el real y comprobarlo añadiendo el *driver* A.

Estrategia Bottom-Up

- Dado el diseño de la figura anterior, la secuencia de comprobaciones en este caso sería la siguiente:
  1. Comprobación del módulo D con el *driver* B.
  2. Sustituir el *driver* B por el real y comprobarlo añadiendo el *driver* A.
  3. Comprobación del módulo C con el *driver* A.

Estrategia Bottom-Up

- Dado el diseño de la figura anterior, la secuencia de comprobaciones en este caso sería la siguiente:
  1. Comprobación del módulo D con el *driver* B.
  2. Sustituir el *driver* B por el real y comprobarlo añadiendo el *driver* A.
  3. Comprobación del módulo C con el *driver* A.
  4. Comprobación del módulo A con el resto de módulos.

Estrategia Bottom-Up

- Dado el diseño de la figura anterior, la secuencia de comprobaciones en este caso sería la siguiente:
  1. Comprobación del módulo D con el *driver* B.
  2. Sustituir el *driver* B por el real y comprobarlo añadiendo el *driver* A.
  3. Comprobación del módulo C con el *driver* A.
  4. Comprobación del módulo A con el resto de módulos.
- Ventajas

Estrategia Bottom-Up

- Dado el diseño de la figura anterior, la secuencia de comprobaciones en este caso sería la siguiente:
  1. Comprobación del módulo D con el *driver* B.
  2. Sustituir el *driver* B por el real y comprobarlo añadiendo el *driver* A.
  3. Comprobación del módulo C con el *driver* A.
  4. Comprobación del módulo A con el resto de módulos.
- Ventajas
  - Permite una comprobación más rápida de un subsistema.

Estrategia Bottom-Up

- Dado el diseño de la figura anterior, la secuencia de comprobaciones en este caso sería la siguiente:
  1. Comprobación del módulo D con el *driver* B.
  2. Sustituir el *driver* B por el real y comprobarlo añadiendo el *driver* A.
  3. Comprobación del módulo C con el *driver* A.
  4. Comprobación del módulo A con el resto de módulos.
- Ventajas
  - Permite una comprobación más rápida de un subsistema.
  - Elimina la necesidad de *stubs* que son a menudo tan difíciles de implementar como el módulo real.

Juegos de Pruebas

- En la construcción de un buen juego de pruebas el objetivo primordial es escoger aquellas pruebas con una mayor probabilidad de detectar errores.

Juegos de Pruebas

- En la construcción de un buen juego de pruebas el objetivo primordial es escoger aquellas pruebas con una mayor probabilidad de detectar errores.
- Conseguir un juego de pruebas exhaustivo, que cubra todos los casos es en la práctica imposible de conseguir, dado el número de casos posibles y el tiempo que necesitaríamos para construirlo.

Juegos de Pruebas

- En la construcción de un buen juego de pruebas el objetivo primordial es escoger aquellas pruebas con una mayor probabilidad de detectar errores.
- Conseguir un juego de pruebas exhaustivo, que cubra todos los casos es en la práctica imposible de conseguir, dado el número de casos posibles y el tiempo que necesitaríamos para construirlo.
- Es por ello que nos limitaremos a construir un pequeño subconjunto representativo de todas las posibles entradas.

Juegos de Pruebas

- Existen varios métodos de construcción de juegos de pruebas

Juegos de Pruebas

- Existen varios métodos de construcción de juegos de pruebas

  Caja Negra Este método sólo necesita la especificación del módulo, con lo cual los juegos de prueba se pueden generar antes de la codificación del módulo. Las estrategias básicas para la construcción son:

Juegos de Pruebas

- Existen varios métodos de construcción de juegos de pruebas

  Caja Negra Este método sólo necesita la especificación del módulo, con lo cual los juegos de prueba se pueden generar antes de la codificación del módulo. Las estrategias básicas para la construcción son:

  - Particiones equivalentes

Juegos de Pruebas

- Existen varios métodos de construcción de juegos de pruebas

  Caja Negra Este método sólo necesita la especificación del módulo, con lo cual los juegos de prueba se pueden generar antes de la codificación del módulo. Las estrategias básicas para la construcción son:

  - Particiones equivalentes
  - Análisis de valores límite

Juegos de Pruebas

- Existen varios métodos de construcción de juegos de pruebas

  Caja Negra Este método sólo necesita la especificación del módulo, con lo cual los juegos de prueba se pueden generar antes de la codificación del módulo. Las estrategias básicas para la construcción son:
  - Particiones equivalentes
  - Análisis de valores límite

  Caja Blanca En ese caso se necesita también la implementación del módulo. Las estrategias básicas para la construcción son:

Juegos de Pruebas

- Existen varios métodos de construcción de juegos de pruebas

  Caja Negra Este método sólo necesita la especificación del módulo, con lo cual los juegos de prueba se pueden generar antes de la codificación del módulo. Las estrategias básicas para la construcción son:
  - Particiones equivalentes
  - Análisis de valores límite

  Caja Blanca En ese caso se necesita también la implementación del módulo. Las estrategias básicas para la construcción son:
  - Cubrimiento de instrucciones

Juegos de Pruebas

- Existen varios métodos de construcción de juegos de pruebas

  Caja Negra Este método sólo necesita la especificación del módulo, con lo cual los juegos de prueba se pueden generar antes de la codificación del módulo. Las estrategias básicas para la construcción son:
  - Particiones equivalentes
  - Análisis de valores límite

  Caja Blanca En ese caso se necesita también la implementación del módulo. Las estrategias básicas para la construcción son:
  - Cubrimiento de instrucciones
  - Decisión y condición

Caja Negra

- El método *Caja Negra* de *particiones equivalentes* consiste en particionar el conjunto de todas las entradas posibles, para después escoger uno o más casos representativos de cada clase. Para tener éxito es necesario que se den los siguientes requisitos:

Caja Negra

- El método *Caja Negra* de *particiones equivalentes* consiste en particionar el conjunto de todas las entradas posibles, para después escoger uno o más casos representativos de cada clase. Para tener éxito es necesario que se den los siguientes requisitos:
  1. Si un error es detectado por un miembro de la clase, la probabilidad de que sea detectado por otro miembro de la misma clase ha de ser alta.

Caja Negra

- El método *Caja Negra* de *particiones equivalentes* consiste en particionar el conjunto de todas las entradas posibles, para después escoger uno o más casos representativos de cada clase. Para tener éxito es necesario que se den los siguientes requisitos:
  1. Si un error es detectado por un miembro de la clase, la probabilidad de que sea detectado por otro miembro de la misma clase ha de ser alta.
  2. Cada representante escogido deberá invocar el máximo número de condiciones diferentes para así minimizar el número total de casos necesarios.

Caja Negra

- El método *Caja Negra* de *análisis de valores límite* complementa la estrategia de particiones equivalentes seleccionando el mejor representante de cada clase, donde "mejor" significa que se sitúa en o cerca de un caso extremo (valores límite).

Caja Negra

- El método *Caja Negra* de *análisis de valores límite* complementa la estrategia de particiones equivalentes seleccionando el mejor representante de cada clase, donde "mejor" significa que se sitúa en o cerca de un caso extremo (valores límite).
- Por ejemplo, la comprobación de un módulo con una función que dados dos enteros n y k con $n \geq k > 0$ calcule el coeficiente binomial $\binom{n}{k}$, debería considerar los siguientes casos en los juegos de pruebas:

k	$n(=k)$
0	0
1	1

k	$n(=k+1)$
0	1
1	2

Caja Blanca

- El método *Caja Blanca* de *cubrimiento de instrucciones* consiste en buscar casos que obliguen a ejecutar cada instrucción del módulo, siempre considerando las diferentes estructuras de control utilizadas.
- Por ejemplo, la comprobación de la instrucción siguiente:

```
if (n > 1 && high && !low) n++;
```

exige que se incluyan dos casos en el juego de pruebas: por ejemplo, cualquier caso con $n = 1$ forzará la ejecución del `else` (implícito, ya que la rama es vacía) y el caso $n = 2$, `high` $=$ cierto, `low` $=$ falso forzará la ejecución de `n++`.

Caja Blanca

- Si ahora componemos secuencialmente dos alternativas

```
if (n > 1 && high && !low) n++;
if (n == 0 || low) n = 2 * n;
```

habremos de preparar, en principio, cuatro casos:

Ejecutado	n	$high$	low
Nada	1	cierto/falso	falso
`n++`	2	cierto	falso
`n = 2 * n`	0	cierto/falso	cierto/falso
`n++; n = 2*n`		Imposible	

Caja Blanca

- Sin embargo, no se comprueba exhaustivamente las condiciónes de los `if`. Para comprobar la primera de ellas necesitaríamos considerar como mínimo $2^3 = 8$ casos en los juegos de pruebas:

n	$high$	low
1	falso	falso
1	falso	cierto
1	cierto	falso
1	cierto	cierto

n	$high$	low
2	falso	falso
2	falso	cierto
2	cierto	falso
2	cierto	cierto

Caja Blanca

- En el método *Caja Blanca* de *decisión y condición* esta basado en los conceptos de:

Caja Blanca

- En el método *Caja Blanca* de *decisión y condición* esta basado en los conceptos de:

  Decisión Punto de un programa en el que existen más de un posible camino a seguir. Ejemplos:

Caja Blanca

- En el método *Caja Blanca* de *decisión y condición* esta basado en los conceptos de:

  Decisión Punto de un programa en el que existen más de un posible camino a seguir. Ejemplos:

  - **if** (2 posiblidades)

Caja Blanca

- En el método *Caja Blanca* de *decisión y condición* esta basado en los conceptos de:

  Decisión Punto de un programa en el que existen más de un posible camino a seguir. Ejemplos:

  - **if** (2 posiblidades)
  - **switch** (n posibilidades)

Caja Blanca

- En el método *Caja Blanca* de *decisión y condición* esta basado en los conceptos de:

  Decisión Punto de un programa en el que existen más de un posible camino a seguir. Ejemplos:

  - **if** (2 posiblidades)
  - **switch** (n posibilidades)
  - **while, for** (m posibilidades, una por vuelta)

Caja Blanca

- En el método *Caja Blanca* de *decisión y condición* esta basado en los conceptos de:

  Decisión Punto de un programa en el que existen más de un posible camino a seguir. Ejemplos:

  - **if** (2 posiblidades)
  - **switch** (n posibilidades)
  - **while, for** (m posibilidades, una por vuelta)

  Condición Componente de una decisión. Es una expresi'on booleana que no contiene los operadores booleanos (**and**, **or**. **not**).

Caja Blanca

- Por ejemplo, dada la siguiente instrucción:

```
if (n > 1 && m > 2 * n + 1) s1;
else s2;
```

  contiene dos condiciones ($n > 1$ y $m > 2 \cdot n + 1$) y dado que se trata de un **if** existen dos posibles decisiones: ejecutar `s1` o ejecutar `s2`.

- La primera decisión se dará por ejemplo, cuando $n = 2$ y $m = 6$ y la segunda cuando $n = 0$ y $m = 1$.

n	m	decision
2	6	cierto
0	1	falso

Caja Blanca

- Ahora bien a partir de las dos condiciones no sabemos si se van a combinar con **and** o con **or**. Es por ello que deberemos contemplar las cuatro posibles combinaciones:

$n > 1$	$m > 2 \cdot n + 1$	n	m
cierto	cierto	2	6
cierto	falso	2	5
falso	cierto	0	2
falso	falso	0	1

Caja Blanca

- Ahora bien a partir de las dos condiciones no sabemos si se van a combinar con **and** o con **or**. Es por ello que deberemos contemplar las cuatro posibles combinaciones:

$n > 1$	$m > 2 \cdot n + 1$	n	m
cierto	cierto	2	6
cierto	falso	2	5
falso	cierto	0	2
falso	falso	0	1

- Los cuatro casos comprueban todas las posibles condiciones y decisiones, y los juegos de pruebas se corresponderán con las dos últimas columnas.

Caja Blanca

- Ahora bien a partir de las dos condiciones no sabemos si se van a combinar con **and** o con **or**. Es por ello que deberemos contemplar las cuatro posibles combinaciones:

$n > 1$	$m > 2 \cdot n + 1$	n	m
cierto	cierto	2	6
cierto	falso	2	5
falso	cierto	0	2
falso	falso	0	1

- Los cuatro casos comprueban todas las posibles condiciones y decisiones, y los juegos de pruebas se corresponderán con las dos últimas columnas.
- En general k condiciones producen 2^k combinaciones que formarán los juegos de prueba. Dependiendo del problema hay combinaciones que son imposibles de probar.