Diseño y Realización de Servicios de
Presentación en Entornos Gráficos

INTRODUCCIÓN A LA OOP CON C++

José María Torresano
entornos.jmt@gmail.com

Octubre 2008

Empezando con C++

Empezando

```
// Un pequeño programa en
#include <iostream>

int main()
{
   std::cout << "Bienvenido a C++" << std::endl;
   return 0;
}
```

Biblioteca estándar

Función *main*. Obligatoria. Requiere devolver un tipo *int*. Algunos compiladores advierten si es del tipo *void*

Nombre de espacio estándar. *::* operador de ámbito o alcance

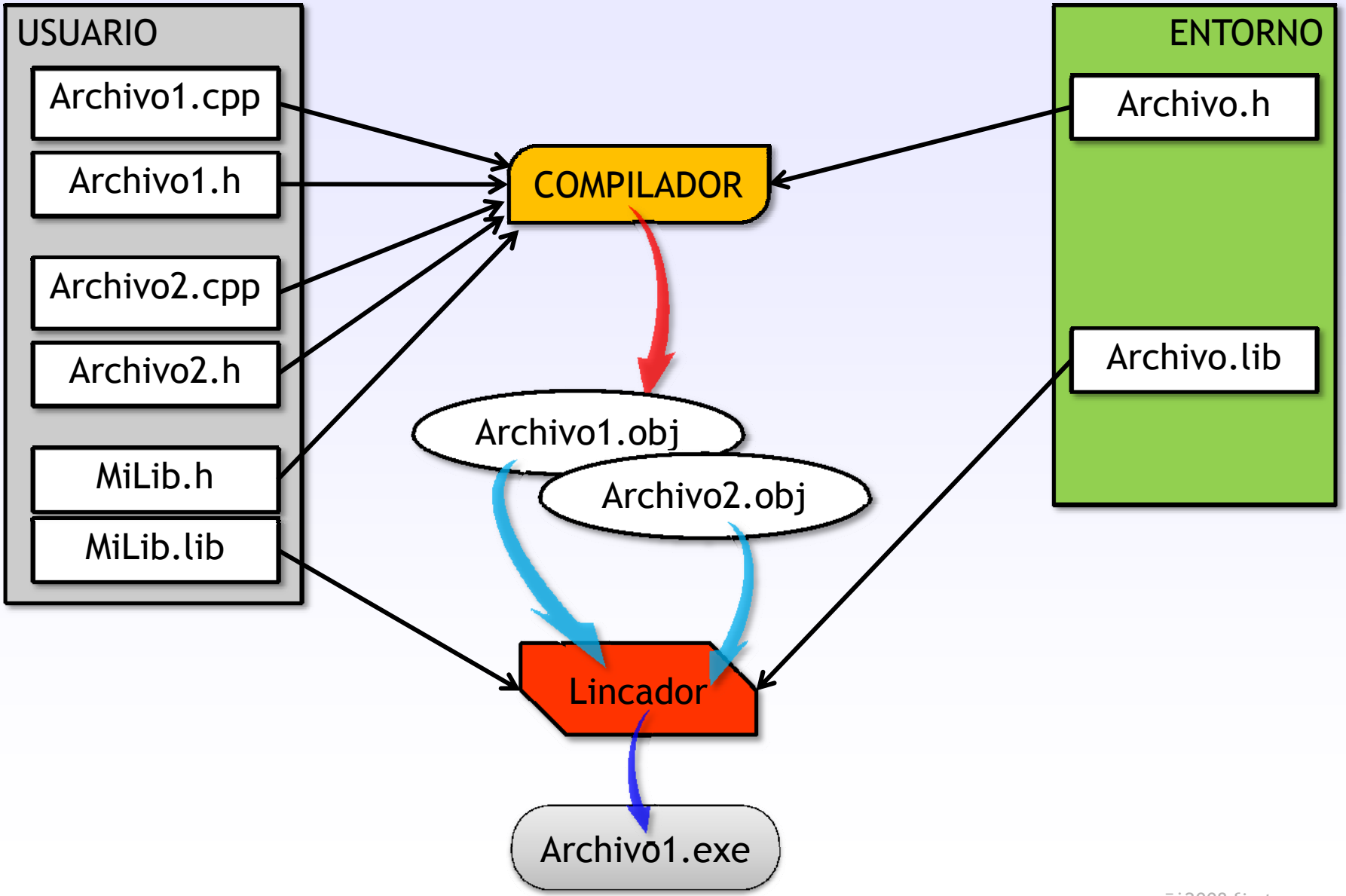
Si no aparece, y *main* devuelve un *int*, el estándar dice que el compilador devolverá *0*

<< operador de salida.

Manipulador

Compilación

Compilación y lincado



Compilación y lincado

Si existen referencias cruzadas entre los archivos de cabecera y en ellos se definen tipos de datos (estructuras, clases, definiciones globales, … el compilador llamará más de una vez a la misma cabecera por lo que se producirá un error de compilación.

Guarda que evita que un archivo de cabecera se compile más de una vez

```
#ifndef _MI_NOMBRE_DE_ARCHIVO_H
#define _MI_NOMBRE_DE_ARCHIVO_H
Contenido del archivo de cabecera
#endif
```

Algunos tipos

Tipos

`bool`

Tipo interno que representa valores booleanos (`true` o `false`)

`unsigned`

Tipo entero que contiene sólo valores positivos.

`short`

Tipo entero que debe contener al menos 16 bits.

`long`

Tipo entero que debe contener al menos 32 bits.

`size_t`

Tipo entero sin signo (de `<cstddef>`) que guarda el tamaño de cualquier objeto.

`string::size_type`

Tipo entero sin signo que guarda el tamaño de cualquier cadena.

El tipo *string*

El tipo **string** se define en la cabecera estándar <string>.

Un objeto del tipo string es una secuencia de cero o mas caracteres.

Si *n* es un entero, *c* un carácter, *is* es un flujo de entrada y *os* un flujo de salida, entonces las operaciones con **string** incluyen:

```
std::string s;
```

Define *s* como una variable del tipo std::string vacia.

```
std::string t = s;
```

Define *t* como una variable del tipo std::string que contiene una copia de *s*. *s* puede ser un literal o un std::string.

```
std::string z(n, c);
```

Define *z* como una variable del tipo std::string que contiene *n* copias del caracter *c*.

El tipo *string*

`os << s;`

Escribe los caracteres de *s*, sin cambio de formato, en el flujo definido por *os*. El resultado de la expresión es *os*.

`is >> s;`

Lee y descarta caracteres del flujo *is* hasta que encuentra un carácter que no es *espacio en blanco*. Lee los siguientes caracteres y los guarda en *s*, sobrescribiendo el valor que tenga, hasta que encuentra un *espacio en blanco*. El resultado de la expresión es *is*.

`s + t;`

El resultado es un std::string que contiene una copia de los caracteres en *s* seguidos de las caracteres en *t*. Cualquiera de las 2 puede ser una expresión literal o un caracter, pero no ambas.

`s.size();`

El número de caracteres en *s*.

El tipo *vector*

El tipo **vector**, definido en <vector>, es un tipo de la biblioteca que es un contenedor que contiene una secuencia de valores del tipo especificado. Crece dinámicamente.

`vector<T>::size_type`

Un tipo que garantiza que puede guardar el `número de elementos del vector mas grande posible.

`vector<T> v;`

Crea un vector vacio que puede albergar elementos del tipo *T*

`v.begin()`

Devuelve el valor que identifica el inicio de *v*.

`v.end()`

Devuelve el valor que identifica el final de *v*.

El tipo *vector*

`v.push_back(e)`

Agrega al final del vector el elemento *e*.

`v.pop_back()`

Elimina el último elemento del vector *v*.

`V[i]`

Accede al valor almacenado en la posición *i* del vector.

`v.at(i)`

Accede al valor almacenado en la posición *i* del vector.

`v.size()`

Devuelve el número de elementos de *v*.

`v.capacity()`

Devuelve el tamaño reservado para el vector *v*.

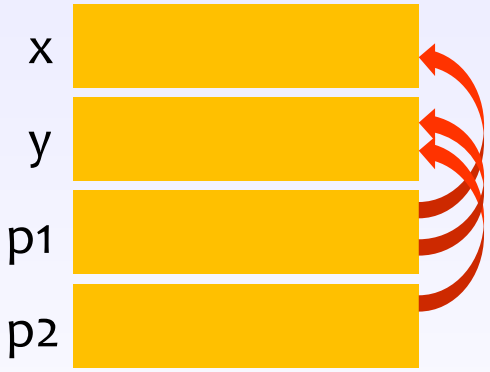
Punteros y referencias

Punteros

Puntero

Una variable que guarda la dirección de otra variable.

```
int x, y;
int* p1;
int* p2;
…
p1 = &x;
p2 = &y;
p1 = p2;
```



& Operador unario ‘dirección de’

Punteros. Tipos

`T* p;` Declaración de una variable tipo puntero donde `T` indica el tipo de variable que se apunta

Punteros a tipos estándar	`int* pi;`
Punteros a clases	`MiClase* pc;`
Punteros a arrays	`int p[10];`
Punteros a funciones	`int (*pf)(void);`
Punteros a punteros	`int** pp;`
Punteros a cualquier tipo	`void* p;`

Punteros. Operaciones

```
&objeto;
```

Tomar la dirección de un objeto

```
*puntero;
```

Desreferenciar: tomar el valor del objeto apuntado por 'puntero'

```
puntero + i;
puntero - i;
 puntero ++;
 puntero --;
ptr1 - ptr2;
```

Aritmética

Punteros y ‘constancia’

```
T* ptr;
```

Puntero a un objeto del tipo T. Sin restricciones de acceso al objeto apuntado

```
const T* ptr;
```

Puntero a un objeto constante del tipo T. No se puede utilizar *prt* para modificarlo

```
T* const ptr = &v;
```

Puntero constante a un objeto del tipo T. No se puede modificar *prt*. Se debe inicializar

```
const T* const ptr = &v;
```

Puntero constante a un objeto constante del tipo T. No se puede modificar *prt* (se debe inicializar) ni el objeto al que apunta.

Referencias

Referencia

Sinónimo o alias de un objeto.

```
int x, y;
int& r1 = x;
int& r2 = y;
…
r1 = 7;
x = 8;
r1 = r2; // NO
```

`T& r = obj;` Declaración de una referencia a un objeto del tipo `T` . *obj* es el objeto que se referencia

Referencias. Reglas

Las referencias NO son OBJETOS, SIno ALIAS a otros objetos

NO hay punteros a referencias

NO hay arrays de referencias

NO hay referencias a referencias

NO hay referencias 'constantes'

NO hay operadores para las referencias

Punteros vs Referencias

	Punteros	**Referencias**
Sintaxis	Se deben declarar expresamente	
Estatus	Son objetos → ocupan memoria	No son objetos sino sinónimos
Valor	Sus valores son direcciones	No tiene valores por si
Inicialización	Se pueden no inicializar	Se deben inicializar.
Operadores	El operador de dirección &	No tienen operadores específicos

Flujos

Flujos

Las entradas y salidas en C++ se realizan mediante flujos (*streams*).

Un flujo es un objeto que hace de intermediario entre el programa y el origen o destino de la información.

Flujos

El tipo básico para las entradas y salidas en C++ es el *stream*. C++ incorpora una jerarquía compleja de tipos *stream*. Los tipos más básicos de strems son:

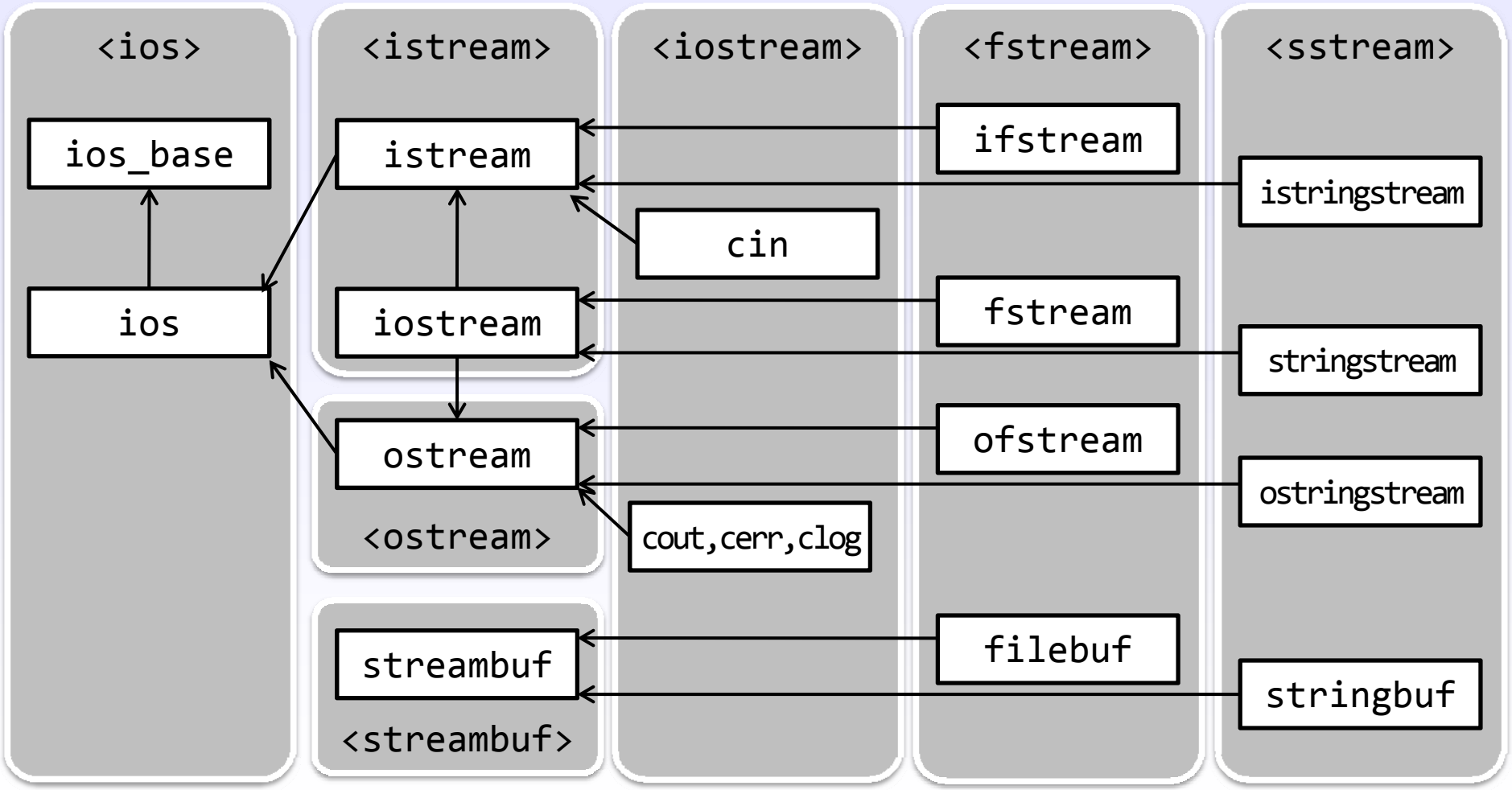
`istream cin`	variable interna de entrada unida, por defecto, al teclado.
`ostream cout`	variable interna de salida unida, por defecto, a la consola.

Ambos se encuentran en la cabecera <iostream> y *namespace* std

C++ también soporta todos los mecanismos de entrada/salida incluidos en el lenguaje C. Sin embargo, los *streams* proporcionan las capacidades de C con mejoras sustanciales.

En C++ utilizaremos siempre ***streams***

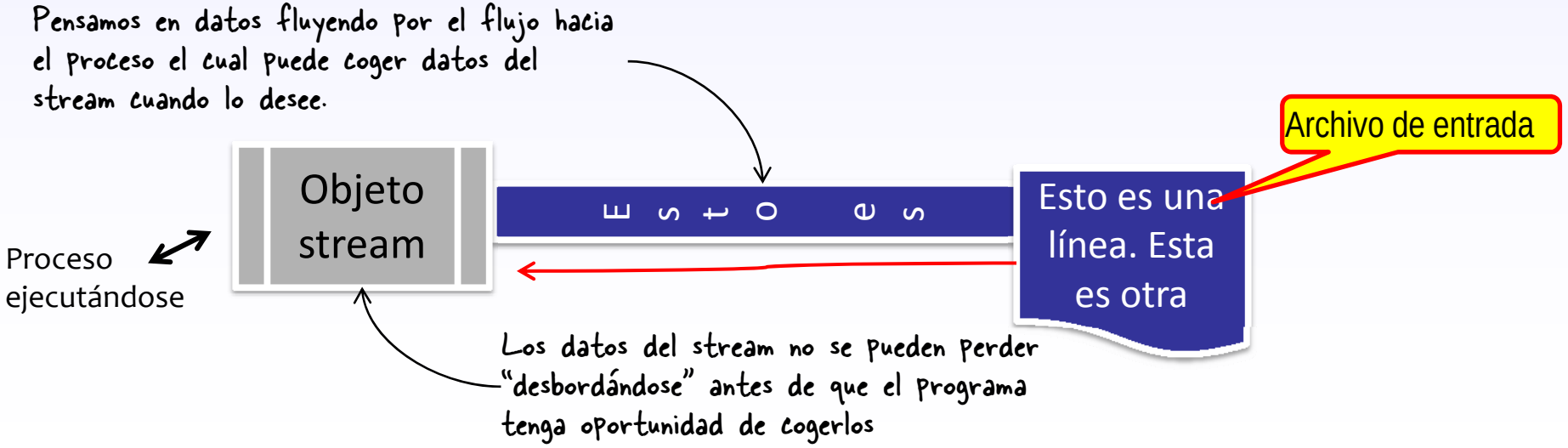
Flujos


<ios>
<istream>
<iostream>
<fstream>
<sstream>
ios_base
ios
istream
iostream
ostream
<ostream>
streambuf
<streambuf>
cin
cout,cerr,clog
ifstream
fstream
ofstream
filebuf
istringstream
stringstream
ostringstream
stringbuf

Flujos

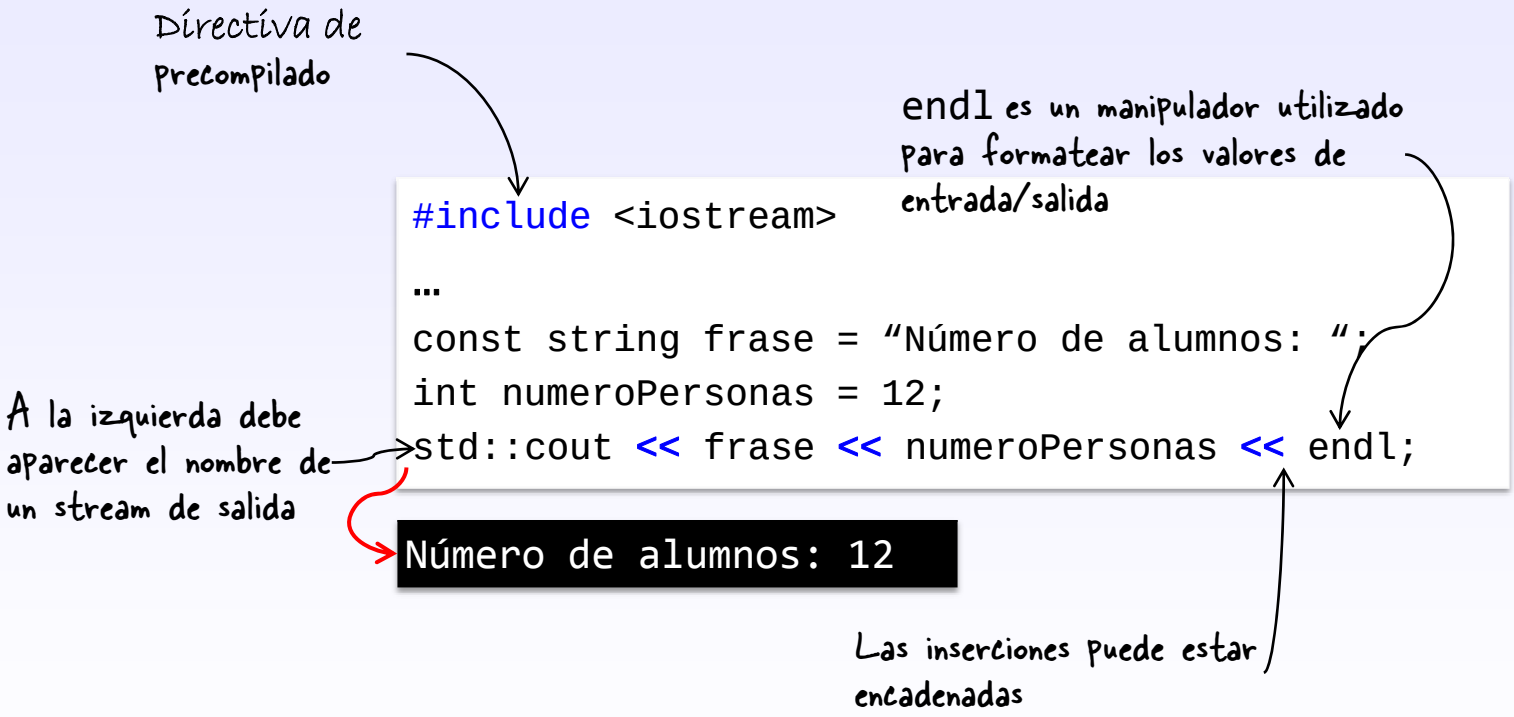
Los *streams* entrada y salida, `cin` y `cout` son realmente objetos C++ mientras que `istream` y `ostream` son clases. Esto es, por ejemplo, `cin` es el nombre de una variable del tipo `istream`.

Un *stream* proporciona una conexión entre el proceso que lo inicializa y un objeto, tal como un archivo, que puede verse como una secuencia de datos. Simplificando, un objeto *stream* es sencillamente una vista serializada de ese otro objeto. Por ejemplo



El operador inserción <<

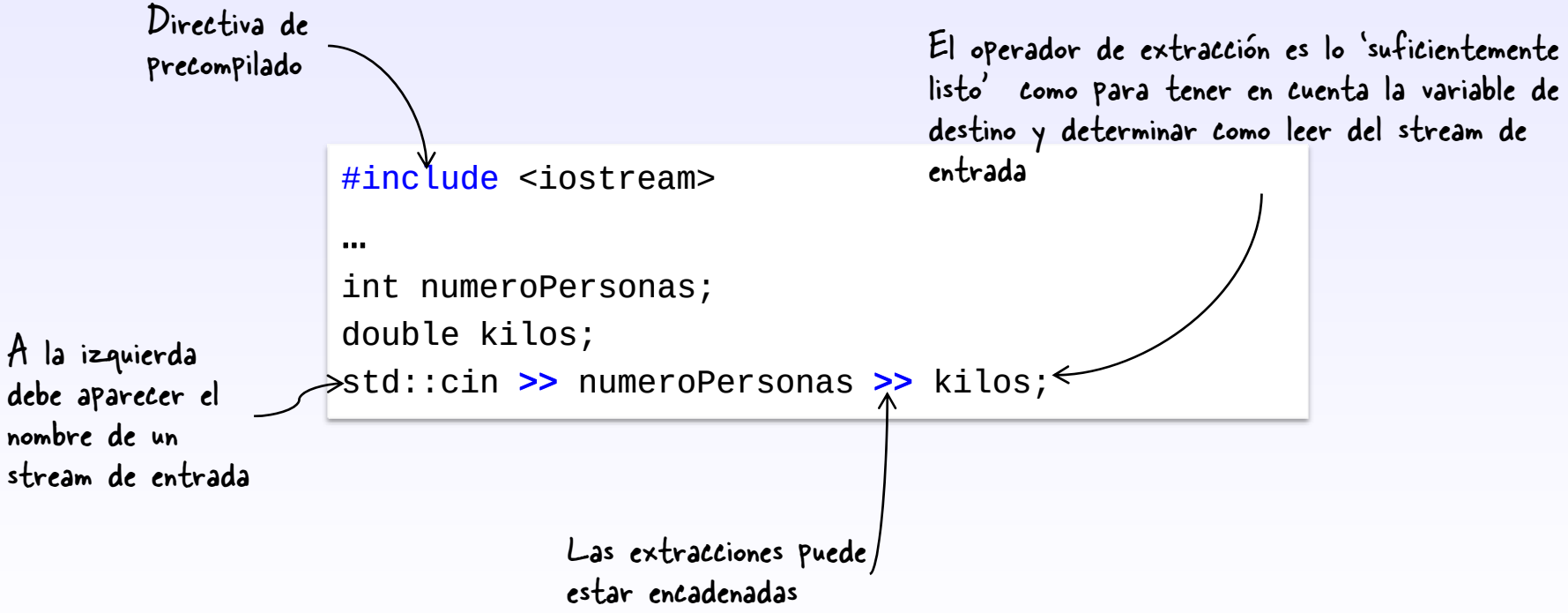
Para mostrar información de un archivo o programa, necesitamos pedirle explícitamente al ordenador que muestre la información deseada. Una forma de hacer esto en C++ es con un stream de salida y el operador inserción.



No se proporciona formateo especial por defecto: alineados, nuevas líneas, anchos, etc. los debe proporcionar el programador.

El operador extracción >>

Para obtener información de un archivo o programa, necesitamos pedirle explícitamente al ordenador que obtenga la información deseada. Una forma de hacer esto en C++ es con un stream de entrada y el operador extracción.



El operador extractor se puede utilizar para leer cadenas terminadas en *caracteres en blanco* en una variable string ignorando los espacios que la preceden y sin incluir el espacio con que termina.

El operador extracción >>

En programación, los caracteres que no producen una imagen visible en una página o en un archivo se les llama *caracteres en blanco*

Los caracteres en blanco más comunes

Nombre	Código
Nueva línea	\n
Tabulador	\t
Blanco	(espacio)
Retorno de carro	\r
Tabulador vertical	\V

Por defecto, el operador extracción en C++ quitará los espacios en blanco del stream que precedan a los caracteres normales. Si necesitamos leerlos, hay que utilizar la función `get()`

Ver más adelante

El operador extracción no consume el separador posterior

Flujo de entrada

ignore()

Para descartar y remover caracteres del flujo de entrada se puede utilizar:

Significa que lea y descarte N caracteres del flujo de entrada o hasta que lea el carácter ch y se descarte. Lo que llegue primero

```
cin.ignore(n, ch);
```

```
cin.ignore(80, '\n');
```

Lee y descarta los siguientes 80 caracteres o lee y descarta caracteres hasta leer uno de nueva línea, lo que llegue primero

```
cin.ignore();
```

Consumir un caracter

flush

Dado que se utilizan entradas/salidas con buffer es posible que las salidas no se reflejan inmediatamente en el monitor. Para asegurarse que la salida se manda a su destinatario de inmediato, se puede utilizar `flush`

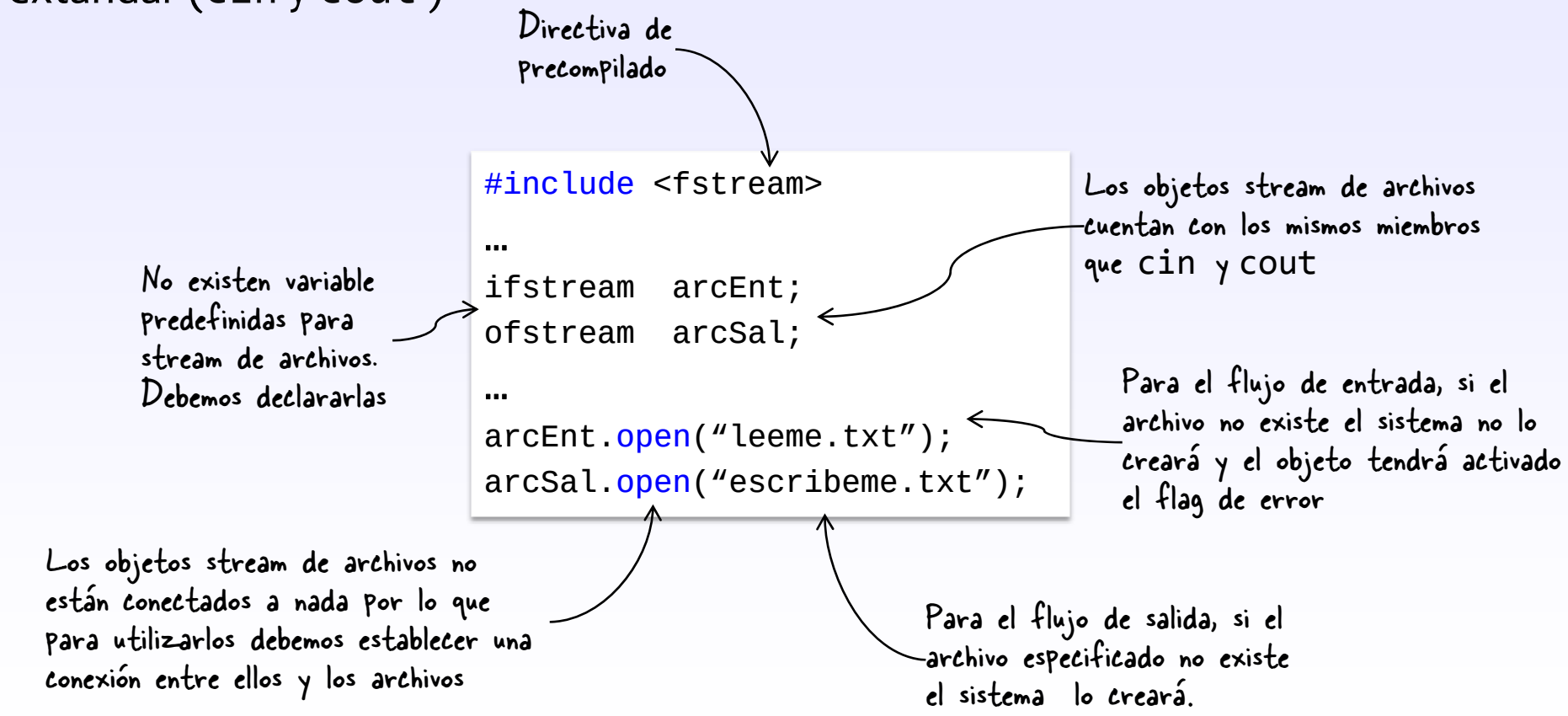
El manipulador flush asegura que la frase aparecerá en la consola antes de la petición

```
cout << "Introduce el dato: " << flush;
cin >> dato;
```

El manipulador `endl` incluye un `flush` implicito

Flujos en para E/S de archivos

C++ proporciona tipos stream para acceder a los archivos almacenados en disco. La mayorñia de las veces, funcionan exactamente igual que los stream de E/S extandar (`cin` y `cout`)



Flujos en para E/S de archivos

Otra forma de conectar los stream con los archivos

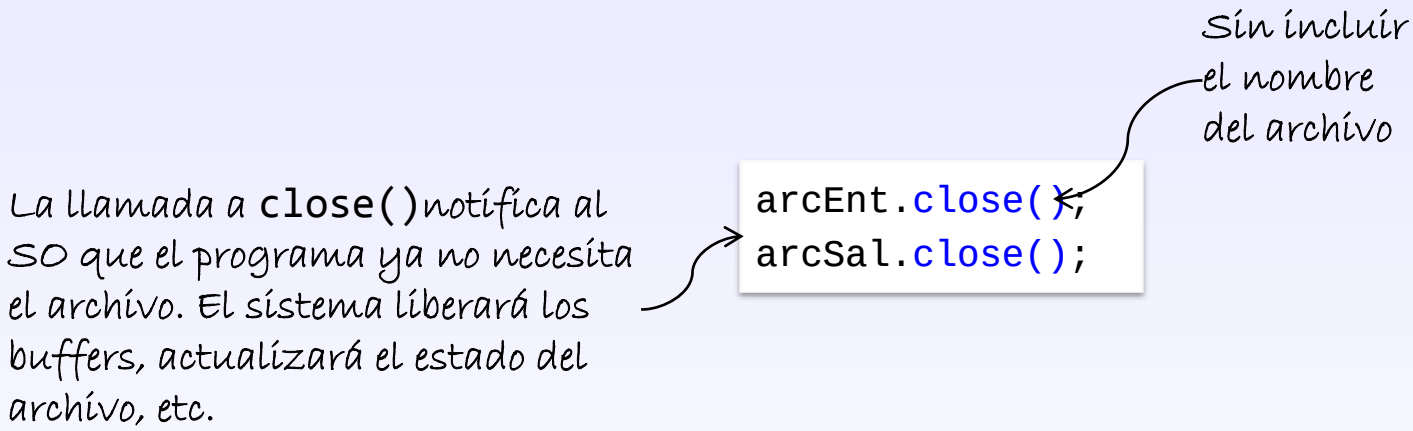
Otra mas

```
#include <fstream>

…
ifstream  arcEnt("leeme.txt");
ofstream  arcSal("escribeme.txt");

…
string nomArc = "QueMeLeas.txt";
ifstream  arcLec(nomArc.c_str());
```

Si utilizamos variables **string** para almacenar los nombres de los archivos, debemos convertirlas a cadenas del tipo C para poder utilizarlas para conectar los streams

Flujos en para E/S de archivos

Cuando un programa no necesita un archivo, debe cerrarlo utilizando el método `close()` asociado con cada objeto stream de archivo.



Siempre es mejor cerrar explícitamente los archivos aunque C++ los cierre automáticamente cuando las variables stream de archivos salgan fueran de su ámbito.

Formateo de salidas numéricas

```
#include <iomanip>
```

`setw(n)`:

Pone la anchura del campo (número de espacios el que se muestra el valor. *n* debe ser entero. Sólo se aplica a la salida del siguiente dato.

`setprecision(n)`:

Pone la precisión, el número de dígitos que se muestran después de la coma decimal. *n* debe ser entero. Se aplica hasta que se encuentra otro `setprecision()`.

Además, para activar el manipulador `setprecision()` debes insertar los 2 siguientes manipuladores una vez. Sino es así, `setprecision()` fallará y hará que los valores reales con 0 en la parte decimal no se impriman.

```
flujo << fixed << showpoint;
```

Otros manipuladores útiles y uso fácilmente deducible:
`bin`, `hex`, `octal`, `dec`, `scientific`,.

Justificación

Justificación Alinear los datos horizontalmente

La justificación por defecto en los campo de salida es a la derecha con relleno a la izquierda. Para cambiarlo:

```
cout << fixed << showpoint;
string nombre = "Picapiedra, Pedro";
double salario = 8.56;
double horas = 45.2;
cout << "01234567890123456789012345678 9" << endl;
cout << left;                    // justificación izquierda
cout << setw(20) << nombre;
cout << right;                   // justificación derecha
cout << setw(10) <<setprecision(2) << salario * horas << endl;
```

```
01234567890123456789012345678 9
Picapiedra, Pedro       386.91
```

Relleno

Relleno

Caracteres utilizados para rellenar el espacio no utilizado del campo de salida

El relleno por defecto en los campo de salida es el espacio en blanco. Se puede cambiar con el manipulador `setfill()`

```
int ID = 423562;
cout << "0123456789" << endl;
cout << setw(10) << ID << endl;
cout << setfill('0');     // rellenar con ceros
cout << setw(10) << ID << endl;
cout << setfill(' ');     // reset a espacios
```

```
0123456789
    423562
0000423562
```

Fallo en la entrada

Cuando intentamos leer un valor de un stream de entrada, el operador extractor o otras funciones de entrada tienen en cuenta el tipo variable que va a guardar el dato. Si hay una conversión por defecto entre los datos del stream y la variable de destino, no hay problema.

En el caso de que no exista o la entrada sea incompatible con el tipo de la variable que la debe guardar, se produce un fallo en el stream de entrada. El efecto que tiene el fallo en la variable destino depende del coimpilador. En Visual C++, la variable normalmente no se modifica (`string` es la excepción)

La variable stream activa un flag interno indicando el 'estado de fallo'. Los consiguientes intentos de leer del stream fallarán. Es responsabilidad del programador diseñar su código para que gestioe este tipo de errore.s

Leyendo de flujos

```
while (cin >> x)
  bloque
```

Lee un valor del tipo apropiado en *x* y comprueba el estado del flujo de entrada. Si el flujo está en un estado de error, la comprobación falla; en caso contrario, se ejecuta el cuerpo del *while*.

Hay varias formas de que intentar leer de un flujo conlleve un error:

1. Hemos llegado al final del archivo de entrada.
2. Hemos encontrado una entrada que es incompatible con el tipo de variable que intentamos leer.
3. El sistema ha detectado un fallo de hardware en el dispositivo de entrada.

Flujo de entrada

`getline()`

Función de la biblioteca estándar para leer caracteres hasta un delimitador

Lee desde la posición actual hasta el carácter de nueva línea '\n'. Incluye los posibles caracteres al inicio.

```
istream& getline ( istream& is, string& str );
istream& getline ( istream& is, string& str, char delim );
```

Igual que la des dos argumentos pero ahora le pasamos el carácter en el que debe parar

El delimitador o '\n' se eliminan del stream peor no se incluye en la variable destino

Seleccionando el carácter apropiado, la función `getline()` puede leer texto formateado con delimitadores conocidos.

Flujo de entrada

get()

Función del stream de entrada para leer caracteres.

```
int get ();
istream& get ( char& ch );
istream& get ( char* s, streamsize n);
istream& get ( char* s, streamsize n, char delim);
istream& get ( streambuf& sb);
istream& get (streambuf& sb, char delim);
```

peek()

Función del stream que proporciona un método de examinar el siguiente carácter del stream de entrada sin removerlo de él.

```
char siguienteCar;
siguienteCar = cin.peek();
```

putback()

Función del stream que proporciona un método de devolver un carácter al stream de entrada.

```
char siguienteCar = '?';
cin.putback(siguienteChar);
```

Flujos

Manipuladores

Son objetos especiales cuyas 'salidas' manipulan los flujos.

Manipulador	*Clase*	*Significado*
`std::flush`	`std::ostream`	Limpia el buffer de salida.
`std::endl`	`std::ostream`	Escribe un '\n' y limpia el buffer de salida
`std::ends`	`std::ostream`	Escribe un '\0' y limpia el buffer de salida
`std::ws`	`std::istream`	Salta los espacios en blanco

Flujos

Estatus

Para representar las condiciones del flujo se utilizan constantes de bit (*flags*)

Constantes bit	*Significado*
goodbit	Todo está perfecto
eofbit	Fin de archivo
failbit	Fallo: error, pero el flujo es utilizable
badbit	Error fatal: no se pude utilizar el flujo

Métodos	*Significado*
good()	Todo está perfecto (se activa ios::goodbit)
eof()	Fin de archivo (se activa ios::eofbit)
fail()	Fallo (se activa ios::failbit o ios::badbit)
bad()	Error fatal (se activa ios::badbit)
rdstate()	Devuelve los bit de estado activados en ese momento
clear()	Borra o activa bits individuales

POO. Clases. Objetos. Encapsulación

Clases y Objetos

Clases Segmentos de código que se utilizan para definir un tipo de objeto.

En el código se define el interfaz que presenta al exterior el objeto: *propiedades*, *métodos* y *eventos*

… y la implementación interna del objeto: el código que se ejecuta cuando se llama a un método o propiedad o cuando se lanza un evento.

Escribimos código para crear clases y código que usa esas clases para crear objetos en tiempo de ejecución.

Clases y Objetos

Objetos Instancias de clases.

Los objetos guardan un conjunto de valores para las propiedades definidas en la clase. Tienen estado. Existen en memoria.

Si las clases son los planos, los objetos son los edificios.

Interactuamos con los objetos al leer/escribir sus propiedades, al llamar a sus métodos y al responder a sus eventos.

Clases y Objetos

Método

Es un algoritmo asociado a un objeto (o a una clase de objetos), cuya ejecución se desencadena tras la recepción de un mensaje.

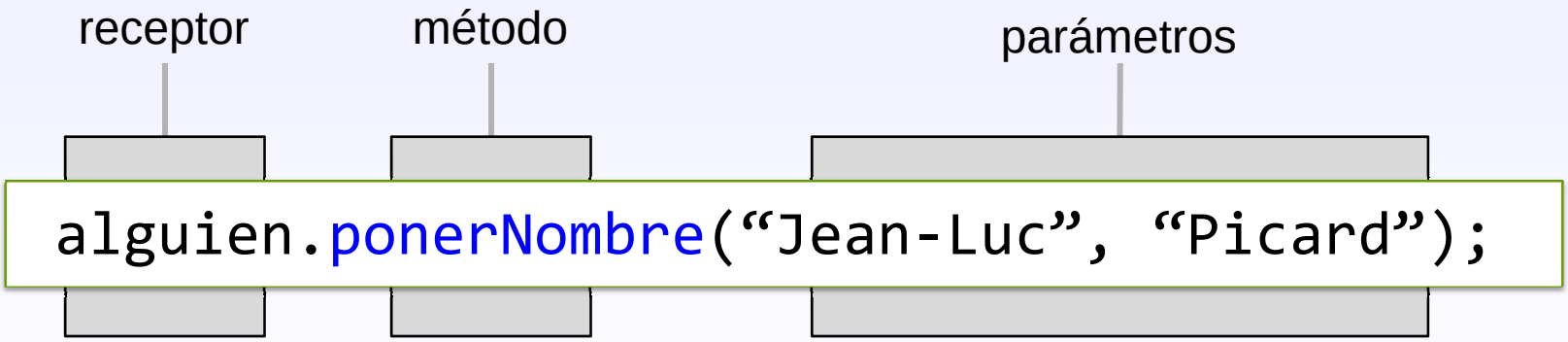
Es lo que el objeto puede hacer

Un método puede producir un cambio en las propiedades del objeto, o la generación de un evento con un nuevo mensaje para otro objeto del sistema.

Clases y Objetos

Mensaje

Es una comunicación dirigida a un objeto, que le ordena que ejecute uno de sus métodos.



Clases y Objetos

Propiedad

Es contenedor de un tipo de datos asociados a un objeto (o a una clase de objetos) cuyo valor puede ser alterado por la ejecución de algún método.

Definen el estado del objeto

Clases y Objetos

Evento

Es un suceso en el sistema (tal como una interacción del usuario con la máquina, o un mensaje enviado por un objeto).

El sistema maneja el evento enviando el mensaje adecuado al objeto pertinente. También se puede definir como evento, a la reacción que puede desencadenar un objeto, es decir la acción que genera.

Clases y Objetos

Clases y Objetos

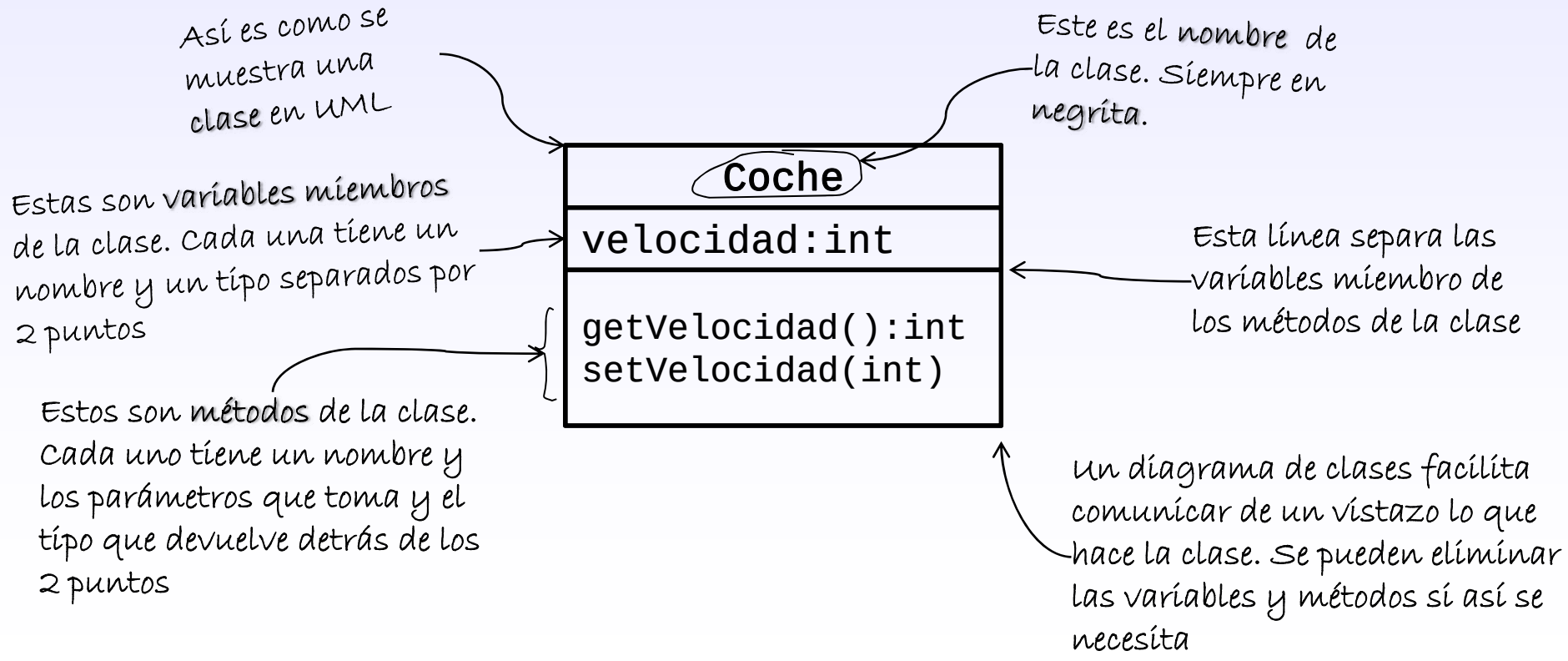
La programación orientada a objetos proporciona 4 ventajas:

1. Encapsulación.
2. Abstracción.
3. Polimorfismo.
4. Herencia.

UML y diagramas de clase

Unified Model Language

Lenguaje utilizado para comunicar los detalles de nuestro código y la estructura de la aplicación a otros sin entrar en detalles innecesarios.



Clases y Objetos

Constructores

Es una operación (un método), que inicializa los atributos (propiedades) del objeto en el momento de su creación.

Cuando se crea un objeto surge la necesidad de asignar valores iniciales a las variables que contiene, esa es la misión del constructor.

Para definir un **constructor** se utiliza el propio **nombre de la clase**. Se pueden definir **varios** constructores para una misma clase, todos ellos tendrán el mismo nombre (el de la clase) pero se distinguirán por sus argumentos. No deben devolver nada. Si no se define ningún constructor, el compilador creará uno vacio por defecto que no hace nada

Clases y Objetos

Constructor copia

Es una operación (un método), que crea un objeto a partir de un objeto existente del mismo tipo.

Para definir un **constructor copia** se utiliza el propio **nombre de la clase** con un único argumento que es una referencia constante objeto de la clase. No debe devolver nada. Si no se define ningún constructor, el compilador creará uno que copiará miembro a miembro.

```
Fraccion a;
…
Fraccion b = a;
```

```
Fraccion a;
…
Fraccion b(a);
```

Constructor copia

```
Fraccion a;
…
Fraccion b;
a = b;
```

```
Fraccion a;
…
a = Fraccion b;
```

Constructor por defecto

Clases y Objetos

Destructores

Es una operación (un método), que se ejecuta cuando el objeto deja de existir.

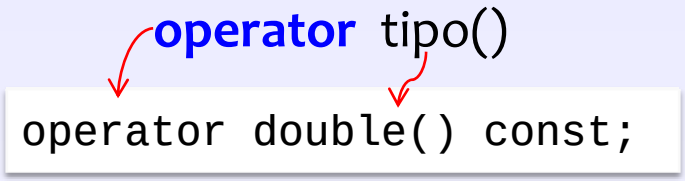
Cuando un objeto deja de existir debe liberar los recursos utilizados al sistema. Esta es la función del destructor.

Para definir un **destructor** se utiliza el propio **nombre de la clase** precedido del carácter **~** No puede tener argumentos ni retornar valores. Es único. Si no se define ningún constructor, el compilador creará uno vacio por defecto que no hace nada

Clases y Objetos

Funciones de conversión

Se declaran con la palabra operator seguida del tipo de conversión.



Se declaran sin ningún tipo de valor de retorno aunque la función devuelve un valor que es el objeto una vez realizada la conversión.

```
Fraccion::operator double() const {
  return double(numerador)/double(denominador)
}
```

Las funciones conversiones tienen las misma prioridad que las conversiones automáticas que realiza el compilador lo que puede dar errores en la interpretación de las conversiones.

Se recomienda no utilizar este tipo de funciones y en cambio definir un método específico

```
double Fraccion::toDouble() const {
  return double(numerador)/double(denominador)
}
```

Clases y Objetos

Encapsulación

La habilidad de poner todos los datos y operaciones sobre los datos en una estructura controlada y abstracta, de forma que quede aislada del resto del sistema

Las clases permiten al programador la libertad de encapsular variables y métodos en una única estructura, de forma que los únicos que podrán acceder a estos datos y operaciones son los objetos de esa clase.

Clases y Objetos

Campo o ámbito

La parte del programa donde es accesible un identificador, variable, función, …

Niveles de acceso a los miembros de una clase

1. ***public***: Público, accesible por métodos de cualquier clase
2. ***protected***: Protegido, accesible sólo por los métodos de su clase y descendientes
3. ***private***: Privado, sólo accesible por métodos de su clase

Por defecto, el acceso a los miembros de una clase es ***privado***

C++

En C++, una clase es un fragmento de código contenido entre las declaraciones `class { … };`.

```
class nombreClase {
  [codigo, declaraciones, …]
};
```

```
class Fraccion {
  int numero;
  int denominador;
  void imprimir();
};
```

C++. Organización del código

La interfaz de la clase se incluye en un archivo cabecera

```
// ----------------------------------------
// cuenta.h - Interfaz de la clase Cuenta
// ----------------------------------------
#ifndef _CUENTA_H
#define _CUENTA_H
class Cuenta {
  char numero[20]; // Variables miembro
  char titular[80];
  float saldo;
  float interes;
public:
  // Funciones miembro
  void ingreso (float cantidad);
  void reintegro (float cantidad);
  void ingreso_interes (void);
  int esta_en_rojos (void) { return saldo < 0; }
  float ver_saldo (void) { return saldo; }
};
#endif
```

Las funciones *inline* se implementan directamente en la interfaz de la clase

C++. Organización del código

La implementación de las funciones miembro que no sean ***inline***, se realiza en un fichero fuente con el mismo nombre que el fichero de cabecera correspondiente.

```
// ------------------------------------------------
// Cuenta.cpp - Implementación de la clase Cuenta
// ------------------------------------------------
#include "cuenta.h"
void Cuenta::ingreso (float cantidad)
{
saldo += cantidad;
}
void Cuenta::reintegro (float cantidad)
{
saldo -= cantidad;
}
void Cuenta::ingreso_interes (void)
{
saldo += saldo * interes / 100;
}
// Fin de Cuenta.cpp -----------------------------
```

C++. Definición de operadores

La definición de operadores es una característica avanzada de algunos lenguajes orientados a objetos que permite definir el comportamiento de determinados operadores (+, -, *, etc.) sobre una clase de objetos

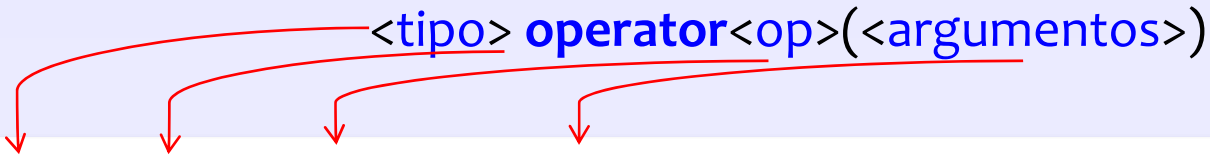
En C++ puede definirse el comportamiento de casi cualquier operador:
+, -,*, /, ++, ,+=, =,*=, /=, &,>, <, ==, >=, <=, =, >,(), ...

Podemos asociar una implementación a un operador que no tiene por qué corresponderse con su significado. Sin embargo esto no es nada recomendable.

La definición de un operador sobre una clase se realiza añadiendo una función miembro con la forma: <tipo> **operator**<op>(<argumentos>)

```
Cuenta &operator+= (float cantidad) {
  ingreso (cantidad);
  return *this;
}
```

C++. Definición de operadores




```
Fraccion operator * (Fraccion f) {
  return Fraccion(numerador * f.denominador, denominador * f.denominador);
}
```

Observar que aunque el operador es binario (operador con 2 operadores), sólo se declara con un parámetro. El otro parámetro, el del propio objeto, está implícito. En operadores unario, no aparece ningún argumento.

Se puede pensar en los operadores como que a*f es, en realidad, a.operator*(f)

C++. Asignación ≠ inicialización

Aunque se utilice el mismo símbolo (=), la asignación y inicialización no son lo mismo. Las diferencias importantes son 2:

1. La asignación (*operator=*) siempre necesita de un valor previo.
2. La inicialización crea el valor

La inicialización se da en:

- ✓ Declaración de variables
- ✓ En parámetros por valor de las funciones
- ✓ En retornos de funciones por valor
- ✓ En constructores

} Controlada por los constructores

La asignación se da sólo cuando se utiliza el operador = en una expresión.

Controlada por el operator=

C++. this

Cada objeto de una clase tiene su propia estructura de datos, pero todos comparten el mismo código. Para que un método conozca la identidad del objeto particular para el que se ha invocado, C++ proporciona un puntero a dicho objeto denominado *this*.

```
Fraccion frac(2,3);
frac.Imprimir();

…
Void Imprimir(void)
{
  cout << numerador << "/" << denominador << endl;
}
```

En realidad:

```
Fraccion *const this = &frac;
this->Imprimir();
```

En realidad:

```
cout << this->numerador << "/" << this->denominador << endl;
```

Clases y Objetos

Funciones `friends`

Las funciones `friend` pueden acceder a los miembros privados de la clase que las declara como amigas.

Se declaran los prototipos de las funciones dentro de la clase antecedida con la palabra `friend`. La implementación de la función no lleva la palabra `friend`.

```
class Fraccion {
  friend bool operator < (const Fraccion&, const Fraccion&);
};

bool operator < (const Fraccion& a, const Fraccion& b) {
  return a.numerador * b.denominador < a.denominador * a.nominador
}
```

`friend` no es transitiva → 'los amigos de mis amigos no tienen por que ser mis amigos'

Excepciones

Excepciones

Para gestionar los errores en situaciones inesperadas se utilizan, normalmente, flags de estatus, gestores de errores y valores de retorno. Estos casos de error se tiene que comprobar y tratar en el código del programa lo que hace que no quede claro dónde se sigue la marcha normal del programa y dónde se tratan los errores.

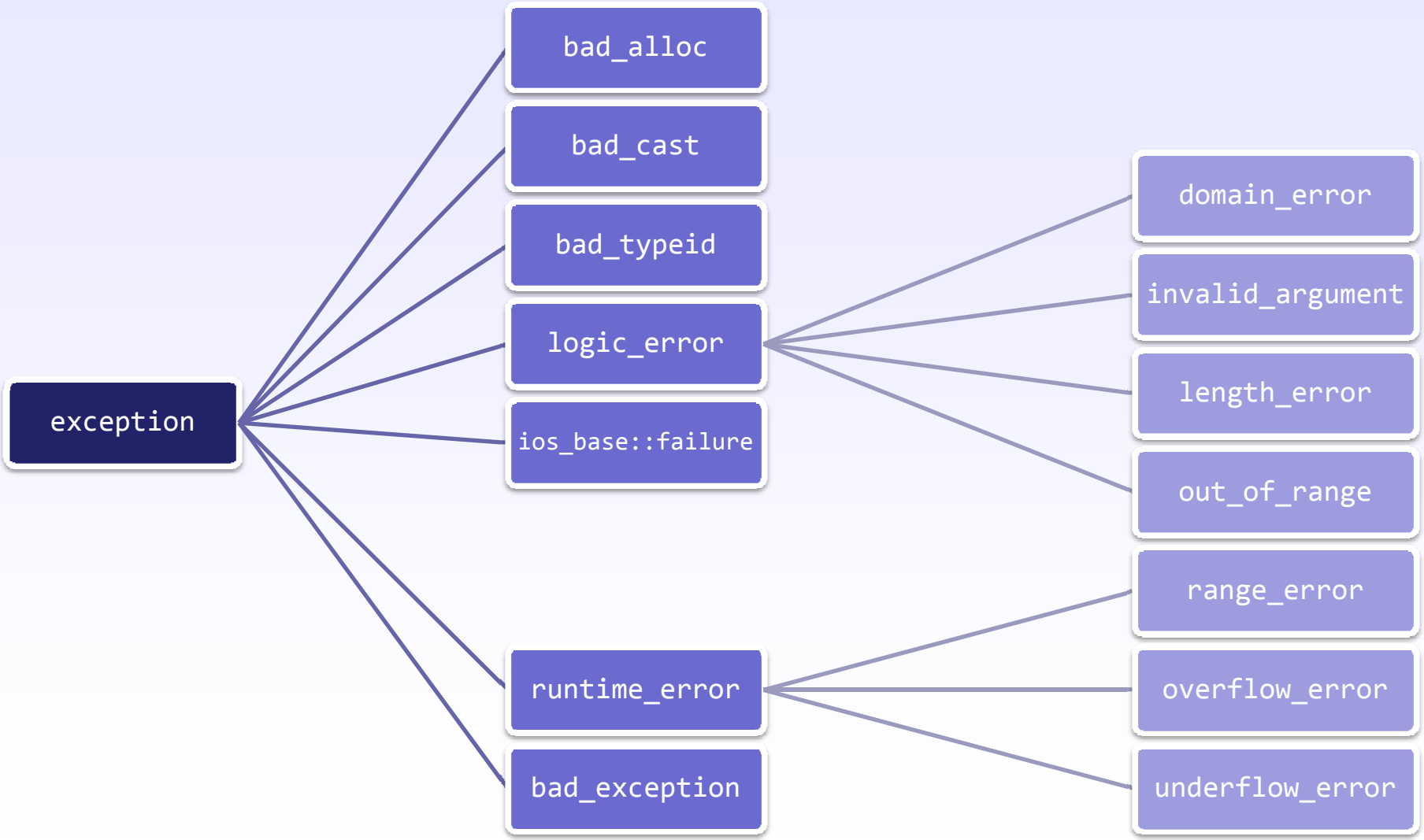
En los lenguajes orientados a objeto, la gestión de errores inesperados se hace mediante excepciones, excepciones que son objetos.

Las excepciones no tienen que ser necesariamente errores y no todos los errores son excepciones.

Gestión de excepciones

- Una excepción es cualquier condición de error o comportamiento inesperado que se produce durante la ejecución de una aplicación.
- Las excepciones pueden aparecer por un fallo en el código, recursos del sistema que no están disponibles, etc.
- Podemos utilizar gestores de errores estructurados para reconocer errores de ejecución tan pronto como ocurran en el programa, suprimir los errores no deseados y ajustar las condiciones del programa de forma que la aplicación pueda tomar el control y seguir ejecutándose.
- En C++ una excepción es un objeto que heredado de la clase **exception**

Gestión de excepciones



Gestión de excepciones

Instrucción *throw*

- Se utiliza para lanzar el correspondiente objeto excepción.
- Pasa del flujo normal de datos a la gestión de excepciones.

Bloque *try*

- Poner el código que pueda causar la excepción dentro este bloque
- Cuando ocurra un error, C++ ignorará el resto del código del bloque y saltará al bloque catch

Bloques *catch*

- Especificar la excepción que queremos atrapar o atrapar cualquier excepción. Una por bloque
- Cada bloque es un manejador de excepciones y controla la excepción del tipo indicado en su argumento
- El código del bloque correspondiente se ejecutará si se lanza esa excepción. En caso contrario, no se ejecutará.

Gestión de excepciones

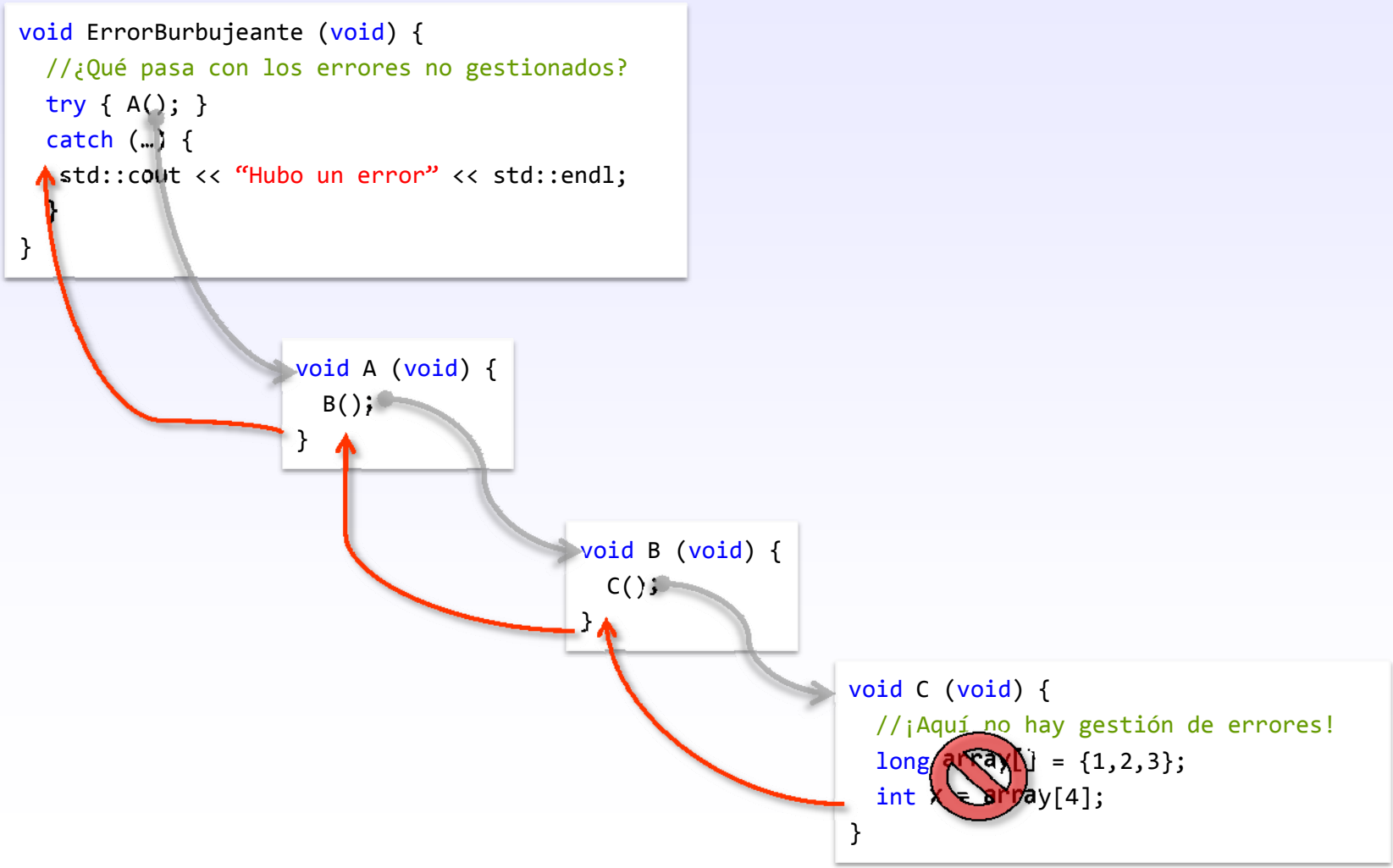
```
try {
    int x, a[] = {1, 2, 3};
    x = a[4];
    std::cout << "Esto no va a salir") << std::endl;
}
catch (const std::range_error& e) {
    x = -1;
    std::cout << e.what << std::endl;
}
catch (const std::exception& e) {
    x = 0;
    std::cout << e.what << std::endl;
}
std::cout << "x = " << x << std::endl;
```

Lógica del programa

Manejadores de excepciones

Gestión de excepciones

Propagación de excepciones



Gestión de excepciones

- Si se dispara una excepción, se crea el objeto correspondiente y se recorren todos los bloques superiores hasta que se interpreta el objeto.
- Si no se gestiona la excepción, se dispara un error de programa que causa la terminación anómala del mismo.
- Hay varias clases estándar de excepciones en C++. `what()` devuelve una cadena específica de la excepción.
- El uso de `throw` nos permite lanzar excepciones.
- Las excepciones se tienen que declarar de la forma '`const tipo&`'
- Un `catch(…)` 'caza' todo tipo de excepciones.

POO. Herencia. Polimorfismo

Herencia

La herencia es un mecanismo que permite la reutilización de código. Con ella se pueden crear nuevas clases a partir de clases ya hechas.

En la herencia, las clases derivadas "heredan" los datos y las funciones miembro de las clases base, pudiendo las clases derivadas redefinir estos comportamientos (polimorfismo) y añadir comportamientos nuevos propios de las clases derivadas

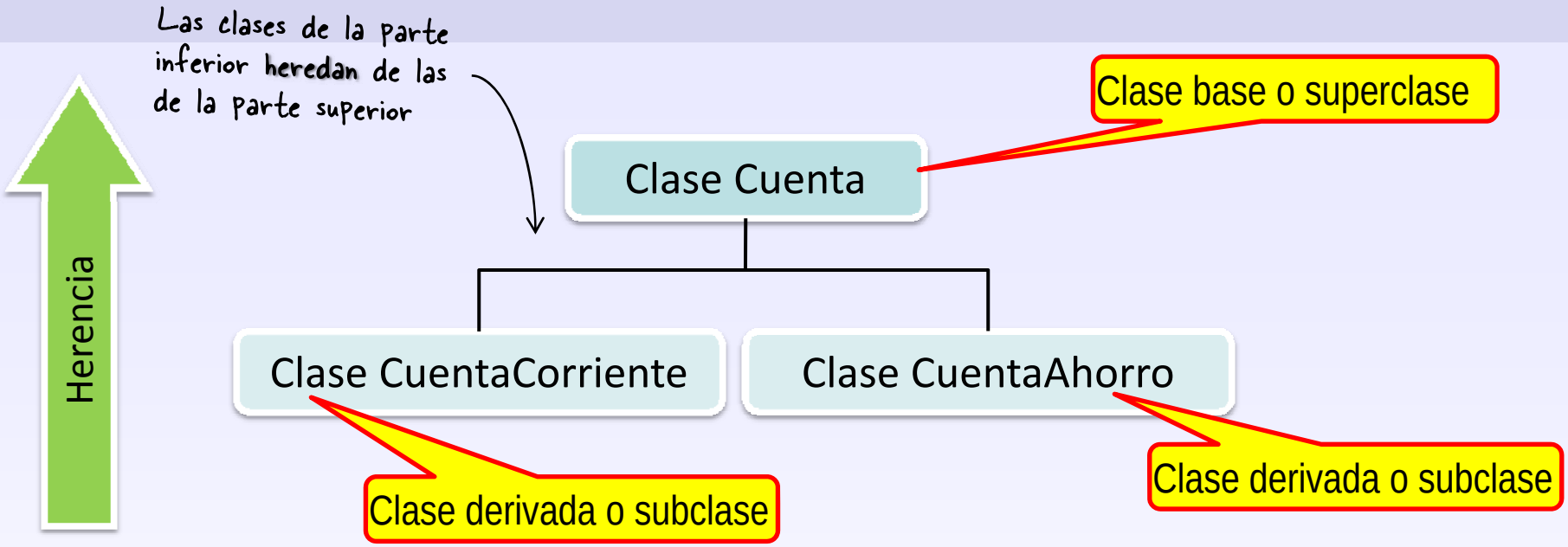
Clase base

La clase existente. También llamada ***superclase***

Clase derivada

La nueva clase derivada de la clase base. También llamada ***subclase***

Herencia



Con la herencia todas las clases están clasificadas en una jerarquía estricta: cada clase tiene su superclase (la clase superior en la jerarquía), y cada clase puede tener una o más subclases (las clases inferiores en la jerarquía).

Herencia

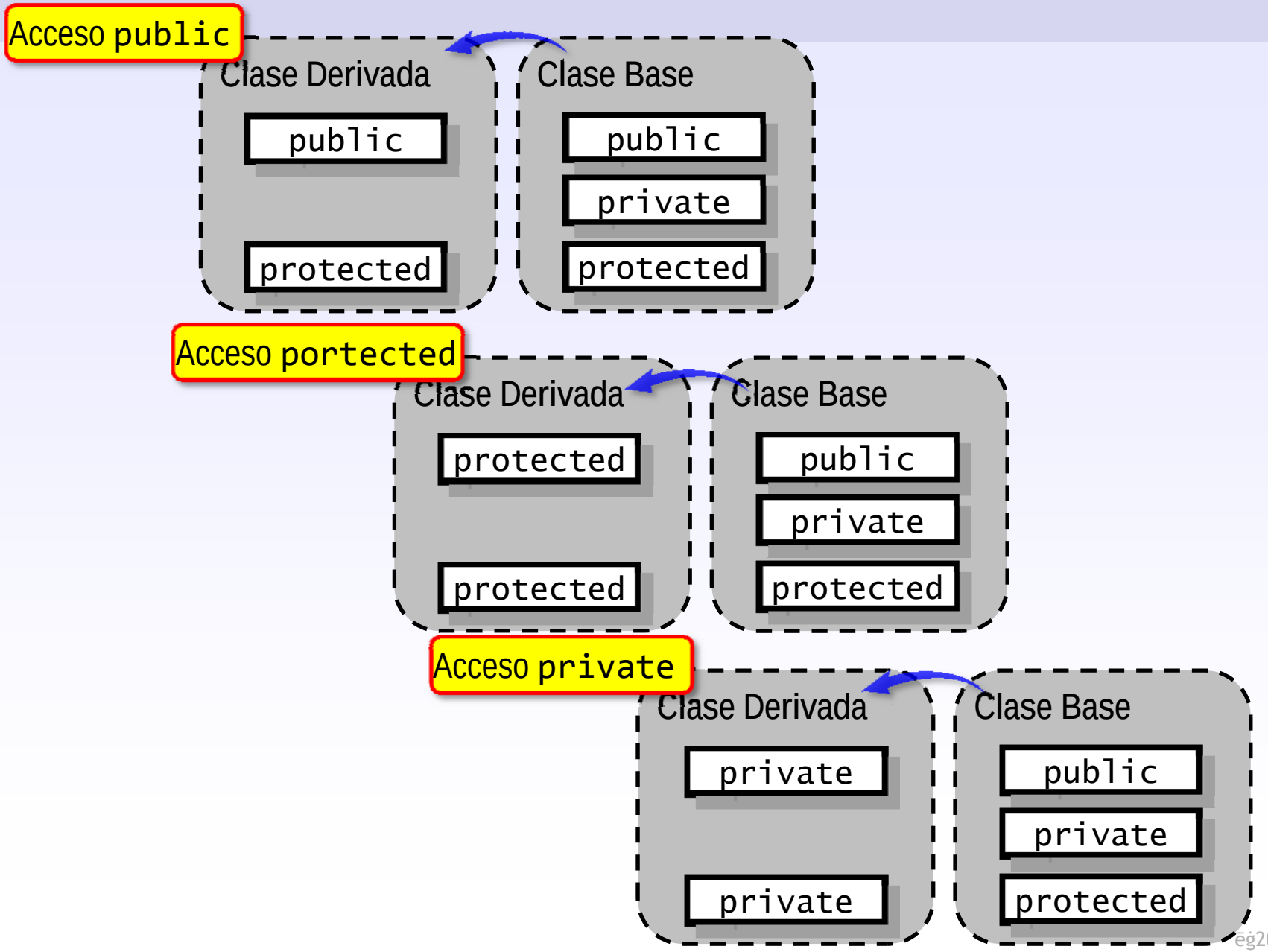
En C++, para declarar una clase como heredera de otra se utiliza la siguiente sintaxis:

```
class nombreClaseDerivada : [acceso] nombreClaseBase {
  [codigo, declaraciones, …]
};
```

public, private, protected

- Una clase derivada hereda todos los miembros de su clase base, excepto los constructores y destructor.
- Una clase derivada no tiene acceso a los miembros privados de su clase base pero si puede acceder a los miembros públicos y protegidos.
- Una clase derivada puede añadir sus propios atributos y métodos
- Los miembros heredados por una clase derivada pueden, a su vez, ser heredados por más clases derivadas de ella.

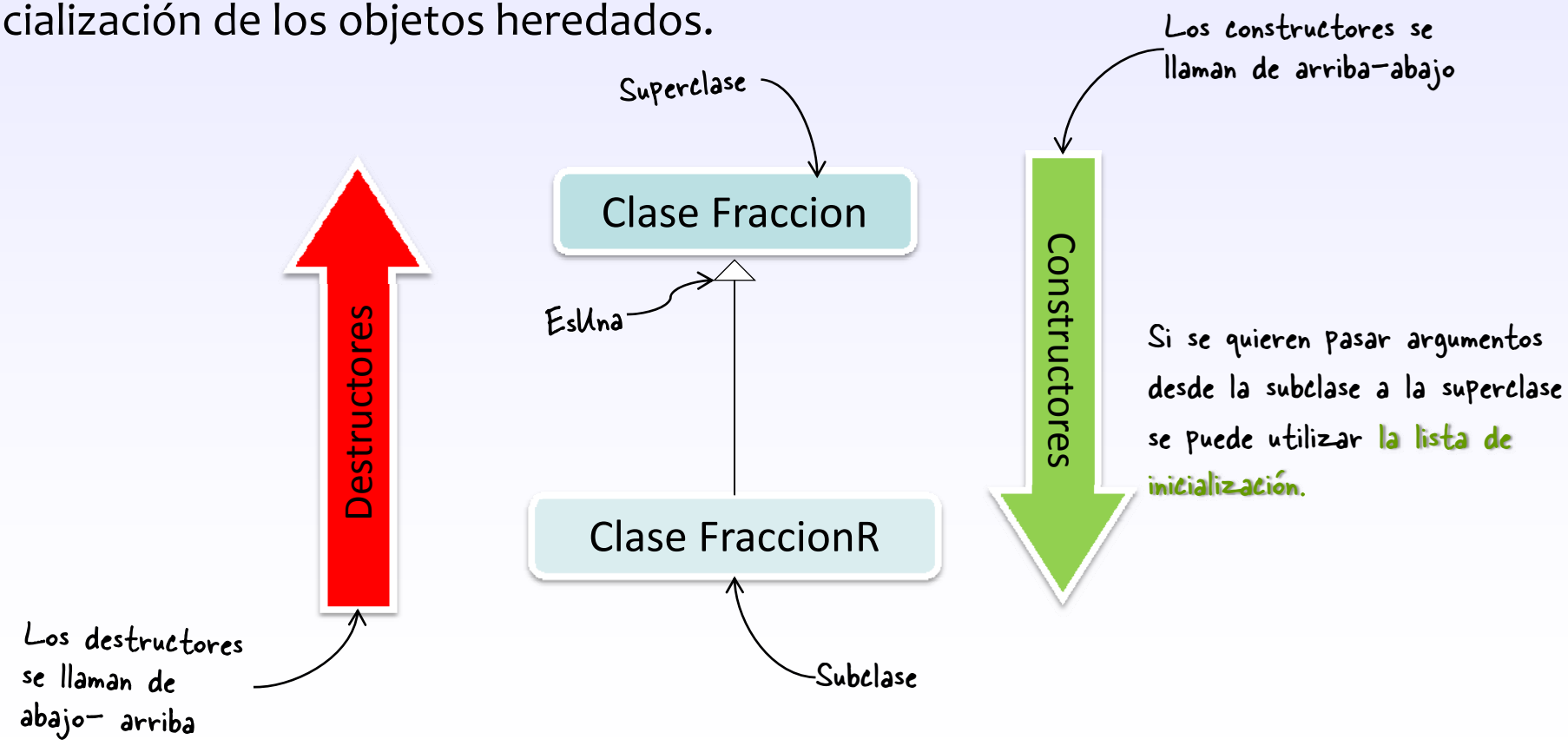
C++


Acceso public
Clase Derivada
public
protected
Clase Base
public
private
protected
Acceso portected
Clase Derivada
protected
protected
Clase Base
public
private
protected
Acceso private
Clase Derivada
private
private
Clase Base
public
private
protected

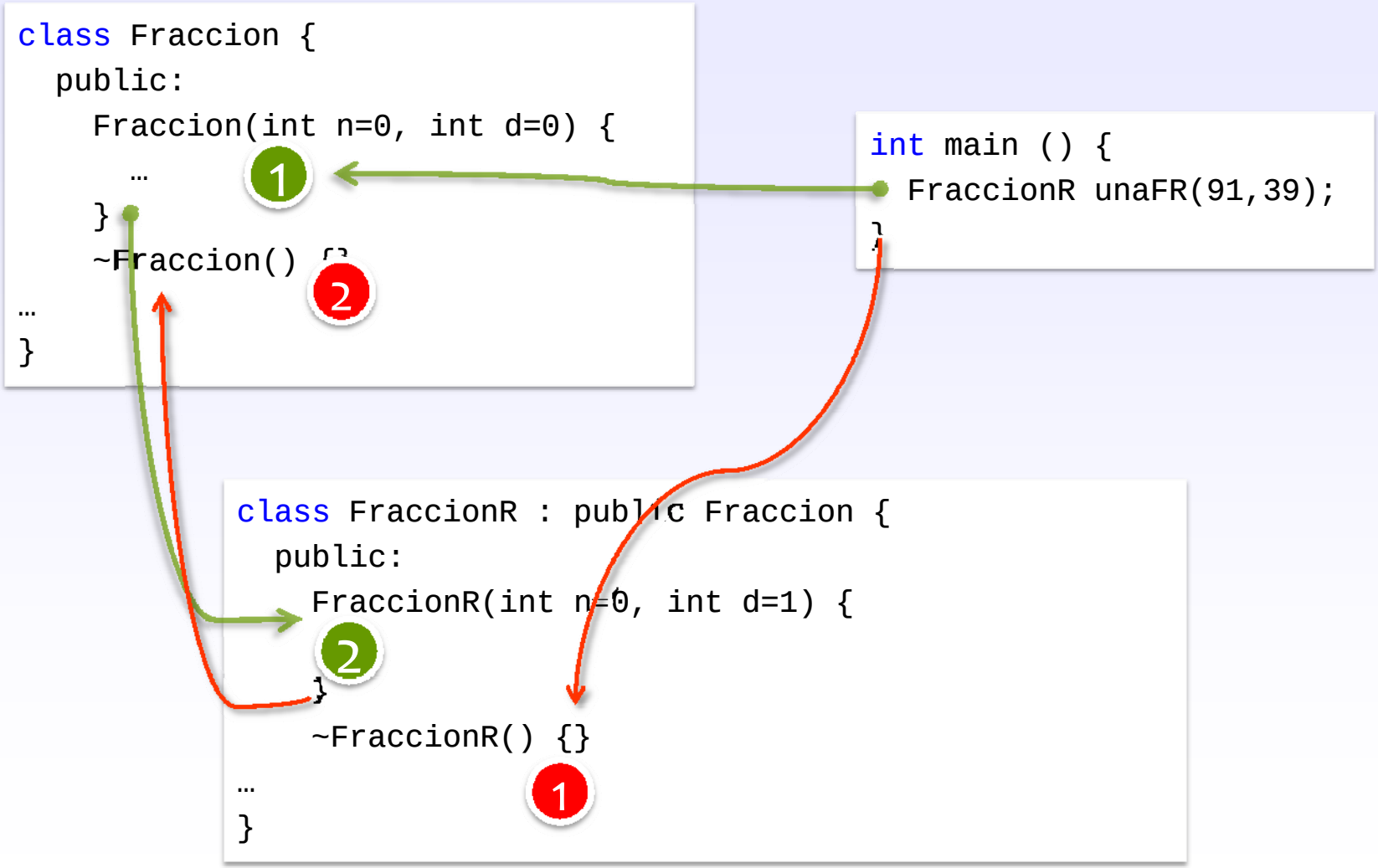
Herencia. Constructores. Destructor

Ni el destructor ni los constructores se pueden heredar

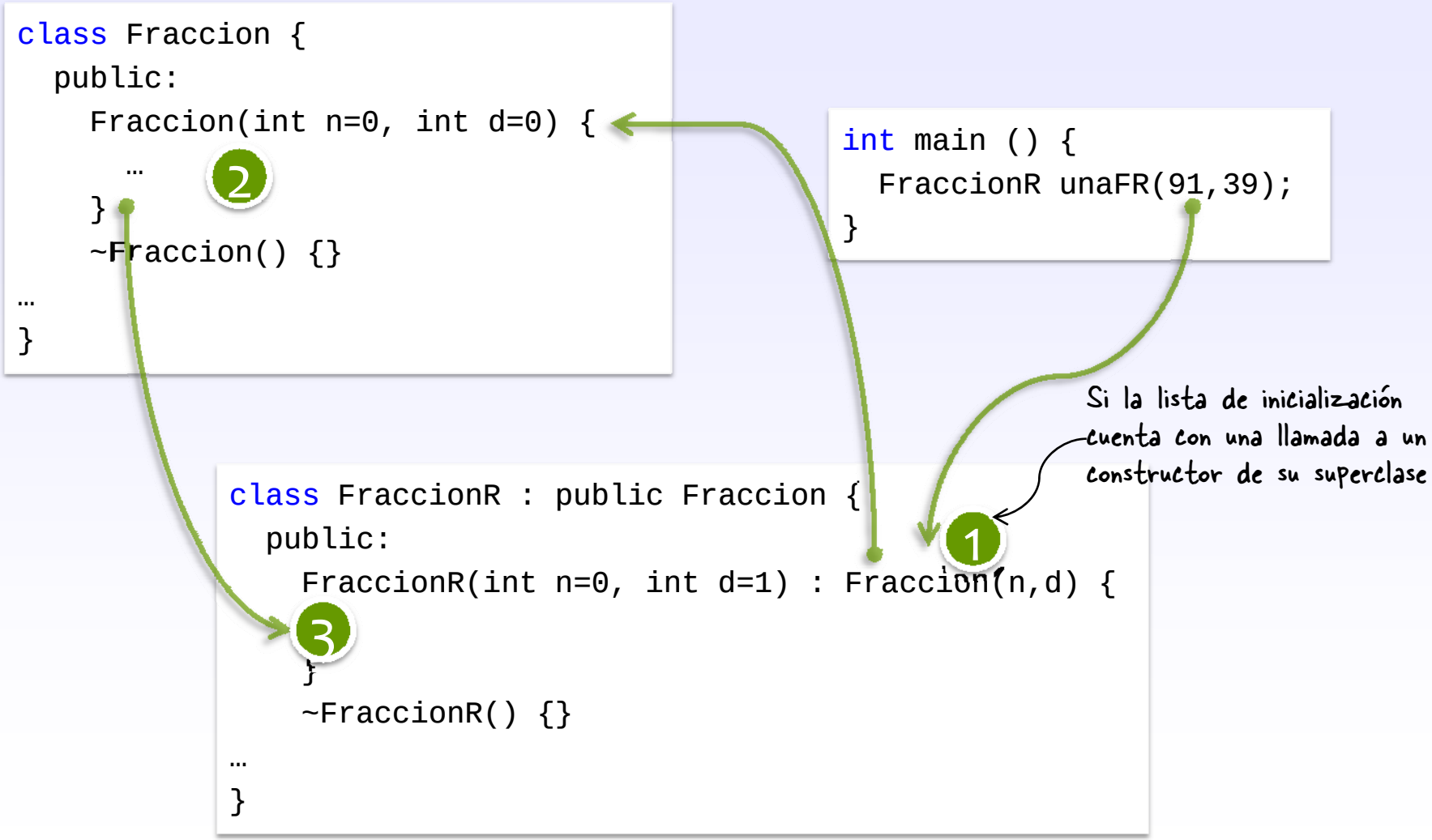
Aún así, tanto constructores como destructor tienen un papel importante en la inicialización de los objetos heredados.



Herencia. Constructores. Destructor



Herencia. Listas inicialización



Overriding (anular, ocultar)

Cuando una clase derivada define atributos con los mismos nombre de su clase base o métodos con el mismo nombre e idénticos atributos se dice que esta **overriding** las propiedades o métodos.

Lo que se produce es el ocultamiento de las propiedades o métodos con el mismo nombre. No se pueden acceder a la definición hecha en la clase base.

Si es necesario acceder a las propiedades o métodos ocultos se debe utilizar el nombre de la clase junto con el operador de ámbito y el nombre de la propiedad o método.

Enlazados

Los problemas relacionados con llamadas a métodos erróneos ocurren cuando:

- Un objeto de la clase derivada se usa como objeto de la clase base
- Un objeto de la clase derivada se utiliza vía referencias o puntero de la clase base.

Debido al que el compilador asume que es un objeto de la clase base, se ejecutan sus métodos incluso si se han *ocultado* en la clase derivada. Esto es así por que el compilador determina el objeto al que se accede y genera código que llama a sus métodos.

Enlazados

Enlazado estático

El objeto al que enlaza una referencia o puntero lo determina el compilador cuando genera el programa.

Enlazado dinámico

El objeto al que enlaza una referencia o puntero se determina en tiempo de ejecución. Se genera más código y es necesario marcar como virtuales aquellos métodos sobre los que se tiene que hacer el enlace dinámico.

Cuando utilicemos herencia debemos seguir las siguientes normas:

- No se deben heredar funciones no virtuales.
- Si son necesarias para la implementación de la clase derivada, no se debe derivar la clase.

Funciones virtuales

Para hacer que métodos con el mismo nombre de la subclase operen de forma diferentes a los de su superclase, se utilizan funciones virtuales.

La palabra clave `virtual` cambia el enlazado estático en dinámico. En tiempo de ejecución, se ejecuta código que determina de qué clase es el objeto.

Al utilizar funciones virtuales es posible *override* (ocultar) funciones de la clase base sin los problemas que ocurren cuando se utilizan punteros y referencias a la clase base.

Funciones virtuales

Una clase base apropiada para la herencia debe cumplir:

- Todas las funciones que se puedan ocultar deben declararse virtuales
- Se debe definir un destructor virtual

Cuando se implementa una clase derivada se debe tener en cuenta:

- Ocultar funciones no virtuales de la clase base puede conllevar problemas.
- Las funciones de la clase base se ocultan si coinciden en los parámetros y el tipo devuelto. Existe una excepción: si la devolución es una referencia o puntero a la clase base, se puede cambiar por una a la clase derivada.
- Funciones ocultas deben tener los mismos argumentos por defecto
- Por medio del operador de ámbito se pueden llamar a las funciones ocultadas.
- Sólo se tiene acceso a los miembros públicos de la clase base cuando ésta se pasa en un parámetro.

Polimorfismo

Propiedad de los cuerpos que pueden cambiar de forma sin variar su naturaleza

En informática

Hacer que un determinado código, tal y como está escrito, se comporte de forma distinta dependiendo de la situación.

C++ soporta polimorfismo por medio de

1. Miembros de clases diferentes tienen el mismo nombre.
2. La ocultación de métodos
3. Funciones virtuales

Polimorfismo

Clase abstracta

Clase que no se puede instanciar. Se pueden utilizar para combinar propiedades comunes de clases diferentes para que trabajen de la misma forma. Una clase que se pueda instanciar recibe el nombre de *clase concreta*

Funciones puras virtuales

Son funciones que se declaran pero no se implementan. Se deben implementar en la clase derivada. Pueden tener implementaciones por defecto.

```
virtual void dibujar() const = 0;
virtual void leer(int&) = 0;
```

Convierte la función virtual en virtual pura