UNIVERSIDADE DA CORUÑA

El lenguaje de programación C++

Fernando Bellas Permuy

Departamento de Tecnologías de la Información y las Comunicaciones (TIC)

Universidad de A Coruña

http://www.tic.udc.es/~fbellas

fbellas@udc.es

Introducción

- C++ extiende el lenguaje de programación C con conceptos de Orientación a Objetos.
- Es un lenguaje compilado.
- Índice:
  - Revisión del lenguaje C.
  - Mejoras (no OO) introducidas por C++.
  - Clases.
  - La herramienta “make” en Unix.
  - Herencia.
  - Sobrecarga de operadores.
  - Plantillas (*templates*).
  - Excepciones.
  - La librería estándar de C++.
  - Bibliografía.

Nociones básicas de C (1)

- Tipos de datos básicos.

Tipo	Indentificador	Ejemplo de valores	Modificadores
Caracteres	char	'a', '9', '#', 10	unsigned
Enteros	int	23, -34, 0	long, short, unsigned
Reales	float	3, 3.0, 3e10	
Reales (doble precisión)	double	3, 3.0, 3e600	long
“Booleans”	int	0 (false), != 0 (true)	

- Operadores aritméticos: =, +, -, %, /, ++, --, *y variantes de* =
- Variables: locales, globales.

```
#include <stdio.h>

float e; /* Variable global */

int main ()
{
    float v, t; /* Variables locales */

    v = 30; /* Velocidad */
    t = 5; /* Tiempo */

    e = v * t;
    printf("Velocidad: %f\nTiempo: %f\n", v, t);
    printf("Espacio recorrido: %f\n", e);

    return 0;
}
```

Nociones básicas de C (y 2)

- Operadores ++, --, y *variantes de* =

```
i = 4;
j = 2 * (i++);
/* i = 5, j = 8 */
```

```
i = 4;
j = 2 * (++i);
/* i=5, j = 10 */
```

```
i = 4;
i %= 3; /* i = i % 3; */
i += 4;  /* i = i + 4; */
/* i = 5 */
```

- Entrada salida con `printf/scanf` y similares.

```
#include <stdio.h>

int main ()
{
    float v, t, e;

    printf("Velocidad: ");
    scanf("%f", &v);
    printf("Tiempo: ");
    scanf("%f", &t);
    e = v * t;

    printf("Velocidad: %5.2f; Tiempo: %5.2f; Espacio: %5.2f\n",v,t,e);

    return 0;
}
```

  - Caracteres de control en `printf`, `scanf`: d, o, x, c, s, f, e, p.

- En C++ hay una alternativa mejor: los *streams* de entrada/salida.

Control de flujo (1)

- Operadores relacionales: **>, >=, <, <=, ==, !=**
- Operadores lógicos: &&, ||, !
- **if.. else..**

```
if (condición) {
    << sentencias >>
} else {
    << sentencias >>
}
```

- **switch-case-default**

```
switch (exp) {
    case A:
        << instrucciones >>
        break;
    case B:
        << instrucciones >>
        break;
    ...
    default:
        << instrucciones por defecto >>
}
```

```
switch (letra) {
    case 'a':
    case 'e':
    case 'i':
    case 'o':
    case 'u': printf("Es una vocal\n");
              break;
    case 'ñ': printf("Nuestra querida ñ\n");
              break;
    default: printf("Cualquier otro caracter");
}
```

Control de flujo (y 2)

- Bucle **while**

```
while (condición) {
   << instrucciones >>
}
```

- Bucle **do.. while**

```
do {
   << instrucciones >>
} while (condición)
```

- Bucle **for**

```
for (inic; cond; incr) {
   << instrucciones >>
}
```

```
for (i=1; i<=10; i++) {
    printf("%i\n", d);
}
```

Operadores de bit

- & (AND), | (OR), ~ (NOT), ^ (XOR), >> (*shift right*), << (*shift left*).

Estructuras de datos estáticas (1)

- Vectores.

```
int vector[] = {1, 2, 3, 4, 5};
int vector2[5] = {1, 2, 3, 4, 5};
int matriz[10][10];
char matriz3[4][5][6];

int main ()
{
    float matriz2[][3] = { {1, 2, 3}, {4, 5, 6} };

    int i, j;
    for (i=0; i<2; i++) {
        for (j=0; j<3; j++) {
             printf("%f\n", matriz2[i][j]);
        }
    }
    return 0;
}
```

- Registros o Estructuras.

```
struct nombre {
    Tipo1 campo1;
    Tipo2 campo2;
    ...
    TipoN campoN;
} variable;
```

```
struct TipoPunto {
    int x;
    int y;
} punto;
```

```
punto.x = 1;
punto.y = 2;
```

- Uniones.

```
union Ejemplo {
    char caracter;
    int entero;
};
```

```
struct FiguraColoreada {
    int color;
    int tipoFigura;
    union {
        struct Esfera e;
        struct Segmento s;
    } figura;
}
```

Estructuras de datos estáticas (y 2)

- Tipos definidos por el usuario (`typedef`).

```
typedef unsigned char TipoByte;

typedef struct {
    int x;
    int y;
} TipoPunto;
```

- Enumeraciones.

```
enum Dia { Lunes, Martes, Miercoles, Jueves, Viernes, Sabado, Domingo};
```

Punteros (1)

- Un **puntero** es una variable cuyo contenido es una *dirección* de memoria.
- Operadores: &, *, ->
- Ejemplos:

```
int x;
int* px;

x = 1;
px = &x;
*px = 2; /* x = 2 */
```

```
TipoPunto* pPunto;

pPunto->x = 1;
pPunto->y = 2;
```

- Punteros y matrices.

```
int i;
int v[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int* p;

for (i=0; i<10; i++) {
    printf("%d\n", v[i]);
}

for (i=0, p=v; i<10; i++, p++) {
    printf("%d\n", *p);
}

for (p=v; p != &(v[10]); p++) {
    printf("%d\n", *p);
}
```

Punteros (y 2)

- Cadenas de caracteres: Vectores de caracteres terminados en 0 ('\0').

```
#include <stdio.h>

char* mensaje = "hola";

int main ()
{
    char cadena[10];
    char* p = cadena;
    unsigned int longitud = 0;

    puts(mensaje);
    scanf("%9s", cadena);

    while (*p != 0) {
        longitud++;
        p++;
    }
    printf("Longitud: %d\n", longitud);
    return 0;
}
```

Funciones

- Un ejemplo.

```
int Suma (int x, int y)
{
    return x+y;
}
```

- Paso de parámetros.

```
void Swap (int* a, int* b)
{
    int temporal;

    temporal = *a;
    *a = *b;
    *b = temporal;
}

int main ()
{
    int x = 4;
    int y = 5;

    Swap(&x, &y);
    /* x = 5, y = 4 */
}
```

- La función `main()`

```
int main (int argc, char* argv[], char* env[])
{
    int i;

    for (i = 0; i < argc; i++) puts(argv[i]);
    for (i = 0; env[i]; i++) puts(env[i]);

    return 0;
}
```

Ejercicio

```
#ifndef _Pila_
#define _Pila_

/* Tipos. */

typedef enum {Pila_OK, Pila_Error}
    Pila_CodigoRetorno;

typedef struct Pila_Nodo {
    int fElemento;
    struct Pila_Nodo* fAnterior;
} Pila_Nodo;

typedef struct {
    Pila_Nodo* fCima;
} Pila;

/* Funciones. */

Pila* Pila_Crear ();

Pila_CodigoRetorno Pila_Introducir (Pila* pila,
    int elemento);

Pila_CodigoRetorno Pila_Sacar (Pila* pila,
    int* elemento);

int Pila_EsVacia (Pila* pila);

void Pila_Destruir (Pila* pila);

#endif
```

Mejoras (no OO) introducidas por C++ (1)

- Referencias.

```
int val = 10;
int& ref = val;
int& ref2; // Error !
```

```
void Swap (int& x, int& y) {
    int tmp = x;
    x = y;
    y = tmp;
}
int main () {
    int x = 5; int y = 7;
    Swap(x, y);

    return 0;
}
```

- Comentario de línea: `//`
- Constantes y declaraciones de variables.

```
const int LONGITUD = 4; // En C antiguo => #define LONGITUD 4

void f ()
{
    int x;

    x = 23;
    cout << x << endl;
    int y = 1;
    cout << y << endl;
}
```

- Tipos enumerados.

```
enum Color {rojo = 0, amarillo = 8, azul = 16};
```

Mejoras (no OO) introducidas por C++ (2)

- Sobrecarga de funciones (y operadores).

```
void print (int entero);
void print (const char* cadena);
void print (double real);

print(2);
print("hola");
print(1.0);
```

- Argumentos por defecto.

```
void print (int valor, int base=10);

void f()
{
    print(31);
    print(31, 10);
    print(31, 16);
}
```

- Asignación de memoria dinámica => operadores `new` y `delete`.

```
int* x = new int(2);
int* y = new int[10];

delete x;
delete []y;
```

- Los prototipos de función son obligatorios.
- Se puede hacer *inlining* de funciones.

```
inline void f (int i) { cout << i << endl; }
```

Mejoras (no OO) introducidas por C++ (y 3)

- Tipo `bool`.
  - Posibles valores `true` (1) y `false` (0).
- *Namespaces*.

```
// Fichero de especificación.

namespace Libreria {

    typedef enum {uno, dos, tres} Tipo;

    double Funcion (const char* str);

}
```

```
// Fichero de implementación.

double Libreria::Funcion (const char* str) { /* ... */ }
```

```
// Uso.

double d = Libreria::Funcion(unaCadena);
```

```
using namespace Libreria;
using namespace Libreria::Funcion;
```

Clases (1)

- Ejemplo:

```
class Fecha {
public:
    void Establecer (unsigned int dia, unsigned mes,
                     unsigned int anho);
    void Obtener (unsigned int& dia, unsigned int& mes,
                  unsigned int& anho) const;
    void Imprimir ();
private:
    unsigned int fDia, fMes, fAnho;
};
```

```
void Fecha::Obtener (unsigned int& dia, unsigned int& mes,
                     unsigned int& anho)
{
    dia = fDia; mes = fMes; anho = fAnho;
}
```

- Especificadores de acceso: `private` (por defecto), `public` y `protected`.
- Constructores: constructor por defecto, constructor copia y constructores adicionales.

```
class Fecha {
// ...
public:
    Fecha (unsigned int dia, unsigned int mes,
       unsigned int anho);
    Fecha (const char* cadena);
    Fecha (); // Constructor por defecto.
    Fecha (const Fecha& fecha); // Constructor copia.
};
```

Clases (2)

- ... continuación del ejemplo.

```
Fecha miCumple(19, 4, 1998);
Fecha miCumple2("19 Abril 1998");
Fecha miCumple3; // Constructor por defecto.
Fecha* miCumple4 = new Fecha(19, 4, 1998);
Fecha miCumple5 = miCumple2; // Inicialización (c. copia).
Fecha miCumple6;
miCumple6 = miCumple5; // Es una asignación.
```

```
void f (Fecha fecha);

f(miCumple);


void f(const Fecha& fecha);

f(miCumple);
```

- Destructor: `~NombreDeLaClase()`

```
class X {
    public:
    // ...
    ~X ();
};
```

```
{
    X x;
    X* x2 = new X;
    X* x3 = new X[10];

    delete x2;
    delete []x3;
}
```

Clases (3)

- Puntero `this`.

```
class X {
public:
    // ...
    void Metodo (int i) { this-> i = i; }
private:
    int i;
};
```

- Miembros `static`.

```
#include <iostream>

using namespace std;

class Ejemplo {
public:
    // ...
    Ejemplo () { cuenta++; }
    static int Cuantos () { return cuenta; }
private:
    static int cuenta;
    // ...
};

int Ejemplo::cuenta = 0;

int main ()
{
    Ejemplo e1, e2;
    cout << Ejemplo::Cuantos() << endl; // 2

    return 0;
}
```

Clases (y 4)

- **Clases utilidad** (*utility classes*): todos los métodos/atributos son `static`.

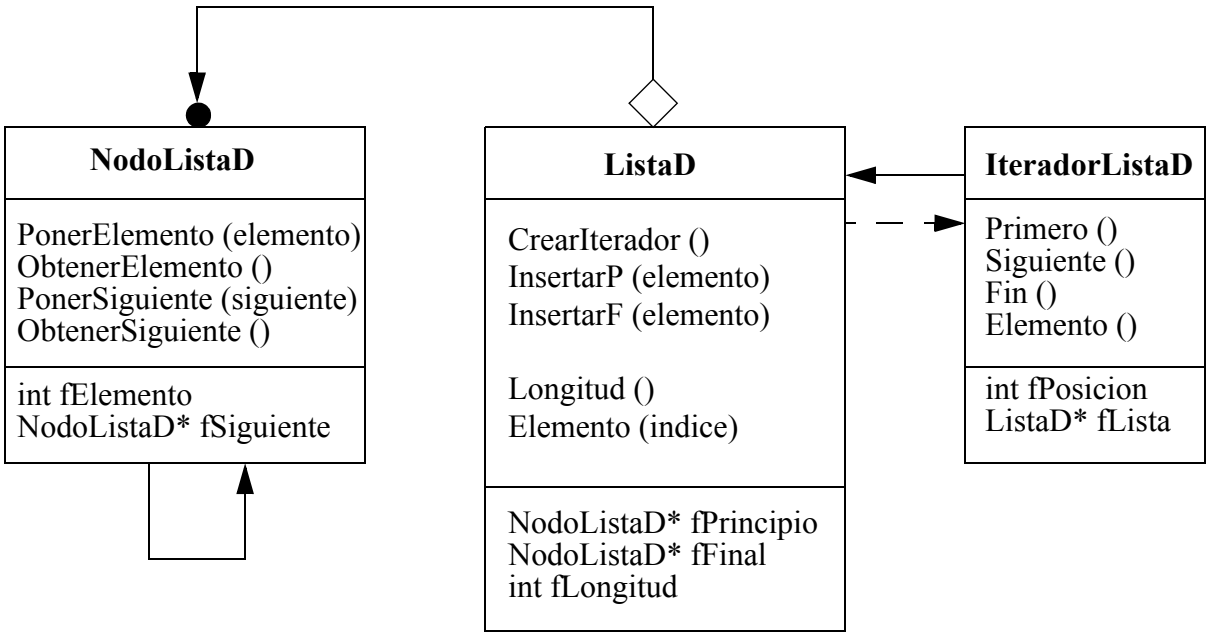
```
class LibreriaMatematica {
public:
    static float Seno (float angulo);
    static float Coseno (float angulo);
    static float Tangente (float angulo);
    // ...
};


float resultado = LibreriaMatematica::Seno(12.1);
```

  - Las clases utilidad evitan el uso de funciones globales.
- Especificador `const` en métodos.
- Funciones, métodos y clases amigas.

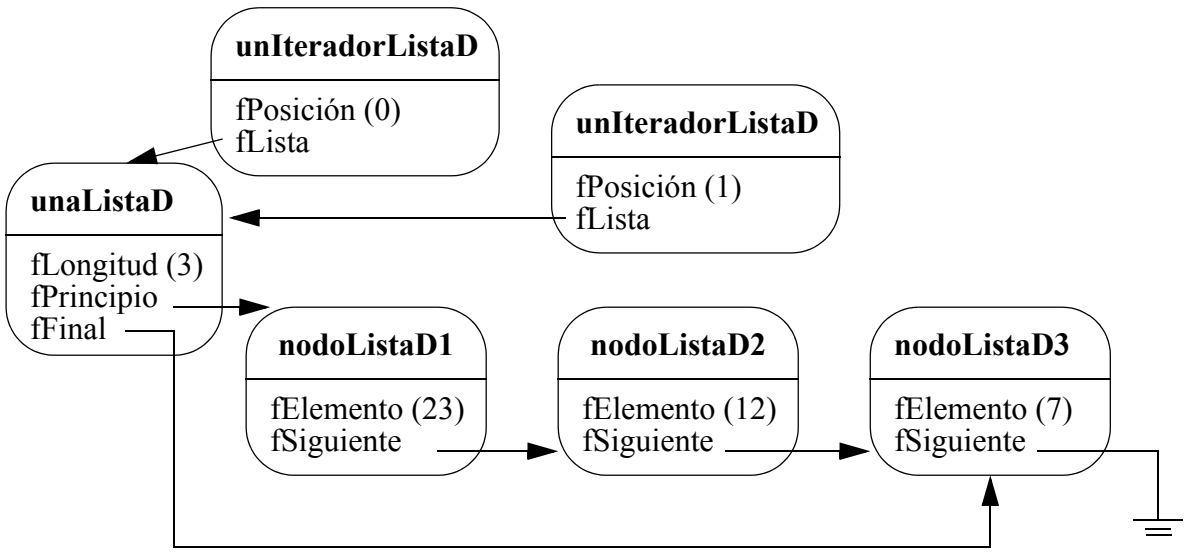
```
class Ejemplo {
// ...
private:
    int i;
    friend void FuncionAmiga (Ejemplo& e);
    friend void X::MetodoAmigo (Ejemplo& e);
    friend Y;
};

void FuncionAmiga (Ejemplo& e)
{
    cout << e.i << endl;
}
```

Ejercicio (1)

- Patrón de diseño (*design pattern*) “Iterador” (*Iterator*).
  - Diagrama de clases.



  - Diagrama de objetos.



Ejercicio (2)

```
#ifndef _ListaD_
#define _ListaD_

#include "NodoListaD.h"

class IteradorListaD;

class ListaD {
public:
    ListaD ();
    ListaD (const ListaD& lista);
    ~ListaD ();
    const ListaD& operator = (const ListaD& lista);
public:
    IteradorListaD* CrearIterador () const;
    void InsertarP (int elemento);
    void InsertarF (int elemento);
private:
    int Longitud () const;
    int Elemento (int indice) const;
    friend IteradorListaD;
private:
    void Destruir ();
    void CopiarDesde (const ListaD& lista);
    void Inicializar ();
private:
    NodoListaD* fPrincipio;
    NodoListaD* fFinal;
    int fLongitud;
};

#endif
```

Ejercicio (3)

```
#ifndef _IteradorListaD_
#define _IteradorListaD_

#include "ListaD.h"

class IteradorListaD {
public:
    IteradorListaD (ListaD* lista);
    ~IteradorListaD ();
public:
    void Primero();
    void Siguiente();
    int Fin () const;
    int Elemento () const;
private:
    IteradorListaD (const IteradorListaD& iterador);
    const IteradorListaD& operator= (
        const IteradorListaD& iterador);
private:
    int fPosicion;
    ListaD* fLista;
};

#endif
```

Ejercicio (4)

```
#ifndef _NodoListaD_
#define _NodoListaD_

class NodoListaD {
public:
    NodoListaD (int elemento, NodoListaD* siguiente);
    ~NodoListaD ();
    void PonerElemento (int elemento);
    int ObtenerElemento () const;
    void PonerSiguiente (NodoListaD* siguiente);
    NodoListaD* ObtenerSiguiente () const;
private:
    NodoListaD (const NodoListaD& nodoListaD);
    const NodoListaD& operator= (
        const NodoListaD& nodoListaD);
private:
    int fElemento;
    NodoListaD* fSiguiente;
};

#endif
```

Ejercicio (y 5)

```
class ClasePrueba {
public:
    void InsertarElementos (ListaD& lista);
    void Listar (IteradorListaD& iterador);
};


void ClasePrueba::InsertarElementos (ListaD& lista)
{
    for (int i = 0; i<10; i++) {
        lista.InsertarF(i);
    }
}


void ClasePrueba::Listar (IteradorListaD& iterador)
{
    iterador.Primero();
    cout << "Lista: " << endl;
    while (!iterador.Fin()) {
        cout << iterador.Elemento() << endl;
        iterador.Siguiente();
    }
}


int main ()
{
    ClasePrueba prueba;
    ListaD lista;

    prueba.InsertarElementos(lista);
    IteradorListaD* iterador = lista.CrearIterador();
    prueba.Listar(*iterador);

    delete iterador;

    return 0;
}
```

La herramienta “make” en Unix (1)

- `make` es una herramienta que permite expresar dependencias temporales entre ficheros mediante reglas.
- Cada regla puede llevar asociada una acción, que se ejecutará si se cumple la regla.
- Las reglas se expresan en un fichero (`Makefile`).
- La utilidad más importante de `make` es la compilación de aplicaciones (ej.: C, C++, etc.).
- Existen versiones para sistemas operativos distintos a Unix.
- **No** es una herramienta estándar.
- Existen herramientas que generan Makefiles automáticamente a partir de un conjunto de ficheros C/C++ (por ejemplo).
- Invocación
  - Si el fichero se llama `Makefile` => `make`
  - En otro caso => `make -f NombreFicheroMakefile`

La herramienta “make” en Unix (2)

- Un primer Makefile (`Makefile1`) ...

```
Prueba: NodoListaD.o IteradorListaD.o ListaD.o Prueba.o
    g++ -o Prueba NodoListaD.o IteradorListaD.o \
        ListaD.o Prueba.o

NodoListaD.o: NodoListaD.cpp NodoListaD.h
    g++ -c NodoListaD.cpp

IteradorListaD.o: IteradorListaD.cpp IteradorListaD.h \
    ListaD.h
    g++ -c IteradorListaD.cpp

ListaD.o: ListaD.cpp ListaD.h NodoListaD.h \
    IteradorListaD.h
    g++ -c ListaD.cpp

Prueba.o: Prueba.cpp ListaD.h IteradorListaD.h
    g++ -c Prueba.cpp

clean:
    rm -rf  NodoListaD.o IteradorListaD.o ListaD.o \
    Prueba.o Prueba *~
```

- A recordar:
  - Reglas separadas al menos por una línea en blanco.
  - La acción empieza con un tabulador.
- Invocaciones ...
  - `make -f Makefile1`
  - `make -f Makefile1 clean`

La herramienta “make” en Unix (3)

- Una primera mejora (variables) ...

```
# Variables.

OBJS = NodoListaD.o IteradorListaD.o ListaD.o Prueba.o

# Reglas.

Prueba: $(OBJS)
    g++ -o Prueba $(OBJS)

NodoListaD.o: NodoListaD.cpp NodoListaD.h
    g++ -c NodoListaD.cpp

IteradorListaD.o: IteradorListaD.cpp IteradorListaD.h \
    ListaD.h
    g++ -c IteradorListaD.cpp

ListaD.o: ListaD.cpp ListaD.h NodoListaD.h \
    IteradorListaD.h
    g++ -c ListaD.cpp

Prueba.o : Prueba.cpp ListaD.h IteradorListaD.h
    g++ -c Prueba.cpp

clean:
    rm -rf  $(OBJS) Prueba *~
```

La herramienta “make” en Unix (4)

- Otra mejora más (reglas implícitas) ...

```
# Variables.

OBJS = NodoListaD.o IteradorListaD.o ListaD.o Prueba.o

# Reglas implicitas.

%.o: %.cpp
    g++ -c $<

#.cpp.o:
#    g++ -c $<

# Reglas.

Prueba: $(OBJS)
    g++ -o Prueba $(OBJS)

NodoListaD.o: NodoListaD.h

IteradorListaD.o: IteradorListaD.h ListaD.h

ListaD.o: ListaD.h NodoListaD.h IteradorListaD.h

Prueba.o: ListaD.h IteradorListaD.h

clean:
    rm -rf  $(OBJS) Prueba *~
```

La herramienta “make” en Unix (5)

- Problemas de los anteriores Makefiles:
  - Difíciles de mantener.
  - Demasiadas recompilaciones (un cambio en un comentario en un fichero cabecera ...).
- Relajando las dependencias ...

```
# Variables.

OBJS = NodoListaD.o IteradorListaD.o ListaD.o Prueba.o

# Reglas implicitas.

%.o: %.cpp
    g++ -c $<

# Reglas.

Prueba: $(OBJS)
    g++ -o Prueba $(OBJS)

clean:
    rm -rf  $(OBJS) Prueba *~
```

- El anterior Makefile puede tener algunos problemas de inconsistencias, pero en general es un opción aconsejable.

La herramienta “make” en Unix (6)

- Cuando un proyecto está estructurado en varios directorios ...
  - En el directorio `Ejemplo3`, el Makefile tiene el siguiente aspecto.

```
include ../EjemplosC++.incl

# OBJS

OBJS = NodoListaD.o ListaD.o IteradorListaD.o Prueba.o

Prueba: $(OBJS)
    $(COMPILADOR) -o Prueba $(OBJS)

clean:
    rm -rf $(OBJS) Prueba *~
```

  - En el directorio padre => `EjemplosC++.incl` =>

```
# Compilador.
COMPILADOR = g++

# Compilación.
%.o: %.cpp
    $(COMPILADOR) -c $<
```

La herramienta “make” en Unix (y 7)

- Cuando un proyecto está estructurado en varios directorios ... (continuación)
  - En el directorio padre existe un Makefile que recompila todo el software ...

```
all:
    for i in Ejemplo*; \
    do if [ -d $$i ]; then cd $$i; make; cd ..; fi; \
    done

clean:
    rm -f *~
    for i in Ejemplo*; \
    do if [ -d $$i ]; then cd $$i; make clean; cd ..; \
    fi; done
```

Herencia (1)

- Un ejemplo:

```
#include <iostream>
using namespace std;
class Persona {
public:
    void Saluda () { cout << "Hola" << endl; }
};

class PersonaEducada : public Persona {
public:
    void BuenosDias () { cout << "Buenos dias" << endl; }
};

int main () {
    Persona a;
    PersonaEducada b;
    a.Saluda();
    b.Saluda();
    b.BuenosDias();
    return 0;
}
```

- Tres tipos de herencia: `public`, `protected` y `private`.

```
class X {
public:
    int a;
protected:
    int b;
private:
    int c;
};

class X2 : public X {
    void f() {
        cout << a; // OK, pública.
        cout << b; // OK, protegida.
        cout << c; // Error, privada en X.
    };
};
```

Herencia (2)

- Constructores.

```
#include <iostream>

using namespace std;

class X {
public:
    X (int i) { cout << "Constructor X: " << i << endl; }
    ~X () { cout << "Destructor X" << endl; }
};

class Y : public X {
public:
    Y (int i) : X(1), x(123) {
        cout << "Constructor Y: " << i << endl; }
    ~Y () { cout << "Destructor Y" << endl; }
private:
    X x;
};

int main ()
{
    Y y(1);

    return 0;
}
```

- Orden de llamada de constructores: primero los de las clases base (de arriba hacia abajo), y luego los de los objetos miembro.
- Orden de llamada de los destructores: a la inversa que los constructores.

Herencia (3)

- Redefinición de métodos.

```
#include <iostream>

using namespace std;

class Persona {
public:
    void Habla () { cout << "Hace un día precioso" << endl; }
};

class Futbolero : public Persona {
public:
    void Habla () { cout << "¿ A qué hora es el partido ?"
                      << endl; }
};

class Pesado : public Persona {
public:
    void Habla () {
        Persona::Habla();
        cout << "Recuerdo una vez que bla, bla, bla ..."
             << endl;
    }
};

int main ()
{
    Persona persona;
    Futbolero futbolero;
    Pesado pesado;

    persona.Habla();   // Hace un día precioso
    futbolero.Habla(); // ¿ A qué hora es el partido ?
    pesado.Habla();    // Hace un día precioso
                       // Recuerdo una vez que bla, bla,
                       // bla ...

    return 0;
}
```

Herencia (4)

- Métodos virtuales.

```
#include <iostream>

using namespace std;

class Vehiculo {
public:
    void Habla () { cout << "Soy un " << Identificacion()
                         << endl; }
    const char* Identificacion () { return "vehículo"; }
};

class Coche : public Vehiculo {
public:
    const char* Identificacion () { return "coche"; }
};

class Barco : public Vehiculo {
public:
    const char* Identificacion () { return "barco"; }
};

int main ()
{
    Vehiculo vehiculo;
    Coche coche;
    Barco  barco;

    vehiculo.Habla(); // Soy un vehículo
    coche.Habla(); // Soy un vehículo
    barco.Habla(); // Soy un vehículo

    return 0;
}
```

- El problema anterior se soluciona declarando `Identificacion()` como `virtual`.

Herencia (5)

- ... es decir ...

```
class Vehiculo {
public:
    void Habla () { cout << "Soy un " << Identificacion()
                          << endl; }
     virtual const char* Identificacion () {
         return "vehiculo"; }
};
```

- ¿ y qué ocurría con ?

```
Vehiculo vehiculo;
Coche coche;

vehiculo = coche;
vehiculo.Habla();
```

- Se produce polimorfismo cuando se accede a la función virtual con un objeto puntero (o referencia) de la clase base.

```
Vehiculo* vehiculo = new Vehiculo;
Coche* coche = new Coche;
Barco* barco = new Barco;

vehiculo->Habla(); // Soy un vehículo
coche->Habla();    // Soy un coche
barco->Habla();    // Soy un barco
Vehiculo* vehiculo2 = barco;
vehiculo2->Habla(); // Soy un barco
```

Herencia (6)

- ... continuación del ejemplo.

```
void f (const Vehiculo& vehiculo)
{
    vehiculo.Habla();
}

int main ()
{
    Coche coche;

    f(coche); // Soy un coche
}
```

- Clases abstractas: **no** se pueden tener instancias de clases abstractas. Su objetivo es definir una *interfaz*.

```
class Figura { // Clase abstracta
public:
    virtual void Dibujar () = 0; // Virtual pura
    virtual float Area () = 0; // Virtual pura
// ...
};

class Rectangulo : public Figura {
public:
    virtual void Dibujar () { // ... }
    virtual float Area () { // ... }
// ...
};

class ListaDeFiguras {
public:
    void Insertar(Figura* figura);
    void Dibujar();
// ...
};
```

Herencia (y 7)

- Herencia múltiple.

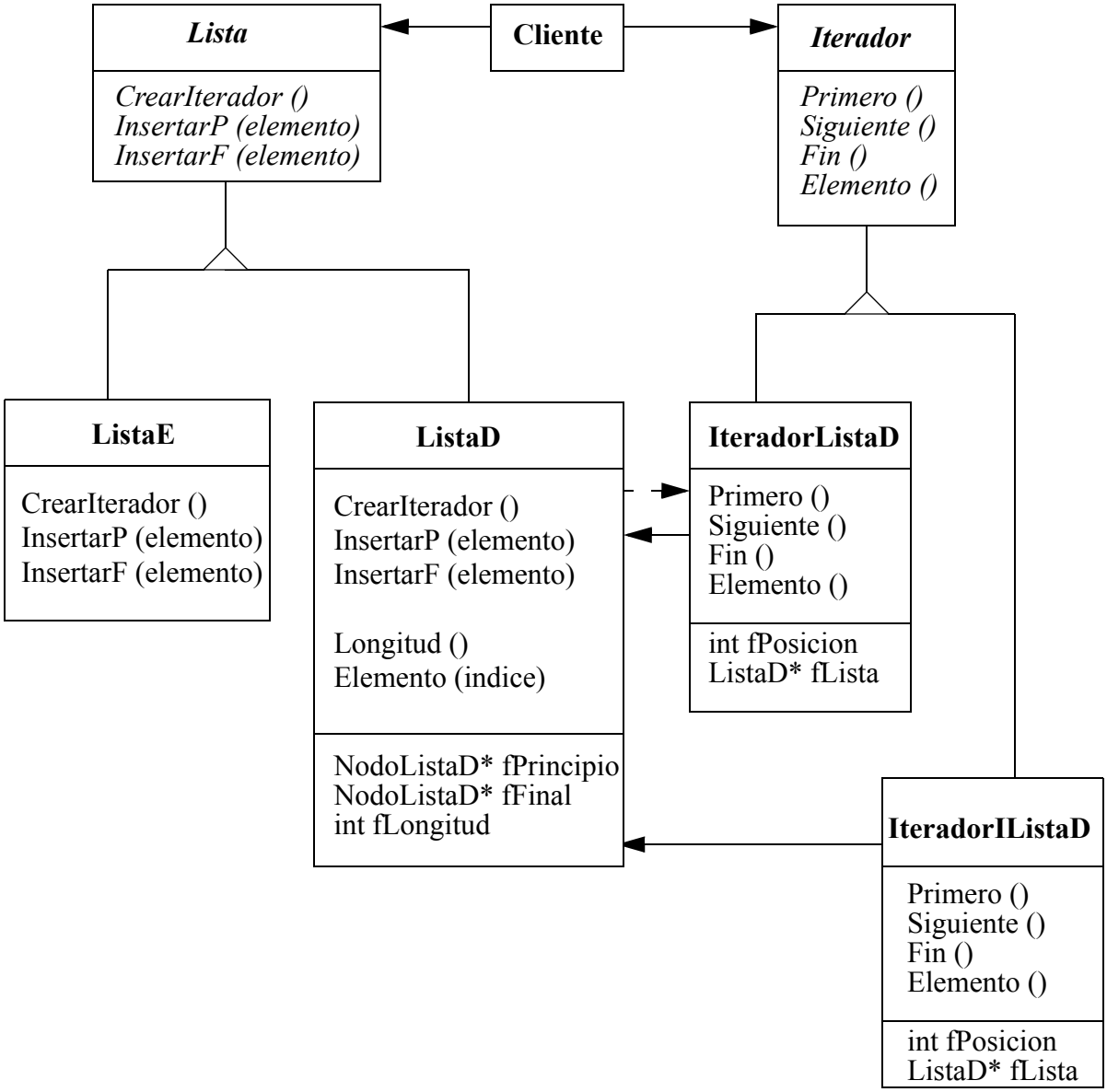
```
class Vehiculo { ...};

class VehiculoTerrestre : public Vehiculo { ... };

class VehiculoMaritimo : public Vehiculo { ... };

class VehiculoAnfibio : public VehiculoTerrestre,
                        public VehiculoMaritimo
{ ... };
```

- Problemas de ambigüedad con la herencia múltiple.

```
class X {
public:
    void f ();
};

class Y {
public:
    void f ();
};

class Z : public X, public Y {
public:
    // ...
};

int main ()
{
    Z z;

    z.f(); // Ambigüedad !
    z.X::f();
    return 0;
}
```

Ejercicio (1)

- Patrón de diseño (*design pattern*): “Iterador” (*Iterator*).



Ejercicio (2)

```
#ifndef _Lista_
#define _Lista_

class Iterador;

class Lista {
public:
    virtual ~Lista();
public:
    virtual Iterador* CrearIterador () const = 0;
    virtual void InsertarP (int elemento) = 0;
    virtual void InsertarF (int elemento) = 0;
};

#endif
```

```
#ifndef _Iterador_
#define _Iterador_

class Iterador {
public:
    virtual ~Iterador ();
public:
    virtual void Primero () = 0;
    virtual void Siguiente () = 0;
    virtual int Fin () const = 0;
    virtual int Elemento () const = 0;
};

#endif
```

Ejercicio (3)

```
#ifndef _ListaD_
#define _ListaD_

#include "NodoListaD.h"
#include "Lista.h"

class IteradorListaD;
class IteradorIListaD;

class ListaD : public Lista {
public:
    ListaD ();
    ListaD (const ListaD& lista);
    virtual ~ListaD ();
    const ListaD& operator = (const ListaD& lista);
public:
    virtual Iterador* CrearIterador () const;
    virtual void InsertarP (int elemento);
    virtual void InsertarF (int Elemento);
private:
    int Longitud () const;
    int Elemento (int indice) const;
    friend IteradorListaD;
    friend IteradorIListaD;
private:
    void Destruir ();
    void CopiarDesde (const ListaD& lista);
    void Inicializar ();
private:
    NodoListaD* fPrincipio;
    NodoListaD* fFinal;
    int fLongitud;
};

#endif
```

Ejercicio (4)

```
#ifndef _IteradorListaD_
#define _IteradorListaD_

#include "ListaD.h"
#include "Iterador.h"

class IteradorListaD : public Iterador {
public:
    IteradorListaD (ListaD* lista);
    virtual ~IteradorListaD ();
public:
    virtual void Primero();
    virtual void Siguiente();
    virtual int Fin () const;
    virtual int Elemento () const;
private:
    IteradorListaD (const IteradorListaD& iterador);
    const IteradorListaD& operator= (
        const IteradorListaD& iterador);
private:
    int fPosicion;
    ListaD* fLista;
};

#endif
```

Ejercicio (y 5)

```
class ClasePrueba {
public:
    void InsertarElementos (Lista& lista);
    void Listar (Iterador& iterador);
};


void ClasePrueba::InsertarElementos (Lista& lista)
{
    for (int i = 0; i<10; i++) {
        lista.InsertarF(i);
    }
}


void ClasePrueba::Listar (Iterador& iterador)
{
    iterador.Primero();
    cout << "Lista: " << endl;
    while (!iterador.Fin()) {
        cout << iterador.Elemento() << endl;
        iterador.Siguiente();
    }
}

int main ()
{
    ClasePrueba prueba;
    ListaD lista;

    prueba.InsertarElementos(lista);
    Iterador* iterador1 = lista.CrearIterador();
    prueba.Listar(*iterador1);

    IteradorIListaD iterador2(&lista);
    prueba.Listar(iterador2);

    delete iterador1;

    return 0;
}
```

Sobrecarga de operadores (1)

- Operadores que se pueden sobrecargar: +, -, *, /, %, ^, &, |, ~, =, <, >, +=, -=, *=, /=, %=, ^=, &=, |=, <<, >>, <<=, >>=, ==, !=, <=, >=, &&, ||, ++, --, ->, [], new, delete.
- Ejemplo:

```
class Complejo {
public:
    Complejo (double real, double imag) {
        fReal = real; fImag = imag;
    }

    Complejo operator+ (const Complejo& c) {
        return Complejo(fReal+c.fReal, fImag+c.fImag);
    }

    Complejo operator++ () {
        return Complejo(++fReal, ++fImag);
    }

    Complejo operator++ (int) {
        return Complejo(fReal++, fImag++);
    }

    const Complejo& operator= (const Complejo& c) {
        if (this != &c) {
            fReal = c.fReal;
            fImag = c.fImag;
        }
        return *this;
    }

    void Imprimir () {
        cout << fReal << " " << fImag << endl;
    }

private:
    double fReal, fImag;
};
```

Sobrecarga de operadores (2)

- ... continuación del ejemplo.

```
Complejo c1(1, 1);
Complejo c2(2, 2);
Complejo c3(3, 3);
Complejo c4(5, 5);

c4 = c1 + ++c2 + c3++;
c4.Imprimir(); // fReal: 7; fImag: 7
c1 = c2 = c4;
c1.Imprimir(); // fReal: 7; fImag: 7
c1 = c2.operator+(c3);
c1.Imprimir(); // fReal: 11; fImag: 11
c1 = c2.operator++();
c1.Imprimir(); // fReal: 7; fImag: 7
c1 = c2.operator++(123);
c1.Imprimir(); // fReal: 131; fImag: 131
```

- Otra forma de sobrecargar operadores: funciones globales.

```
class Complejo {
public:
    double DameReal () const { return fReal; }
    double DameImag () const { return fImag; }
// ...

private:
    double fReal, fImag;
};
```

Sobrecarga de operadores (3)

- ... continuación del ejemplo.

```
Complejo operator+ (const Complejo& c1, const Complejo& c2)
{
    return Complejo(c1.DameReal() + c2.DameReal(),
        c1.DameImag() + c2.DameImag());
}

Complejo operator++ (const Complejo& c)
{
    double real = c1.DameReal();
    double imag = c1.DameImag();
    return Complejo(++real, ++imag);
}
```

- Otra alternativa habría sido el uso de *friends*.
- Problema:

```
Complejo operator+ (Complejo& c, double d)
{
    return Complejo(c.DameReal()+d, c.DameImag());
}

// ...

c3 = c3 + 1;
c3 = 1 + c3; // Error !
```

- Solución => Conversión de tipos + funciones globales.

```
class Complejo {
public:
    Complejo (double real, double imag=0) {
        fReal = real; fImag = imag; }
// ...
};
```

Sobrecarga de operadores (y 4)

- Operador de conversión.

```
class X {
public:
    // ...
    operator int () { return i;}
private:
    int i;
};

// ...

X x(4);
int i;

i = x;
```

Ejercicio

```
#ifndef _Vector_
#define _Vector_

#include <iostream>

using namespace std;

class Vector {
public:
    enum {kTamPorDefecto=10};
    typedef int Elemento;
public:
    Vector (unsigned int tamanho=Vector::kTamPorDefecto);
    Vector (const Vector& vector);
    ~Vector ();
    const Vector& operator= (const Vector& vector);
public:
    Elemento& operator[] (unsigned int indice) const;
    Vector operator+ (const Vector& vector) const;
    Vector operator- (const Vector& vector) const;
    Vector operator* (const Vector& vector) const;
    Vector operator/ (const Vector& vector) const;
    void Imprimir (ostream& salida) const;
    unsigned int Tamanho () const;
private:
    void CopiarDesde (const Vector& vector);
    void Destruir ();
private:
    unsigned int fTamanho;
    Elemento* fDatos;
};

#endif
```

Plantillas (1)

- Definir una función que nos dé el mayor de dos números.

```
int DameElMayor (int x, int y)
{
    if (x > y) {
        return x;
    } else {
        return y;
    }
}
```

- Problema: ¿ y si queremos hacer lo mismo para otros tipos de datos (inclusive clases definidas por nosotros) ?
- Solución: *templates*.

```
template <class Tipo>
Tipo DameElMayor (const Tipo& x, const Tipo& y)
{
    if (x > y) {
        return x;
    } else {
        return y;
    }
}
```

- Ahora es posible hacer ...

```
double d1, d2, d3;

// ...

d3 = DameElMayor(d1, d2);
```

- Si se define una función **no** *template* con el mismo prototipo que otra función *template*, tiene más prioridad la primera.

Plantillas (y 2)

- Las plantillas también se pueden aplicar a clases C++.

```
template <class Elemento>
class Pila {
public:
    // ...
    void Insertar (const Elemento& elemento);
    unsigned int Longitud () const;
private:
    NodoPila<Elemento>* pila;
    // ...
}
```

```
template <class Elemento>
void Pila<Elemento>::Insertar (const Elemento& elemento)
{
    // ...
}

template <class Elemento>
unsigned int Pila<Elemento>::Longitud () const
{
    // ...
}
```

```
class Pila<int> pila;
```

- Si se define una clase **no** *template* con la misma especificación que otra clase *template*, tiene más prioridad la primera.

Excepciones (1)

- Una excepción es una anomalía que sucede en un programa en tiempo de ejecución.
- Idea general.

```
void FuncionA () throw (char*, ErrorRango, ErrorMemoria)
{
    // ...
    try {
        // ...
        FuncionB();
        // ...
    } catch (const char* cadena) {
        // ...
    } catch (const ErrorRango& err) {
        // ...
    } catch (const ErrorMemoria& err) {
        // ...
    } catch (...) {
        throw;
    }

}
```

```
void FuncionB () throw (char*, ErrorRango, ErrorMemoria)
{
    // ...
    if (error) {
        throw ("No hay suficiente memoria");
    }
    // ...
}
```

Excepciones (2)

- Ejemplo:

```
class ExcepcionPila {
public:
    enum Subcategoria { InsuficienteMemoria, EstaVacia,
                        _numeroDeSubCategorias};
public:
    ExcepcionPila (Subcategoria s);
    void Imprimir ();

private:
    Subcategoria fSubcategoria;
    static const char* fMensajes[_numeroDeSubCategorias];
};
```

```
const char* ExcepcionPila::fMensajes[
    ExcepcionPila::_numeroDeSubCategorias] = {
    "Insuficiente memoria", "Pila vacía"
};

ExcepcionPila::ExcepcionPila (Subcategoria s)
{
    fSubcategoria = s;
}

void ExcepcionPila::Imprimir ()
{
    cerr << fMensajes[fSubcategoria] << endl;
}
```

Excepciones (y 3)

- ... continuación del ejemplo.

```
template <class Elemento>
void Pila<Elemento>::Insertar (const Elemento& elemento)
    throw (ExcepcionPila)
{
    // ...
    if (noHayMemoria) {
        throw ExcepcionPila(
            ExcepcionPila::InsuficienteMemoria);
    }
    // ...
}
```

```
try {
    pila.Insertar(elemento);
} catch (const ExcepcionPila& e) {
    e.Imprimir();
} catch (...) {
    cerr << "Excepción desconocida" << endl;
}
```

- Si la excepción se captura, todos los objetos construidos en la pila, se liberan automáticamente. Por tanto, es siempre recomendable hacer uso del mecanismo de excepciones.
- Normalmente se define una jerarquía de excepciones.

Ejercicio (1)

```
#ifndef _ExcepcionLibreria_
#define _ExcepcionLibreria_

class ExcepcionLibreria {
public:
    virtual const char* DameElNombre() const;
};


class DivisionPorCero : public ExcepcionLibreria {
public:
    virtual const char* DameElNombre() const;
};


class MemoriaInsuficiente : public ExcepcionLibreria {
public:
    virtual const char* DameElNombre() const;
};


class IndiceFueraDeRango : public ExcepcionLibreria {
public:
    virtual const char* DameElNombre() const;
};


class DistintaDimension : public ExcepcionLibreria {
public:
    virtual const char* DameElNombre() const;
};

#endif
```

Ejercicio (y 2)

```
#ifndef _Vector_
#define _Vector_

#include <iostream>
#include "ExcepcionLibreria.h"
using namespace std;

template <class TipoElemento>
class Vector {
public:
    enum {kTamPorDefecto=10};
public:
    Vector (unsigned int tamanho=Vector<TipoElemento>::kTamPorDefecto);
        // throw (MemoriaInsuficiente)
    Vector (const Vector<TipoElemento>& vector);
        // throw (MemoriaInsuficiente)
    ~Vector ();
    const Vector<TipoElemento>& operator= (
        const Vector<TipoElemento>& vector);
        // throw (MemoriaInsuficiente)
public:
    TipoElemento& operator[] (unsigned int indice) const;
        // throw (IndiceFueraDeRango);
    Vector<TipoElemento> operator+ (
        const Vector<TipoElemento>& vector) const;
        // throw (DistintaDimension)
    Vector<TipoElemento> operator- (
        const Vector<TipoElemento>& vector) const;
        // throw (DistintaDimension)
    Vector<TipoElemento> operator* (
        const Vector<TipoElemento>& vector) const;
        // throw (DistintaDimension)
    Vector<TipoElemento> operator/ (
        const Vector<TipoElemento>& vector) const;
        // throw (DistintaDimension, DivisionPorCero);
    void Imprimir (ostream& salida) const;
    unsigned int Tamanho () const;
private:
    void CopiarDesde (const Vector<TipoElemento>& vector);
        // throw (MemoriaInsuficiente)
    void Destruir ();
private:
    unsigned int fTamanho;
    TipoElemento* fDatos;
};
#include "Vector.cpp"

#endif
```

La librería estándar de C++

- Proporciona:
  - `string`
  - Entrada/salida por medio de streams
  - Contenedores: `vector`, `list`, `map`, `set`, `stack`, `queue`, etc.
  - Algoritmos: for_each, de comparación, de copia, operaciones matemáticas, mezclado, de búsqueda, de ordenación, etc.
  - Soporte análisis numérico: funciones matemáticas estándar, aritmética de vectores, números complejos, etc.
- Es muy eficiente.
- Todos los componentes están definidos en el espacio de nombres `std`.
- Ficheros cabecera.
  - `<string>`, `<iostream>`, `<fstream>`, `<sstream>`, `<vector>`, `<list>`, `<map>`, `<functional>`, `<algorithm>`, etc.
    - `<iostream.h>`, `<fstream.h>` y `<strstream.h>`, etc. ya no forman parte de la librería estándar C++, si bien, la mayor parte de los compiladores siguen soportándolos (pero no son iguales que las versiones estándares).
    - Obsérvese que `<string>` no tiene nada que ver con `<string.h>`.
  - Para cada cabecera `X.h` de la librería estándar C, existe la cabecera equivalente `<cX>`, que contiene las mismas definiciones en el espacio de nombres `std`.
    - `<cstdlib>`, `<csignal>`, `<cerrno>`, etc.

Strings

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

using namespace std;

int main ()
{
    string str1;
    string str2("Hola");

    str1 = "Adios";

    if (str1 > str2) {
        cout << str1 << " es mayor que " << str2 << endl;
    } if (str1 == str2) {
        cout << str1 << " es igual a " << str2 << endl;
    } else {
        cout << str1 << " es menor que " << str2 << endl;
    }

    return 0;
}
```

Entrada/salida por medio de streams (1)

- Un *stream* es un flujo de datos (fichero, cadena de caracteres).
- La librería de *streams* proporciona las mismas funcionalidades que la librería `stdio`, pero es orientada a objetos.
- Objetos predefinidos: `cin`, `cout`, `cerr`.
- Clase padre: `ios`.
- Clase `ostream`.
  - Hereda de `ios`.
  - Representa un *stream* de salida.
  - La operación de salida se realiza usando el operador <<.
  - Proporciona métodos para introducir datos en el *stream*.
  - << está sobrecargado para los tipos básicos.
  - << ha de aplicarse a un objeto de la clase `ostream` o de uno de sus hijos.
  - << se puede sobrecargar para nuestros tipos (inclusive clases).
- Clase `istream`.
  - Análoga a `ostream` para entrada.

- Ejemplo:

```
#include <iostream>
#include <string>
using namespace std;
int main ()
{
    cout << "Hola" << endl;
    double f = 12.3;
    cout << f << endl;
    cout << "Introduce un entero: ";
    int i;
    cin >> i;
    cout << "Introduce una cadena de caracteres: ";
    string cadena;
    cin >> cadena;
    cout << cadena << endl;
    return 0;
}
```

Entrada/salida por medio de streams (2)

- Sobrecarga de operadores << y >>.

```
#include <iostream>
using namespace std;
class Punto {
public:
    Punto (float x=0, float y=0) { fX = x; fY = y; }
    void PonerX (float x) { fX = x; }
    void PonerY (float y) { fY = y; }
    float ObtenerX () const { return fX; }
    float ObtenerY () const { return fY; }
private:
    float fX, fY;
};

istream& operator >> (istream& s, Punto& p)
{
    float x, y;

    s >> x >> y;
    p.PonerX(x);
    p.PonerY(y);
    return s;
}

ostream& operator << (ostream& s, const Punto& p)
{
    s << p.ObtenerX() << ' ' << p.ObtenerY() << endl;
    return s;
}

int main ()
{
    Punto p;

    cout << "Introduce punto: ";
    cin >> p;
    cout << "Punto introducido: " << p;
    return 0;
}
```

Entrada/salida por medio de streams (3)

- Clase `ofstream`.
  - Hereda de `ostream` y está asociada a un fichero.

```
#include <fstream>
#include <string>
using namespace std;

int main ()
{
    ofstream fichero("Salida.txt");
    string mensaje("Hola y adiós");
    double d = 12.3;
    Punto punto(2, 3);

    fichero << mensaje << endl;
    fichero << d << endl;
    fichero << punto << endl;
    return 0;
}
```

- Clase `ifstream`.
  - Hereda de `istream` y está asociada a un fichero.

```
#include <fstream>
#include <string>
using namespace std;

int main ()
{
    ifstream fichero("Salida.txt");
    string mensaje;
    double d;
    Punto punto;

    getline(fichero, mensaje);
    fichero >> d;
    fichero >> punto;

    cout << mensaje << endl;
    cout << d << endl;
    cout << punto << endl;
    return 0;
}
```

Entrada/salida por medio de streams (4)

- Ejemplo: copia de ficheros.

```
// NOTA ACERCA DE PORTABILIDAD: Este ejemplo debería usar ios_base::out,
// ios_base::trunc, etc. en vez de ios::out, ios::trunc, etc., y hacer un
// #include <ios> (que define la clase ios_base). Se ha utilizado la
// segunda opción, dado que compila en casi cualquier compilador
// (ej.: GNU, MVC++), mientras que la segunda (la estándar) no (ej.: GNU).

#include <fstream>
#include <iostream>
#include <string>

using namespace std;

int main (int argc, const char* argv[])
{
    string nombrePrograma(argv[0]);

    if (argc != 3) {
        cerr <<  "Uso: " + nombrePrograma +
                 " ficheroOrigen ficheroDestino" << endl;
        return -1;
    }

    string nombreFicheroOrigen(argv[1]);
    string nombreFicheroDestino(argv[2]);

    // Los streams siempre se pueden testear tal y como ilustra este
    // ejemplo. Esto es posible, dado que se redefinen los operadores
    // void* y !. El resultado  es "true" si la operación anterior ha
    // tenido éxito, y "false" en caso contrario. Además, se precisa
    // abrir los streams en modo binario (por defecto se abren en modo
    // texto) para que el código sea portable a sistemas operativos
    // que tratan de manera diferente a los ficheros de texto y a los
    // binarios (ej.: MS-Windows).

    ifstream entrada(nombreFicheroOrigen.c_str(), ios::in | ios::binary);
    if (!entrada) {
        cerr << "No se puede abrir " + nombreFicheroOrigen << endl;
        return -1;
    }
```

Entrada/salida por medio de streams (5)

- Ejemplo: copia de ficheros (cont).

```
    char caracter;

    // No valdría "entrada >> caracter" porque con ">>" se saltan los
    // espacios en blanco, fines de línea, etc. "entrada.get" devuelve
    // el stream. En la última iteración, "entrada.get" intenta leer más
    // allá del final del flujo, por lo que el operador void* devuelve
    // "false" (porque la operación no ha tenido éxito).

    while (entrada.get(caracter)) {

        salida.put(caracter);
        if (!salida) {
            cerr << "Error escribiendo en " + nombreFicheroDestino <<
                    endl;
            return -1;
        }
    }

    // Si no se ha terminado de leer la entrada, es que ha habido un
    // problema de lectura.

    if (!entrada.eof()) {
        cerr <<  "Error leyendo de " + nombreFicheroOrigen << endl;
        return -1;

    }

    return 0;
}
```

Entrada/salida por medio de streams (6)

- Ejemplo: copia de ficheros (versión más eficiente).

```
// NOTA ACERCA DE PORTABILIDAD: Este ejemplo debería usar ios_base::out,
// ios_base::trunc, etc. en vez de ios::out, ios::trunc, etc., y hacer un
// #include <ios> (que define la clase ios_base). Se ha utilizado la
// segunda opción, dado que compila en casi cualquier compilador
// (ej.: GNU, MVC++), mientras que la segunda (la estándar) no (ej.: GNU).

#include <fstream>
#include <iostream>
#include <string>

using namespace std;

int main (int argc, const char* argv[])
{
    string nombrePrograma(argv[0]);

    if (argc != 3) {
        cerr << "Uso: " + nombrePrograma +
                " ficheroOrigen ficheroDestino" << endl;
        return -1;
    }

    string nombreFicheroOrigen(argv[1]);
    string nombreFicheroDestino(argv[2]);

    ifstream entrada(nombreFicheroOrigen.c_str(), ios::in | ios::binary);
    if (!entrada) {
        cerr << "No se puede abrir " + nombreFicheroOrigen << endl;
        return -1;
    }

    ofstream salida(nombreFicheroDestino.c_str(), ios::out | ios::trunc |
                                                  ios::binary);
    if (!salida) {
        cerr << "No se puede escribir en " + nombreFicheroDestino <<
                endl;
        return -1;
    }
```

Entrada/salida por medio de streams (7)

- Ejemplo: copia de ficheros (versión más eficiente, cont).

```
    const unsigned int BUFFER_SIZE = 10000;
    char buffer[BUFFER_SIZE];

    while (entrada.read(buffer, BUFFER_SIZE)) {

        salida.write(buffer, BUFFER_SIZE);
        if (!salida) {
            cerr << "Error escribiendo en " + nombreFicheroDestino <<
                    endl;
            return -1;
        }

    }

    if (!entrada.eof()) {
        cerr <<  "Error leyendo de " + nombreFicheroOrigen << endl;
        return -1;

    }

    // El anterior bucle termina cuando quedan menos caracteres que los
    // que se pretenden leer ("read", al igual que "get", también devuelve
    // el stream, y el operador void* devuelve "false" si no se han leído
    // tantos caracteres como los especificados).  "gcount" devuelve
    // el número de caracteres que se han leído en la última operación
    // de lectura.

    salida.write(buffer, entrada.gcount());
    if (!salida) {
         cerr <<  "Error escribiendo en " + nombreFicheroDestino << endl;
         return -1;
    }

    return 0;
}
```

Entrada/salida por medio de streams (y 8)

- Clase `ostringstream`.
  - Hereda de `ostream`.
  - Está asociada a un `string`.
- Ejemplo.

```
#include <sstream>
using namespace std;
int main ()
{
    ostringstream s;
    string mensaje("Hola");
    double d = 12.3;
    Punto punto(2, 3);

    s << mensaje << ' ' << d << ' ';
    s << punto;
    cout << s.str() << endl;

    return 0;
}
```

- Clase `istringstream`.
  - Hereda de `istream`.
  - Está asociada a una cadena de caracteres.
- Ejemplo.

```
#include <sstream>
using namespace std;
int main ()
{
    string cadena("Hola 12.3 2 3");
    istringstream s(cadena);
    string mensaje;
    double d;
    Punto punto;

    s >> mensaje >> d >> punto;
    cout << mensaje << ' ' << d << ' ' << punto
         << endl;
    return 0;
}
```

Contenedores (1)

- Contenedores de objetos de cualquier tipo.
  - `vector`, `list`, `map`, `set`, `stack`, `queue`, etc.
  - Son clases template (ej.: `vector<MiTipo>`).
- Los elementos de un contenedor son copias de los objetos insertados.
  - El contenedor puede copiar elementos usando el constructor copia o el operador de asignación.
  - Por tanto, es conveniente definir ambos métodos para los tipos que usemos con contenedores, con la semántica apropiada.
- Cuando la copia no es apropiada, el contenedor debería tener punteros a objetos en vez de los objetos.
  - Situación frecuente: se desea recorrer una lista de figuras (heredan todos de un mismo tipo base y redefinen operaciones) e invocar una operación polimórfica.
  - Usar `list<Figure*>` y no `list<Figure>`.

Contenedores (2)

- Ejemplo de `vector`.

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

int main ()
{
    vector<int> v(4);

    v[0] = 1;
    v[1] = 4;
    v[2] = 9;
    v[3] = 16; // 1 4 9 16

    v.push_back(30); // El tamaño del vector aumenta.
    for (int i=0; i<v.size(); i++) {
        cout << v[i] << " ";
    } // 1 4 9 16 30
    cout << endl;

    vector<int>::iterator iterador = v.begin();
    for (iterador = v.begin(); iterador != v.end(); iterador++) {
        cout << *iterador << " ";
    }
    cout << endl;

    // En el caso de "vector", el iterador es de acceso aleatorio,
    // es decir, se le pueden sumar y restar enteros (aparte de usar los
    // operadores ++, -- y * como en cualquier iterador).

    iterador = v.begin() + 2; // Avanza al tercer elemento.
    v.insert(iterador, 123); // 1 4 123 9 16 30(inserta antes del iterador)

    for (iterador = v.begin(); iterador != v.end(); iterador++) {
        *iterador = *iterador + 1;
    } // 2 5 124 10 17 31

    v.erase(v.begin()); // 5 124 10 17 31

    for (iterador = v.begin(); iterador != v.end(); iterador++) {
        cout << *iterador << " ";
    }
    cout << endl;

    return 0;
}
```

Contenedores (3)

- Ejemplo de `list`.
  - Mejor que `vector` cuando las inserciones y borrados son frecuentes.

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

using namespace std;

int main ()
{
    list<int> lista;

    lista.insert(lista.end(), 1);
    lista.insert(lista.end(), 4);
    lista.insert(lista.end(), 9);
    lista.insert(lista.end(), 16); // 1 4 9 16

    list<int>::iterator iterador = lista.begin();
    for (iterador = lista.begin(); iterador != lista.end(); iterador++) {
        cout << *iterador << " ";
    }
    cout << endl;

    // En el caso de "list", el iterador es bidireccional, y por tanto,
    // no se le pueden sumar y restar enteros, pero sí usar las operaciones
    // comunes a cualquier tipo de iterador: ++, -- y *.

    iterador = lista.begin();
    for (int i=0; i<2; i++) { // Avanza al tercer elemento.
        iterador++;
    }
    lista.insert(iterador, 123); // 1 4 123 9 16(inserta antes del iterador)

    for (iterador = lista.begin(); iterador != lista.end(); iterador++) {
        *iterador = *iterador + 1;
    } // 2 5 124 10 17

    lista.erase(lista.begin()); // 5 124 10 17

    for (iterador = lista.begin(); iterador != lista.end(); iterador++) {
        cout << *iterador << " ";
    }
    cout << endl;

    return 0;
}
```

Contenedores (y 4)

- Ejemplo de `map`.

```
#include <string>
#include <map>
#include <iostream>

using namespace std;

int main ()
{
    typedef map<string, int> MiMapa;
    MiMapa mapa;

    mapa["Celta"] = 70;
    mapa["BarÇa"] = 40;
    mapa["Depor"] = 80;
    mapa["Compos"] = 75;

    MiMapa::iterator i;
    for (i = mapa.begin(); i != mapa.end(); i++) {
        cout << ((*i).first) << ": " << (*i).second << "; ";
    }
    cout << endl;

    i = mapa.find("Depor");
    if (i != mapa.end()) { // Si existe
        cout << "Depor: " << mapa["Depor"] << endl;
    }

    mapa.erase("BarÇa");
    for (i = mapa.begin(); i != mapa.end(); i++) {
        cout << ((*i).first) << ": " << (*i).second << "; ";
    }
    cout << endl;

    return 0;
}
```

Algoritmos

- Ejemplo de `sort`.

```
#include <functional>
#include <vector>
#include <algorithm>
#include <iostream>

using namespace std;

int main ()
{
    vector<int> years(5);

    years[0] = 1990;
    years[1] = 1980;
    years[2] = 2000;
    years[3] = 1970;
    years[4] = 1960;

   sort(years.begin(), years.end(), greater<int>() );
    vector<int>::iterator i;
    for (i=years.begin(); i != years.end(); i++) {
        cout << *i << endl;
    }
    return 0;
}
```

Recursos

- Libros de C++.
  - S. B. Lippman, J. Lajoie, *C++ Primer, 3rd edition*, Addison-Wesley, 1998.
  - Bjarne Stroustrup. *The C++ Programming Language*, Longman Higher Education, 2000.
  - James O. Coplien. *Advanced C++: Programming Styles and Idioms*, Addison-Wesley, 1992.
  - S. Meyers, *Effective C++, 2nd edition*, Addison-Wesley, 1997.
  - D. R.Musser, G. J. Derge, A. Saini, A. Stepanov, *STL Tutorial and Reference Guide, 2nd edition*, Addison-Wesley, 2001.

- *Cetus links.*
  - *Central site*: `http://www.cetus-links.org`
  - *Mirror* en España: `http://zeus.uax.es/uax/oop`

- Mi página web.
  - `http://www.tic.udc.es/~fbellas`
  - Transparencias, ejemplos, enunciado de la práctica, etc.