INFORMÁTICA INDUSTRIAL

Programación Orientada a Objetos y el lenguaje C++

M. Abderrahim, A. Castro, J. C. Castillo
Departamento de Ingeniería de Sistemas y Automática

uc3m | Universidad Carlos III de Madrid

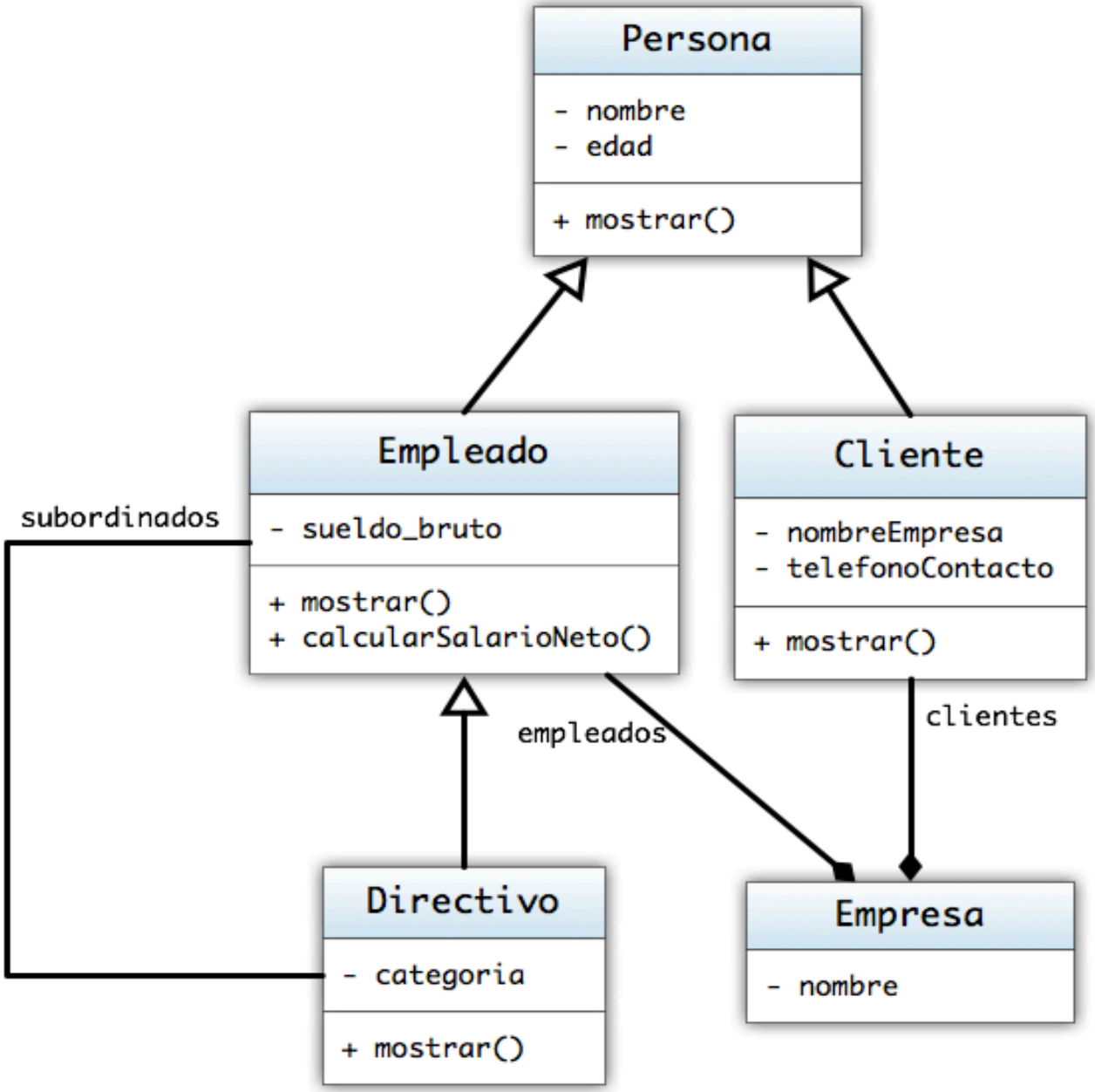
Repaso de conceptos

- **POO:** objetos que se comunican entre si
- **Objeto** = datos + métodos
- **Clase**: descripción de un conjunto de objetos
- Características de la POO
  - **Abstracción**: generalización
  - **Encapsulamiento**: "cajas negras"
  - **Herencia**: "…es un tipo de…"
  - **Polimorfismo**

Repaso de conceptos

- **POO:** objetos que se comunican entre si
- **Objeto** = datos + métodos
- **Clase**: descripción de un conjunto de objetos
- Características de la POO
  - **Abstracción**: generalización
  - **Encapsulamiento**: "cajas negras"
  - **Herencia**: "…es un tipo de…"
  - **Polimorfismo**

Conceptos de POO: Ejemplo 1



Conceptos de POO: Ejemplo 2

- Clase Rectangulo
  - Atributos: altura (real), anchura(real)
  - Métodos:
    - void dibujar()
    - void rotar(real)
    - **Constructores**:
      - Rectangulo(real, real)
      - Rectangulo(real)
  - Instanciación:
    - rect1 = Rectangulo(3.3, 5.6)

Lenguaje C++

- C++ fue diseñado a partir de C, y con pocas excepciones, los programas C son programas C++.
- El diseño de C se basó en expresividad y eficiencia.
- El diseño de C++ hace énfasis en la estructuración (sin perder eficiencia):
  - Orientación a objetos.
  - Programación genérica.
  - Abstracción de datos.
  - Estructuración y modularización mediante espacios de nombres.
  - Manejo de excepciones (errores).

Historia

- B. Stroustrup (AT&T Bell Laboratories), principios de los 80.
- Heredero de "C", que es heredero de "B" (Richards 1980) y de Simula 67.
  - C, especificación formal 1997 (Ritchie y Kernighan).
- Estandarización ANSI. Tercera edición de "The C++ Programming Language" (1997).

Primer Programa en C++

- Hola Mundo

```
#include <iostream>
using namespace std;
// mi primer programa C++
int main(){
      cout << "Hola Mundo! " << 2014 << "\n";
}
```

Cabeceras

- Directiva **include**: 2 variantes

  #include <cabecera>

  #include “fichero fuente”

- La segunda forma no varía con el lenguaje C.
- La primera forma es diferente a C. Una cabecera no tiene que corresponder necesariamente con un fichero, aunque es lo habitual.

  #include <stdio.h> // C

  #include <iostream> // C++

- C++ incluye la librería estándar de C

  #include <cstdio>

- Se sigue pudiendo usar #include <iostream.h>, pero **no se recomienda**.

Entrada / Salida

- *iostream*: Biblioteca estándar de entrada/salida.
- *cout*: Canal de salida estándar.
- *cin*: Canal de entrada estándar.
- *std*: Espacio de nombres donde se declaran todos los identificadores de la biblioteca estándar.
  - Para usar un identificador como *cout* se debe escribir ***std::cout*** en cada ocurrencia o bien escribir una sola vez ***using namespace std;*** al principio del programa.
- Uso:

```
cout << expresión;
cin >> variable; // ¡¡Ojo!! Sin &.
```

Tipos de Datos en C++

- Tipos por defecto (int, char, etc)
- A partir de los anteriores, se pueden construir:
  - tipos punteros (ej. int *)
  - tipos array (ej. char [])
  - tipos **referencia** (ej. double&)
  - Estructuras y **Clases**.

Ámbito

- Una declaración introduce un nombre dentro de un ámbito, y sólo se puede usar en este ámbito.

- Operador de acceso o resolución de ámbito **::**

Ejemplo

- ambito.cpp

```
#include <iostream>
int x;              // x global
int main()
{
    int x;          // x local -> oculta la x global
    x=1;            //uso de la x local
    {
        int x;              // oculta la x local anterior
        ::x = x = 2;        // usa la x del bloque actual y la global
        std::cout << x << " " << ::x << std::endl;
    }                       // la x de este bloque se destruye
        x = 3;              // usa la x local del main
        ::x=4;              // usa la x global

    std::cout << x << " " << ::x << "\n";
}
```

Espacios de nombres

- Un espacio de nombres es un *ámbito con nombre*.
- Ayuda a evitar las colisiones de nombres.
- Se usa la directiva ***using*** para añadir un espacio de nombres a la lista de ámbitos que se está usando.

```
using namespace std;
```

- Relación entre espacios de nombres y cabeceras:

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

equivale a (recomendable):

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

- Ejemplo: http://www.cplusplus.com/doc/oldtutorial/namespaces/

Referencias

- Una referencia es un nombre alternativo para un objeto.
- Su uso principal está en el paso de parámetros y valores de retorno a funciones.
- Notación: X& es una referencia al tipo X.

```
int main()
{
    int i=1;
    int& r=i;    // i y r se refieren al mismo int
    int x=r;     // x=1
    r=2;         // i=2;
}
```

Referencias

- Deben inicializarse en su declaración:

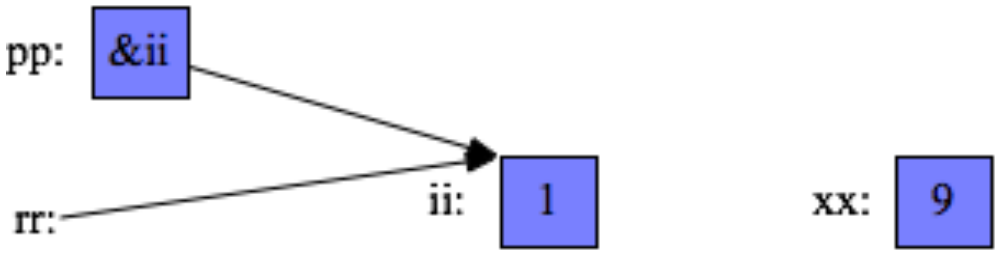
```
int main()
{
    int i=1;
    int& r=i;          // i y r se refieren al mismo int
    int&r2;            // error, r2 no se ha inicializado
    extern int &r3; // ok, r3 se ha inicializado en otro sitio

    int& rr=2;         // error
}
```

Referencias

- Los operadores que se aplican a una referencia afectan en realidad al objeto referenciado.

```
int main()
{
    int ii=0, xx =9;
    int& rr=ii;
    rr++;                   // se incrementa ii
    int * pp=&rr;           // pp apunta a rr
    rr = xx;                // ii y rr valen 9
}
```

- Similar a un puntero (constante) que se derreferencia automáticamente cada vez que se usa.



Referencias

- Una parámetro pasado por referencia permite cambiarlo.

```
void incr(int &a) {a++;}

int main()
{
    int x = 1;
    incr(x);    // x = 2;
}
```

- La semántica es la misma que en la inicialización. En el ejemplo, a es otro nombre para x.
- Hace más eficiente el paso de parámetros ya que **no se copia el valor del parámetro**.

Referencias

- Una referencia devuelta por una función se puede modificar.

```
int &incr(int &a){
    a++;
    return a;
}
int main()
{
    int x = 1;
    incr(x)++;   // x = 3;
    incr(x)=5;   // x = 5;
}
```

- Muy útil para definir y sobrecargar operadores (ej.: cout << “a” << 3;) y para encadenar llamadas: fecha1.add_day(1).add_year(2);

Estructuras y Clases

- Supongamos que se quiere definir un tipo de dato *Fecha* de la siguiente forma:

```
struct Fecha{
    int d, m, a;
};

void inic(Fecha* f, int d1, int m1, int a1)
{
    f->d=d1;
    f->m=m1;
    f->a=a1;
}
```

Estructuras y Clases

- Supongamos que se quiere definir un TAD Fecha, ahora con referencias:

```
struct Fecha{
    int d, m, a;
};

Fecha &inic(Fecha &f, int, int, int);
Fecha &sumarDia(Fecha &f, int);
Fecha &sumarMes(Fecha &f, int);
Fecha &sumarAno(Fecha &f, int);
```

```
Fecha& sumarAno(Fecha& f, int a)
{
    f.a += a;           // es f.a = f.a +a
    return f;
}
```

Estructuras y Clases

- En C++, se puede (y es más intuitivo) declarar las funciones de manipulación dentro de la estructura:

```
struct Fecha{
    int d, m, a;
    Fecha &inic(int d1, int m1, int a1);
    Fecha &sumarDia(int d1);
    Fecha &sumarMes(int m1);
    Fecha &sumarAño(int a1);
};
Fecha &Fecha::inic(int d1, int m1, int a1){
    d = d1;
    m = m1;
    a = a1;
    return *this;   //Operador de ámbito
}

Fecha &Fecha::sumarDia(int d1){
    d += d1;
    return *this;
}
```

Estructuras y Clases

- Las funciones declaradas dentro de clases (las estructuras son un tipo de clase) se llaman funciones miembro, y se llaman de la siguiente forma:

```
Fecha mi_cumple, hoy;
Fecha * pfecha;

hoy.inic(4,10,2010);

mi_cumple.inic(5,1,1984);
mi_cumple.sumarDia(1);

//Equivalente
mi_cumple.inic(5,1,1984).sumarDia(1);

Fecha manana= hoy;

manana.sumarDia(1);
pfecha->sumarMes(3);
```

Estructuras y Clases

- Las clases son muy parecidas a las estructuras

```
class complex
{
    double real;
    double imag;

    void estableceValor(float re,float im) ;
    float obtenModulo(void) ;
    void imprime() ;
};
```

- Una estructura es una clase en la que los miembros son públicos por defecto. **En una clase los miembros son privados por defecto**.
- Las estructuras no permiten encapsulamiento ni herencia

Estructuras y Clases

- ¿Cómo se declaran miembros públicos y privados de una estructura/clase?

```
class complex
{
    double real;
    double imag;

    void estableceValor(float re,float im) ;
    float obtenModulo(void) ;
    void imprime() ;
};
```

Encapsulamiento

- Lograr que cada objeto se comporte de forma autónoma y lo que pase en su interior sea invisible para el resto de objetos

```
class listaNumeros {
private:
    int num;
    int lista[100];
public:
    int agregarNumero(int val);
    int extraerNumero(int ind);
    int numeroNumeros(void);
    listaNumeros(void);
};
```

```
void main()
{
    listaNumeros milista;
    int i,val=1;
    while(val!=0)
    {
        cout<< "Introduzca un numero (finalizar con 0):";
        cin>>val;
        if(val!=0) val=milista.agregarNumero(val);
    }
    cout<<"\nLa lista de números es la siguiente:\n";
    for(i=0; i<milista.numeroNumeros(); i++)
        cout<<milista.extraerNumero(i)<<" ";
    cout<<"\n*********FIN DEL PROGRAMA**********\n";
}
```

Encapsulamiento

- Miembros públicos, privados y protegidos.
  - Por defecto: privados
- Interfaz pública de una clase: miembros públicos (datos y métodos).
  - Se pueden invocar desde fuera de la clase.
- Ejemplo (clase Rectangulo)
  - Datos:
    - privado altura (real)
    - privado ancho (real)
  - Métodos:
    - público void dibujar()
    - público void girar(real)
  - Constructores:
    - público Rectangulo(real, real)
    - público Rectangulo(real)

Encapsulamiento

```
class Rectangulo
{
    float altura; //Privado por defecto
public:
    float area; //Accesible
private:
    float ancho; //Solo accesible para la clase
protected:
    char* descripcion; //Solo para la clase y derivadas

public: //Accesible
    Rectangulo(float alto, float ancho);
    Rectangulo(float lado);
    void dibujar();
    void girar(float angulo);
};
```

Creación y Destrucción de Objetos

- Equivalencia de conceptos con programación clásica:
  - Tipo ⇔ Clase
  - Dato ⇔ Objeto
  - Variable: existe en ambos tipos de programación.
- Creación (uso del constructor):
  - Ejemplo:
    - rect1 = Rectangulo(1,2), rect2 = Rectangulo(3.5)
- Destrucción (uso del destructor)
  - Explícita (heap) o automática (stack) (C++).
  - Automática (Java o Python, *Garbage collection*).

Creación y Destrucción de Objetos

- Constructores
  - Función inicializadora
  - Es una función que no devuelve nada y que se llama igual que la clase.
  - No se hereda.
  - Existen tres tipos fundamentales (sobrecarga):
    - **Constructor vacio**
    - **Constructor parametrizado**
      - Sirve para asignar un valor a los atributos en la creación.
    - **Constructor de copia**
      - Sirve para crear un objeto que sea copia de otro.
- Destructores
  - Función de eliminación

Constructores

- El uso de métodos como inic() es poco elegante y propenso a errores.
- *Constructor:* un método con el propósito de inicializar un objeto. Tiene el nombre de la clase

Constructores

```
class Fecha{
    int d, m, a;
public:
    Fecha(int dd, int mm, int aa);
};

Fecha::Fecha(int dd, int mm, int aa)
{
    d = dd;
    m = mm;
    a = aa;
}

int main() {
    Fecha n = Fecha(10,2,1986);
    Fecha m(10,2,1986);
    Fecha * k = new Fecha(10,9,2003);
    ……..
    delete k;
}
```

Copia de Objetos

- Por defecto, los objetos se pueden copiar.
- Se puede inicializar un objeto con la copia de otro.
- Por defecto se copia cada atributo:

```
Fecha hoy = m;
```

- Si no es lo que se quiere, se ha de definir un constructor copia.
- También se llama en argumentos de métodos y valores de retorno

Copia de Objetos

```
class Fecha{
    int d, m, a;
public:
    Fecha(int dd, int mm, int aa);
    Fecha(const Fecha &fe);
};

Fecha::Fecha(const Fecha &fe)
{
    d = fe.d + 1; // se puede introducir modificaciones al
                  // copiar elementos
    m = fe.m;
    a = fe.a;
}

int main() {
    Fecha m(10,2,1986);
    Fecha hoy = m; // También: Fecha m(hoy);
}
```

Copia de Objetos

- ¿Cómo podemos evitar que un objeto se pueda copiar? Es decir, evitar expresiones del tipo:

Fecha hoy = m;

¿Solución?

Solución

```
class Fecha{
    int d, m, a;
public:
    Fecha(int dd, int mm, int aa);
private:
    Fecha(const Fecha &fe);
};

Fecha::Fecha(const Fecha &fe)
{
    d = fe.d + 1; // se puede introducir modificaciones
al copiar                                  // elementos
    m = fe.m;
    a = fe.a;
}

int main() {
    Fecha m(10,2,1986);
    Fecha hoy = m; //error no se puede acceder
                   //al miembro privado
}
```

Múltiples constructores

```
class Fecha{
    int d, m, a;
public:
    Fecha(int dd, int mm=1, int aa=2014) : d(dd), m(mm), a(aa) {}
    Fecha() : d(0), m(0), a(0) {}
    Fecha(float dd);
};
```

Constructores

Parámetros por defecto, inicialización, conversiones implícitas

```
#include <iostream>

using namespace std;

class Fecha{
    int d, m, a;
public:
    Fecha(int dd, int mm=1, int aa=2014) : d(dd), m(mm), a(aa) {}
    Fecha(int dd, int mm=1, int aa=a); //¿válido?
};
int main()
{
    Fecha n = 2; // Equivalente a Fecha n(2);
    Fecha p(5,6);
    Fecha m(3);
    Fecha q = Fecha(2);
}
```

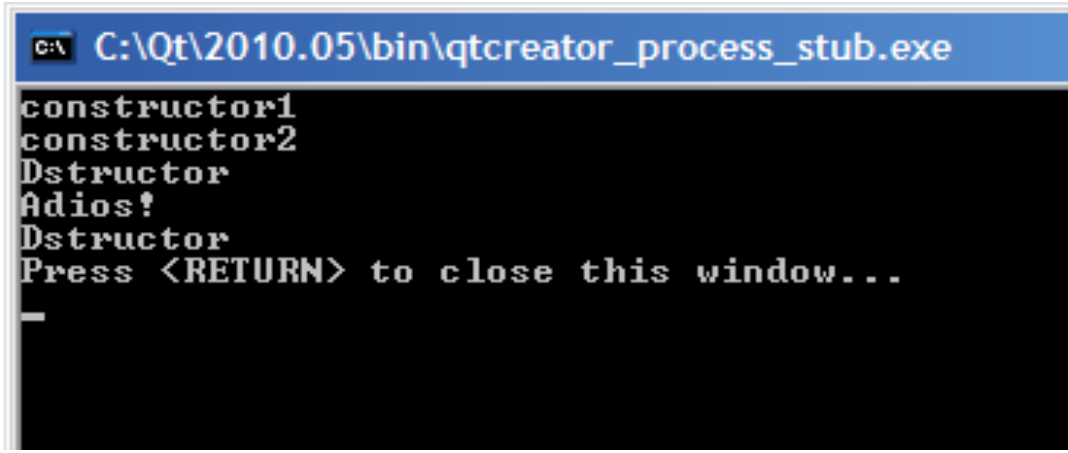
Destructores

- Se llaman cuando un objeto se destruye, y sirven por ejemplo para liberar memoria que se haya reservado durante la vida del objeto.
- Normalmente el compilador los llama implícitamente.

Destructores

```
#include <iostream>

using namespace std;

class Bloque{
    int tamano_b;
    char * b;
public:
    Bloque(int tam) : tamano_b(tam){
        b = new char[tam];
        cout << "constructor1" << endl;
    }
    Bloque(const Bloque &bl){
        b = new char[tamano_b = bl.tamano_b];
        cout << "constructor2" << endl;
    }
    ~Bloque(){ // destructor
        delete [ ] b;
        cout << "Dstructor"<< endl;
    }
};
```

```
void main() {
    Bloque * b = new Bloque(10);
    Bloque c = *b;
    delete b;
    cout << "Adios!" << endl;
}
```



Ejecución de métodos miembro

- Un método es una definición de una función perteneciente a una clase.
  - Se dice que un método se ejecuta cuando el objeto recibe un mensaje de ejecución del método.
  - Puede acceder a otros miembros (públicos o privados) de la clase.

- Ejemplo. Objeto rect1(Rectangulo) :

```
Rectangulo rect1=Rectangulo(3.5, 2.9);
rect1.girar(30.5);
```

Clases
Invocación de Métodos

- *objeto.metodo(args),* si *objeto* no es un puntero:

  Fecha f(1,10,1492);

  f.sumarAño(515)*;*

- *p_objeto->metodo(args),* si p_objeto es un puntero:

  Fecha * d = new Fecha(1,10,1492);

  d->sumarAño(515);

- *objeto operador argumento*

  Fecha d(1,10,1492);

  cout << “una fecha “ << d+1 <<endl;

Organización de Ficheros

- Declaración de una clase X en “X.h”.
- Definición de sus funciones miembro en “X.cpp”.
- Regla de definición una sola vez: inclusión condicional de código para evitar errores.

Ejemplo: Clase Dado

```
/* Dado.h
 * Created on: 17/05/2010
 * Author: wladimir
 */
#ifndef DADO_H_
#define DADO_H_

class Dado {
public:
    void lanzar();
    int getValorCara();
private:
    int valorCara;
};

#endif /* DADO_H_ */
```

Ejemplo: Definición métodos miembros de la Clase Dado

```
/* Dados.cpp
 * Created on: 17/05/2010
 * Author: wladimir
 */
#include “Dado.h“
#include <stdlib.h>

void Dado::lanzar() {
    valorCara = ((rand() % 6) + 1);
}

int Dado::getValorCara() {
    return valorCara;
}
```

Ejemplo: Clase JuegoDados

```
/* JuegoDados.h
 * Created on: 17/05/2010
 * Author: wladimir
 */
#include “Dado.h"

#ifndef JUEGODADOS_H_
#define JUEGODADOS_H_

class JuegoDados{
public:
    bool jugar();
private:
    Dado dado1;
    Dado dado2;
};

#endif /* JUEGODADOS_H_ */
```

Ejemplo: Definición métodos miembros de la Clase JuegoDados

```
/* JuegoDados.cpp
 * Created on: 17/05/2010
 * Author: wladimir
 */
#include “JuegoDados.h"
#include “Dado.h"

bool JuegoDados::jugar() {
    dado1.lanzar();
    dado2.lanzar();

    return ((dado1.getValorCara() + dado2.getValorCara()) == 7 ? true : false);
}
```

Ejemplo: Programa de Prueba

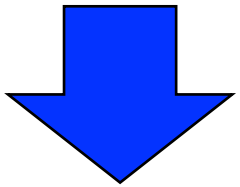
```
/*  JugarDado.cpp */

#include <iostream>
#include <stdio.h>
#include "JuegoDados.h”

using namespace std;

int main(){
    bool ganarPartida, continuarJuego = true;
    char tecla;
    JuegoDados elJuego;
    cout << "\nJuego de dados, lanzar dos dados y sumar 7 para ganar\n";
    while (continuarJuego){
        cout << "\nPulsar cualquier tecla para lanzar dados\n";
        cin >> tecla;
        ganarPartida = elJuego.jugar();
        if (ganarPartida)
            cout << "Ha ganado.\n";
        else
            cout << "Ha perdido.\n";
        cout << "\nDesea jugar otra vez, pulse C\n";
        cin >> tecla;
        continuarJuego = (tecla == 'c')||(tecla == 'C')?true:false;
    }
    cout << "Se termino el juego de dados. Un saludo\n”
}
```

Funciones inline

- Optimización de la eficiencia a costa del tamaño del programa ejecutable.
- Si se puede se sustituyen directamente en el código; no se realiza la llamada a la función.
- Los métodos definidos dentro de la definición de clase se toman como inline.
- Esto se debe reservar a métodos pequeños que se usan frecuentemente.

Ejemplo

```
inline int f(int x) { return x * (x+1); }

int main () {
   int x = 2;
   int y = f(x);
   int z = f(y);
}
```



```
int main () {
   int x = 2;
   int y = x*(x+1);     // Pero no es una simple sustitución de texto
   int z = y*(y+1);
}
```

Funciones inline

- Se aconseja su uso en lugar de #define
- Ventajas:
  - no dan lugar a errores sintácticos

    ej: #define doble (a) a+a
  - Hacen comprobación de tipos

uc3m | Universidad Carlos III de Madrid