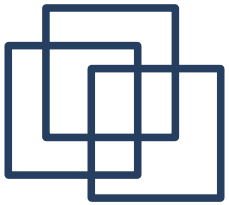
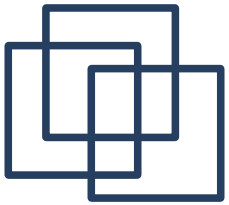
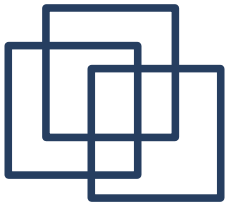


Conceptos básicos sobre C++



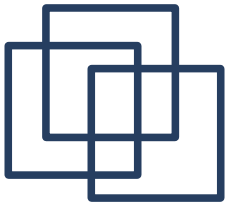
Objetivos Generales

- El lenguaje C++
- Clases
- Sobrecarga de funciones
- Sobrecarga de operadores



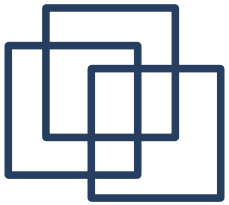
El lenguaje C++

- C++ es un lenguaje que incluye como sublenguaje al ANSI C.
- Soporta múltiples paradigmas de programación. En particular, Programación Dirigida a Objetos.
- Soporte para la creación de nuevos tipos de datos y sus operaciones.
- Soporta sobrecarga de operadores y funciones.



El lenguaje C++ (II)

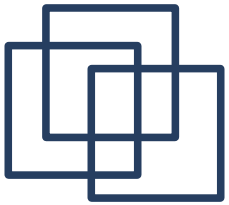
- Tipos referencia
  - Una función es capaz de devolver una referencia a un dato existente.
- Funciones *inline*
  - El compilador sustituye la llamada a la función por el código de la propia función después de realizar todas las comprobaciones semánticas.
- Tipo lógico: *bool*
- *Programación genérica: templates*
  - Herramienta que permite la definición de tipos complejos genéricos capaces de contener otros tipos “parametrizados”



Clases en C++ (I)

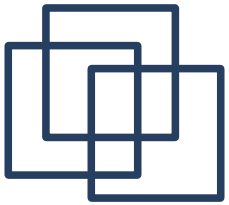
Clase (class) de C++:

construcción que permite construir nuevos tipos de datos, cumpliendo con las condiciones para poder usar abstracción de datos.



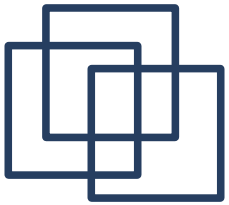
Clases en C++ (II)

- *Ocultación de información:*
  - Control de acceso a los componentes miembros mediante dos niveles de acceso: público (*public*) y privado (*private*).
- *Encapsulamiento:*
  - Sus componentes son tanto datos (*variables de instancia*) como operaciones (*funciones miembro o métodos*). La clase establece un fuerte vínculo entre todos los miembros (datos y operaciones). Ambos aparecen juntos.



Iniciación y destrucción

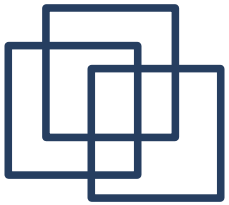
- Iniciación de objetos automática: constructores.
- Destrucción automática de objetos: *destructores*.



(Clase Fecha. Versión 0.1)

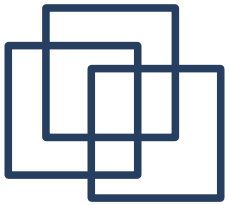
```
class Fecha {
public:
  Fecha (int dia, int mes,
                int anio);
  void Lee(const char * cadena);
  int Dia() const;
  int Mes() const;
  int Anio() const;
  void Escribe(char * cadena)
                const;
  void Siguiente(Fecha * g)
                const;
  void Anterior(Fecha * g) const;
  bool menor(Fecha f2);
  bool menor_o_igual(Fecha  f2);
```

```
private:
  int dia;
  int mes;
  int anio;
};
```



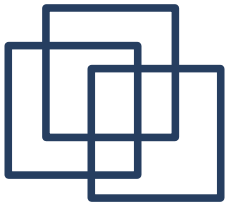
(Clase Fecha. Versión 0.2)

```
class Fecha {
public:
  Fecha (int dia, int mes,
         int anio);
  void Lee(const char *
         cadena);
  int Dia() const;
  int Mes() const;
  int Anio() const;
  void Escribe(char * cadena)
         const;
  void Siguiente(Fecha & g)
         const;
  void Anterior(Fecha & g)
         const;
  bool operator<(const Fecha &
         f2);
  bool operator<=(const Fecha
         & f2);
private:
  int dia;
  int mes;
  int anio;
};
```



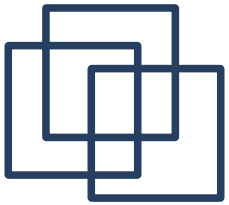
(Clase Fecha. Versión 0.3)

```
class Fecha {
public:
  Fecha (int dia, int mes, int anio);
  //... resto de métodos.
private:
  int dia;
  int mes;
  int anio;
 bool bisiesto(int anio) const;
 const static int dias_mes[12] =
        {31, 28, 31, 30, 31, 30,  31, 31,
        30, 31, 30, 31};
};
```



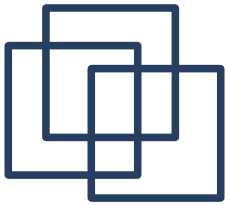
Clases en C++

- Control de acceso:
  - la palabra reservada *class* incorpora un mecanismo de control de acceso a la estructura basada en *private* y *public*.
- Acceso a los miembros:
  - únicamente a miembros *public*.
- Funciones miembro.



Clases en C++

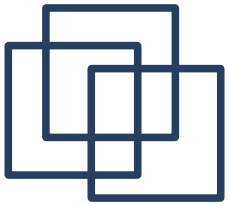
- Constructores y destructores:
  - para crear un dato del tipo e inicializarlo (constructor) o destruirlo (destructor).
- Funciones miembro inline.



Control de acceso

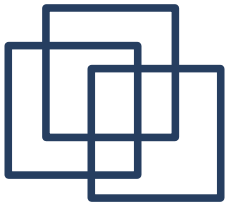
Las clases permiten controlar el acceso a sus componentes (miembros) usando dos nuevas palabras reservadas:

- ***private***: indica que estos miembros son accesibles sólo por otros miembros de la clase, pero no por nadie externo a la clase (ocultación de información).
- ***public***: los miembros definidos son accesibles para cualquiera.



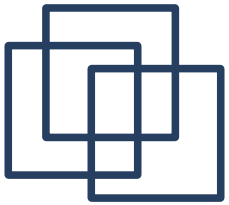
Control de acceso (II)

- Estas palabras se incluyen en la definición de la clase entre las declaraciones de los miembros.
- Tienen efecto desde que se declaran hasta que aparece otra diferente o hasta al final de la clase.
- Por defecto, todos los miembros son privados.



Acceso a los miembros (I)

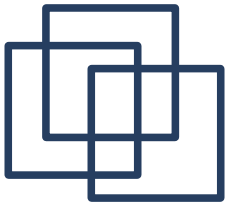
- La definición de una clase incluye:
  - definiciones de datos (variables de instancia),
  - operaciones asociadas (funciones miembro o métodos).
- Junto con el control de acceso (ocultación de información) favorece el encapsulamiento.
- Todas las funciones miembro tienen acceso a todos los miembros de la clase tanto public como private.



Acceso a los miembros (II)

- Cada método posee un parámetro implícito: el propio objeto. A ese parámetro se puede acceder usando un puntero llamado ***this***
- Al ser un parámetro puede pasarse de forma **constante** o **no constante**:

```
class ejemplo {
public:
    int f(int x); // objeto no constante
    int g(int x) const; // objeto constante
private:
   int dato;};
```



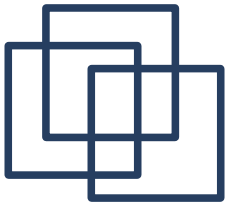
Acceso a los miembros (III)

- Se podría implementar

```
int ejemplo::f(int x)
{
    dato = x;
    return dato;
}
```

- Pero no se podría hacer lo mismo con **g**
- El acceso a otros miembros de la clase se hace nombrándolos directamente o a través de *this*:

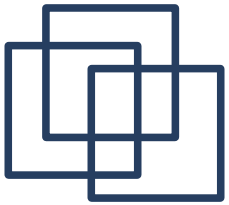
```
Fecha::Fecha(int D, int M, int A)
{dia = D;     this->mes = M;     anio = A; }
```



Funciones miembro (I)

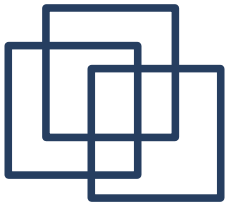
- Se declaran dentro de la definición de la clase y se implementan fuera.
- Para indicar que son miembros de su clase se usa el operador de ámbito (::):

```
int Fecha::Mes() const {
  return mes;
}
void Fecha::Siguiente(Fecha & g)
 {...}
```



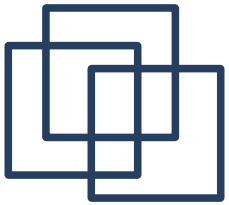
Funciones miembro (II)

- La definición de las funciones miembro es parte de la implementación de la clase.
- Su implementación se incluye en el fichero .cpp y no en el .h o .hpp
- El usuario del tipo no necesita conocer su implementación para usarlas.
- Su implementación sólo es necesaria en tiempo de enlazado, no de compilación.



Funciones miembro *inline*

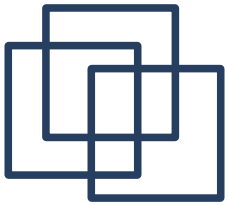
- Existen funciones miembro pequeñas que, por cuestiones puramente de eficiencia, se carguen más rápido: se “incrustan” en el lugar que se usan.
- Esto no afecta al usuario del tipo de dato, pero sí al compilador.
- Es necesario conocer su implementación en tiempo de compilación (antes del enlazado). Por ello es necesario incluir su implementación en el fichero cabecera.
- Las funciones miembro *inline* se pueden definir:
  - a) dentro de la clase o
  - b) fuera de la clase.



Funciones *inline* dentro de la clase

Su implementación debe ir justo después de la declaración de la función. No se incluye la palabra reservada inline:

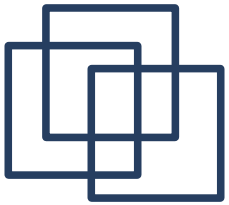
```
class Fecha {
  int Dia() const
    { return dia; }
};
```



Funciones miembro *inline* fuera de la clase

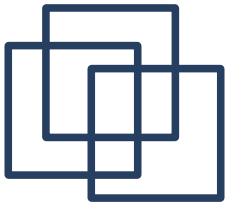
La implementación se hace como en el caso general, pero se precede de la palabra reservada inline y se incluye en el fichero cabera, una vez finalizada la definición de la clase:

```
class Fecha {
  int Dia() const;
};
inline int Fecha::Dia() const {
  return dia;
}
```



Funciones amigas (I)

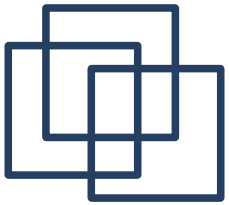
- Se usan para definir funciones que no realizan una operación sobre el objeto, sino con el objeto.
- No usan la sintaxis *objeto.funcion (...)*
- Es una función global (no miembro de una clase) con acceso a sus miembros privados. **Esto implica saltarse los mecanismos de control de acceso**.
- Sólo se deben usar cuando no haya otra opción.



Funciones amigas (II)

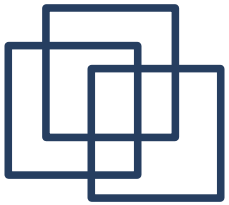
- Se tienen que declarar dentro de la definición de la clase, usando la palabra reservada *friend*:

```
class polinomio {
public:
    polinomio ();
    ...
    friend float Calculo (const polinomio
      &p1, const polinomio &p2);
    ...
};
```



Constructores (I)

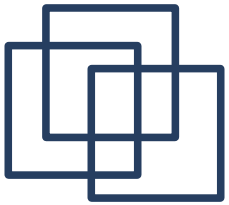
- Son funciones invocadas automáticamente al instanciar un objeto (“declara” una variable perteneciente a una clase).
- Su objetivo es poner el dato en un estado inicial válido: “construirlo”
- No devuelven datos ni tienen tipo de dato de retorno, ni siquiera *void*.
- Su nombre coincide con el de la clase (sólo un contructor puede tener ese nombre)



Constructores (II)

- Puede haber más de un constructor, pero sus declaraciones (prototipos) han de ser distintas (en número y/o tipo de parámetros)
- Un constructor sin argumentos es el constructor por defecto:

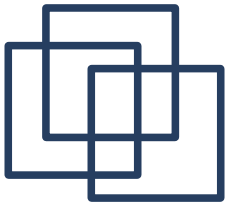
```
class racional {
public:
  racional();
};
```



Constructores (III)

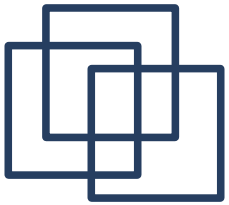
- *Constructor de copia*: crea un objeto de la clase a partir de otro:

```
class racional {
public:
  racional(const racional & r);
};
racional::racional(const racional & r)
{
  num = r.num;
  den = r.den;
}
// también se puede implementar como:
// racional::racional(const racional & r) : num(r.num),
   den(r.den)
//{}
```



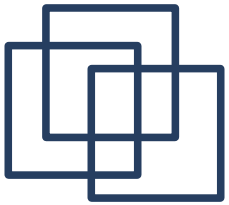
Constructores (IV)

- No es obligatorio definir constructores, aunque sí muy conveniente.
- Si no se define ningún constructor, el compilador crea dos:
  - por defecto
  - de copia



Destructores (I)

- Son las operaciones antagónicas de los constructores.
- Su objetivo es liberar todos los recursos asociados al objeto (p.ej.: memoria).
- No tienen tipo de retorno.

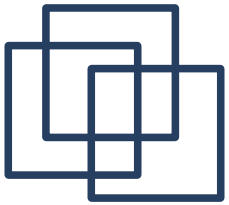


Destructores (II)

- Su nombre se construye anteponiendo ~ al nombre de la clase:

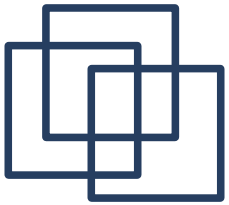
```
class Fecha {
public:
  ~Fecha();
...
};
```

- Se invocan automáticamente cuando un objeto deja de existir (termina su ámbito)



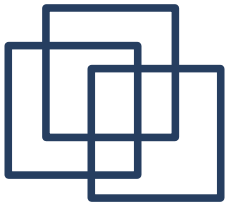
Destructores (II)

```
class array {
public:
  array(int n) {datos = new int[n];};
  ~array() {
     delete [] datos;
  };
private:
  int *datos;
};
```



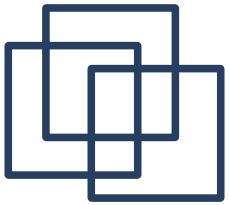
Clase **Fecha**

```
#include "Fecha.h"

Fecha::Fecha(int Dia, int Mes, int Anio)
{
  this->dia = Dia;
  this->mes = Mes;
  this->anio = Anio;
};
```



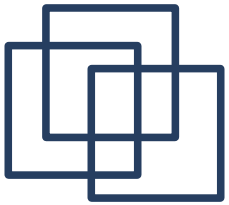
Clase **Fecha**

```
int Fecha::Dia() const
{ return dia; };

int Fecha::Mes() const
{ return mes; };

int Fecha::Anio() const
{ return anio; };
```



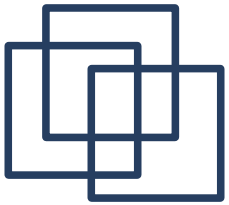
Clase **Fecha**

```
void Fecha::Escribe() const
{
  cout << dia << '/' << mes << '/' << anio;
};
```



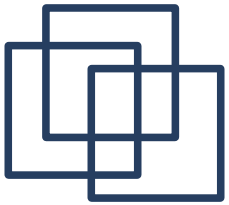
Clase **Fecha**

```
void Fecha::Siguiente(Fecha & g) const
{
  if ((dia < dias_mes[mes - 1]) ||  (bisiesto(anio) && dia
    < 29)) {
          g.dia = dia + 1;
          g.mes = mes;
          g.anio = anio; }
  else {
      g.dia = 1;
      if (mes < 12)
        { g.mes = mes + 1;  g.anio = anio; }
      else
        { g.mes = 1;  g.anio = anio + 1; }
  }
}
```



Sobrecarga de operadores (I)

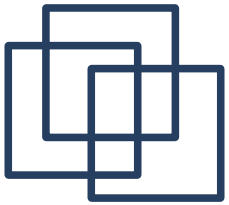
- Los operadores permiten conectar operandos para devolver valores. Ej: + - / * ...
- El conjunto de operadores de C/C++ es amplio, tienen una semántica bien establecida pero fija.
- Su uso es deseable para nuevos tipos de datos: números complejos, racionales, matrices, ...
- C++ permite dar significado a los operadores aplicados a nuevos tipos de datos: sobrecarga de operadores.



Sobrecarga de operadores(II)

- Se define una función con nombre la palabra reservada **operator** seguida del símbolo del operador. Los tipos de retorno y de los argumentos serán los necesarios en cada caso:

```
racional operator+(const racional &r,
        const racional &s);
vect operator*(const vect &v,
        const matriz &m);
Fecha & operator++(Fecha&);
```



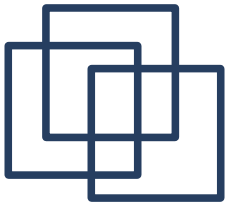
Sobrecarga de operadores(III)

La sobrecarga se puede realizar como:

- Función global.
- Función miembro: el primer argumento se reemplaza por el objeto receptor.

Se pueden sobrecargar todos los operadores excepto:

. * ?: sizeof ::



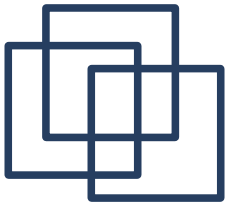
Sobrecarga de operadores(IV)

- Operador * sobrecargado como **función miembro**:

```
racional racional::operator*(const racional &s)
{ return racional(num * s.num, den * s.den); }
```

- Operador * sobrecargado como **función global**:

```
racional operator*(const racional &r,
        const racional &s)
{ return racional(r.numerador() * s.numerador(),
   r.denominador() * s.denominador()); }
```



Sobrecarga de operadores(V)

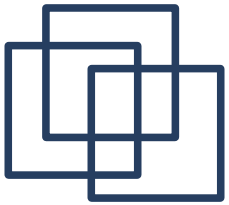
De igual forma,

- Operador ++ postfijo sobrecargado como función miembro:

```
Fecha & Fecha::operator++(int)
```

- Operador ++ postfijo sobrecargado como función global:

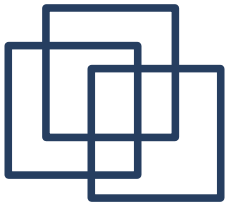
```
Fecha & operator++(Fecha&, int);
```



Operador de asignación (I)

- Es un operador especialmente importante porque permite asignar valores de la clase

```
clase& clase::operator=(const clase &valor)
{
  if (this!=&valor) {
    // se asigna sólo si no hacemos x=x
     // se “elimina” el contenido de *this
     ...
  }
  return *this;
}
```



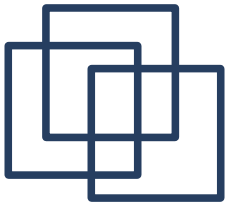
Operador de asignación (II)

- Se diferencia de un constructor en que el objeto ya debe estar creado.
  - Constructor

```
clase x = y; // se construye x como una copia de Y
clase z(y); // otra forma de hacer lo mismo
```

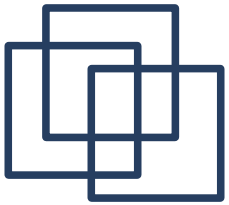
  - Operador de asignación

```
clase x;
x = y; // el objeto x ya existe y se le asigna y
```



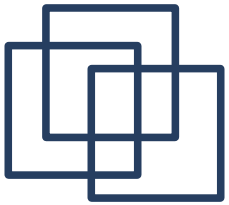
Operador de asignación (III)

```
array& array::operator=(const array& v)
{
  if (this!=&v) {
      this->destruir();
      this->copiar(v);
  }
  return *this;
}
```



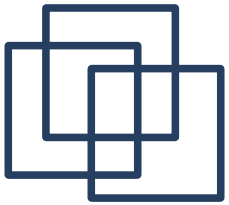
Operador de asignación(IV)

```
void array::destruir() // Método privado
{
  delete [] datos;
}
void array::copiar(const array &v) // Método privado
{
  num= v.num; datos = new int[num];
  for (int i= 0; i<num; i++)
    datos[i] = v.datos[i];
}
```



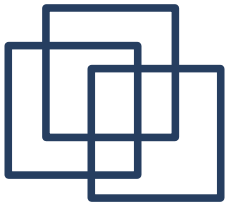
Constructor de copia y destructor

```
array::array(const array &v)
{
  this->copiar(v);
}
array::~array()
{
  this->destruir();
}
```



Ejemplo

Creación de una clase *racional*, que permita almacenar y manejar números racionales.



Ejemplo: clase racional (I)

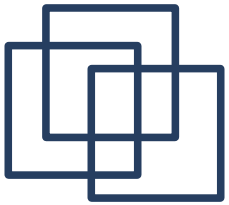
```
class racional {
public:
```

ACCESIBLE DESDE EL EXTERIOR

```
private:
```

OCULTO AL EXTERIOR

```
};
```

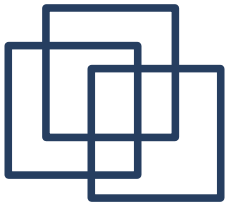


Ejemplo: clase racional(II)

```
class racional {
public:
```

ACCESIBLE DESDE EL EXTERIOR

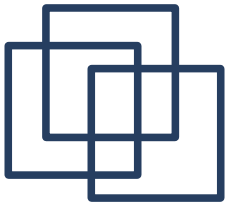
```
private:
  int num;
  int den;

};
```



Ejemplo: clase racional(III)

```
class racional {

public:
  racional();
  racional(const racional &g);
  racional(int n, int d);


private:
  int num;
  int den;
};
```

CONSTRUCTORES



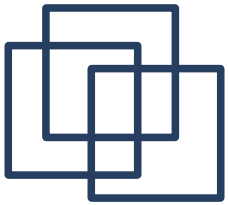
Ejemplo: clase racional(IV)

```
racional::racional() {
    num = 0;
    den = 1;
}

racional::racional(int n, int d) {
    num = n;
    den = d;
}

racional::racional(const racional &g) {
    num = g.num;
    den = g.den;
}
```

CONSTRUCTOR POR DEFECTO

CONSTRUCTOR

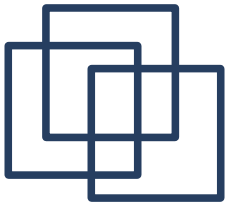
CONSTRUCTOR DE COPIA



Ejemplo: clase racional (V)

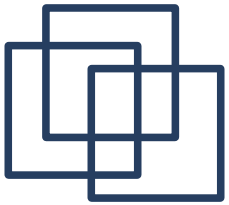
```
class racional {

public:
  racional();
  racional(const racional &g);
  racional(int n, int d);
  int numerador() const;
  int denominador() const;

private:
  int num;
  int den;

};
```

FUNCIONES DE ACCESO



Ejemplo: clase racional(VI)

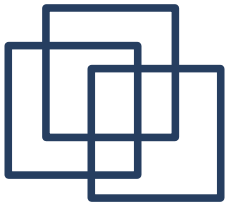
```
int racional::numerador() const {
    return num;
}

int racional::denominador() const {
    return den;
}
```



Ejemplo: clase racional(VII)

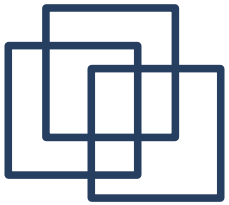
```
class racional {
public:
  racional();
  racional(const racional &g);
  racional(int n, int d);
  int numerador() const;
  int denominador() const;
  racional operator*(const racional &g) const;
  bool operator<(const racional &g) const;
  racional & operator=(const racional &g);

private:
  int num;
  int den;

};
```

OPERADORES



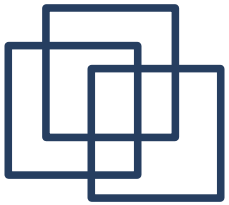
Ejemplo: clase racional(VIII)

```
racional racional::operator*(const racional &g)
    const {
    return (racional (num * g.num,
        den * g.den));
}
bool racional::operator<(const racional &g) const {
    return ((num * g.den) < (den * g.num));
}
racional & racional::operator=(const racional
    &g) {
    num = g.num;
    den = g.den;
    return *this;
}
```



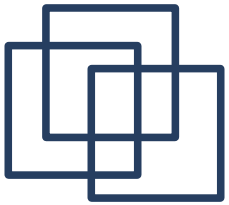
Ejemplo: clase racional(IX)

```
class racional {
public:
  racional();
  racional(const racional &g);
  racional(int n, int d);
  int numerador() const;
  int denominador() const;
  racional operator*(const racional &g) const;
  bool operator<(const racional &g) const;
  racional & operator=(const racional &g);
private:
  int num;
  int den;
};
ostream & operator<<(ostream &f,
          racional g);
```



Ejemplo: clase racional(X)

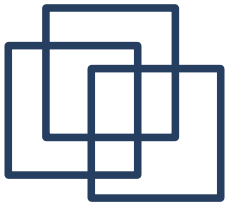
```
ostream & operator<<(ostream &f, racional g) {
    return (f << g.numerador() << "/" <<
        g.denominador());
}
```



Ejemplo: clase racional(XI)

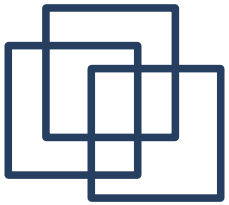
```
class racional {
public:
  racional();
  racional(const racional &g);
  racional(int n, int d);
  int numerador() const;
  int denominador() const;
  racional operator*(racional g) const;
  bool operator<(racional g) const;
  racional & operator=(const racional &g);
 ~racional();
private:
  int num;
  int den;
};
```

} **DESTRUCTOR**



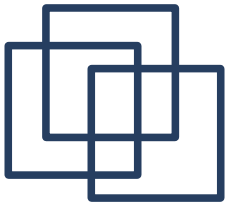
Ejemplo: clase racional(XII)

```
// Está vacío porque no hay que hacer nada
// especial para destruir un racional
racional::~racional() {
}
```



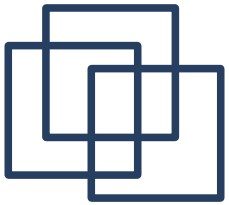
Ejemplo: racional.hpp(XIII)

```
#include <iostream>

class racional {
public:
  racional();
  racional(const racional &g);
  racional(int n, int d);
  int numerador() const;
  int denominador() const;
  racional operator*(const racional &g) const;
  bool operator<(const racional &g) const;
  racional & operator=(const racional &g);
 ~racional();
private:
  int num;
  int den;
};
using namespace std;
ostream & operator<<(ostream &f, racional g);
```



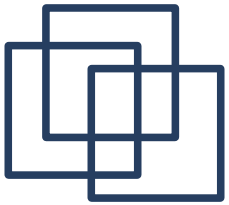
Ejemplo: racional.cpp(XIV)

```
#include "racional.h"

racional::racional() {
    num = 0;
    den = 1;
}

racional::racional(int n, int d) {
    num = n;
    den = d;
}


racional::racional(const racional &g) {
    num = g.num;
    den = g.den;
}
```



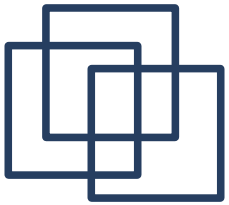
Ejemplo: racional.cpp (XV)

```
racional racional::operator*(const racional &g) const {
    return racional(num * g.num, den * g.den);
}

int racional::numerador() const {
    return num;
}

int racional::denominador() const {
    return den;
}

bool racional::operator<(const racional &g) const
{
  return (num * g.den) < (den * g.num);
}
```



Ejemplo: racional.cpp (XVI)

```
racional & racional::operator=(const racional &g)
{
    num = g.num;
    den = g.den;
    return *this;
}

ostream & operator<<(ostream &f, const racional &g) {
    return (f << g.numerador() << "/" <<
       g.denominador());
}

racional::~racional() {
}
```

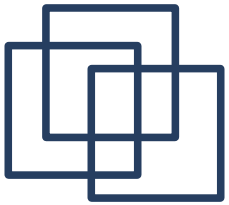


Generalización o parametrización de tipos

Consiste en introducir un parámetro en la definición del tipo de uno o varios de sus componentes.

Ejemplos:

```
vector<int> v;
queue<char> c;
```

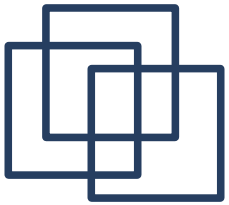


Parametrización de tipos en C++ (I)

- El mecanismo que ofrece C++ para parametrizar tipos es el uso de template de clases.
- Declaración usando template:

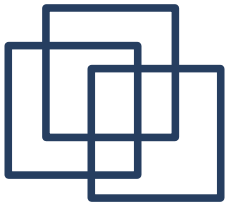
```
template < parámetros > declaración
```

- Los parámetros de la declaración genérica pueden ser:
  - **typename** identificador. Se instancia por un tipo de dato.
  - **tipo-de-dato** identificador. Se instancia por una constante.



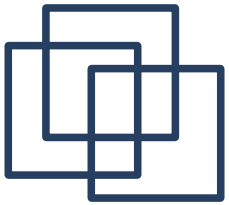
Parametrización de tipos en C++ (II)

```
template<typename T, int n>
class array_n {
private:
  T items[n];
};
array_n<complejo,1000> w;
```



Parametrización de tipos en C++ (III)

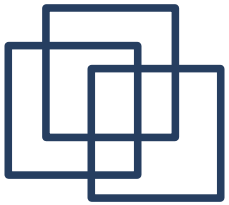
```
template<typename Tipo>
class vector_dinamico {
public:
  vector_dinamico(int tam = 100)
  { datos = new Tipo[tam];
    ndatos = tam;
  }
private:
  Tipo *datos;
  int ndatos;
};
```



Funciones miembro con templates

Definición de las funciones miembros fuera de la clase

```
template <typename T>
T vector_dinamico<T>::componente(int i) const
{
  return datos[i];
}
```



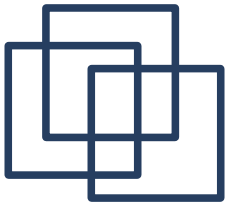
Uso de clases con templates

- La instanciación sería la siguiente:

```
vector_dinamico<int> vI;
vector_dinamico<float> vf;
```

- El compilador genera una versión de la clase para cada instanciación

El código de la clase genérica es necesario
en tiempo de compilación



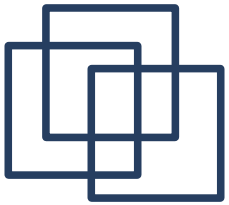
Implementación de clases genéricas (I)

vector_dinamico.hpp

```
#ifndef
__VECTOR_DINAMICO_H__
#define
__VECTOR_DINAMICO_H__
template <typename T>
class vector_dinamico {
   vector_dinamico (int n);
   ...
};
#include "vector_dinamico.hxx"
#endif
```

vector_dinamico.hxx

```
template <typename T>
vector_dinamico<T>::vector_dinamico (int
n)
{
  ...
}
...
```



Implementación de clases genéricas (II)

```
#include "vector_dinamico.hpp"
...
int main()
{
  vector_dinamico<int> vect_enteros(22);
  vector_dinamico<string>vect_cadenas(100);
  ...
}
```

ejemplo.cpp

- Los ficheros .hxx NO se compilan, sólo habría que hacer:

```
g++ -o ejemplo ejemplo.cpp
```