

# Programación en C++ (3) : introducción a clases

Dr. J.B. Hayet

CENTRO DE INVESTIGACIÓN EN MATEMÁTICAS

Septiembre 2009



# The lands of C++



J.B. Hayet

Programación, Septiembre 2009

# The lands of C++



# Outline

- 1 Esconder elementos de un objeto
- 2 Clases
- 3 Constructores y destructores
- 4 Inicializadores de estructuras/clases
- 5 Overloading de funciones



# Outline

- 1 Esconder elementos de un objeto
- 2 Clases
- 3 Constructores y destructores
- 4 Inicializadores de estructuras/clases
- 5 Overloading de funciones



# Previously in PA-I...

Hemos visto que en C++, se puede usar las estructuras (struct) de una manera diferente, con rumbo **Programación orientada objetos**.

- La diferencia esencial : se puede **integrar funciones específicas a una estructura dentro de aquella.**
- Esas funciones (**métodos**) “ven” las variables que constituyen los datos en la estructura.
- Esas funciones pueden tener **acceso al apuntador al objeto que les llama**, gracias al apuntador **this**.
- Asociación conceptual llevada a asociación literal, con, abajo, la capa de funciones C.



# Previously en la clase...

Límite : el “cliente” de tu aplicación tiene acceso a todo lo que compone la estructura ya que le tienes que entregar un .h y una librería. ¿Cómo esconder elementos o al menos **prohibir acceso a unos de ellos** ?



# Esconder elementos...

- Hay **elementos internos** que el cliente no tiene que manipular porque puede cambiar el comportamiento del programa.
- **Cambios de implementación** : eventualmente vamos a cambiar los algoritmos o optimizar código; eso no debe de hacer que el cliente reescriba todo su propio código.
- Este código “inestable”, mejor dejarlo **inaccesible**.

Ejemplo : podrías querer renombrar el elemento width de la estructura Image en w.



# Controlar el acceso

El C++ introduce palabras llaves que definen **niveles de acceso** para los elementos y las funciones específicas a esas clases :

- **private** : acceso únicamente posible dentro de **métodos de la estructura**.
- **protected** : igual que private pero las clases **hijas** también pueden acceder al elemento.
- **public** : acceso **público**.

pero hay excepciones (los friend).



# En una estructura...

Todos los elementos (datos/métodos) son **public por default** :  
pueden ser accesible en todas partes de un código manipulando esas  
estructuras (para conservar la **compatibilidad** con C).

# Ejemplo...

Restringiendo el acceso a los elemento de Image:

```
struct Image {  
private:  
    int width;  
    int height;  
public:  
    int initialize(int w,int h,  
                  int channels,int depth);  
    int freeImage();  
    int copyFrom(const Image *imgsrc);  
    Image* clone();  
    Image *colorToGrey();  
    ...  
};
```



# Ejemplo...

```
int main() {  
    Image img;  
    int w=img.width;  
}
```

error: 'int Image::width' is private



# En una estructura...

- Todos los elementos son **visibles en los métodos** : las palabras llaves de acceso se aplican para **usuarios** de las estructuras (los que instancian objetos de este tipo).
- Todos los elementos (datos/métodos) que siguen una palabra llave de control de acceso están afectados por ese control.
- Puedo **usarlas varias veces** en la misma estructura.



# Ejemplo...

Restringiendo el acceso a los elemento de Image:

```
struct Image {  
private:  
    int width; // Private  
public:  
    int initialize(int w,int h,  
                   int channels,int depth);  
private:  
    int height; // Private  
public:  
    int freelmage(); //Public  
    int copyFrom(const Image *imgsrc);  
    Image* clone();  
    ...  
};
```



# El efecto : barrera de acceso a estructuras

Compilando un archivo main.cpp :

```
Image img;  
...  
cout << img.width;
```

main.cpp:7: error: 'int Image::width' is private

No me deja invocar este elemento.

```
Image img;  
...  
Image *newimg = img.clone();
```

OK



# Orden

No hay ninguna regla de orden de los elementos en función del tipo de acceso ; sin embargo, por razones de legibilidad, **se preferirá poner primero los elementos de tipo public**, que normalmente son los que queremos conocer en primer lugar (a priori, lo demás no nos interesa).

# Orden

```
struct Image {  
public:  
    int initialize(int w, int h,  
                  int channels, int depth);  
    int freemage();  
    int copyFrom(const Image *imgsrc);  
    Image* clone();  
    Image *colorToGrey();  
    ...  
private:  
    int width;  
    int height;  
    ...  
};
```



# Hacer excepciones al control de acceso

Para añadir un poco de **flexibilidad** a este sistema rígido, existe la posibilidad de otorgar acceso explícito a los elementos de una estructura; se hace con la palabra **llave friend**.

Sólo la estructura en que están los elementos a que dar acceso puede otorgar este derecho.

Acceso otorgado a **funciones** globales, **métodos** de otras estructuras, o otras **estructuras** enteras.



# friend: funciones globales

```
struct Image {  
public:  
    int initialize(int w,int h,  
                  int channels,int depth);  
    int freelmage();  
    friend int main(); // Mi main podra leer  
                      // elementos private  
    ...
```



# friend: funciones globales

```
int main() {  
    Image img;  
    int w=img.width;  
    ...  
}
```

Esta vez, compila.



# friend: métodos de otras estructuras

```
struct DerivativeOperator {  
    bool apply(Image *img);  
};  
struct Image {  
    ...  
    friend bool DerivativeOperator::apply(Image *img);  
    ...  
};
```

Permite :

```
bool DerivativeOperator::apply(Image *img) {  
    int w = img->width; // No se podría si no fuera friend  
    ...  
}
```



# friend: otras estructuras

```
struct DerivativeOperator {  
    bool apply(Image *img);  
};  
struct Image {  
    friend struct DerivativeOperator;  
};
```

Todos los métodos de DerivativeOperator podrán tener acceso a los elementos private.



# friend: problema de visibilidad

Si declaro :

```
struct DerivativeOperator {  
    bool apply(Image *img);  
};  
struct Image {  
    friend struct DerivativeOperator;  
};
```

DerivativeOperator no conoce Image todavía : la declaración viene después !

```
main.cpp: error: 'Image' does not name a type
```



# friend: problema de visibilidad

Una solución :

- Existe una manera de hacer una “declaración de existencia” de una estructura, sin detallar su contenido :
- Notar que eso solo permite usar apuntadores (o referencias):

```
struct Image; // Declaracion de existencia
struct DerivativeOperator {
    Image imgTmp;
    ...
};

struct Image {
    friend struct DerivativeOperator;
};
```

No funciona :

```
main.cpp: error: field 'imgTmp' has incomplete type
```



# friend: problema de visibilidad

Otra solución que funciona con objetos:

```
struct Image {  
    friend struct DerivativeOperator;  
};  
struct DerivativeOperator {  
    Image imgTmp;  
};
```

# friend: estructuras anidadas

Una estructura definida **dentro de otra** puede ver todos los elementos de la en que esta definida ? La respuesta es **sí**, por *default*, pero desde hace poco tiempo (antes era no).



# friend: estructuras anidadas

```
struct Image {  
private:  
    int width;  
    int height;  
public:  
    struct rgbPixel {  
        unsigned char r,g,b;  
        rgbPixel invert();  
        int f(Image *img);  
    };  
    ...  
};
```

Notar que los métodos de `rgbPixel` no son métodos de `Image` (y viceversa).



# friend: estructuras anidadas

Antes :

The members of a nested class have no special access to members of an enclosing class, nor to classes or functions that have granted friendship to an enclosing class; the usual access rules (clause 11) shall be obeyed.

The members of an enclosing class have no special access to members of a nested class; the usual access rules (clause 11) shall be obeyed."

Se necesitaba hacer friend de los métodos de las clases anidadas.



# friend: estructuras anidadas

Ir a ver (punto 45.):

[http://anubis.dkuug.dk/jtc1/sc22/wg21/docs/cwg\\_defects.html#45](http://anubis.dkuug.dk/jtc1/sc22/wg21/docs/cwg_defects.html#45)

En 2001 se propuso cambiar por :

A nested class is a member and as such has the same access rights as any other member.

Como que el lenguaje **evoluciona** a través de discusiones. . .



# friend: estructuras anidadas

```
struct Image {  
private:  
    int width;  
    int height;  
public:  
    struct rgbPixel {  
        private:  
            unsigned char r,g,b;  
        public:  
            int f(Image *img);...  
    };
```

Permite :

```
int Image::rgbPixel::f(Image *img) {  
    int w = img->width; // Antes no, ahora si  
}
```



# friend: estructuras anidadas

En cambio, la clase al nivel superior o otras clases anidadas **no** pueden acceder a los elementos `private` de una clase anidada :

```
int Image::g(rgbPixel &color) {  
    unsigned char r = color.r; // NO  
}
```

... al menos que se le otorgue una friendship



# friend: orientado objeto ?

- Rompemos con el paradigma OO : se puede entrelazar las clases, que no son tan autónomas.
- + Puede facilitar la vida del programador (los conceptos cubiertos por las clases pueden tener mucho en común sin que hayan relaciones de herencia).

C++ (como en muchos otros aspectos) eligió la vía de pragmatismo.



# Layout de una estructura

Con una estructura C, podemos saber de manera mas o menos cierta como es la organización de los octetos en memoria; con estructuras (o clases), normalmente es igual (sin control de acceso) pero en general lo único seguro es que las variables correspondiendo a un grupo compartiendo los mismos controles de acceso serán en espacios contiguos de la memoria.

Lo demás dependerá del compilador... No hacer razonamientos basado en un layout “a la C” a partir del momento que aparece el control de acceso !



# Outline

- 1 Esconder elementos de un objeto
- 2 Clases
- 3 Constructores y destructores
- 4 Inicializadores de estructuras/clases
- 5 Overloading de funciones



# Estructuras : un balance

- Partimos de **estructuras C**, que no son mas que agrupamientos útiles de datos contiguos en memoria.
- Se añadió la posibilidad de definir funciones específicas a esas estructuras, o **métodos**, en cuyo código se ve todos los elementos de la estructura.
- Se añadieron **niveles de acceso** a los datos para restringir explícitamente a una variable o una función que por naturaleza son internas.



# Clase

Es casi exactamente lo mismo que una estructura “a la C++”

La única diferencia es que por *default*, los elementos componentes de la estructura son *private*.

Entonces, ¡ya casi saben todo!



# Clase : ejemplo

Podemos muy fácilmente transformar la estructura Image en clase :

```
class Image {  
public:  
    int initialize(int w,int h,  
                  int channels,int depth);  
    int freeImage();  
    int copyFrom(const Image *imgsrc);  
    Image* clone();  
    Image *colorToGrey();  
    ...  
private:  
    int width;  
    int height;  
    ...  
};
```

# Clase : estructura anidada

Igual (y se pudo hacerlo con clases y struct):

```
class Image {  
    int width; // Private  
    int height;  
public:  
    class rgbPixel {  
        unsigned char r,g,b; // Private  
        public:  
        rgbPixel invert();  
        int f(Image *img);  
    }  
    ...  
};
```



# Clase : esconder lo private

Tal cual no alcanzamos una separación total entre interfaz e implementación : se puede leer “en claro” detalles de la implementación que nos gustaría esconder :

- Esos elementos private pueden ser reveladores de unas **elecciones algorítmicas que hay que proteger** (por ejemplo por razones estratégicas, o por brevet).
- Gracias a la descripción detallada de la clase/estructura, se puede, por medio de apuntadores, **acceder a los elementos private !**
- Hay que darle al cliente un código accesible (interfaz) **estable** y al mismo tiempo quedarse con la posibilidad de cometer errores de implementación, de hacer cambios aun fundamentales.



# Clase : esconder lo private

Otro elemento es que al querer cambiar cosas en la declaración de tu clase (por ejemplo al cambiar variables private, que solo son relacionadas a la implementación), **la recompilación implicará a todas las clases que usan esa clase** de otra manera que con simple apuntador (porque los tamaños reservados en memoria tendrán que ser diferentes).

¡ Perdida de tiempo ! (cuando el proyecto tiene gran dimensión)



# Clase : esconder lo private

Una manera de hacerlo : esconder la estructura “verdadera” en un archivo .cpp (que no será disponible a los clientes) y **usar una estructura intermedia, basada en la primera** (a través de un apuntador) que contiene todas los métodos públicos (método del *Cheshire cat*).

Reservar a situaciones criticas (proyectos grandes, gran necesidad de esconder tus elementos private)



# Clase : esconder lo private

En imageHandle.h :

```
#ifndef IMAGE_HANDLE_H
#define IMAGE_HANDLE_H

class ImageHandle {
    struct Image; // Declaracion de existencia
    Image* imgInternal;
public:
    void initialize(int w,int h,int ch,int d);
    void clean();
};

#endif
```



# Clase : esconder lo private

La declaración

`struct Image; // Declaracion de existencia`  
es muy importante, sino, no se puede definir el apuntador.



# Clase : esconder lo private

En imageHandle.cpp :

```
void ImageHandle::initialize(int w,int h,  
                           int ch,int d) {  
    imgInternal = new Image;  
    imgInternal->initialize(w,h,ch,d);  
    ...  
}
```



# Clase : esconder lo private

Y también en el imageHandle.cpp :

```
class Image {  
public:  
    int initialize(int w,int h,  
                  int channels,int depth);  
    int freelmage();  
    ...  
private:  
    int width;  
    int height;  
    ...  
};
```



# Outline

- 1 Esconder elementos de un objeto
- 2 Clases
- 3 Constructores y destructores
- 4 Inicializadores de estructuras/clases
- 5 Overloading de funciones



# Clase : inicialización y limpieza

Entre los problemas mas frecuentes al usar estructuras o clases, especialmente cuando se maneja apuntadores dentro son la inicialización de la clase y su limpieza.

Típicamente es necesario implementar dos métodos específicos :

```
int initialize(int w, int h,  
               int channels, int depth);  
int freelmage();
```

y definir un *flag* como miembro private para saber si la inicialización ha sido hecha o no. (initDone)



# Clase : inicialización y limpieza

El problema es que ese **deja la responsabilidad al usuario final de llamar al método de inicialización y el de limpieza** : puede ser fuente de peligros, y en todos casos no es muy practico para el usuario. Mejor sería si se podía hacer eso automáticamente.



# Clase : constructor

El C++ introduce el **constructor** que permite hacer eso : es una función propia a estructuras/clases que, cuando esta definida, esta llamada **automáticamente** por el programa al crear una instancia de esta clase/estructura.

Esta función **lleva como nombre el nombre de la estructura**, y sus argumentos pueden estar definidos como lo quieras.



# Clase : constructor

Ejemplo :

```
class Image {  
    int width;  
    int height;  
public:  
    Image(int w,int j,int ch, int d); // A constructor  
    Image(); // Another (default !) constructor  
}  
...  
};
```



# Clase : constructor

Ejemplo :

```
Image::Image( int w, int j , int ch , int d ) {  
    width   = w;  
    height  = h;  
    data    = new unsigned char [w*h*ch*d];  
    ...  
};  
};
```

# Clase : constructor

El constructor **no es obligatorio**, y en el caso de que no defines uno, el programa definirá el mismo un constructor por default (sin argumentos).

Ahora, si tú defines un(os) constructor(es) con un cierto tipo de argumentos, entonces solo podrás usar ese(os) constructor(es) (o los que defines ademas).



# Clase : constructor

En Image.h:

```
class Image {  
public:  
    Image(int ,int ,int ,int );  
    ...  
};
```

En prog.cpp

```
int main() {  
    Image img;  
    ...  
}
```

No funcionará (mientras que sí si no defino constructor)!



# Clase : constructor

```
main.cpp:27: error: no matching function
          for call to 'Image::Image()'
main.cpp:20: note: candidates are: Image::Image(int, int, int)
main.cpp:7:   note:           Image::Image(const Image&)
```

# Clase : constructor por default

Es un constructor que no toma argumento,

```
class Image {  
public:  
    Image();  
...  
};
```

Si no hay constructor, el compilador **creará un constructor por default**; si has definido un constructor pero no constructor por default, no será posible invocar :

```
Image imgArray [20];
```

o

```
Image img;
```



# Clase : destructor

Paralelamente al constructor, el lenguaje C++ preve **una función especial para encargarse de los trabajos de limpieza al momento de la destrucción de la estructura** (liberación de los espacios en memoria, de la pila, automáticamente, o del montículo, por un free) : es el **destructor**

Este destructor se identifica por :

`~Clase();`

Donde Clase es el nombre de la clase.



# Clase : destructor

Ejemplo :

```
Image::~Image() {  
    if (imageData!=NULL)  
        delete [] imageData;  
};
```

# Clase : destructor

Es importante ver que está **llamado automáticamente** para toda clase de objetos, cada vez que se sale del scope donde esta definido el objeto (con unas excepciones...).



# Clase : constructor y destructor

- No tienen valor de regreso (son especiales).
- El constructor puede existir en varias versiones (sobrecargar).

# C vs. C++ : ¿ dónde declarar variables ?

Dos filosofías diferentes en cuanto a la definición/inicialización de las variables : en C++, se **inicializan al mismo tiempo que se definen** (a través del constructor); entonces, la filosofía del C inicial (todas las declaraciones primero) ya no se puede aplicar : eventualmente las inicializaciones dependen de valores de variables calculadas dentro del código.



# C vs. C++ : variables

En C (al menos las primeras versiones) :

```
int a, b, c; //  
...  
c = a+b;
```

Declaraciones primero; luego inicializaciones.



# C vs. C++ : variables

En C++ :

```
int w=...;  
...  
int h(a*2);  
...  
Image img(w,h,3,1);
```

Declaraciones/inicializaciones al mas cercano de su utilización !



# C vs. C++ : variables

El caso mas tipico de esa filosofía : los ciclos for

```
for (int i=0;i<10;i++) {  
    ...  
}  
cout << i << endl; // NO !
```



# C vs. C++ : variables

Ahora, bien pensar que hay dos cosas involucradas

- alocación de memoria (generalmente al principio de un bloque)
- inicialización del objeto por la llamada al constructor



# C vs. C++ : variables

El C++ fuerza a que ambas tengan lugar y en particular que el constructor no esté separado del uso de la estructura (por condicionales por ejemplo).

```
switch (b) {  
    case true:  
        Image img1;  
        break;  
    case false:  
        Image img2;  
        break;  
}
```

NO compila



# C vs. C++ : variables

En cambio,

```
switch (b) {
    case true: {
        Image img1;
    }
    break;
    case false: {
        Image img2;
    }
    break;
}
```

Sí compila



# Outline

- 1 Esconder elementos de un objeto
- 2 Clases
- 3 Constructores y destructores
- 4 Inicializadores de estructuras/clases
- 5 Overloading de funciones



# Inicialización de arreglos en C

Se recordarán que podemos inicializar un arreglo en C de esta manera :

```
double arr[10]={1.0, -2.0, 3.0, 1.0};
```

Inicializa el arreglo **con las valores pasadas entre las llaves y completa por ceros.**

Alternativamente :

```
double arr[]={1.0, -2.0, 3.0, 1.0};
```

que toma el tamaño adecuado. Pero ¿cómo accedo al tamaño de arr después ?



# Inicialización de arreglos en C

Acceder al tamaño definido enteramente por su inicialización :

```
double arr []={1.0, -2.0, 3.0, 1.0};  
int tamArr = sizeof(arr)/sizeof(arr[0]);
```

# Inicialización de estructuras en C

Como una estructura en C es nada mas que un “arreglo” heterogéneo :

```
typedef struct ImageStr {
    int width;
    int height;
    char *data;
} Image;
int main() {
    Image img = {100,100,NULL};
    int w = img.width;
    return 0;
}
```



# Inicialización de estructuras en C

Hasta se puede inicializar arreglos de estructuras :

```
Image img[3] = {{100,100,NULL},{200,200,NULL}};
```

En el mismo espíritu que la inicialización de arreglos dobles. El img[2] esta inicializado a 0.

# Inicialización de estructuras en C++

Las cosas son diferentes :

1. Si hay uno de los elementos private

```
struct Image { // Same for class
private:
    int width;
    int height;
    char *data;
};

int main() {
    Image img = {100,100,NULL};
    return 0;
}
```



# Inicialización de estructuras en C++

```
test.cpp: In function 'int main()':  
test.cpp:12: error: 'img' must be initialized by constructor,  
not by '{...}'
```

Necesita usar un constructor, nos dice !



# Inicialización de estructuras en C++

Las cosas son diferentes :

2. Si existe ya un constructor

```
struct Image {  
public:  
    int width;  
    int height;  
    char *data;  
    Image(int ,int );  
};  
int main() {  
    Image img = {100,100,NULL};  
    return 0;  
}
```

Nos da el mismo error.



# Inicialización de estructuras en C++

En este caso, sólo se puede pasar por el constructor. Pero todavía podemos inicializar **arreglos** de estructuras de la manera precedente, adaptada a los constructores :

```
Image img[3] = {Image(100,100), Image(200,200),  
                 Image(200,200)};
```

Remarcar que se puede necesitar absolutamente el constructor por default, como en el caso siguiente :

```
Image img[4] = {Image(100,100), Image(200,200),  
                 Image(200,200)};
```

# Outline

- 1 Esconder elementos de un objeto
- 2 Clases
- 3 Constructores y destructores
- 4 Inicializadores de estructuras/clases
- 5 Overloading de funciones



# Overloading

El sobrecargo, o *overloading* es un concepto fundamental en C++ : permite usar nombres de funciones de maneras multiformes.

```
int draw(Image *img, Rectangle *r);
int draw(Image *img, Point *p);
```

No es autorizado por C...

```
testo.c:21: error: conflicting types for 'draw'
testo.c:20: error: previous declaration of 'draw' was here
```



# Overloading

Mientras no es problema compilarlo en C++. La existencia de esta técnica es también una **consecuencia de la existencia de constructores** : hay sólo un nombre para los constructores (el nombre de la clase) y podemos naturalmente querer hacer la inicialización de varias maneras. En C, serían :

```
int initializeBySize(Image *img, int w, int h);
int initializeByImg(Image *img, const Image *other);
```



# Overloading $\neq$ mismos nombres en diferentes namespace

Habíamos visto que se puede usar hasta prototipos iguales :

```
namespace miEspacioDeNombre {  
    int initializeBySize(Image *img, int w, int h);  
}  
int initializeBySize(Image *img, int w, int h);
```

En realidad, entre funciones de espacios de nombre diferentes, el nombre completo de la función sí es diferente :

```
int miEspacioDeNombre::initializeBySize(Image *img,  
                                         int w, int h);  
int initializeBySize(Image *img, int w, int h);
```



# Overloading

Queremos actuar en un espacio de nombre dado :

```
class Image {  
...  
public:  
int draw( Rectangle *r );  
int draw( Point *p );  
}
```

Y se podría eso ?

```
class Image {  
...  
public:  
bool draw( Point *p );  
int draw( Point *p );  
}
```



# Overloading : ¿de dónde sale la magía ?

En C:

```
//int draw(Image *img, Rectangle *r);  
int draw(Image *img, Point *p);  
  
int main() {  
    Image img;  
    Point p;  
    draw(&img,&p );  
    return 0;  
}
```



# Overloading : ¿de dónde sale la magia ?

En C: el resultado del comando nm prog.o (nm es un utilitario para ver tablas de simbolos)

```
U _draw
00000000 T _main
```

El símbolo objeto correspondiendo a una función es el nombre de la función y ya !



# Overloading : ¿de dónde sale la magia ?

En C++ (el mismo programa):

```
00000020 s EH_frame1
          U __Z4drawP8ImageStrP8PointStr
          U ___gxx_personality_v0
00000000 T _main
0000003c S _main.eh
```

El símbolo objeto conlleva también los argumentos ! El compilador transforma los nombres de función : con los **espacios de nombre Y con los argumentos** (entonces no hay ambigüedades).

# Overloading : ¿de donde sale la magia ?

Consecuencias :

- El *overloading* es posible.
- El link de las funciones es **mas estricto**.

En file1.c :

```
void print(char c) {};
```

En file2.c :

```
void print(int c);
```

...

```
print(200);
```

OK en C (aunque lleva errores de calculo !) pero no se puede ligar en C++

```
/usr/bin/ld: Undefined symbols:  
print(int)
```



# Overloading : un caso en que son necesarios

Uniones :

En C

```
union U {  
    int i;  
    float f;  
};
```

En C++ incluyen control de acceso, métodos :

```
union U {  
private:  
    int i;  
    float f;  
public:  
    U(int a);  
    U(float b);  
};
```



# Overloading : un caso en que son necesarios

Uniones : Para poder aprovechar de la existencia de uniones, debemos poder inicializarlos de las diferentes maneras que nos permite la unión (entonces el *overloading* es necesario)

```
union U {  
private:  
    int i;  
    float f;  
public:  
    U(int a);  
    U(float b);  
    int getInt();  
    float getFloat();  
};
```

Otra manera (sobrecargada) de recuperar el entero o el flotante ?



# Remarcas sobre uniones en C++

- Mejor usarlas **internamente** a estructuras/clases (ya que de todos modos no permiten herencia).
- Para asegurarse que el usuario usa el buen elemento dentro de una unión, esconderla (hacerla private) dentro de una clase/estructura y, por ejemplo, emparejarla con un enum (aquí : entero/floatante)



# Remarcas sobre uniones en C++

```
class U {  
    enum {  
        character,  
        integer,  
        floating_point  
    } vartype; // Define one  
    union { // Anonymous union  
        char c;  
        int i;  
        float f;  
    };  
public:  
    U(char ch);  
    U(int ii);  
    U(float ff);  
};
```



# Remarcas sobre uniones en C++

El C++ permite ver los elementos en el scope corriente como si fueran variables :

```
union {
    int i;
    float f;
};
i = 12;
f = 1.22;
```

