Programación en C++ (2)

Dr. J.B. Hayet

CENTRO DE INVESTIGACIÓN EN MATEMÁTICAS

Septiembre 2009

Outline

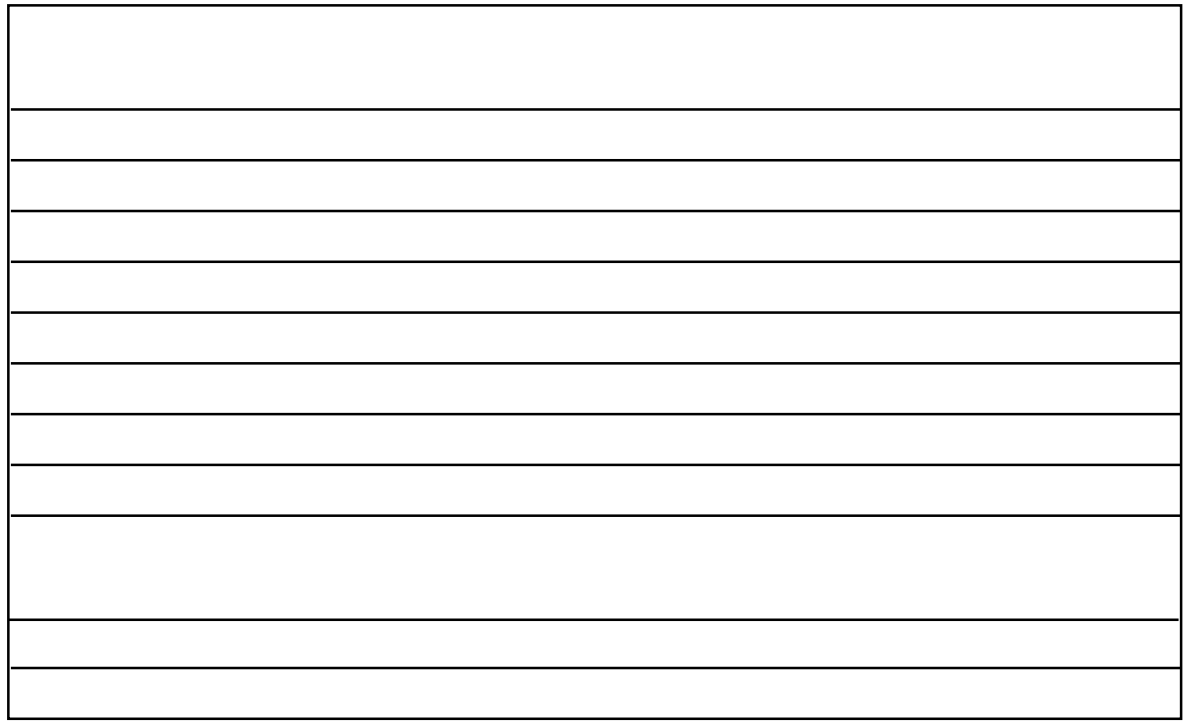
Outline

Memoria

Como nosotros vemos
la memoria :

1 byte

Memoria de 512 Mo



Variables en memoria

En C++ guardamos la misma organización de las variables en memoria :

- Variables locales
- Variables globales
- Variables estáticas
- Datos alocados dinámicamente

Variables en memoria

En C++ guardamos la misma organización de las variables en memoria :

Variables locales — M. PILA
Variables globales
Variables estáticas
Datos alocados dinámicamente

Variables en memoria

En C++ guardamos la misma organización de las variables en memoria :

Variables locales	M. PILA
Variables globales	M. ESTATICA
Variables estáticas	M. ESTATICA
Datos alocados dinámicamente	

Variables en memoria

En C++ guardamos la misma organización de las variables en memoria :

Variables locales	M. PILA
Variables globales	M. ESTATICA
Variables estáticas	M. ESTATICA
Datos alocados dinámicamente	M. MONTICULO

Variables globales

Se pueden acceder desde otros archivos compilados en la misma aplicación, con la palabra llave extern:
En file1.cpp :

```
int myGlobalVariable;
```

En file2.cpp :

```
extern int myGlobalVariable;
```

Variables globales

Cuando hay conflicto entre nombres de variables locales y de variables globales, a priori es la variable local que es considerada. Para utilizar explícitamente la variable global, se puede usar la notación ::var (también es valable para funciones/métodos, dentro de clases):

```
int x=5;
int func(int l) {
        int x;
        x    = l;
        ::x = l+1;
}
```

Alocación dinámica

- Recuerdo : la alocación dinámica permite reservar pedazos de memoria en una zona especifica de la memoria (el montículo, o *heap*), al momento de la ejecución, según las necesidades.
- Espacio reservado a través de un apuntador inicializado por funciones como malloc.
- Espacio no liberado automáticamente (como lo de la pila) : uso de funciones explícitas de liberación, como free.

Alocación dinámica en C

Típicamente, para un arreglo de 100 enteros :

```
int isize = 100;
int *i_ptr = (int *)malloc(isize*sizeof(int));
...
free(i_ptr);
```

free necesario para evitar fugas de memoria.

Alocación dinámica en C++

Otro sistema para la alocación dinámica, tal vez mas intuitivo que usar, con las palabras llaves new y delete :

```
int *i_ptr = new int[isize];
...
delete []i_ptr;
```

delete necesario para evitar fugas de memoria

Alocación dinámica en C++

Más generalmente :

```
Tipo *var      = new Tipo;
Tipo *vararray = new Tipo[size];
```

Y dos sintaxis para liberar, segun que se hizo alocación memoria de objetos simples o de arreglos :

```
delete var;
delete [] vararray;
```

Alocación dinámica en C++

Remarca importante : como con las funciones malloc no olvidar de verificar el resultado de una llamada al operador new !

```
Tipo *var = new Tipo;
if (var==NULL) {
...
}
```

Alocación dinámica en C++

Remarca importante : cuidado a la diferencia entre

```
Tipo *var1  = new Tipo(size);
Tipo *var2  = new Tipo[size];
```

Regresaremos mas en detalles sobre lo del primer caso.

Alocación dinámica en C++

Remarca importante : el operador new no sólo efectúa la alocación dinámica, sino que también llama al constructor por default del objeto alocado, que puede inicializar los elementos del objeto (cuando son objetos instanciando unas clases dadas).

Alocación dinámica en C++ : inicialización

Al alocar objetos con un new, se puede invocar otros versiones del constructor :

```
double *var = new double(1.0);
if (var==NULL) {
        cerr << "Could not perform allocation"
             << endl;
        return false;
}
```

Alocación dinámica en C++

La definición de arreglos múltiples sigue igual,

```
int w = 100;
int h = 100;
int **doble = new int *[h];
if (doble==NULL) {
        return false;
}
for (int i=0;i<h;i++)
        doble[i] = new int[w];
doble[30][20] = 123;
...
for (int i=0;i<h;i++)
        delete []doble[i];
delete []doble;
```

Outline

Cast en C

La mayoría de las conversiones entre tipos son implícitas :

```
int func(double x) {
...
}
```

y se puede usar :

```
int a=17;
func(a);
```

Cast en C

Operación para hacer una conversión explícita cuando no lo está, o forzar una conversión que el compilador no acepta hacer

```
tipo1 a;
tipo2 b = (tipo2)a;
```

En particular sirven para las conversiones entre apuntadores :

```
int tab[10];
int *tab_ptr  = &tab[0];
char *ctab_ptr = tab_ptr;
```

```
error: cannot convert 'int*' to 'char*' in initialization
```

Cast en C

Se requiere en el caso de los apuntadores un cast explícito :

```
int tab[10];
int *tab_ptr  = &tab[0];
char *ctab_ptr = (char *)tab_ptr;
```

Cast en C++ : “a la C”

Se pueden usar exactamente como en C, con una forma especifica al C++ que se parece a un constructor :

```
int a=10;
float x1 = (float)a; // Como en C
float x2(a);         // Típico de C++
```

Limitaciones ?

Cast en C++ : “a la C”

Primera limitación : la misma notación puede de hecho hacer cosas muy diferentes, o aun varias cosas diferentes a la vez!

```
const char *cadena="Soy cadena de caracteres";
unsigned char *cadenaData=(unsigned char*)cadena;
```

Dos cosas pasan : primero, pasar de char * a unsigned char * y segundo, “quitar” el atributo const.

Cast en C++ : “a la C”

Segunda limitación : están un poco perdidos en el código, no se identifican particularmente bien; no se sabe si está hecho a propósito (cast implícitos).

Tercera limitación : se puede hacer cast de casi cualquiera cosa.

```
tipo *p=&x;
int n=(int)p;
```

Cast en C++ : formas explicitas

C++ introduce nuevas formas para el cast, que :

- Hace explícito el cast, permitiendo a todos lo programadores saber que esta por acá.
- Hace diferencia semántica entre diferentes tipos de cast.
- Añade soporte para cast dinámicos (verificación de los tipos al momento de la ejecución), que es algo particularmente interesante en POO.

Cast en C++ : formas explicitas

Cuatro formas de cast son introducidas :

- static_cast : estático (seguro si el programador es seguro de los tipos involucrados, en particular con apuntadores).
- dynamic_cast : dinámico (cuando uno no esta seguro del tipo de apuntador o de referencia, por ejemplo, hay que verificarlo al momento de la ejecución).
- const_cast : modifica las propiedades const y volatile.
- reinterpret_cast : lo mas libre (y peligroso).

static_cast

Uso :

```
tipo1 v1 = ...;
tipo2 v2 = static_cast<tipo2>(v);
```

Remarca : función que de hecho se presenta como template (con variaciones en función de un tipo-parametro).

static_cast

- Todo hecho al momento de la compilación a partir de los tipos involucrados.
- Puede implicar operaciones complejas (clases con herencia multiples...).
- No puede quitar las propiedades de constness.

Usar para conversiones implicitas, tipos de base.

static_cast

```
int entero;
long largo;
float flotante;
...
// Cast implicito, sin perdida
largo    = entero;
flotante = entero;
// Con static_cast
largo    = static_cast<long>(entero);
flotante = static_cast<float>(entero);
```

static_cast

```
// Cast con perdidas
entero = largo;
entero = flotante;
// Con static_cast
entero = static_cast<int>(largo);
entero = static_cast<int>(flotante);
```

static_cast

Idealmente, el static_cast aparecería en todos casos de conversiones implícitas y entre tipos básicos :

```
int func(int entero) {
        ...
}
...
// Conversiones implicitas
double doble = 0.0;
func(doble);                          // BUUUUH !
func(static_cast<int>(doble));        // WAAAAAA
```

static_cast

No permite cualquier cast, en particular para apuntadores

```
int v = 4;
float *d = static_cast<float*>(&j);
```

```
invalid static_cast from type 'int*' to type 'float*'
```

static_cast

```
// Apuntadores void* :
void* apGenerico = &entero;
float* apFlotante = (float*)apGenerico;
// Con static_cast
apFlotante = static_cast<float*>(apGenerico);
```

static_cast

```
struct A {
  double d;
};
struct B : public A {
  int i;
};
struct C {
  int i;
};
int main() {
  A a; B b;
  B *bptr = static_cast<B*>(&a);
  A *aptr = static_cast<A*>(&b);
  C *cptr = static_cast<C*>(&b);
  return 0;
}
```

static_cast

- Los dos primeros cast están aceptados por el compilador (¿por qué?)
- El tercero no:

```
invalid static_cast from type 'B*' to type 'C*'
```

const_cast

Este operador viene para convertir apuntadores hacia/de tipos const

```
const int entero = 0;
int* apEnteroConst = (int*)&entero; // C
apEnteroConst  = const_cast<int*>(&entero); // C++
```

P.D.: no confundir const tipo * y tipo * const

```
const int entero = 0;
int *const a = (int *const)&entero;
a  =(int *const)&entero;
*a = entero;
```

```
assignment of read-only variable 'a'
```

const_cast

```
const int* b = (const int*)&entero;
b =(const int* )&entero;
*b=entero;
```

```
assignment of read-only variable
```

reinterpret_cast

```
float f=10;
unsigned char *p =
        reinterpret_cast <unsigned char*> (&f);
for (int j=0; j<4; ++j)
  cout << p[j] << endl;
```

El reinterpret_cast permite reinterpretar los apuntadores. Evitarlo si posible (el menos seguro).

reinterpret_cast

* A pointer to any integral type large enough to hold it
* A value of integral or enumeration type to a pointer
* A pointer to a function to a pointer to a function of a different type
* A pointer to an object to a pointer to an object of a different type
* A pointer to a member to a pointer to a member of a different class or type, if the types of the members are both function types or object types

reinterpret_cast

```
int d  = reinterpret_cast<int>(&entero);
int *dp= reinterpret_cast<int*>(d);
float f= reinterpret_cast<float>(&entero);
short s= reinterpret_cast<short>(&entero);
```

```
invalid cast from type 'const int*' to type 'float'
cast from 'const int*' to 'short int' loses precision
```

Outline

El camino hacia la POO : estructuras en C

Las estructuras permiten dividir el código en partes relacionadas a esta estructura : por ejemplo, el código relacionado a una estructura esta agrupado en un mismo archivo. Noción de interfaz : los archivos *headers* dan los prototipos y las estructuras.

El camino hacia la POO : estructuras en C

En image.h

```
typedef struct ImageStruct {
   int width;
   int height;
   int nChannels;
   int widthBytes;
   depthEnum depth;
   char *data;
} Image;
```

El camino hacia la POO : estructuras en C

En image.h

```
int initializeImage(Image *img,int w,int h,
                    int channels,int depth);
int freeImage(Image **img);
int copyImage(const Image *imgsrc,Image *imgdst);
Image* cloneImage(const Image *img);
Image *colorToGrey(const Image *img);
Image *loadPPMImage(const char *fileName);
Image *loadPGMImage(const char *fileName);
int savePPMImage(const Image *src,
                 const char *fileName);
int savePGMImage(const Image *src,
                 const char *fileName);
```

El camino hacia la POO : estructuras en C

Algo que podría ser mejorado : siempre usamos funciones que, para modificar una estructura imagen, requiere un apuntador hacia la estructura. Dicho de otra manera, el código puede parecer como inútilmente complicado, ya que sabemos que esas funciones son "exclusivas" para imágenes.

Ademas, para evitar conflictos de nombres, se puede tener que prefijar el nombre de las funciones (problema de visibilidad).

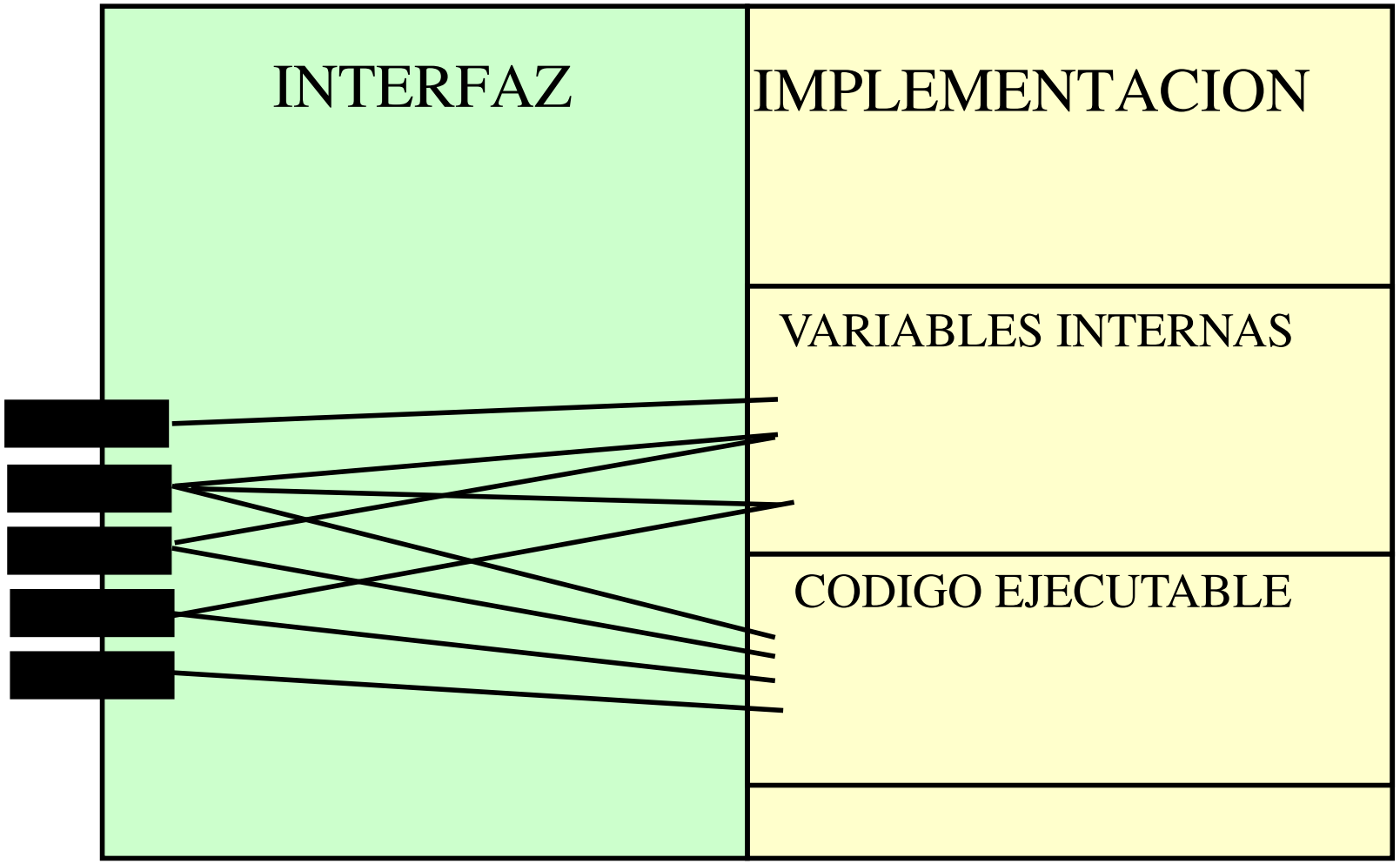
El camino hacia la POO : struct en C++

- La primera etapa natural para ir mas adelante en la POO es la "inclusión" de las funciones dentro de la estructura : la función llamada ve entonces todos los elementos de la estructura que la está llamando.
- Esas funciones son características de la estructura, son como la manera de enviarles mensajes para su modificación...

El camino hacia la POO : struct en C++

```
struct Image {
  int width;
  int height;
  ...
  int initialize(int w,int h,
                 int channels,int depth);
  int freeImage();
  int copyFrom(const Image *imgsrc);
  Image* clone();
  Image *colorToGrey();
  bool loadFromPPM(const char *fileName);
  bool loadFromPGM(const char *fileName);
  ...
};
```

El camino hacia la POO : struct en C++



El camino hacia la POO : struct en C++

Notar que el typedef puede ser omitido, y que aun así podemos crear estructuras :

```
Image img;
```

y llamar sus funciones, o métodos :

```
img.initialize(100,100,3,1);
```

o con estructuras dinámicas :

```
Image *img = new Image;
img->initialize(100,100,3,1);
delete img;
```

El camino hacia la POO : struct en C++

Pero en el archivo .cpp, como hacer manifestar que las definiciones que se escribe son propias a la estructura imagen ? Si mantengo mi código C voy a tener problemas de ambigüedades, ya que varias definiciones iguales pueden ocurrir...

Noción de scope, o visibilidad : similarmente a los $\mathrm{namespace}$, las estructuras tienen un "espacio de nombre" que se accede con $\mathrm{struct}::$, donde struct es el nombre de la estructura.

El camino hacia la POO : struct en C++

```
int Image::savePGMImage(const char *fileName) {
  if (nChannels!=1 || depth!=UCHAR)
    return -1;
  ofstream out(fileName);
 if (out.good()) {
  out << "P2\n #Test \n "
      << width << " "
      << height << "\n255\n";
  unsigned char *udata =
      static_cast<unsigned char *>(data);
  for (int i=0;i<width*height;i++,udata++)
    out << *(udata) << " ";
  }
  out.close();
  return 0;
};
```

struct en C vs. struct en C++

- Declaraciones de métodos son obligatorias ! Fuerza coherencia.
- Aceso a elementos transformados de $\mathrm{img}- >\mathrm{width}$ a width
- Se puede leer mejor !

El camino hacia la POO : struct en C++

Y si, dentro de un método, como Image::savePGMImage, quiero saber qué estructura exactamente me esta llamando, entre todos los objetos instanciando esta estructura, ¿ puedo recuperar la dirección de esta estructura ?

La respuesta es sí : hay una manera de acceder a la dirección de la estructura a través de una palabra llave especial, llamada this.
Entonces, se puede usar :

```
this->width = w;
```

que es equivalente a

```
width = w;
```

struct como objeto

Aquí encontramos una herramienta informática útil para modelar lo que hemos llamado objetos : a la vez una entidad que es una variable, con una dirección en memoria, y también, intrínsecamente asociada con ella, funciones que permiten comunicar con este objeto.

struct como objeto

El tamaño de este struct es el mismo que lo que sería en C (porque el código para los métodos esta almacenado a parte, y común a todas esas struct), excepto...

```
struct s1 {
  char ele1;
  short ele2;
  char ele3;
  void metodo();
};

struct s2 {
  void metodo();
};
```

Tamaño por sizeof : 6 y 1 respectivamente. ¿Por qué ?

struct dentro de struct

Se puede definir una estructura especial que solo tiene sentido para una estructura dada : entonces es lógico encapsular esta definición dentro de la definición de la estructura principal.

struct dentro de struct

Por ejemplo :

```
struct Image {
   int width;
   int height;
   struct rgbPixel {
          unsigned char r,g,b;
          rgbPixel invert();
   }
   struct lookUpTable {
          int size;
          rgbPixel *table;
          initialize(int sz);
   }
   ...
};
```

struct dentro de struct

Definiciones asociadas :

```
Image::rgbPixel Image::rgbPixel::invert() {
        rgbPixel newcolor;
        newcolor.r = 255-r;
        newcolor.g = 255-g;
        newcolor.b = 255-b;
        return newcolor;
};
```

struct dentro de struct

Definiciones asociadas :

```
int Image::lookUpTable::initialize(int sz) {
        ...
        table = new rgbPixel[sz];
        ...
};
```

Uso adecuado de los niveles de resolución de nombres.

struct como objeto

La cosa importante es que finalmente juntamos todo lo relacionado a una estructura dentro de esta estructura (noción de tipo de datos abstracto), y que sus instancias pueden ser vistos como objetos a quienes enviamos mensajes a través de las funciones, o métodos

Problema : nos falta todavía poder forzar nuestros "clientes" a comunicar con los objetos solo a través de los mensajes, sin manipular los elementos de la estructura (data, width...).

Paso siguiente : las clases !