Programación en C++ (5) : control de nombres, referencias, constructor por copia

Dr. J.B. Hayet

CENTRO DE INVESTIGACIÓN EN MATEMÁTICAS

Septiembre 2009

Outline

Outline

La palabra llave static

Dos sentidos coexisten para esta palabra cuando es aplicado a un objeto:

- un sentido relativo a la visibilidad de un objeto, que solo se verá el código objeto de este objeto en el .cpp en que esta definido,
- un sentido relativo al tipo de almacenamiento de una variable : definido dentro de una función o de una clase, no se almacena de manera efímera en el tiempo de existencia de la función/objeto sino permanentemente.

La palabra llave static : funciones

Como ya vimos en C, la primera manera de usar static es dentro de una función. En este caso, la variable modificada ya no esta alocada en la pila de memoria, sino en un espacio estático.

La linea de inicialización esta hecha una vez (a la primera llamada), y luego se puede modificar esta variable (mientras la palabra llave const no está usada).

La palabra llave static : funciones

```
int fonc() {
  static int i= 0;
  cout << i++ << endl;
  return 1;
};
int main() {
  fonc();
  fonc();
};
```

Valores imprimidas ? Notar que si uno no inicializa el valor de las variables estáticas entonces el compilador hace la inicialización a 0 (pero solo por objetos simples).

La palabra llave static : funciones

Para objetos (instancias de clases), no se hace la inicialización a 0 !
Se usan los constructores :

```
int fonc() {
  static Image ig1;
  static Image ig2(100,100);
  return 1;
};
```

Esas dos lineas suponen : un constructor por default (primer caso) y un constructor con dos enteros para el segundo.

Objetos static : constructores

Los objetos están creados a través del constructor, al momento de llamar la función (por primera vez) o justo antes del main() si es un objeto estático global :

```
class Image {
  char c; // Identifier
public:
  Image(char cc) : c(cc) {
    cout << "Image::Image() for " << c << endl;
  }
  ~Image() {
    cout << "Image::~Image() for " << c << endl;
  }
```

Objetos static : constructores

```
Image a('a');
void fonc1() {
  static Image b('b');
}
void fonc2() {
  static Image c('c');
}

int main() {
  cout << "inside main()" << endl;
  fonc1(); // Calls static constructor for b
  cout << "leaving main()" << endl;
}
```

Objetos static : constructores

```
Image::Image() for a
inside main()
Image::Image() for b
leaving main()
Image::~Image() for b
Image::~Image() for a
```

Objetos static : destructores

Los destructores están llamados al salir del main() o usando la función exit() : entonces cuidado al no usar exit() dentro de un destructor (riesgo de recursión infinita)

```
class Image {
  ~Image() {
    cout << "Image::~Image() for " << c << endl;
    exit();      // Problema en perspectiva...
  }
}
```

Visibilidad : external linkage

Por default, si escribo mi archivo File1.cpp:

```
#include "Image.h"
Image img;
...
int main() {
    ...
    return 0;
}
```

El símbolo correspondiendo al objeto img es visible en todo mi código :

```
0000020c S _img
```

La palabra llave extern es implícita...

Visibilidad : external linkage

Y desde otro archivo, como File2.cpp, puedo acceder a este objeto :

```
#include "Image.h"
extern Image img;
...
```

Esta propriedad es la de *external linkage* : las variables globales, por default, son extern.

Visibilidad : external linkage

Excepción: en C++ las variables const globales tienen internal linkage por default.

```
#include "Image.h"
const Image img;
...
int main() {
    ...
    return 0;
}
```

Visibilidad : internal linkage

El comportamiento reverso (visibilidad del código sólo para el archivo que va compilándose) es el de static

```
#include "Image.h"
static Image img;
...
int main() {
    ...
    return 0;
}
```

El símbolo img ya no aparece en la tabla de símbolos de File1.o !

Las dos staticness

```
extern Image img;
```

Implica la no-staticness de img en términos de visibilidad, pero, como para todas variables globales, este objeto tiene staticness en términos de representación en memoria !

Ahora cuando se va de un nivel de visibilidad "global" hasta niveles anidados (funciones, clases...), la palabra static se refiere al tipo de almacenamiento.

Datos miembros static

En el .h :

```
class Image {
        static int variableUtil;
public:
        ...
};
```

En el .cpp :

```
int Image::variableUtil = 100;
```

Tienen que ser declarados como static en la definición de la clase y inicializados a parte en el archivo de implementación. Utiles para por ejemplo sistemas de contadores...

Datos miembros static : inicialización

En el caso de que son const, pueden estar inicializados en la definición de la clase misma (pero también pueden quedar separadas declaración y definición):

```
class Image {
        static const int constanteUtil = 100;
public:
        ...
};
```

Datos miembros static : arreglos

Arreglos :

```
class Image {
    static const int constanteUtiles[];
    static int arregloUtil[];
public:
        ...
};

int Image::arregloUtil[]={2,-1,0,0};
const int Image::constanteUtiles[]={1,3,2,-1};
```

Esos no se pueden inicializar en la definición de la clase.

Datos miembros static : objetos

```
class Procesador {
  static Image buffer[];
};

Image Procesador::buffer[] = {Image(100,100),
                              Image(200,200)};
```

Inicializados igualmente a arreglos de objetos...

Datos miembros static : métodos

Se puede por fin definir métodos estáticos, y sólo pueden acceder a variables estáticas de la clase : los métodos static no reciben el apuntador this; por consecuencia, se pueden usar (como variables static) sin referencia a una instancia particular de esa clase :

```
class Image {
 public:
   static int funcStatic();
};
...
int a = Image::funcStatic();
```

Espacios de nombres

Ya vimos que existe la posibilidad de crear unidades léxicas particulares, los espacios de nombres, que permiten evitar conflictos al uso de nombres.

¿Cual es la otra aportación de C++ que permite reducir también la probabilidad de conflictos ?

Espacios de nombres : creación

Se crean simplemente así :

```
namespace PanchoLib {
   ... // Codigo : clases, variables globales,...
}
```

Bastante similar a clases, por ejemplo, ¿ no ?

Espacios de nombres : creación

Se puede parecer a la de clases pero no lo es :

- No se necesita punto coma después de la llave que cierra el espacio de nombres,
- Solo se puede crear espacios de nombres al nivel global, no se puede definir uno dentro de una clase por ejemplo.
- Se pueden anidar espacios de nombres dentro de otro :

```
namespace MisBonitosTrabajos {
namespace PanchoLib {
        class Pancho {
        };
}
}
MisBonitosTrabajos::PanchoLib::Pancho obj1;
```

Espacios de nombres : creación

Se puede parecer a la de clases pero no lo es :

- Se puede hacer aliases de espacios de nombres

```
namespace MisBonitosTrabajos MisBos;
namespace MisBonitosTrabajos::PanchoLib Panch;
```

- Se puede "rellenar" el espacio de nombre por pedazos, en un mismo archivo o a través de varios archivos :

```
namespace MisBonitosTrabajos {
...
}
...
namespace MisBonitosTrabajos {
...
}
```

Espacios de nombres sin nombres

Se presenta el espacio de nombre sin nombre :

```
namespace {
  class Image {
    ...
  };
}
```

¿ Para qué ?

Espacios de nombres sin nombres

Y es equivalente a :

```
namespace __NombreEscondido__ {
    class Image {
        ...
    };
}
using namespace __NombreEscondido__;
```

Cual es la ventaja de usar una construcción así ?

Espacios de nombres sin nombres

Actúa como otra manera de hacer variables estáticas (al sentido de *internal linkage*) :
En File1.cpp :

```
namespace {
int i,j;
}
```

En File2.cpp :

```
extern int i;  // El linker no lo encontrará !
```

Espacios de nombres y friends

Una declaración de *friendship* dentro de una clase hace que la función declarada friend esta dentro del namespace que incluye esta clase (el nivel mas bajo que incluya la clase) :

```
namespace A {
namespace B {
   class X {
      friend void f(X);  // A::B::f es friend
      };
   void f(X) { /* definition */}
}
}
A::B::X a;
A::B::f(a);
```

Espacios de nombres : combinar los scopes

Se combinan los scopes de clases y de espacios de nombres :

```
namespace A {
  class B {
    static int constante;
  };
  void func();
B inst;
}
int A::B::constante = 9;

class A::C {
  int u, v, w;
public:
  C(int i);
};
```

Espacios de nombres : directiva using

Para usar namespace sin tener que hacer explícitos todos los niveles, usar la directiva using namespace :

```
using namespace A;
B x;
func();
```

Notar que se puede usar esta directiva :

- Dentro de otros namespace.
- Dentro de funciones.

Espacios de nombres : directiva using

Notar también que aun con el uso de using se puede reescribir "sobre" elementos de un namespace :

```
using namespace A;
B inst; // No es el de A !
B &c = A::inst; // Eso si es
```

Espacios de nombres : directiva using

Puede haber casos de colisión, pero solo se presentarán si efectivamente se usa nombres ambiguos de funciones/variables definidos en los dos espacios de nombre :

```
namespace AA {
    void func();
}
using namespace A;
using namespace AA;
... // Hasta ahora nada malo, aunque
... // los espacios de nombres contienen
... // nombres conflictuosos
func(); // Ahora sí problema !
```

Espacios de nombres : declaración using

Si se necesita unas cositas dentro de un espacio de nombres, pues se puede contentar con el uso de una declaración del uso de estas cositas por una declaración using:

```
using namespace A;
using AA::func();
func(); // Llama a AA::func()
A::func();
```

Inclusión dentro de un namespace :

```
namespace A {
        using D::f1();
}
int g() {
        using namespace A;
        func();
}
```

Espacios de nombres : uso practico

Reservar el uso de las directivas `using` a los archivos `.cpp`, y a ellos únicamente porque si se los incluye en los `.h`, puede haber problemas (los espacios de nombres estarán vistos en todos los archivos incluyendo este *header*).

Outline

Apuntadores...

Apuntador = Dirección memoria + Tipo de objeto referido

```
int a = 1;
int *a_ptr = &a;
int b = *a_ptr; // Dereferenciar
*a_ptr= 2;      // Cambio al apuntado
a_ptr = &b; // Cambio al apuntador
```

El apuntador es un tipo como otro con una representación memoria : puedo cambiar su valor y cambiar el valor apuntado.

Apuntadores : del C al C++

La única grande diferencia al nivel de los apuntadores es al nivel de las conversiones. Ya no se permite :

```
int main() {
  int arr[]={1,2,1,1};
  int *a    = &arr[0];
  void *v   = a; // Apuntador genérico
  double *d= v;
}
```

Apuntadores : del C al C++

Apuntadores hacia void : son apuntadores "genéricos" que sirven (sobre todo) para poder escribir funciones genéricas de manipulación de la memoria (como malloc, memset...). Por definición, no se pueden dereferenciar.
Hacen "perder" la noción del tipo (y eso puede resultar peligroso): a la asignación, no se sabe bien que tipo de cast esta ocurriendo.

```
void *memset(void *b, int c, size_t len);
```

Apuntadores : C++

El C++ deja posible usar los void * pero prohibe asignación o cast implícito de ellos hacia otros tipos de apuntadores : se tiene que usar cast explícito

```
int main() {
  int arr[]={1,2,1,1};
  int *a    = &arr[0];
  void *v   = a; // Apuntador genérico
  double *d= static_cast<double *>(v);
}
```

Referencias

Una referencia es algo muy similar a un apuntador constante (de hecho la implementación de referencias esta basada en apuntadores), y que no se tiene que dereferenciar (se hace eso automáticamente); es una capa de C++ destinada a ayudar el desarrollador harto de los apuntadores.

```
Image img;
Image &imgRef= img;
```

Sirve de "alias" al almacenamiento en memoria para el objeto img

Referencias

Una referencia tiene que estar inicializada para estar válida, y la referencia así creada siempre se referirá al mismo objeto !

```
Image &imgRef; // No compila
```

```
error: 'imgRef' declared as reference but not initialized
```

```
Image img;
Image &imgRef= img;
```

imgRef se quedará "ligado" al objeto img en todo el campo de visibilidad, y no hay manera de hacerle referenciar a otra cosa.

Referencias

Cosas que se pueden hacer con apuntadores pero que no se pueden hacer con referencias :

- Manipular directamente la referencia una vez que esta inicializada : cada mención de esta referencia después se refiere al objeto referenciado, eso vale para afectaciones.
- En particular, cambiar el valor de la referencia no es posible.
- Hacer operaciones aritméticas.
- Hacer referencias a NULL.
- Existir no-inicializadas.

Referencias hacia valores constantes

Un caso particular :

```
const double &val= 1.0;
```

El compilador crea un espacio memoria para el doble, que inicializa con el valor pasado, y una referencia a este espacio. La palabra llave const es necesaria !

Referencias y funciones

Se parecen el uso de apuntadores y el uso de referencias, para evitar pasar valores y poder modificar las valores dentro de las funciones :

```
Image *doSomething(Image *img) {
    img->doTruc();
    return img;
}
```

Se puede reescribir con las mismas propiedades:

```
Image &doSomething(Image &img) {
    img.doTruc();
    return img;
}
```

Referencias y funciones

¿ Qué puedo regresar como referencia si no tengo argumentos ?

```
int g() {
  double c[100];
  for (int i=0;i<100;i++) c[i] = i;
  return 0;
}
int &fonc() {
  int a = 1; return a;
}
int main() {
  int &b1 = fonc();
  g();
  cout << b1 << endl;
}
```

Compila, pero ¿uh?

Referencias y funciones

Salida:

100

¿Qué habrá pasado?

Referencias y funciones

Las referencias deben de hacerse hacia variables/objetos que van a permanecer !

```
int &fonc() {
   static int a = 1; return a;
}
```

En este caso, la referencia sí tiene sentido !

Referencias y funciones : const

Si pasamos referencias normales hacia funciones, se podrá cambiar el valor de la variable referenciada (en el caso de tipos básicos) o se podrá aplicar todo tipo de método sobre el objeto

Si pasamos ahora referencias const : no se podrá cambiar el valor (int, double...) o sólo se podrá usar métodos const (caso de objetos)

```
int doSomething(const Image &img) {
    img.doTruc();
    return 0;
}
```

Sólo funciona si doTruc es const (y public, claro).

Referencias y funciones : const

Para el caso en que el compilador tiene que crear por sí mismo las referencias, el const es absolutamente necesario :

```
int doSomething(int &i) {
    i+=10;
    return 0;
}
doSomething(1); // NO SE PUEDE, POR QUÉ ?
```

Referencias a apuntadores

Un apuntador siendo un tipo como todos los otros, no hay ningún problema en crear referencias hacia apuntadores :

```
bool allocateSomething(int **array) {
    *array = malloc(size*sizeof(int));
    ...
}
int *truc;
allocateSomething(&truc);
```

Se puede reescribir en :

```
bool allocateSomething(int *&array) {
    array = new int[size*sizeof(int)];
  ...
int *truc;
allocateSomething(truc);
```

Referencias y funciones

El caso por default en el diseño de función debe de ser de pasar sistemáticamente (al menos para grandes tipos, estructuras y objetos) los parámetros como referencias const !

Pasar por valores puede llevar a problemas...

Funciones : usar valores

Si defino una funcion de prototipo :

```
int fonc(double a,double b);
```

y que la uso después

```
int val = fonc(x,y);
```

¿Qué pasa exactamente atrás ?

Funciones : usar valores

Lo que hace el compilador :

- para llamar la función, copia a los argumentos en el espacio de memoria sobre la pila (como clone),
- al terminar la función, copia otra vez el valor calculado de regreso en un registro.

En todos casos necesita copiar datos. Para los tipos básicos, no hay problema : para copiar un doble, hago una copia conforme de los 4 octetos que componen el doble, y caben en un registro.

Funciones : usar valores

En la pila después de la llamada (mientras se ejecuta la función) :

- Argumentos de la función.
- Dirección de regreso (en el programa, para seguir la ejecución al salir de la función).
- Variables locales.

Funciones : usar valores

Qué pasa ahora cuando consideramos tipos muy diferentes como estructuras o clases ? Sus valores no pueden estar almacenados en registros, si son demasiado gordos.

- Escribiendo el valor de regreso abajo en la pila ? No se puede porque puede occurir interrupción al momento de return y la función manejando la interrupción escribirá sobre el valor de regreso.
- Escribiendo en memoria estática ? Tampoco es posible. . . ¿ Por qué ?
- La solución es por registros si se puede O pasar la dirección de regreso entre los argumentos y copiar el valor de regreso a esta dirección.

Todo llega a un problema de copia de los objetos.

Funciones : usar valores

La copia mas evidente sería hacer una copia conforma (bit a bit) de un objeto ! A ver lo que pasa con esa clase :

```
class Image {
  int x;
  int y;
  static int count;
 public:
  Image() {
    count++;
  }
  ~Image() {
    count--;
  }
  static void printCount() {
    cout << "Counter : " << count << endl;
  }
```

Funciones : usar valores

Defino también una función fonc que manipula objetos de la clase Image en valor de regreso Y en argumento :

```
Image fonc(Image a) {
  return a;
}
```

Según lo visto antes, una llamada a esta función necesitará crear dos objetos temporales de tipo Image : uno para el argumento (sobre la pila), uno para el valor de regreso (al nivel arriba). Esos objetos serán destruidos.

Funciones : usar valores

Ahora :

```
int main() {
  Image::printCount();
  Image a;
  Image::printCount();
  fonc(a);
  Image::printCount();
```

Me imprime :

```
Counter : 0
Counter : 1
Counter : -1
```

Uh ?

Funciones : usar valores

Lo que pasó es que, dos veces, a la llamada de la función fonc, el programa pasó por un constructor que no es el mío ! Y eso es un problema en sí mismo porque, en mi caso, quiero controlar el número de instancias de mi clase.

Funciones : usar valores

Remarca :

```
int main() {
  Image::printCount();
  Image a;
  Image::printCount();
  Image b = fonc(a);
  Image::printCount();
```

Me imprime :

```
Counter : 0
Counter : 1
Counter : 0
```

¿Uh ?

Constructores por copia

El compilador usa, para el caso de construcción de un objeto a partir de otro objeto, lo que se llama un constructor por copia. Eso ocurre por ejemplo :

- a la llamadas de funciones pasando valores,
- a asignaciones en definiciones de tipo :

```
Image a;
Image b=a;
Image c(a);
```

Por default, el compilador hace una copia bit a bit de los objetos.

Constructores por copia

Este constructor por copia tiene un prototipo bien definido :

```
Image::Image(const Image &img);
```

y, por default lo define el compilador. Eso explica que escribir :

```
Image a;
Image c(a);
```

tenga sentido.

Constructores por copia

Ahora, es aconsejado definir su propio constructor por copia para evitar los problemas encontrados mas arriba :

```
class Image {
  int x;
  int y;
  static int count;
 public:
  Image() {
    count++;
  }
 Image(const Image &otra) {
    x = otra.x;
    y = otra.y;
    count++;
  }
```

Constructores por copia

Así llegamos al comportamiento deseado :

```
Counter : 0
Counter : 1
Counter : 2
```

Sí hay bien dos objetos instanciando la clase Image que existen a este momento.

Constructores por copia

Notar que, como para un constructor normal, está bien poder inicializar un objeto como nosotros lo queremos hacer ! Y hay casos en que una copia bit por bit lleva aún a resultados catastroficos !

```
class Image {
  int x;
  int y;
  char *data;
  }
```

Copiaría la dirección de los datos en el objeto copiado, pero su destructor liberaría la memoria, incluso para el objeto original ! En este caso, tendríamos que alocar un nuevo espacio memoria para la imagen copiada.

Constructores por copia : composición

Si defino :

```
class OtraCosa {
 public:
  OtraCosa() {};
  OtraCosa(const OtraCosa &otra) {cout<<"CC OtraCosa"<
};
class Image {
  int x;
  int y;
 public:
  Image() {};
  Image(const Image &otra) {cout<<"CC Image"<<endl;};
};
```

Constructores por copia : composición

Y luego :

```
class Gorda {
  Image i;
  OtraCosa o;
};

int main() {
  Gorda g1;
  Gorda g2 = g1; // LLama al CC de Gorda
}
```

Me imprime los dos mensajes : el constructor por copia de un objeto compuesto de otros objetos (no básicos) llama a sus constructores por copia respectivos ! Cuidado que si defines tu mismo el constructor por copia, esta tarea (copiar los miembros) será tuya !

Constructores por copia : aplicación

Una aplicación interesante: poner el constructor por copia private. En este caso, estamos seguro que todo intento de pasar un objeto de este tipo por valor (o de regreso) será siempre rechazado por el compilador.

```
class Image {
  int x;
  int y;
  Image(const Image &otra) {};
 public:
  Image() {};
};
void f(Image img);
Image g();
```

```
error: 'Image::Image(const Image&)' is private
```

Apuntadores, otra vez

Los apuntadores permiten representar direcciones absoluta en la memoria, hacia objetos o funciones,

```
Image img1;
Image *img_ptr = &img1
```

No habría manera de hacer algo similar dentro de un objeto dado (o sea manipulando esta vez *offsets*) ?

Apuntadores a miembros

Son como apuntadores, pero actuan relativamente, dentro de un objeto dado :

```
int Image::*ptrToMbr = &Image::x;
Image a;
cout << a.*ptrToMbr << endl;
ptrToMbr =  &Image::y;
```

Uso muy limitado (no operaciones aritméticas, comparaciones).

Apuntadores a miembros

- No corresponden a una dirección, sino a un *offset*.
- Similarmente a los apuntadores, también se puede aplicar a apuntadores de objetos,

```
Image *b;
int Image::*ptrToMbr = &Image::x;
cout << b->*ptrToMbr << endl;
```

Apuntadores a métodos

Con el mismo principio a apuntadores hacia funciones :

```
int (*foncPtr)(const Image &) = &fonc;
```

se define apuntadores hacia métodos, con las mismas ventajas (por ejemplo, procesar de la misma manera a datos, pero con algoritmos diferentes) :

```
int main() {
  Image a,b;
  int (Image::*foncPtr)(const Image &)
        = &Image::method1;
  (a.*foncPtr)(b);
};
```

Notar que la definición no hace referencia a una instancia particular de la clase; sin embargo, la llamada a este método tiene que hacerse relativamente a un objeto (apuntador this).

Apuntadores a métodos

Dentro de una clase :

```
class Image {
  int method1(int) {cout << "1" << endl;};
  int method2(int) {cout << "2" << endl;};
  int method3(int) {cout << "3" << endl;};
  int (Image::*foncPtr[3])(int);
public:
  Image() {
    foncPtr[0] = &Image::method1;
    foncPtr[1] = &Image::method2;
    foncPtr[2] = &Image::method3;
  };
  int doStuff(int m,int par) {
    if (m>=0 &&m <3) (this->*foncPtr[m])(par);
  };
};
```

Apuntadores a métodos

Notar que aunque estamos dentro de la clase, el uso del nombre completo de los métodos y el uso del apuntador this son obligatorios.