Programación en C++ (4) : nuevas características de las funciones, constness

Dr. J.B. Hayet

CENTRO DE INVESTIGACIÓN EN MATEMÁTICAS

Septiembre 2009

Outline

Outline

Argumentos por default

A veces puede ser que la función que defines tiene un parámetro que casi siempre usas con un valor dado, y, excepcionalmente, con otro valor...

Argumentos por default

```
class Image {
        ...
 public:
        Image(int w,int h,int ch, int d);
        ...
}
```

En general, tendremos $d = 1$, entonces no podríamos evitar al cliente pasar siempre este argumento ? Una opción sería sobrecargar el constructor con un argumento menor pero es re-escribir código casi igual...

Argumentos por default

En C++, podemos hacer lo siguiente :

```
class Image {
        ...
 public:
        Image(int w,int h,int ch, int d=1);
        ...
}
```

y

```
Image::Image(int w,int h,int ch, int d) {
        ...
}
```

La definición no cambia, pero la declaración tiene los valores *default* de los argumentos.

Argumentos por default

Eso nos permite llamar a la función sin pasar explicitamente el argumento por default :

```
Image *img = new Image(w,h,3);
```

Pero cual es el problema si quiero, por ejemplo, quiero un *h* por default ?

Argumentos por default

Imaginemos que :

```
class Image {
        ...
 public:
        Image(int w,int h=300,int ch, int d=1);
        ...
}

Image *img = new Image(w,h,3);
```

¿Cómo el compilador podría interpretarlo ?

Argumentos por default

Para prevenir este tipo de problemas y levantar ambigüedades:

- sólo los argumentos que vienen en cola de la lista de argumentos pueden tomar valores por *default*;
- si usas un argumento default en una llamada, pues todos los que siguen deben de ser default.

Por ejemplo :

```
Image(int w,int h,int ch=3, int d=1);
...
Image *img = new Image(w,h,4);
```

implica que $ch = 4$ y $d = 1$ (y no $d = 4$)

Valores por default o sobrecargo ?

Los dos pueden parecer similares (permiten llamar a funciones de diferentes maneras); sin embargo, la filosofía es muy diferente !

1. Usar los valores por default cuando se trata de un parámetro de un algoritmo o de inicialización "inocente" y que, estadísticamente, tiene un valor mas frecuente

```
void cvCalcMotionGradient( const CvArr* mhi,
                           CvArr* mask,
                           CvArr* orientation,
                           double delta1,
                           double delta2,
                           int aperture_size=3 );
```

Valores por default o sobrecargo ?

2. Usar sobrecargos de funciones cuando se trata de un código, de un comportamiento muy diferente.
3. Evitar usar el valor por default como simple flag (no es el propósito) :

```
class Image {
  void doThat(int par=1);
};

void Image::doThat(int par) {
 if (par==1) {
        doThis();
        return;
 }
...
```

Valores por default o sobrecargo ?

```
MyString::MyString() {  buf = 0; }
MyString::MyString(char* str) {
  buf = new Mem(strlen(str) + 1);
  strcpy((char*)buf->pointer(), str);
}
```

vs.

```
MyString::MyString(char* str) {
  if(!*str) { // Pointing at an empty string
    buf = 0;
    return;
  }
  buf = new Mem(strlen(str) + 1);
  strcpy((char*)buf->pointer(), str);
} }
```

Outline

const para remplazar valores constantes

En C, la manera más clasica de “almacenar” valores constantes es el uso de comandos del preprocesador :

```
#define MYLIB_SIZEBUF 1024

char buf[MYLIB_SIZEBUF];
```

Solo operaciones textuales (remplazar cadenas de caracteres por otras) hechas por el preprocesador. Habíamos visto que ese sistema puede llevar a problemas.

const para remplazar valores constantes

En C++, se preferirá usar :

```
const int bufSize = 1024;

char buf[bufSize];
```

En este caso, el proceso de remplazar lo lleva a cabo el compilador mismo y no el preprocesador. Se puede usar con todos los tipos básicos. El proceso va un poco mas adelante que simplemente declarar que esa variable no se puede cambiar !

const para remplazar valores constantes

Igualmente que para las directivas al preprocesador, poner esas declaraciones/inicializaciones en un archivo *header* .h : a priori, visibilidad solo para el archivo en que se esta definiendo (internal linkage, sino usar extern).

Dos tipos de uso de const en C++

- Remplazar mas eficientemente los define, que por construcción pueden tener efectos de bordo.
- Uso mas tradicional de "protección" de una variable.

Constantes almacenadas o no

Consideremos el código siguiente :

```
const int a=1;

int main() {
  int b=a;
};
```

Se lo puede compilar en C o en C++, pero (1) en C es sistemáticamente extern (mientras en C++ no) y (2) ...

Constantes almacenadas o no

En C, resultado de nm:

```
0000001c S ___i686.get_pc_thunk.cx
00000018 S _a
00000000 T _main
```

En C++, resultado de nm

```
00000000 T _main
00000000 A _main.eh
```

En C++, y en este caso (variable global), no guarda el compilador un espacio de almacén para *a*, usa un proceso de preparación del código objeto aun en el tiempo de la compilación.

Constantes almacenadas o no

Pero puede ser que no sea así siempre :

```
extern const int a=1;

int main() {
  int b=a;
};
```

Recompilado, nos da esta vez :

```
00000014 S _a
00000000 T _main
00000000 A _main.eh
```

El compilador creó espacio memoria para *a* para poder cambiarle entre los diferentes archivos objeto (porque hemos puesto la palabra llave extern).

Constantes almacenadas o no

- Cuando el compilador lo puede, no almacena constantes globales definidas por const : comportamiento eficiente.
- Como variables globales deben de estar inicializadas, si no :

```
testobis.c:1: error: uninitialized const 'a'
```

Constantes almacenadas o no

```
#include <iostream>

const int a = 1;
const int b = 10+a;
int main() {
   int x = a;
   int y = b;
   const char c = std::cin.get();
};
```

El *c* tiene que tener un lugar de almacenamiento en memoria, ¿por qué ?

const de protección

- Usar const cada vez que piensan que una variable no tendría que estar modificada !
- Una vez la inicialización de esta variable hecha, ya no se puede cambiar (a priori).
- En C++, tienen niveles de visibilidad (lo que es mejor que con los define).

Apuntadores a const

```
const int *ptr;
```

Se lee como “apuntador a una variable en memoria de tipo const int”; el apuntador sí mismo no es una constante, entonces se puede no incializarle.
Una remarca : si existen apuntadores hacia una variable const entonces el compilador tiene que alocarle memoria !

```
const int a = 1;
const int b = 10+a;
int main() {
  int y = b;
  const int *ptr;
  ptr = &a;
  return 0;
};
```

Apuntadores const

```
int *const ptr;
```

Se lee como "apuntador de tipo const hacia un int".

Se usa igual, pero en general es muy poco aconsejado de manipular constantes numéricas con ellos :

```
int *const ptr = 0xE100000; // Uuuuuhh
```

El valor apuntado puede cambiar pero no el apuntador :

```
*ptr = 122; // SI
ptr  = &b;  // NO !
```

Apuntadores const a const

```
const int  *const ptr;
int const *const ptr;
```

Ambas formas se leen como “apuntador de tipo const hacia un int de tipo const ”. Ni el apuntador, ni el valor apuntado se podrán cambiar.

```
*ptr = 122; // NO !
ptr  = &b;  // NO !
```

Verificación del tipo por el compilador

El C++ es particularmente estricto en eso, y las afectaciones que no respetan las constness de los diferentes elementos involucrados no son autorizadas (sin pasar por los casts adecuados):

```
int x = 20;
const int y = 10;
const int *ptr1 = &x; // SI
const int *ptr2 = &y; // SI
//int *ptr3 = &y;     // NO !
int *ptr4 = const_cast<int *>(&y); // SI
```

Verificación del tipo por el compilador

Una excepción, en la que el compilador autoriza :

```
char* chaine = "thisisatest";
chaine[3] = 'p';
```

provoca un error ! Usar mejor :

```
char chaine[] = "thisisatest";
chaine[3] = 'p';
```

const como argumentos de funciones

```
void print(const char c) {
        ...
}
```

Es útil para legibilidad y claridad, aunque no sería absolutamente necesario desde el punto del vista del que llama (por qué ?). Se puede clarificar para el que desarolla :

```
void print(char c) {
        const char &cc = c;
        cc = 'a'; // NO !
};
```

const como argumentos de funciones

Apuntadores y referencias : como vimos, es aconsejado pasar a las funciones referencias o apuntadores en lugar de puras valores; el const se aplica igual a ellos :

```
void print(const myStruct &m) {...};
void print2(myStruct &m) {...};
...
myStruct s1;
const myStruct s2 = s1;
print(s1);  // SI
print(s2); // SI
print2(s1); // SI
print2(s2); // NO !
```

Aquí sí es mas critico el papel de la palabra const !

const como valor de regreso

Igual comportamiento, servirá para que el valor de regreso (por ejemplo un objeto de tipo clase o estructura) no sea modificado :

```
myStruct g() {...};
const myStruct h() {...};
g() = a; // OK pero poco util
h() = b; // NO
```

pero es generalmente raro usarlos así ya que copiamos en la mayoría de los casos el objeto regresado en otro objeto del mismo tipo.

const como valor de regreso

Apuntadores y referencias : regresar una referencia hacia const o hacia un apuntador puede ser útil cuando se trata de variables estáticas, por ejemplo, o de elementos de una clase o estructura :

```
const myStruct &g() {
        static myStruct m;
        ...
        return m;
};
 // Recuperamos una referencia a m
const myStruct &s = g(); // SI
s.doThis(); // NO
s.elemento = 1; // NO;
myStruct &r = g(); // NO
```

const dentro de clases

Se puede definir variables constantes dentro de clases pero :

- necesariamente tendrán una representación memoria,
- no se dan un valor de inicialización, las valores se inicializan en el constructor,
- la palabra const significa aquí que esta variable interna de una clase no va a estar modificada durante la vida del objeto.

const dentro de clases

```
class Image {
private:
        const int width;
        const int height;
        char *data;
public:
        Image(int w,int h,int ch,int d);
        ...
};
```

const dentro de clases

Pero :

```
Image::Image(int w,int h,int ch,int d) {
        width = h;
        ...
};
```

No esta aceptado, porque *width* debe de estar inicializado aun antes de haber entrado dentro del cuerpo del constructor (no se puede dejarlo no inicializado !)

```
testoter.c:10: error: uninitialized member 'Image::width'
with 'const' type 'const int'
```

Lista de inicialización del constructor

Para este problema, se inicializa los miembros de la clase de una forma especial :

```
Image::Image(int w,int h,int ch,int d) : width(w),
                                         height(h) {
        ...
};
```

O sea, una lista de constructores inicializando elementos de la clase. Es obligatorio para elementos const pero se puede usar también para elementos no-const.

static const en clases

El const que hemos visto hasta ahora para clases es solo un const "a la C" (impide modificar la variable pero le da un espacio memoria). ¿Cómo tener una variable const "a la C++" ?

La solución : usar static const

```
class Image {
private:
        static const int constanteUtil = 1200;
        const int width;
        const int height;
public:
        Image(int w,int h,int ch,int d);
        ...
};
```

static const en clases

```
Image::Image(int w,int h,int ch,int d) : width(w),
                                         height(h) {
  int a = constanteUtil;
};
int main() {
   int b = Image::constanteUtil;
  return 0;
};
```

Una particularidad : se puede eventualmente hacer referencia a esta constante sin tener una instancia de un objeto.

Objetos const

Un objeto (=instancia de clase) const es un objeto del cual ningún elemento va a cambiar a partir de su creación/inicialización

```
const Image img(100,100,1,1);
```

Pero, qué tal de los métodos de esta clase ? A priori podrían cambiar el contenido del objeto ? Entonces el lenguaje permite especificar explícitamente que un método es "safe" y que no va a cambiar nada en el valor de los elementos.

Métodos const

Los métodos const son métodos certificando que no tocan al contenido del objeto (y eso es verificado por el compilador) :

```
class Image {
private:
        int width;
        int height;
        char *data;
public:
        Image(int w,int h,int ch,int d);
        int copyFrom(const Image &imgsrc);
        Image* clone() const;
        ...
};
```

Métodos const

Ejemplo :

```
Image img(100,100,1,1);
const Image imgCst(200,200,3,1);
imgCst.copyFrom(img); // NO !
Image *other = imgCst.clone(); // OK
```

Métodos const

Cuidado : para que un método pueda ser const, todos los otros métodos que eventualmente usa tienen que estarlo también ! Por ejemplo :

```
class Image {
public:
        Image(int w,int h,int ch,int d);
        int copyFrom(const Image &imgsrc);
        Image* clone() const;
        int getWidth();
        ...
};

Image::Image *Image::clone const {
        int w = getWidth(); // NO !
        ...
}
```

Métodos const

Notar por fin que la palabra llave aparece en la declaración Y en la definición !

Outline

Macros en C

En C, las macros del preprocesador nos permitían acelerar dramáticamente las llamadas a funciones de corto tamaño utilizando el preprocesador para remplazar directamente el código :

```
#define MAX(x,y) ((x>y)?(x):(y))
int c=MAX(a,b);
```

Se parece a una llamada de función pero ¡no lo es !

Macros en C

Limitaciones :
Efecto de bordo

```
#define pion(X) pi/X
double y = 1/pion(4);
```

La solución es poner sistemáticamente paréntesis. . . pero a veces no es obvio ver el *bug*. . .

Macros en C

Limitaciones :
Podemos olvidar que no es una función normal y escribir cosas con efectos de bordo con consecuencias graves :

```
#define SQUARE(X) ((X)*(X))
int sq = SQUARE(i++);
```

Por eso sirven las mayúsculas : para no olvidar que no podemos hacer cosas como funciones.

Macros en C

Limitaciones :

Ejemplo aun peor :

```
#define BAND(x) (((x)>5 && (x)<10) ? (x) : 0)
a = 4;
b = BAND(++a);
a = 15;
b = BAND(++a);
a = 8;
b = BAND(++a);
```

Macros en C

Limitaciones :

La limitación mas importante en C++ es que no podemos usar noción de macro propia a una clase :

```
class Image {
   int width;
   int height;
#define AUGWIDTH(X,W) ((X).width+=W);
};

int main() {
   Image img;
   AUGWIDTH(img,10); // No se puede
}
```

...porque requiere algo más que el preprocesador...

Macros del preprocesador

La solución que propuso C++ (y que tomó C mas tarde) es dejar el trabajo hecho por el preprocesador al compilador si mismo, para poder tener operaciones mas complicadas que simples cambios de texto, y en particular manejar diferentes niveles de visibilidad, "macros" dentro de clases... Más trabajo al compilador pero código al final tan eficiente !

Un poco de la misma manera que con las variables globales const.

Funciones inline

Porque están procesados por el compilador, las funciones inline parecen realmente similares a funciones. La diferencia es que el código objeto correspondiendo a la función esta pegado al nivel de las llamadas : NO creación de espacios memoria en la pila, NO copia de parámetros...

Macros : remplazan texto por "copy/paste".

Funciones inline : remplazan código objeto por "copy/paste".

Funciones inline : cómo se hacen ?

En general van precedidas por la palabra llave inline y su definición tiene que venir al mismo tiempo que su declaración (sin esto el inline no sirve)
En el .h:

```
inline int getImageProperty(const Image &img) {
        int a = img.somePublicMethod();
        return a;
};
```

Entonces vienen principalmente en los archivos *header* .h

Funciones inline : ¿cómo se hacen ?

Notar que todas las especificaciones de funciones se aplican acá, en particular la verificación de los tipos de los argumentos, las constness y del valor de regreso !

```
inline int band(int x) {
        return ((x>5 && x<10) ?x: 0);
};
```

Ya no hay los efectos de bordo raros de las macros !

Funciones inline vs. macros

Cual es lo más rápido ? Seguramente las macros pero el código se hace más limpio con funciones inline : por eso, es sano remplazar sistemáticamente las macros por funciones inline.

- Uso controlado por el compilador.
- Mas fácilmente debuggeado.

Funciones inline y clases

Se puede escribir funciones inline dentro de clases (y eso es la diferencia esencial con las macros !). Por default, toda función definida dentro de la definición de la clase es inline (no se necesita a priori la palabra llave inline):

```
class Image {
  int width;
  int height;
public:
  void augWidth(int w) { //Está inline !
    width+=w;
  };
};
```

Funciones inline y clases

Luego puedo usar esos métodos igualmente a métodos normales (todo esta pasando detrás de la cortina) :

```
Image img;
img.augWidth(100);
```

Funciones inline : cómo se usan ?

Otra manera de crear funciones inline :
En el .h :

```
class Image {
  int width;
  int height;
public:
  // Tambien se puede dejar el inline
  void augWidth(int w);
};
```

En el .cpp :

```
inline void Image::augWidth(int w) {
    width+=w;
};
```

Funciones inline y clases

Por qué no poner todos los métodos como inline ?

Por construcción, hacer una función inline es interesante solo a partir del momento que el código objeto correspondiendo a una función (que es "pegado") es menos largo que el código correspondiendo a una llamada a una función : eso significa que es mas adecuado para funciones pequeñas.

Métodos inline : métodos de acceso

Ya vimos que queremos que nuestros objetos instanciando nuestras clases guarden sus datos private :

- eventualmente, cambiaremos el nombre, el tipo de esos datos; eso no debe de perturbar nuestros "clientes" !
- podemos querer controlar totalmente la manipulación de los datos de esos objetos y espiar cada acceso; eso requiere forzar el cliente a usar unas funciones nuestras.

Problema : esas funciones van a aparecer muchas veces en el código, tienen que ser MUY eficientes.

Métodos inline : métodos de acceso

En consecuencia es aconsejado usar funciones inline sistemáticamente para esas funciones de acceso :

```
class Image {
  int width;
  int height;
public:
  int getWidth() const {return width;};
  bool setWidth(const int &w) {
        // Eventualmente poner tests
        // sobre w
        width = w;
   };
  ...
};
```

Métodos inline : métodos de acceso

Uno puede usar otra estrategia de acceso, usando los mismos nombres para poner valores y recuperar valores :

```
class Image {
  int width;
  int height;
public:
  int width() const {return width;};
  bool width(const int &w) { width = w;};
  ...
```

Métodos inline : lo que hace el compilador

Como para funciones normales, el compilador crea entradas en la tabla de símbolos para las funciones inline, con su prototipo y el cuerpo de la función. Luego, al identificar llamadas a esa función

- verifica lo correcto de la llamada (argumentos, regreso),
- al lugar de crear código para llamadas, pega el código correspondiendo al cuerpo de la función.

Métodos inline : no se puede siempre...

El problema es que este proceso no es siempre posible : el proceso es costoso para el compilador, y veces demasiado complicado :

- cuando hay estructuras de control, como ciclos,
- cuando hay recursión.

Depende del compilador, pero en general, el proceso de hacer una función inline no se puede a partir de cierto grado de complejidad de la función.

Métodos inline : no se puede siempre...

Otro caso importante es cuando se necesita la dirección memoria de esa función, típicamente para manipulación de apuntadores de funciones (en este caso estará puesto una versión normal de la función y su dirección en memoria).

En todos casos, el hecho de poner la palabra inline no hace la función automáticamente así : depende del compilador, de si lo puede hacer o no.

Métodos inline en clases: forward references

No hay problemas en :

```
class Image {
public:
  inline void haceCosa() {
    augWidth(1);
  };
  inline void augWidth(int w) {
    width+=w;
  };
private:
  int width;
  int height;
};
```

mientras todo se queda dentro de la definición de la clase.

Métodos inline en clases: forward references

```
no inline functions in a class shall be evaluated
until the closing brace of the class declaration.
```

Meetodos inline : cons/des tructores

En este caso particular, cuidado a que un constructor que puede parecer trivial no sea tan trivial por las construcciones automáticas que realiza :

```
class Procesamiento {
public:
    Procesamiento() : counter(0) {};
private:
   int counter;
   Image tmpImg1;
   Image tmpImg2;
};
```

Funciones inline : en resumen

Usar inline cuando :

- Una función representa un *bottleneck* y su optimización es necesaria.
- Esta función es corta.
- Esta función es llamada seguido.

Pero...

Funciones inline : en resumen

. . . no hay que tirar todas las directivas del preprocesador a la basura en todos casos !

```
#define DEBUG(x) cout << #x " = " << x << endl
```

casos de DEBUG

Funciones inline : en resumen

Operaciones textuales (token pasting, ejemplo de OpenCV)

```
#define CV_SEQUENCE_FIELDS()
    CV_TREE_NODE_FIELDS(CvSeq);
    int          total;          /* total number of elem
    int          elem_size;      /* size of sequence eler
    char*        block_max;      /* maximal bound of the
    char*        ptr;            /* current write pointe
    int          delta_elems;    /* how many elements al
    CvMemStorage* storage;       /* where the seq is sto
    CvSeqBlock* free_blocks;     /* free blocks list */
    CvSeqBlock* first; /* pointer to the first sequen

typedef struct CvSeq
{
    CV_SEQUENCE_FIELDS()
}
```