las buenas praticas del c++ moderno

par Rodrigo Pons (home)

Date de publication : 06/07/2010

Dernière mise à jour : 21/07/2010

Este artículo es un conjunto de reglas ampliamente aceptadas por la comunidad de los desarrolladores C++. Proponen maneras de resolver los problemas recurrentes que ocurren cuando se programa en C++. Describen los bases de lo que se llama el C++ moderno, cuales grandes representantes son la STL y boost.
En este artículo, no entramos en los detalles del C++, moderno o no. Entonces, es asequible por cualquier programador que tiene algunas bases del C++.

Introducción

Pienso que más o menos todos los desarrolladores, sobre todo en c++, leyeron u oyeron por lo menos una vez esta frase: "En programación hay mil formas de hacer la misma cosa". Y es la verdad (sobre todo en c++): para cada problema, siempre existen varias soluciones posibles. El C++, gracias a sus paradigmas múltiples, es posiblemente el lenguaje que ofrece el abanico más ancho de soluciones.
La primera etapa para un desarrollador es saber implementar algunas de estas soluciones. La segunda etapa consiste en saber determinar, entre estas varias soluciones posibles, cual es la mejor. Las buenas prácticas de las que hablo aquí tienen por objetivo de ser un conjunto de consejos que permiten ayudar en esta elección de la mejor solución.
Estos consejos están considerados como válidos en el caso general, pero evidentemente, siempre hay unas excepciones.

⚠ *Este artículo fue escrito antes de la oficialización del nuevo estándar (C++ 0x). Entonces, es probable que algunas de las recomendaciones escritas aquí se hacen falsas con este nuevo estándar.*

I - Los basicos

Antes de todo, el C y el C++ son dos languajes distintos. Tienen una raiz común, pero han evolucionado en direcciones distintas, y son cada vez más distintos. Por eso, escribir C/C++ es un error, es como escribir C/java: no tiene sentido. C y C++, C o C++, vale, pero C/C++ no vale.

I-A - Utilizar compiladores recientes

Esto es probablemente el más importante de todo. El c++ es un language que está evolucionando constantemente. Entonces si usted utiliza un compilador antiguo, es muy probable que este no respeta el estándar actual y el código que hacemos con el estará obsoleto. Y aun peor, va a aprender malamente el c++. La serie reciente de los Visual Studio de Microsoft es bastante buena. A partir de la version 9 (visual 2008), los compiladores respetan el estándar oficial mas reciente (de momento es el c++03, digamos). Igual por gcc, el compilador de GNU. Hay que notar que VS10 y las últimas versiones de gcc implementan ya ciertas funcionalidades del próximo estándar C++0x.
El ejemplo típico es Visual C++ 6, que sigue siendo bastante utilizado. Y eso es un error fatal. El principal problema con VS C++ 6 es que no respeta el estándar C++, por la simple y buena razón que el compilador de VS6 fue desarrollado antes de la salida del primer estándar oficial c++98. Existen numerosos otros IDEs (visual 7, visual 8, Code::Blocks, Anjunta, KDevelop, etc.) y compiladores (gcc, mingw, comeau, etc.) que reemplazarán ventajosamente a Visual 6.

I-B - NO volver a inventar la pólvora

Osea, **no** volver a programar funcionalidades que ya han sido progamados.
Sólo con las librerias estándar (SL y STL) y boost, la grande mayoría de los problemas corrientes estan resueltos ya. Ver por ejemplo **los contenedores** y **los algoritmos**.
Estos códigos (por ejemplo la STL, boost, Qt, etc.) han sido programados por programadores muy buenos, han evolucionado mejorandose, y son utilizados en miles programas. Entonces, suelen ser mas estables y eficientes que cualquier código que podemos escribir nosotros.

⚠ *Por supuesto, este consejo no se aplica a los principiantes que necesitan hacer ejercicios para entender los conceptos básicos del lenguaje.*

I-C - Utilizar las verdaderas cabeceras del estándar

Una cabecera es un fichero, generalmente con la extensión h o hpp (*fichero.h* o *fichero.hpp*).

Antes de que el C++ fue estandarizado, <iostream.h> estaba el único fichero cabecera (header en inglés) existente entregado con los compiladores de aquella época. La normalización ISO del C++ en 1998 definió que <iostream> (sin .h) sea la cabecera estándar para las entradas/salidas. La ausencia del *.h* señala que, a partir de ahora, es una cabecera estándar, y entonces, todas sus definiciones forman parte del espacio de nombres std (namespace std). Desde entonces, incluir <iostream.h> es obsoleto (técnicamente, <iostream.h> no es obsoleto porque nunca ha sido parte del estándar, pero su uso si).
Para dejar a los programadores el tiempo de modificar sus código, los compiladores proporcionaron cada uno de estos ficheros cabeceras. Por consiguiente, <iostream.h> está presente únicamente por razón de compatibilidad.
Pero ahora, algunos compiladores como Visual C++ 7 (2003) y versiones siguientes emiten al menos un aviso (warning) de obsolescencia.
Es lo mismo con todos los ficheros cabeceras estándar en C++, incluso con los de la liberia estándar C. Para razones de estandarización, ahora hay que incluirlos sin el .h, y prefijandolos con la letra *c* (para destacar el hecho de que vienen del C).

antigua cabecera	nueva cabecera
iostream.h	iostream
string.h	string
stdlib.h	cstdlib
stdio.h	cstdio

fuentes:
http://cpp.developpez.com/faq/cpp/?page=console#SL_iostream
http://cpp.developpez.com/faq/cpp/?page=strings#STRINGS_string_et_string_h

I-D - Parámetros de funciones: optar por la referencia

Por defecto, en C++, los parámetros de una función están pasado por valor, es decir que es una copia del parámetro que esta manipulado por la función, y no el original.

Entonces, la copia de este parametro puede resultar varios problemas:

- Puede generar una confusión de que el parámetro en el cual estamos trabajando dentro de la función no es el mismo que hemos pasado a la llamada de esta función.
- En caso de que el parámetro es un objecto compuesto (en contrasto con un tipo nativo), la copia es una perdida de tiempo.
- Puede ocurrir problemas cuando el parámetro es un objeto declarado constante, o no es copiable (por ejemplo los flujos estandars, iostream)

Para resolver este problema, utilizamos el pasaje por referencia (o - mejor cuando se puede - por referencia constante). Pasando una referencia, nos aseguramos que es bien el objeto inicial que manipulamos dentro de la función. Si la referencia es constante, nos aseguramos además que este objeto no puede ser modificado dentro de esta función. Entonces, un buen habito es, cuando un parametro de una función no tiene que ser modificado dentro de la función, utilizar sistemáticamente una referencia constante.
En vez de una referencia, se puede utilizar un puntero, pero se suelo aconsejar evitar los punteros cuando es posible evitarlos. En efecto, un puntero es un objeto (que contiene la dirección de objeto punteado, pero un puntero tambien lleva toda la "aritmética de punteros"), entonces es mas pesado (en termino de rapidez de ejecución y de uso de memoria) y mas complejo de uso (entonces los riesgos de errores son mas importantes con punteros que con referencias. Además, una referencia no puede ser invalida, pero un puntero si, y eso es probablemente la mas grande fuente de errores para los principiantes).

Referencias suelen ser preferibles a los punteros cuando no es necesario cambiar el objeto punteado (reset del puntero). Esto generalmente significa que las referencias son más útiles en la interfaz pública de una clase. Las referencias suelen aparecer en "la piel" de un objeto, y punteros en el interior.

Fuentes:

Las referencias en la página de Zator
Passage des paramètres de fonctions
Referencias en la FAQ parashift

I-E - Comprobar su código al mismo tiempo

Se aconseja comprobar su código cada vez que es posible durante el desarollo. Al principio, tendemos a programar lineas y líneas sin siquiera intentar compilar.

Es un error por varias razones:

- Más tardamos a comprobar el codigo, mas dificil sera la fase de despuración. Y a veces, puede ser una perdida de tiempo y de paciencia.
- Si se comprueba un código justo despues de escribirlo, estamos más sereno en cada paso, porque ya no tenemos que pensar en el anterior.
- A veces no es posible, o muy complicado, hacer lo que tenemos en la mente. Y si nos damos cuenta de eso después de haber escrito 1000 líneas de código, puede ser desastroso.

Me ocurrió hace poco. He programado un servidor SOAP (con gSoap) pensando que lo podre conectar sencillamente en un servidor Apache dado. Pasó que este enchufe (gSoap con Apache) no fue tan sencillo que parecía, y hemos tenido que hacer de otra manera. He tirado un día entero de desarollo.

Aconsejo, durante el desarollo de cualquier programa, tener siempre un proyecto "dum" abierto. Lo que llamo un proyecto "dum" es un proyecto "basura", vacio o casi, listo para probar las cosas que vamos a utizar (por ejemplo, si trabajamos en un programa que utiliza boost, pues el proyecto "dum" tiene que tener ya los includes y links configurados para compilar y probar). Y con este proyecto "dum", podemos probar trozos de código antes de ponerlo en nuestro programa.

I-F - No usar el tipo float

Cuando se manipulan números reales, aconsejan utilizar el tipo *double*. La razón basica es que es más eficiente y con mejor precisión. Leer el **GTOW #67**

I-G - Variables locales

En C, hay de declarar las variables al principio de una función, y no se puede declarer y inicializar en el mismo tiempo.

En C++ es al reves:

- 1. Hay que declarar una variable justo antes de que la vamos a utilizar
- 2. Hay que inicialisarla en mismo tiempo que la declaracion

1. En C++, solemos declarar una variable justo antes de utilizarla por la primera vez. La primera razón es que es ma sencillo. Asi no tenemos una lista de declaraciones al principio de las funciones. Le segunda razón es que actuando así, la pila (*stack*) se queda más limpia. Es porque una variable esta depilada cuando la ejecución sale de su ámbito (*range*).
2. Es un buen hábito de inicializar las variables en mismo tiempo que se declaran. Asi estamos siempre asegurados que las variables esten en un estado valido.

II - Construcción/instanciación de un objeto

Existen varios conceptos y fases en la existencia de un objeto:

- La declaración de una clase
- La definición de esta clase

- La instanciación de un objeto de esta clase
- La inicialización del objeto

Para más precisiones, ver **aquí**

II-A - Constructor: Utilizar la lista de iniciadores

De hecho, los contructores deberian inicializar todos sus atributos con la lista de iniciadores. Por ejemplo, el constructor siguiente inicializa la variable miembra con una afectación normal y corriente:

```
MyObject::MyObject( int x )
{
 x_ = x;
}
```

Y el siguente con una lista de iniciadores (en este caso solo hay uno).

```
MyObject::MyObject( int x )
: x_(x)
{
}
```

Estos dos constructores hacen casi la misma cosa. El resultado suele ser igual, pero hay alugnas ventajas al usar una lista de inciadores.
Primero, ganamos en eficacia (rapidez). Por ejemplo, en el segundo constructor, si *x* es igual à *x_*, el compilador no hace ninguna copia, pero afecta directamente el valor de *x_*. Aunque en el primero constructor, el compilador tiene que hacer una copia temporal de x.
Además, si los tipos de *x* y de *x_* son distintos, el compilador suele ser más capaz de resolver la ambigüedad.
Segundo, hay otro problema con el constructor por afectación. En el caso que la variable miembra *x* es un objecto (y no un tipo nativo), esta variable miembra estará construida por su propio constructor por defecto. Y a veces no se sabe exactamente lo que hace un constructor por defecto.
Conclusión: En igualdad de condiciones, el código se ejecutará más rápido si se utilizan listas de iniciadores en lugar de afectaciones.

No hay diferencia en el rendimiento si el tipo de x_ es nativo, como int o char o float. Pero incluso en este caso, mi preferencia personal es para inicializar los datos en la lista de iniciadores, para tener un código coherente. Otro argumento relacionado con la simetría en favor del uso de listas de iniciadores incluso para tipos básicos: el valor de los datos miembro constantes y non estaticos no puede estar inicializado en el constructor, entonces, para mantener la simetría, recomiendo inicializar todo en la lista de iniciadores.

Fuentes:
El capitulo de zator sobre las listas de iniciadores
La FAQ Parashift sobre las listas de iniciadores
La FAQ de developpez.com sobre las listas de iniciadores

II-B - La forma canónica ortodoxa de Coplien

En C++, hay 3 tipos de contructores y un destructor:

- Un constructor por defecto : MyClass();
- Un constructor por copia : MyClass(MyClass const &);
- El operador de afectación : MyClass& operator=(MyClass const &);
- El destructor: ~MyClass();

Si no son definidos explícitamente por el programador, cada uno de estos cuatro constructores/destructores son definidos automáticamente (implícitamente) por el compilador.

La forma canónica ortodoxa de Coplien es una manera de definir la construcción y la destrucción de una clase. Ha sido definido por Coplien para las clases a semántica de valor (clases cuyas objetivo es definir uno o varios valores. Por ejemplo, las clases que solo definen comportamientos no lo son). Señala que, si una clase debe definir una de las cuatro formas, debe definir todas:

- Si uno de estos cuatro constructores/destructor ha sido definido de manera no trivial, entonces es muy probable que los otros tres deben ser definidos.
- Si una clase debe ser la base para un uso polimórfico, el destructor debe ser declarado como virtual. Tenga en cuenta que es probable que entonces la clase sea no copiable.
- La semántica de una clase va a imponer la política de definición de las funciones miembras (¿Qué forma canónica adoptar de acuerdo con la semántica de la clase?).
- Si la definición implícita es apropiada, dada los tres puntos precedentes, no es necesario definir explícitamente estos métodos. Y definirlas puede tener un impacto negativo: hay herramientas para el análisis de código (y, posiblemente, el futuras evoluciones del c++) que verifican el hecho de que se han cumplido estas reglas. Definir una función que no hace nada puede derrotar a estas herramientas.

La FAQ de developpez.com sobre la forma canónica ortodoxa de Coplien
The big four
The rule of three

II-C - Instanciar los objetos en la pila (stack)

Esto significa que: MyObject object(/* params */);
es mejor que: MyObject * object = new MyObject(/* params */);

De manera general, en c++, solemos decir que hay que utilzar los punteros solo cuando no hay otra posibilidades.

- Por varias razónes, utilizar un puntero es más arriesgado que un objeto en la pila. Los dos razónes principales son que es más complicado manejar un puntero y sus operadores, y que un objeto en el montón (heap en inglés) tiene que ser destruido explícitamente, aunque un objeto construido en la pila (stack), está automaticamente destruido al salir del alcance.
- La construcción en la pila es más rapida que la construcción en el montón. Aunque esto es cada vez menos (gracias al progreso de los compiladores que optimizan cada vez mejor).
- El código es más sencillo y claro, que no es poco.

II-D - La palabre clave virtual

Una clase que está destinada a ser heredada debe tener su destructor declarado como virtual. ¿Porque?
Un poco de código para entender el porque:

```
class Madre
{
public:
 Madre() {} // constructor
 ~Madre() {} // destructor no virtual (lo que NO hay que hacer)
};

class Hija : public Madre
{
public:
 Hija( const std::string & name = "n/a" ) : Madre() , name_
 ( new std::string( name ) ) {} // constructor
 ~Hija() { delete name_; } // destructor
```

```
 const std::string & Name() const { return *name_; } // accessor

private:
 std::string * name_;
};

int main()
{
 // construcción de un objeto hija de tipo Madre y de un puntero sobre este objeto.
 Hija * hija = new Madre(); // Este punteo es de tipo Hija*.

 // código
 // ...

 delete hija; // destrucción del objeto apuntado por el puntero hija

 return 0;
}
```

Este código es un poco raro: la variable miembra name_ es un puntero y en nuestro caso es un error, pero suele occurir, por varias razónes, con ojetos de otros tipos (no con una string).
El main() de este código contruye un ojeto de tipo Madre y un puntero de tipo Hija* sobre este objeto. Esta forma de hacer es muy utilizada en C++ para utilizar el polimorfismo de herencia. Lo tipico es cuando queremos tener una colección (vector, lista, etc.) de objetos de tipos distintos que heredan todos de la misma clase madre.
En el código de arriba, pasamos ningun parómetro al contructor, entonces el campo *name_* tendrá el valor por defecto del contructor, es decir "n/a".
Al final del main(), destruimos este objeto hija. Pero ¿que pasa exactamente? Lo que pasa el lo siguiente:
hija es un puntero sobre un objeto de tipo Madre. Entonces, es el destructor de la clase Madre que está llamado. Cómo este destructor no es virtual, el destructor de la clase hija no es llamado, y entonces, la variable miembra *name_* no estará destruida. Eso puede engendrar varios problemas de memoria (huidas, corrupción del montón (heap corruption), etc.).
Por eso, hay que declarar el destructor de Madre en virtual:

```
class Madre
{
public:
 Madre() {} // constructor
 virtual ~Madre() {} // destructor virtual (OK)
};
```

Asi, cuando se destruye el objeto hija, el destructor de hija estará llamado y la memoria estará bien liberada.
FAQ developpez.com sobre el destructor virtual
Página de Zator sobre la palabre clave virtual

II-E - RAII (Resource Adquisition Is Initialization)

RAII se puede traducir, en castellano, por: "Adquirir un recurso es inicializarlo".
A mi parecer, está técnica lleva mal su nombre, porque lo que importa más es la liberación del recurso, y no la inicialización. Entonces seria mejor algo cómo "Resource Liberation Is Destruction" or algo así.
La idea de esta técnica es de asegurarse que en cualquier caso, **hasta en un contexto multihilo**, los recursos sean bien liberados (basicamente, aquí los recursos son la memoria (pila o montón), pero támbien puede ser cualquier recurso (BB.DD, periférico, etc.).
Entonces, esta técnica consiste en el hecho que nuestro objeto sea utilizable inmediatamente después de su construcción, y que estemos seguros que se ha liberado todos los recursos que utiliza solo destruyendole.
Entonces, para esto, hay que **encapsular** los recursos en una clase, y asegurarse que estos recursos esten inicializados limpiamente en el contructor de esta clase, y liberados en su destructor.
Un poco de código para ilustrar el asunto. La gestion de un fichro con el RAII (un fichero no es más que una forma particular de memoria):

```
class Fichero {
  FILE* fptr;

public:
  Fichero(char* file_name, char* mode) { // constructor
    fptr = fopen(file_name, mode);
  }
  ~Fichero() {            // destructor
    fclose(fptr);
  }
};
```

Este código es basicamente lo que hace la clase fstream de la STL (simplificado muchissimo, por supuesto).
Página de Zator sobre el control de recursos
Página de developpez.com el RAII

Los contenedores de la STL y de boost utilizan sistematicamente el RAII.

II-F - Liberación de la memoria dinámica.

Cuando gestionamos memoria dinámica, tenemos que asegurarnos siempre de que liberamos bien la memoria.

casos:

- Atributo de una clase: Podríamos reservar la memoria en el constructor, liberarla en el destructor y protegernos frente a copia con constructor copia y operador de asignación. También podemos usar smart_pointers.
- Variable de una función (miembro o no): Usar smart_pointers para protegernos de salidas prematuras o no controladas.
- Devolución por retorno de función (miembro o no), con transferencia de propiedad, de la memoria entre distintas entidades. Usar smart_pointers.
- Paso por parámetro, con transferencia de propiedad. Usar smart_pointers

III - Los contenedores

Como lo indica su nombre, un contenedor es un objeto que contiene otros objetos. El la base de la programación: colocamos cosas (valores, objetos, etc.) dentro de un contenedor para poder efectuar operaciones en serie (en forma de bucles) sobre esta colección de cosas.
Página de zator sobre los contenedores

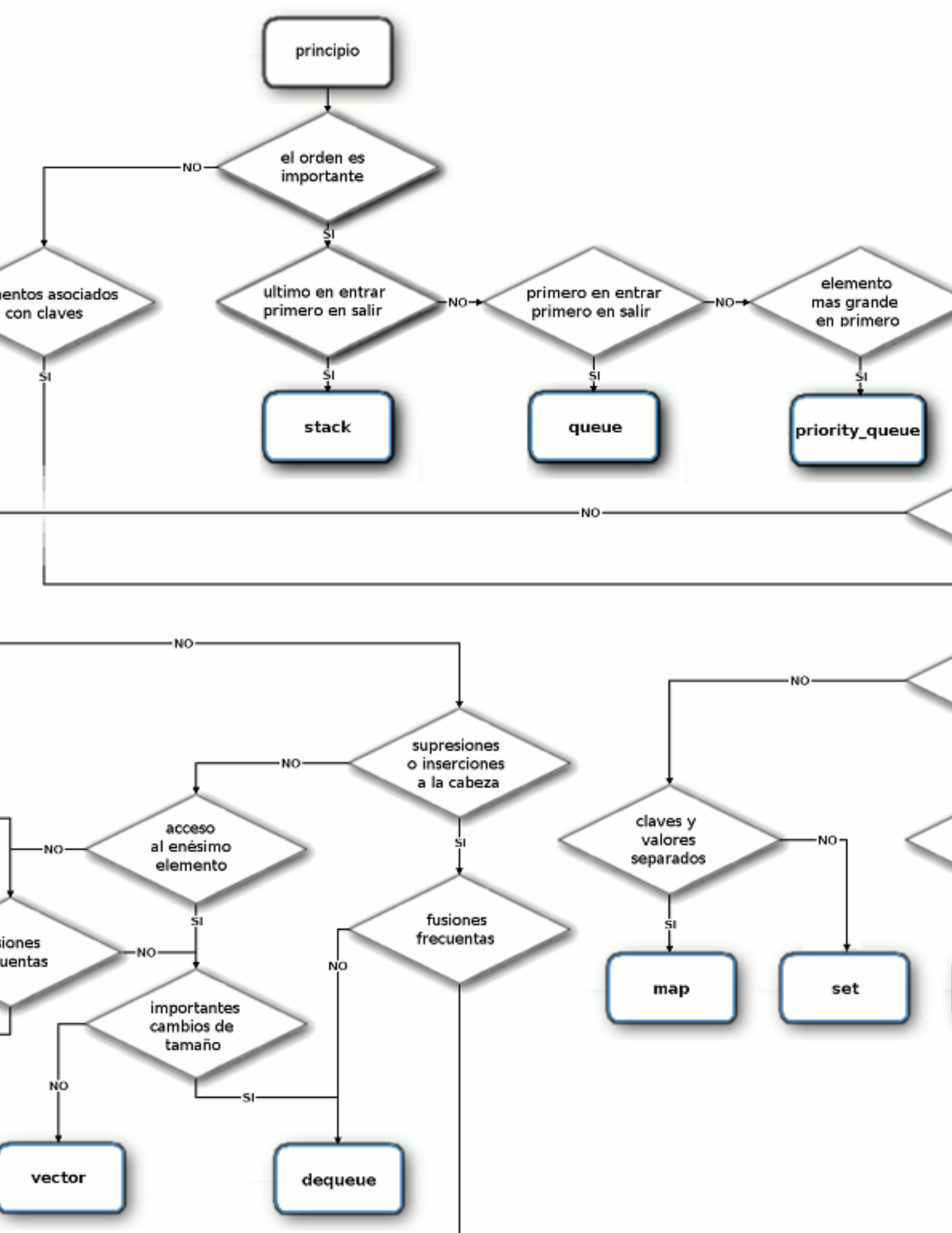
III-A - Utilizar los contenedores de la librería estándar

y no los "c-style array".
Lo que llamamos "c-style array", es un array de tipo C (y no de C++). Es decir que la memoria está reservada por un *malloc* o un *new*. Por ejemplo:
int my_array[10];
o
int * my_array = malloc(10 * sizeof(int));
La librería estándar (SL y STL) proporciona 10 contenedores y varias clases de manipulación de cadenas de caracteres (string, w_string, etc.) que, al final, son contenedores especializados para la gestión de cadenas.
Estos contenedores fueron desarrollados por los mejores expertos del lenguaje. Están estructurados segun sus usos y en la mayoria de los casos, es imposible hacer ma eficiente.

los clasicos son

- vector en vez del array "C-style": type[]
- string en vez de char*

Para eligir el contenedor que encaja mejor:


principio
el orden es importante
NO
SI
entos asociados con claves
ultimo en entrar primero en salir
NO
primero en entrar primero en salir
NO
elemento mas grande en primero
SI
SI
SI
SI
stack
queue
priority_queue
NO
NO
supresiones o inserciones a la cabeza
NO
SI
acceso al enésimo elemento
NO
SI
iones uentas
NO
importantes cambios de tamaño
NO
SI
fusiones frecuentas
NO
claves y valores separados
NO
SI
map
set
vector
dequeue

fuente: **este diagrama** de Laurent Gomilla.

boost implementa algunos contenedores también. Aquí están los principales:

- **array**: **http://www.boost.org/doc/libs/1_39_0/doc/html/array.html**
- **bimap**: **http://www.boost.org/doc/libs/1_39_0/libs/bimap/doc/html/index.html**
- **circular_buffer**: **http://www.boost.org/doc/libs/1_39_0/libs/circular_buffer/doc/circular_buffer.html**
- **disjoint_set**: **http://www.boost.org/doc/libs/1_39_0/libs/disjoint_sets/disjoint_sets.html**
- **dynamic_bitset**: **http://www.boost.org/doc/libs/1_39_0/libs/dynamic_bitset/dynamic_bitset.html**
- **graph**: **http://www.boost.org/doc/libs/1_39_0/libs/graph/doc/table_of_contents.html**
- **intrusive**: **http://www.boost.org/doc/libs/1_39_0/doc/html/intrusive.html**
- **multi_array**: **http://www.boost.org/doc/libs/1_39_0/libs/multi_array/doc/index.html**
- **property_map**: **http://www.boost.org/doc/libs/1_39_0/libs/property_map/property_map.html**
- **unordered**: **http://www.boost.org/doc/libs/1_39_0/doc/html/unordered.html**

cpp reference : muy buena documentación sobre la STL.

III-B - char* vs string

EL char * no es más que un caso especial de contenedor.
Hay que utilizar la clase string en lugar de char*, la asignación de strings en lugar de strcpy, así como otras funciones como sprintf. Hay casos excepcionales en los que se podrían usar algunos de esos tipos, pero tiene que haber una buena razón (rendimiento, evitar dependencias innecesarias cuya eliminación sea importante, etc.).

III-C - La encapsulación

La encapsulación es la manera teórica para definir el uso de estructuras en cualquier lenguaje de programación. Esto consiste simplemente en almacenar datos y/o funcionalidades en una estructura, de modo que cuando se desea acceder a tales datos y/o funcionalidad, es necesario pasar por esta estructura.
En otras palabras, esto quiere decir que vamos a "encerrar" datos y/o funcionalidades dentro de una estructura, de manera que los datos y/o las funcionalidades no sean accesibles por el resto del programa.
Es una manera de dividir su programa de una manera lógica, y de proteger los datos de un uso no querido. Una manera de dividir un problema grande en varios problemas más pequeños, facilitando su resolución.
En C++, y en cualquier lenguaje orientado a objetos, la encapsulación toma nuevas dimensiones, semánticas y lógicas, como las preocupaciones de la modularidad, genericidad en la manipulación de objetos, etc.
Un poco de código par ilustrar:
Queremos programar un algoritmo que utiliza con frecuencia la raíz cuadrada de un número dado. Con el fin de ahorrar tiempo de cálculo, vamos a almacenar el valor de la raíz cuadrada, para evitar tener que calcularla de nuevo cada vez.

```
class Raiz
{
 public:
  Raiz( double numero ) // constructor
  {
   value = sqrt(numero);
  }

  double value; // error
};
```

La clase Raiz no esta bien echa, porque no encapsula correctamente su variable miembra *value*. En efecto, si value es *public*, "todo el mundo" puede acceder y modificarla. Es como si esta variable fuese declarada en el ámbito global, y entonces no está encapsulada. Entonces, no estamos seguros que nunca, en el programa, se va a modificar esta variable.
Entonces lo que hay que hacer es declarar *value* como privado, y hacer un accesor:

```
class Raiz
{
 public:
  Raiz( double numero ) // constructor
  {
   value = sqrt(numero);
  }

  double Value() const { return value; } // accesor

 private:
  double value;
};
```

Una buena encapsulación, en C++, tiene otras implicaciones. Por ejemplo la semántica: por ejemplo no voy a utilizar mi clase *Raiz* para calcular otra cosa que la raiz cuadrada de un número. En este caso, es muy sencillo, porque es un ejemplo sencillo solo para ilustrar, pero en la "verdadera vida", las cosas suelen ser más complicadas, y siempre que añadimos una variable o una functionalidad en una clase, hay que preguntarse si es el bueno lugar, de un punto de vista semántico, pero tambien logico, para añadirlo.

La encapsulación no es siempre imprescindible. Por ejemplo, a veces no hace falta encapsular una variable miembra.

IV - Inclusión de ficheros

IV-A - Inclusion guard

Poner "Inclusion guards" en los headers.
Los "Inclusion guards" consisten en un #ifndef 'Nombre con el que queramos identificar el .h' situado como primera línea del .h y un #endif que cierre el #ifndef al final del mismo. Se usa para evitar inclusiones repetidas.

IV-B - Cuidad a los ficheros que se incluyen

Nunca incluir un fichero .cpp (salvo para los templates, y aun así, aconsejo poner todo el código en el .h, cuando la clase no es demasiado grande).
Incluir el mínimo posible de archivos en un .h. Porque su proprio .h va a ser incluido por otros ficheros, y entonces, los ficheros .h incluidos en su proprio .h van a ser incluidos tambien en los ficheros que van a incluir el suyo. Y eso suele ser una fuente de problemas variados (el más tipico es la redefinición de variables).
Entonces, para evitar la inclusión de .h en sus propios .h, hay que utilizar, cuando es posible, la **declaración adelantada**.

IV-C - No usar "using" en los headers.

Los namespaces se usan para evitar conflictos entre nombres de tipos similares. Por decirlo de otra forma, se acotan ámbitos asociados a conjuuntos de tipos. El using namespace podría considerarse como la publicación de dichos tipos para que tengan visibilidad global, momento a partir del cual pudiera haber colisión de nombres, por lo que dicha publicación debiera ser selectiva a nivel de unidad de compilación.
Si hacemos un using namespace en un .h, estamos propagando la publicación de los nombres a TODAS las unidades de compilación que la incluyan, lo cual nos impide ser selectivos.
De forma mas teórica, utilizar la cláusala *using* en un header rompe el contrato entre el namespace dicho y el código que le utiliza. Porque cuando un código está dentro de un namespace, es una forma de **encapsulación**, y si propagamos la publicación de estos nombres, rompemos esta encapsulación.

IV-D - Declaraciones adelantadas

Se deben usar declaraciones adelantadas (forward declarations en ingles) siempre que se pueda y que no suponga un trabajo extra que no compense. Las declaraciones adelantadas se pueden usar sólo si no necesita la definición del tipo cuyo include queremos evitar.
Por ejemplo, en vez de:

```
#include "B.h"

class A
{
public:
// código

private:
 B* b; // puntero sobre un objeto de tipo B
};
```

Es mejor:

```
class B; // declaración adelantada. El fichero B.h tendra que ser incluido en A.cpp

class A
{
public:
// código

private:
 B* b; // puntero sobre un objeto de tipo B
};
```

Es mejor porque incluir un .h en un otro .h **¡es mal!**

Se pueden dar varios casos:

- Que en la clase no tengamos el tipo por valor. Por referencia o puntero nos vale. Este caso es trivial y cambiamos el include por una declaración adelantada.
- Que en la clase tengamos el tipo por valor. En este caso deberemos cambiarlo a puntero, pero eso tiene conecuencias, ya que debemos crear el objeto en el constructor, destruirlo en el destructor y protegernos contra copia superficial (shallow copy). En este caso debemos decidir si ese sobre esfuerzo aporta ventajas que lo compensen. Por ejemplo, si en la interfaz pública lo usamos como parámetro o tipo de retorno, sabemos que quien use nuestor.

Fuera de los casos generales hay dos situaciones a comentar:

- Templates en general de tipos que no son nuestros. Por ejemplo, los contenedores de la STL tienen declaraciones complejas, que no podremos evitar porque la declaración adelantada de templates requiere sus argumentos. En caso de querer evitarlo, debieramos coger los tipos de esas clases e incluirlos como declaración adelantada en un header común, pero esto nos vale para un compilador, porque no tenemos garantizado que funcione con otro diferente.
- Iostreams. Es una particularización del caso anterior, que está contemplado en el standard mediante un header iosfwd que incluye esas declaraciones adelantadas, por lo que en este caso no tendremos problemas entre compiladores (cada uno llevará el suyo).

V - Una buena semántica

El C++ es un lenguaje de programación, y como cualquier lenguaje, sirve para comunicarse. Se utiliza para comunicarse con el ordenador, por supuesto, pero también con los seres humanos que van a leer el código. A

menudo es el mismo que lo escribe y que lo lee despues. Sin embargo, un código claro, que se entiende cuando se lee, es una excelente señal de calidad. Si el código es claro y legible, esto significa que:
- Los problemas han sido bien separados, y cuando se necesita cambiar algo, el código será fácil de localizar y entender.
- Es fácil de entender para que sirve una clase o una función, sin necesidad de ahondar en el código.
Por lo tanto, un código que respecte una buena semántica tiene varias ventajas.
Obviamente, permite a los programadores que lean el código más tarde (a menudo es la misma persona que lo escribió) para hacer cambios con mayor facilidad.
Pero también permite que el esta programando tenga una visión más clara de lo que está haciendo y lo que queda por hacer.
También ayuda a identificar más rápidamente y con más claridad cuando algo sale mal, o falta algo.

V-A - Nombrar bien los objetos

Aquí utilizo la palabra *objeto* en su sentido largo, es decir todo que puede ser nombrado en un programa (funciones, clases, variables, namespace, etc...).
Eso es la base de una buena semántica: al leer el nombre de una cosa, se debe saber lo que es y para que sirve.

V-B - Const-conformidad (const-correctness)

La palabre clave *const* mejora la semántica del código. En efecto, esta palable clave const da informaciones sobre el uso de esta variable, de para que sirve y como hay que utilizarla.
Por ejemplo, si en una función, recibo un parámetro que no quiero modificar, pues voy a pasar este parámetro por referencia constante, y así el que va a utilizar esta función sabra, sin tener que mirar el código de la función, que esta variable no estará modificada por la función.
Entonces, si su programa entero utiliza la palabre clave const de buena manera y **cada vez que se puede**, su programa estará dicho "const-conforme", y cuando lo vamos a utilizar, sabremos, para cada función, cada variable y cada clase si hay que preocuparse o no de la constancia. Y esto puede ser un ahorro de tiempo muy importante.
Página de zator sobre la palabre clave const
Gotw #6: const-correctness

La const-conformidad también tiene implicaciones en términos de eficiencia. Este problema (complejo y en evolución constante, al mismo tiempo a los compiladores y las normas), no es la cuestión tratada aquí, entonces no voy a hablar de esto. Sin embargo, si usted quiere saber más sobre este tema, lea el enlace siguiente: **Gotw #81: constant optimization**

V-C - Una instrucción por línea

Cuando empezamos a programar, todos estamos tentados a hacer el código lo más corto posible. Si es intelectualmente estimulante, no es necesariamente una buena cosa. Es mejor hacer un código claro, legible y, sobre todo, fácil de depurar y mantener. Por eso la idea de "una instruccioón por línea" es importante, porque facilita mucho la depuración y el mantenimiento. Por ejemplo el código siguiente esta malo:

```
a = ( b == 0 ) ? do_something() : do_something_else();
```

Este código esta malo porque se hacen 3 cosas en una línea. Por ejemplo, en caso de error a la compliación, y que el compilador nos indica la línea donde hay un error, no sabemos directamente donde esta este error.
Además, si el códido compila pero no hace lo que esta supuesto hacer, el problema es el mismo: es mucho más fácil ver donde está el problema cuando solo hay una instrucción por línea.
Por eso, el código arriba es mejor escrito así:

```
a = ( b == 0 )
 ? do_something()
```

```
  : do_something_else();
```

V-D - Una operación por función

Una función, que sea libre o miembre de una clase, tienen que ser consideradas como entidades logicas atómicas (atómica en el sentido de que no pueden ser dividas). Por ejemplo, si tengo una función que se llama *connect()*, esta función no debe hacer más cosas que abrir una connección. He visto código en el cual la función *connect()* abre una connección, pero que tambien envía datos de identificación. Eso es un error, porque son dos cosas distintas, y si tengo un fallo de identificación, va a ser mucho más complicado par encontrar donde está el problema.
Si realmente es necesario que una función haga varias cosas (lo que suele occurir), el nombre de esta función tiene que exprimirlo explícitamente. El el caso del *connect()*, por ejemplo, si por alguna razón queremos enviar datos de identificación, hay que llamar esta función *connect_and_identify()*.
De forma general, las funciones deben ser cortas (algunos recomiendan 10 líneas máximo por función).

Conclusión

Durante el proceso de desarrollo, siempre hay que tener en cuenta que el código que escribimos no sólo debe ser entendido por el compilador, sino también por otros programadores (quizá nosotros mismos).
Cada esfuerzo que hacemos en esta dirección será, tarde o temprano, tiempo ahorrado. Y muchas veces más temprano que tarde, ya que es casi siempre nosotros mismo que vamos a retocar nuesto código cuando vamos a corregir, despurar o mejorarlo.
Y además de ahorrar tiempo, tener buenos hábitos aclara la visión que tenemos de nuestro propio programa, y por lo tanto, del C++ y el desarrollo de software en general.