Clases abstractas no polimórficas para C++

Adolfo Di Mare

Universidad de Costa Rica
Escuela de Ciencias de la Computación e Informática
adolfo.dimare@ecci.ucr.ac.cr

RESUMEN

Se describe cómo lograr que en la clase derivada C++ sean reimplementados los métodos de su clase base. Esta solución usa plantillas en lugar de polimorfismo (métodos virtuales) lo que, en algunas ocasiones, le permite al constructor de programas lograr una mejor implementación.

Palabras clave

Clases abstractas, abstracción, especificación, implementación, ocultamiento de datos.

1. INTRODUCCIÓN

Mediante el uso de clases abstractas es posible lograr que el diseñador C++ construya clases que sirven como plantilla para las clases derivadas que usarán los programadores clientes de sus módulos. Las clases abstractas son un mecanismo del lenguaje que le permite al compilador verificar que la reimplementación de métodos se ha hecho cumpliendo con su especificación. En este escrito se muestra que la potencia expresiva de C++ permite lograr lo mismo usando plantillas y evitando así pagar el costo de funciones virtuales, lo que puede ser útil en muchas aplicaciones en las que la memoria o el tiempo de ejecución tienen gran prioridad.

2. LAS PLANTILLAS C++ COMO SUSTITUCIÓN DEL POLIMORFISMO

```
1  class Base {
2      public: virtual void doIt() = 0;
3  };
4  // ...
5  Base boom; // Error
```

```
|5| error: cannot declare variable `boom' to be of type `Base'
|5| error:   because the following virtual functions are abstract:
|2| error:   virtual void Base::doIt()
```

Figura 1

Si "`Base`" es una clase abstracta es porque contiene algún método "`Base::doIt()`" virtual o polimórfico que no está implementado. No es posible crear objetos de tipo "`Base`"; si alguno fuera creado, al ejecutarle su método "`doIt()`" se produciría un error en tiempo de ejecución porque ese método no ha sido definido (pues en la VMT, la tabla de métodos virtuales de la clase "`Base`", el puntero para el método "`doIt()`" invoca una rutina de error). En la Figura 1 se muestra que el compilador detecta este tipo de error si cualquier programador cliente trata de declarar un objeto de tipo "`Base`".

La diferencia entre "definir" y "declarar" un objeto en C++ es que las declaraciones se ponen en los archivos de encabezados `<*.h>` y `<*.hpp>`, mientras que las definiciones están en los archivos de implementación `<*.c>` y `<*.cpp>`.

- Las declaraciones corresponden a la especificación.
- Las definiciones corresponden a la implementación.

Para facilitar memorizar este hecho vale la pena asociar la palabra "definición" con la directiva #define que sirve para implementar macros en C++:

```
#define max(a,b) ( (a)>(b) ? (a) : (b) ).
```

```
 1  template <class X>
 2  void funBase( X & obj ) {
 3      obj.setVal( 5 );
 4  }
 5
 6  class TheBase {
 7      protected: void setVal( int arg );
 8  };
 9
10  class Derived : public TheBase {
11      public: void setVal( int arg ) { /* ... */ }
12  };
13
14  void call_ok() {
15      Derived val;     // ok
16      val.setVal( 12 );
17      funBase( val );
18  }
```

Figura 2

En la Figura 2 se define la función de biblioteca "`funBase<>()`" como una plantilla, aún antes de declarar las clases "`TheBase`" y "`Derived`". Será cuando esa función sea invocada que el compilador podrá aplicar la plantilla e instanciar la función para determinar si hay errores. Luego está declarada la clase "`TheBase`" que contiene al método "`setVal()`", pero está declarado como protegido (y además no ha sido implementado en ningún sitio). Como "`setVal()`" es un método protegido, si cualquier programador cliente declara un objeto de tipo "`TheBase`", al invocar el método "`setVal()`" o al usar la función de biblioteca "`funBase<>()`" se produciría un error de compilación. El compilador acepta sin problemas todo este bloque de código, pero si se declara la variable "`val`" con el tipo "`TheBase`" el compilador detecta varios errores y emite los mensajes correspondientes.

```
14  void call_error() {
15      TheBase val;     // Error !!!
16      val.setVal( 12 );
17      funBase( val );
18  }
```

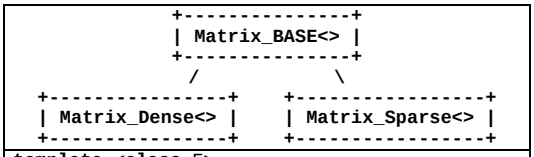
```
|   |In function `void call_error()':
| 7|  error: `void TheBase::setVal(int)' is protected
|16|  error: within this context
|   |In function `void funBase(X&) [with X = TheBase]':
|17|  instantiated from here
| 7|    error: `void TheBase::setVal(int)' is protected
| 3|    error: within this context
```

Figura 3

En la Figura 3 se muestra el mensaje de error que el compilador emite no dice que la clase "`TheBase`" ha sido usada para declarar una variable, como sí está explícitamente dicho cuando se trata de instanciar la clase abstracta "`Base`" en la Figura 1. Sí se hace referencia a que es impropio utilizar un método protegido, que es visible únicamente para clases derivadas: como la función "`call_error()`" no es un método de una clase derivada, apropiadamente el compilador emite el error. La única diferencia de la implementación de esta función y "`call_ok()`" es el uso incorrecto de la clase "`TheBase`" en lugar de la clase "`Derived`" para la que el método "`setVal()`" sí está definido. En otras palabras, usando el truco de declarar como protegido en la clase base el método que se desea reimplementar para toda clase derivada, pero no definiéndolo, se obtiene el apoyo del compilador para que éste verifique que en la clase derivada se ha usado la correcta especificación e implementación. "`TheBase`" es una clase abstracta implementada sin usar polimorfismo. (Aún si se incluye la implementación para esta operación se obtendría un error de compilación).

3. USOS DE CLASES ABSTRACTAS NO POLIMÓRFICAS

La primera aplicación de las clases abstractas no polimórficas es usarlas en el aula, para mostrarle a los estudiantes las ventajas de separar la especificación de la implementación de un módulo. Para mostrar varias implementaciones diferentes del mismo objeto basta declararlo como una clase abstracta base, incluyéndole únicamente sus operaciones, para luego incorporar en cada implementación la definición de los métodos y funciones apropiados.




```
template <class E>
class Matrix_BASE {
    public:
        typedef E value_type;
        E & operator() (unsigned, unsigned);
    friend
        Matrix_BASE operator+ (
            const Matrix_BASE& A,
            const Matrix_BASE& B );
}
```

Figura 4

En la Figura 4 se muestra la clase "`Matrix_BASE<>`" que contiene las operaciones más importantes para matrices. En las clases derivadas, que en este caso son "`Matrix_Dense<>`" y "`Matrix_Sparse<>`", están reimplementados los métodos de la clase base. Como complemento a este ejemplo se puede usar una biblioteca de funciones, en las que posiblemente se encuentren las funciones "`isScalar<>()`" o "`isDiagonal<>()`" que verifican alguna propiedad de la matriz utilizando únicamente los métodos públicos de la implementación.

```
template <class Mat>
bool isDiagonal( const Mat& M ) {
    if ( M.rows() != M.cols() ) {
      return false;
    }
    typename Mat::value_type ZERO = 0;
    for (unsigned i=1; i < M.rows(); i++) {
        for (unsigned j=0; j < i; j++) {
            if ( M(i,j) !=  ZERO ) {
                return false;
            } else if (M(j,i) !=  ZERO) {
                return false;
            }
        }
    }
    return true;
}
```

Figura 5

En la Figura 5 está la implementación de la función de biblioteca "`isDiagonal<>()`" que utiliza los métodos públicos de la clase para determinar si la matriz es o no una matriz diagonal. Al definir la constante "`ZERO`" es necesario usar la palabra reservada C++ "`typename`" para que informarle al compilador que el identificador "`value_type`" es el nombre de un tipo o de una clase, pues es la plantilla "`isDiagonal<>()`" debe ser compilada antes compilar cualquier uso de la versión de las clases a la que esta función será aplicada y, por ende, el compilador no tiene forma de adivinar que "`value_type`" es un tipo. En la práctica ocurrirá que la matriz "`Matrix_BASE<>`" seguramente esté definida en el archivo de encabezado "`Matrix_BASE.h`" y la biblioteca de funciones seguramente estará definida en otro archivo de encabezado, por ejemplo "`Matrix_Lib.h`", el que no necesariamente será usado siempre por todos los programadores. Inclusive el orden de inclusión de estos 2 archivos es irrelevante, pues es válido incluir uno u otro de primero.

Otro ejemplo importante del uso de esta técnica es amalgamar el acceso a archivos. Para esto hay que definir la clase base de acceso "`FILE_Base<>`" que contiene los verbos de acceso más importantes, como "`open<>()`" y "`close<>()`". Luego, en las clases derivadas se incluye la implementación específica para cada tipo de acceso.

En la Figura 6 (de la siguiente página) se muestra como es posible amalgamar el acceso a archivos Btrieve y DBase usando la interfaz común definida en "`FILE_Base<>`". Este enfoque se compara muy favorablemente con otros como el expuesto en [1], y se puede usar para concretar una clase genérica para acceso a tablas o bases de datos.

```
#include <iostream> // cout, etc
struct Person {
    char name[26];
    int  age;
};

template <class X>
class FILE_Base {
    X m_buffer;
protected:
    void open( const char* fileName );
    void close();
    void find( const char * key );
    bool hasNext();
    void write( const X& val  );
public:
    void set( const X& val ) { m_buffer = val; }
    X* operator->() { return & m_buffer; }
};
```

```
template <class X>
class FILE_bt: public FILE_Base<X> {
    void *Btrieve_FBLOCK;
public:
    void open( const char* fName ) {}
    void close() {}
    void find( const char * key ) {}
    bool hasNext() { return false; }
    void write( const X& val) {}
};

void use_Btrieve() {
    FILE_bt< Person > F;
    F.open( "data_file.bt" );
    {   using namespace std;
        F.find( "Phoebe" );
        while ( F.hasNext() ) {
            cout << F->name << F->age;
        }
    }
    F.close();
}
```

```
template <class X>
class FILE_dbf: public FILE_Base<X> {
    void *DBase_BUFF;
public:
    void open( const char* fName ) {}
    void close() {}
    void find( const char * key ) {}
    bool hasNext() { return false; }
    void write( const X& val) {}
};

void use_DBase() {
    FILE_dbf< Person > F;
    F.open( "data_file.dbf" );
    {   using namespace std;
        F.find( "Phoebe" );
        while ( F.hasNext() ) {
            cout << F->name << F->age;
        }
    }
    F.close();
}
```

Figura 6

4. DETALLES DE IMPLEMENTACIÓN

Trasladar a la práctica la idea básica expresada en la Figura 2 requiere de un poco de ingenio pues es inevitable encontrar problemas al usar el lenguaje para algo más allá de lo que fue esperado por su diseñador [5]. La organización de las funciones amigas y los métodos en la clase permite ver cuáles son los métodos abstractos cuya implementación está definida en la clase base "`Matrix_BASE<>`". En el código fuente están agrupadas de manera que sea sencillo reutilizarlas, pues en un solo bloque de código están todas las implementaciones juntas. Además, el constructor por omisión de la clase sí tiene que estar definido porque el compilador lo necesita para inicializar la base de todo objeto de tipo "`Matrix_BASE<>`"; lo mismo sucede con el destructor. Lo demás métodos no están implementados pues son métodos abstractos, cuya implementación debe definirse en las clases derivadas.

Otra dificultad surge con la especificación del operador de suma para matrices, cuya declaración natural es ésta:

```
template <class MAT>
MAT operator+( const MAT& A, const MAT& B );
```

Desafortunadamente, cuando se usa la matriz junto con otras clases, como por ejemplo "`std::string<>`" u otra similar, esta definición produce un conflicto de ambigüedad, pues esta misma definición también calza como operador para la clase "`string<>`". La forma de evitar este choque es declarar "`A`", el primer parámetro, de un tipo marcado por la matriz. Con esta modificación, ya la plantilla de matrices no calza con parámetros de otro tipo y así queda eliminado el conflicto con la clase "`string<>`". Esta es la declaración del primero parámetro "`A`":

```
const Matrix_BASE<typename MAT::value_type>& A;
```

Hay que usar punteros para transformar esta referencia a la clase base en una referencia a la clase derivada; por eso la implementación incluye este renglón:

```
MAT Res = *( (MAT*)(&A) );
```

que deja en la variable "`Res`" una copia del primer parámetro "`A`".

```
/// A+B
template <class MAT> inline MAT operator- (
  const Matrix_BASE< typename MAT::value_type >& A,
  const MAT& B
) {
    MAT Res = *( (MAT*)(&A) );
    add_Matrix(Res,B);
    return Res;
}
```

```
/// Res += M
template <class MAT>
void add_Matrix( MAT& Res, const MAT& M ) {
    // verifica que las dos matrices sean del mismo tamaño
    assert( "Matrix_BASE<E>::add()" && (Res.rows()==M.rows()) );
    assert( "Matrix_BASE<E>::add()" && (Res.cols()==M.cols()) );

    for ( unsigned i=0 ; i<Res.rows() ; ++i ) {
        for ( unsigned j=0 ; j<Res.cols() ; ++j ) {
            Res.operator()(i,j) += M(i,j);
        }
    }
    return;
}
```

Figura 7

En la parte superior de la Figura 7 está la implementación de la operación de suma en la que se usa el primer parámetro "`A`" de la rutina como una etiqueta que evita que la plantilla sea instanciada para tipos que no están derivados de la clase "`Matrix_BASE<>`"; el segundo parámetro "`B`" tiene el tipo correcto para ambos parámetros. Debido a que el tipo del parámetro "`A`" es "`Matrix_BASE<>`", esta versión de "`operator+<>()`" solo se puede aplicar a objetos derivados de la clase "`Matrix_BASE<>`" y, por eso, no hay posibilidad de ambigüedad cuando se usan matrices y otros tipos para los que ha sido sobrecargado "`operator+<>()`".

En esta implementación de la suma de matrices se usa la función auxiliar "`add_Matrix<>()`" para hacer el trabajo de sumar (la diferencia entre "*función*" y "*método*" es que la función nunca tiene un parámetro "`this`", pues existe independientemente de cualquier clase, mientras que el método siempre está relacionado a una clase). Esta función es una implementación abstracta de la suma de matrices, la que puede ser reutilizada en las clases concretas derivadas de "`Matrix_BASE<>`", a menos que sea más conveniente utilizar una implementación específica más eficiente. Por ejemplo, como la clase "`Matrix_Dense<>`" está implementada usando un vector unidimensional que contiene todos los valores de la matriz, la implementación más eficiente consiste en recorrer los 2 vectores por sumar para obtener el resultado. Esta implementación contrasta con la usada para "`Matrix_Sparse<>`", que sí reutiliza la implementación de la clase abstracta.

```
template <class T>
void add_Matrix( Matrix_Dense<T>& Res, const Matrix_Dense<T> & M ) {
    // verifica que las dos matrices sean del mismo tamaño
    assert( "Matrix_Dense<E>::add()" && (Res.rows()==M.rows()) );
    assert( "Matrix_Dense<E>::add()" && (Res.cols()==M.cols()) );

    T *pRes =   Res.m_val;
    T *pM   = & M.m_val[0];
    T *pEND = & Res.m_val[M.m_cols * M.m_rows];
    for ( ; pRes != pEND; ++pRes, ++pM ) {
        *pRes += *pM;
    }
    return;
}
```

Figura 8

Si cualquier programador utiliza matrices de tipo "`Matrix_Sparse<>`" para sumarlas, el compilador usará la plantilla de la suma de la parte superior de la Figura 7 y, por ende, también instanciará la plantilla de que está en la parte inferior de esa misma figura. En contraste, si la matriz es de tipo "`Matrix_Dense<>`", el compilador siempre usará la misma plantilla para "`operator+<>()`" pero usará la implementación de "`add_Matrix<>()`" de la Figura 8 porque es más específica

que la implementación general que aparece en la parte de abajo de la Figura 7. De esta manera no es necesario reimplementar un método abstracto si la implementación provista en la clase base es adecuada, pero sí es posible aportar una implementación específica en los casos en que esa sea el mejor enfoque.

Un problema que surge al usar "`Matrix_BASE<>`" como etiqueta para marcar el parámetro de la suma es que el compilador no puede hacer la conversión automática de un escalar en una matriz, por lo que este código no compila:

```
V = 2 + A;
```

Hay que usar una conversión explícita invocando directamente al constructor de la clase:

```
V = Matrix_List<unsigned>(2) + A;
```

Esta limitación no es exclusiva del uso de clases abstractas no polimórficas, sino que es compartida por cualquier clase C++ para la que se usen operadores sobrecargados (también es correcto argumentar que para evitar errores cualquier programador debe ser consciente de la conversión de un escalar en una matriz y, por eso, no conviene que esa conversión sea automática).

Es perfectamente válido definir campos en la clase abstracta. Sin embargo, para el caso específico de esta matriz, no hace falta incorporar ningún campo común en "`Matrix_BASE<>`" por lo que esa clase solo tiene miembros que son métodos.

Es una buena práctica de construcción de programas especificar y documentar módulos, usando herramientas como Doxygen [6]. Si se usa el enfoque descrito en [2] para definir las especificaciones, ayuda mucho el uso del comando "`\copydoc`" que permite copiar la especificación de los métodos abstractos de la clase base en la documentación de la clase derivada. En la implementación de "`Matrix_BASE<>`" y todas sus clases derivadas se ha aprovechado esta facilidad de Doxygen.

Es interesante expandir este trabajo agregándole restricciones al código similares a las expuestas en [3] o aprovechar nuevas constucciones sintácticas del lenguaje como los "*conceptos*" para las plantillas C++ que han sido propuestos para el nuevo estándar para el lenguaje C++ [4]. El uso de etiquetas para los operadores aritméticos es un paso en esa dirección.

5. MÉTODO DE USO

Para obtener una clase abstracta no polimórfica se pueden seguir estas prácticas de programación:

- Los métodos abstractos deben ser declarados "`protected`".
- Los métodos abstractos no deben ser implementados para que el compilador detecte el uso incorrecto de la clase abstracta base.
- Los constructores y destructores de la clase abstracta base deben ser públicos para evitar errores de compilación.
- Es posible incluir implementaciones para algunos métodos abstractos usando funciones emplantilladas amigas, en las que se incluya un parámetro que sustituya al argumento "`this`" de los métodos. Por ejemplo, para el método abstracto "`set<>(a,b)`" se puede implementar la función "`set_BASE<>(S,a,b)`" que en donde "`S`" es "`*this`".
- La reutilización del código implementado en la clase abstracta se logra invocando a la función amiga de cada método. Este es un ejemplo de cómo implementar el método en la clase derivada:

```
template <class E>
void Derived<E>::set( int a, int b ) {
    set_BASE<E>(S,a,b);
}
```

- Ayuda bastante acomodar el código de manera que queden juntas las operaciones abstractas no polimórficas.
- Es conveniente utilizar el comando "`\copydoc`" de Doxygen para reutilizar la especificación definida para los métodos abstractos no polimórficos.
- El uso de clases abstractas no polimórficas requiere de un buen conocimiento de las cualidades y restricciones de la parametrización en C++, por lo que el programador que vaya a usar esta forma de programación debe conocer bien cómo usar plantillas.

Los ejemplos que se mencionan al final de este artículo pueden ayudar al quienes opten por usar esta técnica de programación en programas reales.

6. CONCLUSIONES

Una de las contribuciones más importantes de la programación orientada a los objetos [OOP] es mostrar la importancia de separar la especificación de la implementación. Al utilizar clases abstractas no polimórficas se obtiene el beneficio de las clases abstractas sin obligar al programador a pagar el costo de uso de métodos virtuales, lo que son invocados indirectamente a través de punteros almacenados en cada instancia de un objeto. El enfoque aquí expuesto tiene al menos 2 aplicaciones inmediatas:

- Permite implementaciones diversas de una clase.
- Permite amalgamar el acceso a archivos diversos.

La primera aplicación es muy relevante en cursos de programación pues le permite a los profesores mostrar varias implementaciones para la misma clase. La segunda le puede facilitar el trabajo a programadores profesionales que necesitan construir módulos o bibliotecas eficientes y portátiles para muchas plataformas. Es importante combinar esta práctica de programación con el uso de herramientas de documentación como Doxygen, para facilitar la especificación y documentación de módulos de programación.

7. AGRADECIMIENTOS

David Chaves y Alejandro Di Mare aportaron varias observaciones y sugerencias para mejorar este trabajo.

8. REFERENCIAS

[1] Di Mare, Adolfo: "genridx.h" Una interfaz uniforme para el uso de archivos indizados en C, Revista Acta Académica, UACA, No.15, pp [35-58], ISSN 1017-7507, 1994.
http://www.di-mare.com/adolfo/p/genridx.htm
http://www.di-mare.com/adolfo/p/src/genridx.zip
http://www.uaca.ac.cr/acta/1994nov/genridx.htm

[2] Di Mare, Adolfo: Uso de Doxygen para especificar módulos y programas, I Congreso Internacional de Computación y Matemática, (CICMA-2008), celebrado del 21 al 23 de agosto en la Universidad Nacional (UNA), Costa Rica, I.S.B.N.: 978-9968-9961-1-5, 2008..
http://www.di-mare.com/adolfo/p/Doxygen.htm

[3] Meyers, Scott: Enforcing Code Feature Requirements in C++, Artima Developers, 2008.
http://www.artima.com/cppsource/codefeatures.html

[4] ISO/IEC ©: Working Draft, Standard for Programming Language C++, N2798=08-0308, 2008.
http://www.open-std.org/
jtc1/sc22/wg21/docs/papers/2008/n2798.pdf

[5] Stroustrup, Bjarne: The C++ Programming Language, 3rd edition, Addison-Wesley; 1998.
http://www.research.att.com/~bs/3rd.html

[6] van Heesch, Dimitri: Doxygen, 2005.
http://www.doxygen.org/index.html

9. CÓDIGO FUENTE

Formato HTML:

```
    http://www.di-mare.com/adolfo/p/BASE.htm

BASE.zip:
    http://www.di-mare.com/adolfo/p/BASE/BASE.zip
    http://www.di-mare.com/adolfo/p/BASE/es/index.html
    http://www.di-mare.com/adolfo/p/BASE/en/index.html
Matrix_BASE<>:
    http://www.di-mare.com/adolfo/p/BASE/es/classMx_1_1Matrix__BASE.html
    http://www.di-mare.com/adolfo/p/BASE/en/classMx_1_1Matrix__BASE.html
Matrix_Dense<>:
    http://www.di-mare.com/adolfo/p/BASE/es/classMx_1_1Matrix__Dense.html
    http://www.di-mare.com/adolfo/p/BASE/en/classMx_1_1Matrix__Dense.html
Matrix_List<>:
    http://www.di-mare.com/adolfo/p/BASE/es/classMx_1_1Matrix__List.html
    http://www.di-mare.com/adolfo/p/BASE/en/classMx_1_1Matrix__List.html
Matrix_Sparse<>:
    http://www.di-mare.com/adolfo/p/BASE/es/classMx_1_1Matrix__Sparse.html
    http://www.di-mare.com/adolfo/p/BASE/en/classMx_1_1Matrix__Sparse.html
Matrix_Lib<>:
    http://www.di-mare.com/adolfo/p/BASE/es/Matrix__Lib_8h.html
    http://www.di-mare.com/adolfo/p/BASE/en/Matrix__Lib_8h.html
Gaussian_Elimination<>:
    http://www.di-mare.com/adolfo/p/BASE/es/Gaussian__Elimination_8h.html
    http://www.di-mare.com/adolfo/p/BASE/en/Gaussian__Elimination_8h.html
test_Matrix.cpp:
    http://www.di-mare.com/adolfo/p/BASE/es/test__Matrix_8cpp.html
```