Alternativas para Implementar Clases Genéricas en C++

Marco V. Alvarado
SYSDE
Apdo 12133-1000
San José, Costa Rica
malvarad@cariari.ucr.ac.cr

Ricardo Villalón
SofTec
Apdo. 591-2100 Guadalupe
San José, Costa Rica
villalon@chacal.ecci.ucr.ac.cr

Marcelo Jenkins
Escuela de Computación e Informática
Universidad de Costa Rica
San José, Costa Rica
mjenkins@cariari.ucr.ac.cr

Resumen

La genericidad es una técnica muy importante para el desarrollo software reusable. En este artículo se discute el conepto de genericidad dentro del ámbito del lenguaje C++, y se presenta un análisis comparativo de dos implementaciones diferentes de una estrutura de datos genérica. Para finalizar, se proporcionan criterios que sirven como base en la comparación de ambos casos. Los puntos descritos en este artículo pueden interesar a aquellos que desean implementar componentes de software reusable.

Palabras clave: lenguajes de programación, programación orientada a objetos, C++.

1. Introducción

El programador de C++ está acostumbrado, en sus proyectos de programación, a contar con múltiples opciones al seleccionar la implementación particular que tendrán sus programas, gracias a la variedad de mecanismos de implementación que provee este lenguaje. Tal variedad le da flexibilidad, pero a la vez se puede prestar para mal interpretar la utilización que debe darse a las características y ventajas que brinda la programación orientada a objetos (POO), provocando que se seleccione una implementación no óptima, aunque si funcional, para resolver un problema particular.

En este artículo se trata el concepto de genericidad aplicado al diseño de clases en el lenguaje C++, mostrando posibles implementaciones de una pila genérica por medio de la clase contenedora *stack*. Para ello se implementa esta clase usando plantillas (*templates*) y herencia. Finalmente, se realiza un análisis de ambas implementaciones.

2. Marco Teórico

El *Polimorfismo* permite que las entidades de datos puedan tener más de un tipo (o clase). En [5]

los autores identifican dos clases de polimorfismo: universal y ad-hoc. Operaciones que son *polimórficas universalmente* trabajan uniformemente sobre un número infinito de tipos. Porlimorfismo *ad-hoc* trabaja sobre un conjunto finito de tipos posiblemente no relacionados. Este último es usualmente implementado usando *sobrecarga* de operadores o *coerción* de operandos.

3. Genericidad

El término genericidad es usado hoy día ampliamente en el ambiente de programación por objetos. "Genericidad es la habilidad de parametrizar un elemento de software (e.g., un paquete o subprograma en Ada) con uno o más tipos. Por ejemplo, un programa genérico para intercambiar valores (*swap*) operará sobre varios tipos de parámetros" (ver Blair et al. [2]).

En ocasiones es necesario imponer algunas restricciones al implementar genericidad, las que recaen sobre el tipo de los parámetros del componente genérico. Las restricciones aplican a las operaciones que el tipo del parámetro debe soportar. Por ejemplo, al implementar una lista ordenada genérica, es necesario que los elementos de la lista posean alguna relación de orden entre sí, lo cual se puede implementar a través de los operadores de orden menor que o mayor que.

El empleo de la genericidad en lenguajes de programación permite escribir código reusable, ya que el código genérico puede ser utilizado innumerable cantidad de veces en aplicaciones sustancialmente diferentes. Por ejemplo, una pila genérica podría ser implementada para crear una pila de números enteros, una pila de hileras, o un apila de algún tipo definido por el programador.

Al aplicar este concepto a la programación orientada a objetos, es posible obtener resultados interesantes, si se piensa en clases genéricas en las cuales se utilicen también las características de herencia y polimorfismo propias de la POO.

A continuación se describe cómo podrían crearse tipos genéricos con plantillas y con herencia, y las características esenciales de cada una de estas implementaciones.

4. Dos formas de implementar tipos genéricos en C++

Al implementar tipos genéricos es importante considerar el concepto de polimorfismo por cuanto, a través de los métodos polimórficos, es que realmente se pueden definir y mantener clases genéricas en C++.

Las dos formas de implementar tipos genéricos que serán discutidas, haciendo uso del polimorfismo, son: utilizando *plantillas o templates*, y usando

herencia con funciones virtuales. Según Barton y Nackman, "la forma de plantillas usa nombres comunes en clases similares; la forma de función virtual usa prototipos de funciones comunes en clases base" [1].

En esta discusión de tipos genéricos se consideran las clases contenedoras (por ej. pilas, colas, listas, etc.) para desarrollarlas, por cuanto son un buen ejemplo de clases reusables, que pueden ser implementadas en forma genérica.

4.1 Tipos genéricos usando plantillas

La definición e implementación de clases contenedoras usando plantillas en C++, se puede realizar por medio de los template; por ejemplo, una clase `stack` genérica puede definirse de la siguiente forma:

```
template <class T>
  class stack {
  private:
    ...
  public:
    ...
};
```

donde `class T` debe ser sustituido por el tipo correcto de los elementos de la pila en el momento de instanciarla o al momento de definir nuevos tipos mediante el `typedef`. Por ejemplo, la invocación correcta para definir una pila de números enteros sería:

```
typedef stack<int> TStackInt;
```

La implementación de la clase `stack<T>` utilizando plantillas, se basa en la capacidad de estas últimas de incluir en forma estática los tipos de los elementos que forman la pila. La flexibilidad que aporta es el hecho de que estos tipos pueden ser declarados en el momento de utilizar la pila (en forma explícita dentro del código fuente), convirtiéndose en una clase parametrizada.

4.2 Tipos genéricos usando herencia

Las clases contenedoras creadas mediante herencia permiten trabajar los punteros a instancias de cualquier subclase como punteros a cualquiera de sus clases base; reduciendo así el problema de crear una pila genérica, a definir una clase contenedora de punteros a instancias de alguna clase base.

Sin embargo, las clases de las instancias que se insertan, deben cumplir con cierta funcionalidad básica dependiente del tipo de contenedor que se está definiendo. Por ejemplo, podrían requerir un constructor de copia, el operador de asignación, etc.

El mecanismo para forzar a que las clases derivadas deban implementar algunas operaciones es definiendo métodos virtuales puros en la(s) clase(s) base. Por ejemplo, la clase `stackItem` definida a continuación se utiliza más adelante para definir una pila genérica usando herencia.

```
class stackItem;
typedef stackItem* pStackItem;

class stackItem {
   friend class stack;
protected:
   virtual pStackItem clone() = 0;
   virtual ostream& write(ostream& os) = 0;
   friend ostream& operator<<(ostream&,
        stackItem&);
  };
```

Note que las funciones `clone` y `write` son funciones virtuales puras, por lo que deben ser redefinidas en las subclases de `stackItem`.

5. Definición de una pila genérica

5.1 Descripción Matemática:

Sea p una lista compuesta por elementos x_i " i I N, tal que $x_0 << x_i << x_z$, donde x_z siempre es el último elemento de la lista. Se dice que la estructura de datos p es una *pila*, si toda inserción de elementos en p tiene lugar luego de la posición z de p, y toda supresión de elementos de p corresponde a la posición z de p.

Las operaciones que normalmente se llevan a cabo sobre esta estructura de datos reciben el nombre de *push* para introducir un elemento en la posición z, y *pop* para extraer un elemento de la posición z en la pila p.

Cuando la pila se encuentra vacía, una extracción debería provocar una excepción, o un valor inválido. Aunque la pila no posee límites teóricos de capacidad máxima, sí existen límites físicos, y estos dependen en particular de la implementación escogida.

5.2 Implementaciones de la pila

Podemos encontrar pilas implementadas mediante arreglos o listas. En las pilas implementadas con arreglos, los elementos se encuentran físicamente juntos y en las operaciones de inserción y borrado son referenciados por su posición; en las implementadas mediante listas, los elementos no están necesariamente juntos y las operaciones de inserción y borrado utilizan punteros a memoria para accesarlos.

Cuando se escoge la implementación por arreglos, estos pueden ser de un tamaño fijo (lo normal), o estar formados dinámicamente, teniendo la capacidad de crecer cuando se agota su capacidad inicial.

Con las listas, el problema de inicialización con un tamaño fijo no existe, pero se ocupa espacio en punteros por cada nodo creado, junto con la complejidad natural del procesamiento de listas.

En este artículo, y dado que el objetivo del mismo no es describir algoritmos para manejo de pilas, se ha escogido una implementación con un arreglo de tamaño fijo (aunque especificado por el usuario al momento de crearlo), lo cual permitirá centrarse en los problemas de genericidad que interesan.

Cuando no hay elementos en la pila, un pop retornará un elemento inválido, y cuando la pila se encuentre llena, el push simplemente no insertará más elementos en la misma.

Las pilas genéricas poseen la cualidad de poder contener más de un tipo de elemento sin necesidad de reimplementarlas, lo cual no ocurre normalmente cuando se utilizan estructuras de datos abstractas (ADT's), o implementaciones particulares para fines específicos sin orientación a objetos.

5.3 Implementación utilizando herencia

La Figura 1 define una clase `stack` que permite insertar instancias de clases derivadas de la clase base `stackItem` definida anteriormente. La línea de código

```
class stack : public stackItem
```

hace que la pila misma herede de `stackItem`, lo que la faculta para crear pilas de pilas.

Como se puede observar en la Figura 1, la clase `stackItem` es una clase virtual pura que obliga a implementar los métodos `clone` (para crear copias del propio objeto), y `write` (que permite enviar el contenido del objeto a un stream), a la vez que implementa el operador amigo << para utilizar en forma genérica el `iostream`. La clase define, además de los contructores y destructores para instancias de la clase, las operaciones `clone()`, `write()`, `emptyItem()`, `push()`, `pop()`, y `size()`, y redefine los operadores = y <<.

```
#include <iostream.h>
// Clase base para los itemes a insertar en
// la pila
class stackItem;
typedef stackItem* pStackItem;

class stackItem {
  friend class stack;
protected:
  virtual pStackItem clone() = 0;
  virtual ostream& write(ostream& os) = 0;
  friend ostream& operator<<
                    (ostream&, stackItem&);
};

ostream& operator<<(ostream& os, stackItem& s)
   {return s.write(os);}

// Definición de una pila generica
// usando herencia
class stack: public stackItem {
  class emptyStack : public stackItem {
  protected:
    pStackItem clone() { return this; }
    ostream& write(ostream& os) {
    return os << "Empty stack"; }
  };
  static emptyStack empty;
// Estructura de datos para la pila
  pStackItem* base;
  int top;
  int sz;

// metodos de la clase
protected:
  virtual pStackItem clone()
       {return new stack(*this);}

public:
  pStackItem emptyItem() const
     { return &empty; }

  stack(int s = 20) {
    sz = s; top = 0;
    base = new pStackItem[sz];
  }
  stack(stack& anStack);
  ~stack();

  void push(pStackItem a) {
    if (size()<sz) base[top++] = a->clone();
```

```
  }
  pStackItem pop() {
    return((top!=0)?base[--top]:emptyItem());
  }
  int size() const { return top; }
  void operator=(stack& anStack);
  virtual ostream& write(ostream& os);
}; // class stack
```

Figura 1. Definición dela clase `stack` utilizando herencia.

La Figura 2 muestra una posible implementación de la clase `stack` definida en la Figura 1.

```
stack::emptyStack stack::empty;
stack::stack(stack& anStack) {
  sz = anStack.sz;
  base = new pStackItem[anStack.sz];
  top = 0;
  for (int i = 0; i < anStack.size(); i++)
    push(anStack.base[i]);
} // stack

stack::~stack() {
  for (int i = 0; i < top; i++)
    delete base[i];
  delete [] base;
} // ~stack

void stack::operator=(stack& anStack) {
  int x = size();
  for (int i = 0; i < x; i++)
    delete pop();
  for (i = 0; i<anStack.size() && i < sz;i++)
    push(anStack.base[i]);
} // operator=

ostream& stack::write(ostream& os) {
  int x = size();
  os << endl << "(Tamano: " << x << endl << "
       Elementos: ";
  for (int i = 0; i < x; i++)
    os << *(base[i]) << ' ';
  os << ')' << endl;
  return os;
} // write
```

Figura 2. Implementación de la clase `stack` utilizando herencia.

El `write` virtual puro evita tener que crear un operador amigo << para cada clase descendiente de `stackItem`, pues su implementación es:

```
ostream& operator<<(ostream& os,stackItem& s)
   { return s.write(os); }
```

La clase `stack` posee una clase anidada `emptyStack`, que permite retornar un puntero a una instancia identificable como inválida (por convención) cuando se ha agotado el contenido de la pila y se efectúan `pop` sobre la misma. Posee un constructor que crea clases de tamaños específicos (20 por defecto), un destructor no virtual que destruye los elementos contenidos, las operaciones `push` y `pop` que trabajan con punteros a instancias de clases que hereden de `stackItem`, y las operaciones misceláneas: `size`, para obtener la cantidad de elementos en la pila, el operador de asignación =, y los métodos virtuales `clone` y `write` mencionados anteriormente.

Su funcionalidad básica es la de una pila creada con un arreglo, y manejada con índices y contadores. Seguidamente se mencionarán algunas de sus propiedades.

En primer lugar, esta pila es polimórfica, aceptando cualquier instancia de las subclases derivadas de `stackItem`, aunque limita el contenido solamente a estos objetos; es decir, nunca podrá verse una pila de enteros o de caracteres, al menos que se definan clases que hereden de `stackItem`, y que contengan enteros y caracteres respectivamente.

La pila crea copias de los elementos que se insertan en la misma, para lo cual existe el método virtual puro `clone`. De esta forma, cada instancia que se inserte en la pila podrá crear una copia de sí misma, la cual será almacenada en la pila, y el elemento que se utiliza para obtener la copia puede ser destruído sin alterar el contenido de la pila.

Como cada elemento conoce su propio comportamiento, el polimorfismo puede llevarse al extremo, al permitir que se inserten instancias de clases súmamente complejas, como árboles, grafos o pilas, pues estos aportarán un método `clone` adecuado; sin embargo, si el método `clone` no funciona correctamente, el desempeño de toda la pila puede llegar a verse afectado, siendo muy sensible a esta delegación de responsabilidades.

Esta pila se basa exclusivamente en enlace dinámico (*dynamic binding*), que es la fuente de su flexibilidad, y puede contener instancias de clases distintas (heredadas de `stackItem`) simultáneamente, es decir, puede se utilizada para implementar una pila heterogénea.

5.4 Implementación utilizando *templates*

Otra alternativa es implementar la pila utilizando plantillas o *templates*, tal y como se muestra en la Figura 3. Aquí, los métodos están completamente definidos para cualquier tipo de componente básico que posea la capacidad de asignación estática mediante el operador =. Se pretende que la pila sea capaz de contener incluso a otras pilas, por lo que es necesario sobrecargar el operador = dentro de la clase `stack`, junto con la definición de un constructor para crear copias estáticas de la pila.

```
#include <iostream.h>

template<class T>
class stack {
// estructurea de datos para la pila
  T* base;
  T* top;
  int sz;

public:
// constructores y destructores
  static T noT;
  stack(int s = 20) {
       sz = s;
       base = new T[sz];
       top = base;
  }
  stack(stack<T>& anStack) {
       sz = anStack.sz;
       base = new T[anStack.sz];
       top = base;
       for (int i = 0; i<anStack.size(); i++)
        push(anStack.base[i]);
  }
  ~stack() { delete [] base; }

// metodos de la clase pila
  void push(T a){if (size()<sz) *(top++)=a;}
  T pop(){return((top!=base)? *(--top):noT);}
  int size() const { return top - base; }

  void operator=(stack<T>& anStack) {
       int x = size();
       for (int i = 0; i < x; i++)
         pop();
       for(i=0;i<anStack.size() && i<sz; i++)
         push(anStack.base[i]);
  }

  friend ostream& operator<<(ostream&,
       stack<T>);
  friend istream& operator>>(istream&,
       stack<T>);
}; // class stack
```

Figura 3. Definición dela clase `stack` utilizando plantillas.

La Figura 4 muestra una implementación utilizando la plantilla definida en la Figura 3.

```
#define declareStack(T, typeName)   \
  typedef stack<T> typeName;        \
  T stack<T>::noT

template<class T>
ostream& operator<<(ostream& os, stack<T> a) {
  int x = a.size();
  os << endl << "(Tamano: " << x << endl
        << "  Elementos: ";
  for (int idx = 0; idx < x; idx++)
       os << a.pop() << ' ';
  os << ')' << endl;
  return os;
} // operator<<

template<class T>
istream& operator>>(istream& is,
  stack<T> a) {
  int size;
  T  e;
  is >> size;
  for (int idx = 0;
       idx < size; idx++) {
       is >> e;
       a.push(e);
  }
  return is;
} // operator>>
```

Figura 4. Implementación dela clase `stack` utilizando plantillas.

Esta pila no contiene referencias ni punteros a elementos, sino los elementos por sí mismos, lo cual obliga al usuario a utilizar instancias que puedan copiarse por sí solas. Muchos de los errores debidos a la falta de estos mecanismos de copia son determinados en tiempo de compilación.

La clase `stack` posee un operador amigo `<<`, que le permite utilizar en forma genérica el iostream:

```
template<class T>
ostream& operator<<( ostream& os, stack<T> a)
  { ... }
```

Como este operador es una función template o plantilla de función, automáticamente se crea una copia del mismo para el tipo específico T, cuando se instancia por primera vez.

Cuando se ejecutan `pop` con la pila vacía, `stack` retorna la variable de clase `noT`, la cual permite realizar comparaciones y determinar si la pila ha sido consumida.

Cuando se sobrepasa la capacidad de la pila, que se determina al momento de crearla, la operación `push` no hace nada, dejando la pila en el mismo estado en que se encontraba antes de efectuar la operación.

Como las clases template basan su poder en la capacidad de definir su contenido en forma estática, el **polimorfismo** permite definir pilas de tipos diferentes, sin tener que reimplementar código en forma manual. Aquí, el enlace estático de pilas instanciadas limita la posiblidad de tener elementos de varios tipos a la coerción que exista entre ellos, teniendo en cuenta que el tamaño de cada elemento de la pila es único (si se inserta un elemento cuyo tamaño sobrepase esta capacidad, el mismo será recortado al espacio disponible para contenerlo).

No se utilizan índices, sino las direcciones de los elementos contenidos en la pila (para poder observar la posibilidad de este tipo de manipulaciones con enlace estático).

6. Conclusiones

Luego de analizar las dos implementaciones de la clase stack expuestas con anterioridad, se pueden obtener algunas conclusiones que sirven como criterios para seleccionar posibles implementaciones de tipos genéricos en C++.

Las facilidad de uso de una u otra implementación depende en forma directa del diseño y programación de las clases, aunque generalmente es más sencillo definir contenedores usando clases template que herencia.

Es más natural implementar clases con templates porque no se requiere un nivel tan alto de abstracción como en la implementación con herencia. Al implementar una clase base, esta debe prever todas las particularidades de sus posibles clases derivadas. Sin embargo, cuando se ha completado la definición de una clase base, las clases derivadas sólo tienen que apegarse a los requerimientos de la misma, que cuando está bien definida provoca diseños muy naturales.

En cambio, las clases template simplemente utilizan los tipos no calificados que las parametrizan y dependen de que estos cumplan bien con los requerimientos del template, lo cual puede provocar un trabajo extra al definir los tipos.

Ambas implementaciones de contenedores son tan generales como para contener instancias de ellas mismas; sin embargo, se puede lograr un mayor grado de generalidad implementando clases template cuyos elementos contenidos sean elementos de alguna clase que posea clases derivadas.

Implementar contenedores heterogéneos es característico de la herencia, no de los templates; para lograr este tipo de polimorfismo en forma natural, es necesario definir una clase base cuyas clases derivadas (de diferentes tipos) definan los elementos almacenados en el contenedor.

Cuando se implementa un contenedor que almacena objetos de un mismo tipo (homogénero) en forma estática y que no están asociados a ninguna jerarquía de clases (sin herencia), parece más factible usar templates por cuanto producen una implementación más eficiente que utilizando herencia, ya que no es necesario recorrer estructuras dinámicas en tiempo de ejecución.

Por otro lado, si se desea implementar un contenedor que almacena objetos de diferentes tipos pero que heredan de una misma clase base (heterogéneo), lo mejor es usar herencia ya que su enlace dinámico permite referenciar los datos en forma completa sin aplicarles coerción hacia una clase base.

Es posible hacer algunas generalizaciones con respecto al espacio en memoria usado en las dos implementaciones descritas, que dependen del diseño de las clases y suponen un uso adecuado de los conceptos de herencia y parametrización por medio de templates

La longitud del código fuente escrito es menor en las clases template que en las clases desarrolladas mediante herencia, reduciéndose el esfuerzo posterior de mantenimiento del código.

A nivel de código ejecutable, la implementación con templates genera más código cuando se definen diferentes tipos de un contenedor, porque multiplica el código; sin embargo, el espacio en memoria que ocupan los elementos del contenedor es menor, al no requerirse punteros para referenciarlos. Por su parte, la herencia permite instanciar un solo contenedor de una clase base e incluir en ella elementos de cualquier clase derivada, lo cual posibilita una mayor reutilización del código, pero requiere un puntero adicional para manipular cada objeto en memoria.

7. Referencias

[1] Barton, J.J. y L.R. Nackman. Two faces of polymorphism, *C++ Report*. 7 (9):55-58, 65 Nov./Dic. 1995.

[2] Blair, G.S. et al. Genericity vs Inheritance vs Delegation vs Conformance vs..., *Journal of Object-Oriented Programming*, Set./Oct. 1989.

[3] Breymann, U. y N. Hughes. Composite templates and Inheritance -The perfect balance, *C++ Report*. 7 (7): 33-40, 76, Sep. 1995.

[4] Carroll, M. Tradeoffs of runtime parameterization, *C++ Report*. 7 (9): 20-27, Nov./Dic. 1995.

[5] Cardelli, L. y Wegner, P. On Understanding types, data-abstraction, and polymorphism. *Computing Surveys*, Vol 17, No. 4, págs. 471-522, 1985.

[6] Keffer, T. The dymanic range of C++. *C++ Report*. 7 (8): 40-43, Oct. 1995.

[7] Seidewitz, E. Genericity versus inheritance reconsidered: self-reference using generics, Proceedings of OOPSLA' 94 Conference. p. 153-163.

[8] Stroustrup, B. The C++ programming language, segunda ed. Addison-Wesley, Reading, MA, 1991.

[9] Stroustrup, B. The design and evolution of C++, Addison-Wesley, 1994.

[10] Vilot, M.J. Design generalization in the C++ standard library, C++ Report. 7 (8): 75-78, Oct. 1995.