Cómo enseñar diseño de estructuras de datos: un enfoque incremental

Ernesto Cuadros-Vargas* Roseli A. Francelin Romero* Markus Mock§ Instituto de Ci^encias Matem´aticas e de Computa¸c˜ao Univ. de S˜ao Paulo en S˜ao Carlos-Brasil §Depto. de Ciencias de la Computación de la Universidad de Pittsburgh ecuadros@spc.org.pe, rafrance@icmc.usp.br, mock@cs.pitt.edu

Abstracto

Si bien hay muchas formas de enseñar el diseño de estructuras de datos, es importante distinguir diseños de estructuras de datos alternativos en términos de mantenibilidad, flexibilidad, escalabilidad y grado de abstracción de datos. Este artículo presenta un enfoque incremental basado en exponer a los estudiantes a diferentes alternativas y mostrar sus ventajas e inconvenientes. Creemos que este novedoso enfoque incremental que se basa específicamente en la comparación de diseños alternativos es particularmente instructivo debido a su atención a la expresividad, es decir, los métodos y lenguajes de implementación y la eficiencia de la implementación, es decir, un buen diseño de algoritmos.

1 Introducción

Las técnicas y metodologías de programación han experimentado importantes avances desde la aparición de los primeros lenguajes de programación de alto nivel Fortran y Cobol. Por ejemplo, en los primeros días el uso de la declaración "GOTO" era común entre los programadores. Primero, debido a la falta de estructuras de control de nivel superior y luego también debido a la familiaridad de los programadores con la programación a nivel de ensamblador. A pesar de los muchos avances en la metodología y los lenguajes de programación, todavía hay mucho código escrito de maneras que dificultan su mantenimiento y evolución. Además, muchos libros de texto de ciencias de la computación que enseñan estructura de datos y diseño de algoritmos presentan ejemplos de código que ignoran estos principios en aras de la simplicidad, lo que puede proporcionar a los estudiantes malos ejemplos para sus futuros proyectos de software. No es raro encontrar incluso estudiantes de informática que escriben programas que emplean enfoques indeseables, incluidos algoritmos demasiado restrictivos y un uso intensivo de variables globales, por nombrar sólo dos ejemplos.

Durante la evolución de las técnicas y lenguajes de programación, se han desarrollado muchos principios de diseño diferentes (por ejemplo, programación estructurada y programación orientada a objetos). Algunos de ellos han cambiado profundamente nuestras nociones de programación. Hoy, enseñamos a los estudiantes diseño de estructuras de datos para brindarles los componentes básicos fundamentales para la construcción de software. Además, les enseñamos algoritmos eficientes para la solución de problemas comunes. En este artículo, presentamos un enfoque incremental para la enseñanza de estructuras de datos que se centra en su implementación de acuerdo con diferentes metodologías de programa. Para cada alternativa,

Este trabajo contó con el apoyo parcial de la FAPESP-Brasil mediante la Beca N° 99/11835-7

Examinamos las ventajas y desventajas de enseñar a los estudiantes principios de diseño importantes para sus propios proyectos.

Durante la evolución de los lenguajes de programación, han surgido algunos principios nuevos e importantes, como los presentados por Knuth [5], que han cambiado profundamente nuestros conceptos sobre la programación.
Las estructuras de datos se utilizan para organizar los datos con el fin de recuperar la información lo más rápido posible. Además de la idea representada por una estructura de datos, también son importantes los algoritmos utilizados para su gestión (e implementación). Este artículo se centra en los algoritmos utilizados para implementar estructuras de datos. Las diferentes técnicas que aquí explicamos son el resultado de aplicar diferentes paradigmas de programación y recursos del lenguaje de programación. El objetivo final de este enfoque es hacer que los estudiantes seleccionen soluciones eficientes y flexibles que hagan que el software sea más fácil de mantener y evolucionar.

Es importante resaltar que no mostraremos nuevas técnicas o metodologías de programación. Sin embargo, al contrastar diferentes enfoques según su mantenibilidad, reutilización, escalabilidad y rendimiento, nos esforzamos por impartir a los estudiantes la importancia de un cuidadoso equilibrio entre la simplicidad en el diseño (por ejemplo, el uso de variables globales) y principios importantes de ingeniería de software. El segundo objetivo de nuestro artículo es presentar, desde un punto de vista didáctico, la importancia de enseñar todas estas técnicas a programadores y comparar sus ventajas e inconvenientes cuando se aplican a diferentes problemas. Si enseñamos todas estas técnicas, los estudiantes también entenderán las consecuencias de utilizar cada una de ellas en sistemas reales.

Para ilustrar diferentes alternativas, utilizamos un ejemplo en ejecución, una estructura de datos vectoriales. Sin embargo, los principios que examinamos son aplicables a cualquier estructura de datos. Tratamos la estructura de datos vectoriales como una caja negra, proporcionando algunas funciones básicas, como funciones de acceso (por ejemplo, Insert()) o métodos para cambiar el tamaño (por ejemplo, Resize()). Los detalles de implementación, que están ocultos para el usuario final de este componente, serán el foco de este artículo. También mostraremos código que involucra otras estructuras de datos, como listas vinculadas o árboles binarios, para mostrar más problemas de implementación. Usamos ANSI C++ [7] porque es un lenguaje de programación de propósito general, con un sesgo hacia la programación de sistemas que admite computación eficiente de bajo nivel, abstracción de datos, programación orientada a objetos y programación genérica y también se usa comúnmente en la clase. habitación.

El documento está organizado de la siguiente manera. En la sección 2, clasificamos los enfoques de diseño según varios criterios, como dinámica, protección de datos, abstracción de datos y encapsulación de datos.
En la sección 3, presentamos diferentes técnicas para implementar estructuras de datos. En la sección 4, presentamos una breve discusión para mostrar cómo la calidad de los algoritmos utilizados puede afectar directamente el producto final. En la sección 5, analizamos la importancia de conocer las características del compilador para mejorar el producto final. Finalmente, el resumen y las conclusiones se presentan en la sección 6.

2 Descripción general

Las técnicas de diseño de estructuras de datos existentes se pueden clasificar según varios criterios, como dinámica, protección de datos, encapsulación, abstracción de datos, entre otros. Usando la dinámica como criterio, podemos llegar a la clasificación que se muestra en la Figura 1. Usando la dinámica como criterio de clasificación, se distinguen claramente los diseños de la estructura de datos vectoriales que usan asignación de memoria estática, un diseño inflexible y un diseño flexible que usa memoria dinámica. asignación.

La protección de datos es otro criterio de clasificación que cobró importancia con la llegada de la programación orientada a objetos. La Figura 2 distingue las alternativas de diseño versus. protección de datos, por ejemplo, utilizando variables globales (Secciones 3.1, 3.2), variables en espacios de nombres (Sección 3.7) y aquellos miembros de clase que fueron declarados como públicos (Sección 3.3). También podemos distinguir variables con protección parcial como miembros protegidos (en clases). La última posibilidad es totalmente

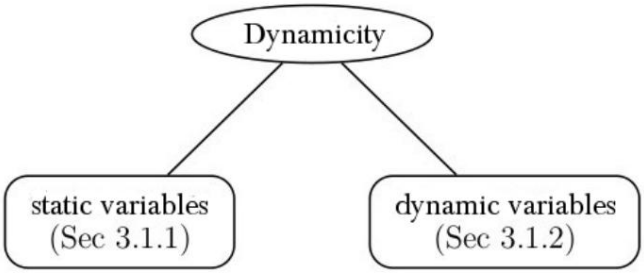

Figura 1: Clasificación según la Dinámica

miembros protegidos que son más conocidos como miembros privados.

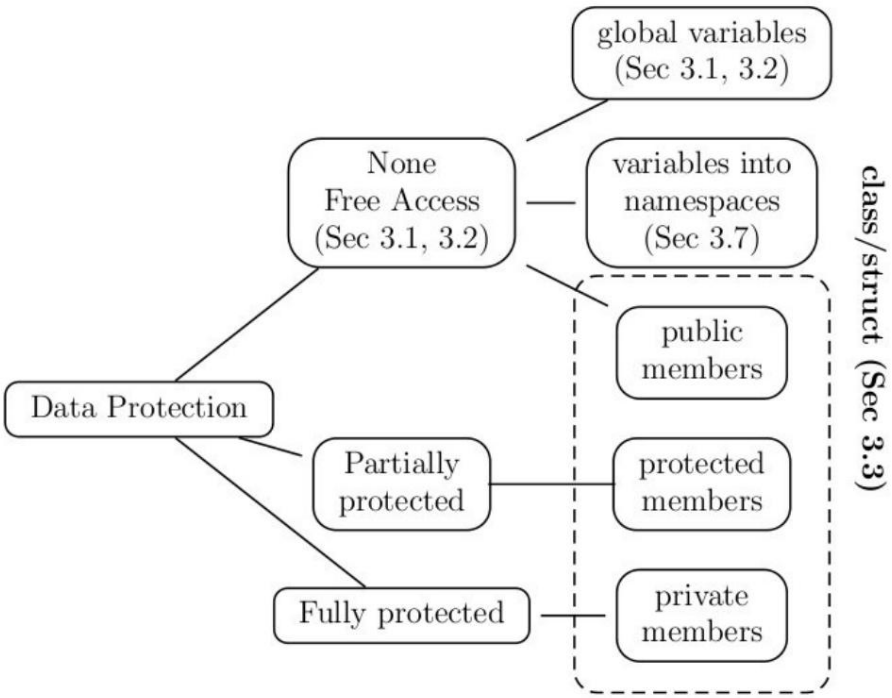

Figura 2: Clasificación según la protección de datos

Otro criterio importante para la clasificación de las técnicas existentes es la encapsulación. En la Figura 3 presentamos una posible clasificación.

Las variables y funciones globales representan aquellos casos en los que no se utiliza la encapsulación de datos. El segundo caso está representado por clases. Una clase puede contener miembros, métodos (Sección 3.3) e incluso clases y estructuras anidadas (Sección 3.6).

Cuando una clase se vuelve más compleja, son necesarias vistas parciales de ella (Sección 3.9). Las interfaces proporcionan esta técnica para acceder parcialmente a una clase.

Desde un punto de vista superior, podemos ver los espacios de nombres como el nivel más alto de encapsulación (Sección 3.7). Un espacio de nombres puede contener variables globales (con acceso público), funciones globales, clases, estructuras e incluso espacios de nombres anidados.

También podemos clasificar esas técnicas según la abstracción de datos (figura 4). Esta cuestión es especialmente importante al enseñar tipos de datos abstractos.

El primer grupo está representado por técnicas inflexibles. En este grupo las estructuras de datos son

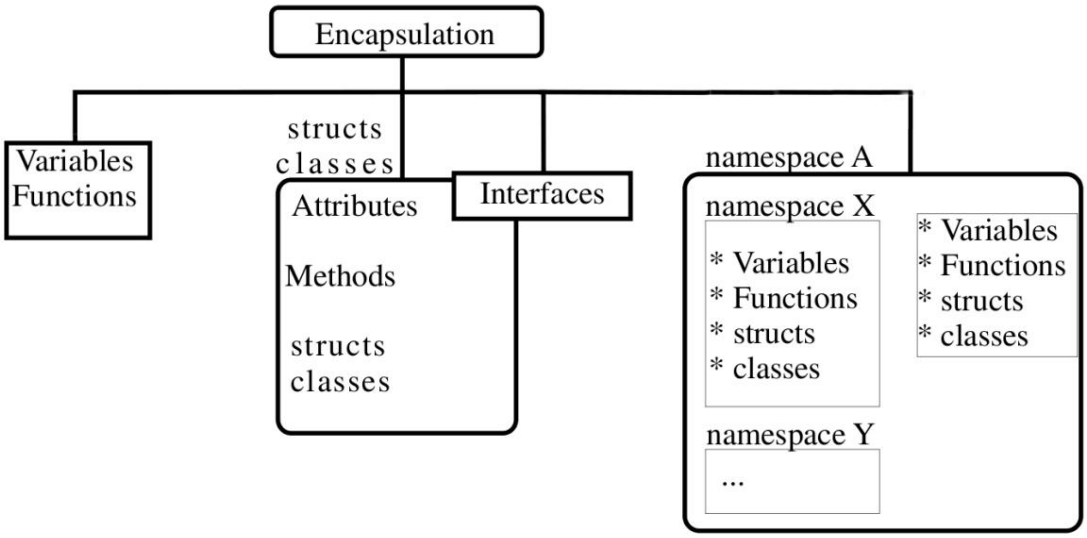

Figura 3: Clasificación según encapsulación

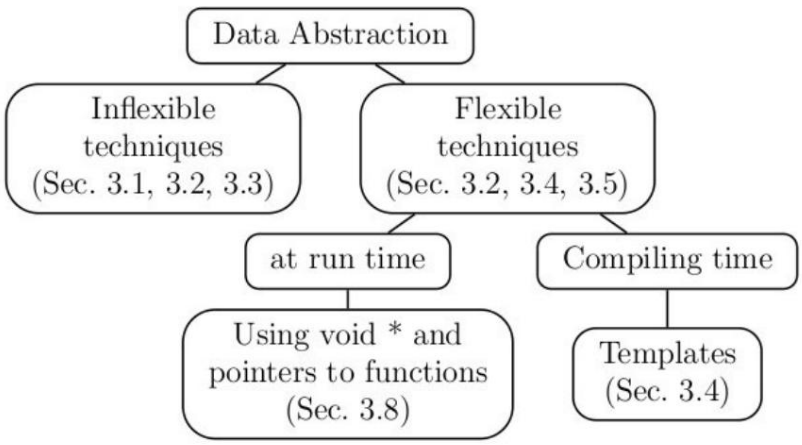

Figura 4: Clasificación según la abstracción de datos

diseñado para un tipo de datos específico (Secciones 3.1, 3.2, 3.3). También podemos diseñar estructuras de datos para tipos de datos abstractos usando plantillas (Secciones 3.4, 3.5). Las plantillas son útiles para reducir el mantenimiento del código.

Por otro lado, si el tipo de datos sólo se conoce en tiempo de ejecución, no podríamos utilizar plantillas, pero podemos utilizar diferentes técnicas como las presentadas en el apartado 3.8.

En los siguientes apartados explicamos cada una de estas técnicas en detalle.

3 alternativas de implementación

3.1 Vectores y variables de tamaño fijo

Si el objetivo es crear un componente que encapsule un vector, una de las soluciones más primitivas podría ser declarar un vector global de tamaño fijo que solo se muestra aquí como la solución más simple, en términos de líneas de código. La segunda alternativa, que probablemente todavía sea primitiva, es

NOTE:
The sequence used to present the following sections does not necessarily represent a sequence in terms of performance. For example, section 3.1.2 discusses the use of dynamic vectors and is presented before templates (section 3.4) but it **does not imply** that all the templates are dynamics. In fact, we can use templates with static vectors. In this case the final performance correspond to the one presented in section 3.1.1 (Fixed size vectors) instead of the one claimed for templates.

modificar este primero y utilizar vectores dinámicos pero aún variables globales. Presentamos estas dos técnicas en las secciones 3.1.1 y 3.1.2 respectivamente.

3.1.1 Vectores de tamaño fijo

En esta sección usaremos un vector global de tamaño fijo (gVect), para almacenar los elementos insertados de tipo entero y, un contador global (gnCount), para controlar el número de elementos utilizados en el vector. Estas variables se declaran de acuerdo con el siguiente código.

```
int gVect[100]; // Buffer para guardar los elementos int gnCount;
                                    // Contador para saber el número de elementos utilizados
```

Figura 5: Variables globales

Como el usuario del vector necesita algunas funciones para interactuar con estas variables internas, implementar una primera versión para el Insert

```
Insertar vacío (int elem) {

    if( gnCount < 100 )                                    //solo podemos insertar si hay espacio
        gVect[gnCount++] = elem; // Insertamos el elemento al final
}
```

Figura 6: Función de inserción según el primer paradigma

La única ventaja probable de este enfoque es que no necesitamos más tiempo para abordar con gestión de memoria (alloc, malloc, gratis, nuevo, eliminar).

Evidentemente, esta técnica tiene muchos inconvenientes pero destacamos los siguientes:

- El inconveniente más grave a solucionar en esta primera aproximación es, sin duda, su inflexibilidad. Si el usuario/sistema necesita menos de 100 elementos, no hay forma de ahorrar el espacio restante. Por otro lado, si nuestro usuario/sistema necesita insertar más de 100 elementos, este enfoque ya no es útil.

- No es posible utilizar este código para más de un vector en el mismo sistema. No es flexible para más de un tipo de datos. Sin embargo, para solucionar este último problema podríamos duplicar el mismo código y cambiar el tipo. Probablemente no sea una buena opción para fines prácticos.
  propósitos.

3.1.2 Vectores dinámicos y variables globales

Como vimos en la Subsección 3.1.1, se necesitan estructuras más flexibles. Intentaremos resolver este problema en esta sección, pero es necesario tener algunos conocimientos previos sobre punteros y memoria asignada dinámicamente (ver [7] para una buena referencia).

Para superar su inflexibilidad, comenzaremos con un puntero y aumentaremos el tamaño del vector gradualmente según los requisitos. En la Figura 7 podemos ver las variables iniciales para este caso.

```
int *gpVect = NULL; // Búfer dinámico para guardar los elementos int gnCount = 0; gnMáx = 0;
                                    // Contador para saber el número de elementos usados int
                                    // Variable que contiene el tamaño de la // memoria asignada
```

Figura 7: Variables globales para vector dinámico

Usamos gnCount para controlar cuántos elementos estamos usando considerando que ha asignado memoria para las posiciones de gnMax. La brecha entre gnCount y gnMax representa el espacio disponible para nuevos elementos posibles. Otro problema que está fuera del alcance de este artículo es determinar el delta utilizado para aumentar el vector cuando ocurre un desbordamiento. Sólo por motivos prácticos usaremos delta = 10.

El código para Insertar ahora es un poco diferente (Figura 8).

```
Insertar vacío (int elem) {

    si( gnCount == gnMax )          // No hay espacio en este momento para elem
        Cambiar tamaño();           // Cambiar el tamaño del vector antes de insertar elem
    gpVect[gnCount++] = elem; // Inserta el elemento al final de la secuencia }
```

Figura 8: Función de inserción usando variables globales pero vector dinámico

La función de cambio de tamaño podría definirse según la Figura 9.

```
cambio de tamaño vacío()
{
        constante int delta = 10;                                   // Se utiliza para aumentar el tamaño del vector.
        gpVect = realloc(gpVect, sizeof(int) * (gnMax + delta));
        // También puedes usar el siguiente código: // int *pTemp,
        i; // pTemp = nuevo
        int[gnMax + delta]; // Asignar un nuevo vector // for(i = 0; i < gnMax; i++)
                                                                    // Transferir los elementos //
        pTemp[i] = gpVect[i]; // también podemos usar la función memcpy // eliminar [ ] gpVect; // eliminar el vector
        antiguo // gpVect = pTemp;
                                                                    //Actualiza el puntero

        gnMax += delta;                                             // El Max debe incrementarse en delta
}
```

Figura 9: Función de cambio de tamaño usando variables globales pero vector dinámico

Entre las ventajas de este enfoque, podemos señalar:

- Esta estructura es más flexible que la presentada en el apartado anterior.
- El espacio de memoria asignado siempre será igual o un poco mayor que el del usuario necesidades.

Entre los inconvenientes podemos destacar:

- Este enfoque todavía se basa en el uso de variables globales, por lo que solo un vector por programa es posible.
- En comparación con la solución anterior, otro inconveniente es el tiempo extra necesario para ocuparse de la gestión de la memoria (incluso cuando los beneficios superan claramente este problema). Este último inconveniente se ve compensado por la flexibilidad que introducen los punteros.

Como podemos ver, este enfoque no habría sido posible sin el uso de punteros (en C++).
Desde nuestro punto de vista, cuando el estudiante aprende a lidiar con los indicadores que debe comprender lo importantes que son especialmente para este tipo de problemas.

Resolveremos parte de estos inconvenientes utilizando la Programación Modular en la siguiente sección.

3.2 Programación modular

Una posible solución para permitir la coexistencia de dos o más vectores en el mismo sistema es enviar los parámetros involucrados (es decir, gpVect, gnCount y gnMax, en este caso) para la función Insert(). Los parámetros deben enviarse por referencia para permitir alguna modificación interna.
De acuerdo con este enfoque, podemos redefinir la función Insertar de la siguiente manera. (Figura 10)

```
void Insert(int *& rpVect, int& rnCount, int& rnMax, int elem) {

    si (rnCount == rnMax)                    // Verificar el desbordamiento
        Cambiar tamaño(rpVect, rnMax); // Cambiar el tamaño del vector antes de insertar elem rpVect[rnCount+
    +] = elem; // Inserta el elemento al final de la secuencia }
```

Figura 10: Preparación de la función Insertar para usar múltiples vectores en el mismo sistema

La función de cambio de tamaño podría definirse según la Figura 11.

```
cambio de tamaño vacío (int *& rpVect, int& rnMax) {

        constante int delta = 10; pgpVect          // Se utiliza para aumentar el tamaño del vector.
        = realloc(gpVect, sizeof(int) * (rnMax + delta));
                                                    // El Max debe incrementarse en delta rnMax += delta;
}
```

Figura 11: Preparación de la función Cambiar tamaño para usar múltiples vectores en el mismo sistema

Incluso considerando que Insertar y Cambiar tamaño funcionan con los mismos datos, sus parámetros no son necesariamente los mismos. La función Resize solo necesita lidiar con rpVect y rnMax para aumentar el tamaño del vector, mientras que Insert los necesita todos. En general, diferentes funciones pueden necesitar parámetros diferentes incluso cuando pertenecen al mismo grupo y trabajan con los mismos datos. Una solución útil para resolver el caso general donde necesitamos n parámetros es agrupar todos los parámetros posibles creando una estructura C (Ver Figura 12).

```
estructura vectorial
{
        int*m_pVect, // Puntero al búfer m_nCount, // Controla
                cuántos elementos se utilizan realmente m_nMax, // Controla cuántos se asignan como
                máximo m_nDelta; // Para controlar el crecimiento

};
```

Figura 12: Vector de estructura que encapsula todos los datos para esta estructura de datos

Usando una estructura como Vector para cada vector existente en nuestro sistema, solo necesitamos enviar el puntero de la estructura a la función. Esta técnica también nos permite acceder a mp Vect, m Count y m Max a través de un solo parámetro adicional (un único puntero a la estructura). En la Figura 13, vemos el nuevo código para las funciones Insertar y Cambiar tamaño.

Entre las ventajas de este enfoque, tenemos:

- Es posible tener más de un vector por programa, definiendo variables locales que serán pasadas a las funciones.

- Además de la flexibilidad, no necesitamos un número variable de parámetros adicionales. Es importante porque reduce el mantenimiento.

```
void Insertar(Vector *pThis, int elem) {

        si (pThis->m_nCount == pThis->m_nMax)                    // Verificar el desbordamiento
                Cambiar tamaño(pThis);          // Cambiar el tamaño del vector antes de insertar el elemento
        // Inserta el elemento al final de la secuencia pThis->m_pVect[pThis-
        >m_nCount++] = elem;
}

cambio de tamaño vacío (Vector *pThis)
{
        pThis->m_pVect = realloc(gpVect, tamaño de(int) *
                                                (pThis->m_nMax + pThis->m_nDelta));
        // El máximo debe incrementarse en delta pThis->m_nMax
        += pThis->m_nDelta;
}
```

Figura 13: Función Insertar y Cambiar tamaño con la información de la estructura de datos encapsulada mediante una estructura

Entre los inconvenientes podemos destacar:

• No existe protección de datos. Todos los miembros de una estructura C son públicos de forma predeterminada.

3.3 Programación orientada a objetos

Si prestamos atención al código presentado en el apartado anterior, veremos que se acerca bastante a la Programación Orientada a Objetos [7]. Si analizamos detenidamente, hemos encapsulado nuestros datos m pVect, m Count, m Max y m Delta en la estructura Vector. Cuando enviamos esta estructura como primer parámetro para todas las funciones que trabajan alrededor de nuestra estructura de datos (vector en este caso), estamos frente al mismo principio utilizado por las clases de C++. En las Figuras 14 y 15 podemos ver el mismo código, pero usando Programación Orientada a Objetos.

La función Insertar de la Figura 13 ahora está implementada como método de clase vectorial (consulte la Figura 15).

Entre las ventajas de este enfoque, cabe señalar que:

• Ahora tenemos protección de datos. Sólo las funciones autorizadas pueden acceder a nuestros datos.

Entre los inconvenientes, podemos señalar:

• Si necesitamos, en el mismo sistema, dos vectores que usen tipos diferentes (es decir, int, double, etc.), debemos codificar dos clases diferentes con el mismo código (excepto las declaraciones de tipo) en todas las funciones.

```
// Definición de clase CVector
clase CVvector
{
        privado: int
        *m_pVect, // Puntero al búfer
                m_nCount, // Controla cuántos elementos se utilizan realmente m_nMax, m_nDelta; //
                Para                    // Controla cuántos se asignan como máximo
                controlar el vacío creciente Init(int delta); // Inicia
                nuestras variables privadas, etc. void Resize();
                                                        // Cambia el tamaño del vector cuando ocurre un desbordamiento


        público:
                CVector(int delta = 10); // Constructor void Insert(int elem);
                                                                // Insertar un nuevo elemento


                // Más métodos van aquí
};
```

Figura 14: Diseño de un vector usando programación orientada a objetos

```
void CVector::Insert(int elem) {

        si (m_nCount == m_nMax)                 // Verificar el desbordamiento
                Cambiar tamaño();               // Cambiar el tamaño del vector antes de insertar elem
        m_pVect[m_nCount++] = elem; // Insertamos el elemento al final
}
```

Figura 15: Insertar función usando programación orientada a objetos

3.4 Tipos de datos abstractos

Como hemos visto al principio de este artículo, el requisito era: “el usuario necesita alguna caja negra que encapsule la estructura de datos vectoriales”. Pero no se especificó el tipo de elementos a insertar. Es decir, el cuadro negro debería ser útil independientemente del tipo de datos. Un primer intento de resolver este problema se encuentra frecuentemente en muchos sistemas en los que se crea una definición de tipo global, como se muestra en la Figura 16.

Esta técnica sólo permite reducir el coste de mantenimiento del código si necesitamos algún otro tipo como float, double, etc. Sin embargo, este código todavía no es útil para un usuario que necesita dos (o más) vectores con diferentes tipos (es decir, char, flotante, corto, etc) al mismo tiempo en el mismo sistema. Afortunadamente, algunos lenguajes como C++ [7] tienen plantillas que permiten parametrizar el tipo de elemento. En la Figura 17 podemos ver una breve implementación del concepto de vector considerando un tipo de datos abstracto.

El uso de plantillas es importante desde el punto de vista didáctico porque el estudiante realmente puede ver, en términos de un lenguaje de programación, el concepto de Tipos de Datos Abstractos. Si los estudiantes aprenden la filosofía detrás de las plantillas, se centrarán principalmente en el concepto de estructura de datos y no en implementaciones de tipos específicos. Las plantillas de C++ son probablemente la construcción de programación más cercana para implementar el tipo de datos abstracto.

```
typedef int Tipo;
clase CVvector
{
        privado:
        Escriba*m_pVect;                        // Puntero al buffer
                ...

        público:
        Insertar vacío (Escriba elem);          // Insertar un nuevo elemento
                ...
};
```

Figura 16: Primero intente crear una clase de vector genérica

```
plantilla <nombretipo Tipo> clase CVector
{
        privado:
        Escriba*m_pVect;                // Puntero al buffer
        int m_nCount,                   // Controla cuántos elementos se utilizan realmente
                m_nMax,                 // Controlar el número de elementos asignados
                m_nDelta;               // Para controlar el crecimiento
        inicio vacío (int delta); // Inicia nuestras variables privadas, etc.
        cambio de tamaño vacío();       // Cambia el tamaño del vector cuando ocurre un desbordamiento

        público:
        CVector(int delta = 10); Insertar       // Constructor
        vacío (Escriba elem); //...           // Insertar un nuevo elemento

};
```

Figura 17: Clase de vector genérico usando plantillas

```
// Plantilla de implementación de clase CVector
<tiponombre Tipo> CVector<Tipo>::CVector(int delta) {

        Inicial(delta);
}

plantilla <tiponombreTipo> void CVector<Tipo>::Insert(Tipo &elem) {

        si (m_nCount == m_nMax)                 // Verificar el desbordamiento
                Cambiar tamaño();               // Cambiar el tamaño del vector antes de insertar elem
        m_pVect[m_nCount++] = elem; // Insertamos el elemento al final
}
```

Figura 18: Implementación de métodos para la clase de plantilla CVector

Entre las ventajas de este enfoque, tenemos:

- Esta implementación del concepto de vector es útil para tantos tipos de datos como sea necesario.
- El mantenimiento en torno al código correspondiente al concepto de “vector” está bien ubicado. Cualquier modificación de la implementación de esta estructura de datos será en una sola clase. Solo necesitamos modificar el código una vez.
- El uso de plantillas no afecta el rendimiento porque el código específico del tipo se genera cuando el programa se compila de acuerdo con los tipos de datos requeridos por el usuario.

Entre los inconvenientes podemos destacar:

- El parámetro abstracto para una plantilla debe definirse antes de compilar el programa. No sería útil si necesitáramos definir el tipo en tiempo de ejecución.

3.5 Patrones de diseño

Un patrón es una solución recurrente a un problema estándar. Cuando se entrelazan patrones relacionados, forman un "lenguaje" que proporciona un proceso para la resolución ordenada de problemas de desarrollo de software. Los lenguajes de patrones no son lenguajes formales, sino más bien una colección de patrones interrelacionados, aunque proporcionan un vocabulario para hablar sobre un problema particular. Tanto los patrones como los lenguajes de patrones ayudan a los desarrolladores a comunicar conocimientos arquitectónicos, ayudan a las personas a aprender un nuevo paradigma de diseño o estilo arquitectónico y ayudan a los nuevos desarrolladores a ignorar trampas y escollos que tradicionalmente se han aprendido sólo mediante una experiencia costosa1 .

Los programadores expertos no piensan en los programas en términos de elementos de un lenguaje de programación de bajo nivel, sino en abstracciones de orden superior. Esto se puede ver en las diferentes secciones de este artículo y el lector puede comparar las ventajas y desventajas de estos distintos niveles de abstracción.

Esta sección se centra en los patrones de diseño y específicamente en los iteradores. Puede encontrar más información sobre este tema en [2]

1Editorial invitado para Comunicaciones de la ACM, Número especial sobre patrones y lenguajes de patrones, vol. 39, núm. 10, octubre de 1996

Una estructura de datos robusta siempre debe proporcionar algún mecanismo para ejecutar algunas operaciones sobre los datos que contiene. Existen varias técnicas para lograr este objetivo. Uno de ellos fue introducido por la Biblioteca de plantillas estándar (STL) [6]. STL implementa varios contenedores para las estructuras de datos más comunes, como lista vinculada (lista de clases), vector (vector de clase), cola (cola de clases), que se implementan mediante plantillas. Todas estas clases también tienen un tipo interno definido por el usuario llamado iterador que permite iterar a través de la estructura de datos. Las instrucciones internas para recorrer árboles binarios son diferentes de las que se utilizan para las listas enlazadas. Sin embargo, una función para imprimir e iterar a través de estas dos estructuras de datos será exactamente la misma y se puede implementar como una plantilla. En la Figura 19 podemos ver la estructura de datos de lista y vectorial impresa por la misma función (Write()).

```
#incluir <vector> // sin .h
#incluir <lista>
#incluir <iostream>

usando el espacio de nombres estándar;

plantilla <tiponombreContenedor> void Write(Contenedor &ds, ostream &os) {

        Contenedor::iterador iter = ds.begin(); for( ; iter !=
        ds.end() ; iter++ ) os << *iter << "\n";

}

int principal(int argc, char* argv[]) {

        lista<flotante>         mi lista;
        vector<flotador>        mivector;

        para (int i = 0; i < 10; i++) {

                mi lista .push_back(yo);
                mivector.push_back(i+50);
        }

        Escribir(milista, cout);
        Escribir(mivector, cout); devolver
        0;
}
```

Figura 19: Uso de una función común tanto para una lista como para un vector

En la Figura 19 debemos prestar especial atención a la función Write(). Ahora explicaremos esa función con más detalles. Primero, la función Write() tiene que ser una plantilla para poder recibir cualquier clase posible como parámetro (lista y vector en este caso). En segundo lugar, todos los contenedores proporcionados por STL tienen métodos para recuperar la información al principio y al final.
Esa es la razón para utilizar los métodos comenzar() y finalizar().

Probablemente la cuestión más importante a destacar es la abstracción representada por el instructivo.

iteración ++ contenida en la declaración for. Esta operación representa el operador++ que ha sido sobrecargado por ambos iteradores. Obviamente, el código para pasar al siguiente elemento de la lista es diferente al de un vector, pero el operador++ proporciona un mayor nivel de abstracción. Finalmente, *iter nos da acceso al elemento donde se encuentra el iterador en ese momento.

Otra técnica bien conocida utilizada para aplicar operaciones comunes a todos los elementos de un contenedor son las funciones de devolución de llamada. En esta técnica, el usuario proporciona un puntero a una función que el contenedor llamará para cada elemento (la función de devolución de llamada). Para aplicar la operación implementada por la función callback a los elementos del contenedor, la estructura de datos solo necesita implementar un método para recorrer los elementos que llaman a esta función para cada uno de ellos.

Una técnica más sofisticada para superar este problema se conoce como objetos de función.
En la Figura 20 podemos ver un ejemplo donde aumentamos en uno todos los elementos del contenedor.
Esta técnica es especialmente útil para aquellos a quienes no les gusta utilizar punteros a funciones. En términos de seguridad, los objetos de función son probablemente más seguros que los punteros de función.

Entre las ventajas de este enfoque, podemos destacar especialmente el mayor nivel de abtracción.

El único inconveniente probable para un nuevo programador STL es comprender los mensajes de error y advertencias extremadamente largos generados por las plantillas anidadas. Este problema se puede superar rápidamente leyendo la bibliografía presentada en [6].

3.6 Preparando el código para futuros proyectos de grupos de trabajo

Un punto importante a discutir aquí es preparar al estudiante/programador (o usuario) para trabajar en grupos. Si cada estudiante/programador desarrolla una parte de un sistema más grande, es necesario decidir primero cómo se comunicarán los subsistemas entre sí cuando todo el sistema esté terminado. Como programadores, sabemos que diseñar software requiere mucho cuidado.
Sin embargo, no es fácil evitar algunos conflictos cuando unimos todos los subsistemas. En esta sección mostraremos un problema común y presentaremos una posible solución.

Para explicar el problema suponemos que existen sólo dos programadores que se encargan de programar una Lista Enlazada y un Árbol Binario, respectivamente. También suponemos que los programadores ya han leído la sección 3.4 sobre plantillas y quieren utilizar ese nivel de abstracción para resolver el problema.

El primer estudiante debería implementar la Lista Enlazada en una clase, probablemente llamada CLinkedList.
Esta estructura de datos también requiere una estructura interna para almacenar cada nodo que se llamará NODO. Esta estructura interna debe diseñarse para almacenar los datos del nodo y el puntero al siguiente nodo. Como vimos anteriormente, CLinkedList se diseñará como una clase de plantilla. Es decir, la estructura para los nodos internos (NODO) también debe diseñarse como plantilla.

Una posible solución de diseño se muestra en la Figura 21.

El código para definir el árbol binario estará representado por la clase CBinaryTree y podría ser similar excepto que la estructura NODE debería considerar los punteros m pLeft y m pRight en lugar de m pNext.

Si intentamos utilizar estos dos archivos, LinkedList.h y BinaryTree.h, en el mismo sistema, tendremos un nombre duplicado porque el nombre NODE existe en ambos pero tiene diferentes estructuras internas. Si cambiamos uno de ellos, probablemente lo cambiemos por otro identificador que podría no ser tan intuitivo como NODE. Si prestamos más atención a este problema, veremos que el problema aquí no es cambiar el nombre porque es apropiado para ambos.

Otro problema en estos códigos es que, en ambos casos, la estructura NODE fue diseñada para uso interno únicamente. Es decir, el usuario no necesita saber de su existencia. En otras palabras, deberían haberse definido dentro de las respectivas clases y no al mismo nivel que

```
clase CMyComplexDataStructure {

        vector<flotador> m_container; público:
        void
        Insert(float fVal) { m_container.push_back(fVal); } plantilla <typename objclass>
        void sumone(objclass funobj) {


            vector<float>::iterador iter = m_container.begin(); for (; iter !=
            m_container.end() ; iter++) funobj(*iter);

        }
};

plantilla <typename objclass> clase
funcobjclass {

        público:
        operador vacío ()(objclass& objinstance) {

            objeción++;
        }
};

int principal(int argc, char* argv[]) {

          CMyComplexDataStructure ds;
          ds.Insertar(3.5);
          ds.Insertar(4.5);
          ds.Insertar(6.5);
          ds.Insertar(3.9);

          funcobjclass<flotador> x;
          ds.sumone(x);

          devolver 0;
}
```

Figura 20: Implementación de un objeto de función

```
// Código para el estudiante implementando una Lista Enlazada // Archivo
LinkedList.h

// Estructura para nodos en una plantilla de lista vinculada
<typename T> struct NODE {

        T m_datos;                          // Los datos van aquí
        estructura NODO<T> *m_pNext; // Puntero al siguiente nodo static long id;
                                            // identificación del nodo

        NODO()                              // Constructor:
                m_data(0), m_pNext(NULL) {}

        // Más métodos ve aquí
};

NODO largo::id = 0; // Inicialización del id del nodo

plantilla <tiponombre T> clase CLinkedList {

        privado:
        NODO<T> *m_pRoot;                   // Puntero a la raíz

        público:
        // Más métodos van aquí
};
```

Figura 21: Primera definición de clase CLinkedList

él. Si el programador diseña el código siguiendo este consejo evitamos problemas futuros. El siguiente código muestra el mismo código con estas modificaciones.

Esta mejor técnica de diseño resuelve parcialmente el problema; sin embargo, los problemas realistas no involucran solo dos clases. Por ejemplo, en el caso de un software que utiliza una hoja de cálculo y un editor de texto de dos desarrolladores diferentes, es poco probable que no haya conflictos de nombres. Cómo evitarlos, lo discutiremos en la siguiente sección. De este enfoque, podemos destacar las siguientes ventajas:

- Podemos evitar conflictos de nombres locales cuando el software es desarrollado por un grupo.
- El alcance de la estructura interna del NODO está mejor definido ahora, evitando problemas futuros. lemas.

Entre los inconvenientes podemos destacar:

- No permite evitar algunos conflictos cuando el software es realizado por desarrolladores independientes o grandes grupos.

```
// Código para el estudiante implementando una Lista Enlazada
// Archivo ListaEnlazada.h
plantilla <tiponombre T> clase CLinkedList
{
        privado:                // Estructura para nodos en una lista enlazada
        estructura NODO
        {
                t                       m_datos; // Los datos van aquí
                estructura NODO * m_pNext; // Puntero al siguiente nodo
                // Algunos métodos van aquí
        };
        NODO* m_pRoot;                          // Puntero a la raíz
        público:            // Más métodos van aquí
};
```

Figura 22: Segunda definición de clase CLinkedList

```
// Código para el estudiante implementando un Árbol Binario
// Archivo BinaryTree.h

plantilla <nombre de tipo T>
clase CBinaryTree
{
        privado:                // Estructura para nodos en un árbol binario
        estructura NODO
        {
                t                       m_datos;                // Los datos van aquí
                struct NODE * m_pLeft, m_pRight;// Puntero al nodo izquierdo y derecho
                // Algunos métodos van aquí
        };
        NODO* m_pRoot;                                          // Puntero a la raíz
        público:            // Más métodos van aquí
};
```

Figura 23: Definición de clase CBinaryTree

3.7 Espacios de nombres

Pensemos en un software que necesita utilizar una hoja de trabajo y un editor de texto. Evidentemente, no intentaremos construirlos desde cero. Intentaremos, por ejemplo, utilizar Excel y Word para conseguir nuestro objetivo. La misma idea también podría aplicarse para cualquier Sistema Operativo como UNIX, Linux, etc. En el caso específico de MS Excel y MS Word ponen a disposición algunos métodos y DLL's para interactuar con ellos directamente por programa. Podemos extraer el código contenido de las DLL y convertirlo a clases y métodos de C++ usando la directiva #import según el código presentado en la Figura 24.

```
#import <mso9.dll> no_namespace rename("DocumentProperties", "DocumentPropertiesXL")

#importar <vbe6ext.olb> no_namespace #importar
<excel9.olb>

rename("DialogBox", "DialogBoxXL") \ rename("RGB",
                                        "RBGXL") \ rename("DocumentProperties",
                                        "DocumentPropertiesXL") \ no_dual_interfaces


#importar <msword9.olb> renombrar("DialogBox", "DialogBoxWord") \
                                        renombrar("RGB", "RBGWord") \
                                        renombrar("DocumentProperties", "DocumentPropertiesWord") \ no_dual_interfaces
```

Figura 24: Importación de código desde una DLL para poder utilizarlo

La directiva #import genera dos archivos con extensiones TLH (Encabezado de biblioteca de tipos) y TLI (Implementaciones de biblioteca de tipos) que se agregan automáticamente al proyecto mediante un #include oculto.

Como el lector puede imaginar, la cantidad de clases generadas a partir de un software como Excel es grande. Si aplicamos la directiva #import para extraer las clases y métodos de Word y Excel en el mismo sistema, podrás ver muchas clases con el mismo nombre en ambos. Para evitar este tipo de problemas, ANSI C++ se enriqueció con espacios de nombres.

La sintaxis para crear un espacio de nombres es bastante similar a la de clases y estructuras. La diferencia es que no podemos crear instancias usando espacios de nombres. Fueron diseñados exclusivamente para proporcionar un nivel de encapsulación nuevo y superior que puede contener variables, funciones, métodos, clases y estructuras globales. En la Figura 25 podemos ver el código de un posible espacio de nombres llamado MyNS.

Cuando usamos espacios de nombres estamos aumentando en uno el nivel de encapsulación. Es decir, para identificar cualquier miembro tenemos que agregar el nombre del espacio de nombres NS1 y el operador de resolución de alcance :: como acceder a un método. La Figura 26 ilustra cómo implementar los miembros para el espacio de nombres MyNS. Entre las ventajas de este enfoque, podemos destacar:

- Podemos evitar conflictos cuando el software es desarrollado por dos grupos diferentes o incluso por diferentes fabricantes.

No vemos inconvenientes considerables en el uso de plantillas.

```
// Archivo MyNS.h
espacio de nombres
MyNS {
        int gnContar; // contador global

        // Alguna pequeña función implementada en línea double
        Addition(double a, double b) // Función global {
                devolver a+b;           }

        // Algunos prototipos long
        factorial(int n); // función global

        prueba CT de clase

        { público:
                        Prueba();
                vacío   Método1();
        };
        // Más variables, funciones, métodos, clases y estructuras van aquí
};
```

Figura 25: Uso de espacios de nombres

3.8 Definición del tipo de datos en tiempo de ejecución

Como vimos en la sección 3.4, el usuario se ve obligado a definir el parámetro abstracto antes de usar una plantilla. Sin embargo, hay varios casos en los que el tipo sólo se conoce en tiempo de ejecución.

Probablemente el caso más obvio es un motor de base de datos donde el usuario define, en tiempo de ejecución, si un campo es entero, char, flotante, etc. pero no es necesario compilar el programa nuevamente. Toda esta flexibilidad debe incorporarse al programa intentando minimizar el posible impacto en el mantenimiento y el rendimiento.

Una posible solución, que se encuentra frecuentemente en los programas, es el uso de un interruptor para determinar el tipo de datos seleccionado por el usuario. En la Figura 27 podemos ver esta técnica.

Este código resuelve parcialmente el problema, pero crea otros nuevos. Entre los nuevos problemas podemos destacar los siguientes:

- Cada nuevo tipo posible requeriría una nueva caja en el interruptor, lo que básicamente contienen el mismo código, solo cambiando el tipo.
- Al utilizar este enfoque, casi nos vemos obligados a implementar la función Insertar elementos() como plantilla. Este problema se propaga a las funciones utilizadas internamente por In-sert Elements().
- Esta técnica tiene efectos sobre la interpretación porque cada vez que tenemos que utilizar el plantilla probablemente necesitaremos un interruptor similar.

```
//Archivo NS1.cpp
#include "MyNS.h" long MyNS::factorial(int n) {

        // Aquí va un código complejo
}

// Constructor para la clase CTest
MiNS::CTest::CTest() {

        // Inicializa algo
}

void MyNS::CTest::Método1() {

        // Tienes que escribir tu código aquí
}
// Más implementación para prototipos NS1.h aquí
```

Figura 26: Implementación de métodos contenidos en espacios de nombres

Ante este problema, es común minimizar el posible efecto de este pequeño e inadecuado si, pero si se repite varias veces la diferencia podría ser considerable. Si el lector quiere saber esa diferencia, pruebe el código que se muestra en la Figura 28 y luego repita el mismo experimento sin el if que significa, con el bucle vacío.

Si ejecutamos el código mostrado en la Figura 28 con el bucle vacío usando un ordenador convencional (Procesador Intel Pentium III de 1,2 GHz, ejecutando Windows XP, 512 Mb de memoria RAM, utilizando el compilador MS Visual C++ 6.02), y comparamos su resultado con el obtenido con el bucle vacío. Veremos una diferencia considerable (¡solo inténtalo!). Esto es suficiente para tener una idea del efecto cuando se agrega un if como este.

2No cambiamos los modificadores de optimización utilizados por este compilador.

```
cin >> opción;
cambiar(opción) {

        caso INT_TYPE_ENUM: {
                  CArray<int>a;
                Insertar_Elementos(a);
                romper;
        }
        caso DOUBLE_TYPE_ENUM: {
                  CArray<doble> a;
                Insertar_Elementos(a);
                romper;
        } // Más casos para otros tipos van aquí
}
```

Figura 27: Primero intente utilizar tipos definidos por el usuario en tiempo de ejecución

```
const unsigned long max = 500 * 1000 * 1000; const paso largo sin
firmar = 100 * 1000 * 1000; cuenta larga sin firmar;


        for(cuenta = 0L; cuenta < máx; cuenta++) if( cuenta % paso
                == 0L) printf("algo de texto...\n");
```

Figura 28: Comparación del rendimiento después de agregar un if único en un bucle

Pero la pregunta ahora es: ¿es posible diseñar una clase genérica capaz de funcionar incluso cuando el tipo de datos está definido en tiempo de ejecución? La respuesta es: sí. Imaginemos nuestra antigua clase CVector, pero la rediseñaremos considerando las restricciones discutidas en esta sección. La nueva clase CVector se muestra en la Figura 29.

```
clase CVvector
{
        privado:
        vacío **m_pVect; int          // Puntero al buffer
        m_nCount,                     // Controla cuántos elementos se utilizan realmente
                                      // Controla el número de elementos asignados m_nMax,
                m_nDelta,             // Para controlar el crecimiento
                m_nElemSize; // tamaño del elemento

        // Puntero a la función para comparar int
        (*m_lpfnCompare)(void *, void*); inicio vacío (int
        delta); // Inicia nuestras variables privadas, etc. void Resize();
                                              // Cambia el tamaño del vector cuando ocurre un desbordamiento

        público:

        CVector( int (lpfnCompare)(void *, void*), int nElemSize, int
                                      delta = 10); // Constructor void Insert(void * elem);
                                                          // Insertar un nuevo elemento

        // Más métodos van aquí
};
```

Figura 29: Diseñando un vector sin plantillas

La primera restricción si solo conocemos el tipo de datos en tiempo de ejecución es que no podemos usar plantillas ni definir nuestro vector dinámico usando un tipo específico. Esa es la razón para tener nuestro m ppVect como un vacío . Si no conocemos el tamaño de cada elemento necesitamos una variable para controlarlo (m nElemSize). Otro problema nuevo es cómo comparar dos elementos para insertarlos en la posición correcta. Como el usuario sabrá exactamente el tipo en tiempo de ejecución, también podría proporcionar una función genérica para comparar dos elementos (m lpfnCompare). Volveremos a explicar este último miembro después de analizar la Figura 30.

```
// Implementación de clase CVector
CVector::CVector(int (*lpfnCompare)(void *, void*), int nElemSize, int delta)


{       Inicial(delta);
        m_lpfnCompare = lpfnCompare;
        m_nElemSize = nElemSize;
}

void CVector::Insert(void *pElem) {

      si (m_nCount == m_nMax)                  // Verificar el desbordamiento
            Cambiar tamaño();                  // Cambiar el tamaño del vector antes de insertar elem
      m_pVect[m_nCount++] = DupBlock(pElem); // Insertamos el elemento al final
}

vacío* CVector::DupBlock(vacío *pElem) {

   vacío *p = nuevo carácter[m_nElemSize];
   devolver memcpy(p, pElem, m_nElemSize);
}

// Más implementaciones van aquí
```

Figura 30: Intentando definir tipos de usuarios en tiempo de ejecución

En la Figura 31 se puede ver una posible implementación de una función para comparar números enteros.

```
int fnIntCompare( vacío *pVar1, vacío *pVar2 ) {

        // < 0 si *(int *)pVar1 < *(int *)pVar2, // > 0 si *(int *)pVar1
        > *(int *)pVar2 // más 0


        retorno *(int *)pVar1 - *(int *)pVar2;
}
```

Figura 31: Función utilizada para comparar dos números enteros considerando dos direcciones genéricas

3.8.1 Pasando al mundo real

Un caso real lo presenta la Open Database Connectivity (ODBC) [3] que se utiliza frecuentemente para acceder a bases de datos heterogéneas. ODBC es básicamente un estándar que proporciona mecanismos para acceder a diferentes bases de datos (es decir, Oracle, Microsoft SQL Server, Sybase, etc.) de forma homogénea.

El punto principal a discutir aquí es cómo ODBC puede manejar varias bases de datos de diferentes fabricantes y formatos minimizando el efecto sobre el rendimiento. Esta pérdida de rendimiento puede descartarse y obtenemos algo a cambio.

Una de las mejores técnicas para lograr ese objetivo es utilizar un puntero a algunas funciones. Según esta técnica, cada fabricante (para cada nuevo controlador ODBC) debe proporcionar un conjunto de funciones comunes predeterminadas para ejecutar todas las operaciones posibles con una base de datos (es decir, insertar, eliminar y actualizar registros, ejecutar alguna instrucción SQL, etc.). Este conjunto de funciones comunes se registra en el sistema operativo y se reconoce como un nuevo controlador.

Sólo para simplificar el proceso suponemos en esta sección que las operaciones requeridas para todas las bases de datos son: Open(), InsertNewRegistry(), DeleteRegistry(), MoveNext(), MovePrev() y Close(). El POOL que contiene el puntero a funciones podría haberse definido en la Figura 32.

```
// odbc.h: archivo compartido para nuevos fabricantes de bases de datos

estructura PISCINA
{
        long (*lpfnOpen)(char *pszDBName); void
        (*lpfnInsertNewRecord)(DBID largo); void
        (*lpfnDeleteRecord)(DBID largo); vacío
        (*lpfnMoveNext)(DBID largo); vacío
        (*lpfnMovePrev)(DBID largo); vacío (*lpfnClose)
        (DBID largo);
        // Más operaciones van aquí
};
```

Figura 32: Conjunto de punteros a funciones

Si un nuevo fabricante de bases de datos, ABC, quiere agregar su controlador a ODBC, probablemente lo harán. preparar su propia piscina según la Figura 33.

```
PISCINA ABCPool = {&ABC_Open,                    &ABC_InserNewRegistry,
                   &ABC_DeleteRegistry, &ABC_MoveNext,
                   &ABC_MovePrev,                &ABC_Cerrar};

long ABC_Open(char *pszDBName) { // Algunos códigos complejos van aquí }
```

Figura 33: Preparación del conjunto de punteros a funciones

Cuando el usuario abre una fuente de base de datos, ODBC crea una nueva conexión con el POOL de funciones correspondientes a esa base de datos. Después de ese punto, el cliente podrá ejecutar todas las operaciones directamente utilizando esos punteros. Toda esta complejidad queda oculta por ODBC y es transparente para el usuario final. El único requisito es que todas las funciones Open tengan el mismo prototipo. La misma idea se aplica para el resto de funciones.

Frecuentemente, los punteros a funciones se confunden con “técnicas de programación complejas”, de hecho esa es la razón por la que existen muchos lenguajes que no utilizan punteros. Mostramos su potencialidad en el siguiente apartado.

3.8.2 Reducir el tamaño del código mediante punteros

Usar punteros a funciones podría ayudarnos a reducir considerablemente el código, pero si no se usan correctamente pueden crear más problemas de los que resuelven. Es importante comprender las ventajas y desventajas, en términos de rendimiento y especialmente de mantenimiento del código, entre los dos códigos que se muestran en las Figuras 34 y 35 respectivamente.

```
// Código A
float a, b, c; int optar;

// ingresa los operandos cin >>
a >> b;

// OPT-> 0-Suma, 1-Resta, 3-Multiplicación, 4-División cin >> opt;


cambiar (optar) {

        caso 0: c = Suma(a, b); caso 1: c =                    romper;
        Resta(a, b); caso 2: c = Multiplicación(a, b);         romper;
        caso 3: c = División(a, b); //más operaciones de      romper;
        casos van aquí                                         romper;

}
```

Figura 34: Llamar a funciones similares usando el interruptor

Como puede ver el lector, las cuatro funciones involucradas tienen el mismo prototipo. Todos ellos toman dos valores flotantes como parámetros y devuelven un valor flotante. Usando punteros a funciones podemos reducir el número de casos y, en consecuencia, eliminar el cambio. Por otro lado, codificar usando punteros.

```
// Código B
// Tipo de usuario para simplificar la declaración

typedef float (*lpfnOperation)(flotador, flotante);
// CVector del puntero a funciones lpfnOperation
vpf[4] = {&::Addition,                                    &::Sustracción,
                                    &::Multiplicación, &::División};

flotar a, b, c; int optar; // ingresa
los operandos cin >> a >> b;


// ingresa la operación 0-Suma, 1-Resta, 3-Multiplicación, 4-División cin >> opt;


// La siguiente línea reemplaza el modificador y reemplaza todo el modificador c = (*vpf[opt])(a, b);
```

Figura 35: Llamar a funciones similares mediante el uso de puntero a funciones

No es fácil de optimizar por parte de los compiladores. La última instrucción en la Figura 35 c = (vpf[opt])(a, b); potencialmente podría llamar a varias funciones y el compilador tendrá dificultades para optimizarlas.
Por otro lado, el código que se muestra en la Figura 34 es más fácil de optimizar para un compilador.
Aún más, si creamos el vector vpf usando memoria asignada dinámicamente, podría ser aún más flexible.

Entre las ventajas de este enfoque, podemos señalar:

- Ahora tenemos la posibilidad de seleccionar, en tiempo de ejecución.
- Escalabilidad. Permite personalizar los punteros y, en consecuencia, ejecutar acciones específicas del usuario. tareas incluso cuando el programa ya está compilado.
- Podemos reducir drásticamente nuestro código y simplificar el mantenimiento, especialmente cuando switch tiene muchos casos similares.

Entre los inconvenientes podemos destacar:

- Más complejidad para los programadores. Los punteros a funciones son útiles en términos de flexibilidad. pero son peligrosos si se usan incorrectamente.
- Más complejidad para los compiladores. El uso de punteros de función optimiza el compilador. difícil y puede resultar en un rendimiento degradado.
- Este código no evita posibles conflictos cuando el código es programado por muchos desarrolladores simultáneamente. Sin embargo, explicaremos cómo superar este inconveniente en la siguiente sección.

3.9 Interfaces

La idea que inspiró el concepto de clase es la encapsulación. Se crea una clase basada en la siguiente idea: si tenemos las mismas variables

que se usan solo para grupos de funciones, encapsulémoslas y creemos una clase. Las variables se convierten en atributos y las funciones ahora se conocen como métodos [7].

Este proceso se conoce como encapsulación y se considera la piedra angular de la programación orientada a objetos. Consideraremos la clase CVector presentada en la sección 3.3 como punto de partida. Además de los métodos para modificar los datos contenidos (es decir, Insert() y Remove(), etc.), la clase debe tener algunos métodos como Write() y Read() para hacer que los objetos de esta clase sean persistentes. Hasta ahora, estos cuatro métodos pertenecen a nuestra clase pero, si observamos, los métodos Insert() y Remove() no tienen relación semántica con Write() y Read().

La pregunta es: si no existe una relación semántica entre estos dos grupos, pero pertenecen a la misma clase, ¿cómo deberíamos codificarlos? la respuesta es: crear dos interfaces, una que contenga Write() y Read() y la segunda que contenga Insert() y Remove(). Llamaremos a estas dos interfaces IPersist e IContainer y en este caso la clase CVector debe implementar ambas.

Figura 36: Una clase con interfaces

Para fines de codificación, una interfaz se puede implementar como una clase abstracta. Una interfaz no tiene código, su único objetivo es definir los métodos para comunicar a los usuarios con el objeto. Si una clase se hereda de una interfaz, debe implementar todos los métodos contenidos en ella.

```
// Definición de clase CVector
clase CAritmética
{

        privado:
        // Algunos atributos van aquí

        público:
        doble adición estática                                  (doble a, doble b); Sustracción
        doble estática (doble a, doble b); multiplicación doble estática (doble a, doble
        b); (doble a, doble b);
        doble división estática
        // Más métodos van aquí
};
```

Figura 37: Clase para ejecutar operaciones aritméticas

Este nuevo enfoque nos muestra claramente una mejor organización semántica que es útil para fines de mantenimiento y escalabilidad. Clases como CVector, CList, CBinaryTree necesitarán implementar

Mencionan las mismas interfaces pero lo harán con códigos diferentes porque, por ejemplo, Insert() un elemento es diferente en CList y CVector. La implementación es diferente pero el nombre del método será el mismo.

Hay algunas bibliotecas como ATL (ActiveX Template Library) [8] que ya ha implementado varias interfaces predeterminadas, solo necesitamos usarlas. Java y muchos otros lenguajes modernos utilizan la misma idea.

Entre las ventajas de este enfoque, tenemos:

- Los métodos están claramente bien organizados semánticamente.
- Facilita el mantenimiento y mejora la escalabilidad previniendo problemas futuros.

Entre los inconvenientes podemos señalar:

- Es necesario estar más familiarizado con plantillas, etc. Este inconveniente se puede minimizar si lo comparamos con las ventajas que tenemos usando ATL.

3.10 Creación de software tolerante a fallos

Un punto importante a considerar cuando se diseña un software es la tolerancia a fallas. Esto significa que, si sucede algo inesperado, el software debería poder regresar a un punto conocido sin obligar al usuario a cerrarlo. Sólo para visualizar este enfoque, pensemos en una clase CAritmética para ejecutar operaciones aritméticas como Suma(), Resta(), Multiplicación() y División(); todos ellos devuelven un valor doble según el siguiente código.

```
doble x, y, z; ... prueba //
nuestra sección crítica con posibles excepciones comienza aquí {

        z = CAritmética::División(x, y); cout <<z;

}

capturar (int código de error) // capturar la excepción {

        // Mostrar algún mensaje, etc.
}
```

Figura 38: Uso de una excepción

```
doble CAritmética::División(doble a, doble b) {

        si ( b == 0.0 ) lanzar
                (int)0; // aquí se lanza una excepción
        devolver a/b;
}
```

Figura 39: Lanzar una excepción

A veces podemos controlar si sucedió algo mal porque la función devuelve un código especial. Por ejemplo, la función factorial solo puede devolver valores positivos y podríamos devolver -1 si el usuario la llama con un parámetro negativo pero, desafortunadamente, no siempre es posible. Pensemos en el método División(x,y) llamado con y = 0. En este caso no podríamos devolver -1 o 0 para decir que algo salió mal porque ambos valores (0 y -1) también podrían ser el resultado de un división.

Una solución elegante para este caso es utilizar excepciones. En las Figuras 38 y 39 podemos ver cómo úsalos.

Si sucede algo mal, el contador del programa saldrá de la función restaurando la pila y buscando alguna captura con el tipo apropiado (int en este caso).

Entre las ventajas de este enfoque, podemos destacar:

• Tolerancia a fallos y claridad.

Entre los inconvenientes podemos destacar:

• Se requiere más complejidad y conocimiento.

3.11 Consideraciones del lenguaje de programación

El lenguaje de programación elegido es, evidentemente, otro punto importante a analizar. Cada lenguaje de programación tiene sus propias fortalezas y debilidades. El objetivo es utilizar cada lenguaje de programación para aquellas tareas en las que sea más efectivo.

En el caso específico de estructuras de datos recomendamos el uso de ANSI C++ es un lenguaje de programación muy popular, que proporciona muchas características para el diseño de estructuras de datos flexibles.
Con C comparte compiladores muy eficientes pero ofrece mejor soporte para la abstracción de datos a través de sus características orientadas a objetos. Fue diseñado como un lenguaje de programación de propósito general, con un sesgo hacia la programación de sistemas que soporta computación eficiente de bajo nivel, abstracción de datos, programación orientada a objetos y programación genérica [7].

Hoy en día existen muchos lenguajes de programación, pero si el objetivo es velocidad y eficiencia tenemos que considerar C++. Esta es la razón principal por la que se eligió C++ como lenguaje de programación predeterminado para UNIXTM, Linux, Microsoft Windows, Solaris, etc. Por otro lado, también debemos considerar el tiempo extra necesario para depurar programas en C++.

JavaTM (Sun Microsystems, Inc.) [4] es también otra alternativa interesante especialmente para el desarrollo web que está ganando cada vez más popularidad. Si bien su rendimiento sigue siendo algo inferior al de los programas C++, su soporte de administración automática de memoria y su estricta disciplina de verificación de tipos previene muchos errores de programación comunes. Dado que el tiempo del programador es más valioso y más caro que el tiempo de la CPU, hoy en día y los compiladores ya han mejorado considerablemente para Java, es probable que aumente el desarrollo futuro de software en Java. Su sólida verificación de tipos y recolección de basura lo hacen también atractivo para cursos de programación para estudiantes, ya que previenen errores comunes y permiten a los estudiantes concentrarse en la tarea principal que tienen entre manos y no perder tiempo en errores que se pueden prevenir a nivel del lenguaje de programación.

Por lo tanto, cuando el rendimiento del programa no es crítico (como es el caso de la mayoría de los proyectos de software), las características del lenguaje deberían ser más importantes que el bajo nivel de eficiencia. Si bien Java es un gran paso en la dirección correcta para prevenir errores simples de programación, muchos, en particular los programadores de C++, omiten plantillas. Sin embargo, las versiones futuras pueden incorporar esa característica también, lo que convertiría a Java en el vehículo perfecto para enseñar estructuras de datos en el aula.

3.12 Más allá de las barreras del lenguaje de programación y del sistema operativo

Todos los lenguajes de programación existentes presentan sus propias fortalezas y debilidades. Para mejorar nuestro producto final, tenemos que utilizar lo mejor de cada idioma. Según este punto de vista, y, más aún, considerando el crecimiento exponencial de Internet, nuestro software debería estar diseñado para ser utilizado no sólo por clientes programados con el mismo lenguaje sino también por programas de otros lenguajes.

Teniendo en cuenta estas consideraciones, es muy importante crear objetos en lenguajes de programación cruzada. El mismo requisito se aplica a la compatibilidad entre diferentes sistemas operativos. Existen muchos lenguajes y modelos de programación que permiten lograr este objetivo.

Entre los más importantes podemos citar C++, Java de SUN Microsystems (java.sun.com), y los más nuevos como la plataforma Microsoft .NET, etc.

También es importante considerar STL [6] como una alternativa para crear estructuras de datos portátiles. Esta biblioteca es parte de ANSI C++ y debe ser compatible con cualquier compilador de C++.

Utilizando diferentes lenguajes de programación, tenemos la oportunidad de aprovechar las ventajas mejores características de cada uno de ellos.

4 La calidad de los algoritmos utilizados

Todas las técnicas presentadas en las secciones 3.1 a 5 no serían útiles si el algoritmo utilizado no fuera eficiente. La eficiencia y el coste computacional de un algoritmo siempre serán el punto más importante a la hora de diseñarlo [5]. También debemos prestar atención a la claridad y el mantenimiento del código.

Sólo para ilustrar este punto, analizamos la conocida serie de números de Fibonacci, que se define recursivamente de la siguiente manera: fibo(0) = 0; fibo(1) = 1; fibo(n) = fibo(n-1) + fibo(n-2). La recursividad es una construcción importante y útil proporcionada por la mayoría de los lenguajes de programación, que permite la formulación de soluciones elegantes y concisas para muchos problemas. Sin embargo, el uso ingenuo de la recursividad puede degradar el rendimiento y las soluciones iterativas suelen ofrecer alternativas más rápidas.

La Figura 40 muestra una implementación recursiva directa de la serie de Fibonacci en el lenguaje de programación C:

```
fibo largo sin firmar (n sin firmar) {

        si (n < 2)
                devuelve 1;
        devolver fibo(n-1) + fibo(n-2);
}
```

Figura 40: Implementación recursiva para números de Fibonacci

En la Figura 41 se muestra una versión iterativa alternativa. Requiere más código que la primera solución y es algo más difícil de entender, aunque, dada la pequeñez de la función, todavía es fácil de comprender en general. Por otro lado, requiere menos memoria (espacio de pila para parámetros o locales). Aún más importante, es considerablemente más rápido ya que la versión recursiva hace mucho más trabajo, como se muestra en la Figura 41. En contraste, la implementación iterativa solo ejecuta un bucle para producir el mismo resultado a un costo lineal. Además, como $\frac{Fn+1}{fn} \to \frac{(1+\sqrt{5})}{2}$, entonces Fn > 1,6 y para calcular Fn necesitamos 1,6 llamadas recursivas, es decir, el tiempo de ejecución de la versión recursiva es exponencial en el tamaño de la entrada. Evidentemente, el uso ingenuo de

La recursividad, aunque atractiva debido a la correspondencia directa con el problema original, puede tener serios inconvenientes. La Figura 42, que muestra los cálculos recursivos realizados para fibo(4), demuestra el despilfarro de la implementación recursiva al hacer las operaciones repetidas cálculo de los mismos valores obvio.

```
fibo largo sin firmar (n sin firmar) {

        si (n < 2)
                devuelve 1;
        largo sin firmar f1 = 0, f2 = 1; salida larga sin firmar;
        hacer

        {
                salida = f1 + f2; f1 = f2; =
                salida; }mientras(--n
                f2              >= 2); salida
        de retorno;

}
```

Figura 41: Implementaciones iterativas para números de Fibonacci

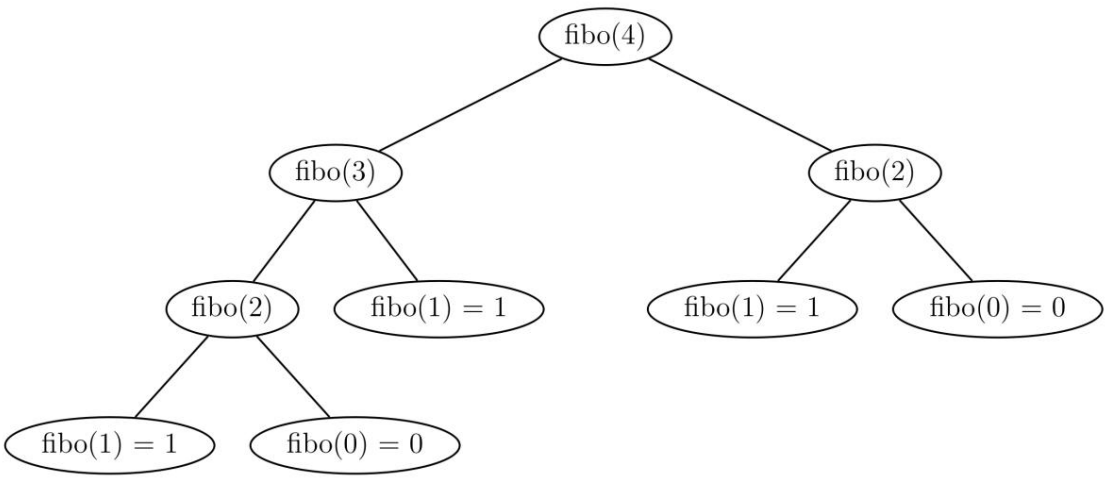

Figura 42: Una clase con interfaces

5 Con un poco de ayuda del compilador: escribir de forma clara pero eficaz código antiguo

Mientras enseñamos a los estudiantes la importancia de una estructura de datos eficiente y un diseño e implementación de algoritmos, no queremos perder de vista el objetivo principal de escribir código claro y legible. En el pasado, cuando la optimización de la tecnología de compilación aún estaba en su infancia, la claridad y la eficiencia a veces estaban reñidas. Con frecuencia, esto lleva a los programadores, particularmente a los programadores de C, a escribir código eficiente pero difícil de leer y comprender. Sin embargo, los compiladores de optimización actuales pueden ayudarnos a concentrarnos en los pasos esenciales para elegir un buen algoritmo y estructuras de datos adecuadas y luego dejar gran parte del ajuste del rendimiento al optimizador del compilador. Por lo tanto, saber qué pueden y no pueden hacer los compiladores es importante cuando enseñamos a los estudiantes cómo implementar sus algoritmos.

En la sección 4, utilizamos la función de Fibonacci como ejemplo para mostrar a los estudiantes que deben desconfiar de una implementación recursiva simple, o más bien ingenua, debido a los costos exponenciales de tiempo y espacio asociados. Sin embargo, a menudo la recursividad es la forma más clara de establecer una solución inicial y luego podemos transformarla en iteración manualmente o, en algunos casos, dejar que el compilador haga esa tarea por nosotros. Para ilustrar este punto, podemos usar la rutina de búsqueda binaria que se muestra en 43, que encuentra una clave de valor en una sección de la matriz a, desde el elemento a[l] hasta a[h], inclusive. La matriz contiene números de doble precisión y asumimos que está ordenada en orden descendente. Si se encuentra el valor clave, se devuelve su posición en la matriz; de lo contrario, la rutina devuelve -1 para indicar que el valor no está presente.

```
int bsearch(doble a[], doble clave, int l, int u) {

        int metro;
        si (u<l)
                devuelve -1;
        metro = (l+u) / 2; si
        (a[m] == clave) devuelve
                m; si (a[m]
        <tecla)
                devolver bsearch(a, clave, l, m-1);
        demás
                devolver bsearch(a, clave, m+1, u);
}
```

Figura 43: Rutina de búsqueda binaria recursiva. Dado que bsearch() es recursivo de cola, un buen compilador optimizador convertirá automáticamente la recursividad de cola en iteración, como se muestra en 44.

La búsqueda binaria tiene una formulación recursiva muy clara y concisa. Primero, no se puede encontrar ningún valor en una sección de matriz vacía; esto se comprueba comparando u y h. Luego, el índice del elemento pivote se calcula en una variable auxiliar m, y si se encuentra la clave allí, se devuelve el índice. Si la clave es mayor que el elemento pivote, dado que la matriz está ordenada en orden descendente, la clave solo se puede encontrar en la mitad inferior y se busca de forma recursiva en la mitad inferior; de lo contrario, se busca en la mitad superior en una llamada recursiva. Dado que no hay cálculos después de la llamada recursiva a bsearch, la rutina es recursiva de cola y podemos mostrar a los estudiantes cómo dicha rutina se puede convertir mecánicamente en una versión iterativa: simplemente realice las asignaciones de los reales a los formales en la llamada recursiva y reemplazar la llamada por un ir al inicio

de la rutina (por supuesto, en la práctica uno podría fusionar el goto y el if en un bucle while) y no usar goto en el programa.

Dado que este es un procedimiento simple y mecánico, debería ser fácil para los estudiantes darse cuenta de que, de hecho, un buen compilador debería poder hacer esto automáticamente y, a modo de ilustración, podemos presentar el resultado del compilador GNU C ampliamente disponible para este rutina, que se muestra para el procesador Alpha en 44.

Por supuesto, no todas las funciones recursivas se pueden convertir en versiones iterativas sin simular explícitamente la pila. Sin embargo, muchas funciones que no son naturalmente recursivas de cola se pueden reescribir fácilmente para que sean recursivas de cola utilizando un acumulador, que se utiliza para transportar resultados intermedios de una invocación a otra.

```
bbúsqueda:
        .marco $30,0,$26,0
$bbuscar..ng:
        .prólogo 0
$L5:
        cmmplt $19,$18,$1 beq       # !(u<l) ir a L2
        $1,$L2 lda                  #
        $0,-1 br                    # m = -1 y regresar (desde L7)
        $31,$L7
$L2:
        addl $19,$18,$1 srl         # calcular u+l
        $1,63,$2 addq               # dividir por 2 con turno
        $1,$2,$1 sra                #
        $1,1,$0 s8addq              # metro = (u+l)/2
        $0,$16,$3 ldt $f11,0($3)    # calcular la dirección de a[m]
        cmpteq $f11,$f17,$f10       # a[m] -> $f11
        # a[ m] == tecla ir a L7
        fbne $f10,$L7
        cmptlt $f11,$f17,$f10 # a[m] <tecla ir a L4
        fbeq $f10,$L4
        subl $0,1,$19 br            # tu = m-1
        $31,$L5                     # ir a L5
$L4:
        adicional $0,1,$18          # l = m+1
        br $31,$L5                  # ir a L5
$L7:
        ret $31,($26),1 .end bsearch  # devolución m (en $0)
```

Figura 44: La rutina de búsqueda binaria escrita recursivamente de la Figura 43 compilada por GNU Compilador C (gcc) con opción de optimización nivel O2 para el procesador Alpha. el compilador automáticamente convirtió la recursividad en iteración y eligió realizar la división de enteros por 2 con una operación de turno. Este código es tan eficiente como cualquier versión iterativa escrita a mano, pero la formulación recursiva es más clara.

La Figura 45 muestra un ejemplo sencillo de esta técnica para la función factorial. factorial es reescrito utilizando una función auxiliar (fact helper) que toma dos argumentos a y n, y que calcula recursivamente an!. Al acumular resultados intermedios en el primer argumento de la función auxiliar, la función recursiva se vuelve recursiva de cola y la función factorial luego se implementa simplemente como una llamada específica a la función auxiliar que inicializa el acumulador a 1. Al declarar la función auxiliar estática[3], el programador se comunica con el compilador que la función auxiliar se use solo localmente, de modo que el compilador pueda eliminar de forma segura su código si puede incorporar al ayudante en (todos) sus interlocutores. Además, cuando compilamos esta versión del función factorial con un compilador optimizador, automáticamente convertirá la recursividad en iteración, inserte la función auxiliar en la función de hecho, lo que elimina la llamada adicional

[3] static es una palabra clave en el lenguaje de programación C que, cuando se usa para calificar una función, restringe la alcance (visibilidad) de la función al archivo en el que se define la función, de modo que no se pueda llamar a la función de funciones en diferentes archivos.

sobrecarga introducida por la llamada a la función auxiliar y eliminar la función auxiliar del texto del programa ya que no se llama desde ningún otro lugar. Es decir, sin ninguna sobrecarga adicional en el tamaño del programa o el tiempo de ejecución, el programador puede escribir la función de forma recursiva, en una forma muy cercana a su definición matemática.

```
typedef unsigned largo ulong;

estático ulong fact_helper(ulong a, ulong n) {

    si (n == 0)
        devolver un;
    demás
        devolver fact_helper(a*n, n-1);
}

hecho ulong (ulong n) {

    devolver fact_helper(1,n); }
```

Figura 45: Versión recursiva de cola de la función factorial. Usando una función auxiliar, el factorial se puede implementar con recursividad de cola, que un compilador optimizador convierte automáticamente en iteración.

Obviamente, el ejemplo es lo suficientemente trivial como para que sería fácil escribirlo de forma iterativa en la práctica. Sin embargo, con esta técnica se pueden implementar funciones recursivas más complicadas de forma clara y concisa sin sacrificar el rendimiento, gracias a la moderna tecnología de compilación4 .

Lo que los estudiantes pueden aprender de estos ejemplos es que vale la pena saber un poco sobre lo que sucede debajo de las sábanas. El código eficiente no tiene por qué ser ilegible y, en la mayoría de los casos, el código que es difícil de optimizar para un compilador también será difícil de entender, leer y mantener para los programadores humanos. Por lo tanto, cuando enseñamos diseño de algoritmos y estructuras de datos eficientes, es importante no perder de vista cuál debería ser el objetivo primordial: producir diseños buenos y fáciles de mantener sin descuidar el rendimiento.

Y como lo han ilustrado los ejemplos, la eficiencia y la simplicidad generalmente se pueden lograr al mismo tiempo.

6 Resumen y conclusiones

En este artículo presentamos diferentes enfoques para implementar estructuras de datos. Casi todos estos enfoques se pueden utilizar para cualquier estructura de datos.

En la sección 2, presentamos una categorización de todas esas técnicas según algunos criterios.
como dinámica, protección de datos, encapsulación y abstracción de datos.

En la sección 3, presentamos doce enfoques diferentes para implementar estructuras de datos como Programación Modular, Programación Orientada a Objetos, Tipos de Datos Abstractos, Patrones de Diseño, código para grupos de trabajo, espacios de nombres, tolerancia a fallas, entre otros.

4Por supuesto, en la práctica siempre se debe comprobar que el compilador en cuestión sea lo suficientemente inteligente como para realizar las transformaciones descritas. De lo contrario, reescribir una función de la forma descrita sigue siendo útil como primer paso para convertir manualmente la implementación recursiva en una implementación iterativa.

En la sección 4, presentamos la importancia de diseñar un buen algoritmo. es importante preste atención antes de usar la recursividad. Es una excelente alternativa cuando se usa correctamente.

En la sección 5, explicamos la importancia de aprovechar un compilador optimizador para código de diseño que sea legible y eficiente.

Existen muchas alternativas para acelerar todo el rendimiento de un programa. El primero y más importante es mejorar el código involucrado y las estructuras de datos, pero también es importante aprovechar el tiempo de inactividad de la CPU producido por las operaciones de E/S. Una alternativa interesante para lograr este objetivo son los hilos [1]. Cuando diseñamos un programa Multi-Threading no sólo estamos preparando el código para un entorno multiprocesador sino también para aprovechar el tiempo de inactividad incluso con un solo procesador.

Otro punto importante para la codificación de estructuras de datos y el desarrollo de software es la documentación. Facilita el trabajo en grupo y el mantenimiento del software durante todo el ciclo de desarrollo.

Entre las conclusiones finales podemos destacar:

- Una estructura de datos eficiente es el resultado de combinar lo mejor de todas estas técnicas según las circunstancias.
- Es muy importante reconocer las diferencias potenciales entre los enfoques presentados. para poder aplicarlos correctamente.

Expresiones de gratitud

Los autores desean agradecer a Alex Cuadros-Vargas y Eduardo Tejada-Gamero de la Universidad de Sao Paulo-Brasil y a Miguel Penabad de la Universidade da Coru˜na-España por sus útiles comentarios. También quisieran agradecer a Julia Vel´asquez-M´alaga de la Universidad Católica San Pablo-Per´ú por su trabajo revisando y corrigiendo errores tipográficos.

Referencias

[1] Aaron Cohen, Mike Woodring y Ronald Petrusha (Editor). Programación multiproceso Win32. O'Reilly y asociados, 1998.

[2] Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides. Patrones de diseño de elementos de software reutilizable orientado a objetos. Serie de Computación. Addison Wesley Profesional, octubre de 1994.

[3] K. Geiger. Dentro de ODBC. Prensa de Microsoft, 1995.

[4] Bill Joy, Guy Steele, James Gosling y Gilad Bracha. Especificación del lenguaje Java(tm). Addison-Wesley Pub Co, segunda edición, junio de 2000.

[5]Donald Ervin Knuth. El arte de la programación informática. Addison-Wesley, Reading, MA, 1-3, 1973.

[6] Alexander Stepanov y Meng Lee. La biblioteca de plantillas estándar. HPL 94-34, laboratorios HP, Agosto de 1994.

[7] B. Stroustrup. El lenguaje de programación C++ (edición especial). Addison-Wesley, 2000.

[8] Andrew Troelsen. Taller de desarrolladores para com y atl 3.0. Publicación de Wordware, 2000.

Contenido