APUNTES SOBRE

PROGRAMACIÓN

ORIENTADA A OBJETOS

ADAPTACIÓN RESUMIDA DEL TEXTO DE TIMOTHY BUDD

AN INTRODUCTION TO OBJECT−ORIENTED PROGRAMMING

2001

ÍNDICE

1. INTRODUCCIÓN 2

2. CLASES Y MÉTODOS 9

3. MENSAJES, EJEMPLARES E INICIALIZACIÓN 13

4. HERENCIA 21

5. ENLACES ESTÁTICO Y DINÁMICO 33

6. REEMPLAZO Y REFINAMIENTO 38

7. HERENCIA Y TIPOS 48

8. HERENCIA MÚLTIPLE 59

9. POLIMORFISMO 65

10. VISIBILIDAD Y DEPENDENCIA 71

1. INTRODUCCIÓN

1.1. UN NUEVO PARADIGMA

*** Paradigmas clásicos**

Programación imperativa, programación funcional, programación lógica.

*** Significado de la palabra paradigma**

Ejemplo ilustrativo, enunciado modelo. El historiador Thomas Kuhn extiende tal definición: conjunto de teorías, estándares y métodos que en conjunto representan una forma de organizar el conocimiento, es decir, una forma de ver la realidad. Luego, la programación orientada a objetos (POO) es un nuevo paradigma.

1.2. UNA MANERA DE VER LA REALIDAD

*** Ejemplo ilustrativo**

Paco desea enviar de regalo un loro a su amigo Pepe quien vive en una lejana ciudad. Para ello, accede a Perico, el pajarero local, a quien le describe las características del loro objeto del regalo; con esto, Paco puede tener la certeza de que el loro será oportunamente entregado.

*** Mensajes y métodos**

En la POO, la acción se inicia mediante la transmisión de un mensaje a un agente (un objeto) responsable de la acción. El mensaje tiene codificada la petición de una acción y se acompaña de cualquier información adicional (argumentos) necesaria para llevar a cabo la petición. El receptor es el agente al cual se envía el mensaje. Si el receptor acepta el mensaje, acepta la responsabilidad de llevar a cabo la acción indicada. En respuesta a un mensaje, el receptor ejecutará algún método para satisfacer la petición.

*** Diferencias entre mensaje y llamada a un procedimiento**

El mensaje posee un receptor designado; el receptor es algún agente al cual se envía el mensaje; la llamada a un procedimiento carece de un receptor designado, aunque se podría adoptar la convención, por ejemplo, de siempre llamar receptor al primer argumento de un procedimiento.

La interpretación del mensaje (es decir, la selección de un método para responder a un mensaje) depende del receptor y puede variar con diferentes receptores. Por ejemplo, si Paco da el mismo mensaje a su hermana Peca, ella lo entenderá y se obtendrá un resultado satisfactorio. Sin embargo, el método que Peca use para satisfacer la petición (seguramente sólo transmitirá la solicitud a Perico) será diferente del ejecutado por Perico como respuesta a la misma demanda.

*** Enlaces temprano y tardío**

Por lo general, el receptor específico para cualquier mensaje dado, no se conoce sino hasta el tiempo de ejecución, por lo que la determinación del método a usar no puede hacerse hasta ese momento. Luego, se dice que hay un enlace tardío entre el mensaje (nombre de la función) y el fragmento de código (método) usado para responder al mensaje. Esta situación es opuesta a la de enlace temprano (en tiempo de compilación o de ligadura) entre el nombre y el fragmento de código en llamadas a procedimientos convencionales. Es importante notar que el comportamiento se estructura en términos de las responsabilidades. La solicitud de Paco sólo indica el resultado deseado (un loro para Pepe). Por conocimientos previos, Pepe (y cada usuario) tiene una idea de cómo funciona una pajarería y, por lo tanto, espera un detrminado comportamiento. El pajarero Perico, es libre de seguir cualquier técnica para lograr su objetivo. Por otra parte, puede usarse el término Pajarero para representar la categoría (o clase) de todos los pajareros. Como Perico pertenece a la categoría Pajarero, es decir, es una instancia de los Pajareros, esperamos un determinado comportamiento.

*** Clases y ejemplares**

Todos los objetos son ejemplares de una clase. El método invocado por un objeto, en respuesta a un mensaje, queda determinado por la clase del receptor. Todos los objetos de una clase dada usan el mismo método en respuesta a mensajes similares. Perico, el pajarero, cobrará por la transacción y a cambio del pago otorgará un recibo. Estas acciones son válidas para cualquier comerciante. Como la categoría Pajarero es una forma más especializada de la categoría Comerciante, cualquier conocimiento que se tenga de los comerciantes es también válido para los pajareros. La forma de pensar en cómo se ha organizado el conocimiento acerca de Perico es en términos de una jerarquía de categorías. Perico es Pajarero, pero Pajarero es una forma especializada de Comerciante; Comerciante es también un Humano; así se sabe, por ejemplo, que Perico es bípedo. De esta manera, gran parte del conocimiento que se tiene y que es aplicable también a la categoría más específica se llama herencia. Se dice que Pajarero heredará atributos de la categoría Comerciante.

*** Representación gráfica de una estructura jerárquica de clases**

Objeto material

Animal Vegetal

Mamífero Flor

Perro Humano Ornitorrinco

Artista Comerciante Dentista

Ceramista Pajarero

Ciro Perico Dante Oto

*** Herencia**

Las clases se pueden organizar en una estructura de herencia jerárquica. Una subclase heredará atributos de una superclase que esté más arriba en el árbol. Una superclase abstracta es una clase (como Mamífero) que se usa sólo para crear subclases y para la cual no hay ejemplares directos. Sin embargo, el ornitorrinco representa un problema en la estructura planteada pues, aunque se trata de un Mamífero, se reproduce por huevos. Luego, se requiere un método que codifique excepciones a la regla general. Esto se consigue indicando que la información contenida en una subclase puede anular información de una superclase. La implantación de este enfoque adopta la forma de método de una subclase que tiene el mismo nombre que el método de la superclase, en combinación con una regla que indique cómo se llevará a cabo la búsqueda de un método que corresponda a un mensaje específico.

*** Enlace de métodos**

La búsqueda para encontrar un método que pueda invocarse en respuesta a un mensaje dado, empieza con la clase del receptor. Si no se encuentra un método apropiado, se lleva la búsqueda a la superclase de dicha clase. La búsqueda continúa hacia arriba de la cadena de la superclase, hasta que se encuentra un método o se agota la cadena de la superclase. En el primer caso, el método se ejecuta; en el último, se emite un mensaje de error. Aun cuando el compilador no pueda determinar qué método se invocará en tiempo de ejecución, en muchos lenguajes OO se puede determinar si habrá un método apropiado y enviar así el mensaje de error como diagnóstico en tiempo de compilación más que como un mensaje en tiempo de ejecución. El que Peca y Perico respondan al mismo mensaje de Paco, ejecutando diferentes métodos, es un ejemplo de una forma de polimorfismo. El hecho de que no se necesita conocer qué método usará Perico para responder al mensaje de Paco, es un ejemplo de ocultación de la información.

*** Computación como simulación**

El modelo tradicional que describe el comportamiento de un computador al ejecutar un programa es el de sucesión de estados. En éste, el computador es un manejador de datos que sigue un patrón de instrucciones a través de la memoria, obtiene valores de diversas posiciones, los transforma de alguna manera y devuelve los resultados a otras posiciones. Aunque este modelo representa aproximadamente lo que pasa al interior de un computador, poco ayuda a entender cómo resolver problemas por medio de él. En cambio, en el marco OO, nunca se mencionan direcciones de memoria, variables, asignaciones o algún otro de los términos convencionales de programación. En su lugar se habla de objetos, mensajes y responsabilidad por alguna acción. Ver la programación como herramienta capaz de crear un universo de objetos se asemeja al estilo denominado simulación determinada por eventos discretos. En una simulación de esta naturaleza, el usuario

crea modelos de computación con diferentes elementos de la simulación, describe como interactúan unos con otros y los pone en marcha. En un programa OO promedio, el usuario describe lo que son las diversas entidades en el universo del programa y como actuarán recíprocamente para, por último, ponerlas en movimiento. Así, en la POO se tiene el enfoque de que computación es simulación.

*** Una concepción recurrente**

Los objetos no siempre pueden responder a un mensaje solicitando a otro objeto que ejecute alguna acción, pues el resultado sería un círculo infinito de solicitudes. En algún punto, al menos unos cuantos objetos necesitan realizar algún trabajo que no sea el de pasar peticiones a otros agentes. En lenguaje híbridos (como C++ y Object Pascal) se efectúa mediante métodos escritos en el lenguaje base (no OO). En lenguajes OO puros (como Smalltalk) se lleva a cabo mediante la introducción de operaciones primitivas dadas por el sistema subyacente.

1.3. MECANISMOS DE ABSTRACCIÓN

Las técnicas orientadas a objetos se pueden ver como el resultado natural de una evolución histórica desde los subprogramas, módulos, TAD y objetos.

*** Subprogramas**

Los procedimientos y las funciones fueron uno de los primeros mecanismos de abstracción usados ampliamente por los lenguajes de programación. Permitían que las tareas repetitivas o que manifestaran ligeras variaciones, fueran agrupadas en un lugar y reutilizadas, en vez de multiplicar el código. Los subprogramas proveyeron la primera posibilidad para la ocultación de la información, sin llegar a ser un mecanismo efectivo pues resolvieron sólo parcialmente el problema del uso de idénticos nombres por múltiples programadores. Para ilustrar lo anterior, considérese el TAD stack con las operaciones create, push, pop y empty. Sin importar la implementación (estática o dinámica) es fácil ver que los datos contenidos en el stack no pueden ser locales para ninguno de los operadores, pues deben ser compartidos por todos. Pero, si las únicas opciones son variables locales o globales, entonces los datos del stack deben mantenerse en variables globales. Sin embargo, si las variables son globales, no hay modo de limitar la accesibilidad o visibilidad de estos nombres. El mecanismo de ámbito de bloque de ALGOL y sus sucesores, como Pascal, ofrece un control algo mayor sobre la visibilidad de los nombres. Sin embargo no resuelve el problema de ocultación de la información, pues cualquier ámbito que permita el acceso a los cuatro operadores mencionados debe también permitirlo a sus datos comunes.

*** Módulos**

Los módulos pueden considerarse simplemente una técnica mejorada para crear y manejar espacios de nombres. El ejemplo del stack es típico en cuanto a que existe cierta información (las rutinas de interfaces) amplia y públicamente disponible, mientras que hay otros datos (los contenidos en el stack) de carácter restringido. Luego, un módulo brinda la capacidad de dividir un espacio de nombres en dos partes: una pública, accesible desde fuera del módulo, y una privada, accesible sólo dentro del módulo. Todos los tipos, los datos (variables) y los procedimientos pueden ser definidos en cualquier parte. David Parnas, quien popularizó la noción de módulo, describió los siguientes dos principios para su uso apropiado:

− Se debe proporcionar al usuario toda la información necesaria para usar correctamente el módulo, y nada más.

− Se debe proporcionar al implementador toda la información necesaria para completarlo, y nada más.

Este encubrimiento explícito e intencional es la ocultación de la información. Los módulos resuelven algunos

de los problemas del desarrollo de software, pero no todos. Por ejemplo, los módulos permitirán al programador ocultar los detalles de la implementación del stack pero, ¿qué pasa si otros usuarios desean tener dos o más stacks? Los módulos por sí mismos ofrecen un método efectivo para la ocultación de la información pero no permiten la creación de ejemplares (instanciación) la cual es la capacidad de hacer copias múltiples de las áreas de datos.

*** Tipos abstractos de datos**

Un tipo abstracto de dato (TAD) es aquél definido por el programador, que puede ser manipulado de una manera similar a los definidos por el sistema. Consta de un conjunto de valores permitidos y de una colección de operaciones que pueden ejecutarse sobre ellos. El mecanismo de construcción de un TAD, debe permitir:

− Exportar una definición de tipo.

− Proporcionar un conjunto de operaciones que puedan usarse para manipular los ejemplares del tipo.

− Proteger los datos asociados con el tipo de tal manera que se pueda operar con ellos sólo mediante las operaciones provistas.

− Crear múltiples ejemplares del tipo.

Los módulos proveen sólo los puntos segundo y tercero, aunque los otros dos pueden tener cabida si se usan técnicas de programación apropiadas. Los packages, encontrados en lenguajes como CLU o Ada, son un intento por alcanzar en forma más directa todos los puntos indicados. Desde el punto de vista de la definición, un objeto es simplemente un TAD; desde el punto de vista de la implementación y del uso, un objeto involucra importantes innovaciones en lo que a compartición y reusabilidad de código se refiere.

*** Objetos: mensajes, herencia y polimorfismo**

Las técnicas de POO agregan algunas ideas nuevas e importantes al concepto de TAD. La primera de ellas es la idea de paso de mensajes. La actividad se inicia con una solicitud hecha a un objeto específico, no por la invocación de una función que use datos específicos. En gran medida, esto es meramente un cambio de énfasis: el enfoque convencional concede importancia primaria a la operación, mientras que el OO la otorga al valor mismo (¿se llama al operador push con un stack y un valor o se pide al stack que ponga un valor en sí mismo?). Sin embargo, además del paso de mensajes existen mecanismos poderosos para sobrecargar nombres y reutilizar software.

En el paso de mensajes está implícita la idea de que un mensaje puede variar de acuerdo con los diferentes objetos. Esto es, el comportamiento y la respuesta que el mensaje provoca dependerán del objeto que lo recibe. Así, push puede tener un significado para un stack y otro muy diferente para el controlador de un brazo mecánico. Puesto que los nombres de las operaciones no necesitan ser únicos, pueden usarse formas simples y directas, lo que llevará a un código más legible y comprensible.

La herencia permite que diferentes tipos de datos compartan el mismo código, lo que conduce a una reducción del tamaño del código y a un incremento en su funcionalidad.

El polimorfismo permite que este código compartido sea adaptado para que se ajuste a las circunstancias específicas de cada tipo de datos individual. El énfasis en la independencia de componentes individuales permite un proceso de desarrollo incremental, en el cual unidades de software individuales se diseñan, programan y prueban antes de combinarse en un sistema grande.

1.4. SOFTWARE REUTILIZABLE

Durante mucho tiempo, la gente se ha preguntado porqué la construcción del software no refleja más de cerca la elaboración de otros objetos materiales. Por ejemplo, cuando se construye un edificio, un automóvil o un aparato electrónico, por lo general se ensambla cierto número de componentes prefabricados, en vez de fabricar cada nuevo elemento partiendo de nada. Una razón importante para ello es la estrecha interconexión de la mayoría de los programas que se crearon en forma convencional. Cómo se vio antes, es difícil extraer elementos de software de un proyecto que puedan usarse fácilmente en otro no relacionado, porque es típico que cada parte de código tenga interdependencias con otras partes del mismo. Tales interdependencias pueden ser resultado de definiciones de datos o pueden ser dependencias funcionales. Por ejemplo, organizar registros en una tabla y realizar operaciones de búsqueda indexada en dicha tabla son, tal vez, algunas de la operaciones más comunes de la programación. Y, aun así, las rutinas de búsqueda vuelven a escribirse casi siempre para cada nueva aplicación. Esto, porque en los lenguajes convencionales el formato de registro de los elementos está muy ligado al código más general de inserción y búsqueda. Es difícil escribir un código que funcione para datos arbitrarios y para cualquier tipo de registro. Las técnicas orientadas o objetos proveen el mecanismo para separar con limpieza la información esencial (inserción y eliminación) de la información de poca importancia (formato de registros particulares). De esta manera, usando las técnicas OO, se puede llegar a construir grandes componentes de software reutilizable.

2. CLASES Y MÉTODOS

2.1. ENCAPSULACIÓN

Considérense los objetos como ejemplos de TAD's. La programación mediante abstracciones de datos es un enfoque metodológico para la solución de problemas, en el que la información se oculta conscientemente en una parte del programa. En particular, el programador desarrolla una serie de TAD's. Cada TAD se puede ver como si tuviera dos caras: desde el exterior, el usuario ve nada más que un conjunto de operaciones que definen el comportamiento de la abstracción; desde el interior, el programador (que define la abstracción) ve las variables de datos que se usan para mantener el estado interno del objeto.

Una clase es el mecanismo que permite realizar abstracciones de datos en lenguajes de programación orientados a objetos.

Se ha usado el término ejemplar para nombrar a un representante de una clase. Luego, se usará el término variable de ejemplar para aludir a una variable interna mantenida por un ejemplar. Cada ejemplar tendrá su propia colección separada de variables de ejemplar. Estos valores no deben ser modificados directamente por los clientes, sino por los métodos asociados con la clase.

Así, una visión sencilla de un objeto es la combinación de estado y comportamiento. El estado se describe mediante las variables de ejemplar, en tanto que el comportamiento se caracteriza por los métodos. Desde el exterior, los clientes sólo pueden ver el comportamiento de los objetos; desde el interior, los métodos proporcionan el comportamiento apropiado por medio de las modificaciones del estado.

2.2. INTERFAZ E IMPLEMENTACIÓN

Parte de la evolución de las ideas de la POO conservan y amplían conceptos anteriores sobre modularidad y ocultamiento de la información. En particular, los principios de Parnas se pueden adaptar a las técnicas OO:

1. La definición de una clase debe proporcionar al usuario toda la información necesaria para manipular correctamente un ejemplar de la clase, y nada más.

2. Un método debe recibir toda la información necesaria para llevar a cabo sus responsabilidades, y nada más.

Estos principios admiten dos formas de percepción del mundo de un objeto. Por una parte está el mundo real

tal como lo ve el usuario de un objeto, vista a la cual se le llamará interfaz ya que describe cómo se conecta el objeto con el mundo. Por otra parte está la visión que surge desde el interior del objeto, a la cual se le llamará implementación. Al usuario de un objeto se le permite acceso sólo a lo que está descrito en la interfaz. La implementación describe cómo se logra la responsabilidad prometida en la interfaz.

La separación entre interfaz e implementación no corresponde exactamente a la encapsulación de datos pues, este último, es un concepto abstracto, mientras que el primero es un mecanismo de implementación. Dicho de otra manera, los módulos se pueden usar en el proceso de implementación de objetos o de TAD's, pero un módulo por sí mismo no es un TAD.

2.3. CLASES Y MÉTODOS

En C++ es útil distinguir entre archivos de interfaz (que convencionalmente reciben una extensión .h) y archivos de implementación (cuyo sufijo varía en cada sistema).

```
// descripción de interfaz para la clase Stack

# ifndef Stack.h

# define Stack .h // incluye este archivo sólo una vez

class Stack

{private:

int v[100]; // Elementos enteros. Implementación estática

int tope; // Indice del top del stack

int Full() // Método que determina si el stack está "lleno"

public:

Stack(); // Constructor

int Empty(); // Inspector que determina si el stack está o no vacío

int Push(int); // Agrega un elemento al tope del stack

int Pop(); //Retorna el elemento que está en el tope del stack

}

# endif
```

Un archivo de interfaz puede contener descripciones de más de una clase aunque, en general, esta sólo se hace si las clases están muy relacionadas. C y C++ no permiten el uso de la palabra clave import, pero las directivas condicionales de inclusión pueden usarse para conseguir el mismo efecto: la primera vez que se incluya el archivo StackDef, no se habrá definido la clase Stack (que se supone no aparece en ninguna otra parte) y así se cumplirá el enunciado **ifndef** y se leerá el archivo. La clase se conocerá y el archivo se obviará en las sucesivas lecturas del archivo en el módulo.

La descripción de clase es muy parecida a la definición de una estructura, sólo que se permite incluir encabezados de procedimientos así como valores de datos. La palabra clave private precede las partes a las que se puede tener acceso sólo por los métodos de la clase misma, mientras que la palabra clave public indica la verdadera interfaz: los elementos que son accesibles desde fuera de la clase. La descripción de las variables de ejemplar privadas se da sólo para que las conozca el compilador, de tal manera que este pueda determinar los requisitos de memoria para un objeto. Estos campos permanecen inaccesibles para el usuario de una clase.

Cuando aparece void como tipo de retorno de un método, significa que éste se usa a manera de procedimiento por sus efectos laterales y por su resultado funcional.

Los métodos Empty, Push y Pop ilustran la declaración de tipos de parámetros como parte de la declaración de una función, estilo que se conoce como prototipo y se ha adoptado como parte del estándar ANSI del lenguaje C.

Ya que los métodos son tratados simplemente como tipos especiales de campos en un objeto y de otra manera no se distinguen de los campos de datos, no es posible permitir que un método y un campo de datos compartan el mismo nombre.

El archivo de implementación para una clase debe proporcionar definiciones para los métodos descritos en el archivo de interfaz. El cuerpo de un método se escribe como una función convencional en C, con la salvedad de que el nombre de la clase y dos veces dos puntos preceden el nombre del método y de que se puede hacer referencia como variables a las variables de ejemplar (campos) dentro de una clase.

```
// implementación de la clase Stack

# include Stack.h

Stack::Stack()

{ tope=−1;}

int Stack::Full()

{ return (tope= =99);}

int Stack::Empty()

{ return (tope= =−1);}

int Stack::Push(int e)

{ if (!Full())

{ tope++;

v[tope]=e;

return 1;

}

else
```

```
return 0:

}

int Stack::Pop()

{ int e=v[tope];

tope−−;

return e;

}
```

Para motivar el uso de los principios de abstracción y encapsulamiento, C++ proporciona la capacidad de definir funciones en línea. Una función en línea se ve para el que la llama exactamente igual que una función no en línea y se usa la misma sintaxis para parámetros y argumentos. La única diferencia es que el compilador puede escoger expandir funciones en línea directamente a código en el punto de llamada, evitando así el tiempo de procesamiento adicional de la llamada al procedimiento y el retorno de instrucciones.

```
class Stack

{private:

int v[100];

int tope;

int Full()

public:

Stack();

int Empty();

int Push(int);

int Pop();

}

inline Stack::Stack()

{ tope=−1;}

inline int Stack::Full()

{ return (tope= =99);}

inline int Stack::Empty()
```

```
{ return (tope= =−1);}

inline int Stack::Push(int e)

{ if (!Full())

{ tope++;

v[tope]=e;

return 1;

}

else

return 0:

}

inline int Stack::Pop()

{ int e=v[tope];

tope−−;

return e;

}
```

3. MENSAJES, EJEMPLARES E INICIALIZACIÓN

3.1. SINTAXIS DEL PASO DE MENSAJES

Paso de mensajes se entiende como el proceso de presentar a un objeto una solicitud para realizar una acción específica.

En C++, un método se describe como una función miembro y al proceso de paso de un mensaje a un objeto se alude como invocación a una función miembro.

Como en Object Pascal, la sintaxis usada para invocar a una función miembro es similar a la usada para tener acceso a miembros de datos (lo que se ha estado denominando variables de ejemplar). La notación describe al receptor, seguido por un punto, seguido por el selector de mensaje (el cual debe corresponder al nombre de una función miembro), seguido, finalmente, por los argumentos de una lista encerrada entre paréntesis. Si S se declara como ejemplar de la clase Stack, la siguiente instrucción agrega el valor 3 al tope del stack S

```
Stack S;

:

S.Push(3);
```

:

Aun cuando no existan argumentos, es necesario un par de paréntesis para distinguir la invocación de una función miembro del acceso a datos miembro. De esta forma, el siguiente código puede usarse para determinar si el stack S está o no vacío, antes de extraer un valor:

if (!S.Empty()) { ... }

else { ... }

Así como en otros lenguajes de programación orientados a objetos, en C++ hay una seudovariable asociada con todo método que mantenga el receptor del mensaje que invocó el método. Sin embrago, en C++ esta variable se denomina this y es un puntero al receptor, no el valor mismo del receptor. Así, la indirección del apuntador (−−>) debe usarse para enviar mensajes subsecuentes al receptor.

En el cuerpo de un método, la llamada de otro método sin el receptor se interpreta mediante el receptor actual.

El puntero this es accesible sólo dentro de funciones miembro de una clase, de una struct o una union. Apunta al objeto para el cual la función miembro fué llamada. Las funciones static no tienen este puntero.

Cuando se invoca una función miembro para un objeto, en forma oculta, se pasa a la función la dirección del objeto como parámetro.

Así, la invocación: S.Push(e)

se puede interpretar de la siguiente forma: Push(&S,e)

La dirección del objeto está disponible dentro de la función como el puntero this. Por lo tanto, aunque no es necesario, está permitido utilizar el puntero this cuando se referencie miembros de una clase.

La expresión (*this) se utiliza generalmente , para retornar el objeto actual desde una función miembro. Por ejemplo, las siguientes sentencias son equivalentes:

tope = −1;

this−>tope = −1;

(*this).tope = −1

3.2. CREACIÓN E INICIALIZACIÓN

*** Direccionamiento por stack versus direccionamiento por heap**

La problemática de la asignación de almacenamiento en stack versus heap se relaciona con la forma en que se reserva y libera el almacenamiento para las variables y con los pasos específicos que debe emprender el programador como parte de tales procesos. Se puede distinguir entre variables automáticas y dinámicas.

En C++, la asignación dinámica de memoria se logra generalmente mediante la operación new.

Stack *S;

...

S = new Stack;

...

delete S;

La diferencia esencial entre la asignación de almacenamiento basada en stacks y la basada en heaps es que el almacenamiento se asigna para un valor automático (residente en el stack), sin ninguna directiva explícita por parte del usuario, mientras que el espacio se asigna para un valor dinámico sólo cuando es requerido.

*** Recuperación de memoria**

Cuando se emplean técnicas de asignación basadas en heaps, se deben proporcionar algunos medios para recuperar memoria que ya no se esté usando. Lenguajes como Pascal, C, Object Pascal y C++, exigen la liberación explícita de los espacios. Otros lenguajes, como Lisp y Smalltalk, pueden detectar automáticamente los valores en desuso recolectando y reciclando los correspondientes espacios para asignaciones futuras. Este proceso se conoce como recolección de basura (garbage collection). En aquellos lenguajes en los cuales se requiere que el programador maneje el área dinámica de memoria, son frecuentes los siguientes errores:

− La memoria asignada no se libera (eliminación lógica ! garbage).

− Se intenta usar memoria que ya ha sido liberada (! dangling reference)

− Se libera en más de una oportunidad una misma localidad de memoria.

Para evitar esto, con frecuencia es necesario asegurar que cada localidad de memoria asignada dinámicamente tenga un propietario designado. El propietario de la memoria es responsable de asegurar que la localidad de memoria se use adecuadamente y se libere cuando ya no se necesita más. En grandes programas, las disputas sobre la propiedad de recursos compartidos puede ser fuente de dificultades.

*** Alcance y extensión de datos**

El alcance o ámbito de una variable es el rango de código en el cual esa variable es conocida. La extensión o vida se refiere al tiempo de ejecución durante el cual la variable tiene asignado un espacio en memoria para almacenar algún valor.

*** Punteros**

Los punteros o apuntadores son un medio eficiente y efectivo para manejar dinámicamente la información, razón por la cual se usan en casi todas las implementaciones de lenguajes OO. En algunos lenguajes (como Smalltalk y Object Pascal), los objetos se representan internamente como punteros, pero nunca son usados como tales por el programador. En otros lenguajes (como C++), se emplean con frecuencia los punteros explícitos.

*** Creación inmutable**

Una propiedad deseable para ciertas abstracciones de una clase es que los valores asociados a ciertos miembros de datos se establecieran cieran una vez y que no se alterasen de ahí en adelante. Variables de ejemplar, que no alteran sus valores en el curso de la ejecución, son conocidas como variables de asignación única o variables inmutables. Un objeto en el cual todas las variables de ejemplar son inmutables se conoce a su vez como objeto inmutable.

Se deben distinguir los valores inmutables de las constantes del programa aunque, en gran medida, la distinción está relacionada con un ámbito y tiempo de ligadura. En la mayoría de los lenguajes (p. e. Object Pascal), una constante debe conocerse en tiempo de compilación, tiene un ámbito global y permanece fija. Un valor inmutable es una variable que puede asignarse, pero sólo una vez. No se determina el valor sino hasta la ejecución, cuando se crea el objeto que contiene ese valor.

Los objetos inmutables pueden construirse siempre por convención. Por ejemplo, dar el mensaje para establecer el valor de variables miembro y simplemente confiar en que el cliente usará este recurso sólo una vez durante el proceso de inicialización de un nuevo objeto y no varias veces para alterar los valores de un objeto existente. Los creadores más cautelosos prefieren no depender de la buena voluntad de sus clientes y aplican mecanismos lingüísticos que aseguren un uso adecuado.

3.3. MECANISMOS PARA CREACIÓN E INICIALIZACIÓN

C++ sigue a C (Pascal y otros lenguajes de la familia ALGOL) en que tiene tanto variables automáticas como dinámicas. Un cambio con respecto a C establece que una declaración no necesita aparecer nada más antes del primer uso de la variable declarada.

```
...

Complejo a;

a.real = 3;

...

Complejo b;

...

b.imag = ...
```

Un ejemplo de acoplamiento flexible entre variables y valores en C++ es la introducción de las declaraciones de referencia. Una declaración de referencia, indicada por el símbolo & antes del nombre declarado, establece una variable de asociación única, esto es, una variable que se establece una vez y cuya referencia no puede cambiar a partir de ahí; genera un alias, es decir, liga a un nombre la dirección de otra variable.

```
void f(int &i)

{ i = 1;}

main()

{ int j = 0;

f(j);

printf(%d\n,j);

}
```

En **f**, el parámetro **i** tendrá la misma dirección que la variable **j**.

La inicialización implícita se facilita en C++ a través de los constructores de objetos. Un constructor es un método que tiene el mismo nombre que la clase objeto. El método se invoca siempre que se crea un objeto de la clase asociada. Esto sucede típicamente cuando se declara una variable, aunque también ocurrirá con objetos creados dinámicamente con el operador new.

```
class Complejo

{ public:

float real;

float imag;

private:

Complejo();

Complejo(float pr);

Complejo(float pr, float pi);

...

}
```

Esta clase contiene una función constructora (el método llamado Complejo) que asignará automáticamente el valor 0 a las partes real e imaginaria de un número complejo cuando se declare un identificador. Por medio de constructores, un programador puede asegurar que se asignen valores sólo durante el proceso de creación a campos inmutables de una clase

El mecanismo de constructores se hace considerablemente más poderoso cuando se combina con la capacidad de sobrecargar funciones en C++. Se dice que una función está sobrecargada cuando hay dos o más cuerpos de función conocidos por el mismo nombre. En C++ se obvia la ambigüedad de las funciones sobrecargadas por medio de las diferencias en las listas de parámetros. Se puede usar esta característica para proporcionar más de un estilo de inicialización. Por ejemplo, a veces se desearía inicializar un número complejo dando sólo la parte real y otras dando las partes real e imaginaria. A continuación se presenta esta funcionalidad mediante la definición de tres versiones diferentes para la función constructora, cuya selección dependerá de los argumentos proporcionados por el usuario como parte del proceso de creación.

```
Complejo::Complejo()

{ real = imag = 0; }

Complejo::Complejo(float pr)

{ real = pr; imag = 0.0; }

Complejo::Complejo(float pr, float pi)

{ real = pr; imag = pi; }
```

Si se desea usar la tercera forma del constructor, puede escribirse

```
Complejo *c;

...

c = new Complejo(3.14159,2.4);

...
```

La declaración de tres variables, cada una usando un constructor diferente, puede ser:

```
Complejo x, y(4.0), z(3.14159,2.4);
```

A menudo, los inicializadores por omisión pueden usarse para reducir la necesidad de sobrecarga. Por ejemplo, cada uno de los tres constructores antes definidos pueden sustituirse por un procedimiento único:

```
class Complejo

{ public:

float real;

float imag;

Complejo(float pr = 0.0, float pi = 0.0);

...

~Complejo();

};

Complejo::Complejo(float pr, float pi)

{ real = pr; imag = pi; }

Complejo::~Complejo()

{ printf(liberando complejo %g %g\n,real,imag); }
```

Una inicialización combina una definición de una variable con la asignación de un valor inicial, lo que generalmente garantiza que la variable tenga un valor significativo.

Un inicializador por omisión proporciona un valor que será usado en lugar de un argumento, en caso de que no lo proporcione el usuario. Así, si el usuario no da argumentos, los valores por omisión para los parámetros pr y pi serán 0; si el usuario proporciona un valor, el valor por omisión para pi seguirá siendo 0; y si el usuario da los dos valores, ninguno de los valores por omisión será usado. Notar que la inicialización de valores por omisión aparece en la declaración de clase, no en el cuerpo de la función subsecuente.

Los inicializadores por omisión operan con base en la posición; no es posible dar un valor al segundo argumento sin dar antes un valor al primer argumento. Aunque se necesita con menos frecuencia, también es posible definir en C++ una función, llamada destructor, que se invoque automáticamente siempre que se libere memoria de un objeto. Para variables automáticas se libera el espacio cuando se termina el procedimiento que

contiene la declaración de la variable. Para variables dinámicas, el espacio se libera cuando se aplica el operador delete. La función destructora se escribe como el nombre de la clase precedido por el símbolo ~. Esta función no acepta argumentos y rara vez se invoca directamente por el usuario. En el código anterior se usa un destructor que sólo imprime el valor del número complejo que se libera.

En C++, se puede asignar y liberar almacenamiento automático no sólo al entrar y salir de un procedimiento sino en cualquier secuencia de enunciados. Se puede aprovechar este hecho y combinarlo con características de los destructores para crear un mecanismo simple de rastreo del flujo de control de un programa durante su depuración.

```
class Lengua

{ char *token;

public:

Lengua(char *t);

"Lengua();

};

Lengua::Lengua(char *t)

{ token = t;

cout << "Lenguaje" << token << endl;

}

Lengua::"Lengua()

{ cout << "orientado a objetos es" << token << endl; }

void main()

{ Lengua bb("Object Pascal");

cout << "es compilado." << endl;

{ Lengua cc("Pascal");

cout << "también lo es. Pero no" << endl;

}

cout << "En cambio sí" << endl;

}
```

La clase Lengua define un constructor que, como efecto lateral, imprime un mensaje en la salida estándar. El destructor asociado también imprime un mensaje. Se puede rodear cualquier sección de código que se quiera

investigar con una secuencia de enunciados en la cual se declare un ejemplar de la clase Lengua.

Los enunciados que se muestran en el cuerpo del void main() producirán la siguiente salida:

Lenguaje Object Pascal

es compilado.

Lenguaje Pascal

también lo es. Pero no

orientado a objetos es Pascal

En cambio sí

orientado a objetos es Object Pascal

4. HERENCIA

4.1. DESCUBRIMIENTO DE LA HERENCIA

Una vez que se ha establecido el diseño inicial, el reconocimiento de partes comunes puede facilitar el posterior desarrollo. Dicho de otro modo, la herencia es una técnica de implementación, no una técnica de diseño.

Existen dos relaciones de suma importancia en esta segunda etapa del diseño OO; éstas se conocen informalmente como la relación es−un y la relación tiene−un. Se da la relación es−un entre dos conceptos cuando el primero es un ejemplar especializado del segundo. Sea la afirmación X es un Y, donde X e Y son conceptos que se están examinando. Si la afirmación es correcta (si concuerda con nuestro conocimiento), entonces se concluye que X e Y guardan una relación es−un.

Por otra parte, la relación tiene−un se da cuando el segundo concepto es un componente del primero. Este mecanismo se denomina composición.

Ejemplo gráfico de relaciones es−un y tiene−un:

La relación es−un define jerarquías de clase−subclase, mientras que la relación tiene−un describe datos que se deben mantener dentro de una clase. Al buscar código que pueda ser reutilizado o generalizado, se enfatiza el reconocimiento de estas dos relaciones. Encontrar clases existentes que puedan ser reutilizadas es un enfoque más o menos top−down (en el sentido de la jerarquía de clases). Un enfoque bottom−up intenta encontrar generalidad en la descripción de las clases, promoviendo las características comunes hacia una nueva superclase abstracta.

Seguidamente, se pueden examinar las clases individuales para ver si su comportamiento puede dividirse en componentes con utilidad independiente.

Por último, se examinan las relaciones tiene−un para ver si los valores que se mantienen pueden ser representados mediante herramientas existentes.

```
class Cuadrado

{private:

double s;

public:

Cuadrado(double);

double Perimetro();

double Area();

}

//Implementación

Cuadrado::Cuadrado(double r)

{s=r;}

double Cuadrado::Perimetro()

{return 4*s;}

double Cuadrado::Area()

{return s*s;}
```

Como ambas clases, Cuadrado y Triángulo heredan miembros de datos y métodos (o funciones miembros) de la clase base, se puede reescribir la declarativa, en la cual la clase base incluye información que es general a los polígonos regulares, como ser cantidad de lados y su longitud.

4.2. EL CONCEPTO

Para aprender a programar de una manera OO deberemos adquirir conocimientos sobre cómo usar con efectividad clases organizadas bajo una estructura jerárquica basada en el concepto de herencia. Se entiende por herencia a la propiedad que dota a los ejemplares de una clase hija (o subclase) del acceso tanto a los datos como al comportamiento (métodos) de una clase paterna (superclase o clase base). La herencia es siempre transitiva, por lo que una clase puede heredar características de superclases alejadas muchos niveles. Esto es, si la clase Perro es una subclase de la clase Mamífero y ésta, a su vez, una subclase de la clase Animal, entonces Perro heredará atributos tanto de Mamífero como de Animal.

Regresemos a Perico, el pajarero del capítulo 1. Existe cierto comportamiento que se espera sigan los pajareros, no porque sean pajareros, sino simplemente porque son comerciantes. Por ejemplo, se espera que Perico pida dinero por la transacción y, a su vez, se espera que entregue un recibo. Como los pajareros son una subclase de la categoría general Comerciante, el comportamiento asociado a ésta se relaciona automáticamente con la subclase.

La herencia significa que el comportamiento y los datos asociados con las clases hijas son siempre una extensión (es decir, un conjunto estrictamente más grande) de las propiedades asociadas con las clases

paternas. Una subclase debe reunir todas las propiedades de la clase paterna y otras más. Por otra parte, ya que una clase hija es una forma más especializada (o restringida) de la clase paterna, en cierto sentido es también, una contracción del tipo paterno. Semejante contraposición entre herencia como expansión y herencia como contracción es fuente de la fuerza inherente a la técnica pero, al mismo tiempo, origina mucha confusión en cuanto a su uso. Se emplearán estas visiones opuestas en el momento en que se establezcan heurísticas para saber cuando usar herencia y clases.

Esta herencia de atributos se extiende tanto a los datos (variables de ejemplar y campos) como al comportamiento (métodos) en los lenguajes OO. Se puede establecer una definición precisa de herencia en términos del ámbito: algo que esté de acuerdo con el ámbito de un identificador de componente (variable de ejemplar o campo) y que abarque el dominio de la clase descendiente. Sin embargo, como los detalles exactos difieren de un lenguaje a otro, bastará aceptar la noción intuitiva de herencia.

4.3. LOS BENEFICIOS DE LA HERENCIA

*** Reusabilidad del software**

Cuando se hereda comportamiento de otra clase, no se necesita reescribir el código que lo proporciona. Esto significa evitar escribir código que ya se ha escrito.

Otros beneficios del código reutilizado incluyen una mayor confiabilidad (a medida que se reutiliza se puede ir mejorando) y un menor costo de mantención gracias a la compartición del código por varios usuarios.

*** Compartición de código**

La compartición de código puede ocurrir en diversos niveles cuando se usan técnicas OO. En un nivel, muchos usuarios o proyectos independientes pueden emplear las mismas clases. Se puede hacer referencia a éstas como componentes de software. Otra forma de compartición ocurre cuando dos o más clases diferentes, desarrolladas por un solo programador como parte de un proyecto, heredan de una clase paterna única. Por ejemplo, un Conjunto y un Arreglo pueden ser considerados como una forma de colección; cuando esto sucede, significa que dos o más tipos diferentes de objetos compartirán el código que heredan. Dicho código se necesita escribir sólo una vez y sólo una vez contribuirá al tamaño del programa resultante.

*** Consistencia de la interfaz**

Cuando múltiples clases heredan de la misma superclase se asegura que el comportamiento que heredan será el mismo en todos los casos. De esta manera, es más fácil garantizar que las interfaces para objetos similares sean en efecto similares y al usuario no se le presenta una colección confusa de objetos que sean casi lo mismo pero que se comporten e interactúen en forma muy diferente.

*** Modelado rápido de prototipos**

Cuando un sistema de software puede construirse en su mayor parte con componentes reutilizables, el tiempo de desarrollo puede concentrarse en entender la parte del sistema que es nueva. Así, se pueden generar más rápida y fácilmente los sistemas de software mediante la técnica de programación conocida como modelado rápido de prototipos o programación explosiva: Se desarrolla un sistema prototipo, los usuarios experimentan con él, se produce un segundo sistema basado en la experiencia con el primero, se efectúa una exploración adicional y se repite el proceso. Esta técnica es útil cuando los objetivos y requerimientos del sistema se entienden vagamente al momento de comenzar el proyecto.

*** Polimorfismo**

El software producido en forma convencional se escribe, generalmente, de manera bottom−up, aunque puede diseñarse de manera top−down. Es decir, se desarrollan las rutinas de bajo nivel y sobre éstas se producen abstracciones de un nivel un poco más alto y sobre éstas se generan elementos aún más abstractos, y así sucesivamente (ladrillo, muro, habitación, departamento, edificio, condominio, ...).

Regularmente, la transportabilidad decrece en la medida en que uno asciende en los niveles de abstracción. Esto es, las rutinas de más bajo nivel pueden usarse en diferentes proyectos y tal vez pueda reutilizarse incluso el siguiente nivel de abstracción, pero las rutinas de nivel más alto están ligadas íntimamente a una aplicación particular. Las piezas de nivel más bajo pueden llevarse a un nuevo sistema y, por lo general, tienen sentido por sí mismas.

El polimorfismo de los lenguajes de programación permite al programador generar componentes reutilizables de alto nivel que pueden adaptarse para que se ajusten a diferentes aplicaciones mediante el cambio de sus partes de bajo nivel. Considérese el siguiente algoritmo:

```
función búsqueda(x : clave) : boolean;

var pos : posición;

comienzo

pos := posicInicial(x);

mientras no vacío(pos,x)

si encontrado(pos,x) entonces

devuelve verdadero

o bien

pos := sigPosic(pos,x);

devuelve falso;

fin;
```

Con una sustitución adecuada de las funciones posicInicial, vacío, encontrado y sigPosic este algoritmo puede representar una búsqueda lineal, binaria, de hashing, en un abb y más. El algoritmo es de alto nivel y se puede especializar reemplazando las funciones de bajo nivel sobre las cuales está construido.

*** Ocultación de la información**

Cuando un programador reutiliza un componente de software sólo necesita entender la naturaleza del componente y su interfaz. No es necesario conocer aspectos de implementación. Así se reduce la interconexión entre los sistemas de software.

4.4. EL COSTO DE LA HERENCIA

*** Velocidad de ejecución**

Los métodos heredados, creados para tratar con subclases arbitrarias, son a menudo más lentos que el código

especializado. Sin embargo, la reducción en la velocidad de ejecución puede compensarse con el aumento en la velocidad de desarrollo del software.

*** Tamaño del programa**

Generalmente, el uso de cualquier biblioteca de software impone una sanción al tamaño del código. Sin embargo, nuevamente es más relevante producir con rapidez código de alta calidad que limitar el tamaño de los programas.

*** Tiempo adicional**

Una excesiva preocupación por el costo adicional en tiempo del paso de mensajes, con respecto a la simple invocación de procedimientos, es un aspecto que atiende el detalle y descuida el conjunto. El tiempo de procesamiento adicional del paso de mensajes será mucho mayor en lenguajes ligados dinámicamente, como Smalltalk, y mucho menor en lenguajes ligados estáticamente, como C++. También, en este caso, el incremento en costo debe evaluarse junto con los muchos beneficios de la técnica OO.

*** Complejidad**

Frecuentemente, el abuso de la herencia sustituye una forma de complejidad por otra. Entender el flujo de control de un programa que usa herencia puede requerir múltiples exploraciones hacia arriba y hacia abajo en la gráfica de herencia.

4.5. HEURÍSTICA PARA CREAR SUBCLASES

Así como muchos programadores novatos en OO se sienten inseguros en cuanto a como reconocer las clases en la descripción de un problema, muchos también dudan sobre cuándo una clase debe convertirse en subclase de otra o cuándo es más apropiado otro mecanismo.

La regla más importante es que, para que una clase se relacione con otra por medio de la herencia, debe haber una relación de funcionalidad entre ambas.

Recordando la regla es−un, al decir que una subclase es una superclase, se está diciendo que la funcionalidad y los datos asociados con la subclase forman un superconjunto de la funcionalidad y los datos asociados con la superclase. Sin embargo, dentro de este marco general hay lugar para muchas variaciones.

*** Especialización**

El uso más común de la herencia y la subclasificación es en la especialización. Las instancias de las subclases tienen alguna propiedad que NO poseen las instancias de la clase base. Esto implicará:

- Redefinir algún método de la clase base para adaptarse a la nueva clase.
- La subclase incorpore un nuevo campo.
- Ambas

La subclasificación para la especialización es el uso más obvio y directo de la regla es−un. Si se consideran dos conceptos abstractos A y B, y tiene sentido el enunciado A es un B, entonces es probablemente correcto hacer A una subclase de B (p. e., un triángulo es un polígono).

Por otra parte, si es más evidente el enunciado A tiene un B, entonces probablemente sea una mejor decisión de diseño hacer que los objetos de la clase B sean atributos (campos o variables de ejemplar) de la clase A (p. e., un automóvil tiene un motor).

Otra pregunta útil es determinar si se desea contar con subclases, o ejemplares de una clase, para tener acceso a datos o comportamiento de una posible superclase. Por ejemplo, considérese un tipo de datos Stack, implementado mediante un arreglo de tamaño fijo. En este caso, el arreglo es simplemente un área útil de almacenamiento para el stack y, tal vez, exista poca funcionalidad que la clase stack pueda heredar de la clase Arreglo. Luego, usar herencia sería un error en este caso, siendo una mejor alternativa mantener el arreglo como un campo (o variable de ejemplar) dentro del stack. Esto es, decir que un Stack es−un Arreglo es válido sólo desde el punto de vista del implementador, no desde el punto de vista del usuario. Ya que el usuario no necesita usar nada de la funcionalidad de la clase Arreglo cuando usa un Stack, la subclasificación sería un error.

*** Especificación**

Otro uso frecuente de la herencia es garantizar que las clases mantengan cierta interfaz común; esto es, que implementen los mismos métodos, o sea, la clase base proporciona una especificación pero NO la implementación. El propósito de la clase base es definir qué se debe hacer pero NO cómo debe realizarse. La subclase redefine los métodos. La clase paterna puede ser una combinación de operaciones implementadas y de operaciones que se delegan a las clases hijas. A menudo no hay cambio de interfaz entre el supertipo y el subtipo; la hija incorpora el comportamiento descrito (aunque no implementado) en la clase paterna. Por ejemplo, la clase Polígono deberá indicar que todos los polígonos regulares deben tener área o pueden ser dibujados , pero cada subclase debe implementar su propio método para determinar el área o para dibujar

Se trata en realidad de un caso especial de subclasificación por especialización, excepto que las subclases no son refinamientos de un tipo existente, sino más bien relaciones de una especificación abstracta incompleta. Tal clase es conocida a veces como una clase de especificación.

La subclasificación por especificación puede reconocerse cuando la superclase no implementa ningún comportamiento real, sino que nada más describe el comportamiento que será implementado en las subclases. Esta forma de subclasificación es común en lenguajes que hacen uso extensivo de funciones virtuales.

*** Construcción**

Muchas veces una clase puede heredar casi toda la funcionalidad de una superclase, tal vez con sólo cambiar los nombres de los métodos usados para relacionar la clase o al modificar los argumentos de cierta manera.

*** Generalización**

Usar herencia para crear subtipos para generalización es, en cierto sentido, lo opuesto a la creación de subtipos para especialización. Aquí, un subtipo extiende el comportamiento del supertipo para crear una especie de objeto. Por ejemplo, considérese un sistema de visualización gráfica en el cual se ha definido la clase Ventana para mostrar en un simple fondo blanco y negro. Se puede crear un subtipo VentanaAColor que permita elegir cualquier color para el fondo añadiendo un campo adicional para guardar el color y anulando el código de visualización de ventana heredado.

Como regla general, se debe evitar la creación de subtipos por generalización optando mejor por invertir la jerarquía del tipo y usar la creación de subtipos por especialización, aunque no siempre es posible hacerlo.

*** Extensión**

Mientras que la subclasificación por generalización modifica o expande la funcionalidad existente en un objeto, la subclasificación por extensión agrega capacidades totalmente nuevas.

La subclasificación por extensión puede distinguirse de la subclasificación por generalización en que esta

última debe anular al menos un método del padre y la funcionalidad unida a éste, mientras que la extensión tan sólo añade nuevos métodos a los del padre y la funcionalidad queda menos fuertemente unida a los métodos existentes en éste.

*** Limitación**

Subclasificación por limitación es una forma especializada de la subclasificación por especificación, en la que el comportamiento de la subclase es más reducido o está más restringido que el comportamiento de la superclase.

Por ejemplo, suponer que una biblioteca provee la clase Deque (Double−ended−queue). Sin embargo, el programador desea una clase Stack que refuerce la propiedad de que los elementos pueden ser insertados o retirados sólo por un extremo del stack. De manera similar a la subclasificación por construcción, el programador puede hacer de la clase stack una subclase de la clase existente y puede modificar o anular los métodos no deseados de tal manera que produzcan un mensaje de error si se les usa.

La subclasificación de subtipos por limitación constituye una violación explícita del enfoque es−un de la herencia. Por tal razón, esta técnica debe evitarse siempre que sea posible.

*** Variación**

La creación de subtipos por variación es útil cuando dos o más clases tienen implementaciones similares, pero no parece haber ninguna relación jerárquica entre los conceptos representados por las clases. Por ejemplo, el código necesario para controlar un ratón puede ser casi idéntico al código necesario para controlar una tarjeta gráfica. Sin embargo, no existe razón alguna por la que cualquier clase Ratón deba ser una subclase de la clase Tarjeta o viceversa. Se puede seleccionar arbitrariamente una de las dos clases como padre, haciendo que el código común sea heredado por el otro y el código específico del dispositivo sea proporcionado como métodos anulados.

Sin embargo, una mejor opción es descomponer en factores el código común de una clase abstracta Dispositivo y hacer que ambas clases hereden de este ancestro común. También esta alternativa podría no estar disponible si se está construyendo sobre una base de clases existentes.

*** Combinación**

Una situación común ocurre cuando una subclase representa una combinación de características de dos o más clases paternas. Por ejemplo, un Ayudante es tanto un Profesor como un Alumno y, lógicamente, se debería comportar como ambos. La capacidad de una clase para heredar de dos o más clases paternas se conoce como herencia múltiple y, por desgracia, no se proporciona en todos los lenguajes OO.

4.6. ÁRBOL VERSUS BOSQUE

Existen dos enfoques alternativos en cuanto a cómo deben estructurarse las clases como un todo en los lenguajes OO.

Enfoque 1 (Smalltalk y Objective−C): Todas las clases deben estar contenidas en una sola gran estructura de herencia.

La gran ventaja de este enfoque es que la funcionalidad proporcionada en la raíz del árbol (en la clase Object) es heredada por todos los objetos. De esta manera, cada objeto responderá al mensaje imprime que muestra la representación imprimible del objeto pues este método se encuentra en la superclase abstracta Object. En un lenguaje así, toda clase definida por el usuario debe ser una subclase de alguna ya existente que, por lo

general, es la clase Object.

Enfoque 2 (C++ y Object Pascal) : Las clases que no están lógicamente relacionadas deben ser por completo distintas. El resultado es que el usuario se enfrenta a un bosque compuesto por diversos árboles de herencia. Una ventaja de este enfoque es que la aplicación no está obligada a cargar con una gran biblioteca de clases, de las cuales podrían usarse sólo unas pocas. La desventaja es que no puede garantizarse que todos los objetos posean una funcionalidad definible por el programador.

Object

Boolean Magnitude Collection

True False Char Number Point Set KeyedCollection

Integer Float Dictionary SequenceableCollection

ArrayedCollection

Array String

En parte, los enfoques discordantes de objetos parecen depender de la distinción entre lenguajes que usan asignación dinámica de tipos y los que usan asignación estática de tipos. En los lenguajes dinámicos, los objetos se caracterizan principalmente por los mensajes que entienden. Si dos objetos entienden el mismo conjunto de mensajes y reaccionan de la misma manera, para todo propósito práctico son indistinguibles, independientemente de las relaciones de sus clases respectivas. En estas circunstancias es útil hacer que todos los objetos hereden gran parte de su comportamiento de una clase base común.

5. ENLACES ESTÁTICO Y DINÁMICO

5.1. TIEMPO DE ENLACE

Tiempo de enlace es simplemente el tiempo en que se determina la naturaleza exacta de un atributo para una entidad en un programa. Los tiempos de enlace abarcan los tiempos de compilación, ligado, carga y ejecución. Aun durante ejecución es posible distinguir entre características que son enlazadas cuando empieza un programa, las que lo son cuando se inicia un subprograma y aquellas que no se enlazan sino hasta que se ejecuta la instrucción que contiene a la característica.

5.2. ASIGNACIÓN ESTÁTICA Y DINÁMICA DE TIPOS

*** Fundamento**

Para entender la diferencia entre asignación estática y dinámica de tipos, previamente se debe recordar los elementos que conforman una variable, a saber, nombre, tipo, referencia y valor. El término valor describe el contenido actual de la memoria del computador asociado a una variable. Dependiendo del lenguaje, un tipo se puede asociar con una variable (nombre) o con un valor.

En lenguajes con asignación estática de tipos, como Object Pascal o C++, los tipos se asocian con las variables, es decir, se asocian a un nombre mediante una declaración explícita.

En tiempo de ejecución, una variable computacional es tan sólo una caja de bits. Es el tipo estático de la variable lo que se usa para darle significado a los bits. Algunas veces este significado puede ser incorrecto (p. e., al usar registros con variantes en Pascal). Los lenguajes en los cuales se puede garantizar que en tiempo de

compilación todas las expresiones tienen el tipo correcto se llaman lenguajes con asignación fuerte de tipos.

Los lenguajes OO introducen una nueva particularidad en el problema de acoplar nombres y tipos. Si las clases se ven como tipos, entonces una jerarquía de clases es una jerarquía de tipos. La relación es−un afirma que cualquier ejemplar de una clase hija es un representante de la clase paterna; por ejemplo, los ejemplares de la clase Pajarero son ejemplares de la clase Comerciante. Así, un valor de tipo Pajarero, como Perico, puede asignarse a una variable declarada de tipo Comerciante.

En forma más precisa, cualquier valor de una clase específica puede asignarse a una variable declarada de esa clase o de cualquier superclase de la clase. La situación inversa, en la que las variables declaradas como ejemplares de una superclase se asignan a variables de una subclase, está relacionada con lo que se llama el problema del contenedor. En situaciones en las que el tipo (clase) de un valor no concuerda con la clase de la variable a la que pertenece el valor, la clase de la variable se conoce como la clase estática y la clase del valor se conoce como la clase dinámica.

En los lenguajes con asignación dinámica de tipos, los tipos no se enlazan a los nombres sino más bien a los valores. Informalmente se puede pensar en la distinción como la diferencia entre ser conocido por asociación y ser conocido por cualidades específicas. Por ejemplo, en los lenguajes con asignación dinámica de tipos no se puede afirmar que una variable sea un entero, sino sólo que actualmente contiene un valor entero. Así, no es posible asignar a una variable un valor ilegal, ya que todos los valores que representan objetos legítimos son legales. En vez de que un valor sea nada más una caja de bits sin interpretar, cada valor debe llevar consigo alguna identificación, algún conocimiento que indique su naturaleza exacta. Esta etiqueta de identificación queda ligada al valor y se transfiere por asignación. Con el paso del tiempo, a un solo nombre se le pueden asignar valores de muchos tipos diferentes.

*** El problema del contenedor**

Suponer que se define una clase Conejo y dos subclases, ConejoNegro y ConejoBlanco. A continuación, se dejan caer un ConejoNegro y un ConejoBlanco en una caja y luego se escoge aleatoriamente uno de ellos. Sin duda, el objeto resultante se puede considerar un Conejo, pudiéndose asignar a una variable declarada de ese tipo. Pero, ¿es un ConejoNegro? ¿Puede ser asignado a una variable declarada como clase ConejoNegro?

En un lenguaje con asignación dinámica de tipos, es el mismo conejo el que determina si es un ConejoNegro o un ConejoBlanco. El conejo lleva en él la información necesaria para conocer su clase (color). Luego, por ejemplo, se puede determinar el color del conejo con sólo preguntar a qué clase pertenece.

En un lenguaje con asignación estática de tipos, el conejo mismo no sabe de qué color es, es decir, el valor del conejo puede que no sepa a qué clase pertenece. De esta manera, no es posible determinar si está permitido asignar el valor del resultado a una variable declarada del tipo ConejoNegro.

El problema subyacente a este ejemplo es bastante común. Considérese el desarrollo de estructuras de datos muy usadas, como conjuntos, stacks, diccionarios, tablas y otros. Tales estructuras se usan para mantener colecciones de objetos. Un beneficio valioso de la POO es la producción de componentes de software reutilizable. Los contenedores de colecciones serían, indudablemente, candidatos para tales componentes.

Sin embargo, un contenedor de colecciones es, en cierto modo, exactamente la caja en que se depositan los conejos del ejemplo citado. Si un programador coloca dos objetos diferentes en un conjunto y más tarde saca uno ¿cómo sabe qué tipo de objeto obtendrá?

5.3. ENLACES ESTÁTICO Y DINÁMICO DE MÉTODOS

*** Asociación de métodos y mensajes**

Relacionado con el problema de cómo un lenguaje asocia tipos con nombres está la cuestión de cómo un método se asocia con un mensaje.

Considérese, como ejemplo, el siguiente juego: Pedro sostiene un gato en sus brazos, pero lo oculta a la vista de María. Pedro le dice a María que está cargando un mamífero pequeño y le pregunta si éste tiene o no pelo. María podría razonar de la siguiente manera: No importa qué tipo de animal (qué clase de objeto) sea en realidad, pues sé que es una subclase de Mamífero. Todos los mamíferos tienen pelo. Por lo tanto, el animal que Pedro carga tiene pelo.

Ahora, suponer que Pedro carga un ornitorrinco y pregunta a María si los ejemplares hembras de la clase de animal que está cargando dan a luz crías vivas. Si María razona como antes dirá que sí. Sin embargo, estará equivocada ya que en este caso el comportamiento común a todos los mamíferos es anulado por la clase Ornitorrinco.

Computacionalmente, se puede imaginar una variable declarada de la clase Mamífero que tiene un valor del tipo Ornitorrinco. ¿La búsqueda de un enlace apropiado para un método debe basarse en la clase estática de la declaración (es decir, Mamífero) o en la clase dinámica del valor (esto es, Ornitorrinco)?

Al primero se le conoce como enlace estático, ya que el enlace del método al mensaje se basa en las características estáticas de la variable. La segunda opción, en la que la definición de un método se basa en el tipo del valor, y no en el tipo de la declaración, se conoce como enlace dinámico.

Aun cuando el enlace dinámico se usa para acoplar un método con un mensaje, no es raro para los lenguajes OO que usan asignación estática de tipos, decidir si una expresión de paso de mensajes es válida con base en la clase estática del receptor. Es decir, si p se declara como un ejemplar de la clase Mamífero, no es válido pasar a p el mensaje ladra a menos que este mensaje sea entendido por todos los ejemplares de la clase Mamífero. Aun cuando se sabe que en tiempo de ejecución p siempre tendrá de hecho un valor de la clase Perro, el compilador no retornará el mensaje a menos que sea definido en la clase paterna.

*** Méritos del enlace estático versus el enlace dinámico**

Comparativamente, y en forma concisa, la asignación estática de tipos y el enlace estático son más eficientes; en tanto que, la asignación dinámica de tipos y el enlace dinámico son más flexibles.

Según el problema del contenedor, la asignación dinámica de tipos implica que cada objeto lleva el control de su propio tipo. Si los objetos se conciben como individualistas rigurosos (esto es, cada objeto deba cuidarse a sí mismo y no depender de otros para su mantenimiento), entonces la asignación dinámica de tipos parecería ser la técnica más orientada a objetos. Es claro que la asignación dinámica de tipos simplifica en forma importante problemas como el desarrollo de estructuras de datos de propósito general, pero el costo involucrado es la búsqueda durante ejecución para descubrir el significado (es decir, el código a ejecutar) cada vez que se usa una opción en un valor de datos. Esta es una causal importante en el uso de la asignación estática de tipos por la mayoría de los lenguajes.

La asignación estática de tipos simplifica la implementación de un lenguaje de programación, aun si (como en Object Pascal y C++ en algunos casos) se usa enlace dinámico de métodos con mensajes. Cuando los tipos estáticos son conocidos para el compilador, el almacenamiento puede asignarse eficientemente para las variables, además de generar código eficiente para operaciones ambiguas (como +).

Si la asignación estática de tipos simplifica una implementación, el enlace estático de métodos la simplifica aún más. Si el compilador puede descubrir un acoplamiento entre método y mensaje, entonces el paso de mensajes (sin importar la sintaxis usada) puede ser implementado como una simple llamada a un procedimiento, sin que necesite mecanismos en tiempo de ejecución para llevar a cabo la búsqueda del

método. Esto es casi lo más eficiente que razonablemente se puede esperar (aunque las funciones en línea de C++ aun pueden eliminar el tiempo de procesamiento adicional de la llamada a procedimiento).

Por otra parte, el enlace dinámico de métodos siempre requiere la ejecución de cuando menos algún mecanismo en tiempo de ejecución, aunque sea primitivo, para acoplar un método a un mensaje. En general, en lenguajes que usan asignación dinámica de tipos (y por ello deben usar enlace dinámico de mensajes a métodos), no es posible determinar si el receptor entenderá una expresión de paso de mensajes. Cuando el receptor no entiende un mensaje, no queda otra alternativa que generar un mensaje de error en tiempo de ejecución. Un punto importante a favor de los lenguajes que usan enlace estático de métodos es el hecho de que muchos errores pueden aparecer en tiempo de compilación.

Otro argumento a favor del uso de la asignación estática de tipos y el enlace estático, en lugar de sus correspondientes dinámicos, implica la detección de errores. Tales argumentos reflejan la polémica en torno a la asignación fuerte versus débil de tipos en lenguajes de programación más convencionales. Cuando se usan tipos estáticos, en tiempo de compilación se pueden detectar errores como la asignación de tipos incompatibles o valores no permitidos como argumentos de funciones. Cuando se usan valores dinámicos tales errores pasan con frecuencia inadvertidos hasta que se produce un desenlace catastrófico durante la ejecución del programa.

Así, el usuario debe decidir qué es más importante: ¿eficiencia o flexibilidad? ¿corrección o facilidad de uso?. Brad Cox comenta que las respuestas a tales preguntas dependen de qué nivel de abstracción representa el software y de si se adopta el punto de vista del productor o del consumidor del sistema de software. Cox afirma que la POO será la herramienta primaria (aunque no la única) en la revolución industrial del software.

La eficiencia es la principal preocupación en el nivel más bajo de construcción. Conforme se eleva el nivel de abstracción, se hace más importante la flexibilidad.

En forma similar, las prestaciones y la eficiencia son con frecuencia preocupaciones esenciales para el desarrollador de abstracciones de software. Para un consumidor interesado en combinar sistemas de software desarrollados independientemente en formas nuevas e ingeniosas, la flexibilidad puede ser mucho más importante.

Por lo tanto, no hay una sola respuesta correcta a la pregunta de qué técnicas de enlace son más apropiadas en un lenguaje de programación. Existe una variedad de esquemas y cada técnica es útil en alguna forma.

5.4. ENLACE EN C++

Dos de los objetivos primarios en el diseño de C++ fueron la eficiencia en tiempo y espacio. El lenguaje se ideó como una mejora de C para aplicaciones tanto OO como no OO. Un principio básico subyacente en C++ es que ninguna característica debe suponer un costo (ya sea en espacio o en tiempo de ejecución) a menos que se use. Por ejemplo, si se pasaran por alto las características de C++ OO, el resto del lenguaje se debería ejecutar tan rápido como C convencional. Así, no es de sorprender que la mayoría de las características de C++ estén estática, más que dinámicamente, ligadas. Las variables se declaran estáticamente; sin embargo, se les puede asignar valores derivados de subclases.

En C++ no existe ningún recurso que se pueda usar para determinar la clase dinámica de un objeto. Ello se debe a la conservación de la filosofía de que se debe incurrir en costos sólo cuando sea necesario. Semejante limitación haría más difícil resolver el problema de la clase contenedora si no fuera por la gran flexibilidad que C++ proporciona en cuanto a sobrecarga y anulación de métodos.

C++ proporciona tanto enlace estático como enlace dinámico de métodos y reglas complejas para determinar qué forma se usa. En parte, el enlace dinámico depende del uso de la palabra clave virtual en una declaración

de método. Sin embargo, aun cuando se use la palabra clave virtual, puede ser que se use enlace estático a menos que el receptor se use como puntero o referencia.

Incluso cuando el enlace dinámico se utilice para acoplar métodos y mensajes, se prueba la validez de cualquier expresión particular de paso de mensajes, como en Object Pascal, basándose en el tipo estático del receptor.

6. REEMPLAZO Y REFINAMIENTO

6.1. GENERALIDADES SOBRE LA HERENCIA

Hasta ahora, el modelo intuitivo de herencia presentado ha sido aquel en que el conjunto de valores de datos y métodos asociados con una clase hija es siempre estrictamente más grande que la información asociada a la clase paterna. Esto es, una subclase simplemente agrega siempre nuevos datos o métodos a los proporcionados por la superclase. Sin embargo, está claro que este modelo no es general. Basta recordar el ejemplo acerca de la clase Mamífero y la subclase Ornitorrinco. Se puede decir que esta última hace algo más que agregar al conjunto de información conocida sobre la clase Mamífero. Por el contrario, la clase hija cambia o altera de verdad una propiedad de la clase paterna.

6.2. AÑADIR, REEMPLAZAR Y REFINAR

Hasta el momento se ha supuesto que los datos y métodos añadidos por una subclase a los heredados de la clase paterna son siempre distintos. Esto es, el conjunto de métodos y valores de datos definidos por la clase hija es disjunto del conjunto de valores y métodos definidos por las clases paternas (y antecesoras). Para definir la situación, se dice que tales métodos y valores de datos se agregan al protocolo de la clase paterna.

Una situación diferente ocurre cuando una clase hija define un método con el mismo nombre que el usado en la clase paterna. El método en la clase hija efectivamente oculta, o anula, el método de la clase paterna. Cuando se busca un método para usarlo en una situación dada, se encontrará y usará el método de la clase hija. Esto es análogo a la forma en la cual el conocimiento específico de la clase Ornitorrinco anula el conocimiento general de la clase Mamífero.

Recordar que, cuando se envía un mensaje a un objeto, la búsqueda del método correspondiente siempre empieza con el examen de los métodos asociados con el objeto. Si no se encuentra ningún método, se examinan los métodos asociados con la superclase inmediata del objeto. Si, una vez más, no se encuentra ningún método, se examina la superclase inmediata de esa clase, y así sucesivamente, hasta que ya no quedan más clases (en cuyo caso se indica un error) o se encuentra un método apropiado.

Se dice que un método de una clase que tiene el mismo nombre que el método de una superclase anula el método de la clase paterna. Durante el proceso de búsqueda de un método por invocar en respuesta a un mensaje, se encontrará naturalmente el método de la clase hija antes que el método de la clase paterna.

Un método definido en la clase hija puede anular un método heredado por uno de dos propósitos: Un reemplazo de método sustituye totalmente el método de la clase paterna durante la ejecución, es decir, el código en la clase paterna nunca será ejecutado cuando se manipulan los ejemplares de una clase hija. El otro tipo de anulación es un refinamiento de método, el cual incluye, como parte de su comportamiento, la ejecución del método heredado de la clase paterna; de esta manera, se preserva y se aumenta el comportamiento del padre.

6.3. REEMPLAZO

En Smalltalk, los números enteros y reales son objetos; hay ejemplares de la clase Integer y Float,

respectivamente. Ambas clases son, a su vez, subclases de una clase más general Number. Suponer que se tiene una variable x que actualmente contiene un entero de Smalltalk y que se envía a x el mensaje sqrt. No existe ningún método que corresponda a este nombre en la clase Integer, así que se busca en la clase Number, en donde se encuentra el método que pasa el mensaje asFloat a self, el cual representa al receptor del mensaje sqrt. El mensaje asFloat da como resultado un valor real de la misma magnitud que el número entero. El mensaje sqrt se pasa entonces a ese valor. Esta vez, la búsqueda de un método empieza con la clase Float. A veces sucede que la clase Float contiene un método diferente llamado sqrt, el cual anula el método de la clase Number para los valores reales.

La capacidad para anular el método sqrt significa que los números que no sean ejemplares de la clase Float pueden compartir la única rutina por omisión que existe en la clase Number. Este comportamiento evita tener que repetir el código por cada una de las diferentes subclases de Number. Las clases que requieren un comportamiento diferente al dado por omisión pueden simplemente anular el método y sustituir el código alternativo.

Una de las principales dificultades de la anulación, tanto para el reemplazo como para el refinamiento, es conservar las características exactas de la relación es−un. Esto es, al anular un método, generalmente no se tiene garantía alguna de que el comportamiento exhibido por la clase hija tendrá alguna relación con el de la clase paterna.

La anulación en C++ se complica por el entrelazamiento de los conceptos de anulación, sobrecarga, funciones virtuales (o polimórficas) y constructores. Sobrecarga y funciones virtuales se verán más adelante.

Los reemplazos simples ocurren cuando los argumentos de la clase hija se acoplan en forma idéntica al tipo y número de la clase paterna y el modificador virtual se usa en la declaración del método de la misma.

Una operación importante con un polígono regular, será determinar el área de una determinada figura. La subclase Cuadrado tiene una implementación diferente para el método Area(), sin embargo, la subclase Triángulo utiliza la general definida en la clase base Polígono. En la clase Polígono este método debe declararse como virtual, el cual puede tener una implementación que será ejecutada por defecto, si una de sus clases derivadas no la redefinió, o bien, puramente virtual, asignando un 0 en la definición en la clase base, lo que obligará a que las subclases deben definir esta función miembro.

* **Desarrollo de un ejemplo: Solitario utilizando la clase Carta**

Utilizaremos la abstracción de un naipe para jugar al solitario.:

La palabra clave const se usa para describir cantidades que no se alterarán y puede aplicarse a variables de ejemplar, argumentos u otros nombres. En este caso, el ancho y el largo de la carta son valores constantes. Por el momento, se usará la creación de una constante separada para cada ejemplar de VistaDeCarta.C++ no oculta la distinción entre un objeto y un puntero a un objeto, mientras que en Object Pascal todos los objetos quedan representados internamente como punteros, aunque el usuario nunca los vea. Los pormenores sobre punteros implícitos y explícitos se tratarán en el contexto de las técnicas de asignación de memoria.

```
// implementación de la clase VistaDeCarta

# include carta.h

int VistaDeCarta::incluye(int a, int b)

{ if ((a >= x()) && (a <= x() + AltoCarta) &&(b >= y()) && (b <= y() + AnchoCarta))
```

return 1;

}

Esta aplicación necesitará la clase PilaDeCartas. Sin embargo, existen diversos tipos de pilas en los juegos de solitario: está lLa pila que se va formando con todas las cartas de cada palo; la pila de mesa, en donde tiene lugar la mayor parte del juego; la pila de la baraja, de la cual se sacan las cartas; y la pila de descarte, en donde se colocan las cartas que se han eliminado. Cada una de estas pilas está representada por una subclase diferente de la clase PilaDeCartas.

Una parte importante del juego del solitario consiste en mover cartas de una pila a otra. En la solución a implementar, cada pila será responsable de decidir si una carta dada se puede mover hacia ella. La acción por omisión (de la clase PilaDeCartas) es sólo di no; ninguna carta puede moverse ilegalmente. Una PilaDePalos, que representa a las cartas que están encima de la superficie de juego ordenadas por palos desde ases hasta reyes, puede tomar una carta si la pila está vacía y la carta es un as, o si la pila no está vacía y la carta es la siguiente en la secuencia. Un conjunto, o pila de mesa (representado por la clase PilaDeMesa) puede aceptar una carta si ésta es un rey y la pila está vacía o si ésta puede ser jugada sobre la carta superior de la pila.

#define nilLink(NodoCarta *) 0

int PilaDeCartas::puedoTomar(Carta *unaCarta)

{ // sólo di no

return 0;

}

int PilaDeMesa::puedoTomar(Carta *unaCarta)

{ if (tope == nilLink)

{ // puede tomar reyes en pila vacía

if (unaCarta−>rango() == 13)

return 1;

return 0;

}

// ver si los colores son diferentes

if (tope−>carta())−>color() == unaCarta−>color())

return 0;

// ver si los números son válidos

if (tope−>carta())−>rango() − 1)== unaCarta−>rango())

return 1;

return 0;

}

int PilaDePalos::puedoTomar(Carta *unaCarta)

{ if (tope == nilLink)

{ // estamos vacíos, puedo tomar un as

if (unaCarta−>rango() == 1)

return 1;

return 0;

}

if (tope−>carta())−>palo() != unaCarta−>palo())

return 0;

if (tope−>carta())−>rango() + 1)== unaCarta−>rango())

return 1;

return 0;

}

Para el compilador de C++ existe una diferencia sutil en el significado, que depende de que el método puedoTomar sea declarado virtual en la clase PilaDecartas. Ambos son válidos. Para que el método trabaje en lo que se ha descrito como una forma OO, se debe declarar como virtual. El modificador virtual es opcional como parte de las declaraciones de las clases hijas; sin embargo, por razones de documentación se repite generalmente en todas las clases.

class PilaDeCartas

{ protected:

NodoCarta *tope // primera carta de la pila

int x; // ubicación x de la pila

int y; // ubicación y de la pila

public:

PilaDeCartas()

```
{ tope = nilLink; }

PilaDeCartas(int a, int b)

{ x = a; y = b; tope = nilLink; }

...

int virtual puedoTomar(carta *c);

...

};
```

class PilaDePalos : public PilaDeCartas

```
{ public:

int virtual puedoTomar(carta *c);

};
```

class PilaDeMesa : public PilaDeCartas

```
{ int columna; // muestra columna de números

public:

PilaDeMesa(int c, int a, int b)

{ x = a; y = b; columna =c; }

...

int virtual puedoTomar(carta *c);

...

};
```

Si no se proporciona el modificador virtual, de cualquier modo el método reemplazaría al de nombre similar que hay en la clase paterna; sin embargo, se alteraría el enlace del método con el mensaje. Los métodos no virtuales son estáticos. Esto es, el enlace de una llamada a un método no virtual se lleva a cabo en tiempo de compilación, basándose en el tipo (estático) declarado del receptor y no en tiempo de ejecución. Si la palabra clave virtual se eliminara de la declaración de puedoTomar, las variables declaradas como PilaDeCartas ejecutarían el método de la clase PilaDecartas (sin considerar a la clase real del valor contenido en la variable) y las variables declaradas como PilaDeMesa o PilaDePalos ejecutarían el método de su clase respectiva.

Según C++, si no se emplean funciones virtuales, entonces la herencia no impone en absoluto ningún tiempo de procesamiento adicional en la ejecución. Sin embargo, el hecho de que ambas formas sean válidas, aunque difieran en sus interpretaciones, con frecuencia es causa de errores sutiles en los programas C++.

En cuanto a como se escriben los constructores cuando las clases usan herencia, la clase PilaDeCartas contiene un constructor que toma dos argumentos que representan las coordenadas de una esquina de la pila de cartas. Para la clase PilaDeMesa, se desea agregar un tercer argumento correspondiente al número de columna de la pila en la mesa de juego. Como se vio, los constructores no son como los otros métodos. Por ejemplo, se invocan sólo indirectamente cuando se crea un nuevo objeto. Además, el nombre del constructor de una subclase es necesariamente diferente del nombre del constructor en la superclase. Una tercera diferencia es que por lo general se desea extender, y no anular, el constructor de la clase base. Es decir, todavía se desea que se efectúe la inicialización de la clase base, pero además se quiere llevar a cabo más acciones.

Un constructor como el incluido en la clase PilaDeMesa, siempre invocará al constructor sin argumentos en la clase base antes de ejecutar el cuerpo del constructor dado en la subclase. Se dará un mensaje de error si no hay ningún constructor en la clase paterna que no tome argumentos. Observar que, en este caso, es necesario que el constructor de la clase PilaDeMesa duplique parte del código contenido en el constructor de dos argumentos para la clase PilaDeCartas.

Los miembros de datos (variables de ejemplar) también pueden anularse en C++. Sin embargo, tales campos nunca son virtuales. Esto es, a cada variable se le asignan localidades bien determinadas de memoria y, para tener acceso a variables específicas, el programador debe usar un nombre calificado, el cual se forma anteponiendo al nombre del identificador el nombre de la clase en que se define a la variable.

6.4. REFINAMIENTO

Un método es anulado por una clase hija con la misma frecuencia tanto para añadir una mayor funcionalidad como para sustituir una funcionalidad nueva. Por ejemplo, un método que inicializa una estructura de datos puede necesitar llevar a cabo la inicialización de la superclase y luego alguna inicialización particular adicional para las clases derivadas. Ya que, en la mayoría de los lenguajes OO el acceso a los datos y a los métodos se hereda en la clase hija, la adición de la nueva funcionalidad se podría lograr con sólo copiar el código anulado de la clase paterna. Sin embargo, semejante enfoque viola algunos de los principios importantes del diseño OO; por ejemplo, reduce el compartimiento de código, bloquea la ocultación de información y reduce la confiabilidad debido a que los errores corregidos en la clase paterna pueden no transmitirse a las clases hijas. Así resulta útil que haya algún mecanismo desde dentro de un método anulado que invoque el mismo método y que reutilice de este modo el código del método anulado. Cuando un método invoca el método anulado de la clase paterna de esta manera, se dice que el método refina el de la clase paterna.

Existen dos técnicas para proporcionar la capacidad de encontrar un método de una superclase: la primera, usada en Object Pascal, simplemente cambia la clase inicial para la búsqueda del método, de la clase del receptor a la superclase de la clase del método que se está ejecutando en el momento; la segunda, usada en C++, nombra en forma explícita la clase de la cual se va a tomar un método, eliminándose por completo el mecanismo de búsqueda en tiempo de ejecución y permitiendo que la llamada al procedimiento realice el paso de mensajes. La segunda técnica es menos elegante que la primera, especialmente si algún método se copia de una clase a otra no relacionada, o más poderosa al poder designarse los métodos de ancestros no inmediatos (aun si son anulados en clases intermedias), y los conflictos entre múltiples superclases pueden resolverse con el uso de la herencia múltiple.

C++ usa la técnica de nombres calificados para dar acceso a métodos de las clases paternas en métodos anulados. Un nombre calificado se escribe como un nombre de clase seguido por un par de dos puntos y el nombre de un procedimiento. El nombre calificado elimina el mecanismo virtual de paso de mensajes y asegura que el método se tome de la clase nombrada.

Retomando el ejemplo de programa para jugar al solitario, una Pila usa el método agregarCarta para agregar

una nueva carta a la pila (suponiendo que la validez de la jugada se ha verificado por medio del método puedoTomar). El método en la clase PilaDeCartas se encarga de las acciones básicas del enlace de cartas en una lista, la actualización del valor de la variable de ejemplar tope, etc. La clase PilaDeMesa, que representa al conjunto, debe además preocuparse por la colocación física de una carta en la superficie de juego. Si la pila del conjunto está vacía, la carta se coloca simplemente de forma normal. Por otra parte, si la pila del conjunto tiene una carta boca arriba y se está jugando el argumento carta, entonces la última carta debe bajarse un poco de modo que aún pueda verse la carta superior. Sin embargo, para conservar el área de juego de la mesa, sólo se muestran las cartas descubiertas superior e inferior, y cualquier carta descubierta intermedia queda oculta por la carta superior.

// colocar una sola carta en una pila de cartas

void PilaDeCartas::agregarCarta(carta *unaCarta)

{ if (unaCarta != cerocartas)

{ unaCarta−>ponerNodo(tope);

tope = unaCarta;

tope−>moverA(x,y);

}

}

// colocar una sola carta en una pila de mesa

void PilaDeMesa::agregarCarta(carta *unaCarta)

{ int tx, ty;

if (tope == cerocartas) // si la pila está vacía, sólo colocarla normalmente

PilaDeCartas::agregarCarta(unaCarta);

else // si no se debe mostrar adecuadamente

{ tx = tope−>ubicx();

ty = tope−>ubicy();

// calcular dónde colocar la carta

if (tope−>anverso() && tope−>siguiente() != cerocartas &&

(tope−>siguiente())−>anverso()); // no hacer nada, colocarla sobre la carta

superior

else

```
ty += 30; // si no, bajarla un poco

PiladeCartas::agregarCarta(unaCarta);

}

}
```

Se observó antes que un constructor en una clase hija invoca siempre a un constructor de la clase paterna. La técnica delineada en la sección 7.3 invoca al constructor de la clase paterna que no tiene argumentos, aunque éste pueda ser sólo uno de los diferentes constructores definidos por la clase paterna. Ahora se mostrará una técnica que permite que la descripción de la clase hija seleccione cualquier constructor deseado con propósitos de inicializar la parte de la clase heredada del padre.

```
class PilaDeMesa : public PilaDeCartas

{ int columna; // muestra número de columna

public:

PilaDeMesa(int, int, int);

...

};

...

PilaDeMesa::PilaDeMesa(int c, int x, int y) : PilaDeCartas(x,y)

{ columna = c; }
```

Aquí, para la clase PilaDeMesa se puede contrastar el constructor presentado en la sección 7.3. Observar los dos puntos que siguen al cierre del paréntesis de la lista de argumentos en el cuerpo del constructor. Este signo es seguido por una invocación al constructor de la clase paterna. De esta manera, el constructor de dos argumentos para la clase PilaDeCartas, que se muestra en la sección 7.3, se invocará justamente antes de la ejecución del cuerpo del constructor que se muestra aquí. Si varían los tipos de valores pasados en esta segunda lista de argumentos, el programador puede escoger selectivamente cual de los diversos constructores de la clase paterna deberá invocarse.

7. HERENCIA Y TIPOS

7.1. LA RELACIÓN ES−UN

Se ha descrito la relación es−un como una propiedad fundamental de la herencia. Una forma de ver dicha relación es considerarla como un medio para asociar un tipo (como el tipo de una variable) con un conjunto de valores; esto es, los valores que puede tomar legalmente una variable. Si una variable x se declara como ejemplar de una clase específica, por ejemplo Ventana, entonces será válido que x tome los valores de la clase Ventana. Si se tiene una subclase de Ventana, por ejemplo VentanaDeTexto, puesto que VentanaDeTexto es−una Ventana, entonces ciertamente deberá tener sentido que x pueda tener un valor del tipo VentanaDeTexto.

Lo anterior tiene sentido en forma intuitiva pero, desde un punto de vista práctico, existen dificultades asociadas con la implementación de los lenguajes OO.

7.2. DISTRIBUCIÓN EN MEMORIA

Considérese la siguiente pregunta: ¿Cuánta memoria se deberá asignar a una variable que se declara de una clase específica? O, más concretamente, ¿Cuánta memoria se deberá asignar a la variable x descrita como un ejemplar de la clase Ventana?

Comúnmente, se cree que las variables asignadas en el stack, como parte del proceso de activación del procedimiento, son más eficientes que las asignadas en el heap. Por esto, los diseñadores e implementadores de lenguajes hacen grandes esfuerzos para lograr que las variables se asignen en el stack. Pero existe un problema importante con la asignación: el requerimiento de memoria debe determinarse estáticamente, en tiempo de compilación o, a más tardar, en tiempo de llamada al procedimiento. Dichos tiempos son muy anteriores al momento en que se conocen los valores que tendrá la variable.

La dificultad radica en que las subclases pueden introducir datos que no estén presentes en la superclase. Por ejemplo, la clase VentanaDeTexto probablemente trae consigo áreas de datos para buffers de edición, colocación del punto de edición actual, etc.

```
class Ventana

{ int altura;

int ancho;

...

public:

virtual void oops();

};

class VentanaDeTexto : public Ventana

{ char *indice;

int ubicaciónCursor;

...

public:

virtual void oops();

};

Ventana x;

Ventana *y;
```

...

y = new VentanaDeTexto;

¿Se deben tomar en consideración estos valores de datos cuando se asigna espacio para x? Al parecer hay, al menos, tres respuestas plausibles:

− Asignar la cantidad de espacio necesario sólo para clase base. Esto es, asignar a x sólo las áreas de datos declaradas como parte de la clase Ventana, pasando por alto los requisitos de espacio de las subclases.

− Asignar la cantidad de espacio máximo para cualquier valor legal, ya sea de la clase o de la subclase.

− Asignar sólo la cantidad de espacio necesario para tener un puntero único; asignar el espacio necesario para el valor en tiempo de ejecución en el heap y actualizar apropiadamente el valor del puntero.

*** Mínima asignación estática de memoria**

Lenguaje C se diseñó teniendo en mente la eficiencia en tiempo de ejecución y, por ello, dada la amplia creencia de que la asignación de memoria basada en el stack da como resultado tiempos más rápidos de ejecución de los que son posibles con variables dinámicas, no es de sorprender que C++ escogiera conservar los conceptos tanto de variables no dinámicas como dinámicas (asignadas en tiempo de ejecución).

En C++, la distinción se hace en términos de cómo se declara una variable y en conformidad a si se usan punteros para tener acceso a los valores de una variable. Con respecto al ejemplo anterior, se asignará espacio para la variable x en el stack cuando entre el procedimiento que contiene la declaración. El tamaño de esta área será el de la clase base sola. Por otra parte, la variable y contiene sólo el puntero. El espacio para el valor apuntado por y se asignará dinámicamente cuando se ejecute una sentencia new. Ya que es este el momento en el cual se conoce el tamaño de VentanaDeTexto, no hay problemas asociados con asignar una cantidad suficiente de memoria en el heap para contener una VentanaDeTexto. ¿Qué pasa cuando el valor apuntado por y es asignado a x? Esto es, el usuario ejecuta la sentencia

x = *y;

El espacio asignado a x es sólo lo suficientemente grande para acomodar una Ventana, mientras que el valor apuntado por y es más grande.

x y

Está claro que no pueden ser copiados todos los valores apuntados por y. El comportamiento por omisión consiste en copiar sólo los campos correspondientes. Luego, está claro que alguna información se pierde. Algunos autores usan el término slicing (corte) para este proceso, ya que los campos del lado derecho que no se encuentran en el izquierdo se cortan durante la asignación.

¿Es importante que se pierda información? Sólo el usuario puede notar la diferencia. ¿Cómo se puede hacer? La semántica del lenguaje asegura que sólo los métodos definidos en la clase Ventana se pueden invocar usando x. Los métodos definidos en Ventana e implementados en esa clase no pueden modificar los datos definidos en subclases, por lo que ningún acceso es posible ahí. Con respecto a los métodos definidos en la clase Ventana, pero anulados en la subclase, considerar lo siguiente:

void Ventana::oops()

{ printf(Ventana oops); }

```
void VentanaDeTexto::oops()

{ printf(VentanaDeTexto oops %d,ubicaciónCursor); }
```

Si el usuario ejecutara x.oops() y se seleccionara el método de la clase VentanaDeTexto, se intentará mostrar el valor del dato x.ubicaciónCursor, el cual no existe en x. Ello ocasionaría una violación del uso de la memoria o (con más probabilidad) produciría basura.

La solución a este dilema consiste en cambiar las reglas que se usan para enlazar un procedimiento con la invocación de un método virtual. Para los objetos asignados en el stack en tiempo de llamada al procedimiento, el enlace para una llamada en un procedimiento virtual se toma de la clase estática (la clase de una declaración) y no de la clase dinámica (la clase del valor actual).

En forma más precisa, durante el proceso de asignación el valor se cambia del tipo que representa a la subclase (de *y) al valor del tipo representado por la clase paterna (de x). Esto es análogo a la forma en que una variable entera se puede cambiar durante el proceso de ejecución a una variable real. De esta forma, se puede asegurar que, para las variables basadas en el stack, la clase dinámica sea siempre la misma que la clase estática. Dada esta regla, no es posible que un procedimiento tenga acceso a campos que físicamente no estén presentes en el objeto. El método seleccionado en la llamada x.oops() sería encontrado en la clase Ventana y el usuario no notaría el hecho de que se perdió memoria durante la ejecución.

Sin embargo, esta solución se produce únicamente a expensas de introducir una inconsistencia sutil. Las expresiones que involucran punteros enlazan los métodos virtuales en la forma descrita en capítulos anteriores. Luego, estos valores se ejecutarán en forma diferente de la que involucran valores no dinámicos. Considérese lo siguiente:

```
Ventana x;

VentanaDeTexto *y, *z;

...

y = new VentanaDeTexto;

x = *y;

z = y;

x.oops();

(*z).oops();
```

Aunque es probable que el usuario piense que x y el valor al que apunta z son lo mismo, es importante recordar que la asignación a x ha transformado el tipo del valor y debido a lo cual, la primera llamada a oops() invocará el método que se encuentra en la clase Ventana.

*** Máxima asignación estática de espacio**

Una solución diferente para el problema de decidir cuánto espacio asignar a una declaración de objeto sería asignar la cantidad máxima de espacio usado para cualquier valor que la variable posiblemente pudiera tener, ya fuera en una clase nombrada en la declaración o de cualquier subclase posible. Semejante enfoque es similar al usado en la presentación de tipos superpuestos en los lenguajes convencionales, como los registros

con variantes en Pascal o las estructuras de unión en C. Así, no sería posible asignar un valor de tamaño superior al de la capacidad de destino. Esto es, aparentemente, la solución ideal de no ser por un pequeño problema: no se puede conocer el tamaño de cualquier objeto sino hasta que se ha examinado el programa completo. Debido a que este requisito es muy limitante, ningún lenguaje OO usa este enfoque.

*** Asignación dinámica de memoria**

El tercer enfoque no guarda en absoluto el valor de los objetos en el stack. Cuando se asigna espacio para un nombre en el stack al principio del procedimiento, el espacio es suficientemente grande para un puntero. Los valores se conservan en un bloque separado de datos, el heap, que no está sujeto al protocolo LIFO de asignación del stack. Ya que todos los punteros tienen un tamaño fijo constante, no se genera ningún problema cuando el valor de una variable se asigna a la variable declarada como perteneciente a una superclase.

Este enfoque es el usado por Object Pascal (Smalltalk y Objective−C). Tanto para punteros como para objetos, es necesario invocar el procedimiento new para asignar espacio antes de que pueda manejarse un objeto. Así mismo, es necesaria la llamada explícita a dispose para liberar el espacio asignado al objeto.

Además de requerir asignación explícita de memoria por parte del usuario, otro problema que ocurre con esta técnica es que con frecuencia está ligada al uso de la semántica de punteros para asignación. Cuando se usa la semántica para punteros, el valor transferido en un enunciado de asignación es simplemente el valor del puntero, más que el valor apuntado. Considerar el código para implementar un buffer de una palabra que pueda ser puesta y recuperada por el usuario:

```
type bufferEnteros = object;

valor : integer;

end;

var x, y : bufferEnteros;

begin

new(x);

x.valor := 5;

writeln(x.valor);

y := x;

y.valor := 7;

writeln(x.valor);

end;
```

Obsérvense las dos variables x e y declaradas como ejemplares de dicha clase. La ejecución de la última sentencia imprime el valor 7.

Este resultado se debe a que x e y apuntan al mismo objeto. El uso de semántica de punteros para objetos en

Object Pascal es particularmente confusa ya que la alternativa, la semántica de copia, se usa para todos los otros tipos de datos. Si x e y fueran registros, por ejemplo, la asignación de y a x supondría la copia de la información de y a x.

7.3. ASIGNACIÓN

Existen dos interpretaciones diferentes que se pueden dar a una asignación de la forma

y = x

− Semántica de copia: La asignación cambia el r−valor de y por el r−valor de x.

− Semántica de punteros: Semántica de copia en la cual el r−valor es un l−valor.

Por lo general, si se usa una semántica de punteros, los lenguajes proporcionan algún medio para producir una copia verdadera. Normalmente se da también el caso de que la semántica de punteros se usa con más frecuencia cuando los objetos son asignados por heap (dinámicamente), más que por stack (automáticamente). Cuando se usa semántica de punteros es común que un valor sobreviva al contexto en el cual se usó.

Los lenguajes OO difieren en las semánticas usadas, proporcionando una, la otra o una combinación de ambas.

El algoritmo por omisión que se usa en C++ para asignar un valor de clase a una variable consiste en copiar recursivamente los campos de datos correspondientes. Sin embargo, es posible sobrecargar el operador de asignación (=) para producir cualquier comportamiento deseado.

Cuando se usa sobrecarga de asignación, la interpretación es que el operador = es un método de la clase del lado izquierdo, con argumento del lado derecho. El resultado puede ser void si no son posibles las asignaciones contenidas, aunque en forma más típica el resultado es una referencia al lado izquierdo.

```
Cadena &Cadena::operator = (Cadena &derecha)

{ long = derecha.long; // copia el tamaño

c = derecha.c; // copia el puntero a los valores

return (*this);

}
```

El ejemplo anterior muestra la asignación de un tipo de datos cadena, la cual redefine la asignación de modo que dos copias de la misma cadena comparten caracteres.

Una fuente más común de confusión en C++ es el hecho de que el mismo símbolo se usa para asignación y para inicialización. En C,

```
int límite = 300;
```

es una abreviación de

```
int límite;
```

límite = 300;

En C++, una asignación usada en una declaración puede seleccionar los constructores invocados. Es decir, al usar los constructores de la clase Complejo declarada en la sección 4.3, el enunciado

Complejo x = 4;

significa lo mismo que la declaración

Complejo x(4);

La inicialización se usa con frecuencia con las variables de referencia y produce una situación parecida a la semántica de punteros. Por ejemplo, si s es una Cadena válida, lo siguiente hace a t un alias de s:

Cadena &t = s;

Las variables de referencia se usan más a menudo para implementar el paso de parámetros de llamada por referencia. Se puede considerar este uso como una forma de asignación por puntero, en la que se asigna el valor del argumento al parámetro. Por supuesto, la semántica de punteros en C++ se puede lograr también por medio del uso de variables puntero.

En general, el paso de parámetros se puede ver en parte como una forma de asignación. Considérense las siguientes definiciones:

class Base

{ public:

virtual void ver();

};

class Derivada

{ public:

virtual void ver();

};

void f (Base);

void g (Base &);

Derivada z;

f(z); g(z);

Aquí, tanto la función f como la g toman como argumento un valor declarado como el tipo base, pero g declara el valor como un tipo de referencia. Si se llama a f con un valor de tipo derivado, el valor es convertido (cortado) para crear un valor del tipo base como parte de la asignación de los argumentos. Así, si se invoca ver desde dentro de f, se usará la función virtual de la clase Base. Por otra parte, esta conversión, o

corte, no ocurre como parte del paso de argumentos a g. Luego, si se invoca ver desde dentro de g, se usará el procedimiento de la clase derivada. Esta diferencia de interpretación, que depende sólo de un carácter en la cabecera de la función, se conoce a veces como el problema del corte.

La sobrecarga del símbolo de asignación en C++ no altera el significado del símbolo definido por defecto.

7.4. IGUALDAD

Como la asignación, el proceso de decidir si un objeto es equivalente a otro es más sutil de lo que pudiera parecer. La dicotomía más obvia reflejará la distinción entre la semántica de punteros y la semántica de copia en la asignación. Muchos lenguajes usan equivalencia de los punteros. Se considera que dos referencias a objetos son equivalentes por apuntamiento si apuntan al mismo objeto. A esta equivalencia se le conoce también como identidad de objetos.

Con frecuencia, un programador no está tan interesado en saber si dos variables apuntan a objetos idénticos, como en que dos objetos posean el mismo valor. Para los números o los strings, el mecanismo es por lo general la igualdad determinada por bits. Con esta interpretación, dos objetos son equivalentes si sus representaciones de bits en la memoria son las mismas.

Para los objetos compuestos, como los registros de Pascal, puede no ser suficiente la igualdad determinada por bits. Con frecuencia, la distribución de memoria para tales valores puede tener espacios vacíos, o relleno, que no están relacionados con los valores que tiene el objeto. Tal relleno debe ser considerado en la determinación de la igualdad. De este modo, se usa el mecanismo conocido como igualdad de miembros, en el cual se comparan los miembros correspondientes para determinar su igualdad, aplicando la regla en forma recursiva hasta que se encuentran los miembros no registrados, en los cuales se aplica entonces la igualdad determinada por bits. Si todos los miembros coinciden, entonces se considera que dos registros son iguales. Si hay alguna discrepancia, entonces son desiguales. Esta relación se conoce a veces como equivalencia estructural.

Las técnicas de POO agregan su propio giro a la prueba de la equivalencia estructural. Como se ha visto, los ejemplares de una subclase pueden tener campos que no se encuentran en las clases paternas. Si la interpretación de la comprobación x = y se basa sólo en los campos que tiene x, entonces puede suceder (si x e y son ejemplares de clases diferentes) que x = y es verdadera, pero y = x es falsa.

Existe un sentido en el cual el problema de la igualdad es más fácil de resolver que las dificultades correspondientes a la asignación. Aunque la asignación se considera, en general, como parte de la sintaxis y la semántica de un lenguaje, y puede ser que no sea modificable, el programador es siempre libre de proporcionar sus propios métodos para la prueba de la igualdad. De esta forma, no existe un significado consistente único para la igualdad y ésta puede significar algo diferente para cada clase de objetos.

C++ no proporciona un significado por omisión para la igualdad. Cada clase puede proporcionar su propio significado de sobrecarga para el operador ==. Se usan las mismas reglas para eliminar la ambigüedad de los operadores sobrecargados que los que se usan con otras funciones sobrecargadas.

```
class A

{ public:

int i;

A(int x)

{ i = x; }
```

```
int operator == (A &x)

{ return i == x.i; }

};

class B : public A

{ public:

int j;

B(int x, int y) : A(x)

{ j = y; }

int operator == (B &x)

{ return (i == x.i ) && (j == x.j); }

};
```

En este ejemplo, si a y b son ejemplares de las clases A y B, respectivamente, entonces, tanto a == b como b == a usarán el método que se encuentra en la clase A, mientras que b == b usará el método que se encuentra en la clase B.

7.5. CONVERSIÓN DE TIPOS

En lenguajes OO con asignación estática de tipos, como C++ y Object Pascal, no es válido asignar el valor de una clase a una variable declarada como ejemplar de una subclase. No se puede asignar el valor que es conocido (para el compilador) sólo como ejemplar de la clase Ventana a una variable declarada como de la clase VentanaDeTexto.

Sin embargo, hay ocasiones en las que es deseable romper dicha regla. La situación se presenta con mucha frecuencia cuando el programador sabe, por información adicional, que un valor, pese a mantenerse en una variable de alguna superclase, es realmente ejemplar de una clase más específica. En estas circunstancias es posible (aunque no se aconseja) burlar el sistema de tipos.

En C++ se logra burlarlo mediante el constructor de C llamado cast. La conversión de tipos dirige al compilador para convertir el valor de un tipo en otro. Esta técnica se usa muy a menudo con punteros, en cuyo caso ninguna transformación física tiene lugar, sino sólo la transformación lógica.

Suponer que en vez de hacer que cada ejemplar de Carta mantenga un puntero a una lista enlazada, se tiene una clase Nodo generalizada:

```
class Nodo

{ protected:

Nodo *nodo;

Nodo *siguiente();
```

```
void ponerNodo(Nodo *ele);

};

inline void Nodo::ponerNodo(Nodo *ele)

{ nodo = ele; }

inline Nodo *Nodo::siguiente()

{ return nodo; }
```

Entonces, la clase Carta se podría convertir en subclase de Nodo. Ya que el método ponerNodo se hereda de la superclase, no necesita repetirse. Sin embargo, existe un problema con el método heredado siguiente: éste exige que se devuelva un puntero a un ejemplar de Nodo y no de Carta. No obstante, es un hecho que el objeto es realmente una Carta ya que es la única especie de ese objeto que se está insertando en la lista. Por lo tanto, se reescribe la clase Carta para anular el método siguiente y usar una conversión de tipos para cambiar el tipo de retorno.

```
class Carta : public Nodo

{

public:

...

Carta *siguiente();

};

inline Carta *Carta::siguiente()

{ return (Carta *) Nodo::siguiente(); }
```

Nótese que el método siguiente no se declara como virtual. No es posible alterar el tipo de retorno de un procedimiento virtual. Es importante recordar que, en C++, esta conversión de tipos es válida sólo con punteros y no con los objetos mismos. C++ tampoco mantiene información completa en tiempo de ejecución acerca del tipo de los objetos. Así, el uso erróneo de tales formas puede ocasionar grandes estragos.

8. HERENCIA MÚLTIPLE

8.1. EL CONCEPTO

Hasta ahora, se ha supuesto que una clase hereda de una sola clase paterna. Aunque semejante situación es ciertamente común, existen sin embargo, ocasiones en las que los conceptos son heredados de dos o más bases independientes. Si se piensa en que las clases corresponden a categorías y alguien trata de describirse a sí mismo en términos de los grupos a los cuales pertenece, es muy probable que se encuentre con muchas clasificaciones que no se superpongan. Por ejemplo, padre, profesor, hombre y chileno; ninguna categoría de las anteriores es subconjunto propio de otra.

Aunque, normalmente se ve la herencia simple como una forma de especialización (un Alfarero es un Artista),

es más común ver la herencia múltiple como un proceso de combinación (un PintorDeRetratos es tanto un artista como un pintor).

Pintor Artista

PintorDeCasas PintorDeRetratos Alfarero

8.2. MENÚS ANIDADOS

Se cita como ejemplo una biblioteca para la creación de interfaces gráficas para usuarios. En este sistema, los menús son descritos por una clase Menú. Los ejemplares de Menú conservan características como el número de entradas, una lista de sus elementos, etc. La funcionalidad asociada con un menú incluye la capacidad para ser mostrado en una pantalla gráfica y para seleccionar uno de sus elementos. Cada elemento de un menú está representado por un ejemplar de la clase ItemDeMenú. Los ejemplares de ItemDeMenú mantienen su texto, su menú paterno y el mandato a ejecutar cuando se seleccione un elemento del menú.

Un recurso común de las interfaces gráficas para usuario, es el que se conoce como un menú anidado (a veces llamado menú en cascada), el cual se necesita cuando un elemento del menú tiene diversas alternativas.

Un menú anidado es claramente un Menú. Mantiene la misma información y debe funcionar como un Menú. Además, también es claramente un ItemDeMenú: debe mantener su nombre y la capacidad para ejecutar (mostrándose a sí mismo) cuando la entrada asociada se selecciona desde el menú paterno. Se puede conseguir un comportamiento importante con poco esfuerzo al permitir que la clase MenúAnidado herede de ambos padres. Por ejemplo, cuando se le pide al menú anidado que ejecute su acción asociada (heredada de la clase ItemDeMenú), mostrará a su vez su menú completo (mediante la ejecución del método de dibujo heredado de la clase Menú).

Como en el uso de la herencia simple, la propiedad importante que hay que recordar cuando se usa herencia múltiple es la relación es−un. En este caso, la herencia múltiple es propiedad porque es claramente el caso en el que ambas afirmaciones tienen sentido: un menú anidado es−un menú y un menú anidado es−un ítem de menú.

Cuando no se emplea la relación es−un, se puede emplear mal la herencia múltiple.

Cuando se usa adecuadamente la herencia múltiple ocurre un cambio sutil pero no menos importante en la forma de ver la herencia. La interpretación es−un de la herencia ve una subclase como una forma más especializada de otra categoría, representada por la clase paterna. Al usar la herencia múltiple, una clase se ve como una combinación o colección de componentes diferentes, cada uno de los cuales ofrece un protocolo distinto y algún comportamiento básico, que se especializa para atender el respectivo caso.

8.3. HERENCIA MÚLTIPLE EN C++

Se ilustrará el uso de la herencia múltiple en C++ trabajando en un pequeño ejemplo. Suponer que, en un proyecto previo, un programador ha desarrollado un conjunto de clases para trabajar con listas enlazadas. La abstracción ha sido dividida en dos partes: la clase Nodo representa los nodos individuales en la lista y la clase Lista representa a la cabeza de la lista. La funcionalidad básica asociada con las listas enlazadas implica añadir un nuevo elemento. Pidiendo prestada una técnica de la programación funcional, las listas enlazadas proporcionan la capacidad para ejecutar una función en cada elemento de la lista, en la que la función se pasa como un argumento. Ambas actividades son permitidas por rutinas asociadas de la clase hija.

class Lista

```
{ public:
Nodo *elementos;
Lista()
{ elementos = (Nodo *) 0; }
void insertar(Nodo *n)
{ if (elementos) elementos−>insertar(n); else elementos = n; }
void enTodos(void f(Nodo *))
{ if (elementos) elementos−>enTodos(f); }
}
class Nodo
{ public:
Nodo *siguiente;
Nodo()
{ siguiente = (Nodo *) 0; }
void ponerNodo(Nodo *n)
{ siguiente = n; }
void insertar(Nodo *n)
{ if (siguiente) siguiente−>insertar(n); else ponerNodo(n); }
void enTodos(void f(Nodo *))
{ f(this); if (siguiente) siguiente−>enTodos(f); }
}
```

Se forman tipos especializados de nodos asignando subclases a la clase Nodo.

```
class NodoDeEnteros: public Nodo
{ int valor;
public:
NodoDeEnteros(int i) : Nodo()
```

```
{ valor = i; }

print()

{ printf(%d\n,valor); }

}

void mostrar(NodoDeEnteros *x)

{ x−>print() ; }

main()

{ Lista lista;

lista.insertar(new NodoDeEnteros(3));

lista.insertar(new NodoDeEnteros(17));

lista.insertar(new NodoDeEnteros(32));

lista.enTodos(mostrar);

}
```

Aquí, la clase NodoDeEnteros se usa para mantener valores enteros asociados con cada nodo. Se muestra, además, un pequeño programa que ilustra cómo se usa esta abstracción de datos.

Suponer ahora que, en el proyecto actual, el programador debe desarrollar un tipo de datos Arbol y descubre que se puede concebir un árbol como una colección de listas enlazadas. En cada nivel del árbol se usan los campos de nodo para apuntar a los hermanos. Sin embargo, cada nodo también apunta a una lista enlazada que representa a sus hijos.

De esta manera, un nodo en un árbol es tanto una Lista (pues mantiene un puntero a la lista de sus hijos) como un Nodo (pues mantiene un puntero a su hermano). En C++ se indica una herencia múltiple como ésta, listando simplemente los nombres de las superclases, separados por comas, después de los dos puntos de la descripción de la clase. Como en la herencia simple, cada superclase debe ir precedida por una palabra clave de visibilidad, ya sea public o private.

```
class Arbol: public Nodo, public Lista

{ int valor;

public:

Arbol(int i)

{ valor = i; }

print()
```

```
{ printf(%d\n,valor); }

void insertar(Arbol *n)

{ Lista::insertar(n); }

void insertarHijo(Arbol *n)

{ Lista::insertar(n); }

void insertarHermano(Arbol *n)

{ Nodo::insertar(n); }

void enTodos(void f(Nodo *))

{ /* primero procesa hijos */

if (elementos) elementos−>enTodos(f);

/* luego opera en sí mismo */

f(this);

/* después lo hace en los hermanos */

if (siguiente) siguiente−>enTodos(f);

}

}

main()

{ Arbol *t = new Arbol(17);

t−>insertar(new Arbol(12));

t−>insertarHermano(new Arbol(25));

t−>insertarHijo(new Arbol(15));

t−>enTodos(mostrar);

}
```

Ahora se debe resolver el problema de la ambigüedad en los nombres. El primer problema se refiere al significado de la función insertar. La ambigüedad en los nombres es en realidad un reflejo de la ambigüedad en el significado. Existen dos sentidos de la operación insertar en un árbol: uno es insertar un nodo hijo; el otro insertar un nodo hermano. El primer sentido es el que da la operación insertar de la clase Lista; el segundo lo proporciona la función insertar de la clase Nodo. Se decide que insertar signifique agregar un hijo y también ofrecer dos nuevas funciones que nombren específicamente el propósito.

Obsérvese que las tres funciones son, en cierto sentido, sólo renombramientos. No proporcionan una nueva funcionalidad, sino que tan sólo redireccionan la ejecución de la función previamente definida.

La ambigüedad del método enTodos es más compleja. Aquí la acción apropiada sería ejecutar un recorrido del árbol, como un recorrido en orden previo (hijo, raíz, hermano). Así, la ejecución es una combinación de las acciones proporcionadas por las clases subyacentes Nodo y Lista.

En ocasiones es necesario renombrar debido a una interacción sutil de C++ entre los mecanismos de herencia y sobrecarga paramétrica. Cuando se usa un nombre sobrecargado en C++, el mecanismo de herencia se emplea primero para encontrar el ámbito del nombre en el cual está definida la función. Los tipos de los parámetros se usan entonces para quitar la ambigüedad al nombre de la función dentro de ese ámbito.

```
class A

{ public:

void virtual mostrar(int i)

{ printf(en A %d\n,i); }

};

class B

{ public:

void virtual mostrar(double d)

{ printf(en B %g\n,d); }

};

class C: public B, public A

{ public:

void virtual mostrar(int i)

{ A::mostrar(i); }

void virtual mostrar(double d)

{ B::mostrar(d); }

};

main()

{ C c;

c.mostrar(13);
```

```
c.mostrar(3.14);

}
```

Aquí existen dos clases A y B que definen un método mostrar pero que toman argumentos diferentes. Se podría pensar que, debido a que se les puede quitar la ambigüedad a los dos usos de mostrar por el tipo de sus parámetros, una clase hija puede heredar de ambos padres y tener acceso a ambos métodos. Por desgracia, la herencia sola no es suficiente. Cuando el usuario llama al método mostrar con un argumento entero, el compilador no puede decidir si usar la función de la clase A (que corresponde al argumento en forma más cercana) o la clase B (que es el primer método que encuentra en la búsqueda de métodos y que es aplicable mediante la ejecución de una conversión automática del valor del parámetro). Afortunadamente, el compilador lo advertirá; sin embargo, la advertencia se produce en el momento en el cual se invoca el método ambiguo, no en el momento en el que se declara la clase. La solución consiste en redefinir ambos métodos en la clase hija C, con lo cual se elimina la controversia entre herencia y sobrecarga: ambas terminan por examinar de la clase C, en donde queda claro para el compilador que la sobrecarga paramétrica es sin duda intencional.

9. POLIMORFISMO

9.1. SIGNIFICADO

En los lenguajes de programación, un objeto polimórfico es una entidad, como una variable o argumento de una función, a la que se le permite tener valores de diferentes tipos en el transcurso de la ejecución. Las funciones polimórficas son funciones que tienen argumentos polimórficos.

La forma más común de polimorfismo en los lenguajes de programación convencionales es la sobrecarga, como la sobrecarga del signo + para significar tanto suma de enteros como suma de reales.

Una forma de ver el polimorfismo es en términos de abstracciones de alto y bajo nivel. Una abstracción de bajo nivel es una operación básica, como una operación en una estructura de datos, que se construye sobre unos cuantos mecanismos subyacentes. Una abstracción de alto nivel es un plan más general, como un algoritmo de ordenamiento, que da el enfoque general que se seguirá, sin especificar los detalles.

Los algoritmos, por lo general, se describen en una forma de alto nivel, mientras que la implementación real de un algoritmo particular debe ser una abstracción de alto nivel que se construye sobre una estructura de datos específica de bajo nivel.

En un lenguaje convencional, una abstracción de alto nivel debe fundamentarse en tipos específicos de estructuras de datos. Por lo tanto, es difícil llevar una abstracción de alto nivel de un proyecto a otro en un sentido que no sea más que el algorítmico. De esta manera, aun para tareas simples, como calcular la longitud de una lista, las abstracciones de alto nivel tienden, en consecuencia, a escribirse y reescribirse para cada nueva aplicación.

La fuerza del polimorfismo radica en que permite que los algoritmos de alto nivel se escriban una vez y se utilicen repetidamente con diferentes abstracciones de bajo nivel.

En los lenguajes de POO, el polimorfismo se presenta como un resultado natural de la relación es−un y de los mecanismos de paso de mensajes y de herencia.

El polimorfismo puro se presenta cuando una función puede tener varias interpretaciones (ser aplicada a argumentos de muchos tipos). El otro extremo ocurre cuando varias funciones diferentes se denotan todas por el mismo nombre, situación conocida como sobrecarga (o polimorfismo ad hoc).

9.2. OBJETOS POLIMÓRFICOS

Con excepción de la sobrecarga, el polimorfismo en los lenguajes OO se hace posible gracias a la existencia de los objetos polimórficos. Un objeto polimórfico es un objeto con muchas caras; esto es, un objeto que puede adoptar valores diferentes.

En lenguajes con enlace dinámico (como Smalltalk y Objective−C) todos los objetos son potencialmente polimórficos, es decir, pueden tomar valores de cualquier tipo.

En los lenguajes con asignación estática de tipos (como C++ y Object Pascal), la situación es ligeramente más compleja. En estos lenguajes, el polimorfismo aparece por medio de la diferencia entre la clase (estática) declarada y la clase (dinámica) real del valor contenido en la variable. Dicha diferencia se mantiene dentro de la estructura de la relación es−un. Una variable puede tener un valor del mismo tipo del de la clase declarada de la variable o de cualquiera de las subclases de la clase declarada.

En Object Pascal, esto es válido para todas las variables declaradas como de tipo objeto. En C++, cuando se usan declaraciones estáticas, los objetos polimórficos tienen lugar sólo por medio del uso de punteros y referencias. Cuando no se usan punteros, la clase dinámica de un valor se ve siempre forzada a ser la misma que la clase estática de la variable.

Sin embargo, cuando se usan punteros o referencias, el valor retiene su tipo dinámico.

```
class Uno

{ public:

virtual int valor()

{ return 1; }

};

class Dos: public Uno

{ public:

virtual int valor()

{ return 2; }

};

void asignaDirecta(Uno x)

{ printf(el valor por asignación es %d\n,x.valor()); }

void porApuntador(Uno *x)

{ printf(el valor por apuntador es %d\n,x−>valor()); }

void porReferencia(Uno &x)
```

```
{ printf(el valor por referencia es %d\n,x.valor()); }

main()

{ Dos x;

asignaDirecta(x);

porApuntador(&x);

porReferencia(x);

}
```

En este caso, la clase Uno define un método virtual valor que produce el valor 1. Este método es reemplazado en la clase Dos por un método que produce el valor 2. A continuación se definen tres funciones que toman un argumento de tipo Uno, pasándolo por valor, como puntero y por referencia. Finalmente, se invocan estas funciones con un parámetro actual de clase Dos. En el primer caso, el valor del parámetro actual es convertido en un valor de clase Uno, produciendo el resultado 1. Sin embargo, las otras dos funciones definen argumentos polimórficos. En ambos casos, el parámetro actual retendrá su clase dinámica y el valor impreso será 2.

9.3. SOBRECARGA

Se dice que el nombre de una función está sobrecargado si hay dos o más cuerpos de funciones asociados con ese nombre.

En la sobrecarga, es el nombre de la función el que es polimórfico; esto es, el que tiene muchas formas. Otra forma de pensar en la sobrecarga y el polimorfismo es que hay una sola función abstracta que toma diversos tipos diferentes de argumentos. El código real ejecutado depende de los argumentos que se den. El hecho de que un compilador a menudo pueda determinar la función correcta en tiempo de traducción (en un lenguaje con asignación predefinida de tipos) y puede generar una única secuencia de código es simplemente una optimización.

Suponer que un programador está desarrollando una biblioteca de clases que represente estructuras de datos comunes. Una parte de las diferentes estructuras de datos se puede usar para mantener una colección de elementos (conjuntos, diccionarios, arreglos y colas de prioridad, por ejemplo) y pudiera ser que todas éstas definieran un método insertar para añadir un nuevo elemento en la colección.

Esta situación −en la cual dos funciones completamente separadas se usan para proporcionar acciones semánticas similares para tipos de datos diferentes− ocurre a menudo en todos los lenguajes de programación y no nada más en los lenguajes OO. Tal vez, el ejemplo más común sea la sobrecarga del operador +. Aunque el código generado por un compilador para una suma de enteros es con frecuencia radicalmente diferente al código generado para una suma de reales, los programadores tienden a pensar en las operaciones como en una entidad única: la función suma.

En este ejemplo es importante destacar que puede ser que la sobrecarga no sea la única actividad que tenga lugar. A menudo, una operación separada semánticamente, la coerción, se relaciona también con las operaciones aritméticas. La coerción se manifiesta cuando el valor de un tipo se convierte en otro de tipo diferente. Si se permite la aritmética de tipo mixto, entonces la suma de dos valores se puede interpretar de diferentes maneras:

− Puede haber cuatro funciones que correspondan a entero + entero, entero + real, real + entero y real + real. En este caso, hay sobrecarga pero no coerción.

− Puede haber dos funciones diferentes para entero + entero y real + real. En los otros dos casos, el valor entero es forzado a ser cambiado a un valor real. De esta forma, la aritmética de tipo mixto es una combinación de sobrecarga y coerción.

− Puede haber sólo una función para la suma real + real. Todos los argumentos son forzados a ser reales. En este caso, sólo hay coerción pero no sobrecarga.

No hay nada intrínseco en el mecanismo de sobrecarga que requiera que las funciones asociadas con un nombre sobrecargado tengan similitud semántica.

Todos los lenguajes OO citados permiten el estilo de sobrecarga con métodos de un solo nombre en clases independientes y no relacionadas. Lo anterior no significa que las funciones o los métodos tomen argumentos de cualquier naturaleza. La naturaleza de la asignación estática de tipos de C++ y Object Pascal requiere todavía declaraciones específicas de todos los nombres y, por ello, el polimorfismo ocurre con más frecuencia como resultado de la anulación. La lista de tipos de argumentos, usada para quitar la ambigüedad a las invocaciones de funciones sobrecargadas, se llama signatura de argumentos.

Existe otro estilo de sobrecarga en el cual se permite a los procedimientos (o funciones o métodos) compartir un nombre en el mismo contexto y se le quita la ambigüedad mediante el número y tipo de los argumentos dados. Esto se llama sobrecarga paramétrica y se presenta en C++, en algunos lenguajes imperativos (como Ada) y en muchos lenguajes funcionales.

9.4. ANULACIÓN

En una clase (típicamente, una superclase abstracta) existe un método general definido para un mensaje particular. Casi todas las subclases de la clase en la que está definido ese método heredan y usan el mismo método. No obstante, en al menos una subclase se define un método con el mismo nombre. Tal método oculta el acceso al método general para ejemplares de esta clase. Así, se dice que el segundo método anula al primero.

La anulación suele ser transparente para el usuario de la clase y, como con la sobrecarga, a menudo se piensa que las dos funciones constituyen semánticamente una sola entidad.

La técnica de anulación de un método contribuye al compartimiento de código en la medida en que los ejemplares de las clases que no anulan el método pueden compartir todos el código original. Es sólo en aquellas situaciones en las que el método no es adecuado, en las que se proporciona un fragmento de código alternativo. Sin el mecanismo de anulación sería necesario que todas las subclases proporcionaran su propio método individual para responder al mensaje, aun cuando fueran idénticos muchos de tales métodos.

Un aspecto potencialmente confuso de la anulación en C++ es la diferencia entre anular un método virtual y anular un método no virtual. Como se vio en la sección 8.5, la palabra clave virtual no es necesaria para que se presente la anulación. Sin embargo, en estos dos casos es muy diferente. Si se quita la palabra virtual de los métodos valor de la sección 10.2, el resultado 1 sería impreso por las tres funciones. Sin la palabra clave virtual, el tipo dinámico de una variable (aun de una variable puntero o de referencia) es pasado por alto cuando se envía un mensaje a la variable.

9.5. MÉTODOS DIFERIDOS

Se puede pensar en un método diferido (a veces llamado método genérico) como una generalización de la

anulación. En ambos casos, el comportamiento descrito en una superclase es modificado por las subclases. Sin embargo, en un método diferido, el comportamiento en la superclase es esencialmente nulo, un retenedor de lugar, y toda la actividad útil es definida como parte del método anulado.

Una ventaja de los métodos diferidos es conceptual, en tanto que su uso permite al programador pensar en una actividad como asociada con una abstracción en un nivel más alto de lo que realmente puede ser el caso.

Por ejemplo, en una colección de clases que presentan formas geométricas, se podría definir un método para dibujar la forma en cada una de las subclases Círculo, Cuadrado y Triángulo. Se podría haber definido un método similar en la superclase Forma. En realidad, tal método no podría producir ningún comportamiento útil, ya que la superclase Forma no tiene suficiente información para dibujar la figura en cuestión. Sin embargo, la sola presencia de dicho método permitiría al usuario asociar el concepto dibujar con la clase única Forma y no con los tres conceptos separados Cuadrado, Triángulo y Círculo.

Existe una segunda razón, más práctica, para usar métodos diferidos. En lenguajes OO con asignación estática de tipos, como C++ y Object Pascal, a un programador se le permite enviar un mensaje a un objeto sólo si el compilador puede determinar que en efecto hay un método correspondiente que se acoplará con el selector del mensaje. Suponer que el programador desea definir una variable de la clase Forma que, en diversos momentos, contendrá ejemplares de cada una de la diferentes formas. Tal asignación es posible, ya que una variable declarada de una clase siempre puede contener ejemplares de subclases de dicha clase. Sin embargo, el compilador permitiría que se usara el mensaje dibujar con esta variable sólo si se pudiera asegurar que el mensaje va a ser entendido por cualquier valor que pueda esta asociado con la variable. Asignar un método a la clase Forma brinda de modo efectivo tal seguridad, ya que este método no puede invocarse a menos que sea anulado por la subclase del valor de la variable. Lo anterior es verdad aun cuando nunca se ejecute en realidad el método de la clase Forma.

En C++, un método diferido (llamado método virtual puro) debe declararse explícitamente mediante la palabra clave virtual. No se da el cuerpo de un método diferido; en su lugar, se asigna el valor 0 a la función. No es posible definir ejemplares de una clase que contiene un método virtual puro que no se ha anulado sin producir una advertencia del compilador. La redefinición de métodos virtuales puros debe ser manejada por una subclase inmediata. Aunque lo permitirán algunos compiladores, estrictamente hablando, no es legal que un método virtual puro sea ignorado por una subclase inmediata y sea redefinido más abajo en la jerarquía de clases.

```
class Forma

{ public:

Punto esquina; // esquina superior izquierda

void ponerEsquina(Punto &p)

{ esquina = p; }

void virtual dibujar() = 0;

};

class Círculo: public Forma

{ public:
```

```
int radio;

void ponerRadio(int i)

{ radio = i; }

void dibujar()

{ dibujarCírculo(esquina + radio, radio); }

};
```

9.6. POLIMORFISMO

Muchos autores reservan el término polimorfismo para situaciones en las que una función puede usarse con una variedad de argumentos, como opuestas a la sobrecarga en la que hay múltiples funciones definidas todas con un solo nombre. Tales recursos no se restringen a los lenguajes OO. Por ejemplo, en Lisp o ML es fácil escribir funciones que manipulen listas de elementos arbitrarios; tales funciones son polimórficas, ya que el tipo del argumento no se conoce en el momento en que se define la función. La capacidad para formar funciones polimórficas es una de las técnicas más poderosas de la POO. Esta permite que el código se escriba una vez, con un alto nivel de abstracción, y que sea ajustado como sea necesario para corresponder a una variedad de situaciones. En general, el programador lleva a cabo este ajuste enviando más mensajes al receptor del método. A menudo estos mensajes subsecuentes no se asocian con la clase en el nivel del método polimórfico, sino que más bien son métodos virtuales definidos en las clases bajas.

10. VISIBILIDAD Y DEPENDENCIA

10.1. VALORES ACTIVOS

Un valor activo es una variable para la cual se desea ejecutar una acción cada vez que cambia el valor de la variable. La construcción de un sistema de valor activo ilustra porqué es preferible el acoplamiento de parámetros que otras formas de acoplamiento, particularmente en los lenguajes OO.

Suponer que existe la simulación de una planta de energía nuclear, la cual incluye una clase Reactor que mantiene diversos elementos de información acerca del estado del reactor. Entre estos valores está la temperatura del moderador de calor, esto es, el agua que rodea las barras de enfriamiento. Más aún, suponer que la modificación de este valor ha sido designada de modo que se ejecute, según exigencias OO, invocando el método ponerCalor.

Imaginar que cuando el programa ya ha sido desarrollado y está en explotación, el programador decide que sería útil mostrar visualmente en forma continua la temperatura actual del moderador, conforme progresa la simulación.. Es aconsejable hacerlo de una forma lo menos invasora posible; en particular, el programador no desea cambiar la clase Reactor.

Una solución sencilla es crear una subclase ReactorGráfico, que no hace más que invalidar el método ponerCalor, actualizando la salida gráfica antes de invocar los métodos de la superclase. De esta manera, el programador sólo necesita reemplazar la creación de los nuevos objetos Reactor con objetos ReactorGráfico, creación que probablemente ocurra sólo una vez durante la inicialización.

10.2. CLIENTES DE SUBCLASES Y CLIENTES USUARIOS

Varias veces se ha hecho notar que los objetos y los métodos tienen una cara pública y una cara privada. El

lado público abarca todas las características, como métodos y variables de ejemplar, a las que se tiene acceso o que pueden ser manipuladas mediante código exterior al módulo. La cara privada incluye la cara pública, así como los métodos y variables de ejemplar accesibles sólo dentro del objeto. El usuario de un servicio proporcionado por un módulo necesita saber los detalles sólo del lado público. Los detalles de la implementación y otras características internas no importantes para la utilización del módulo, se pueden ocultar a la vista.

Ciertos investigadores han mencionado que la inclusión del mecanismo de la herencia significa que las clases en un lenguaje OO tienen una tercera cara; esto es, aquellas características accesibles para las subclases aunque no necesariamente para otros usuarios. El diseñador de una subclase para una clase dada probablemente necesitará conocer más detalles de implementación internos de la clase original que un usuario a nivel de ejemplar, pero puede no necesitar tanta información como el diseñador de la clase original.

Tanto el diseñador de una subclase como el usuario de una clase se pueden considerar clientes de la clase original, ya que hacen uso de los recursos proporcionados por la clase. Sin embargo, puesto que ambos grupos tienen requisitos, es útil distinguirlos mediante la clasificación en clientes de subclases y clientes usuarios. Estos últimos crean ejemplares de la clase y pasan mensajes a estos objetos. Los primeros crean nuevas clases basadas en la clase.

```
class PilaDeCartas

{ public:

PilaDeCartas();

PilaDeCartas(int, int);

void virtual agregaCarta(NodoCarta *);

int virtual puedoTomar(Carta *);

int contiene(int, int);

void virtual muestra();

void virtual inicia();

NodoCarta *retiraCarta();

void virtual select(int, int);

private:

int x; // ubicación x de la pila

int y; // ubicación y de la pila

protected:

NodoCarta *tope; // primera carta de la pila

};
```

Aquí, sólo los métodos asociados con la clase PilaDeCartas se dividen en las categorías pública, privada y protegida. Se puede tener acceso a las variables x e y o puede modificarse sólo mediante métodos asociados con la clase PilaDeCartas. Además, se puede tener acceso a la variable tope y modificarla mediante métodos asociados con la clase o subclases, como los métodos asociados con la clase PilaDeMesa. La única interfaz pública es a través de métodos; no hay variables de ejemplar accesibles públicamente. Al eliminar las variables de ejemplar accesibles públicamente, el lenguaje proporciona mecanismos que pueden usarse para ayudar a asegurar que ningún acoplamiento de datos se permita entre esta clase y otros componentes de software.

Se puede pensar en la evolución y modificación del software en términos de clientes usuarios y de subclases. Cuando un diseñador de clases anuncia las características públicas de una clase, está haciendo un contrato para proporcionar los servicios así descritos. Puede considerar e implementar libremente cambios en el diseño interno, en tanto permanezca sin cambio la interfaz pública (o tal vez sólo crezca). En forma similar, aunque quizá menos común y menos obvia, el diseñador de una clase está también especificando una interfaz para subclases. Cuando se cambian los detalles internos de una clase y dejan de operar las subclases, se presenta una fuente común y sutil de errores de software. Al separar, aunque sólo sea por convención, el programador establece las fronteras del cambio y la modificación aceptables. La flexibilidad para hacer cambios al código existente en forma segura es determinante para la mantención de sistemas de software grandes y duraderos.

El concepto de cliente de subclase tiene sentido en la medida en que el diseñador de una subclase y el diseñador de la clase original son dos individuos diferentes. Para el diseñador de cualquier clase, es una buena práctica de POO considerar la posibilidad de que en el futuro se puedan crear subclases de su clase y proporcionar la documentación y las conexiones de software para facilitar el proceso.

10.3. CONSTRUCCIÓN Y HERENCIA

Cuando una clase se subclasifica con propósitos de construcción, parte o todo el comportamiento de la superclase se oculta conceptualmente a los clientes. Sin embargo, el mecanismo puro de la herencia no oculta la visibilidad de los métodos heredados de la superclase.

Los diversos lenguajes OO difieren en el grado en el cual pueden usarse para reforzar la ocultación de la superclase. C++ es bastante completo en ese sentido ya que una clase puede heredar de manera pública o privada de otra clase.

10.4. CONTROL DE ACCESO Y VISIBILIDAD

C++ proporciona una gran cantidad de recursos destinados al control del acceso a la información, mediante las palabras claves public, private y protected. Los datos definidos como public están disponibles tanto para clientes de subclases como para usuarios. Los datos definidos como private son accesibles sólo por los ejemplares de la clase misma y no por la subclase o los clientes usuarios. Los datos definidos como protected son accesibles sólo dentro la clase y las subclases, es decir, por los clientes de las subclases pero no por los clientes usuarios. En ausencia de cualquier designación inicial por omisión los campos se consideran privados.

Un punto más sutil es que los especificadores controlan el acceso a los miembros, no a la visibilidad.

```
int i;

class A

{ private:

int i;
```

```
};

class B: public A

{ void f(); };

B::f()

{ i++;} // error, A::i es privada
```

Aquí aparece un error pues la función f intenta modificar la variable i, la cual es heredada de la clase A, aunque es inaccesible (porque está declarada como private). Si los modificadores de acceso controlaran la visibilidad, más que la accesibilidad, la variable i sería invisible y hubiera sido actualizada la i global.

Los modificadores de acceso definen las propiedades de una clase, no de los ejemplares. Así la noción de campos privados en C++ no corresponde exactamente al concepto desarrollado en la sección 11.5. En ella, los datos privados eran accesibles sólo a un objeto en sí, mientras que, en C++, los campos privados son accesibles a cualquier objeto de la misma clase. Esto es, se permite a un objeto manipular los miembros privados de otro ejemplar de la misma clase.

Considérese la situación presentada a continuación.

```
class Complejo

{ private:

double pr;

double pi;

public:

Complejo(double a, double b)

{ pr = a; pi = b; }

Complejo operator + (Complejo &x)

{ return Complejo(pr + x.pr, pi + x.pi);}

};
```

Aquí, los campos pr y pi, que representan las partes real e imaginaria de un número complejo, están marcados como privados. La operación binaria + se anula para dar como nuevo significado la suma de dos números complejos. No obstante la naturaleza privada de los campos pr y pi, la función del operador se permite para tener acceso a estos campos el argumento x, ya que éste es de la misma clase del receptor.

Los constructores y destructores, para el mismo ejemplo, como la función constructora Complejo, por lo general se declaran públicas. Declarar un constructor como protegido implica que sólo las subclases o las amigas pueden crear ejemplares de la clase, mientras que declararlo como privado restringe la creación sólo a amigas u otros ejemplares de la clase.

En parte, la forma flexible de la Ley de Demeter puede ponerse en vigor si se declaran todos los campos de datos como protegidos. La forma estricta corresponde a declarar tales campos como privados.

Un problema serio relacionado con el grado de control proporcionado por C++ es que la facilidad con la que se puede formar una clase depende, con frecuencia, de cuánto previó el diseñador de la clase original la posibilidad de una subclasificación. Ser extremadamente protectores (al declarar privada información que debería ser protegida) puede dificultar la subclasificación. Se originan problemas si el diseñador de la subclase no puede modificar la forma fuente de la clase original, por ejemplo, si el original se distribuye como parte de una biblioteca.

− Herencia privada: Las palabras clave public y private se usan también para introducir el nombre de una superclase en la definición de una clase. Cuando se usan de esta manera, la visibilidad de información de la superclase es alterada por el modificador. La subclase que hereda de manera pública de otra corresponde a la noción de herencia que hasta el momento se ha usado. Si una clase hereda de una manera privada, las características públicas de la superclase se reducen al nivel modificador.

Cuando una clase hereda de una manera no pública de otra clase, los ejemplares de la subclase no pueden asignarse a los identificadores del tipo de la superclase, lo que es posible con la herencia pública. Una forma fácil de recordar esta limitación es en términos de la relación es−un. La herencia pública es una afirmación evidente de que es válida la relación es−un y, por tanto, de que se puede usar un ejemplar de la subclase cuando se llama a la superclase. Por ejemplo, un Perro es−un Mamífero y, por ende, un Perro puede usarse en cualquier situación en la que se llame a un mamífero. Por otra parte, la herencia privada no mantiene la relación es−un, ya que los ejemplares de una clase que hereda de esa manera de una clase paterna no siempre pueden ser usados en lugar de la clase paterna.

− Funciones amigas: Una función amiga es simplemente una función (no un método) que se declara por medio del modificador friend en la declaración de una clase. A las funciones amigas se les permite leer y escribir los campos privados y protegidos dentro de un objeto.

Considerar el código presentado a continuación:

```
class Complejo

{ private:

double pr;

double pi;

public:

Complejo(double, double);

friend double abs(Complejo&);

};

Complejo::Complejo(double a, double b)

{ pr = a;

pi = b;
```

```
}

double abs(Complejo& x)

{ return sqrt(x.pr * x.pr + x. pi * x.pi); }
```

Aquí, los campos pr y pi de la estructura de datos que representa a los números complejos son declarados privados y, por ello, no son en general accesibles fuera de los métodos asociados con la clase. La función abs, que de paso sobrecarga una función del mismo nombre definida para valores de doble precisión, es simplemente una función. Sin embargo, al ser declarada como amiga de la clase compleja, se le permite tener acceso a todos los campos de la clase, aun a los privados.

Las funciones amigas son una herramienta poderosa, pero es fácil abusar de ellas. En particular, introducen exactamente un tipo de acoplamiento de datos (tratado anteriormente) dañino para el desarrollo de software reutilizable. Siempre que sea posible, deberá optarse por técnicas de encapsulamiento OO (como los métodos) que por funciones amigas. Sin embargo, hay ocasiones en que ninguna otra herramienta puede usarse, como cuando una función necesita tener acceso a la estructura interna de dos o más definiciones de clase. En estos casos, las funciones amigas son una abstracción.

− Miembros constantes: En C++, la palabra clave const se usa para indicar una cantidad que no cambia durante el tiempo de vida de un objeto. Las variables globales así declaradas se convierten en constantes globales. Las variables constantes (o variables de sólo lectura) que se declaran locales a un procedimiento son accesibles sólo dentro de él y no pueden modificarse excepto en el enunciado de inicialización que las crea.

Con frecuencia, los miembros de datos actúan como constantes, pero su valor inicial no puede determinarse sino hasta que se crea el objeto.

```
class Complejo

{ public:

const double pr;

const double pi;

Complejo(double, double);

};

Complejo::Complejo(double a, double b) : pr(a), pi(b)

{ /* instrucción vacía */ }
```

Aquí, los miembros de datos no deben alterarse una vez que se crea el número complejo. Anteriormente, a tales campos se les llamó inmutables. La palabra clave const proporciona otra forma más de creación de campos inmutables.

Aunque no se permite asignar a los miembros de datos constantes, en C++ pueden inicializarse mediante la sintaxis usada para invocar a los constructores de la clase paterna como parte del proceso constructor. En el ejemplo anterior, los campos pr y pi fueron declarados constantes, por lo cual no hay ningún peligro de hacerlos públicos, ya que no pueden modificarse. Para dar un valor inicial, el constructor parece invocar a pr y pi como si fueran superclases. Esta es la única manera (además de la inicialización en tiempo de compilación)

en la que pueden asignarse los miembros de datos constantes. Una vez que el cuerpo del constructor se empieza a ejecutar, no puede alterarse el valor del miembro de datos constante.

Los miembros de datos declarados como variables de referencia pueden inicializarse de la misma manera. La palabra clave const puede aplicarse también a miembros de funciones o procedimientos.

10

// descripción de interfaz para la clase Carta (carta.h)

ifndef cartah

define cartah // incluye este archivo sólo una vez

class Carta

{public:

int fijarPaloYCuenta(int,int); /* inicialización */

int palo();

int color();

int rango();

private:

int p; //pinta

int r; //

}

Definición de la clase derivada Triángulo

Definición de la clase derivada Cuadrado

void main()

{Cuadrado C;

Triangulo T(5.0f);

cout<<"Cuadrado: Longitud lado: "<<C.Longitud()<<endl;

cout<<"Perímetro del cuadrado: "<<C.Perimetro()<<endl;

cout<<"Área: "<<C.Area()<<endl<<endl;

cout<<"Triángulo: Longitud lado: "<<T.Longitud()<<endl;

```
cout<<"Perímetro del Triángulo:"<<T.Perimetro()<<endl;
cout<<"Área: "<<T.Area()<<endl<<endl;
Poligono P(6,7.0);
cout<<"Área: "<<P.Area()<<endl<<endl;
//Se invoca destructor de Triángulo: No hay=> Polígono
//Se invoca destructor de Cuadrado : Ok y éste invoca al de Polígono
}
class Triangulo: public Poligono
{public:
Triangulo(double r=1.0f);
double Perimetro();
//No es necesario definir Area(). Usará la de Polígono
//Usará el destructor de Polígono
};
//Implementación
Triangulo::Triangulo(double r): Poligono(3,r){}
double Triangulo::Perimetro()
{return 3*Longitud();}
```

Definición clase base Polígono

```
class Cuadrado: public Poligono
{public:
Cuadrado(double r=1.0f);
double Perimetro();
double Area();
~Cuadrado();
};
```

//Implementación

Cuadrado::Cuadrado(double r): Poligono(4,r){}

double Cuadrado::Perimetro()

{return 4*Longitud();}

Cuadrado::~Cuadrado()

{cout<<"Destruyendo cuadrado"<<endl;}

//Redefino el método Area().

double Cuadrado::Area()

{return Square(Longitud());}

//Clase Polígonos regulares

#include <iostream.h>

#include <math.h>

#define PI 3.1415926536

class Poligono

{private:

int n;

double s;

protected:

double Square(double);

public:

Poligono(int x,double y):n(x), s(y){};

int Lados();

double Longitud();

double Perimetro();

virtual double Area();

~Poligono();

};

//Implementación

double Poligono::Area()

{cout<<"calculando área....en super clase Polígono"<<endl;

return n*Square(s)/4.0/tan((double)(n−2)/(2*n)*PI);}

double Poligono::Square(double)

{return Longitud()*Longitud();}

int Poligono::Lados()

{return (n);}

double Poligono::Longitud()

{return s;}

double Poligono::Perimetro()

{return n*s;}

Poligono::~Poligono()

{cout<<"Destruyendo polígono"<<endl;}

class Triángulo

{private:

double s;

public:

Triangulo(double);

double Perimetro();

double Area();

}

//Implementación

Triangulo::Triangulo (LongLados r)

{s=r;}

double Triangulo::Perimetro()

{return 3*s;}

double Triangulo::Area()

{return (s*s)/2;}

Polígonos Regulares: PL

Hexágono Cuadrado Triángulo

Medio de transporte

Avión Vehículo

Puerta

Camión Automóvil Motor

Capota

Llanta

...

Stack S;

S.Push(3);

:

Stack P;

:

int x=P.Pop();

...

altura

ancho

...

altura

ancho

...

índice

ubicaciónCursor

...

class VistaDeCarta

{public:

const int AnchoCarta = 65;

const int AltoCarta = 75;

Carta *carta();

void dibujar();

void borrar();

int anverso();

void reverso();

int incluye(int,int);

int x();

int y();

void moverA(int,int);

private:

Carta *laCarta;

int cara;

int ubicx;

int ubicy;

}

endif