El lenguaje de programación C++

Omer Manuel Salcedo Galván
Universidad Tecnológica de Bolívar, Colombia
osalcedo@unitecnologica.edu.co

Apuntes de las clases de la asignatura de
Programación II.

Cartagena de Indias D.T. y C.
1P2011.

AGRADECIMIENTOS

Quisiese agradecer públicamente por la realización de estos apuntes :

A Dios como Señor y dador de nuestras vidas.

A mis familiares especialmente Kattia, mi señora esposa por su apoyo incondicional.

Al director del programa de Ingeniería de sistemas de la Universidad Tecnológica de Bolívar Msc. Moisés Quintana por sus valiosas recomendaciones en el proceso enseñanza – aprendizaje.

A mis compañeros de clases; motivo principal en el desarrollo de estos apuntes.

Y en especial a todas las personas que nos enseñan a nunca perder las esperanzas.

Gracias Totales

Omer Salcedo

Índice de contenido

Generalidades

Introducción.

El presente Capitulo tiene como objeto contextualizar al lector acerca del lenguaje de programación C++, tocando algunos aspectos tales como su Historia, Importancia, y los elementos básicos del lenguaje.

Un poco de Historia.

El lenguaje de programación C++ tuvo su nacimiento de la mano de Bjarne Stroustrup en los laboratorios Bell de AT&T en 1980 donde se desarrolló el Sistema Operativo UNIX.

C++ inicialmente se concibió como un precompilador de aspectos muy similares al preprocesador de C, el cual convertía su propio código fuente (C++) a código fuente para lenguaje C. para luego ser compilado por un compilador de C estándar.

Desde 1980 hasta 1990 han habido varios compiladores de C++ (Borland, Apple, Mingw, Microsoft Visual, Sun Studio, Etc) En la cual las características fundamentales del lenguaje están disponibles en todos los compiladores; Pero algunas otras características propias del compilador y de sus fabricantes tales como el uso de librerías (conio de Borland), diferencias de sintaxis (Pro C/C++ de Oracle, Objective C de Apple, Etc.) entre otras, genera en algunos casos inconvenientes a la hora de llevar un código fuente de un compilador a otro.

Para resolver el problema de la portabilidad entre compiladores, en 1989 se formó un comité ANSI lo que permitió la estandarización del lenguaje a nivel americano y a mediados del año 1998 su sintaxis fue estandarizada bajo la norma ISO a nivel internacional. Para 1998 el lenguaje ANSI/ISO C++ fue aprobado y por ende todos los compiladores modernos cumplen las especificaciones exigidas por el estándar.

¿Porque C++ y no otro lenguaje?

Para responder esta pregunta, debemos tener en cuenta las principales ventajas que presenta el lenguaje C++[1]:

- Difusión: al ser uno de los lenguajes más empleados en la actualidad, posee un gran número de usuarios y existe una gran cantidad de libros, cursos, páginas web, etc. dedicados a él.
- Versatilidad: C++ es un lenguaje de propósito general, por lo que se puede emplear para resolver cualquier tipo de problema, desde aplicaciones de consola hasta sistemas críticos incluyendo Sistemas Operativos.
- Portabilidad: el lenguaje está estandarizado y un mismo código fuente se puede compilar en diversas plataformas.
- Eficiencia: C++ es uno de los lenguajes más rápidos en cuanto ejecución.

1 Lujan, Sergio. C++ Paso a paso. [En Linea] <http://gplsi.dlsi.ua.es/~slujan/materiales/cpp-muestra.pdf> [citado en 2011/01/26]

- Herramientas: existe una gran cantidad de compiladores, depuradores, librerías, etc.
- Su sintaxis ha sido tomada como referencia para otros lenguajes tales como php, java, C# entre otros. Esta ventaja permite abarcar estos lenguajes reduciendo significativamente su curva de aprendizaje.

C++ es un lenguaje de programación multi-paradigma, lo cual permite desarrollar aplicaciones fundamentadas en los conceptos de la programación estructurada así como aplicaciones fundamentadas con el paradigma de la programación orientada a objetos, por lo que también se considera C++ como un lenguaje híbrido.

Además desde el punto de vista del paradigma de la programación orientada a objeto es mas visible algunos de sus conceptos tales como la Herencia y el Polimorfismo.

Entrando un poco en materia

Mas allá del Hola mundo

A través de un código fuente desarrollado en C++ que tiene como objeto obtener el área de un circulo conocido su radio se expondrán algunas de las características básicas que tiene este lenguaje de programación.

El código fuente se muestra a continuación:

```
//archivo: areacirculo.cpp
//autor: Omer Salcedo <osalcedo@unitecnologica.edu.co>
//version: 1.0

#include <iostream>
#define PI 3.141592654

using namespace std;

int main()
{
	float Area, radio;

	cout<<"r=";
	cin>>radio;

	Area=PI*radio*radio;

	cout<<"area="<<Area<<endl;

	return 0;
}
```

Desde la linea 1 hasta la línea 3 son *lineas de comentarios*. Una linea de comentario inicia anteponiendo la barra inclinada 2 veces // esta instrucción le indica al compilador que ignore todo cadena de símbolos posteriores a // hasta el fin de la línea.

En la linea 5 aparece la instrucción #include <iostream> esta es una *directiva del preporcesador*. El preprocesador tiene como objeto preparar el código fuente para que este sea compilado. Las directiva del preprocesador son un conjunto de instrucciones que permiten al preprocesador, en el momento de analiza el código fuente antes de la fase de compilación real, realiza las sustituciones de macros, inclusión de funciones, objetos, definiciones desde otros archivos. También es el responsable de la eliminación de los comentarios. El objetivo de la directiva #include es insertar archivos externos dentro de nuestro archivo de código fuente. Estos archivos son conocidos como archivos incluidos, archivos de cabecera o "headers". Que en nuestro caso el archivo de cabecera es iostream.h que contiene un conjunto de objetos, macros y funciones que permiten el manejo eficiente de la entrada y salida haciendo uso de los flujos de bytes (streams) tales como los objetos **cin** (*Console in*) para entrada y **cout** (*console out)* para la salida que en por defecto son el teclado y la pantalla respectivamente.

En la línea 6 aparece otra directiva del preprocesador #define PI 3.141592654. La directiva #define, sirve para definir macros. Las macros suministran un sistema para la sustitución de palabras, con y sin parámetros. En este caso se esta definiendo la macro PI con valor 3,141592654.

En la línea 8 la instrucción using namespace std; tiene como objeto evitar re definir objetos, funciones, macros, etc, previamente definidas al ponerles igual nombre, para evitar esa ambigüedad se creó la figura de los *espacios de nombres o namespace*. En este caso hay un espacio de nombres llamado std, en la cual se incluyen las definiciones de todas las funciones y clases que conforman la librería estándar de C++.

En la línea 11 entra en escena la función main, esta es la función principal y punto inicial para la ejecución de un programa escrito en C++, es análogo a *inicio* en algoritmos representados ya sea en seudocódigos o en diagramas de flujos. Desde el punto de vista sintáctico main es una función la cual debe cumplir la siguiente estructura:

```
tipo_valor_retorno nombre_funcion(lista de parametros){
        //sentencias
        return valor_retorno;
}
```

En este caso de main tiene como valor de retorno int, que en C++ representa el tipo de dato entero (int de integer) y la función no presenta parámetros, claro está, que esta no es la única configuración de la función main, ya que ésta puede no retornar valor y se especifica haciendo uso de la la palabra reservada void así como también puede recibir 2 parámetros (int argc, char** argv) el primero se utiliza para contar los argumentos que el archivo ejecutable puede recibir y el segundo los valores de los argumentos respectivamente de superar exitosamente el proceso de compilación, enlace y constricción realizado por el compilador.

En la línea 12 la llave { abre el cuerpo de la función.

En la línea 13 se definen do variables de nombres radio y Area de tipo de datos de coma flotante (float) de ámbito local para la función main (ya que están definidas dentro del bloque encerrado por llaves que delimitan el alcance de la función main. Toda variable declarada por fuera de main y no definida dentro de un bloque de otra función diferente se llama variable de ámbito global o sutilmente variables

globales.

En la línea 20 la instrucción Area=PI*radio*radio; indica al compilador que debe hacer la multiplicación (el operador binario * representa la multiplicación) del valor presente en la variable radio leída desde el teclado a través del uso del objeto **cin** (línea 16) consigo misma y luego con el valor de la macro PI, el valor del resultado se almacena en la variable Area que es mostrada por la pantalla a través del uso del objeto **cout** (línea 20).

Finalmente en la línea 24 la función main lleva al punto de devolución del valor de retorno lo que permite el fin de la ejecución del programa haciendo uso de la instrucción return.

Finalmente en la línea 25 la llave } cierra el cuerpo de la función.

Tipos de variables

C++ presenta en su sintaxis algunos tipos de variables fundamentales, a saber: void, char, int, float y double, C++ incluye también el tipo bool. También existen ciertos modificadores, que permiten ajustar ligeramente ciertas propiedades de cada tipo; los modificadores pueden ser: short, long, signed y unsigned. También C++ soporta los tipos enumerados, enum.

Palabras reservadas.

A continuación se muestran el conjunto de palabras reservadas del lenguaje de programación C++, vale la pena recordar que estas palabras son de uso exclusivo del lenguaje y no pueden ser utilizada para otros propósitos, por ejemplo como nombre de variables:

- asm
- auto
- break
- case
- catch
- char
- class
- const
- continue
- default
- delete
- do
- double
- else
- enum
- extern
- float
- for
- friend
- goto
- if
- inline
- int
- long
- new
- operator
- private
- protected
- public
- register
- return
- short
- signed
- sizeof
- static
- struct
- switch
- template
- this
- throw
- try
- typedef
- union
- unsigned
- virtual
- void
- volatile
- while

Estructuras básicas del lenguaje.

A continuación se exponen algunas estructuras sintácticas del lenguaje de programación C++[2], vistas en el curso de programación 1 y expongo a manera de recordatorio.

Declaración de variables

[signed|unsigned|long] tipo_datos <identificador>[,<identificador2>[,<identificador3>]...];

Ejemplos:
signed char cuenta, cuenta2, total;
unsigned char letras;
char caracter, inicial, respuesta;

Prototipo de funciones

[extern|static] <tipo_valor_retorno> [<modificadores>] <identificador>(<lista_parámetros>);

Ejemplos:
int Mayor(int a, int b);
int mayor(int,int);

Declaración de funciones

```
[extern|static] <tipo_valor_retorno> [modificadores] <identificador>(<lista_parámetros>)
{
  [sentencias]
}
```

Ejemplos:

```
int Mayor(int a, int b)
{
  if(a > b)
        return a;
  else
        return b;
}
```

2 Pozo, Salvador. C++ con clase [En línea] <http://c.conclase.net/> [Citado en 2011/01/26].

Arreglos

<tipo> <identificador>[<núm_elemen>][[<núm_elemen>]...];

Ejemplos:
int Vector[10];
int Tabla[10][10];
char DimensionN[4][15][6][8][11];

El paradigma de la Programación Orientada a Objetos.

Introducción.

El presente capítulo tiene como objetivo dar al lector los conceptos fundamentales del paradigma de la Programación Orientada a Objetos (POO), partiendo de sus orígenes teniendo en cuenta la generación de los lenguajes de programación y su respectiva evolución. Se explorarán los fundamentos del modelo de objetos, el análisis orientado a objetos y el diseño orientado a objetos.

La evolución del modelo de objeto

El análisis y diseño orientado a objeto no fue generado espontáneamente, pero si, de las cenizas de incontables fallas en los proyectos de software desarrollados por paradigmas o metodologías anteriores. Este paradigma no es una desviación radical de los enfoques anteriores, de hecho, se fundamenta con sus mejores ideas.

Por otro lado, se examinará la evolución de las herramientas de desarrollo de software para así comprender los orígenes y la aparición de la metodología orientada a objetos.

Las generaciones de los lenguajes de programación

A continuación se muestran la clasificación de los mas populares lenguajes de programación en generaciones de acuerdo a las características propias de los lenguajes.

- Primera generación (1954-1958)
  - FORTRAN I: Expresiones matemáticas.
  - ALGOL 58: Expresiones matemáticas.
  - Flowmatic: Expresiones matemáticas.
  - IPL V: Expresiones matemáticas.
- Segunda generación (1959-1961)
  - FORTRAN II: Subrutinas, compilación separada.
  - ALGOL 60: Estructura en bloques, tipos de datos.
  - COBOL: descripción de datos, manejo de archivos.
  - Lisp: procesamiento de listas, apuntadores, recolección de basura (*garbage collection*).
- Tercera generación (1962-1970)
  - PL/1: FORTRAN + ALGOL + COBOL.
  - ALGOL 68: Sucesor riguroso de ALGOL 60.
  - Pascal: Sucesor simple de ALGOL 60.

- Simula: Clases y abstracción de datos.

- La brecha generacional (1970-1980)

  En esta época se inventaron nuevos lenguajes de programación, pero débilmente soportados. Dentro de ellos vale la pena destacar:

  - C: Eficiente, para desarrollos pequeños.
  - FORTRAN 77: Primer lenguaje de programación estandarizado por la ANSI.

- Boom de la programación orientada a objetos (1980-1990)
  - Smalltalk 80: Lenguaje orientado a objeto puro.
  - C++: Derivado de C y Simula.
  - Ada83: tipado fuerte; fuertemente influenciado por Pascal.
  - Eifell: Derivado de Ada y Simula.

- Surgimientos de los frameworks (1990-hoy)
  - Visual Basic: Desarrollo fácil de interfaz gráfica de usuarios para aplicaciones sobre Windows.
  - Java: Sucesor de Oak; diseñado para que sus aplicaciones sean portables.
  - Python: Lenguaje de script orientado a objetos.
  - J2EE: Framework basado en Java para desarrollos empresariales.
  - .NET: Framework basado en objetos de Microsoft ™

En lo sucesivo de las generaciones, los diferentes mecanismos de abstracción que soportan cada lenguaje cambian. Los lenguajes de primera generación fueron usados principalmente para desarrollar aplicaciones científicas e ingenieriles. Lenguajes como FORTRAN I fue desarrollado para ayudar al programador a escribir fórmulas matemáticas liberándolo de las complejidades del lenguaje ensamblador o código maquina.

En los lenguajes de segunda generación se enfatizaban en la abstracción algorítmica, para aquella época, las maquinas comenzaron a tener mayor potencia de computo y en la economía de la industria de la computación significa que una mayor clase de problemas podían ser automatizados, especialmente los problemas empresariales, tales como administrar la nómina de una empresa, obtener el balance financiero de esta y de paso imprimirla.

Para finales de 1960, con la llegada de los transistores y los circuitos integrados el costo de las maquinas de computo a nivel de hardware bajaron dramáticamente, reduciendo su tamaño pero su potencia de computo tuvo un crecimiento exponencial, una serie de grandes problemas podía ser resuelto, pero dicha solución demandaba la manipulación de mas clases de datos. Entre los lenguajes de programación de tercera generación como ALGOL 60, y posteriormente, Pascal facilitaron esa abstracción de datos que los problemas de la época demandaban. Ahora un programador podía describir, el tipo de datos y dejar al lenguaje que tomar la mejor decisión a la hora de la construcción del software.

En la década 1970 hubo un frenesí en investigaciones referentes a lenguajes de programación,

resultando la creación de, literalmente, miles de lenguajes, que a su vez permitían desarrollar aplicaciones que eran inadecuadas con el uso de lenguajes de programación anteriores ya que fueron desarrollados con el fin de hacerle frente a esas limitaciones. En la actualidad muy pocos sobreviven.

Los lenguajes de programación orientada a objetos tienes un gran interés debido a que ofrecen el mejor soporte para la descomposición del software. El número de lenguajes que soportar este paradigma tuvo su máxima expansión o boom entre la década de 1980 y los comienzos de la década de 1990. Desde 1990 pocos lenguajes han emergido con el respaldo de los fabricantes de herramientas de programación comerciales (por ejemplo, java, C++). El uso emergente de los frameworks (J2EE, .NET) ofrecen una cantidad de componentes y servicios que simplifican las tareas de programación comunes y corrientes, obteniéndose una alta productividad y demostrando la difícil promesa del reuso de componentes.

La evolución de las metodologías de programación

En un principio los lenguajes de programación presente en la primera y segunda generación se caracterizaban por utilizar la metodología de la programación imperativa. Su origen es la propia arquitectura de Von Neumann, que consta de una secuencia de celdas (memoria) en las cuales se pueden guardar datos e instrucciones, y de un procesador capaz de ejecutar de manera secuencial una serie de operaciones (ó comandos) principalmente aritméticas y booleanas. En general, un lenguaje imperativo ofrece al programador conceptos que se traducen de forma natural al modelo de la máquina. El programador tiene que traducir la solución abstracta del problema a términos muy primitivos, cercanos a la máquina, por lo que los programas son más "comprensibles" para la máquina que para el hombre. Esto es una desventaja para el programador que hace que sea sumamente complicado construir código en lenguaje imperativo. Lo bueno de este lenguaje es que es tan cercano al lenguaje de la máquina que la eficiencia en la ejecución es altísima[3].

Para los lenguajes de tercera generación soportaban la metodología de la programación procedimental Los matemáticos resuelven problemas usando el concepto de función, la cual se define como una relación matemática la cual, de acuerdo a los datos que recibe como entrada, se obtiene un resultado como salida. Sabiendo cómo evaluar una función, usando la computadora, se pueden resolver automáticamente muchos problemas. Este fue el pensamiento que llevó a la creación de los lenguajes de programación funcionales. Además se aprovechó la posibilidad que tienen las funciones para manipular datos simbólicos, y no solamente numéricos, y la propiedad de las funciones que les permite componer, creando de esta manera, la oportunidad para resolver problemas complejos a partir de las soluciones a otros más sencillos. También se incluyó la posibilidad de definir funciones recursivamente. Un lenguaje funcional ofrece conceptos que son muy entendibles y relativamente fáciles de manejar. El lenguaje funcional más antiguo y popular es LISP, diseñado por McCarthy en la segunda mitad de los años 50. Se usa principalmente en Inteligencia Artificial. En los 80 se añadió a los lenguajes funcionales la tipificación y algunos conceptos modernos de modularización y polimorfismo. Programar en un lenguaje funcional significa construir funciones a partir de las ya existentes. Por lo tanto es importante conocer y comprender bien las funciones que conforman la base del lenguaje, así como las que ya fueron definidas previamente. De esta manera se pueden ir construyendo aplicaciones cada vez más complejas. La desventaja es que está alejado del modelo de la máquina de Von Neumann y, por lo tanto, la eficiencia de ejecución de los intérpretes de lenguajes funcionales es peor que la ejecución de los programas imperativos precompilados[4].

3 RIVERO, Jessica. Historia de la Programación. [En línea] <http://www.it.uc3m.es/jvillena/irc/practicas/estudios/Lenguajes_de_Programacion.pdf>[Citado en 2011/01/28]

4 RIVERO, Jessica. Historia de la Programación. [En línea]

Entre los lenguajes de programación presentes entre la tercera generación y la brecha generacional estos soportaban la metodología de la programación modular, la cual, podían agrupar funcionalidades comunes dentro de módulos. Bajo esta metodología un programa puede no esta compuesto por una única parte sino que puede estar dividido en varias partes que interactuan a través de llamadas de procedimientos. Cada módulo tienes sus propios datos, lo que permite a cada módulo administrar un estado interno que puede ser modificado a través de procedimientos definidos dentro del mismo módulo. Como ejemplos son los programa que administra su información a través de estructuras de datos, existen varias estructuras de datos tales como arreglos, listas, grafos, arboles, pilas, colas, conjuntos y otras; los cuales se caracterizan por tener su propia estructura y sus métodos de acceso[5]. Algunas desventajas de este paradigma son: Creación y destrucción explícita, datos y operaciones desacoplados y poca representación.

El paradigma de la programación orientada a objetos

El paradigma de la programación orientada a objetos resuelve varios de los problemas recientemente mencionados, en contraste con otras metodologías, se fundamenta en una red de objetos que interactúan entre si cada uno conservando su propio estado.

Tipos abstractos de datos

Comprendiendo el sistema de cosas

Cuando se emprende un proyecto de desarrollo de software. El producto final tiene como objeto resolver determinado problema, que generalmente, son de la vida real e incluso cotidiana. Como es sabido, los problemas del mundo real pueden ser nebulosos el cual se debe comprender y entenderlo con el fin de obtener los detalles y clasificarlos de acuerdo a su importancia en la resolución del problema, obteniéndose su propia vista abstracta, o *modelo*, Este proceso de modelado se llama *abstracción*.

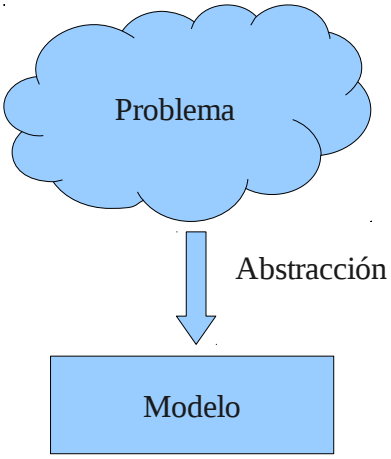


Ilustración 1: Proceso de Abstracción

<http://www.it.uc3m.es/jvillena/irc/practicas/estudios/Lenguajes_de_Programacion.pdf> [Citado en 2011/01/28]

5 MÜLLER, Peter. Introduction to Object-Oriented Programming Using C++. [En línea] <http://www.desy.de/gna/html/cc/Tutorial/tutorial.html> [Citado en 2011/01/30]

El modelo define una vista abstracta del problema, lo cual implica que la fidelidad del modelo respecto al problema real depende fuertemente en el enfoque de los aspectos propios del problema el cual presenta el modelo. Esos aspectos son conocidos como las propiedades del problema, esas propiedades incluyen:

1. Los datos que pueden ser afectados
2. Las operaciones que han sido identificadas

Como ejemplo considere un administrador de empleados de una institución. Las directivas de la institución donde usted trabaja se acercan a usted y le piden que desarrolle un programa que permita administrar los datos de sus empleados. Bien, como el problema no ha sido especifico, por ejemplo, ¿Que información de los empleados considera las directivas necesaria? ¿Que tareas deben permitir realizar? Se sabe que los empleados son personas reales que se caracterizan por tener una serie de propiedades, a continuación se mencionarán algunas:

- Nombre
- estatura
- fecha de nacimiento
- cargo
- color de cabello
- sexo
- hobby
- Número de Identificación
- Profesión

Ciertamente, algunas de las propiedades mencionadas, no son necesarias para resolver el problema de la administración. En el momento que se crea un modelo debe asegurarse que en la selección de las propiedades éstas sean necesarias para cumplir con los requerimientos, que en el ejemplo, solicita la administración y en termino general el cliente. Estas propiedades se conocen como *datos* o *atributos*, estos datos o atributos permiten describir, para el ejemplo, personas reales a través de la abstracción de empleados.

Por supuesto no solo basta con la pura descripción, también deben existir operaciones definidas por la directivas de la institución, la cual puedan manipular dichos empleados abstractos. Tales como crear empleados, editar los datos de algún empleado previamente creado o eliminar los datos de un empleado, así como validación de sus datos. Estas operaciones se conocen como *métodos*.

Para resumir, la abstracción permite definir de un problema real varias entidades con sus datos y operaciones. En consecuencia, esas entidades combinan datos y operaciones las cuales no están desacopladas.

Propiedades de los tipos abstractos de datos

Con base al proceso de abstracción se obtienen entidades bien definidas con sus propiedades, estas entidades definen la estructura de datos de un conjunto de elementos.

Las estructura de datos pueden ser accedidas por operaciones bien definidas el conjunto de estas operaciones se llaman interfaces, y estas son exportados por la entidad. Una entidad con las propiedades descritas anteriormente se conocen como Tipo Abstracto de Datos TAD (*ADT - Abstract Data Type*).

La siguiente figura muestra un TAD que consiste un conjunto de atributos y métodos. Solo los métodos son accedidos por fuera de la TAD y definen sus interfaces.

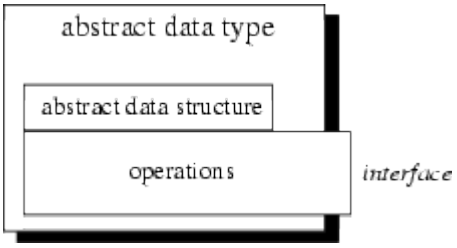


Ilustración 2: Un Tipo Abstracto de Datos

Tipos abstractos de datos y la Orientación a objetos

Los TAD permiten la creación de instancias con propiedades bien definidas y comportamientos. En la orientación a objetos los TAD se refieren a las ***clases***. Por lo tanto una *clase* define las propiedades de los objetos los cuales son instanciados en un ambiente orientado a objetos.

Los tipos abstractos de datos definen funcionalidades poniendo principal énfasis en los datos involucrados, sus estructuras, sus métodos así como axiomas y precondiciones. Consecuentemente, la programación orientada a objetos es equivalente a decir " Programación con TAD": Combinando las funcionalidades de diferentes tipos abstractos de datos para resolver un problema, por lo tanto, las instancias (objetos) de los TAD son dinámicamente creados, usados y destruidos.

Conceptos orientados a objetos

En esta sesión se definirán los principales conceptos de la programación orientada a objetos.

Clases

Una clase es la implementación de un tipo de datos abstractos (TAD). En cual se definen los atributos y los métodos que implementan la estructura de datos y operaciones del TDA respectivamente. Las instancias de las clases son llamados objetos . En consecuencia, las clases definen las propiedades y el comportamiento de conjuntos de objetos.

Objetos

Un Objeto es la instancia de una clase, que es identificada por su nombre y define un estado el cual es representado por los valores de su atributo en un tiempo determinado.

El estado de un objeto cambia de acuerdo a la aplicación de sus métodos, al referirse a los posibles cambios de estado de un objeto, estamos hablando de su comportamiento.

Comportamiento

El comportamiento de un objeto es definido por el conjunto de métodos los cuales pueden ser aplicados en él.

Mensajes

Un mensaje es la solicitud realizada a un objeto para que éste invoque alguno de sus métodos, por lo tanto un mensaje contiene:

- El nombre del método
- Los argumentos del método.

En consecuencia, la invocación de un método es la reacción causada por la recepción de un mensaje; esto solo es posible si el método es conocido por el objeto.

Aterrizando los conceptos

Con base a la abstracción de la entidad Empleado, observaremos como se identifican cada uno de los conceptos vistos en esta sesión.

Clase:

Empleado

Atributos:

nro_id: int

Nombre : char*

Estatura: float

Fecha_nacimiento: *Date*

Cargo: char*

Profesión: char*

Metodos:

getNroId(void): in

setProfesion(const char*):void

getEdad():int

getCargo():char*

setCargo(const char*): void

Objetos:

Empleado emp, q;

Mensajes:

emp.getNombre(“Adaluz”);

emp.setProfesion(“Ingeniero de Sistemas”);

q.getProfesion(emp.getProfesion());

Referencias

BOOCH Grady et al. Object Oriented analisys and Design with Applications. 3 Ed., 2007. Addison-Wesley. 691 pp. ISBN 0-201-89551-X.

Clases: Definición, constructores y destructores.

Introducción

El presente capitulo tiene como objeto dar al lector los conceptos relacionados con la implementación de los tipos abstractos de datos que, en el lenguaje de programación C++, se conocen como clases (*class*). Se explorarán los conceptos de atributos, operaciones, modificadores de acceso, constructores y destructores.

Definiciones

Salvador Pozo[6] define los conceptos claves en la programación orientada a objetos, con el fin de contextualizar al lector acerca de este paradigma de programación:

POO: Siglas de "Programación Orientada a Objetos". En inglés se escribe al revés "OOP". La idea básica de este paradigma de programación es agrupar los datos y los procedimientos para manejarlos en una única entidad: el objeto. Cada programa es un objeto, que a su vez está formado de objetos que se relacionan entre ellos.

Objeto: Un objeto es una unidad que engloba en sí mismo datos y procedimientos necesarios para el tratamiento de esos datos.

Mensaje: El mensaje es el modo en que se comunican e interrelacionan los objetos entre si.

Método: Se trata de otro concepto de POO, los mensajes que lleguen a un objeto se procesarán ejecutando un determinado método de ese objeto.

Clase: Una clase se puede considerar como un patrón para construir objetos.

Interfaz: Se trata de la parte del objeto que es visible para el resto de los objetos. Es decir, que es el conjunto de métodos (y a veces datos) de que dispone un objeto para comunicarse con él.

Herencia: Es la capacidad de crear nuevas clases basándose en clases previamente definidas, de las que se aprovechan ciertos datos y métodos, se desechan otros y se añaden nuevos.

Jerarquía: Definición del orden de subordinación de un sistema clases.

Polimorfismo: Propiedad según la cual un mismo objeto puede considerarse como perteneciente a distintas clases.

Entrando en materia

A continuación se definen las sintaxis en C++ de los conceptos esenciales de clases en la POO usando el lenguaje de programación C++

Declaración de una clase:

La sintaxis, para la declaración de clases, se expresa a continuación:

6 Pozo, Salvador. C++ con clase [En línea] <http://c.conclase.net/> [Citado en 2011/02/07].

class *<identificador de clase> [<:lista de clases base>] {*

<lista de miembros>

} [<lista de identificadores de objetos>];

La lista de miembros será en general una lista de operaciones (métodos) y atributos (datos).

Los atributos se declaran del mismo modo en que lo hacíamos hasta ahora, salvo que no pueden ser inicializados.

Las operaciones pueden ser simplemente declaraciones de prototipos, que se deben definir aparte de la clase pueden ser también definiciones.

Cuando se definen fuera de la clase se debe usar el operador de ámbito "::".

El siguiente ejemplo muestra la declaración de la clase entero:

```
class Entero{
    private:
        //atributos
        int valor;
    public:
        //métodos
        Entero sumar(Entero);
        Entero restar(Entero);
        Entero invAd();
        void setValor(int);
        int getValor();
 };
void Entero::setValor(int v)
{
   valor=v;
}
int Entero::getValor(){ return valor;}
Entero Entero::sumar(Entero a)
{
     Entero suma;
     suma.setValor(valor+a.getValor());
     return suma;
```

```
}


Entero Entero::restar(Entero a)
{
    Entero resta;
    return sumar(a.invAd());
}
Entero Entero::invAd()
{
    Entero inv;
    inv.setValor(-1*valor);
    return inv;
}
```

La clase Entero tiene un miembro de tipo de datos o atributo llamado *valor* y cinco operaciones, una para leerlo (*setValor*), uno para mostrarlo (*getValor*) y el resto para efectuar operaciones con él (*sumar*, *restar* e *invAd*). Nótese que su sintaxis es similar a la del tipo de datos definidos por el usuario *struct*.

Modificadores de acceso

Los modificadores son elementos del lenguaje que se colocan delante de la definición de variables locales, datos miembro, métodos o clases y que alteran o condicionan el significado del elemento en la cual cada miembro puede tener diferentes niveles de acceso. C++ tiene 3 modificadores de acceso para los miembros de las clases a saber:

public: Cualquier miembro público de una clase es accesible desde cualquier parte donde sea accesible el propio objeto.

private: Los miembros privados de una clase sólo son accesibles por los propios miembros de la clase y en general por objetos de la misma clase, pero no desde funciones externas o desde funciones de clases derivadas.

protected: Con respecto a las funciones externas, es equivalente al acceso privado, pero con respecto a las clases derivadas se comporta como público.

Cada una de éstas palabras, seguidas de ":", da comienzo a una sección, que terminará cuando se inicie la sección siguiente o cuando termine la declaración de la clase. Es posible tener varias secciones de cada tipo dentro de una clase.

Si no se especifica nada, por defecto, los miembros de una clase son privados.

Constructores y destructores

En C++, las clases contienen dos categorías de métodos especiales los cuales son esenciales para el

buen funcionamiento de la clase. Estos son los constructores y los destructores.

Los constructores

Los constructores son métodos especiales que sirven para inicializar un objeto de una determinada clase al mismo tiempo que se declara.

Los constructores son especiales por varios motivos:

1. Tienen el mismo nombre que la clase a la que pertenecen.
2. No tienen tipo de retorno, y por lo tanto no retornan ningún valor.
3. No pueden ser heredados.
4. Por último, deben ser públicos, no tendría ningún sentido declarar un constructor como privado, ya que siempre se usan desde el exterior de la clase, ni tampoco como protegido, ya que no puede ser heredado.

Su sintaxis es:

```
class <identificador de clase> {
  public:
    <identificador de clase>(<lista de parámetros>);
    ...
};
<identificador de clase>::<identificador de clase>(<lista de parámetros>) [: <lista de constructores>] {
        <código del constructor>
}
```

La clase Entero, al añadir un constructor, quedaría así:

```
class Entero{
    private:
        //atributos
        int valor;
    public:
        //constructor
        Entero(int);
        //metodos
        Entero sumar(Entero);
        Entero restar(Entero);
        Entero invAd();
```

```
        void setValor(int);
        int getValor();
};
Entero::Entero(int v){
        valor=v;
}
```

…

Al desarrollar el constructor este permite realizar instancias de objetos de tipo Entero siguiendo la siguiente sintaxis:

```
int main (){
  Entero a(6); //crea un Objeto de tipo entero en el cual su valor inicial es 6
}
```

Constructor por defecto

Es un constructor sin parámetros inicialmente creado por el compilador de forma automática cuando a la clase no se le define un constructor de manera explícita.

Cuando se crean Objetos de forma local los atributos de estos tendrías valores al azar trayendo "basura" que hubiese en la memoria asignada al objeto. En cambio, si se tratan de objetos globales, dichos atributos se inicializarían en cero.

Para declarar objetos usando el constructor por defecto o un constructor que hayamos declarado sin parámetros no se debe usar el paréntesis.

Ejemplo:

```
Entero gl; //Crea un objeto de ámbito global de tipo entero el cual su valor inicial es cero
int main (){
  Entero a; //crea un Objeto  de ámbito local de tipo entero el cual su valor inicial es desconocido
}
```

Inicialización de objetos

En C++ las variables de tipos básicos como int, char o float son objetos. En C++ cualquier variable (u objeto) tiene, al menos un constructor, el constructor por defecto, incluso aquellos que son de un tipo básico.

Sólo los constructores de las clases admiten inicializadores. Cada inicializador consiste en el nombre del atributo a inicializar, seguida de la expresión que se usará para inicializarla entre paréntesis. Los inicializadores se añadirán a continuación del paréntesis cerrado que encierra a los parámetros del constructor, antes del cuerpo del constructor y separado del paréntesis por dos puntos ":".

Por ejemplo la implementación del constructor:

```
Entero::Entero(int v){
        valor=v;
}
```

Quedaría así:

```
Entero::Entero(int v):valor(v) { }
```

Sobrecarga de constructores

Los constructores son funciones, y por ende en C++, también pueden definirse varios funciones con el mismo nombre (en nuestro caso constructores) para cada clase, es decir, el constructor puede sobrecargarse. La única limitación (como en todos los casos de sobrecarga) es que no pueden declararse varios constructores con el mismo número y el mismo tipo de argumentos.

Por ejemplo, es posible agregar un constructor adicional a la clase Entero que simule el constructor por defecto:

```
class Entero{
    private:
        //atributos
        int valor;
    public:
        //constructores
        Entero(int);
        Entero();
        //métodos
        Entero sumar(Entero);
        Entero restar(Entero);
        Entero invAd();
        void setValor(int);
        int getValor();
};
Entero::Entero(int v){
        valor=v;
}
Entero::Entero():valor(0){}
//inicializa en cero el valor del entero en caso de que no se use el otro constructor
```

Constructores con argumentos por defecto

También pueden asignarse valores por defecto a los argumentos del constructor, de este modo se reduce significativamente el número de constructores necesarios. La asignación se realiza en la definición del constructor

El ejemplo muestra la clase Entero haciendo uso de un constructor el cual tiene su único argumento por defecto y este valor es igual a cero.

```
class Entero{
    private:
        //atributos
        int valor;
    public:
        //constructor
        Entero(int=0);
        //métodos
        Entero sumar(Entero);
        Entero restar(Entero);
        Entero invAd();
        void setValor(int);
        int getValor();
};
Entero::Entero(int v){
        valor=v;
}
```

Si la función main es la siguiente:

```
Entero gl;
int main (){
  Entero a;
  cout<<gl.getValor()<<" , "<< a.getValor()<<endl;
}
```

La salida fuera esta:

0 , 0

Ya que al no realizar las instancias de los objetos con valores, el compilador los inicializa de acuerdo a la definición del los argumentos por defecto presente en la definición del constructor, el cual es cero.

Constructor copia

Un constructor de este tipo crea un objeto a partir de otro objeto existente. Estos constructores sólo tienen un argumento, que es una referencia a un objeto de su misma clase.

En general, los constructores copia tienen la siguiente forma para sus prototipos:

```
tipo_clase::tipo_clase(const tipo_clase&);
```

Y para su implementación:

```
tipo_clase::tipo_clase(const tipo_clase& obj){
        <código del constructor>
}
```

El ejemplo muestra la clase Entero haciendo uso de un constructor con argumento por defecto y la implementación del constructor de copia.

```
class Entero{
    private:
        //atributos
        int valor;
    public:
        //constructores
        Entero(int=0);
        Entero(const Entero &); //constructor copia
        //métodos
        Entero sumar(Entero);
        Entero restar(Entero);
        Entero invAd();
        void setValor(int);
        int getValor();
};
Entero::Entero(int v){
        valor=v;
}
```

```
//implementación del constructor copia
Entero::Entero(const Entero& obj){
        valor=obj.valor;
}
```

El siguiente ejemplo muestra el uso del constructor copia:

```
int main () {
   Entero a(3);
   cout<<a.getValor()<<endl;
   Entero b=a; //uso por defecto del constructor copia
   cout<<b.getValor()<<endl;
   Entero c(b); //uso explícito del constructor copia.
   cout<<c.getValor()<<endl;
   cin.get();
   return 0;
}
```

Destructores

Los destructores son métodos especiales que sirven para eliminar un objeto de una determinada clase. El destructor realizará procesos necesarios cuando un objeto termine su ámbito temporal, por ejemplo liberando la memoria dinámica utilizada por dicho objeto o liberando recursos usados, como ficheros, dispositivos, etc.

Al igual que los constructores, los destructores también tienen algunas características especiales:

- También tienen el mismo nombre que la clase a la que pertenecen, pero tienen el símbolo ~ delante.
- No tienen tipo de retorno, y por lo tanto no retornan ningún valor.
- No tienen parámetros.
- No pueden ser heredados.
- Deben ser públicos, no tendría ningún sentido declarar un destructor como privado, ya que siempre se usan desde el exterior de la clase, ni tampoco como protegido, ya que no puede ser heredado.
- No pueden ser sobrecargados, lo cual es lógico, puesto que no tienen valor de retorno ni parámetros, no hay posibilidad de sobrecarga.

Cuando se define un destructor para una clase, éste es llamado automáticamente cuando se abandona el ámbito en el que fue definido. Esto es así salvo cuando el objeto fue creado dinámicamente con el operador new, ya que en ese caso, cuando es necesario eliminarlo, hay que hacerlo explícitamente usando el operador delete.

En general, los destructores tienen la siguiente forma para sus prototipos:

tipo_clase::~tipo_clase();

Y para su implementación:

tipo_clase::~tipo_clase(){

<código del destructor>

}

Referencias

POZO, Salvador. C++ con clase [Recurso disponible en En línea] <http://c.conclase.net/> [Citado en 2011/02/07].

Calificadores, Clases y funciones amigas.

Introducción.

El presente capitulo tiene como objetivo dar al lector los conceptos relacionados con algunos calificadores de C++ tales como const y static, también se abordarán los conceptos de clases y funciones amigas, así como el uso del apuntador this.

El calificador const

Es un principio de buenas practicas de programación identificar que objetos necesitan ser modificables y cuales nos. La palabra reservada ***const*** permite especificar que un objeto no es modificable y que cualquier intento de modificarlo, resulte en un error de compilación. Ejemplo:

const Hora medio_dia(12,0,0);

Declara un objeto constante medio_dia de tipo Hora y es inicializado con 12:00:00.

Cuando se aplica el calificador *const* a un objeto de una clase, C++ deshabilita todas las operaciones que son miembros de la clase a no ser que la operación también halla sido declarada como *const*. Ademas el compilador no permite que las operaciones declaradas como *const* puedan modificar atributos de la clase.

Para ilustrar con un ejemplo, se tendrá en cuenta la definición del la clase Hora:

```
class Hora{
    private:
        int horas;
        int minutos;
        int segundos;
    public:
        void setHoras(int);
        void setMinutos(int);
        void setSegundos(int);
        int getHoras(void);
        int getMinutos(void);
        int getSegundos(void);
        Hora sumar(Hora);
        Hora restar(Hora);
        int aMinutos();
```

```
        int aSegundos();
        void mostrarLocal();
        void mostrarMilitar();
        Hora(int=0,int=0,int=0);
    };
```

Suponga que se desea usar una función llamada siguienteHora(Hora&) cuyo objetivo es avanzar una hora el objeto de tipo Hora el cual es pasado por parámetro:

```
void siguienteHora(Hora& h){
        h.setHoras(h.getHoras()+1);
}
```

Ahora en la función main se declaran dos objetos de tipo Hora en la que se define uno de ellos como constante:

```
int main()
{
   const Hora medio_dia(12,0,0); //medio_dia, declarado constante
   Hora otra(17,0,0);
   siguienteHora(otra); //OK
   otra.mostrarMilitar(); //18:00:00
   siguienteHora(medio_dia); //Error invalida inicialización de referencia de tipos
   medio_dia.mostrarMilitar(); //Error los métodos no declarados como const son descartados
   return 0;
}
```

El primer error ocurre debido a que se esta tratando de pasar por referencia un objeto que esta definido como *const*, recuerde que cuando un parámetro de una función es pasado por referencia, éste último puede ser modificado en el cuerpo de la función lo cual ese intento para un objeto declarado como *const* no esta permitido generando un error de compilación. Para resolver este problema la única solución sería quitar el calificador *const* en la declaración de la variable, o abstenerse de modificar el objeto.

El segundo error ocurre al llamar una operación del objeto que no esta declarado como *const*, recuerde que a los objetos declarados como *const*, C++ deshabilita todas las operaciones que son miembros de la clase que no halla sido declarada con ese calificador.

Para habilitar el uso de operaciones para los objetos declarados como *const* estos deben ser calificados como constante en la definición de la clase. Es altamente recomendado que las operaciones tanto de salida (*getters*) como las de impresión (*print*) sean calificadas con *const*.

Para nuestro ejemplo la definición de la clase Hora, teniendo en cuenta la calificación seria la siguiente:

```
class Hora{
    private:
        int horas;
        int minutos;
        int segundos;
    public:
        void setHoras(int);
        void setMinutos(int);
        void setSegundos(int);
        int getHoras(void) const ; // método get, normalmente declarada como const
        int getMinutos(void) const ; // método get, normalmente declarada como const
        int getSegundos(void) const ; // método get, normalmente declarada como const
        Hora sumar(Hora);
        Hora restar(Hora);
        int aMinutos();
        int aSegundos();
        void mostrarLocal()  const ; // método de impresion, normalmente declarada como const
        void mostrarMilitar()  const ; // método de impresion, normalmente declarada como const
        Hora(int=0,int=0,int=0);
    };
```

La implementación de los métodos calificados sería la siguiente:

```
int Hora::getHoras() const {return horas;}

int Hora::getMinutos() const {return minutos;}

int Hora::getSegundos() const {return segundos;}

void Hora::mostrarMilitar() const
{
    (horas<10)?cout<<"0"<<horas<<":":cout<<horas<<":";
    (minutos<10)? cout<<"0"<<minutos<<":":cout<<minutos<<":";
    (segundos<10)? cout<<"0"<<segundos<<endl:cout<<segundos<<endl;
```

```
}

void Hora::mostrarLocal() const
{
    (horas<10)?cout<<"0"<<horas<<":":(horas>12)?cout<<horas%12<<":":cout<<horas<<":";
    (minutos<10)? cout<<"0"<<minutos<<":":cout<<minutos<<":";
    (segundos<10)? cout<<"0"<<segundos:cout<<segundos;
    (horas>12)?cout<<" PM"<<endl:cout<<" AM"<<endl;
}
```

Teniendo el cuenta tanto la funciones main como siguienteHora(Hora&) y la correcta calificación de las operaciones constantes, la salida esperada para estas funciones es:

```
void siguienteHora(Hora& h){
        h.setHoras(h.getHoras()+1);
}

int main()
{
    const Hora medio_dia(12,0,0); //medio_dia, declarado constante
    Hora otra(17,0,0);
    siguienteHora(otra); //OK
    otra.mostrarMilitar(); //18:00:00
    medio_dia.mostrarMilitar(); //12:00:00
    return 0;
}
```

El calificador static

Ciertos miembros de una clase pueden ser declarados como static. Los miembros static tienen algunas propiedades especiales.

En el caso de los datos miembro static sólo existirá una copia que compartirán todos los objetos de la misma clase. Al consultar el valor de ese dato desde cualquier objeto de esa clase se obtendrá siempre el mismo resultado, y si es modificado, se modificará para todos los objetos.

Por ejemplo, para la definición de la clase Hora:

```
class Hora{
    private:
        int horas;
        int minutos;
        int segundos;
    public:
        void setHoras(int);
        void setMinutos(int);
        void setSegundos(int);
        int getHoras(void) const ; // método get, normalmente declarada como const
        int getMinutos(void) const ; // método get, normalmente declarada como const
        int getSegundos(void) const ; // método get, normalmente declarada como const
        Hora sumar(Hora);
        Hora restar(Hora);
        int aMinutos();
        int aSegundos();
        void mostrarLocal()  const ; // método de impresion, normalmente declarado como const
        void mostrarMilitar()  const ; // método de impresion, normalmente declarado como const
        Hora(int=0,int=0,int=0);
};
```

Si quisiésemos saber cuantos objetos de tipo Hora han sido instanciados en la ejecución de un programa, sería necesario tener un atributo que fuera compartido por todos los objetos de esa clase, de tal manera que pudiese llevar la cuenta del número de objetos instanciados.

Entonces para resolver esa inquietud habría que agregar un nuevo atributo de tipo entero llamado nroInstancias, lo cual para que sea compartido por todos los miembros de la clase, este debe ser declarado de la siguiente forma:

```
static int nroInstancias;
```

Note que la palabra reservada static precede la declaración del tipo de dato.

Es necesario declarar e inicializar los miembros static de la clase, esto es por dos motivos. El primero es que los miembros static deben existir aunque no exista ningún objeto de la clase, declarar la clase no crea los datos miembro estáticos, en necesario hacerlo explícitamente. El segundo es porque no lo hiciéramos, al declarar objetos de esa clase los valores de los miembros estáticos estarían indefinidos, y los resultados no serían los esperados.

La forma de inicializar el atributo estático (y en general cualquier atributo estático) es el siguiente:

```
int Hora::nroInstancias=0;
```

En el caso de la operaciones miembro static, estas no pueden acceder a los miembros de los objetos, sólo pueden acceder a los datos miembro de la clase que sean static, ademas, las operaciones declaradas static no pueden ser declaradas como const.

Para nuestro ejemplo se desarrolla un método que muestre el número de instancias, su definición en la clase Hora sería :

```
static int getNroInstancias() ;
```

Con base a lo anterior, la definición e implementación de los atributos y métodos estáticos en la clase Hora seria:

```
class Hora{
    private:
        int horas;
        int minutos;
        int segundos;
       //atributos declarados static
        static int nroInstancias;
    public:
        …
       //metodos declarados static
       static int getNroInstancias() ;
};

//inicialización de un atributo estático
int Hora:: nroInstancias=0;

//implementación del método estático
static int Hora::getNroInstancias()  {return nroInstancias;}

Hora::Hora(int h, int m, int s)
{
  setHoras(h);
  setMinutos(m);
  setSegundos(s);
  //cuenta las instancias creadas
```

```
    nroInstancias++;
}
```

Si la función main es la siguiente:

```
int main()
{
        Hora h1,h2,h3;
        cout<<Hora::getNroInstancias()<<endl;
        return 0;
}
```

la salida es 3.

Los métodos declarados como static suelen ser usados con su nombre completo, incluyendo el nombre de la clase y el operador de ámbito (::).

El apuntador this

Es un apuntador que tiene asociado cada objeto y que apunta a si mismo. Ese a se puede usar, y de hecho se usa, para acceder a sus miembros.

Ejemplo: teniendo en cuenta la definición de la clase Entero:

```
class Entero{
    private:
        int valor;
    public:
        Entero sumar(Entero);
        Entero restar(Entero);
        Entero invAd();
        void setValor(int);
        int getValor() const;
        Entero(int=1);
        Entero(const Entero&);
        ~Entero();
  };
```

Los constructores implementados de la siguiente forma:

```
Entero::Entero(int val)
{
  valor=val;
}

Entero::Entero(const Entero& n)
{
  valor=n.valor;
}
```

Haciendo uso del apuntador this, pueden ser reimplementados así:

```
Entero::Entero(int valor)
{
        this->valor=valor;
}

Entero::Entero(const Entero& n)
{
        (*this).valor=n.valor;
}
```

Usando el apuntador this para realizar llamadas de operaciones en cascada.

Otro uso del apuntador this es con el fin de realizar llamadas de operaciones miembros en cascada, es decir, invocar múltiples operaciones en la misma sentencia.

Ejemplo: en la definición de la clase Hora, es necesario reescribir las operaciones setHoras(int), setMinutos(int) y setSegundos(int), todas con valor de retorno Hora&, con el fin de habilitar las llamadas de operaciones en cascada:

```
class Hora{
     private:
            int horas;
            int minutos;
            int segundos;
     public:
           Hora& setHoras(int);
           Hora& setMinutos(int);
           Hora& setSegundos(int);
          …
};
Hora& Hora::setHoras(int h) {
          this->horas=h;
          return *this; //habilita la cascada
}
Hora& Hora::setMinutos(int m)
{
   if(m>=60)
     (*this).minutos=59;
   else if(m<0)
     this->minutos=0;
   else
     (*this).minutos=m;
   return *this;
}
Hora& Hora::setSegundos(int s) {
    if(s>=60)  segundos=59;
    else if(s<0)
      segundos=0;
    else
      segundos=s;
   return *this;
}
```

Si la función main es esta:

```
int main()
{
   Hora h;
   h.setHoras(12).setMinutos(30).setSegundos(12); //llamada de operaciones en cascada
   h.mostrarMilitar();
   h.setHoras(1);
   h.mostrarLocal();
   return 0;
}
```

La salida esperada es:

12:30:12

01:30:12 AM

Funciones y clases amigas.

Función Amiga

Una función amiga de una clase es una función, generalmente declarada e implementada por fuera del ámbito de la clase, en la cual, esta puede acceder a los miembros, tanto público como privados, de la clase.

Ejemplo: para la clase Entero, cuya definición es:

```
class Entero{
    private:
        int valor;
    public:
        Entero sumar(Entero);
        Entero restar(Entero);
        Entero invAd();
        void setValor(int);
        int getValor();
        Entero(int=1);
        Entero(const Entero&);
        ~Entero();
```

```
};
```

Se desea desarrollar una función llamada *establecer(Entero&, int)* de tipo *void* que permita, establecer el valor del objeto recibido como parámetros, al estilo del manejo de las estructuras en el lenguaje C, sin hacer uso de los métodos de entrada y salida de la clase entonces es necesaria declararla friend en la definición de la clase tal como se muestra a continuación:

```
class Entero{
    private:
        int valor;
    public:
        friend void establecer(Entero&, int);  //declaración de la función amiga
        //métodos
        …
};

//implementación de la función amiga
void establecer(Entero& n, int v)
{
        n.valor=v; //acceso a los miembros privados del objeto
}
//implementación de los métodos
…
```

El uso de la función amiga no difiere de las demás funciones, por ejemplo, note su uso en la función main.

```
int main(){
        Entero a;
        establecer(a,5);
        cout<< " el valor de a es "<<a.getValor()<<endl; //muestra 5
}
```

Clase Amiga

Se dice que una clase A es amiga de una clase B cuando esta ultima puede acceder a los miembros, tanto público como privados, de la clase A.

La amistad entre clases es otorgada mas no tomada, es decir que si la clase A es amiga de la clase B y la clase B es amiga de la clase C, esto no quiere decir que la clase C sea amiga de la clase A, ya que para ello debe existir un contrato entre la clase A y C respectivamente.

Para declara que todos los miembros de la clase A son amigos de la clase B debe declararse en la definición de la clase B la siguiente sentencia:

```
friend class A
```

Ejemplo: note la definición de la clase Fecha:

```
class Fecha{
    private:
        int dia;
        int mes;
        int agnio;
        Hora hora;
        static int diasxmes[12];
    public:
        Fecha(int,int,int,Hora); //dia, mes, año, hora
        Fecha& setDia(int);
        Fecha& setMes(int);
        Fecha& setAgnio(int);
        bool esBisiesto();
        int contarDias(const Fecha&);
        string mostrarFormaCorta() const;
        string mostrarFormaLarga() const;

        //declaración de la clase Hora como amiga de la clase Fecha
        friend class Hora;
};
```

Referencias

DEITEL H. M. & DEITEL P. J. C++: how to program. 6 edicion. Prentice Hall, ISBN: 0-13-615250-3. NewJersey. 2008.

Asignación dinámica de memoria.

Introducción.

El presente capitulo tiene como objetivo dar al lector los conceptos relacionados con la asignación dinámica de memoria, primero en el predecesor (lenguaje C) así como en C++. Se explorará el uso de los operadores de C++ new y delete, también se abordarán los conceptos de composición de clases.

Definición de asignación dinámica de memoria

La asignación dinámica de memoria es una característica de C y C++. Le permite al usuario crear tipos de datos y estructuras de cualquier tamaño de acuerdo a las necesidades que se tengan en la solución del problema por computadora.

Asignación dinámica en C

La función *malloc* es empleada comúnmente para intentar "tomar" una porción contigua de memoria. Esta definida como:

```
void *malloc(size_t size);
```

Lo anterior indica que regresará un apuntador del tipo void *, el cual es el inicio en memoria de la porción reservada de tamaño size. Si no puede reservar esa cantidad de memoria la función regresa un apuntador nulo o NULL

Dado que void * es regresado, C asume que el apuntador puede ser convertido a cualquier tipo. El tipo de argumento size_t esta definido en la cabecera stddef.h y es un tipo entero sin signo.

Por lo tanto:

```
char *cp;
cp = (char *) malloc(100);
```

Intenta obtener 100 bytes y asignarlos a la dirección de inicio a cp.

Es usual usar la función sizeof() para indicar el número de bytes, por ejemplo:

```
int *ip;
ip = (int *) malloc(100 * sizeof(int) );
```

El compilador de C requiere hacer una conversión del tipo. La forma de lograr la coerción (cast) es usando (char *) y (int *), que permite convertir un apuntador void a un apuntador tipo char e int respectivamente. Hacer la conversión al tipo de apuntador correcto asegura que la aritmética con el apuntador funcionará de forma correcta.

Es una buena práctica usar sizeof() aún si se conoce el tamaño actual del dato que se requiere, ya que de esta forma el código se hace independiente del dispositivo (portabilidad).

La función sizeof() puede ser usada para encontrar el tamaño de cualquier tipo de dato, variable o estructura. Simplemente se debe proporcionar uno de los anteriores como argumento a la función.

Por lo tanto:

```
int i;
struct COORD {float x,y,z};
struct COORD *pt;
```

sizeof(int), sizeof(i), sizeof(struct COORD) y sizeof(pt) son también sentencias correctas.

En el siguiente ejemplo se reserva memoria para la variable ip, en donde se emplea la relación que existe entre apuntadores y arreglos, para manejar la memoria reservada como un arreglo. Por ejemplo, se pueden hacer cosas como:

```
main()
{
   int *ip, i;
   ip = (int *) malloc(100 * sizeof(int) );
   ip[0] = 1000;
   for (i=0; i<100; ++i)
      scanf("%d",ip++);
}
```

Existen dos funciones adicionales para reservar memoria, *calloc()* y *realloc()*. Los prototipos son dados a continuación:

```
void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);
```

Cuando se usa la función *malloc*() la memoria no es inicializada (a cero) o borrada. Si se quiere inicializar la memoria entonces se puede usar la función calloc. La función calloc es computacionalmente un poco más cara pero, ocasionalmente, más conveniente que malloc. Se debe observar también la diferencia de sintaxis entre calloc y malloc, ya que calloc toma el número de elementos deseados (nmemb) y el tamaño del elemento (size), como dos argumentos individuales.

Por lo tanto para asignar a 100 elementos enteros que estén inicializados a cero se puede hacer:

```
int *ip;
ip = (int *) calloc(100, sizeof(int) );
```

La función *realloc* intenta cambiar el tamaño de un bloque de memoria previamente asignado. El nuevo tamaño puede ser más grande o más pequeño. Si el bloque se hace más grande, entonces el contenido anterior permanece sin cambios y la memoria es agregada al final del bloque. Si el tamaño se hace más pequeño entonces el contenido sobrante permanece sin cambios.

Si el tamaño del bloque original no puede ser sobredimensionado, entonces realloc intentará asignar un nuevo bloque de memoria y copiará el contenido anterior. Por lo tanto, la función devolverá un nuevo apuntador (o de valor diferente al anterior), este nuevo valor será el que deberá usarse. Si no puede ser reasignada nueva memoria la función realloc devuelve NULL.

Si para el ejemplo anterior, se quiere reasignar la memoria a 50 enteros en vez de 100 apuntados por ip,

se hará:

```
ip = (int *) realloc ( ip, 50*sizeof(int) );
```

la función *free* libera la memoria asignada por cualquiera de las funciones de asignación previamente mencionadas. El prototipo de la función free se muestra a continuación:

```
void free (void* ptr);
```

Si se desea liberar la memoria asignada al apuntador ip se hará:

```
if(ip!=NULL)
        free(ip);
```

Asignación dinámica en C++

El C clásico proporciona formas específicas de solicitar al compilador que reserve y/o inicie espacio de tamaño arbitrario en el montón. También dispone de formas para rehusarlo cuando ya no se considera necesario. Son las funciones de librería malloc, calloc, realloc y free. C++ mantiene (por compatibilidad) las formas clásicas, pero introduce cuatro nuevos operadores: new, delete, new [ ] y delete [ ], que no solo crean y rehúsan espacio de forma más cómoda y eficiente, sino que incluso pueden crear objetos persistentes. Los dos primeros se utilizan para objetos de cualquier tipo, mientras que new [ ] y delete [ ] se usan con matrices.

El operador new

El operador new (palabra clave C++) proporciona espacio de almacenamiento persistente, similar pero superior a la función de Librería Estándar malloc. Este operador permite crear un objeto de cualquier tipo, incluyendo tipos definidos por el usuario, y devuelve un puntero (del tipo adecuado) al objeto creado.

Sintaxis:

```
[::]new [<emplazamiento>] <tipo> [(<inicialización>)]

[::]new [<emplazamiento>] (<tipo>) [(<inicialización>)]

[::]new [<emplazamiento>] <tipo>[<número_elementos>]

[::]new [<emplazamiento>] (<tipo>)[<número_elementos>]
```

El operador opcional :: está relacionado con la sobrecarga de operadores. Lo mismo se aplica a emplazamiento.

La inicialización, si aparece, se usará para asignar valores iniciales a la memoria reservada con new, pero no puede ser usada con arreglos.

Las formas tercera y cuarta se usan para reservar memoria para arreglos dinámicos. La memoria reservada con new será válida hasta que se libere con delete o hasta el fin del programa, aunque es aconsejable liberar siempre la memoria reservada con new usando delete. Se considera una práctica muy sospechosa no hacerlo.

Si la reserva de memoria no tuvo éxito, new devuelve un puntero nulo, NULL.

Las diferencias de new con las funciones malloc y calloc de las librerías C tradicionales comportan una

clara ventaja a favor de la primera:

- malloc y calloc devuelven un puntero a void, mientras que new devuelve un puntero conformado al tipo específico del objeto creado.
- malloc y calloc simplemente reservan memoria (calloc la inicializa a 0), mientras que new construye el objeto, lo que hace que este operador esté estrechamente relacionado con los constructores cuando se aplica a clases.
- new y delete pueden ser sobrecargados como el resto de los operadores, lo que comporta ciertas ventajas adicionales.

Ejemplos:

```
Fecha* pf=new Fecha[30];
```

Asigna dinámicamente memoria al apuntador pf de tipo Fecha obteniendo un arreglo de 30 objetos de tipo fecha.

El operador delete

El operador delete se usa para liberar la memoria dinámica reservada con new.

Sintaxis:

```
[::]delete [<expresión>]
[::]delete[] [<expresión>]
```

La expresión será normalmente un puntero, el operador delete[] se usa para liberar memoria de arreglos dinámicos.

Es importante liberar siempre usando delete la memoria reservada con new. Existe el peligro de pérdida de memoria si se ignora esta regla.

Cuando se usa el operador delete con un puntero NULL, no se realiza ninguna acción. Esto permite usar el operador delete con punteros sin necesidad de preguntar si es nulo antes.

De todos modos, es buena idea asignar el valor 0 a los punteros que no han sido inicializados y a los que han sido liberados. También es bueno preguntar si un puntero es nulo antes de intentar liberar la memoria dinámica que le fue asignada.

Ejemplo:

```
Fecha* pf=new Fecha[30];
…
if(pf!=NULL)
        delete pf;
```

Caso de estudio: la clase Pila

Una pila (*stack* en inglés) es una lista ordinal o estructura de datos en la que el modo de acceso a sus elementos es de tipo LIFO (del inglés *Last In First Out*, último en entrar, primero en salir) que permite almacenar y recuperar datos. Esta estructura se aplica en multitud de ocasiones en el área de

informática debido a su simplicidad y ordenación implícita de la propia estructura.

En el paradigma de la Programación Orientada a Objetos la Pila puede definirse como un *Tipo Abstracto de Datos* la cual tiene como atributos un arreglo finito de tamaño *n* y cuatro operaciones a saber:

meter: agrega un elemento a la cima de la pila.

sacar: quita el elemento que esta en la cima de la pila.

estaLlena: indica si la pila ha superado el máximo número de elementos que puede almacenar.

estaVacia: indica si la pila no tiene elementos.

En C++ la definición de este *Tipo Abstracto de Datos* se muestra a continuación:

```
class Pila{
        private:
                int tam;
                int *contenido;
                int ind;
        public:
                Pila(int);
                ~Pila();
                void meter(int);
                int sacar();
                bool estaVacia();
                bool estaLlena();
};
```

En la definición del arreglo de tamaño finito *n*, se hace uso de la asignación dinámica de memoria con el fin de usar un uso eficiente de la memoria (ni un bit mas, ni un bits menos) de acuerdo a las necesidades o requerimientos de la aplicación.

En el constructor se hace uso del operador new, para crear un arreglo de tipo int de tamaño *n*:

```
Pila::Pila(int n){
        if(n>0)
                tam=n;
        else
                tam=10;
        contenido=new int[tam];
        ind=0;
}
```

En el destructor se hace el uso del operador delete, con el fin de liberar la memoria asignada por new:

```
Pila::~Pila()
{
        if(contenido!=NULL)
                delete contenido;
}
```

La implementación de las demás operaciones de la clase Pila (*meter, sacar, estaLlena*, *estaVacia*) se muestran a continuación:

```
void Pila::meter(int v)
{
        if(estaLlena())
                cout<<"la pila esta llena"<<endl;
        else
        {
                contenido[ind]=v;
                ind++;
        }
}
int Pila::sacar(){
        int v=-1;
        if(estaVacia())
                cout<<"la pila esta vacia"<<endl;
        else
        {
                v=contenido[ind-1];
                ind--;
        }
        return v;
}
bool Pila::estaVacia(){
        return (ind-1<0)?true:false;
}
```

```
bool Pila::estaLlena()
{
        return (ind==tam)?true:false;
}
```

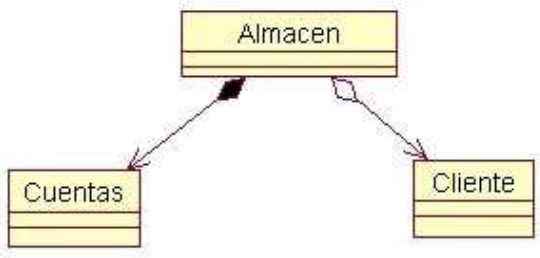
Composición y agregación de clases

En el paradigma de la Programación Orientada a Objetos, Para modelar objetos complejos, no bastan los tipos de datos básicos que proveen los lenguajes: enteros, reales y secuencias de caracteres. Cuando se requiere componer objetos que son instancias de clases definidas por el desarrollador de la aplicación, tenemos dos posibilidades:

Por Valor: Es un tipo de relación estática, en donde el tiempo de vida del objeto incluido esta condicionado por el tiempo de vida del que lo incluye. Este tipo de relación es comúnmente llamada Composición (el Objeto base se construye a partir del objeto incluido, es decir, es "parte/todo").

Por Referencia: Es un tipo de relación dinámica, en donde el tiempo de vida del objeto incluido es independiente del que lo incluye. Este tipo de relación es comúnmente llamada Agregación (el objeto base utiliza al incluido para su funcionamiento).

Un Ejemplo, haciendo uso de un diagrama de clases del Lenguaje Unificado de Modelado (UML), es el siguiente:



En donde se destaca que:

- Un Almacén posee Clientes y Cuentas (los rombos van en el objeto que posee las referencias).
- Cuando se destruye el Objeto Almacén también son destruidos los objetos Cuenta asociados, en cambio no son afectados los objetos Cliente asociados.
- La composición (por Valor) se destaca por un rombo relleno.
- La agregación (por Referencia) se destaca por un rombo transparente.

La flecha en este tipo de relación indica la navegabilidad del objeto referencial. Cuando no existe este tipo de particularidad la flecha se elimina.

En C++ podemos definir el modelo mostrado en el diagrama de clases anterior de la siguiente forma:

```
class Almacen{
        private:
                Cuentas cuentas[100]; //composición de cuentas por valor (composición)
                Cliente* clientes[100]; //composición de clientes por referencia (agregación)
        public:
                void registrarCliente(Cliente*);
                Cuentas darCuentas(Cliente*);
                // otros métodos
};
```

Referencias

TEJEDA, Héctor. Manual de C - Asignación dinámica de memoria y Estructuras dinámicas [en Linea] <http://www.fismat.umich.mx/mn1/manual/node10.html>[Citado en 2011/02/19]

ZATOR Systems. Curso C++ - Operador new [en Linea] <http://www.zator.com/Cpp/E4_9_20.htm>[Citado en 2011/02/19]

POZO, Salvador. C++ con clase [en Linea] <http://c.conclase.net/curso/?cap=013b>[Citado en 2011/02/19]

SALINAS, Patricio. Hitschfeld, Nancy. UML - modelo del clases. [en Linea] <http://www.dcc.uchile.cl/~psalinas/uml/modelo.html>[Citado en 2011/02/19]

Sobrecarga de Operadores.

Introducción.

El presente capitulo tiene como objetivo dar al lector los conceptos relacionados con la sobrecarga de operadores, haciendo énfasis en los operadores de tipo unario y binarios. Se identificarán los operadores que pueden ser sobrecargados. Se explorará el uso de la palabra reservada de C++ operator, así como las diferentes técnicas de sobrecarga.

¿Que es un operador?

Un operador es un símbolo (+, -, *, /, etc) que tiene una función predefinida (suma, resta, multiplicación, etc) y que recibe sus argumentos de manera infija, en el caso de tener 2 argumentos (a operador b), o de manera prefija o postfija, en el caso de tener uno solo (operador a , o bien, a operador).

En C/C++ existen una gran variedad de operadores, que se pueden agrupar de la siguiente manera:

- Operadores aritméticos
- Operadores relacionales
- Operadores lógicos
- Operadores a nivel de bit (bitwise operators)
- Operadores especiales

La sobrecarga de operadores

En C++ el principio de la sobrecarga no solo se aplica a funciones, sino también a los operadores, esto permite extender el significado de los operadores a los tipos definidos por el usuario. De este modo, un programador puede proporcionar su propio operador de una clase por la sobrecarga del operador integrado para realizar algún cálculo específico cuando el operador se utiliza con los objetos de esa clase.

Para sobrecargar los operadores se pueden hacer uso de varias técnicas las cuales se conocerán en el siguiente apartado.

La siguiente lista muestra los operadores en C++ que pueden ser sobrecargados:

```
+     -     *     /     %     ^     &     |
~     !     ,     =     <     >     <=    >=
++    --    <<    >>    ==    !=    &&    ||
+=    -=    /=    %=    ^=    & =   |=    *=
<<=   >>=   [ ]   ( )   ->    ->*   new   delete
```

Sobrecarga de operadores unarios

Un operador unario, es aquel que es aplicado a un operando. Son ejemplos de ellos el operador lógico de negación (!), los operadores de incremento y decremento (++,--) ya sea en modo prefijo y postfijo, entre otros.

La sintaxis para sobrecargar un operador unario es la siguiente:

En la definición de la clase:

```
<tipo> operator<operador unario>();
```

Y en su implementación:

```
<tipo> <identificador de clase>::operator<operador unario>(){
  //Cuerpo de la implementación
}
```

Normalmente el <tipo> es la clase para la que estamos sobrecargando el operador. Nótese que en la definición del método este lleva la palabra reservada operator y, por ser unario, en la mayoría de los casos no tiene parámetros ya que el proceso se aplica sobre las propiedades de la clase.

Como ejemplo observe la definición de la clase Entero:

```
class Entero{
    private:
        int valor;
    public:
        Entero sumar(Entero);
        Entero restar(Entero);
        Entero invAd();
        void setValor(int);
        int getValor();
        Entero(int=1);
        Entero(const Entero&);
        ~Entero();
};
```

Si se desea sobrecargar el operador ++ en prefijo, entonces en la definición de la clase podemos definir la sobrecarga del operador ++ de la siguiente forma:

En la definición de la clase Entero:

```
class Entero{
    private:
        int valor;
    public:
        Entero operator ++(); //sobrecarga del operador incremento en prefijo
        //otros métodos
        . . .
};
```

Mientras su implementación es:

```
Entero Entero::operator ++()
{
        ++valor;
        return *this;
}
```

Al sobrecargar el operador de incremento prefijo podemos usarlo de la siguiente forma:

```
int main () {
   Entero a;
   for(a.setValor(0); a.getValor() < 11 ; ++a)
    cout<<a.getValor()<<endl;
   return 0;
}
```

En el caso de la sobrecarga del operador incremento posfijo, es necesario que se le defina un parámetro de tipo int con el fin de diferenciarlo del operador incremento prefijo. Como ejemplo observe su definición en la clase Entero:

```
class Entero{
    private:
        int valor;
    public:
        Entero operator ++(); //operador incremento prefijo
        Entero operator ++(int); //operador incremento posfijo
        //otros métodos
};
```

Mientras que su implementación es:

```
Entero Entero::operator ++(int a) {
        valor++;
        return *this;
}
```

Nótese que el parámetro no es usado, esto solo se hace con el fin de evitar un error sintáctico cuando se tratase de sobrecargar ambos operadores sintácticamente similares. De igual manera ocurre al sobrecargar el operador de decremento tanto en modo prefijo como postfijo.

Sobrecarga de operadores binarios

Para sobrecargar operadores binarios (+,-,*,/,%, Etc.), su sintaxis no difiere del caso de la sobrecarga de los operadores unarios, la diferencia es que, en la mayoría de los casos, recibe como parámetro, un objeto de la misma clase.

Su sintaxis en la definición de la clase es:

```
<tipo> operator<operador binario>(<tipo>);
```

Y en su implementación:

```
<tipo> operator<operador binario>(<tipo> <identificador>){
        //implementación del cuerpo del método
}
```

Como ejemplo se va a sobrecargar el operador + con el fin de realizar la adición entre dos objetos de tipo Entero:

```
class Entero{
    private:
        int valor;
    public:
        Entero operator ++(); //operador incremento prefijo
        Entero operator ++(int); //operador incremento posfijo
        Entero operator + (Entero); //sobrecarga del operador binario +
        //otros métodos
        . . .
};
```

Una forma de implementar la sobrecarga del operador binario + puede ser la mostrada a continuación:

```
Entero Entero::operator +(Entero e) {
        return sumar(e);
}
```

Otro operador que es muy útil sobrecargar es el de asignación (=) cuya implementación en la clase Entero es la siguiente:

```
Entero& Entero::operator=(const Entero& e) {
        if(this!=&e)    {
                setValor(e.valor);
        }
        return *this;
}
```

Hay que tener en cuenta la posibilidad de que se asigne un objeto a si mismo. Por eso se compara el puntero this con la dirección del parámetro, si son iguales es que se trata del mismo objeto, y no debemos hacer nada. Esta es una de las situaciones en las que el puntero this es imprescindible.

Con base a los operadores binarios sobrecargados para la clase Entero, se pueden escribir expresiones de la siguiente forma:

```
int main ()
{
   Entero a(1),b(2),c;
   c=a+b;
   cout<<c.getValor()<<endl;  //muestra 3
   return 0;
}
```

También es posible sobrecargar los operadores lógicos como, por ejemplo los de comparación (==, <, >). En el ejemplo se muestra la implementación de la sobrecarga del operador de comparación menor que (<) para la clase Entero.

```
bool Entero::operator<(Entero e)
{
        if(valor<e.getValor())
                return true;
        else
                return false;
}
```

Al sobrecargar los operadores lógicos, esto permite que los objetos puedan ser utilizados en sentencias condicionales, u otras que hagan uso operaciones lógicas. El siguiente ejemplo hace uso del operador menor que (<) sobrecargado:

```
int main () {
   Entero lim(11);
   for(Entero i(0);i<lim;i++)
        cout<<i.getValor()<<endl;
   return 0;
}
```

Sobrecarga de operadores haciendo uso de funciones amigas

También es posible realizar el proceso de sobrecarga de operadores usando funciones amigas. Observe como ejemplo la implementación de la sobrecarga del operador – haciendo uso de una función amiga de la clase Entero:

```
class Entero{
    private:
        int valor;
    public:
        friend Entero operator-(Entero,Entero); //definición de la función amiga
        //otros métodos
        . . .
};
//Implementación de la función amiga
Entero operator - (Entero a, Entero b) {
        return a.restar(b);
}
```

Nótese es que necesario definir los dos argumentos en el caso de la sobrecarga de los operaciones binarios y un argumento en el caso de los operadores unarios debido a que la función amiga no hace parte de la clase.

Aun cuando la sobrecarga se realizó utilizando una función amiga, su uso no difiere de los casos anteriores, por ejemplo note su uso en la función main:

```
int main () {
        Entero lim(11);
        for(Entero i(0);i<lim;i++)
                cout<<(-(lim-i)).getValor()<<endl;
        return 0;
}
```

Referencias

PONS, Ramón. Curso de C - Operadores. [Recurso disponible en Línea] <http://laurel.datsi.fi.upm.es/~rpons/personal/trabajos/curso_c/node44.html> [Con Acceso 2011/02/27].

POZO, Salvador. C++ con clase [en Linea] <http://c.conclase.net/>[Citado en 2011/02/19]

Mas sobrecargas de Operadores

Introducción.

El presente capitulo tiene como objetivo extender al lector los conceptos relacionados con la sobrecarga de operadores, haciendo énfasis en los operadores de extracción e inserción de flujos (stream). Se mencionara la definición de la sobrecarga de operadores especiales como [], new y delete.

Sobrecargando los operadores de inserción y extracción

Las clases pueden ser adaptadas para que sus objetos puedan ser extraídos de un flujo de entrada (input) e insertados de un flujo de salida (output).

La clase *std::istream* define el operador de extracción para los tipos básicos de datos como int, char, float, double, bool, Etc.

La clase *std::ostream* define el operador de inserción para los tipos básicos de datos.

Sobrecarga del operador de inserción

Es posible extender la funcionalidad del operador de inserción (<<) para los tipos abstractos de datos de tal forma que si está instanciado un objeto *obj* de la clase *MiClase*, la expresión:

cout<<obj;

Muestre la información relevante del objeto *obj* (por ejemplo sus propiedades u otro mensaje definido por el usuario) por la salida (sea pantalla u otra previamente especificada).

En la sentencia *cout<<obj* el objeto *cout*, por ser de la clase *ostream*, usa el método miembro *operator<<* definido en la clase *ostream*, Este operador recibe dos operandos a saber *ostream&* y *MiClase&*.

Para realizar la acción requerida es necesario sobrecargar el operador << haciendo uso de funciones amigas cuya definición dentro de la clase *MiClase* es de la siguiente:

```
friend ostream& operator<<(ostream&, const MiClase&) ;
```

Su implementación obedece la siguiente sintaxis:

```
ostream& operator<<(ostream& salida, const MiClase& obj){
        //sentencias
        return salida;
}
```

Sobrecarga del operador de extracción

Es posible extender la funcionalidad del operador de extracción (>>) para los tipos abstractos de datos de tal forma que si tenga instanciado un objeto *obj* de la clase *MiClase*, la expresión:

cin>>obj;

Extrae, de la entrada estándar, la información relevante del objeto *obj* de acuerdo a un formato previamente establecido.

En la sentencia *cin>>obj* el objeto *cin*, por ser de la clase *istream*, usa el método miembro *operator>>* definido en la clase *istream*, Este operador recibe dos operandos a saber *istream&* y *MiClase&*.

Para realizar la acción requerida es necesario sobrecargar el operador >> haciendo uso de funciones amigas cuya definición dentro de la clase *MiClase* es de la siguiente:

```
friend istream& operator>>(istream&, MiClase&) ;
```

Note que ambos argumentos son pasados por referencia de tal forma que, dentro del cuerpo de la función, puedan ser editados las propiedades del objeto de nuestra clase.

La implementación de la función presenta la siguiente sintaxis:

```
istream& operator<<(istream& entrada, MiClase& obj){
    //sentencias
    return entrada;
}
```

Sobrecarga del operador de indexación

Es posible sobrecargar el operador de indexación [], es muy útil para acceder a objetos que se encuentran dentro de arreglos *n – dimensionales*. Este solo acepta ser sobrecargado como método de la clase; por ende su definición dentro de la clase obedece la siguiente sintaxis:

```
<tipo_dev> operator [](const <tipo>& );
```

Mientras que su implementación es:

```
<tipo_dev> <clase>::operator [](const <tipo>& ){
    //sentencias
}
```

Un objetivo de sobrecargar este operador es con el fin de evitar accesos ilegales o desbordamientos cuando se acceden a arreglos de objetos de la clase, la cual se le esta extendiendo la funcionalidad del operador.

Sobrecarga de los operadores new y delete

También es posible sobrecargar los operadores new y delete con el objetivo darle alguna comportamiento especial a los objetos de la clase la cual se define la sobrecarga de estos operadores.

Su sintaxis en la definición de la clase como métodos de ésta se definen a continuación:

```
void* operator new(size_t t);          //Clase *obj= new Clase;
void* operator new[](size_t t);        //Clase *obj= new Clase[n];
void delete(void* x);                  //delete obj;
```

```
void delete[](void* x);                    //delete[] obj;
```

Su implementación no difiere de las implementaciones de los métodos de la clase.

Caso de estudio: Arreglos Dinámicos

En las ciencias de la computación, un arreglo dinámico es una estructura de datos, de acceso aleatorio y tamaño variable que permite agregar o quitar elementos. En el paradigma de la programación orientada a objetos este puede definirse como un Tipo Abstracto de Datos el cual como atributos tendría un apuntador hacia el tipo de datos y un número que indica la longitud de este; como operaciones básicas se define:

agregar(v: tipo): agrega el valor v al final del arreglo

quitar(p: entero): quita del arreglo el valor ubicado en la posición p

getValor(p:entero): devuelve el valor almacenado en la posición p

setValor(p:entero,v:entero) establece v como el valor ubicado en la posición p del arreglo.

Entre otras.

En C++ es posible definir el TAD mencionado anteriormente como una Clase la cual, su interfaz es puesta a continuación:

```
class ArregloDinamico{
    private:
        int* arreglo;
        int longitud; //tamaño del arreglo
    public:
        ArregloDinamico(int=0);
        ~ArregloDinamico();
        void setValor(int,int); //valor, posición
        void agregar(int); //agrega el valor al final del arreglo
        void quitar(int);  //quita el valor ubicado la posición del arreglo
        int getValor(int) const; //da el valor según la posición
        int getLongitud() const; //devuelve la longitud del arreglo
        void setLongitud(int); //reasigna el tamaño del arreglo
        ArregloDinamico operator +(const ArregloDinamico&); //concatena dos arreglos
        //sobrecarga de los operadores de extracción e inserción de flujos
        friend istream& operator>>(istream&, ArregloDinamico&);
        friend ostream& operator<<(ostream&, const ArregloDinamico&);
};
```

Observe la implementación de las funciones amigas que sobrecargan los operadores de extracción e inserción de flujos:

```
ostream& operator <<(ostream& salida, const ArregloDinamico &a){
        salida<<"{";
        for(int i=0;i<a.getLongitud();i++)
                salida<<a.getValor(i)<<((i<a.getLongitud()-1)?",":"}");
        return salida;
}
```

La salida (cout) es una arreglo de la forma {a,b,c,d,e,...,n} siendo cada uno de los elementos enteros.

Mientras que la entrada (cin) es de la forma {a,b,c,d,e,...,n} siendo cada uno de los elementos enteros:

```
istream& operator>>(istream& entrada, ArregloDinamico &a){
        char c;
        int valor;
        a.setLongitud(0);
        entrada>>c;
        if(c=='{')      {
                while(c!='}'){
                        entrada>>valor>>c;
                        if(c!=','&&c!='}')
                                entrada.clear(ios_base::badbit);
                        else
                                a.agregar(valor);
                }
        }
        return entrada;
}
```

La implementación de los demás métodos se describe a continuación:

```
ArregloDinamico::ArregloDinamico(int n)
{
        longitud=0;
        if(n>0)
        {
                longitud=n;
```

```
                arreglo= new int[longitud];
                for(int i=0;i<longitud;i++)
                        arreglo[i]=0;
        }
}

ArregloDinamico::~ArregloDinamico()
{
        delete arreglo;
}

void ArregloDinamico::agregar(int v)
{
        if(longitud>0)
        {
                int* arreglotemp=new int[longitud+1];
                for(int i=0;i<longitud;i++)
                  arreglotemp[i]=arreglo[i];
                arreglotemp[longitud]=v;
                delete arreglo;
                arreglo= new int[longitud+1];
                for(int i=0;i<=longitud;i++)
                  arreglo[i]=arreglotemp[i];
                delete arreglotemp;
        }
        else
        {
                arreglo= new int[1];
                arreglo[0]=v;
        }
        longitud++;
}
```

```
void ArregloDinamico::quitar(int p)
{
	if(p>=0&&p<longitud)
	{
		int* arreglotemp=new int[longitud-1];
		int t=0;
		for(int i=0;i<longitud;i++)
		{
			if(i!=p)
			{
				arreglotemp[t]=arreglo[i];
				t++;
			}
		}
		delete arreglo;
		arreglo= new int[longitud-1];
		for(int i=0;i<longitud-1;i++)
		  arreglo[i]=arreglotemp[i];
		delete arreglotemp;
		longitud--;
	}
}

void ArregloDinamico::setValor(int p, int v)
{
	if(p>=0&&p<longitud)
		arreglo[p]=v;
}

int ArregloDinamico::getValor(int p) const
{
	if(p>=0&&p<longitud)
		return arreglo[p];
```

```
        return -1;
}

int ArregloDinamico::getLongitud() const
{
        return longitud;
}

ArregloDinamico ArregloDinamico::operator+(const ArregloDinamico &otro)
{
        ArregloDinamico suma(longitud+otro.getLongitud());
        for(int i=0; i<(longitud+otro.getLongitud());i++)
        {
                if(i<longitud)
                        suma.setValor(i,getValor(i));
                else
                        suma.setValor(i,otro.getValor(i-longitud));
        }
        return suma;
}

void ArregloDinamico::setLongitud(int n)
{
        int *temp;
        if(n>0)
        {
                temp=new int[n];
                for(int i=0;i<n;i++)
                {
                        if(i<longitud)
                                temp[i]=getValor(i);
                        else
                                temp[i]=0;
```

```
                }
                longitud=n;
                delete arreglo;
                arreglo=new int[longitud];
                for(int i=0;i<longitud;i++)
                  arreglo[i]=temp[i];
                delete temp;
        }
        else if(n==0)
        {
                delete arreglo;
                longitud=0;
        }
}
```

Referencias

BROKKEN, Frank. C++ Annotations. [Recurso disponible en Línea] <http://sourceforge.net/projects/cppannotations/ > [Con Acceso 2011/02/27].

POZO, Salvador. C++ con clase [en Linea] <http://c.conclase.net/>[Citado en 2011/02/19]

Herencia

Introducción

El presente capítulo tiene como objetivo dar al lector los conceptos relacionados con la Herencia de clases, se definen los conceptos de clases bases y clases derivadas, también se define el concepto de clase base abstracta y jerarquía de clases.

Necesidad de la herencia

Según Javier García, *"La mente humana clasifica los conceptos de acuerdo a dos dimensiones: pertenencia y variedad"*. Se puede decir que el *Ford Fiesta* es un tipo de *coche* (variedad o, en inglés, una relación del tipo *is a*) y que una *rueda* es parte de un *coche* (pertenencia o una relación del tipo *has a*). Antes de la llegada de la herencia, en C ya se había resuelto el problema de la pertenencia mediante las estructuras, que podían ser todo lo complejas que se quisiera. Con la herencia, se consigue clasificar los tipos de datos (*abstracciones*) por variedad, acercando así un paso más la programación al modo de razonar humano.

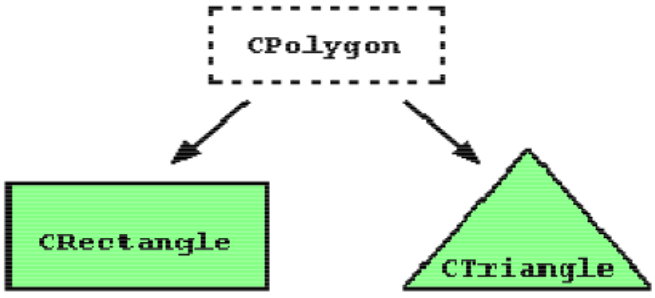
Herencia

Una característica principal de C++ es la herencia. Bajo esta característica es posible crear clases, las cuales son derivadas de otras clases, por ende este incluye de forma automática los miembros de los padres mas los suyos, por ejemplo, supóngase que se van a desarrollar una serie clases que representen polígonos tales como Rectángulo, Triangulo, Etc. Estos últimos tienen propiedades comunes tales como: perímetro, área, largo, ancho.

Esto puede ser representado en el mundo de las clases con una clase llamada *Polígono* en la cual se derivan dos mas: *Triangulo* y *Rectángulo*.

La clase *Polígono* contienen los miembros que son comunes para ambos tipos de polígonos. En nuestro caso: ancho y alto. *Rectángulo* y *Triangulo* serían sus clases derivadas, con características que son diferentes de un tipo de polígono a otro.

Gráficamente podemos expresar dicha característica de la siguiente forma:



La herencia, según Javier García, es entendida como una característica de la programación orientada a objetos y más concretamente del C++, que permite definir una clase modificando una o más clases ya existentes. Estas modificaciones consisten habitualmente en añadir nuevos miembros (atributos u operaciones), a la clase que se está definiendo, aunque también se puede redefinir variables o funciones miembro ya existentes.

La clase de la que se parte en este proceso recibe el nombre de ***clase base***, y la nueva clase que se obtiene se denomina ***clase derivada***. Ésta a su vez puede ser clase base en un nuevo proceso de derivación, iniciando de esta manera una ***jerarquía de clases***. De ordinario las clases base suelen ser más generales que las clases derivadas. Esto es así porque a las clases derivadas se les suelen ir añadiendo características, en definitiva atributos y operaciones que diferencian, concretan y particularizan.

En algunos casos una clase no tiene otra utilidad que la de ser clase base para otras clases que se deriven de ella. A este tipo de clases base, de las que no se declara ningún objeto, se les denomina *clases base abstractas* (*ABC*, *Abstract Base Class*) y su función es la de agrupar miembros comunes de otras clases que se derivarán de ellas. Por ejemplo, se puede definir la clase *vehículo* para después derivar de ella *coche*, *bicicleta*, *avión*, etc., pero todos los objetos que se declaren pertenecerán a alguna de estas últimas clases; no habrá vehículos que sean sólo vehículos.

Las características comunes de estas clases (como un atributo que indique si está parado o en marcha, otro que indique su velocidad, la función de arrancar y la de frenar, etc.), pertenecerán a la clase base y las que sean particulares de alguna de ellas pertenecerán sólo a la clase derivada (por ejemplo el número de platos y piñones, que sólo tiene sentido para una bicicleta, o la función embragar que sólo se aplicará a los vehículos de motor con varias marchas).

Variables y funciones miembros protected

Uno de los problemas que aparece con la herencia es el del control del acceso a los datos. ¿Puede una función de una clase derivada acceder a los datos privados de su clase base? En principio una clase no puede acceder a los datos privados de otra, pero podría ser muy conveniente que una clase derivada accediera a todos los datos de su clase base. Para hacer posible esto, existe el tipo de dato **protected**. Este tipo de datos es privado para todas aquellas clases que no son derivadas, pero público para una clase derivada de la clase en la que se ha definido la variable como **protected**.

Por otra parte, el proceso de herencia puede efectuarse de dos formas distintas: siendo la clase base **public** o **private** para la clase derivada. En el caso de que la clase base sea **public** para la clase derivada, ésta hereda los miembros **public** y **protected** de la clase base como miembros **public** y **protected**, respectivamente. Por el contrario, si la clase base es **private** para la clase derivada, ésta hereda todos los datos de la clase base como **private**. La siguiente tabla puede resumir lo explicado en los dos últimos párrafos.

Para el caso de las Tipos abstractos de datos *Polígono*, *Triangulo* y *Rectángulo* es posible representarlos en C++ como sigue a continuación:

```
class Poligono{
protected:
        int width, height;
public:
```

```
        void set_values (int a, int b)  { width=a; height=b;}
};

class Rectangulo: public Poligono{
public:
        int area () { return (width * height); }
};

class Triangulo: public Poligono{
public:
        int area () { return (width * height / 2); }
};

int main () {
        Rectangulo rect;
        Triangulo trgl;
        rect.set_values (4,5);
        trgl.set_values (4,5);
        cout << rect.area() << trgl.area() << endl;
}
```

Referencias

GARCÍA, Javier Et al. Aprenda C++ como si fuera en primero. Universidad de navarra . San Sebastian - España, abril de 1998.

SOULIÉ, Juan. C++ Language Tutorial. [En línea]<http://www.cplusplus.com/doc/tutorial/> [Con acceso 2011/01/01]

Mas sobre Herencia

Introducción

El presente capitulo tiene como meta profundizar al lector los conceptos relacionados con la Herencia de clases. Se define algunas técnicas de implementación de los constructores y métodos para las clases derivadas, también se define el concepto de Herencia múltiple.

Sintaxis para la derivación de clases

La forma general de declarar clases derivadas es la siguiente:

```
class <clase_derivada> : [public|private] <base1> [,[public|private] <base2>] {};
```

Se puede observar que para cada clase base se pueden definir dos tipos de acceso, public o private. Si no se especifica ninguno de los dos, por defecto se asume que es private.

- public: los miembros heredados de la clase base conservan el tipo de acceso con que fueron declarados en ella.
- private: todos los miembros heredados de la clase base pasan a ser miembros privados en la clase derivada.

De momento se declaran las clases base como public, al menos que exista la utilidad de hacerlo como privadas.

Como ejemplo observe la definición de las clases *Polígono*, *Triangulo* y *Rectángulo*:

```
class Poligono{
    protected:
        int width, height;
    public:
        Poligono(int=0,int=0);
        ~Poligono();
        int area();
        void set_values (int a, int b);
};
```

A continuación se define la clase *Rectángulo* como derivada de *Polígono*:

```
class Rectangulo: public Poligono{
    public:
        Rectangulo(int=0,int=0);
        ~Rectangulo();
        int area();
```

```
};
```

También se define la clase *Rectángulo* como derivada de *Polígono*:

```
class Triangulo: public Poligono{
        public:
                Triangulo(int=0, int=0);
                ~Triangulo();
                int area();
};
```

Constructores de clases derivadas

Cuando se crea un objeto de una clase derivada, primero se invoca al constructor de la clase o clases base y a continuación al constructor de la clase derivada. Si la clase base es a su vez una clase derivada, el proceso se repite recursivamente.

Lógicamente, si no se han definido los constructores de las clases, se usan los constructores por defecto que crea el compilador.

Su sintaxis es la siguiente:

```
<clase_derivada>::<clase_derivada>(<lista_de_parámetros>)
:<clase_base>(<lista_de_parámetros>) {}
```

Si el constructor de la clase base *Polígono* esta implementado de la siguiente forma:

```
Poligono::Poligono(int a, int b){
        set_values(a,b);
}
```

Se puede implementar el constructor de la clase derivada *Rectángulo* como sigue a continuación:

```
Rectangulo::Rectangulo(int a,int b):Poligono(a,b){}
```

Sobrecarga de constructores de clases derivadas

Por supuesto, los constructores de las clases derivadas también pueden sobre cargarse, podemos crear distintos constructores para diferentes inicializaciones posibles, y también usar parámetros con valores por defecto.

Destructores de clases derivadas

Cuando se destruye un objeto de una clase derivada, primero se invoca al destructor de la clase derivada, si existen objetos miembro a continuación se invoca a sus destructores y finalmente al destructor de la clase o clases base. Si la clase base es a su vez una clase derivada, el proceso se repite recursivamente.

Al igual que pasaba con los constructores, si no hemos definido los destructores de las clases, se usan los destructores por defecto que crea el compilador.

Por ejemplo si se define el destructor de la clase Poligono como sigue a continuación:

```
Poligono::~Poligono(){
        cout<<"llamada al destructor: Poligono"<<endl;
}
```

Ahora si se define el destructor de la clase Triangulo como sigue a continuación:

```
Triangulo::~Triangulo(){
        cout<<"llamada al destructor: Triangulo"<<endl;
}
```

Si la función main es la siguiente:

```
int main{
        Triangulo t;
}
```

La salida es la siguiente:

```
llamada al destructor: Triangulo
llamada al destructor: Poligono
```

Redefinición de funciones en clases derivadas

En una clase derivada se puede definir una función que ya existía en la clase base, esto se conoce como "*overriding*", o superposición de una función.

La definición de la función en la clase derivada oculta la definición previa en la clase base.

En caso necesario, es posible acceder a la función oculta de la clase base mediante su nombre completo.

La sintaxis es la siguiente:

```
[<objeto>.]<clase_base>::<método>;
```

Por ejemplo si el método área de la clase Polígono tiene la siguiente implementación:

```
int Poligono::area(){return width * height;}
```

Es posible superponer el método área para la clase Triangulo como se muestra a continuación:

```
int Triangulo::area(){return width * height/2;}
```

Por ende si declaramos un objeto trig de tipo Triangulo este objeto tendría dos definiciones del método área: Triangulo::area() y Poligono::area().

Superposición y sobrecarga.

Cuando se superpone una función, se ocultan todas las funciones con el mismo nombre en la clase

base.

Ahora bien, no es posible acceder a ninguna de las funciones superpuestas de la clase base, aunque tengan distintos valores de retorno o distinto número o tipo de parámetros. Todas las funciones superpuesta (*overrided*) de la clase base han quedado ocultas, y sólo son accesibles mediante el nombre completo.

Por ejemplo si el método área de la clase Polígono tiene la siguiente implementación:

```
int Poligono::area(){return width * height;}
```

Es posible superponer el método área para la clase Triangulo como se muestra a continuación:

```
int Triangulo::area(){return Poligono::area()/2;}
```

Si se declara un objeto trig de la clase Triangulo, cuando se invoca trig.area() estamos invocando al método Triangulo::area() por lo tanto trig.area() es equivalente a:

trig.Triangulo::area()

Si se desea acceder al método de la clase base el cual fue superpuesto lo podemos hacer de la siguiente forma:

trig.Poligono::area()

Herencia Múltiple.

Según Bruce Eckel, El concepto básico de la herencia múltiple (HM) suena bastante simple: puede crear un nuevo tipo heredando de más una una clase base. La sintaxis es exactamente la que espera, y en la medida en que los diagramas de herencia sean simples, la HM puede ser simple también.

Salvador Pozo define la sintaxis de la Herencia múltiple como sigue a continuación:

```
<clase_derivada>(<lista_de_parámetros>) : <clase_base1>(<lista_de_parámetros>)
[, <clase_base2>(<lista_de_parámetros>)] {}
```

A continuación se muestra un ejemplo de derivación múltiple:

```
class ClaseA {
    public:
        ClaseA() : valorA(10) {}
        int darValor() const { return valorA; }
    protected:
        int valorA;
};
```

```
class ClaseB {
        public:
                ClaseB() : valorB(20) {}
                int darValor() const { return valorB; }
        protected:
                int valorB;
};
class ClaseC : public ClaseA, public ClaseB {
        public:
                int darValor() const { return ClaseA:: darValor()+ ClaseB:: darValor();}
};
```

Si la función main es la siguiente:

```
int main(){
        ClaseC c;
        cout<<c. darValor()<<endl;
}
```

La salida es 30.

Referencias

ECKEL, Bruce. Thinking in C++, Volume 1: Introduction to Standard C++ (2nd Edition). Prentice Hall. NJ, USA. 2000.

SOULIÉ, Juan. C++ Language Tutorial. [En línea]<http://www.cplusplus.com/doc/tutorial/> [Con acceso 2011/01/01]

POZO, Salvador. C++ con clase [en Linea] <http://c.conclase.net/>[Citado en 2011/02/19]

Excepciones

Introducción

El presente capitulo tiene como objeto dar al lector los conceptos relacionados con en manejo de excepciones en C++. Partiendo desde el manejo de errores en tiempo de diseño como en tiempo de ejecución, se conocerán varias técnicas partiendo desde el manejo de señales en su predecesor (Lenguaje C) hasta el uso correcto de la tripleta throw, try, catch en el lenguaje C++.

Manejando errores

Hay varias formas posibles para manejar errores en nuestras aplicaciones. A continuación se mencionan algunas técnicas.

Definiendo su comportamiento en términos de una variable lógica.

Según Hayet , la manera mas simple para manejar errores es definiendo el comportamiento erróneo en términos de una variable lógica. Ejemplo la siguiente función verifica que el requerimiento se cumpla, de lo contrario interrumpe la aplicación:

```
void  verificar ( bool requerimiento, const  std : : string & msg = “el requerimiento fallo” ) {
        using namespace std;
        if(! requerimiento) {
                cerr << msg << endl ;
                exit (1);
        }
}
```

De tal forma que en cualquier implementación se pueda verificar el cumplimiento algún requerimiento como se muestra a continuación:

```
int main(){
        int a=120;
        verificar( a <100 , “valor fuera de rango” ) ;
}
```

La macro assert

Existe una función estándar (de C) para hacer este tipo de verificaciones, la macro assert. Cuya definición se muestra a continuación:

```
void assert(int test) ;
```

Esta comprueba la condición 'test' y dependiendo del resultado, puede abortar el programa. Se trata de

una macro que se expande como una sentencia "if", si 'test' se evalúa como cero, la macro aborta el programa y muestra el siguiente mensaje en stderr:

Assertion failed: <condición>, fichero <nombre de fichero>, line <número de línea>

El nombre de archivo y el número de línea corresponden con el archivo y línea en la que está la macro.

Por ejemplo el siguiente código fuente:

```
#include <cassert>
int main ( ) {
   bool d= false ;
   assert (d);
}
```

Produce una descripción mínima de lo que pasó y sale:

```
assert: assert.cpp:4: int main(): Assertion 'd' failed.
Cancelado
```

Ese comportamiento frente errores puede estar bien en fase de desarrollo del programa, pero no en fase de *release*, hay que prever cuando es posible un procesamiento en linea de las errores. En práctica, separar el caso de fase de *debug* de fase “normal” :

```
#include <cassert>
…
bool d;
…
#ifdef DEBUG
        assert (d);
#else
        // Prever el procesamiento de error !
#endif
```

Manejar errores fuera del contexto

Si, dentro de la función o del método en que se produce el error se tiene todo lo necesario para manejar el error y salir “bien” no hay problemas. Si no es el caso y que se tiene que salir del contexto para manejar el error, hay varias maneras de hacerlo en C :

1. Afectar un código de nuestro error al valor de regreso a la función y propagar este código a niveles en que el procesamiento es posible :

```
#define ERR_MALLOC 1
...
```

```
if (!(f=malloc(1000∗sizeof(double)))
    return ERR_MALLOC;
```

2. Usar un sistema de códigos global, que provee por ejemplo C a través de errno y perror.

```
archivo= fopen ( “datos.txt” , ” r ” ) ;
if(!archivo) {
   perror ( “Error tratando de abrir el archivo datos.txt” ) ;
   return −1;
}
```

   Las funciones estándar, en caso de error, ponen la variable global errno, y perror imprime el texto más la descripción del errno.

3. Generar una señal desde el programa frente a un error y escribir los mecanismos para manejarlos:

```
#include <signal.h>
int raise ( int sig );
```

   Manejado con:

```
void sig_catcher ( int signum ) {
   ...
}
int main ( ) {
   // catch signal
   signal ( sig , sig_catcher ) ;
   ...
   raise ( SIGERRVAL ) ;
}
```

Excepciones

En C++, el mecanismo estándar para el manejo de errores es el de excepciones: en caso de errores al momento de la ejecución, y que no se puede corregir localmente, la idea es, en lugar de salir de la función en que estamos, se lanza por fuera del contexto un mensaje de alerta y esperar que algo en otra parte del código lo pueda manejar.

La palabra reservada throw

Para el envío de mensajes que se esperan manejar por fuera del contexto, se define objetos que “lanzar” (excepciones: se puede tomar de cualquier tipo) y la palabra reservada throw nos permite emitirlos:

Supóngase que los objetos de la clase HelpMe son los que van a lanzar en el momento que se presente

una excepción, la definición de esta clase se muestra a continuación:

```
class HelpMe {
        string message;
public:
        HelpMe ( const string &s ) : message ( s ) { } ;
        const string& getMessage ( ) { return message; }
};
```

Si en la función validar tiene como objeto verificar algún requerimiento y se desea manejar el error por fuera de su contexto esta puede definirse de la siguiente forma:

```
void validar ( bool b ) {
        if(!b) {
                throw ( HelpMe ( “Estoy en un problema!! ” ) ) ;
        }
}
```

Características de throw

la palabra reservada throw tiene las siguientes características:

1. Actúa como un return, en el sentido de que regresa un valor, el objeto acompañándolo.
2. Efectúa el cleanup de las variables locales de la función en que se encuentra.
3. No regresa al nivel superior en la pila (a la llamada de la función) sino en un lugar especial de la memoria para manejar excepciones.
4. Enviará preferentemente objetos específicos al error encontrado.
5. Puede o no tomar paréntesis.

El bloque try catch

Conocido el mecanismo para lanzar excepciones, es necesario ver el mecanismo que permite asociar una excepción a un procesamiento. Ejemplo, el código :

```
validar ( false ) ;
```

Genera dicha excepción obteniendo la siguiente salida (u otra dependiendo del compilador):

```
Terminate called after throwing an instance of ’HelpMe’
Abort
```

En cualquier lugar del código de la llamada de la función, se puede definir un bloque try que se prepara a recibir excepciones:

```
try {
   ...
   validar (false) ;
}
```

El bloque try evita salir de la función en que está este bloque : el procesamiento de los errores puede estar definido en este nivel.

El bloque try está seguido por uno o varios bloques catch que están específicos a la excepción generada dentro del bloque try:

```
try {
   validar (false) ;
} catch ( int i ) {
}
cout << “la vida sigue” << endl ;
```

La palabra reservada catch, seguida de un tipo t y de un identificador id, define un bloque donde se va a manejar una excepción id de tipo t.

Para el caso específico, aun genera la siguiente salida (u otra dependiendo del compilador):

```
Terminate called after throwing an instance of ’HelpMe’
Abort
```

Ahora si el catch maneja objetos de tipo HelpMe:

```
try {
   validar (false) ;
} catch ( HelpMe h ) {
   c o u t << “Hola escuché tu mensaje” << “\” ”<<h . getMessage ( ) << ” \” ” << endl ;
}
```

Sigue “normalmente” a la salida del bloque catch.

Más generalmente, se puede tener algo como :

```
try {
...
} catch (ExceptionType1 ex1 ){
...
} catch (ExceptionType2 ex2 ){
...
} catch (ExceptionType3 ex3 ){
...
```

```
} ...
} catch (ExceptionTypeN exN ){
...
}
```

Ese mecanismo es interesante porque :

1. Separa bien el código normal del código encargado de manejar los errores,
2. Permite dejar cualquier código de nivel superior en la pila manejar esas errores,
3. Limpia correctamente al nivel de la función en que se invoca la excepción.

Elección del buen handler

No se necesita correspondencia perfecta entre el tipo de la excepción lanzada y el tipo de recepción por catch, por ejemplo las clases derivadas están puestas en correspondencia (referencia/valor).

Observe la definición de la clase PleaseHelpMe:

```
class PleaseHelpMe : public HelpMe {
        public:
                PleaseHelpMe (const string &s ) : HelpMe (s) { } ;
};
```

El throw con un PleaseHelpMe está manejado en un catch para HelpMe. Por ejemplo, si la función validar es la siguiente:

```
void validar ( bool b ) {
        if(!b) {
                throw PleaseHelpMe ( “Estoy en un problema!! ” )  ;
        }
}
```

Y si el el bloque try - catch maneja objetos de tipo HelpMe:

```
try {
   validar (false) ;
} catch ( HelpMe h ) {
   cout << “Hola escuché tu mensaje” << “ \” ”<<h . getMessage ( ) << ” \” ” << endl ;
}
```

Entonces la salida es la siguiente:

```
Hola escuché tu mensaje “Estoy en un problema!!”
```

Debido a que un objeto de tipo PleaseHelpMe es un objeto de tipo HelpMe.

Es posible hacer un handler que agarre toda excepción generada en el try :

```
try {
        validar (false) ;
} catch ( . . . ) {
        cout << “ha ocurrido una excepción ” << endl;
}
```

Eso puede ser útil cuando hay que separar el procesamiento de los errores en dos partes : una parte genérica que puede incluir liberación de memoria, por ejemplo; una parte especifica que está procesada en un segundo tiempo.

Re-lanzar una excepción

Para implementar ese mecanismo de procesamiento en varios tiempos, se necesita propagar la excepción en diferentes niveles: por eso, se usa otra vez la palabra reservada throw, que re-lanza la excepción hacia niveles superiores,

```
try {
        validar ( false) ;
} catch ( . . . ) {
        cout<<”relanzando la excepcion”;
        throw ;
}
```

Excepciones estándar

El lenguaje C++ provee clases para excepciones, que se puede usar gracias a la inclusión del archivo de cabecera <exception> Todas heredan de la clase exception, que contiene un mensaje a que se puede acceder por el método what(). Varias clases heredan de exception:

1. *logic_error* : errores interceptables en la función que lanza la excepción.
2. *runtime_error* : errores interceptables sólo en el momento de la ejecución (por fuentes externas: falta de memoria. . .)
3. *ios::failure* : errores en la manipulación de iostreams.

Todas esas subclases se construyen con un mensaje string.

Especificaciones de excepciones

Al escribir código que será usado por terceros, puede ser muy útil especificar cuales son las excepciones que hay que esperar de las funciones, o métodos de las clases desarrolladas, Ejemplo:

```
void procesar( ) throw ( HelpMe , PleaseHelpMe ) { //...
}
```

Significa que hay que esperar de esa función excepciones de tipo HelpMe y PleaseHelpMe.

Mientras que:

```
void f ( ) ;
```

significa, por default, que la función puede emitir cualquiera excepción, y

```
void f ( ) throw ( ) ;
```

significa que la función no va a emitir ninguna excepción.

Cuando usar u evitar las excepciones

Cuando se puede evitar usar excepciones:

- En eventos asíncronos,
- En errores que se podrían tratar localmente,
- En programas con imperativos fuertes en términos de tamaño del código objeto,
- En destructores.

Cuando se puede usar excepciones

- En Constructores,
- Para reintentar llamadas a funciones que fallan,
- Para intentar otra cosa al lugar de la función que falla,
- Para compartir el procesamiento de las errores entre varias capas de tratamiento.

Referencias

HAYET. J.B. Programación avanzada en C++ - Excepciones, Centro de investigacion en Matematicas. Mexico [en Linea]<www.cimat.mx/~jbhayet/CLASES/PROGRAMACION/slides/clase17.pdf> [Con acceso el 2011/02/19].

POZO, Salvador. C++ con clase [en Linea] <http://c.conclase.net/>[Citado en 2011/02/19]

Polimorfismo

Introducción

El presente capitulo tiene como objeto dar al lector un concepto fundamental en el paradigma de la programación orientada a objetos tal como es el polimorfismo. Se abordaran temas relacionados como la conversión hacia la clase base (*upcasting*), las llamadas a funciones por enlace temprano como por enlace tardío, las funciones virtuales e introducción al concepto de clase base abstracta.

Upcasting

Una característica muy importante de la herencia es la relación particular entre la clase base y sus clases derivadas que hace que una referencia a una instancia de una clase derivada puede "estar vista" sin problema como una referencia a una instancia de la clase base.

Por ejemplo observe las definiciones de las clases Poligono, Rectángulo y Triangulo:

```
class Poligono{
        protected:
                int width, height;
        public:
                Poligono(int=0,int=0);
                void set_values (int a, int b);
                int getWidth();
                int getHeight();
                std::string getName();
};

class Rectangulo: public Poligono{
        public:
                Rectangulo(int,int);
                int area ();
                std::string getName();
};
```

```
class Triangulo: public Poligono{
        public:
                Triangulo(int, int);
                int area ();
                std::string getName();
};
```

Es posible definir una función llamada dibujar cuyo objeto es dibuja un polígono en la librería gráfica X, en el cual recibe como parámetro por referencia un objeto de tipo Poligono

```
void dibujar(const Poligono& p){
        …
}
```

Si la función main es la siguiente:

```
int main(){
        Triangulo t(1,2);
        dibujar(t);
}
```

El uso de la función dibujar, recibiendo como parámetro un objeto de tipo Triangulo, es completamente válido.

Este mecanismo es lógico ya que todos los métodos (públicos) que están eventualmente llamados dentro de la función desde un objeto de tipo ***Poligono*** están disponibles, por definición, para objetos de tipo ***Triangulo***.

El *upcasting* es la razón de peso que permite elegir herencia sobre composición: en unos casos es posible querer manipular referencias a objetos de una clase derivada como si fueran referencias a objetos de la clase “genérica”.

El *upcasting* en termino general se aplica en asignaciones de referencias o apuntadores, por ejemplo:

```
int main () {
        Rectangulo rect;
        Triangulo trgl;
        Poligono &r= rect;
        Poligono *t=&trgl;
        r->set_values(4,5);
        t->set_values(4,5);
}
```

Vale la pena aclarar que cuando se usa una referencia a la clase base, al invocar un método este aplica el comportamiento definido para la clase base por ejemplo si en la en las siguientes sentencias:

```
Triangulo trgl;
Poligono &t= trgl;
t.getName();
```

Aunque la clase Triangulo tenga definido un método getName(), El compilador llamará a priori, al método definido en la clase Poligono (base), debido a que es la única es visible en tiempo de compilación.

En C++ hay un mecanismo que permite realizar *upcasting* pero conservando de una manera u otra la "memoria" de que venimos de una clase derivada particular, y entonces llamar los métodos de ésta, este mecanismo se llama funciones virtuales.

Llamadas a funciones

Las llamadas a funciones pueden hacerse de dos formas; ya sea en tiempo de compilación (*early binding*) o en tiempo de ejecución (*late binding*)

Enlace temprano

C++ por defecto, en tiempo de compilación, traduce las llamadas a la funciones por código objeto con la dirección memoria de la función a llamar, de tal forma que hará el programa ir a ejecutar las instrucciones de esa función en particular (con los argumentos dados), Este proceso se conoce como Enlace temprano (*early binding*) ante esto no hay elección posible, es una y una sola función que será invocada:

Eso explica el comportamiento del programa al llamar un método después de un upcasting:

```
Triangulo trgl;
Poligono &t= trgl;
t.getName(); //llama el de poligono, a priori.
```

Enlace Tardío

Ahora, conceptualmente, la única manera de poder llamar al método de correcto después de realizar un *upcasting* será resolviendo la llamada al método, no en tiempo de compilación sino en tiempo de ejecución, este mecanismo se conoce como Enlace Tardío (*late binding*).

Eso implica que el programa pueda verificar de que tipo se está tratando realmente (sea el método de la clase bases o una de sus derivadas). Hasta ahora no tenemos lo suficiente, pero el C++ provee lo necesario para hacerlo.

Funciones virtuales

Para poder llamar a alguna función en enlace tardío, se necesita usar, en la clase base, la palabra reservada virtual con la declaración de la función sobre la que queremos llamar en enlace tardío.

Ejemplo, si se desea llamar en tiempo de ejecución el método getName(), entonces en la definición de la clase Polígono (Clase base), debemos anteponer la palabra reservada virtual en la definición del método getName() quedando de la siguiente forma:

```
class Poligono{
        protected:
                int width, height;
        public:
                Poligono(int,int);
                void set_values (int a, int b);
                int getWidth();
                int  getHeight();
                virtual std::string getName();
};
```

Eso hará que toda llamada a getName(), dentro de una clase derivada que lo define, a partir de una referencia a Poligono obtenida por *upcasting*, llamará al método getName() de la clase derivada.

Extensibilidad

La Extensibilidad es uno de los importantes beneficios del polimorfismo ya que si tengo una parte del programa escrita en términos de una versión genérica de mis objetos (por ejemplo, en términos de la clase base), es posible, sin problema escribir versiones especializadas de la clase base (a través de clases derivadas) y usarlas en esta parte del código, sin realizarle cambios a este ultimo.

En ese aspecto no es necesario saber cuales son las derivadas, ya que el código definido con los mecanismos adecuados seguirá funcionando.

Clases abstractas

Existe en C++ la posibilidad de Convertir de una clase en clase abstracta, o sea que no pueda ser implementada o instanciada. Para eso, es suficiente declarar una de sus funciones virtuales como función virtual pura, añadiendo a su declaración un “=0” .

Ejemplo:

```
class Figura {
        public :
                virtual int getArea () = 0;
};
```

Los métodos virtuales puros obligan a implementar versiones de esos métodos en las clases derivadas con el fin de facilitar las siguientes practicas de programación:

- Ofrece una manera de separar aún mas interfaz de implementación, esta vez a través de varias clases (clases abstractas/clases “normales”).
- Ofrece una manera de obligar los que van a heredar las clases que funciones implementar.

Referencias

HAYET. J.B. Programación avanzada en C++ - Herencia, Centro de investigación en Matemáticas. México [en Linea] <www.cimat.mx/~jbhayet/CLASES/PROGRAMACION/slides/clase13.pdf> [Con acceso el 2011/02/19].

HAYET. J.B. Programación avanzada en C++ - Polimorfismo, Centro de investigación en Matemáticas. México [en Linea]<www.cimat.mx/~jbhayet/CLASES/PROGRAMACION/slides/clase14.pdf> [Con acceso el 2011/02/19].

Plantillas de funciones y clases

Introducción

El presente capitulo tiene como objeto dar al lector los conceptos relacionados con la programación genérica, así como los mecanismos que permiten implementarlo en el lenguaje de programación C++ como son las plantillas, aplicado tanto en funciones como clases.

Programación genérica

La programación genérica[7] es un paradigma de programación el cual permite desarrollar bibliotecas sean de funciones o clases eficientes y reutilizable. Este paradigma fue desarrollado por primera vez por Alexander Stepanov y David Musser. La programación genérica tuvo su mayor éxito cuando las bibliotecas de plantillas estándar (*Standard Template Library - STL*) pasó a ser parte del estándar ANSI/ISO C++, desde entonces la programación genérica ha sido utilizada con el fin de desarrollar un sin numero de bibliotecas genéricas.

La programación genérica hace referencia a las características de determinados lenguajes que permite, desarrollar un sin número de aplicaciones de diferentes dominios tratando en lo posible utilizar un conjunto de librerías que se ajusten al dominio de la aplicación sin necesidad de reescribirlas.

Por ejemplo, en C++, una plantilla es una rutina en la que algunos parámetros son calificados por un tipo de variable. La generación de código en C++ depende de los tipos de concreto, la plantilla está especializada para cada combinación de tipos de argumentos que se producen en algunas instancias.

Plantillas

Hayet en su curso de programación avanzada en C++ afirma que: “Hasta ahora, las herramientas que vimos (herencia, composición, polimorfismo), permiten la reutilización de código objeto; los patrones (o *templates*) permiten reutilizar código fuente en otros contextos que el para que estaba diseñado”[8].

En nuestras aplicaciones es muy común reescribir funciones que tienen la misma funcionalidad pero con diferentes argumentos:

Por ejemplo para el diseño de una calculadora se definen las funciones:

```
int sumar(int,int);

float  sumar(float, float);

int sumar(int, float);

float sumar(float, int);
```

Con el fin de sumar dos argumentos sean de tipo int o tipo float respectivamente, como observa la función sumar, para los cuatro casos, pueden definirse, en términos generales de la siguiente forma:

---

7 Generic Programming. Indiana University. [Recurso disponible en linea]<http://www.generic-programming.org/> [con acceso 2011/04/23]

8 HAYET. J.B. Programación avanzada en C++ - Plantillas, Centro de investigación en Matemáticas. México [en Linea]<http://www.cimat.mx/~jbhayet/CLASES/PROGRAMACION/slides/clase16.pdf>[Con acceso el 2011/02/19].

```
sumar(a,b){ return a+b;}
```

Observe que los argumentos a y b pueden pertenecer a dominios diferentes pero la función se cumplirá siempre y cuando se defina, para ambos argumentos la operación +

Entonces, en C++, se puede resumir las cuatro operaciones *sumar* en una sola haciendo uso de una plantilla de función.

Plantillas de función

Una plantilla de función son funciones especiales que operan con tipos de datos genéricos, esto permite crear un conjunto de funciones cuya funcionalidad puede ser adoptada por mas de un tipo de dato o clases sin sobrecargar la función para cada tipo o clase que reciba como argumentos.

En C++, se puede lograr, haciendo uso de parámetros de plantillas. Un parámetro de plantilla es una clase de parámetros especiales que pueden ser usados para pasarles un tipo de datos como argumentos. Las plantillas de función pueden usar esos parámetros como si fuesen variables de funciones normales.

La sintaxis para declarar una plantilla de función con sus parámetros de plantilla es la siguiente:

```
template <class identifier> function_declaration;
template <typename identifier> function_declaration;
```

Si se quiere que la función *sumar* pueda operar con tipos de datos genéricos se puede definir su prototipo de la siguiente forma:

```
template <typename tipo>
tipo sumar(tipo,tipo);
```

Mientras que su implementación se puede expresar de la siguiente forma:

```
template <typename tipo>
tipo sumar(tipo a, tipo b){
    return a+b;
}
```

Su uso no difiere de cualquier otra función, por ejemplo, observe su uso en la función main con diferentes tipos de datos:

```
int main(){
    cout<<sumar(1, 2)<<endl;  //muestra 3
    cout<<sumar(1.0, 1.1)<<endl;  //muestra 2.1
    cout<<sumar(string("a"), string("b"))<<endl; //muestra ab
    return 0;
}
```

En el ejemplo se observa que se ha definido e implementado una función con *tipo* como su parámetro de plantillas, este parámetro de plantillas representa un tipo que no ha sido especificado (forma implícita). En este contexto, la función sumar retorna la suma de dos parámetros de tipos aun

desconocidos.

Si se desea utilizar la plantilla de función para un tipo específico, se puede explicitar el tipo de dato para la plantilla de acuerdo a la siguiente sintaxis.

```
function_name <type> (parameters);
```

Por ejemplo, si se desea usar la plantilla de función para que sus parámetros sean de tipo int se puede usar de la siguiente forma:

```
int main(){
        int x=1, y=2;
        int z=sumar<int>(x,y);
        return 0;
}
```

Plantillas de función con varios parámetros de plantillas

El ejemplo de la plantilla de función *sumar* definida anteriormente solo puede utilizar para usarse con parámetros del mismo tipo, para el diseño de la calculadora, resolverían la definición y uso de las siguientes funciones:

```
int sumar(int,int);
float  sumar(float, float);
```

Mas no resolverían la definición y su posterior uso de las restantes funciones a saber:

```
int sumar(int, float);
float sumar(float, int);
```

Entonces para resolver este problema, se puede redefinir la plantilla de función de tal forma que se puedan usar varios tipos como sigue a continuación:

```
template <typename T, typename U>
T sumar(T, U);
```

Mientras que su implementación se puede expresar de la siguiente forma:

```
template <typename T, typename U>
T sumar(T a, U b){
        return a+b;
}
```

En este caso, la plantilla de función tiene dos parámetros de plantillas de diferentes tipos y retorna el tipo de dato definido por el primer parámetro de plantilla. Su uso puede ser de forma implícita o explicita tal como se muestra a continuación:

```
sumar(1,1.2);
sumar<int,float>(1,1.2);
```

De forma implícita una plantilla de función con dos parámetros de plantillas <T,U>, puede comportarse en una plantilla de función con un mismo tipo de dato si T=U.

Plantillas de clases

También existe la posibilidad de implementar plantillas de clases, es decir clases que tengan, como miembros, atributos o métodos que usen parámetros de plantillas como tipos.

Por ejemplo, note el uso de una clase Pareja cuyo objetivo es obtener una dupla de cualquier clase genérica.

Su definición se muestra a continuación:

```
template <class T>
class Pareja
{
        private:
                T a;
                T b;
        public:
                Pareja(T,T);
                T getA();
                T getB();
};
```

Mientras que la implementación de sus métodos se muestran a continuación:

```
template <class T>
Pareja<T>::Pareja(T a1, T a2): a(a1),b(a2){}
template <class T>
T Pareja<T>::getA(){ return a;}
template <class T>
T Pareja<T>::getB(){ return b;}
```

La clase Pareja se ha definido con el fin de almacenar dos elementos de cualquier clase o tipo de datos (vale la pena recordar que en C++ los tipos de datos son clases). Para su uso es necesario explicitar el tipo de dato o clase. Note los siguientes ejemplos:

```
Pareja<int> mi_par(1,2); //par ordenado de tipo int
Pareja<double> dpar(2.0,4.2); //par ordenado de tipo double
Pareja<Persona> par(p,q); //par ordenado de Personas
Pareja<bool> logic_par(true,false); //par ordenado de bool
```

Especialización de plantillas

Es posible definir diferentes implementaciones para una plantilla cuando reciba un tipo específico vía parámetros de plantillas.

Por ejemplo, se define una plantilla de clases llamada Dato, cuyo objetivo es almacenar cualquier tipo de datos:

```
// Plantilla de clase:
template <class T>
class Dato {
        T elemento;
    public:
        Dato (T arg) {elemento=arg;}
        T incrementar () {return ++elemento;}
};
```

Es posible especializar la plantilla de clases Dato, de tal forma que cuando el tipo de dato que recibe el parámetro de plantilla sea char, esté presente un método llamado *mayus()* cuyo objeto es obtener la letra mayúscula del elemento.

```
// Especialización de la plantilla de clase:
template <>
class Dato <char> {
        char elemento;
    public:
        Dato (char arg) {elemento=arg;}
        char mayus (){
            if ((elemento>='a')&&(elemento<='z'))
                elemento+='A'-'a';
            return elemento;
        }
};
```

Note el uso de la palabra reservada template sin tipo de parámetros en la cabecera de la clase, debido a que este se define de forma explicita (char). Observe sus respectivos usos en la siguiente función main:

```
int main () {
    Dato<int> myint (7);
    Dato<char> mychar ('j');
    cout << myint.incrementar() << “,” << mychar.mayus() << endl;
```

```
}
```

Plantillas sin tipo de parámetros

Es posible inicializar los parámetros de plantillas sea con un tipo de dato, clase u valor válido, de tal forma que pueden usarse plantillas sin parámetros de plantillas. Por ejemplo note la inicialización de la cabecera de la plantilla de clase Pareja.

```
template <class T=int>
class Pareja
{
        private:
                T a;
                T b;
        public:
                Pareja(T,T);
                T getA();
                T getB();
};
```

La siguiente función main muestra la declaración objetos usando la plantilla de clase Pareja sin parámetros:

```
int main()
{
        Pareja<float> mi_par(1,2);
        Pareja<> par(3,4);  //Plantilla sin parámetros, por defecto int
        …
}
```

Referencias

SOULIÉ, Juan. C++ Language Tutorial. [En línea]<http://www.cplusplus.com/doc/tutorial/> [Con acceso 2011/01/01]

Biblioteca estándar de plantillas

Introducción

El presente capitulo tiene como objetivo dar una introducción y posterior uso sobre un gran conjunto de estructuras de datos y algoritmos los cuales conforman las Biblioteca estándar de plantilla (*Standard Template Library – STL*). Se exploraran los elementos de la librería STL de acuerdo a su funcionalidad.

La biblioteca de plantillas estándar

La biblioteca de plantillas estándar son un conjunto de bibliotecas que constan de contenedores, algoritmos genéricos, iteradores, objetos funciones, asignadores, adaptadores y estructuras de datos.

Los algoritmos y las estructuras de datos usadas dentro del a biblioteca son abstractos, en el sentido de que estos pueden ser usados, prácticamente, con cualquier tipo de datos.

Especificaciones básicas para el uso de las STL

Clases agradables

Una clase agradable (*nice class*) es cualquier clase la cual cumpla con los siguientes requerimientos:

1) Constructor de copia
2) Operador de asignación
3) Operador de igualdad
4) Operador de desigualdad

La sintaxis de su definición se muestra a continuación:

```
class Nice{
        public:
                Nice(const Nice &Copy);
                Nice &operator= (const Nice &Copy);
                bool operator== (const Nice &param) const;
                bool operator!= (const Nice &param) const;
};
```

Para el uso de las STL es recomendable que las clases la cual van a manipular cumplan los requisitos de la clases agradables. Aunque actualmente la biblioteca de plantillas tiene definido los operadores relacionales (==, !=, <, <=, >, >=) lo ideal es que se definan de acuerdo al contexto de nuestras clases.

Objeto función

Es un objeto de una clase que tiene definido el operador de llamada a función (operator()), por ejemplo,

observe la definición de la siguiente clase:

```
class less {
        public:
                less (int v) : val (v) {}
                bool operator () (int v) {
                        return v < val;
                }
        private:
                int val;
};
```

Esta clase permite saber si un objeto o valor de función es menor que el entero el cual contiene. Por ejemplo note su uso en la función main:

```
int main(){
        less menor_que(5);
        int z=5;
        if(menor_que(z))  //uso del operador de llamada a función
                cout<<"el numero es menor que 5"<<endl;
}
```

Componentes de la biblioteca estándar de plantillas

La STL se puede considerar como una biblioteca de componentes de software, se define por componente de software un conjunto de objetos, funciones, etc generalmente precompilados con interfaces bien definidas y listos para ser usados en diferentes contextos.

La STL esta compuesta por 5 componentes principales:

1) Contenedores: Objetos que permiten administras otros objetos.
2) Algoritmos: procedimiento computacional que opera sobre diferentes contenedores.
3) Iteradores: algoritmos de accesos aplicados sobre contenedores.
4) Objetos función: clases que tienen definido el operador de llamada a función.
5) Adaptadores: Permiten encapsular un componente para ofrecer otras interfaces.

Principales componentes de la STL[9]

Es muy importante elegir una clase de la STL que esté de acuerdo con lo que se necesita. Ciertas

9 Kioskera, Introducción a la STL en C++ [recurso disponible en linea]<http://es.kioskea.net/faq/3168-introduccion-a-la-stl-en-c-standard-template-library>[con acceso: 2011/05/]

estructuras son más eficaces que otras para acceder a memoria o en términos de reasignación de memoria cuando se hacen importantes. El desafío de esta parte consiste en presentar las ventajas y desventajas de cada una de ellas.

Previamente es necesario tener algunas nociones de dificultad. Sea n el tamaño de un contenedor. Un algoritmo es llamado lineal (en *O(n)*) si su tiempo de calculo es proporcional a n. Igualmente, un algoritmo puede ser instantáneo (*O(1)*), logarítmico *O(log(n))*, polinomial *O(n^k)*, exponencial *O(e(n))*, etc...

Para resumir, a continuación la lista de dificultades en orden creciente de la proporción de tiempo de cálculo:

- *O(1)*
- *O(log(n))*
- *O(n)*
- *O(n^k)*
- *O(e(n))*

Contenedores

A continuación se da una breve revisión de los contenedores mostrando el interés, principalmente a la dificultad de acceso (búsqueda) a un dato almacenado en un contenedor y a la dificultad para insertar un dato.

std::pair<T1,T2>

Un par es una estructura conteniendo dos elementos eventualmente de tipos diferentes. Ciertos algoritmos de la STL (por ejemplo find) devuelven pares (posición del elemento encontrado y un booleano que indica si ha sido encontrado).

Dificultad: inserción y acceso en O(1)

```
#include <pair>

// en la práctica este include es sobreentendido ya que implícitamente

//se hace cuando utilizamos <vector>, <set> ...

#include <iostream>

#include <string>

int main(){

  std::pair<int,std::string> p = std::make_pair(5,"pouet");

  std::cout << p.first << ' ' << p.second << std::endl;

  return 0;

}
```

std::list<T,...>

La clase list provee una estructura genérica de listas enlazadas pudiendo eventualmente contener repeticiones.

Dificultad

- Inserción (al inicio o fin de lista): O(1)
- Búsqueda: O(n) en general, O(1) para el primer y el ultimo eslabón

Este ejemplo muestra cómo insertar los valores 4,5,4,1 en una lista y cómo mostrar su contenido:

```
#include <list>
#include <iostream>


int main(){
      std::list<int> mi_lista;
      mi_lista.push_back(4);
      mi_lista.push_back(5);
      mi_lista.push_back(4);
      mi_lista.push_back(1);
      std::list<int>::const_iterator
            lit (mi_lista.begin()),lend(mi_lista.end());
      for(;lit!=lend;++lit) std::cout << *lit << ' ';
            std::cout << std::endl;
      return 0;
}
```

std::vector<T,...>

La clase vector es muy similar a la de array de C. Todos los elementos contenidos en el vector están contiguos en memoria, lo que permite acceder inmediatamente a cualquier elemento. La ventaja de vector comparada a la de array de C es su capacidad a reasignarse automáticamente en caso de que sea necesario, por ejemplo durante un push_back. Sin embargo al igual que el array clásico, el operador [] únicamente puede acceder a una casilla si ha sido asignada previamente (si no se produce un error de segmentación).

Dificultad:

- Acceso O(1)
- Inserción: O(n) al inicio del vector (pop_back), O(1) al final del vector (push_back). En los dos

casos puede ocurrir una reasignación.

No hay que olvidar que una reasignación de memoria es costosa en términos de tiempo de cálculo. Así, si el tamaño de un vector es conocido de antemano, en lo posible hay que crearlo directamente con este tamaño. Ejemplo:

```
#include <vector>
#include <iostream>
int main(){
      std::vector<int> mi_vector;
      mi_vector.push_back(4);
      mi_vector.push_back(2);
      mi_vector.push_back(5);
      // para recorrer un vector podemos utilizar los iteradores o los
      //índices
      for(std::size_t i=0;i<mi_vector.size();++i)
            std::cout << mi_vector[i] << ' ';
      std::cout << std::endl;

      std::vector<int> v(5,69); // crea el vector 69,69,69,69,69
      v[0] = 5;
      v[1] = 3;
      v[2] = 7;
      v[3] = 4;
      v[4] = 8;
      return 0;
}
```

std::set<T,...>

La clase set permite describir un conjunto ordenado y sin repetición de elementos. Previamente es necesario parar este orden como parámetro template un objeto función. Por defecto, el objeto función std::less (basado en el operador <) es utilizado, lo que equivale a tener un conjunto de elementos clasificados del más pequeño al más grande. Concretamente, basta con implementar el operador < de una clase o una estructura de tipo T para poder definir un std::set<T>. Además, el tipo T debe disponer de un constructor vacío T().

Dificultad:

- O(log(n)) para la búsqueda e inserción. Efectivamente, la estructura std::set saca partido del orden sobre T para construir una estructura de árbol roja y negra, lo que permite la búsqueda logarítmica de un elemento

```
#include <set>
#include <iostream>

int main(){
	std::set<int> s; // equivale a std::set<int,std::less<int> >
	s.insert(2); // s contiene 2
	s.insert(5); // s contiene 2 5
	s.insert(2); // el repetido no es insertado
	s.insert(1); // s contiene 1 2 5
	std::set<int>::const_iterator sit (s.begin()), send(s.end());
	for(;sit!=send;++sit) std::cout << *sit << ' ';
		std::cout << std::endl;
	return 0;
}
```

Atención: el eliminar o agregar un elemento en un std::set vuelve invalido a sus iteradores. No se debe modificar un std::set en un bucle for basado en sus iteradores.

std::map<K,T,...>

Un map permite asociar una clave a un dato. El map toma al menos dos parámetros templates:

- el tipo de la clave K
- el tipo del dato T

Al igual que std::set, el tipo K debe ser ordenado (este orden puede ser pasado como 3er parámetro template, std::less<K> por defecto) y, el tipo T solo impone tener un constructor vacío.

Dificultad:

- O(log(n)) para la búsqueda e inserción. En efecto, la estructura std::map saca partido del orden sobre T para construir una estructura de árbol rojo y negro, lo que permite una búsqueda logarítmica de un elemento.

Atención: el eliminar o agregar un elemento en un std::map vuelve invalido a sus iteradores. No se

debe modificar un std::map en una bucle for basada en sus iteradores.

Atención: el hecho de acceder a una clave vía el operador [] inserta esta clave (con el dato T()) en el map. Por ello, el operador [] no es adaptado para verificar si una clave está presente en el map, es necesario utilizar el método find. Además no asegura la constancia del map (a causa de las potenciales inserciones) y por lo tanto no puede ser utilizado en un const std::map.

Ejemplo:

```
#include <map>
#include <string>
#include <iostream>

int main(){
      std::map<std::string,unsigned> map_mes_idx;
      map_mes_idx["enero"] = 1;
      map_mes_idx["febrero"] = 2;
      std::map<std::string,unsigned>::const_iterator
            mit (map_mes_idx.begin());
      std::map<std::string,unsigned>::const_iterator
            mend(map_mes_idx.end());
      for(;mit!=mend;++mit)
            std::cout << mit->first << '\t' << mit->second << std::endl;
      return 0;
}
```

Los iteradores

En la sección anterior se observó que los iteradores permiten recorrer fácilmente una estructura de la STL. Un iterador recuerda un poco la noción de puntero, pero no es una dirección. A continuación se observan cuatro iteradores clásicos de la STL.

Estos son definido para todas las clases de la STL dichas anteriormente (entre ellas por supuesto std::pair)

iterator y const_iterator

Un iterador (y un const_iterator) permite recorrer un contenedor de inicio a fin. Un const_iterator contrariamente a un iterator, da acceso únicamente para la lectura del elemento deseado. Así, un recorrido con const_iterator no produce cambios en el contenedor. Es por ello que un contenedor “const” puede ser recorrido por const_iterators pero no por iterators.

En general, cuando se debe elegir entre iterators y const_iterators, siempre hay que preferir los const_iterators ya que estos vuelven la sección de código a la cual sirven más genérica (aplicable a los contenedores const o no const)

- begin(): devuelve un iterador que apunta al primer elemento
- end(): devuelve un iterador que apunta justo después del ultimo elemento
- ++: Permite incrementar el iterador haciéndolo pasar al elemento siguiente.

Ejemplo:

```
#include <list>
#include <iostream>

const std::list<int> crear_lista(){
      std::list<int> l;
      l.push_back(3);
      l.push_back(4);
      return l;
}

int main(){
      const std::list<int> mi_lista(crear_lista());
      std::list<int>::const_iterator
            lit2 (mi_lista.begin()), lend2(mi_lista.end());
      for(;lit2!=lend2;++lit2)
            std::cout << *lit2 << ' ';
      std::cout << std::endl;
      return 0;
}
```

reverse_iterator y const_reverse_iterator

El principio de reverse_iterator y const_reverse_iterator es similar a los iterators y const_iterator pero el recorrido se hace en el sentido opuesto.

Utilizamos:

- rbegin() : devuelve un iterador que apunta hacia el último elemento

- rend() : devuelve un iterador que apunta justo antes del primer elemento
- -- : permite disminuir el reverse_iterator haciéndolo pasar al elemento precedente.

```
#include <set>
#include <iostream>

int main(){
  std::set<unsigned> s;
  s.insert(1); // s = {1}
  s.insert(4); // s = {1, 4}
  s.insert(3); // s = {1, 3, 4}
  std::set<unsigned>::const_reverse_iterator
      sit (s.rbegin()), send(s.rend());
  for(;sit!=send;++sit)
      std::cout << *sit << std::endl;
  return 0;
}
```

... muestra:

4

3

1

Algoritmos

La biblioteca estándar de plantillas ofrece una variedad diversas de algoritmos. Estos estan presentes en el archivo de cabecera `<algorithm>` los cuales se mencionan a continuación[10]:

operaciones sobre secuencias:

for_each	Aplica una funcion sobre un rango (plantilla de función)
find	Busca un valor dentro del rango (plantilla de función)
find_if	Busca un elemento dentro del rango (plantilla de función)
find_end	Busca la ultima subsecuencia dentro del rango (plantilla de función)
find_first_of	Busca la primer registro de un valor dentro de un rango (plantilla de función)

10 C++ Reference – Algorithms [recurso disponible en linea]<http://www.cplusplus.com/reference/algorithm/>[con acceso: 2011/05/01]

adjacent_find	Buscar la igualdad de elementos adyacentes en el rango (plantilla de función)
count	Cuenta las apariciones de un valor dentro de un rango (plantilla de función)
count_if	Retorna un numero de elementos los cuales cumplen cierta condición (plantilla de función)
mismatch	Devuelve la primera posición en la cual dos rangos difieren (plantilla de función)
equal	Comprueba si los elementos en dos rangos son iguales (plantilla de función)
search	Busca subsecuencias en el rango (plantilla de función)
search_n	Busca la sucesion de valores iguales dentro de un rango (plantilla de función)

Operaciones de modificación de secuencias:

copy	Copia un rango de elementos (plantilla de función)
copy_backward	Copia rango de elementos hacia atrás (plantilla de función)
swap	Intercambia los valores de dos elementos (plantilla de función)
swap_ranges	Intercambia los valores de dos rangos (plantilla de función)
iter_swap	Intercambia el valor de un rango mediante iteradores (plantilla de función)
transform	Aplica la función a un rango (plantilla de función)
replace	Remplaza un valor dentro de un rango (plantilla de función)
replace_if	Remplaza un valor de acuerdo a una condición (plantilla de función)
replace_copy	Copia rango de sustitución de valores (plantilla de función)
replace_copy_if	Copia rango de sustitución de valores (plantilla de función)
fill	Llena un rango con valores (plantilla de función)
fill_n	Llena un rango con valores (plantilla de función)
generate	Genera valores para un rango (plantilla de función)
generate_n	Genera valores por secuencias (plantilla de función)
remove	Remueve valores de un rango (plantilla de función)
remove_if	Remueve valores de un rango (plantilla de función)
remove_copy	Copia un rango removiendo los anteriores valores (plantilla de función)
remove_copy_if	Copia un rango removiendo los anteriores valores (plantilla de función)
unique	Remueve valores duplicados (plantilla de función)
unique_copy	Copia un rango sin valores repetidos (plantilla de función)
reverse	Invierte un rango de valores (plantilla de función)
reverse_copy	Copia un rango invertido (plantilla de función)
rotate	Rota los elementos dentro de un rango (plantilla de función)
rotate_copy	Copia un rango rotado (plantilla de función)
random_shuffle	Reorganiza los elementos de forma aleatoria (plantilla de función)
partition	Divide un rango en dos (plantilla de función)
stable_partition	Divide un rango en dos ordenamiento estable (plantilla de función)

Ordenamiento:

sort	Ordena los elementos dentro de un rango (plantilla de función)

stable_sort Ordena los elementos manteniendo el orden de equivalencia (plantilla de función)
partial_sort Ordena parcialmente los elementos dentro de un rango (plantilla de función)
partial_sort_copy Copia un rango ordenado parcialmente (plantilla de función)
nth_element Ordena los elementos dentro de un rango (plantilla de función)

Busqueda Binario (operaciones sobre rangos ordenados):
lower_bound Retorna un iterador del limite inferior (plantilla de función)
upper_bound Retorna un iterador del limite inferior (plantilla de función)
equal_range Obtiene subrangos de iguales elementos (plantilla de función)
binary_search Prueba si un valor esta dentro de un rango ordenado (plantilla de función)

Merge (operaciones sobre rangos ordenados):
merge Combinan datos ordenados (plantilla de función)
inplace_merge Combinan datos de varios rangos ordenados (plantilla de función)
includes Prueba si un rango ordenados incluye otra serie ordenada (plantilla de función)
set_union Une dos rangos ordenados (plantilla de función)
set_intersection Intercepta dos rangos ordenados (plantilla de función)
set_difference Obtiene la diferencia de dos rangos ordenados (plantilla de función)
set_symmetric_difference Obtiene la diferencia simetrica de dos datos ordenados (plantilla de función)

Heap:
push_heap Mete elementos dentro de un rango (plantilla de función)
pop_heap Saca elementos dentro de un rango (plantilla de función)
make_heap Crea un montículo dentro de un rango (plantilla de función)
sort_heap Ordena los elementos dentro de un montículo (plantilla de función)

Min/max:
min Returna el menor de dos argumentos (plantilla de función)
max Retorna el mayor de dos argumentos (plantilla de función)
min_element Retorna el menor elemento dentro de un rango (plantilla de función)
max_element Retorna el menor elemento dentro de un rango (plantilla de función)
lexicographical_compare Compara lexicográficamente dos argumentos (plantilla de función)
next_permutation Transforma los datos a una próxima permutación (plantilla de función)
prev_permutation Transforma los datos a una próxima permutación (plantilla de función)

Referencias

SOULIÉ, Juan. C++ Language Tutorial. [En línea]<http://www.cplusplus.com/doc/tutorial/> [Con acceso 2011/01/01].

C++ Reference – Algorithms [En linea]<http://www.cplusplus.com/reference/algorithm/>[con acceso: 2011/05/01].

Kioskera, Introducción a la STL en C++ [En linea]<http://es.kioskea.net/faq/3168-introduccion-a-la-stl-en-c-standard-template-library>[con acceso: 2011/05/01]