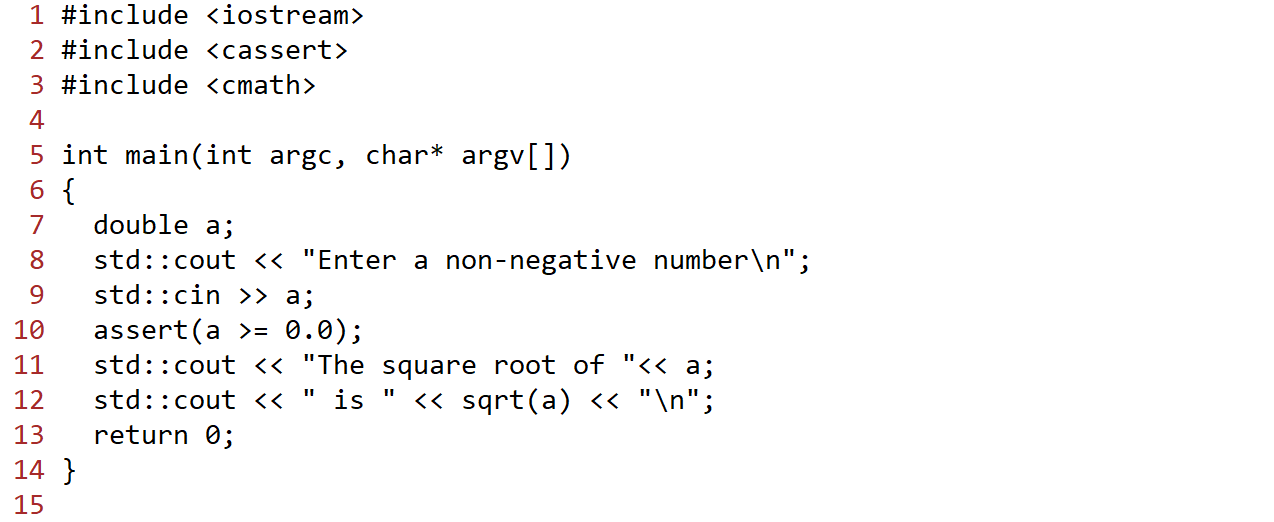
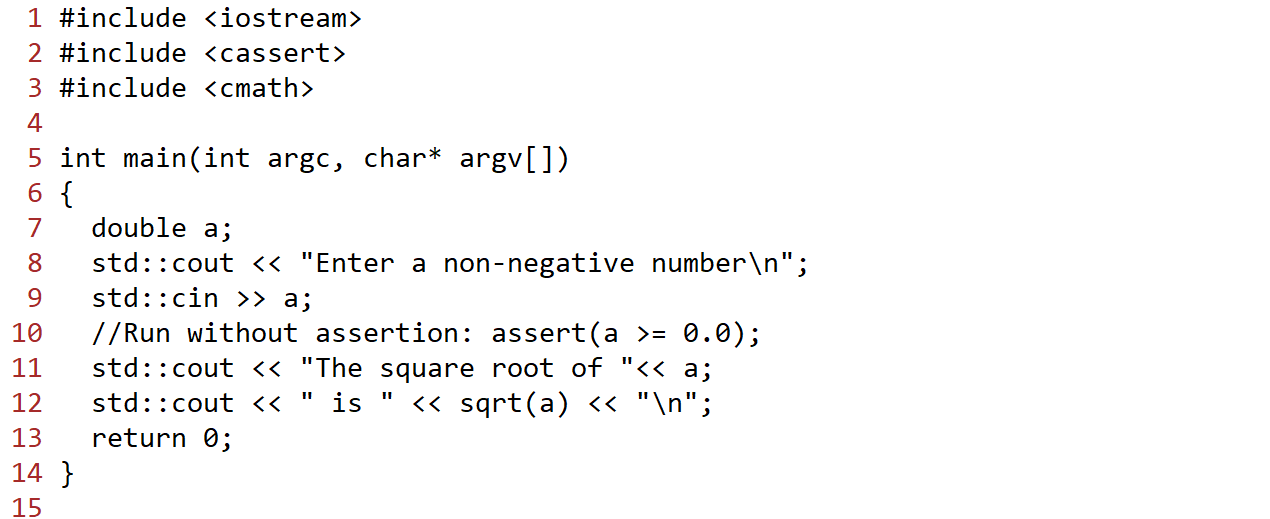
# Errores, Excepciones y pruebas

Utilizaremos el concepto de una sentencia assert. Esta es una forma de forzar a un programa a terminar su ejecución, dado que algo inesperado ha sucedido. A continuación, se presenta un ejemplo de uso de una sentencia assert. En el siguiente programa se calcula la raíz cuadrada de un número introducido en la línea de comandos.



Al ejecutar el programa, si se teclea un número real positivo, el programa muestra la raíz cuadrada del número introducido. Cuando durante la ejecución de este programa se introduce un número negativo, el programa termina su ejecución en la línea 10, indicando que no se cumplió con la condición a >= 0.0.

Ahora se presenta una versión del programa anterior donde la aserción se ha removido convirtiéndola en un comentario.



¿Qué sucede cuando el usuario ignora lo que se le pide e introduce un número negativo en la línea de comandos? Sin la sentencia assert en la línea 10 típicamente el programa se ejecutará completo sin error. Esto es porque la unidad de punto flotante de la computadora produce el resultado de algunos cálculos como

sqrt(-1.0) como “not a number” o nan por brevedad.

Otros ejemplos de operaciones de punto flotante que producen la respuesta nan son 0.0/0.0, y log(0.0). Algunos cálculos tales como 1.0/0.0 se resolverán a una representación de punto flotante de infinito (inf).

## Precondiciones

Cada sección de un programa (donde una sección podría ser una función, un método, un bloque, el cuerpo de una iteración de un ciclo for etc.) se puede interpretar como que tiene la tarea de producir una *postcondición* cuando se cumple una *precondición* válida. La postcondición del programa de arriba es que imprima la raíz cuadrada de un número dado. Esto hace que la precondición sea que el número es no negativo.

# Tres niveles de errores

Algunas de las decisiones más importantes que un programador tiene que tomar son las que tienen que ver con cómo se deben tratar los errores. ¿Qué debe suceder si el usuario no entiende bien lo que se le pide e introduce alguna entrada inválida? ¿Qué debe suceder si el escritor de aplicación accidentalmente permuta los argumentos de entrada de una función de biblioteca? ¿Qué debe suceder si algún esquema numérico ha generado inf o nan debido a alguna divergencia?

La respuesta a todas esas preguntas es la misma: “eso depende”. Es bueno tratar los errores dependiendo de su severidad, en ambos sentidos, en términos de cómo suceden y en términos de qué tan fácilmente podría arreglarse el problema y continuar. El difícil balance de saber qué tan severo podría ser un error se debe a que para tratar con el mismo error diferentes programadores o grupos de programadores podrían elegir formas completamente diferentes. Un grupo de programadores podría decidir que el error se puede arreglar de forma trivial, mientras que otro grupo podría decidir abortar la ejecución del programa.

El framework de tres niveles de errores propuesto en [1] es el siguiente

1 Si el error se puede arreglar en forma segura, entonces arreglarlo. Si es necesario enviar una advertencia al usuario.

2 Si el error podría ser causado por alguna entrada de usuario razonable entonces propagar una excepción (*exception*) hacia el código que hizo la llamada, dado que el código que hizo la llamada tiene suficiente contexto para arreglar el problema.

3 Si el error no debe suceder bajo circunstancias normales entonces poner una aserción (assertion).

Estos tres niveles básicos podrían ser refinados. Se podría distinguir entre errores para los cuales poner unas aserciones (las cuales normalmente son removidas en código optimizado) y los errores que deben detener el programa bajo cualquier circunstancia. En el otro extremo de la escala, se podría distinguir entre formas de arreglar errores las cuales son silenciosas y aquellas que deben dar advertencias al usuario que algo ha sido cambiado.

El nivel de error *exception* es un compromiso entre parchar el problema para continuar y detener la ejecución completamente. Se usa en circunstancias donde al parte del código que llama a una función podría tener suficiente información para ser capaz de tratar con el error.

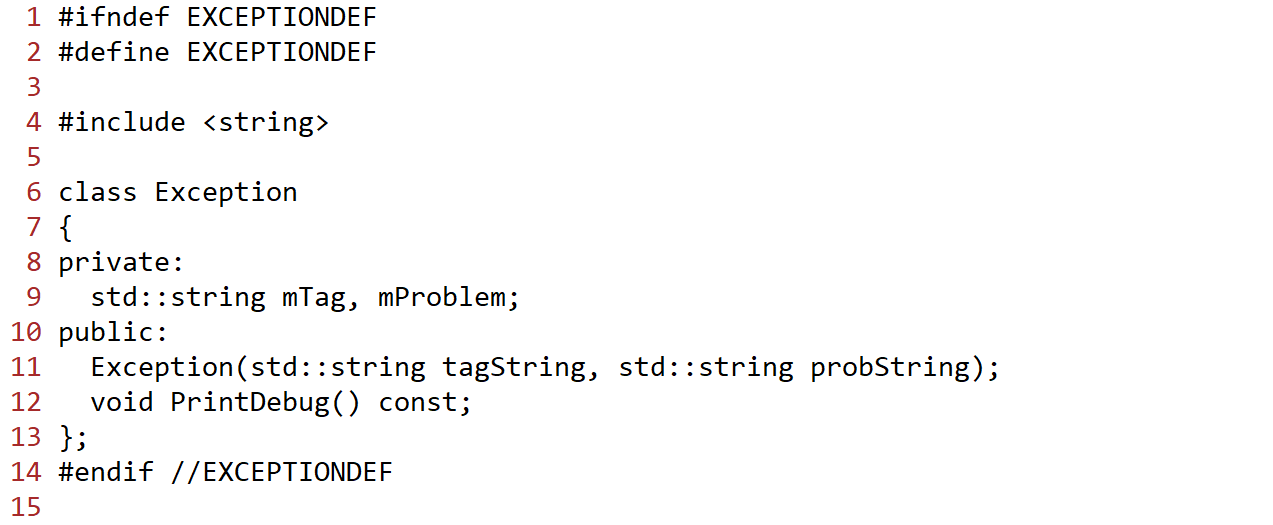
# Las excepciones en C++

Una excepción en C++ es una forma de interrumpir el flujo de control normal de un programa y propagar un atado de información al código que está haciendo la llamada a función. Este atado de información es encapsulado dentro de un objeto. En el ejemplo siguiente (un poco más adelante) se define una clase llamada Exception, pero se podrían propagar objetos de cualquier clase entre funciones para señalar un error.

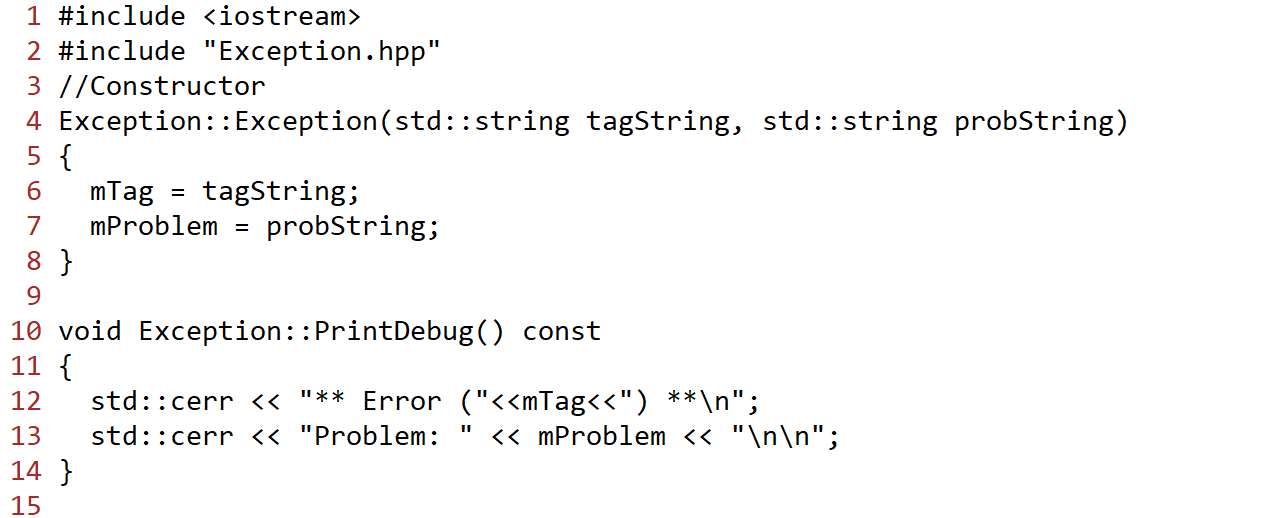
El uso de excepciones requiere las palabras clave try, throw y catch.

* try se usa en el código que hace una llamada y le dice al programa que ejecute algunas sentencias sabiendo que podría suceder algún error.
* throw se usa cuando se identifica un error. La función llamada encapsulará información acerca del error en un objeto Exception y lo propagará de regreso al código que hizó la llamada.
* catch se usa en el código que llama para mostrar como intentar arreglar el error. Cada bloque de código que tiene la palabra clave try debe estar acompañado por un bloque catch.
* Excepciones que no pueden ser atrapadas por el código que hizo la llamada podrían causar que el programa se detenga.

Cuando un error ocurre queremos que el código propague dos piezas de información: un resumen de una palabra del problema y una descripción más larga del error. En la clase Exception (mostrada abajo) se almacenan esas dos piezas de información, y se le da la habilidad de imprimir esta información cuando se requiere.



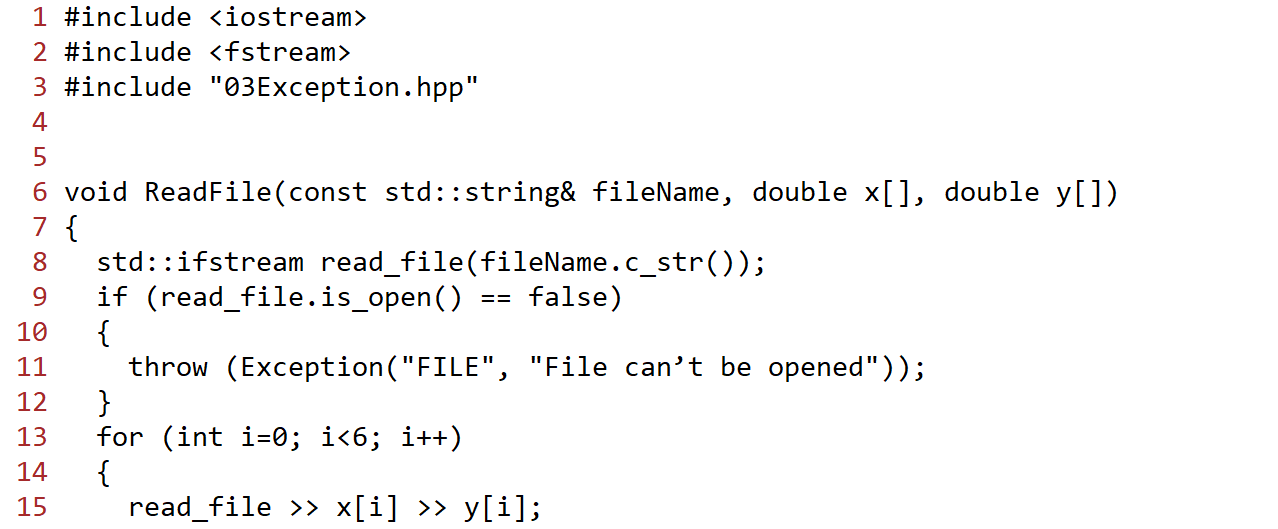
Archivo Exception.hpp

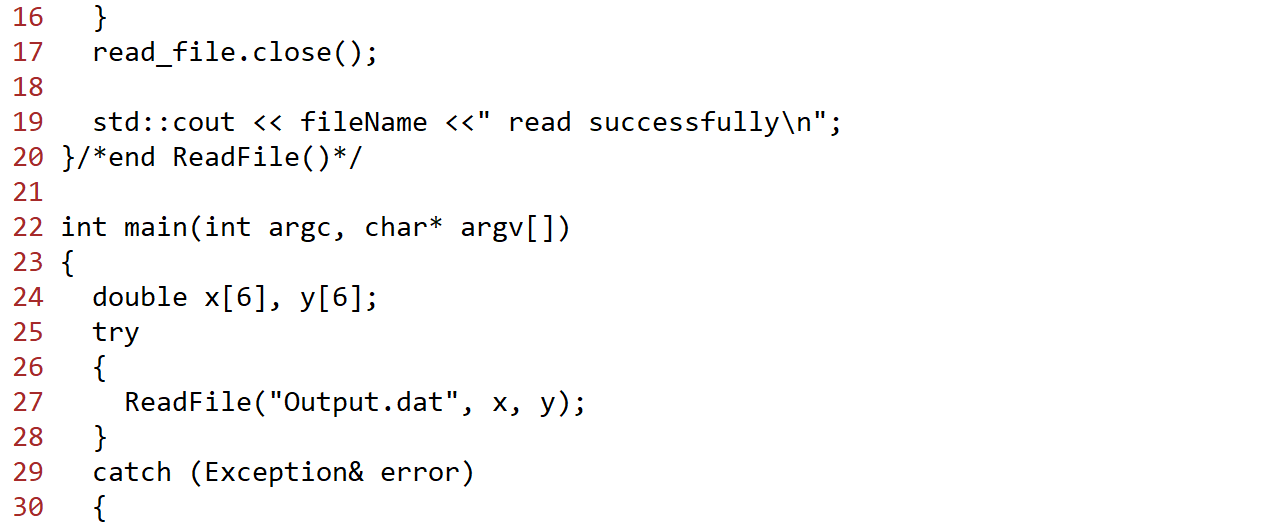


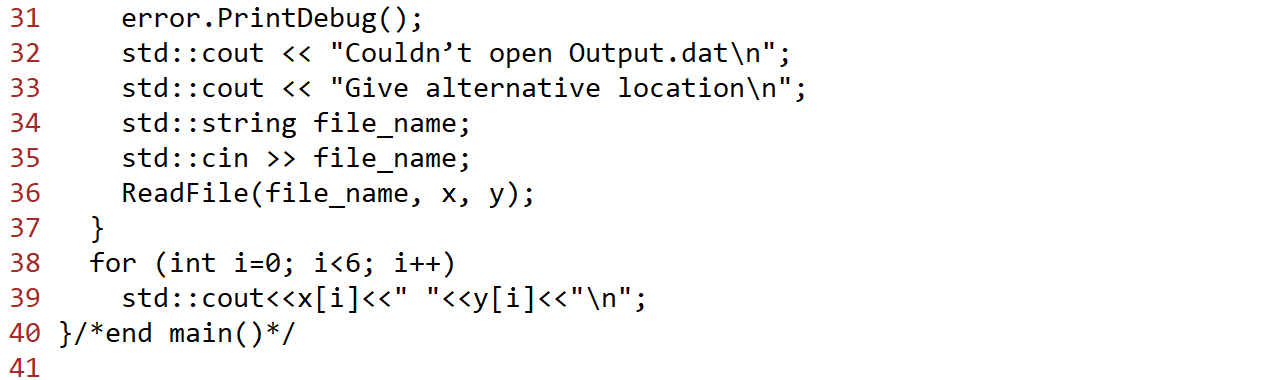
Archivo Exception.cpp

## Uso de excepciones

En el siguiente programa se lee de un archivo llamado Output.dat. Se puede asumir que este archivo existe y usamos una aserción si no existe. En el código siguiente se presenta un programa más sofisticado para abrir un archivo el cual usa excepciones para intentar corregir el problema. Si la función ReadFile no puede abrir el archivo, se propaga una excepción. Esta excepción es atrapada por el código que le pide al usuario un nombre de archivo alternativo. Note que la función ReadFile toma el nombre del archivo como una cadena de C++ la cual es convertida a una cadena de C en la línea 8 (usando el método c\_str).







Archivo TestException.cpp

Testing software

Frecuentemente sucede que se necesita partir de un programa el cual ha sido desarrollado en el pasado y buscar extender su funcionalidad, tal vez para responder a alguna nueva pregunta de investigación. Suponiendo que seamos capaces de entender el funcionamiento del código original porque está bien documentado y tiene un buen estilo de codificación. Supongamos que agregamos la nueva funcionalidad, la usamos para resolver nuestro nuevo problema de investigación, pero más tarde descubrimos que la funcionalidad original del código ha cambiado. Tal vez ya no seamos capaces de reproducir los resultados que son necesarios para una publicación. Esta pifia podría haber sido evitada teniendo una estrategia de “software testing” apropiada se hubiese usado para el código original.

Por las razones mencionadas y otras más, el software siempre debe ser puesto a prueba para que pueda ser usado con confianza. Hay sin embargo, menos concenso sobre cuánto esfuerzo se debe poner en hacer las pruebas, y sobre la metodología que se debe usar para poner el software a prueba. Una razón para la falta de un punto de vista unificado es que el rigor requerido depende de muchas características del software. El esfuerzo que se debe dedicar para poner a prueba el software depende de muchos factores.

## Unit Testing

Una técnica efectiva para poner software a prueba, que es particularmente útil para software que podría ser extendido en el futuro es conocida como pruebas de unidad (*unit testing*). Cuando se usa esta técnica, se escriben una colección de pruebas, conocidas como pruebas de unidad (*unit tests*). Cada prueba de unidad es diseñada para probar una sección de código particular, por ejemplo, un solo método de una clase. Cada prueba debe ser ejecutada cuando se agrega nueva funcionalidad; si una prueba falla entonces es claro que la nueva funcionalidad ha roto una parte existente de la funcionalidad original.

Las pruebas de unidad son particularmente efectivas cuando: (i) cada prueba de unidad cubre solamente un número muy pequeño de líneas de código del código original; y (ii) cada línea del código original es cubierta por al menos una prueba. Siempre que agreguemos una pequeña cantidad de nueva funcionalidad podemos volver a ejecutar cada prueba. Si hemos roto alguna funcionalidad existente, se espera que al menos una prueba falle (si se cumple que cada línea de código está cubierta por al menos una prueba). Además, cada prueba cubre solamente unas cuantas líneas de código, saber cuáles pruebas han fallado debe ayudarnos a señalar donde la funcionalidad original ha sido rota.

Una estrategia ampliamente recomendada para escribir pruebas de unidad es escribir las pruebas para nueva funcionalidad antes de agregar esta nueva funcionalidad. Esta prueba claramente fallará inicialmente. Todas las pruebas ---incluyendo la nueva prueba--- son entonces ejecutadas cuando la nueva funcionalidad ha sido agregada, asegurando que la nueva funcionalidad ha sido correctamente implementada y que el software que ya existía no ha sido roto. Este método de desarrollo de software es conocido como desarrollo dirigido por pruebas (*test driven development*).

Existen varias bibliotecas de frameworks de pruebas para C++, tales como CxxTest, Boost.Test y googletest.

## Extendiendo el software

Es muy raro que un paquete de software sea escrito desde cero. Es más común que un software que ya existe sea extendido. Por ejemplo, se podría esperar que usted extienda la funcionalidad del software escrito por un colega. Alternativamente usted podría desarrollar software que se basa en bibliotecas de fuentes externas. Aun si se cree que el software existente es confiable, el usuario debe al menos poner a prueba su propia implementación de la funcionalidad ofrecida. Esto se puede hacer simplemente probando toda la funcionalidad del software, sin entender la implementación de la funcionalidad ---esto es conocido como prueba de caja negra (*black box testing*). Esto es apropiado para bibliotecas maduras, con buen soporte, que muchas personas consideran que son robustas y confiables. Hay, sin embargo, errores potenciales asociados con black box testing. Suponga que estamos usando algún software escrito externamente que contiene la funcionalidad para resolver un sistema lineal. Cuándo se está usando una prueba de caja negra, simplemente revisaríamos que esta funcionalidad trabaja para un sistema lineal dado. Sin embargo, si hemos tomado un curso de álgebra lineal, sabríamos que no hay solución para algunos sistemas lineales, y que no hay unicidad de solución para otros sistemas lineales. Para limitar que los errores de software escrito externamente se propaguen en el código que desarrollamos, podríamos querer saber cómo el software maneja estos casos; esto dependerá de la implementación de la funcionalidad para esos casos especiales. En esos casos, deliberadamente probaríamos el software escrito externamente escogiendo un ejemplo de sistema lineal sin solución, y un ejemplo de sistema lineal sin solución única. En este caso podríamos también investigar el algoritmo en el que se basa la funcionalidad del sistema, permitiéndonos entender cómo el software dado maneja esos sistemas de ecuaciones. Esta variedad de pruebas es conocida como pruebas de caja blanca (*white box testing*).

Ahora se expliacará cómo pueden hacerse black box testing y white box testing. Se ilustran los conceptos mencionados arriba usando la biblioteca CxxTest, aplicada a una clase usada para representar números complejos. Nos enfocaremos en los principios de las pruebas, para que estos principios puedan ser aplicados para poner a prueba otras bibliotecas. Como tal, no nos enfocamos fuertemente en los detalles de uso de CxxTest; una guía para el uso de esta biblioteca se puede encontrar en <http://www.cxxtest.com>.

Una clase para representar números complejos

En esta sección se desarrolla una clase de números complejos. Se desarrolla esta clase de números complejos para mostrar el desarrollo de la misma. Cabe mencionar que C++ tiene un tipo de dato número complejo el cual está basado en plantillas.

Un número complejo tiene parte real y parte imaginaria. Una clase de números complejos deberá tener atributos que representen esas cantidades. El constructor por defecto debe establecer ambas partes (real e imaginaria) a cero. También sería deseable tener un constructor que permita establecer el número complejo z=x+jy, donde x y y son variables de punto flotante de doble precisión, usando sentencias de la forma

double x = 4.0;

double y = -3.0;

ComplexNumber z(x,y);

Además, podríamos también incluir miembros de la clase que sean métodos que calculen el módulo y el argumento de este número complejo. Otro método podría usarse para elevar el número complejo a una potencia especificada.

Sobrecarga de operadores

Si hemos declarado a, b, c y d como variables enteras entonces podríamos fácilmente relacionar esas variables a través de sentencias como

int a,b,c,d;

a=b;

c=-a;

d=a+b;

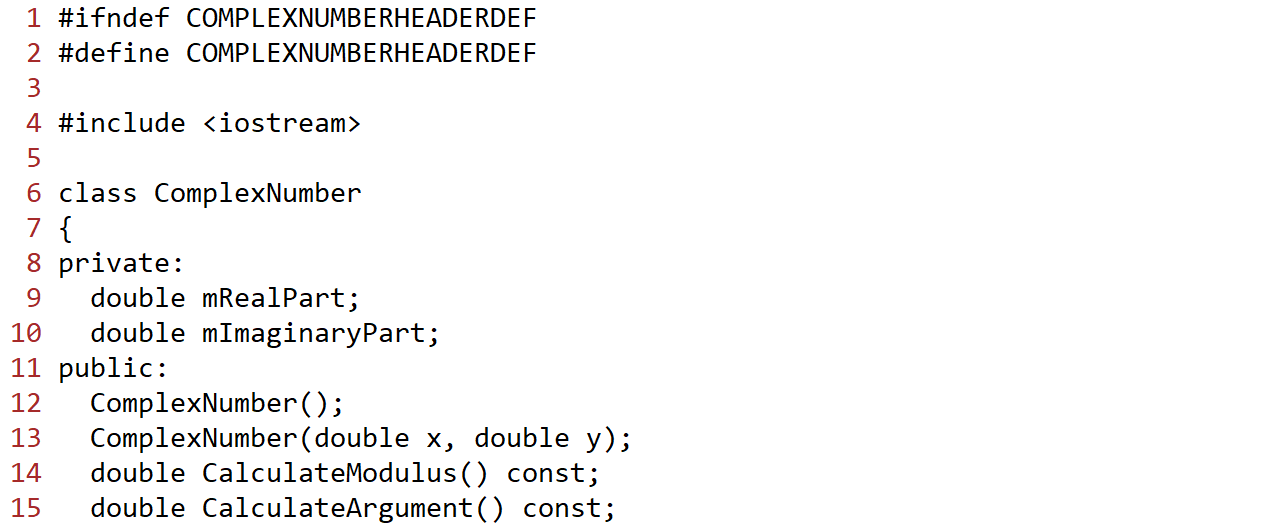
Nos gustaría también escribir sentencias como estas si a, b, c y d fueran números complejos en lugar de enteros. Antes de que podamos hacer esto, necesitamos definir: (i) que significa el operador de asignación para número complejos; (ii) qué significa el operador unario menos ---esto es, qué significa la expresión “-a” si a es un número complejo; y (iii) qué significa el operador de adición binaria ---esto es, qué significa a+b para a y b números complejos. Definir esos operadores para clases se conoce como *sobrecarga de operadores*.

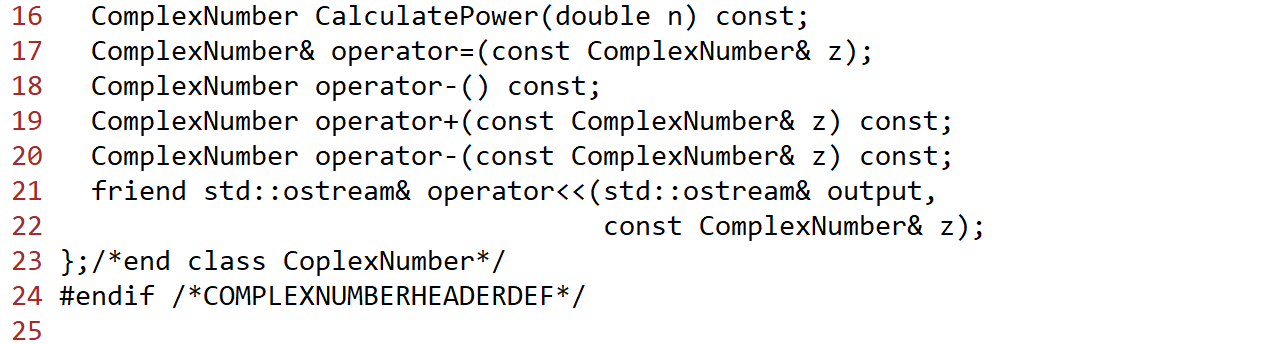
La clase de números complejos

A la luz de la discusión presentada, escribiremos una clase de números complejos con los siguientes miembros.

* Una variable de punto flotante de doble precisión mRealPart que contiene la parte real del número complejo.
* Una variable de punto flotante de doble precisión mImaginaryPart que contiene la parte imaginaria del número complejo.
* Un constructor por defecto sobrecrito ComplexNumber() que inicializa la parte real y la parte imaginaria a cero.
* Un constructor ComplexNumber(double x,double y) que inicializa la parte real a x y la parte imaginaria a y.
* Un método CalculateModulus() que devuelve una variable de punto flotante de doble precisión que contiene el módulo (o magnitud) del número complejo.
* Un método calculateArgument() que devuelve una variable de punto flotante de doble precisión que contiene el argumento (o fase) del número complejo.
* Un método CalculatePower(double n) que devuelve el número complejo calculado cuando se eleva el número complejo original a la potencia n.
* Sobrecarga del operador de asignación.
* Sobrecarga del operador unario de substracción.
* Sobrecarga de los operadores binarios de adición y de substracción.
* Sobrecarga del operador de inserción de flujo de salida (<<) lo cual da control del formato de salida para números complejos.

Un archivo de cabecera adecuado para esta clase se muestra abajo. Este debe ser guardado como ComplexNumber.hpp. Hemos hecho los datos asociados con cada número complejo ---esto es, la parte real y la parte imaginaria--- miembros privados de esta clase para evitar modificaciones inadvertidas de esos miembros. Esos miembros, por supuesto, ser accedidos por los métodos de la clase.





Archivo ComplexNumber.hpp

A continuación se muestra el código del archivo ComplexNumber.cpp

CONTINUAR page 123 Intro to scientific computing in C

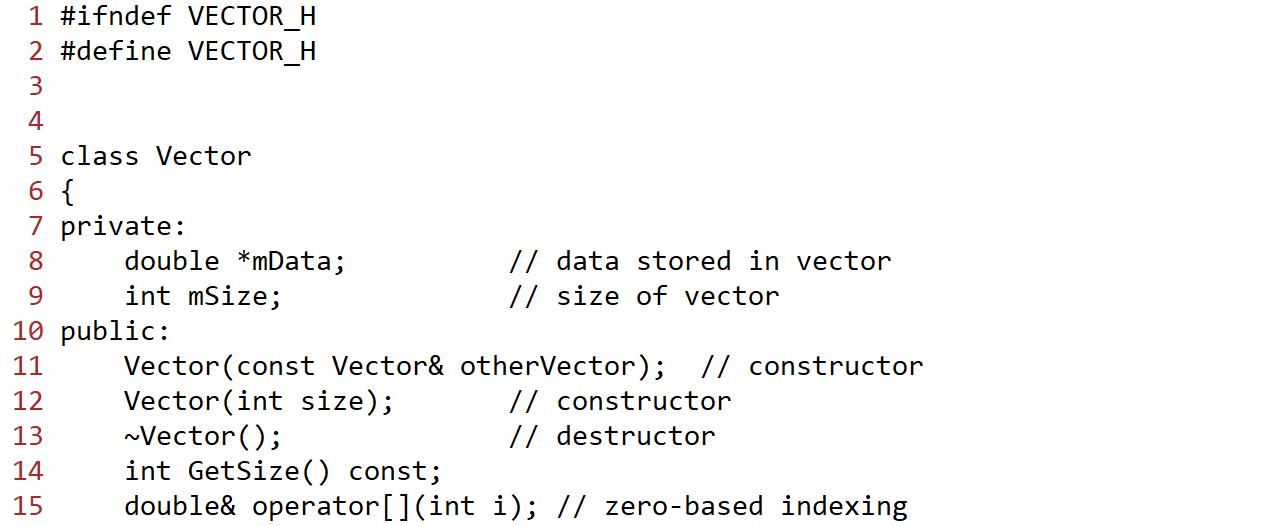
Black Box Testing

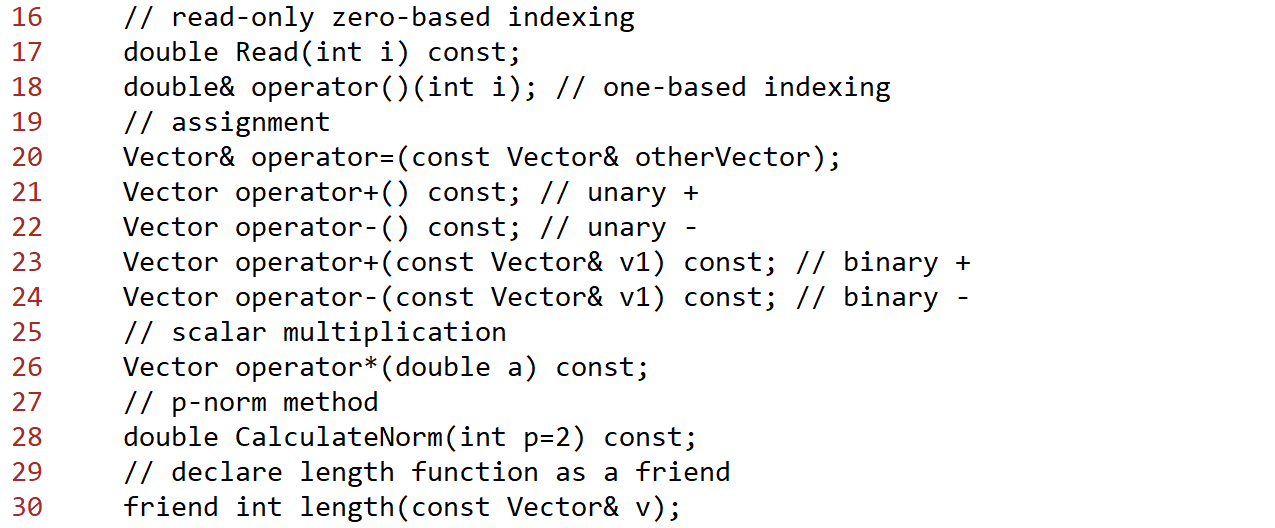
CONTINUAR PAG. 169 open in PDF Studio Viewer.

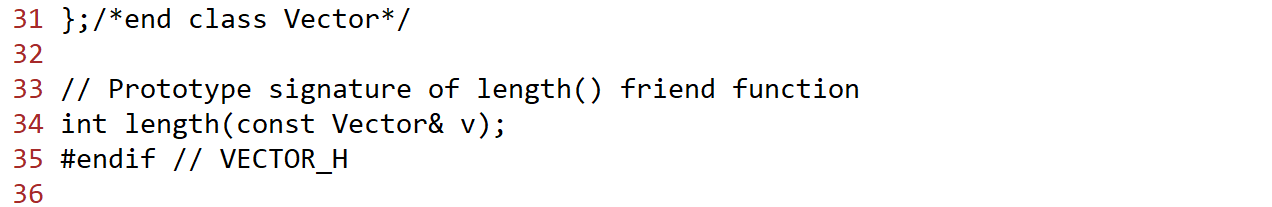
# Requerimientos para la elaboración de clases para hacer operaciones con vectores y matrices

Se buscará desarrollar una clase de vectores llamada Vector, una clase de matrices llamada Matrix y una clase de sistemas lineales llamada LinearSystem. Las clases de vectores y matrices incluirán constructores y destructores que se ocuparán del manejo de memoria. Estas clases sobrecargarán los operadores de asignación, suma, resta y multiplicación, permitiéndonos escribir código como “u=A\*v;” donde u y v son vectores, y A es una matriz: esos operadores sobrecargados incluirán revisiones de que los vectores y las matrices son del tamaño correcto. El operador corchete (o paréntesis cuadrado) será sobrecargado para la clase vector para proporcionar una manera de revisar que el índice del arreglo está dentro del rango correcto, y el operador paréntesis redondo será sobrecargado para permitir que las entradas del vector o matriz sean accedidas usando notación estilo Matlab, indexando desde 1 en lugar de desde cero.

El resto de esta sección se enfoca en el desarrollo de una clase de vectores. El archivo de cabecera para la clase de vectores se incluye a continuación:

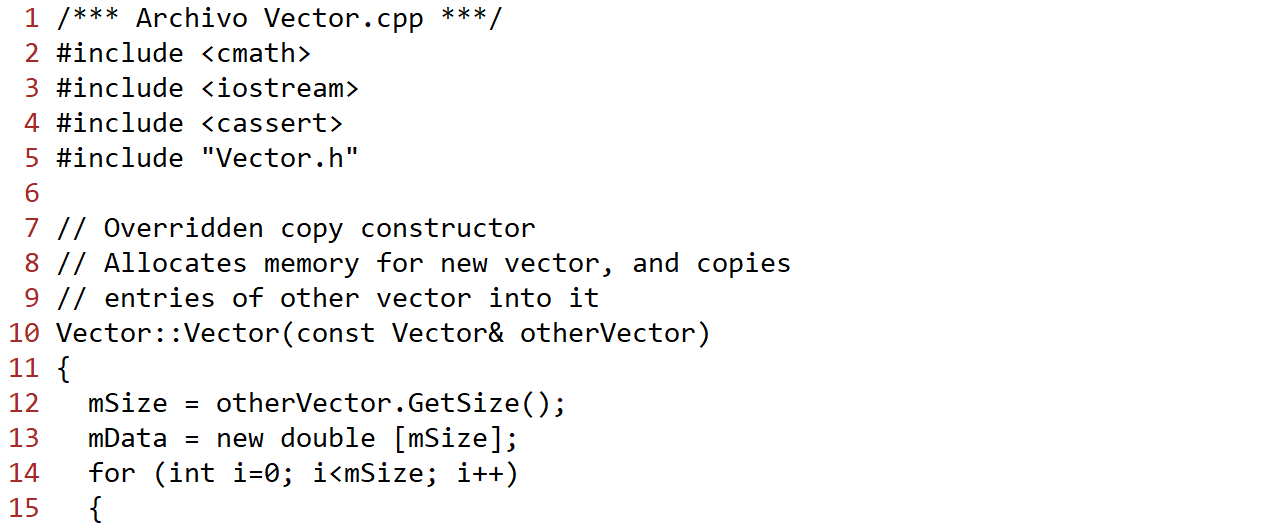


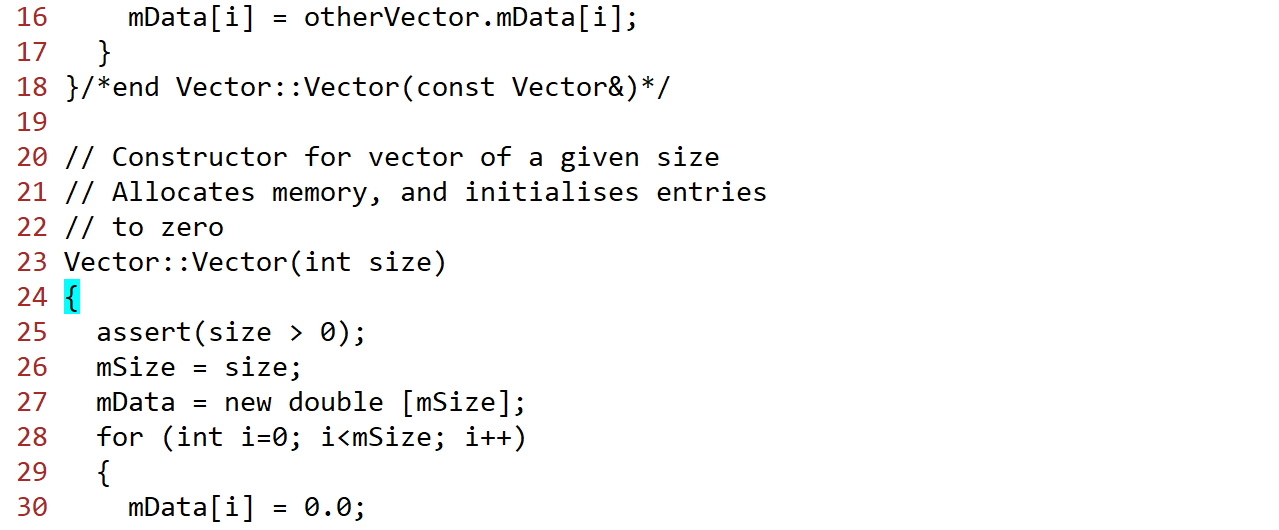


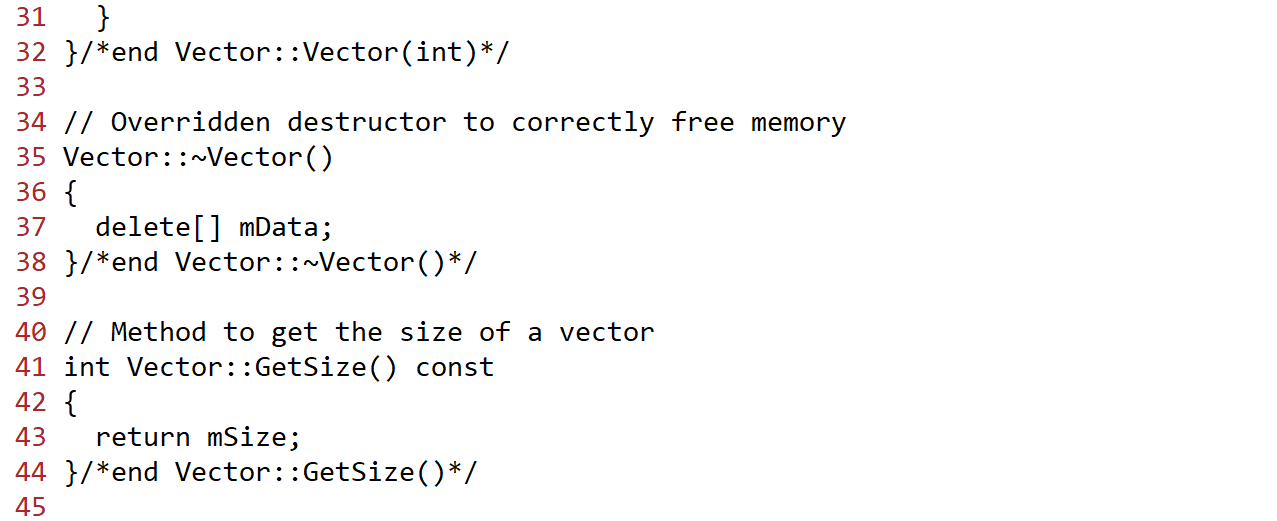


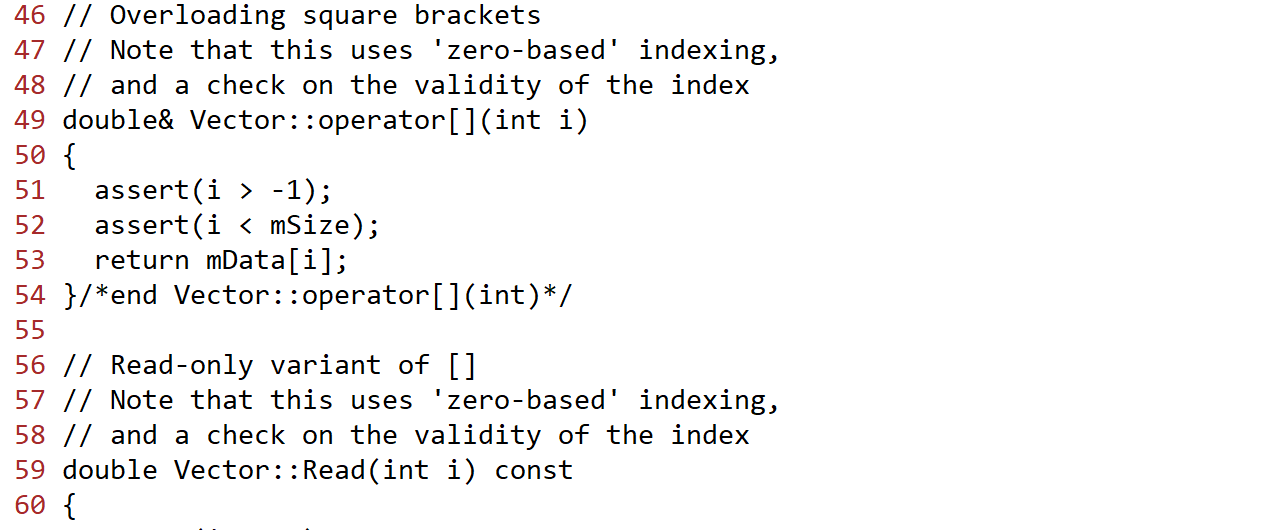
Las dos variables en las que cada instancia de la clase Vector está basadas son un apuntador a una variable de punto flotante de doble precisión, mData, y el tamaño del arreglo, mSize. Se ha declarado a ambas variables como privadas. Claramente necesitamos escribir métodos para acceder a los valores del arreglo y para establecer los valores del arreglo. Se enfatiza que el tamaño del arreglo es establecido a través de un constructor. Como tal, no se permitirá al usuario cambiar esta variable a través de método alguno, pero escribiremos un método público que nos permite acceder al tamaño de un vector dado.

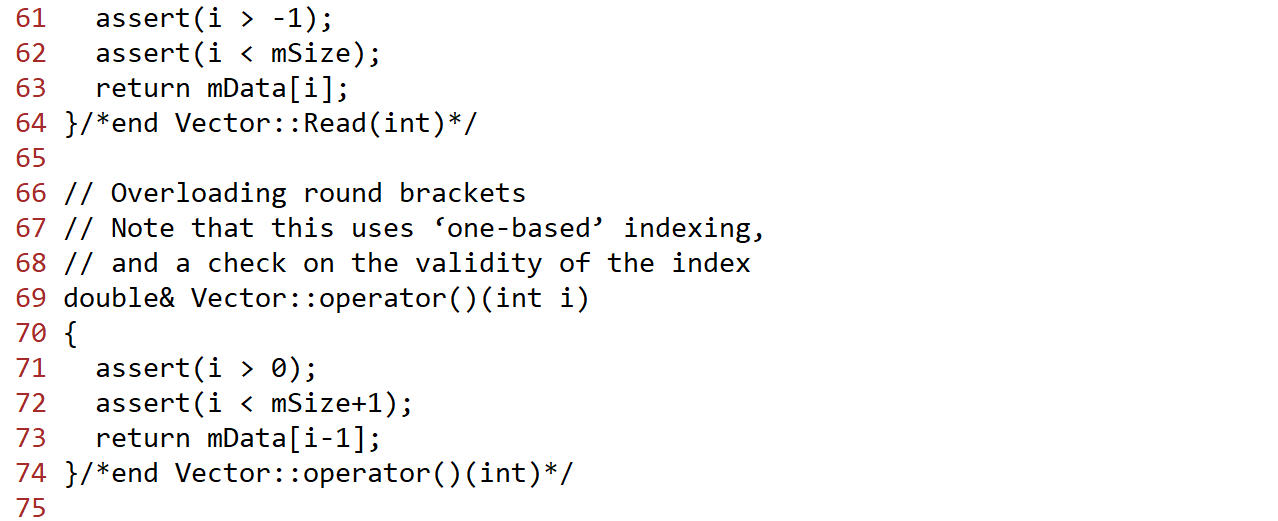
A continuación, se incluye el código del archivo de implementación de la clase Vector.

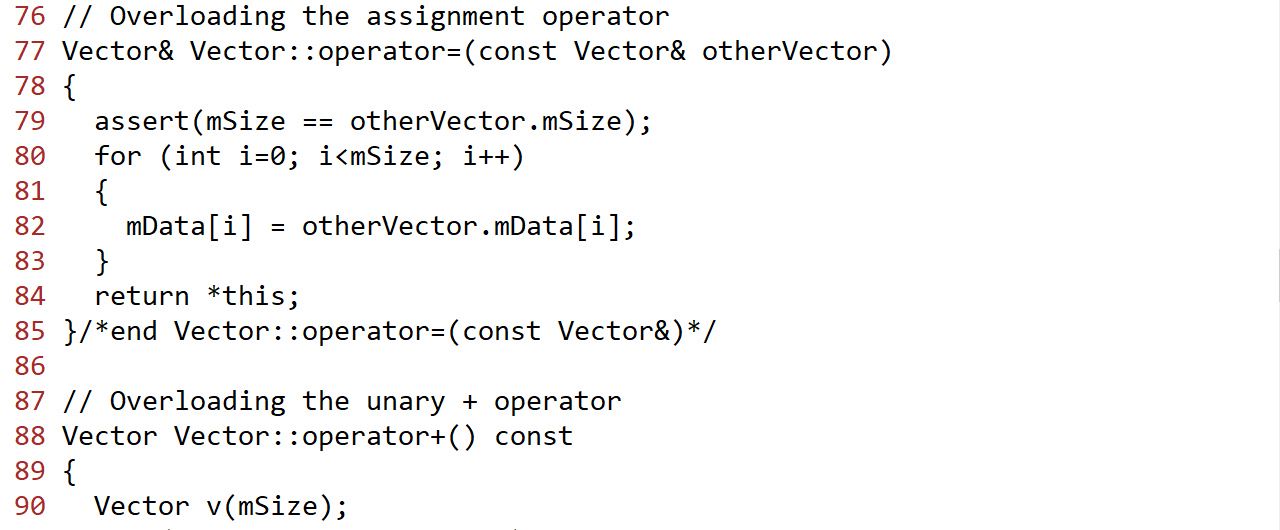


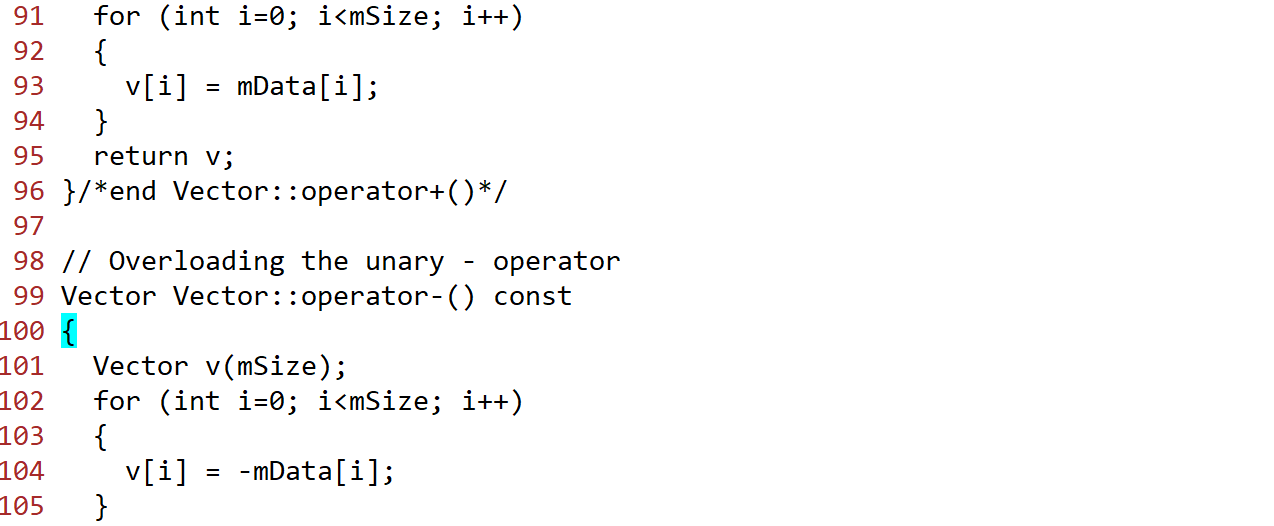


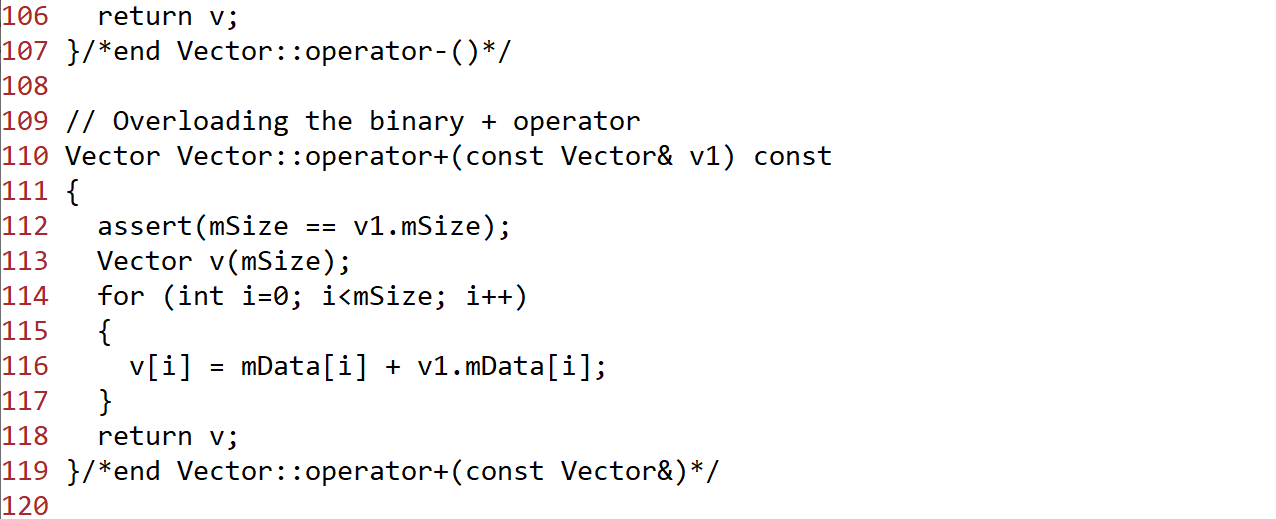


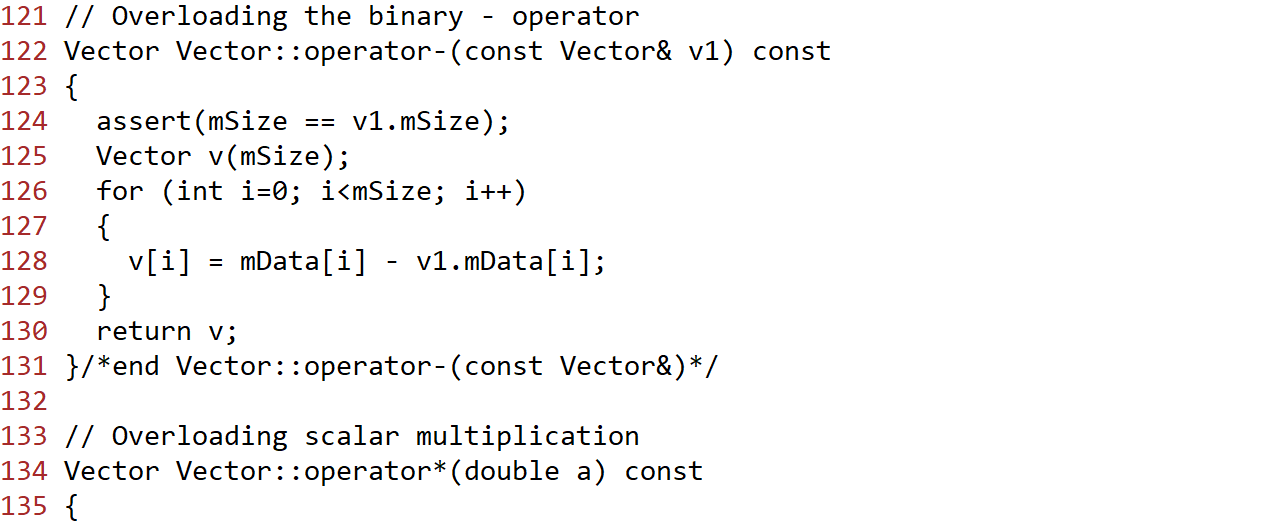


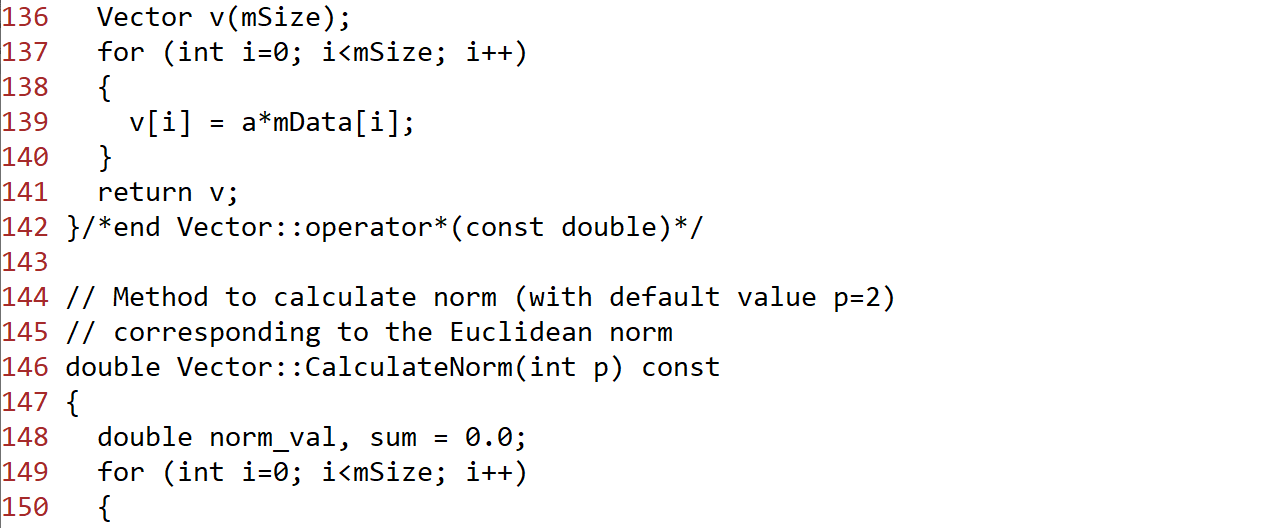


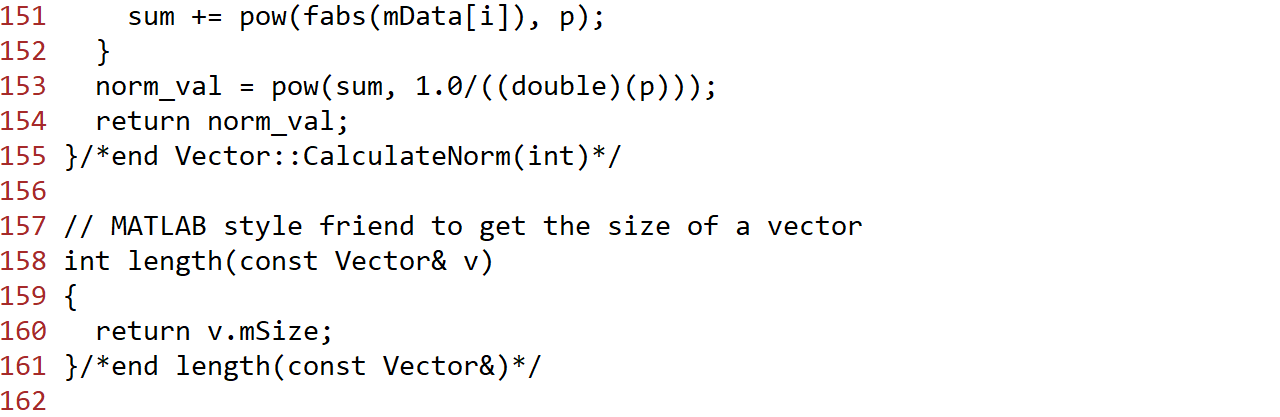












# Constructores y destructores

Por cada utilización de la palabra reservada **new** debe usarse también en algún punto subsecuente del código la palabra reservada **delete**. Escribir constructores y destructores apropiados para las clases Vector y Matrix nos permite automáticamente hacer coincidir las parejas de un uso de delete para cada uso de new (a través del llamado del destructor cuando un objeto ya no se va a usar). Ahora se describen los constructores y destructores adecuados para la clase Vector.

## El constructor por defecto

Supongamos que tenemos una instancia de la clase Vector llamada u. Si utilizáramos el constructor de copia generado automáticamente para crear otro vector llamado v, entonces este constructor no realizaría las tareas que requerimos del **constructor de copia**. EL miembro mSize sería correctamente establecida. Sin embargo, el constructor de copia generado automáticamente no asignaría memoria alguna para la nueva copia de los datos, y así sería imposible que las entradas del vector sean copiadas correctamente. Lo que en realidad sucedería es que el apuntador mData en el vector original u sería asignado al apuntador mData en el nuevo vector v. Como no se asignaría nueva memoria, esto tendría el efecto de que v simplemente se convertiría en un nombre diferente para el vector original u: habría solamente un vector almacenado y cambiar las entradas de v por lo tanto el efecto que no se intenta de cambiar las entradas de u, y viceversa. Una complicación adicional de no sobrescribir el constructor de copia por defecto sería que, debido a que dos vectores tienen el mismo valor en su apuntador mData, ambos vectores intentarían desasignar la misma pieza de memoria (llamando a delete en su destructor) cuando las instancias ya no se usen más.

Lo que realmente queremos que suceda cuando se llama al constructor de copia es que el miembro mSize del nuevo vector v sea establecido al mismo valor que mSize tenía en el vector original u. Se debe asignar memoria de alojamiento para las entradas/elementos del nuevo vector (los double a los que apuntará el apuntador mData), de tal manera que v tenga el mismo número de entradas/elementos que u, y los elementos de u deben ser copiados en la posición correcta en el nuevo vector v. Por lo tanto, sobrescribimos el constructor de copia generado automáticamente de tal manera que nuestra versión del constructor de copia establezca el tamaño de v igual al tamaño de u, obtenga memoria de alojamiento propia del tamaño correcto para el vector v, y copie las entradas/elementos de u a v.

## Un constructor especializado

No se ha proporcionado una definición para el constructor por defecto (el constructor que no recibe argumentos) para asegurar que nunca se use, y hemos sobrescrito el constructor de copia de tal manera que si ya tenemos un vector podríamos crear una copia de ese vector. También incluimos un constructor que requiere un entero positivo como entrada, el cual representa el tamaño del vector. Este constructor establece el atributo mSize a este valor y obtiene espació de almacenamiento (o de alojamiento) para el vector, e inicializa todas las entradas/elementos a cero.

## Destructor

El destructor generado automáticamente ejecutará delete sobre el apuntador mData y el entero mSize (borrará ambas cosas) cuando una instancia de la clase Vector ya no sea accesible (ya no se pueda usar), pero no liberará la memoria de espacio de almacenamiento de la instancia: esto sería similar a no proporcionar un uso de una sentencia delete para un uso de una sentencia new. Por lo tanto, sobrescribimos el destructor generado automáticamente para liberar la memoria de espacio de almacenamiento para una instancia de la clase Vector cuando ya no se pueda acceder a la instancia.

# Accediendo a atributos privados de la clase

Los atributos mSize (el tamaño del vector) y mData (apuntador a las entradas/elementos del vector) están en la sección private de la clase. Esto tiene la ventaja de que solamente podemos establecer el tamaño del vector a través del constructor (asegurando que este atributo es un entero positivo, y previniendo que inadvertidamente lo cambiemos mientras el código se está ejecutando), y nos permite realizar una validación de que el índice de una entrada/elemento de un vector es correcto antes de intentar acceder a esa entrada. A continuación, se describe como se escribieron los métodos para acceder a esos atributos privados.

## Accediendo al tamaño del vector

El tamaño o longitud de un vector es obtenido a través del método público GetSize. Este método no recibe argumentos y devuelve el valor del atributo privado mSize.

## Sobrecargando el operador paréntesis cuadrados (corchetes)

Sobrecargamos el operador corchetes de tal manera que si v es un vector, entonces v[i] devuelve la entrada de v con índice i usando indexado basado en 0. Este método primero revisa que el índice esté dentro del rango correcto ---esto es, un entero no negativo que es menor que mSize--- y entonces devuelve una referencia al valor almacenado en esta entrada del vector.

## Acceso de solo lectura a las entradas del vector

El operador corchetes sobrecargado puede ser usado para ambas cosas, para leer datos del vector y para cambiar las entradas del vector, a través de una referencia. Dado que podríamos necesitar garantizar que algunas funciones las cuales leen de un vector no lo cambien, también proporcionamos una versión const de solo lectura. Este método público Read es similar al operador corchetes. Usa indexación basada en 0 y primero revisa que el índice caiga dentro del intervalo correcto, y entonces devuelve una copia del valor almacenado en esta entrada del vector.

## Sobrecargando el operador paréntesis

El operador paréntesis redondos (paréntesis) es sobrecargado para permitirnos acceder a las entradas de un vector usando indexado basado en uno. Hemos elegido el operador paréntesis para este propósito porque esto permite una notación similar a la que se usa en Fortran y Matlab, los cuales usan indexado basado en uno. De la misma forma que en la sobrecarga del operador corchetes, este método primero valida el índice antes de devolver la entrada apropiada del vector.

## Sobrecarga de operadores para operaciones de vectores

Si usted ha programado en Matlab, conocerá la característica de ese sistema que le permite al usuario escribir sentencias tales como “v=-w;” y “a=b+c;” donde v, w, a, b, c son vectores de un tamaño adecuado. Permitiremos que se pueda escribir código de apariencia similar para los vectores desarrollados en este proyecto de programación a través de la sobrecarga de operadores: esto es, definiremos el operador de asignación, y varios operadores unarios y binarios. Para revisar que los tamaños de los vectores sean los adecuados se utilizarán sentencias assert.

## El operador de asignación

El operador de asignación sobrecargado primero revisa que el vector en el lado izquierdo de la sentencia de asignación sea del mismo tamaño que el vector en el lado derecho. Si esta condición se cumple, las entradas del vector del lado derecho son copiadas en el vector del lado izquierdo.

## Operadores unarios

Los operadores de adición y substracción unarios sobrecargados primero declaran un vector del mismo tamaño que el vector al que el operador unario es aplicado. Las entradas de este nuevo vector son asignadas y entonces se devuelve este nuevo vector. Note que en la sentencia de ejemplo “v=-w;” anterior, es responsabilidad del operador de asignación revisar que los tamaños de v y w coincidan y que la substracción unaria no necesita revisar esto, de hecho, no necesita revisión de error alguno.

## Operadores binarios

Los operadores binarios sobrecargados primero revisan que los dos vectores sobre los que están operando sean del mismo tamaño. Si es así, se crea un nuevo vector del mismo tamaño. Se asignan las entradas de este nuevo vector, y se devuelve este nuevo vector. En la sentencia de ejemplo “a=b+c;” de arriba, es responsabilidad del operador de adición binario revisar que los tamaños de los vectores b y c coincidan, pero es responsabilidad del operador de asignación revisar que el resultado pueda ser asignado al vector a de forma segura.

# Funciones

Una función para calcular la norma p de un vector en nuestra clase de vectores. Esta implementación permite al usuario llamar a la función con un argumento opcional p: si este no se especifica, el valor por defecto p=2 (correspondiente a la norma euclidiana) será utilizado.

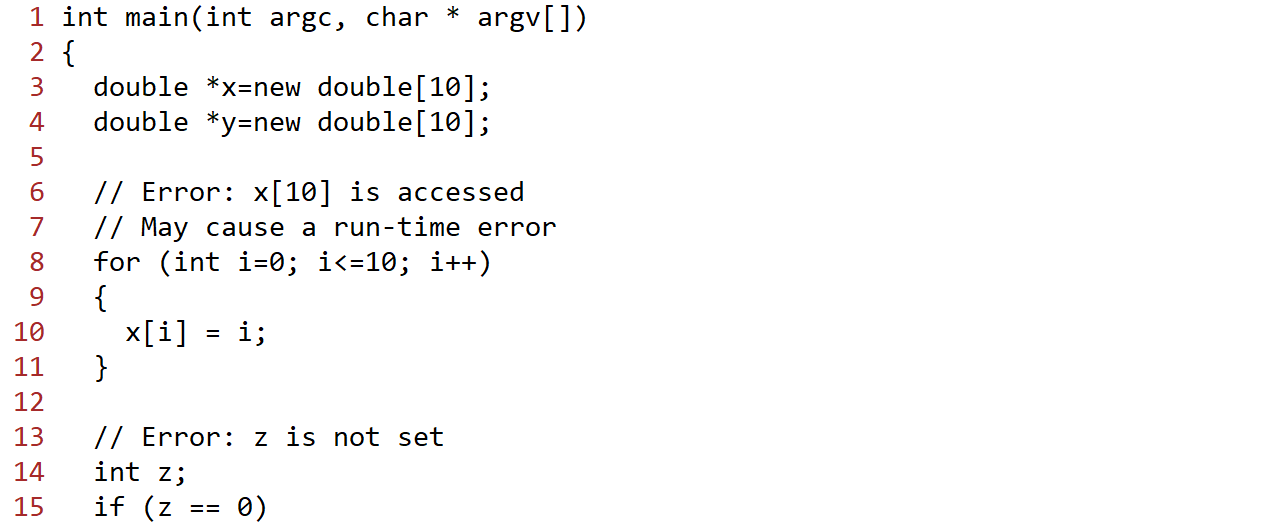
## Funciones método versus funciones amigas

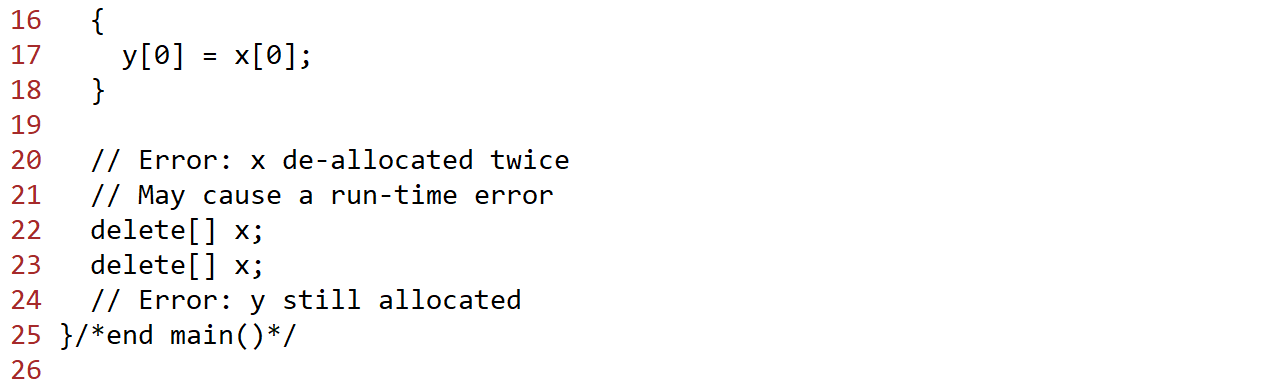
Notamos que la mayoría en la clase está dada a través de métodos y operadores miembros de la clase. Para calcular la norma 2 de un vector o inspeccionar su tamaño, debemos escribir “u.CalculateNorm();” o “u.GetSize();”, respectivamente. Esto podría ser considerado una sintaxis torpe para algunos usuarios, especialmente para aquellos con experiencia en Matlab, así que proporcionamos una función alternativa length para complementar el método GetSize. La función length es declarada como *friend* dentro de la clase lo cual le da la posibilidad de leer el atributo privado mSize. Note que, muchos de los miembros de la clase (miembros atributos y miembros métodos) son declarados const al final de su firma ---para asegurar que no cambian la clase misma--- la función length garantiza que el vector que se le pasa como argumento permanecerá constante haciendo el argumento una variable de tipo referencia constante.

# Herramientas de depuración de memoria

En los párrafos anteriores se enfatizó que cada uso de la palabra reservada new debe ir acompañada con un uso de la palabra reservada delete. Esto es especialmente importante cuando un programa obtiene espacio de alojamiento (o almacenamiento) en la memoria dentro de un ciclo. Si un programa que se ejecuta durante mucho tiempo repetidamente obtiene espacio de alojamiento/almacenamiento sin liberar ese espacio de alojamiento, entonces, eventualmente el programa ocupará innecesariamente toda la memoria disponible de la computadora. Este problema ---conocido como una *memory leak*--- eventualmente causará que el programa falle.

Existen otros problemas relacionados con la memoria además del memory leakage. El siguiente código ilustra algunos errores de uso de la memoria comunes. El ciclo en las líneas 8 – 11 tiene una cota superior incorrecta y entonces el programa intenta escribir a x[10] lo cual no concuerda con los 10 elementos para los que se obtiene espacio de alojamiento para x en la línea 3. La variable z nunca es inicializada, lo cual significa que el flujo del programa en la sentencia if en la línea 15 es impredecible. La segunda sentencia delete ---en la línea 23--- es un error dado que intenta liberar memoria la cual ya ha sido liberada en la línea previa. Finalmente, a la memoria para y, la cual fue alojada en la línea 4 nunca se le aplica el operador delete.





Listado del archivo Broken.cpp

Los cuatro problemas en el programa de arriba no impiden que el código se pueda compilar. El programa podría correr como se espera hasta la sentencia delete final, pero detener su ejecución (“crashear”) en ese punto. Así que, en este programa, la mayoría de los errores de memoria son indetectables en circunstancias normales.

Esos errores pueden ser detectados con una herramienta de depuración de memoria tal como los programas de código abierto *Valgrind* o *Electric Fence*. Estas herramientas corren un archivo ejecutable mientras inspeccionan todas las llamadas de acceso a memoria. Algunas herramientas (tal como Electric Fence) hacen esto remplazando las bibliotecas de memoria usuales con otras que interceptan las llamadas. Otras herramientas (tal como Valgrind) corren el programa dentro de una máquina virtual y externamente monitorean los accesos a memoria ---un proceso más lento, pero un proceso que no requiere recompilación del programa.

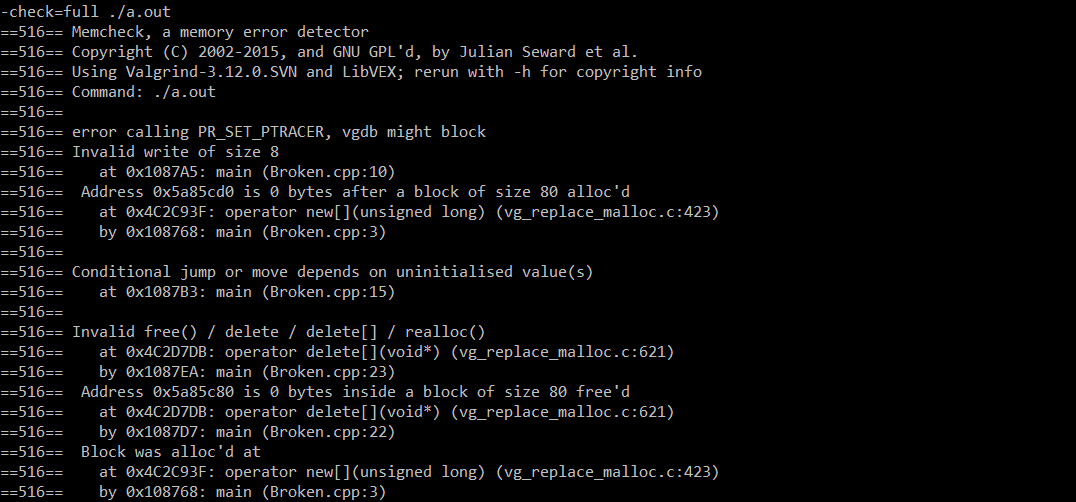
Ejecutando el programa dado en el listado anterior a través de Valgrind se detectan los cuatro problemas de memoria. Primero construimos el programa ejecutable compilando sobre la aplicación Debian con el comando

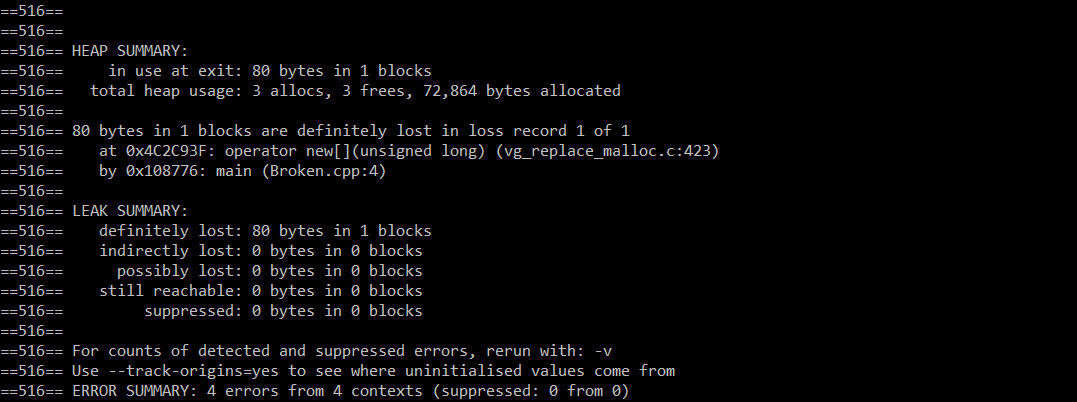
g++ -g Broken.cpp

Después, ejecutamos el comando

valgrind --leak-check=full ./a.out

obtenemos las siguientes líneas de salida:





Resultado de la ejecución de valgrind con el ejecutable de Broken.cpp

# Ejercicios

Los ejercicios aquí planteados le guiarán para construir sobre la clase Vector con una clase adicional Matrix. Estas clases son entonces combinadas en una clase LinearSystem (o, en el ejercicio final, una clase alternativa derivada de ésta) la cual tiene un método para resolver sistemas de la forma **Ax=b** para **x**. A través de ilustraciones se mostrarán soluciones típicas a los ejercicios con un diagrama de colaboración para todas las clases producidas por esos ejercicios. Este diagrama usa sintaxis UML.

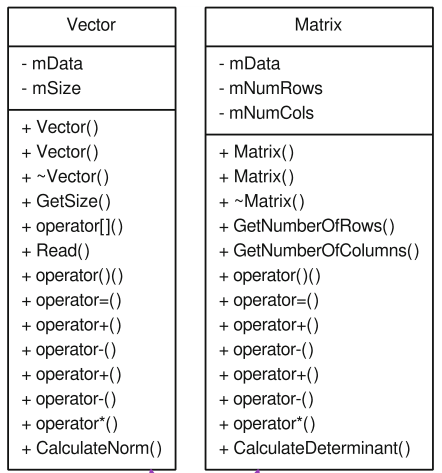
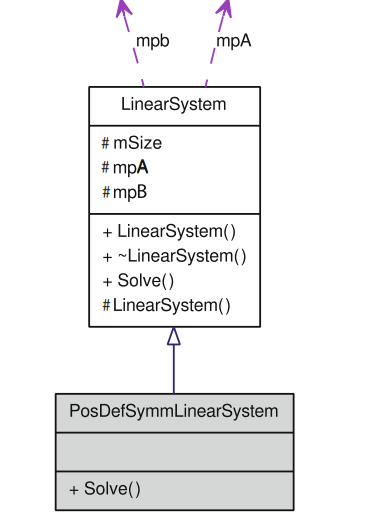


Diagrama de colaboración de clases para PosDefSymLinearSystem

### Ejercicio 1

Escriba una suite de pruebas para hacer una prueba de caja negra de la clase de vectores.

CONTINUAR PAGINA 192