Instituto Tecnológico y de Estudios Superiores de Monterrey Campus Estado de México
Escuela de Ingeniería y Ciencias
Departamento de Computación

Proyecto 02. Base matemática. Gráficas computacionales.

Apegándome al Código de Ética de los Estudiantes del Tecnológico de Monterrey, me comprometo a que mi actuación en este examen esté regida por la honestidad académica. Acepto: _____

Profesor: Oriam Renan De Gyves López	Clave y grupo: TC3022. ____
Alumno: _____	Matrícula: _____

Lee cuidadosamente todas las instrucciones antes de comenzar a resolver cada uno de los problemas.

Fecha de entrega: 17 de febrero, antes de la2 23:59 horas.

NOTA MUY IMPORTANTE: Este proyecto debe ser elaborado de manera individual. Puedes trabajar de manera colaborativa para resolver dudas conceptuales, pero el código debe ser realizado únicamente por ti. Recuerda citar correctamente todas las fuentes de inspiración que ocupaste para la realización del mismo. Cualquier indicio de copia, trampa o fraude será penalizado con 1 de calificación y será reportado al comité de integridad académica. Se penalizará tanto al copiado como al copiador.

Documentación:
Manual de referencia de C++: http://www.cplusplus.com/reference/

Deberán leer:
Clases amigas y herencia: http://www.cplusplus.com/doc/tutorial/inheritance/
Funciones inline: http://www.cplusplus.com/articles/2LywvCM9/
Sobrecarga de operadores: https://en.cppreference.com/w/cpp/language/operators
Reinterpret cast: https://en.cppreference.com/w/cpp/language/reinterpret_cast
Capítulo 4 de OpenGL SuperBible (7ma edición): Math for 3D Graphics.

Evaluación
El proyecto será evaluado utilizando los siguientes criterios:

100	El proyecto cumple con todos los requerimientos. Pasa todas las pruebas de unidad.
1	El programa fuente contiene errores sintácticos. No compila.
1 - 99	El programa produce errores al momento de correrlo. No pasa todas las pruebas de unidad.
Falta a la integridad académica	La solución es un plagio.

I. Dentro del proyecto **`Math`**, implementar la clase **`vec2`** con las especificaciones que se muestran a continuación. La clase **`vec2`** debe estar contenida en un *`namespace`* llamado **`cgmath`**. El proyecto **`Math Test`** contiene las pruebas unitarias (**`vec2_test`**) para verificar tu resultado.

vec2
+ x:float = 0.0f + y:float = 0.0f
+ vec2() + vec2(x:float, y:float) + operator[](i:int):float& + *<<{const}>>* operator[](i:int):const float& + operator*=(s:float):vec2& + operator/=(s:float):vec2& + operator+=(v:const vec2&):vec2& + operator-=(v:const vec2&):vec2& + *<<{const}>>* operator==(v:const vec2&):bool + *<<{const}>>* magnitude():float + normalize():void + magnitude(v:const vec2&):float + normalize(v:const vec2&):vec2 + dot(a:const vec2&, b:const vec2&):float + *<<{friend}>>* operator<<(os:ostream&, v:const vec2&):ostream& + *<<{inline}>>* operator*(v:const vec2&, s:float):vec2 + *<<{inline}>>* operator*(s:float, v:const vec2&):vec2 + *<<{inline}>>* operator/(v:const vec2&, s:float):vec2 + *<<{inline}>>* operator+(a:const vec2&, b:const vec2&):vec2 + *<<{inline}>>* operator-(a:const vec2&, b:const vec2&):vec2 + <<{inline}>> operator-(v:const vec2&):vec2

1. `+ vec2()`
   Constructor que inicializa las variables de instancia a sus valores por default.
   Ejemplo:
   ```
   vec2 a; // Representa el vector (0.0f, 0.0f)
   ```

2. `+ vec2(x:float, y:float)`
   Constructor que inicializa las variables de instancia utilizando los parámetros del mismo.
   Ejemplo:
   ```
   vec2 a(1.0f, 2.0f); // Representa el vector (1.0f, 2.0f)
   ```

3. `+ operator[](i:int):float&`
   El operador [ ] permite tener acceso de escritura a las variables de instancia utilizando un índice. El método no hace ningún chequeo del índice utilizado.
   Ejemplo:
   ```
   vec2 a(1.0f, 2.0f);
   a[1] = 4.0f; // El vec2 a es ahora (1.0f, 4.0f)
   ```

4. `+ <<{const}>> operator[](i:int):const float&`
   El operador [ ] permite tener acceso de sólo lectura a las variables de instancia utilizando un índice. El método no hace ningún chequeo del índice utilizado.
   Ejemplo:
   ```
   vec2 a(1.0f, 2.0f);
   std::cout << a[0] << std::endl; // Imprime 1.0f
   ```

5. `+ operator*=(s:float):vec2&`
   El operador *= multiplica cada una de las variables de instancia por el valor escalar `s`. Se debe regresar la referencia de la instancia modificada de `vec2`.
   Ejemplo:
   ```
   vec2 a(1.0f, 2.0f);
   a *= 3.0f; // El vec2 es ahora (3.0f, 6.0f)
   ```

6. `+ operator/=(s:float):vec2&`
   El operador /= divide cada una de las variables de instancia por el valor escalar `s`. Se debe regresar la referencia de la instancia modificada de `vec2`.
   Ejemplo:
   ```
   vec2 a(2.0f, 4.0f);
   a /= 2.0f; // El vec2 a es ahora (1.0f, 2.0f)
   ```

7. `+ operator+=(v:const vec2&):vec2&`
   El operador += suma cada una de las variables de instancia con la componente correspondiente de `v`. Se debe regresar la referencia de la instancia modificada de `vec2`.
   Ejemplo:
   ```
   vec2 a(1.0f, 2.0f);
   vec2 b(4.0f, 5.0f);
   a += b; // El vec2 a es ahora (5.0f, 7.0f). El vec2 b no ha tenido cambios.
   ```

8. `+ operator-=(v:const vec2&):vec2&`
   El operador -= resta cada una de las variables de instancia con la componente correspondiente de `v`. Se debe regresar la referencia de la instancia modificada de vec2.
   Ejemplo:
   ```
   vec2 a(1.0f, 2.0f);
   vec2 b(4.0f, 5.0f);
   b -= a; // El vec2 b es ahora (3.0f, 3.0f). El vec2 a no ha tenido cambios.
   ```

9. + *<<{const}>>* operator==(v:const vec2&):bool
   El operador == realiza una comparación de la instancia actual de vec2 con el vector v. La comparación se realiza a nivel de componente, es decir, las componentes (x, y) de ambos vectores deben ser idénticas. Se debe regresar true si todas las componentes son iguales en ambos vectores, false de lo contrario.
   Ejemplo:
   ```
   vec2 a(1.0f, 2.0f);
   vec2 b(1.0f, 2.0f);
   std::cout << a == b << std::endl; // El resultado es true.
   ```

10. + *<<{const}>>* magnitude():float
    El método magnitude() calcula y regresa la magnitud de la instancia sobre la cual se manda a llamar.
    Ejemplo:
    ```
    vec2 a(3.0f, 4.0f);
    std::cout << a.magnitude() << std::endl; // Imprime 5.0f
    ```

11. + normalize():void
    El método normalize() normaliza el vector de la instancia sobre la cual se manda a llamar.
    Ejemplo:
    ```
    vec2 a(3.0f, 4.0f);
    a.normalize(); // El vec2 a es ahora (0.6f, 0.8f).
    ```

12. + magnitude(v:const vec2&):float
    El método estático magnitude(…) calcula y regresa la magnitud del vector v.
    Ejemplo:
    ```
    vec2 a(3.0f, 4.0f);
    std::cout << vec2::magnitude(a) << std::endl; // Imprime 5.0f
    ```

13. + normalize(v:const vec2&):vec2
    El método estático normalize(…) calcula el vector normal de v y lo regresa como un nuevo objeto vec2. El método no modifica el vec2 v.
    Ejemplo:
    ```
    vec2 a(3.0f, 4.0f);
    vec2::normalize(a); // Regresa un vec2 (0.6f, 0.8f). El vec2 a no se modifica.
    ```

14. + dot(a:const vec2&, b:const vec2&):float
    El método estático dot(…) calcula y regresa el producto punto entre los vectores a y b.
    Ejemplo:
    ```
    vec2 a(1.0f, 2.0f);
    vec2 b(3.0f, 4.0f);
    std::cout << vec2::dot(a, b) << std::endl; // Imprime 11.0f
    ```

15. + *<<{friend}>>* operator<<(os:ostream&, v:const vec2&):ostream&
    El operador << crea un string con una representación en texto de la instancia sobre la cual se manda a llamar. La representación correcta debe ser los componentes del vector entre paréntesis, separados por comas. Permite asociar un *output stream* (cout por ejemplo) con un vec2.
    Ejemplo:
    ```
    vec2 a(1.0f, 2.0f);
    std::cout << a << std::endl; // Imprime el vector utilizando cout. Imprime (1.0, 2.0)
    ```

16. + *<<{inline}>>* operator*(v:const vec2&, s:float):vec2
    El operador * permite realizar una multiplicación entre un vector (operando izquierdo) y un escalar (operando derecho). Cada una de las componentes de v deberán ser multiplicadas por s y el resultado almacenado en un nuevo objeto vec2 que deberá regresarse como resultado de ejecutar este método.
    Ejemplo:
    ```
    vec2 a(1.0f, 2.0f);
    a = a * 3.0f; // El vec2 a es ahora (3.0, 6.0)
    ```

17. + *<<{inline}>>* `operator*(s:float, v:const vec3&):vec3`
    Similar al ejercicio 16. Este operador habilita la propiedad conmutativa de la multiplicación entre un vector y un escalar.
    Ejemplo:
    ```
    Vec2 a(1.0f, 2.0f);
    a = 3.0f * a; // El vec2 a es ahora (3.0, 6.0)
    ```

18. + *<<{inline}>>* `operator/(v:const vec2&, s:float):vec2`
    El operador / permite realizar una división entre un vector (operando izquierdo) y un escalar (operando derecho). Cada una de las componentes de `v` deberán ser divididas por `s` y el resultado almacenado en un nuevo objeto `vec2` que deberá regresarse como resultado de ejecutar este método.
    Ejemplo:
    ```
    vec2 a(2.0f, 4.0f);
    a = a / 2.0f; // El vec2 a es ahora (1.0, 2.0)
    ```

19. + *<<{inline}>>* `operator+(a:const vec2&, b:const vec2&):vec2`
    El operador + permite realizar una suma entre dos vectores. Cada componente del vector `a` deberá ser sumado con la componente correspondiente en el vector `b` y el resultado almacenado en un nuevo objeto `vec2` que deberá regresarse como resultado de ejecutar este método.
    Ejemplo:
    ```
    vec2 a(1.0f, 2.0f);
    vec2 b(3.0f, 4.0f);
    std::cout << a + b << std::endl; // Imprime (4.0, 6.0)
    ```

20. + *<<{inline}>>* `operator-(a:const vec2&, b:const vec2&):vec2`
    El operador − permite realizar una entre dos vectores. Cada componente del vector `b` deberá ser restado de la componente correspondiente en el vector `a` y el resultado deberá almacenado en un nuevo objeto `vec2` que deberá regresarse como resultado de ejecutar este método.
    Ejemplo:
    ```
    vec2 a(1.0f, 2.0f);
    vec2 b(3.0f, 4.0f);
    std::cout << a – b << std::endl; // Imprime (-2.0, -2.0)
    ```

21. + *<<{inline}>>* `operator-(v:const vec2&):vec2`
    El operador − permite negar las componentes del vector `v`. El resultado deberá ser almacenado en un nuevo objeto `vec2`, que deberá regresarse como resultado de ejecutar este método.
    Ejemplo:
    ```
    vec2 a(1.0f, 2.0f);
    std::cout << -a << std::endl; // Imprime (-1.0, -2.0)
    std::cout << a << std::endl; // Imprime (1.0, 2.0)
    ```

II. Dentro del proyecto **Math**, implementar la clase **vec3** con las especificaciones que se muestran a continuación. La clase **vec3** debe estar contenida en un *namespace* llamado **cgmath**. El proyecto **Math Test** contiene las pruebas unitarias (**vec3_test**) para verificar tu resultado. Los métodos de la clase **vec3** son equivalentes a los de la clase **vec2** pero utilizando 3 componentes en lugar de 2.

vec3
+ x:float = 0.0f + y:float = 0.0f + z:float = 0.0f
+ vec3() + vec3(x:float, y:float, z:float) + operator[](i:int):float& + *<<{const}>>* operator[](i:int):const float& + operator*=(s:float):vec3& + operator/=(s:float):vec3& + operator+=(v:const vec3&):vec3& + operator-=(v:const vec3&):vec3& + *<<{const}>>* operator==(v:const vec3&):bool + *<<{const}>>* magnitude():float + normalize():void + magnitude(v:const vec3&):float + normalize(v:const vec3&):vec3 + dot(a:const vec3&, b:const vec3&):float + cross(a:const vec3&, b:const vec3&):vec3 + *<<{friend}>>* operator<<(os:ostream&, v:const vec3&):ostream& + *<<{inline}>>* operator*(v:const vec3&, s:float)vec3 + *<<{inline}>>* operator*(s:float, v:const vec3&):vec3 + *<<{inline}>>* operator/(v:const vec3&, s:float):vec3 + *<<{inline}>>* operator+(a:const vec3&, b:const vec3&):vec3 + *<<{inline}>>* operator-(a:const vec3&, b:const vec3&):vec3 + <<{inline}>> operator-(v:const vec3&):vec3

1. + cross(a:const vec3&, b:const vec3&):float
   El método estático `cross(…)` calcula y regresa el producto cruz entre los vectores a y b.
   Ejemplo:

```
vec3 a(1.0f, 0.0f, 0.0f);
vec3 b(0.0f, -1.0f, 0.0f);
vec3 c = vec3::cross(a, b); // El vector c es (0.0f, 0.0f, -1.0f)
```

III. Dentro del proyecto **Math**, implementar la clase **vec4** con las especificaciones que se muestran a continuación. La clase **vec4** debe estar contenida en un *namespace* llamado **cgmath**. El proyecto **Math Test** contiene las pruebas unitarias (**vec4_test**) para verificar tu resultado. Los métodos de la clase **vec4** son equivalentes a los de la clase **vec2** pero utilizando 4 componentes en lugar de 2.

vec4
+ x:float = 0.0f + y:float = 0.0f + z:float = 0.0f + w:float = 0.0f
+ vec4() + vec4(x:float, y:float, z:float, w:float) + operator[](i:int):float& + *<<{const}>>* operator[](i:int):const float& + operator*=(s:float):vec4& + operator/=(s:float):vec4& + operator+=(v:const vec4&):vec4& + operator-=(v:const vec4&):vec4& + *<<{const}>>* operator==(v:const vec4&):bool + *<<{friend}>>* operator<<(os:ostream&, v:const vec4&):ostream& + *<<{inline}>>* operator*(v:const vec4&, s:float):vec4 + *<<{inline}>>* operator*(s:float, v:const vec4&):vec4 + *<<{inline}>>* operator/(v:const vec4&, s:float):vec4 + *<<{inline}>>* operator+(a:const vec4&, b:const vec4&):vec4 + *<<{inline}>>* operator-(a:const vec4&, b:const vec4&):vec4 + <<{inline}>> operator-(v:const vec4&):vec4

IV. Dentro del proyecto **Math**, implementar la clase **mat3** con las especificaciones que se muestran a continuación. La clase **mat3** debe estar contenida en un *namespace* llamado **cgmath**. El proyecto **Math Test** contiene las pruebas unitarias (**mat3_test**) para verificar tu resultado. La matriz debe estar codificada en formato *column-major*.

mat3
- n:float[3][3] = { 0 }
+ mat3() + mat3(diagonal:float) + mat3(a:const vec3&, b:const vec3&, c:const vec3&) + operator[](column:int):vec3& + *<<{const}>>* operator[](column:int):const vec3& + *<<{const}>>* operator==(m:const mat3&):bool + determinant(m:const mat3&):float + inverse(m:const mat3&):mat3 + transpose(m:const mat3&):mat3 + *<<{friend}>>* operator<<(os:ostream&, m:const mat3&):ostream& + *<<{inline}>>* operator*(m:const mat3&, v:const vec3&):vec3 + *<<{inline}>>* operator*(m1:const mat3&, m2:const mat3&):mat3

1. + mat3()
   Constructor que inicializa las variables de instancia a sus valores por default. El valor por default de n contiene todos sus valores inicializados en 0.
   Ejemplo:
   mat3 m; // Representa la matriz $\begin{bmatrix}0 & 0 & 0\\0 & 0 & 0\\0 & 0 & 0\end{bmatrix}$

2. + mat3(diagonal:float)
   Constructor que inicializa las variables de instancia utilizando el parámetro diagonal para dar valor a la diagonal de la matriz.
   Ejemplo:
   mat3 m1(1.0f); // Representa la matriz $\begin{bmatrix}1 & 0 & 0\\0 & 1 & 0\\0 & 0 & 1\end{bmatrix}$

   mat3 m2(3.0f); // Representa la matriz $\begin{bmatrix}3 & 0 & 0\\0 & 3 & 0\\0 & 0 & 3\end{bmatrix}$

3. + mat3(a:const vec3&, b:const vec3&, c:const vec3&)
   Constructor que inicializa las variables de instancia utilizando los parámetros a, b y c como vectores columna.
   Ejemplo:

```
vec3 a(1.0f, 2.0f, 3.0f);
vec3 b(4.0f, 5.0f, 6.0f);
vec3 c(7.0f, 8.0f, 9.0f);
```

   mat3 m(a, b, c); // Representa la matriz $\begin{bmatrix}1 & 4 & 7\\2 & 5 & 8\\3 & 6 & 9\end{bmatrix}$

4. + operator[](column:int):vec3&
   El operador [ ] permite tener acceso de escritura a los vectores columna almacenados en esta matriz utilizando un índice. El método no hace ningún chequeo del índice utilizado.
   Ejemplo:

```
vec3 a(1.0f, 2.0f, 3.0f);
vec3 b(4.0f, 5.0f, 6.0f);
vec3 c(7.0f, 8.0f, 9.0f);
```

   mat3 m(a, b, c); // Representa la matriz $\begin{bmatrix}1 & 4 & 7\\2 & 5 & 8\\3 & 6 & 9\end{bmatrix}$

   m[0] = c; // m es ahora $\begin{bmatrix}7 & 4 & 7\\8 & 5 & 8\\9 & 6 & 9\end{bmatrix}$

5. + *<<{const}>>* operator[](column:int):const vec3&

   El operador [ ] permite tener acceso de sólo lectura a los vectores columna almacenados en esta matriz utilizando un índice. El método no hace ningún chequeo del índice utilizado.

   Ejemplo:

```
vec3 a(1.0f, 2.0f, 3.0f);
vec3 b(4.0f, 5.0f, 6.0f);
vec3 c(7.0f, 8.0f, 9.0f);
mat3 m(a, b, c); // Representa la matriz
```

$$\begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$$

```
std::cout << m[1][2] << std::endl; // Imprime 6.0f
```

6. + *<<{const}>>* operator==(m:const mat3&):bool

   El operador == realiza una comparación de la instancia actual de `mat3` con la matriz `m`. La comparación se realiza a nivel de componente, es decir, todos los elementos correspondientes de ambas matrices deben ser idénticos. Se debe regresar `true` si todas las componentes son iguales en ambas matrices, `false` de lo contrario.

   Ejemplo:

```
mat3 m1(1.0f);
mat3 m2(0.0f);
m2[0][0] = 1.0f;
m2[1][1] = 1.0f;
m2[2][2] = 1.0f;
std::cout << a == b << std::endl; // El resultado es true.
```

7. + determinant(m:const mat3&):float

   El método estático `determinant(…)` calcula y regresa el determinante de la matriz 3x3 `m`.

   Ejemplo:

```
mat3 m(3.0f);
mat3::determinant(m); // Regresa 27.0f
```

8. + inverse(m:const mat3&):mat3

   El método estático `inverse(…)` calcula la matriz inversa de la matriz 3x3 `m` y la regresa como un nuevo objeto `mat3`. El método no modifica la `mat3 m`.

   Ejemplo:

```
mat3 m1(2.0f);
mat3 m2 = mat3::inverse(m1); // Representa la matriz
```

$$\begin{bmatrix} 0.5 & 0 & 0 \\ 0 & 0.5 & 0 \\ 0 & 0 & 0.5 \end{bmatrix}$$

9. + transpose(m:const mat3&):mat3

   El método estático `transpose(…)` calcula y regresa la matriz transpuesta de la matriz 3x3 `m`.

   Ejemplo:

```
vec3 a(1.0f, 2.0f, 3.0f);
vec3 b(-4.0f, -5.0f, -6.0f);
vec3 c(7.0f, 8.0f, 9.0f);
mat3 m1(a, b, c); // Representa la matriz
```

$$\begin{bmatrix} 1 & -4 & 7 \\ 2 & -5 & 8 \\ 3 & -6 & 9 \end{bmatrix}$$

```
mat3 m2 = mat3::transpose(m1); // Representa la matriz
```

$$\begin{bmatrix} 1 & 2 & 3 \\ -4 & -5 & -6 \\ 7 & 8 & 9 \end{bmatrix}$$

10. + *<<{friend}>>* operator<<(os:ostream&, m:const mat3&):ostream&

    El operador << crea un `string` con una representación en texto de la instancia sobre la cual se manda a llamar. La representación correcta debe imprimir las filas de la matriz, separadas en nuevas líneas. Cada componente de la fila debe estar separado por comas. Permite asociar un *output stream* (`cout` por ejemplo) con una mat3.

    Ejemplo:

```
vec3 a(1.0f, 2.0f, 3.0f);
vec3 b(4.0f, 5.0f, 6.0f);
vec3 c(7.0f, 8.0f, 9.0f);
mat3 m(a, b, c); // Representa la matriz [1 4 7; 2 5 8; 3 6 9]

std::cout << m << std::endl; // Imprime:     1 4 7
                                            2 5 8
                                            3 6 9
```

11. + *<<{inline}>>* operator*(m:const mat3&, v:const vec3&):vec3

    El operador * permite realizar una multiplicación entre una matriz 3x3 (operando izquierdo) y un vector columna de 3 componentes (operando derecho). El resultado debe ser almacenado en un nuevo objeto `vec3` que deberá regresarse como resultado de ejecutar este método.

    Ejemplo:

```
vec3 v1(1.0f, 2.0f, 3.0f);
mat3 m1(1.0f);

vec3 r1 = m1 * v1; // Representa el vector (1.0f, 2.0f, 3.0f)

vec3 v2(1.0f, 2.0f, 3.0f);
vec3 col1(1.0f, 2.0f, 3.0f);
vec3 col2(4.0f, 5.0f, 6.0f);
vec3 col3(7.0f, 8.0f, 9.0f);
mat3 m2(col1, col2, col3);

vec3 r2 = m2 * v2; // Representa el vector (30.0f, 36.0f, 42.0f)
```

12. + *<<{inline}>>* operator*(m1:const mat3&, m2:const mat3&):mat3

    El operador * permite realizar una multiplicación entre una matriz 3x3 (operando izquierdo) y otra matriz 3x3 (operando derecho). El resultado debe ser almacenado en un nuevo objeto `mat3` que deberá regresarse como resultado de ejecutar este método.

    Ejemplo:

```
vec3 col1(1.0f, 2.0f, 3.0f);
vec3 col2(4.0f, 5.0f, 6.0f);
vec3 col3(7.0f, 8.0f, 9.0f);
mat3 m(col1, col2, col3);
mat3 r = m * m; // Representa la matriz [30 66 102; 36 81 126; 42 96 150]
```

V. Dentro del proyecto **Math**, implementar la clase **mat4** con las especificaciones que se muestran a continuación. La clase **mat4** debe estar contenida en un *namespace* llamado **cgmath**. El proyecto **Math Test** contiene las pruebas unitarias (**mat4_test**) para verificar tu resultado. Los métodos de la clase **mat4** son equivalentes a los de la clase **mat3** pero utilizando 16 componentes en lugar de 9. La matriz debe estar codificada en formato *column-major*.

mat4
- n:float[4][4] = { 0 }
+ mat4() + mat4(diagonal:float) + mat4(a:const vec4&, b:const vec4&, c:const vec4&, d:const vec4&) + operator[](column:int):vec4& + *<<{const}>>* operator[](column:int):const vec4& + *<<{const}>>* operator==(m:const mat4&):bool + inverse(m:const mat4&):mat4 + *<<{friend}>>* operator<<(os:ostream&, m:const mat4&):ostream& + *<<{inline}>>* operator*(m:const mat4&, v:const vec4&):vec4 + *<<{inline}>>* operator*(m1:const mat4&, m2:const mat4&):mat4