project

2

Διανύσματα στις 3 Διαστάσεις

Περιεχόμενα:

Prj02.1 Το Πρόβλημα **485**
Prj02.2 Ο Τύπος *Vector3* και οι Δημιουργοί **486**
Prj02.3 Οι Τελεστές Σύγκρισης **487**
Prj02.4 Οι Τελεστές "+", "-", "*", "^" **488**
Prj02.5 Ο Ενικός Τελεστής "-" **489**
Prj02.6 Οι Τελεστές Εκχώρησης **489**
Prj02.7 Ο Τελεστής "<<" **490**
Prj02.8 ... και το Ευκλείδιο Μέτρο **490**
Prj02.9 Το Πρόγραμμα **490**

Prj02.1 Το Πρόβλημα

Το Project αυτό είναι ένα μεγάλο παράδειγμα επιφόρτωσης τελεστών. Είναι καλό να το διαβάσεις προσεκτικά διότι *αργότερα θα το ξαναδούμε και θα αλλάξουμε μερικά από αυτά πόυ θα δούμε εδώ.*

Ένα τρισδιάστατο (ελεύθερο) ***διάνυσμα*** *(vector) είναι μια διαταγμένη τριάδα πραγματικών αριθμών (x, y, z).*

Όρισε έναν τύπο δομής Vector3 που τα αντικείμενά της θα είναι διανύσματα του χώρου τριών διαστάσεων. Όρισε έναν ερήμην δημιουργό που θα μας δίνει ελεύθερο διάνυσμα με συνιστώσες (0, 0, 0) αλλά θα μπορεί να πάρει και αρχικές τιμές των συνιστωσών.

Α. Έστω ότι έχουμε δύο διανύσματα $\mathbf{v}_1 = (x_1, y_1, z_1)$ *και* $\mathbf{v}_2 = (x_2, y_2, z_2)$.

1. Τα $\mathbf{v}_1$ *και* $\mathbf{v}_2$ *μπορεί να συγκριθούν για ισότητα και μόνον (δηλαδή έχουν νόημα μόνον οι "==" και "!="). Είναι ίσα αν και μόνον αν* $x_1 = x_2 \wedge y_1 = y_2 \wedge z_1 = z_2$. *Επιφόρτωσε τους τελεστές σύγκρισης "==" και "!=" για τον Vector3.*

2. Μπορούμε ακόμη να τα προσθέσουμε ή τα αφαιρέσουμε και να πάρουμε ένα διάνυσμα $\mathbf{v}_1 \pm \mathbf{v}_2$ *με συνιστώσες* $(x_1 \pm x_2, y_1 \pm y_2, z_1 \pm z2)$.
Επιφόρτωσε τους τελεστές "+" και "-" για τον Vector3.
Επιφόρτωσε τους τελεστές "+=" και "-=" για τον Vector3.

3. Πολλαπλασιάζοντας ένα διάνυσμα $\mathbf{v}_1$ *επί έναν πραγματικό αριθμό α (*$\alpha\mathbf{v}_1$ *ή* $\mathbf{v}_1\alpha$*) παίρνουμε ένα νέο διάνυσμα με συνιστώσες* $(\alpha x_1, \alpha y_1, \alpha z_1)$.
Επιφόρτωσε καταλλήλως τον τελεστή "" για τον Vector3.*
Επιφόρτωσε τον τελεστή "=" για τον Vector3.*

4. Ειδική περίπτωση: το $(-1)\mathbf{v}_1$ *μπορεί να γραφεί και ως* $-\mathbf{v}_1$.
Επιφόρτωσε καταλλήλως τον ενικό τελεστή "-" για τον Vector3.

5. Μπορούμε να πολλαπλασιάσουμε τα $\mathbf{v}_1$ *και* $\mathbf{v}_2$ *και να πάρουμε τον πραγματικό αριθμό:* $\mathbf{v}_1 \cdot \mathbf{v}_2 = x_1x_2 + y_1y_2 + z_1z_2$ *που λέγεται* ***εσωτερικό γινόμενο*** *(inner ή dot product).*
*Επιφόρτωσε (άλλη μια φορά) καταλλήλως τον τελεστή "*****" για τον Vector3.*
6. Μπορούμε όμως να πάρουμε και το ***εξωτερικό γινόμενο*** *(outer ή cross product)* $\mathbf{v}_1 \times \mathbf{v}_2$ *που είναι διάνυσμα:* $\mathbf{v}_1 \times \mathbf{v}_2 = (y_1z_2-z_1y_2,\ x_2z_1-z_2x_1,\ y_2x_1-x_2y_1)$.
*Επιφόρτωσε καταλλήλως τον τελεστή "**^**"για τον Vector3 ώστε να υπολογίζει το εξωτερικό γινόμενο.*
*7. Καλό θα είναι να μπορούμε να γράψουμε στην οθόνη ή σε ένα αρχείο text την τιμή του διανύσματος με τον "**<<**".*
*Επιφόρτωσε τον "**<<**" για αντικείμενα τύπου Vector3.*

8. Τέλος, για κάθε διάνυσμα ορίζεται ένα ευκλείδιο μέτρο: $|\mathbf{v}| = \sqrt{x^2+y^2+z^2}$ *(=* $\sqrt{\mathbf{v}\cdot\mathbf{v}}$ *) που είναι το μήκος του διανύσματος.*
Γράψε συνάρτηση Vector3_abs που να δίνει το ευκλείδιο μέτρο ενός αντικειμένου τύπου Vector3.
Β. Για μια απλή εφαρμογή των παραπάνω, γράψε πρόγραμμα που θα υπολογίζει τη δύναμη Laplace $\mathbf{F}_L = q(\mathbf{v} \times \mathbf{B})$ *που ασκείται σε φορτίο* $q = 1\ \mu Cb$ *που εισέρχεται με ταχύτητα* $\mathbf{v} = (10^8, 0, 0)$ *σε μαγνητικό πεδίο* $\mathbf{B} = (0, 0, 10)$. *Επιβεβαίωσε ότι η δύναμη είναι κάθετη στα* $\mathbf{v}$ *και* $\mathbf{B}$ *(δηλαδή:* $\mathbf{F}_L \cdot \mathbf{v} = 0$ *και* $\mathbf{F}_L \cdot \mathbf{B} = 0$*).*
Το εμβαδόν παραλληλογράμμου που οι πλευρές του είναι παράλληλες προς τα ελεύθερα διανύσματα $\mathbf{a}$ *και* $\mathbf{b}$ *είναι το διάνυσμα* $\mathbf{S} = \mathbf{a} \times \mathbf{b}$. *Αν έχουμε το παραλληλόγραμμο μέσα σε ένα μαγνητικό πεδίο τότε η μαγνητική ροή που διέρχεται από αυτό είναι ίση με* $\mathbf{B} \cdot \mathbf{S}$. *Συμπλήρωσε το πρόγραμμά σου με εντολές που θα υπολογίζουν τη μαγνητική ροή που διέρχεται από παραλληλόγραμμο με πλευρές* $\mathbf{a} = (0.1, 0.2, 0.3)$ *και* $\mathbf{b} = (0.2, 0.3, 0.4)$ *όταν βρίσκεται στο μαγνητικό πεδίο που δώσαμε παραπάνω.*
Με τα $\mathbf{a}$ *και* $\mathbf{b}$ *επιβεβαίωσε την ταυτότητα:* $|\mathbf{a}|^2|\mathbf{b}|^2 = (\mathbf{a}\cdot\mathbf{b})^2 + |\mathbf{a}\times\mathbf{b}|^2$.

Prj02.2 Ο Τύπος *Vector3* και οι Δημιουργοί

Ο τύπος αυτός είναι σαν τον *complex* αλλά με τρία μέλη:

```
struct Vector3
{
   double x;
   double y;
   double z;
}; // Vector3
```

Γράφουμε τον ερήμην δημιουργό με αρχικές τιμές όπως κάναμε και στην *complex*:

```
   Vector3( double ax=0, double ay=0, double az=0 )
   {  x = ax;  y = ay;  z = az;  }
```

Πρόσεξε ότι με αυτόν τον δημιουργό μπορούμε να κάνουμε τις εξής δηλώσεις:

```
   Vector3 v0;                                 // (0, 0, 0)
   Vector3 v1( 1 );                            // (1, 0, 0)
   Vector3 v2( sqrt(2), 3 );                   // (1.41421, 3, 0)
   Vector3 v3( sqrt(3), sqrt(5), 2*sqrt(2) ); // (1.73205, 2.23607, 2.82843)
```

Να λοιπόν ο ορισμός του τύπου μας:

```
struct Vector3
{
   double x;
   double y;
   double z;
   Vector3( double ax=0, double ay=0, double az=0 )
```

```
   {  x = ax;  y = ay;  z = az;  }
}; // Vector3
```

Prj02.3 Οι Τελεστές Σύγκρισης

Σύμφωνα με όσα είπαμε στην §14.6.4, ένας δυαδικός τελεστής ("`@`") επιφορτώνεται με μια συνάρτηση της μορφής:

```
Trv operator@( Tl lhs, Tr rhs )
```

Για έναν τελεστή σύγκρισης, όπως είναι ο "`==`", *Trv* είναι ο `bool`. Για την περίπτωσή μας *Tl* και *Tr* είναι, κατ' αρχήν, ο *Vector3*. Θα μπορούσαμε λοιπόν να γράψουμε:

```
bool operator==( Vector3 lhs, Vector3 rhs )
{
   return (lhs.x == rhs.x) &&
          (lhs.y == rhs.y) && (lhs.z == rhs.z);
} // operator==( Vector3
```

Αυτή είναι απολύτως σωστή, αλλά, παρ' όλα αυτά, θα την αλλάξουμε λιγάκι, ως προς τις παραμέτρους:

```
bool operator==( const Vector3& lhs, const Vector3& rhs )
{
   return (lhs.x == rhs.x) &&
          (lhs.y == rhs.y) && (lhs.z == rhs.z);
} // operator==( Vector3
```

Ποιο είναι το πλεονέκτημα της δεύτερης μορφής; Κάθε φορά που την καλούμε θα περάσουν, ως παράμετροι, δύο βέλη ενώ στην πρώτη μορφή θα περάσουν δυο τιμές *Vector3*. Σε ψηφιολέξεις αυτό θα μπορούσε να σημαίνει (ενδεικτικώς) 8:48· εξαπλάσιο! Ναι, αλλά το 48 είναι πολύ μικρό για να μας δημιουργήσει πρόβλημα. Υπάρχουν όμως και περιπτώσεις αντικειμένων που μπορεί να είναι πολύ μεγάλα· σκέψου, για παράδειγμα, ένα αντικείμενο τύπου *string* με τιμή το κείμενο ενός βιβλίου 1000 σελίδων. *Τα πολύ μεγάλα αντικείμενα δεν τα περνάμε ως παραμέτρους τιμής αλλά ως παραμέτρους αναφοράς με "`const`"*. Θα χρησιμοποιούμε λοιπόν αυτόν τον τρόπο παντού στις επιφορτώσεις τελεστών για να τον συνηθίσεις(!)

Ας έλθουμε τώρα στον "`!=`"· χρειάζεται να τον γράψουμε; Αφού έχουμε τον "`==`" θα μπορούμε να γράφουμε `!(a == b)` αντί για `a != b`. Φυσικά, αλλά γενικώς θα τηρούμε την εξής προγραμματιστική πρακτική:[1]

- ***Όταν επιφορτώνουμε έναν τελεστή σύγκρισης "@" επιφορτώνουμε και τον αντίθετό του με χρήση του "@".***

Έχουμε λοιπόν:

```
bool operator!=( const Vector3& lhs, const Vector3& rhs )
{
   return !(lhs == rhs);
} // operator!=( Vector3
```

Έτσι, έχουμε και τους δύο τελεστές χωρίς ασυμβατότητες (σίγουρα). Αν θέλεις μπορείς να πας ένα βήμα παραπέρα και να αποφύγεις μια κλήση συνάρτησης:

```
bool operator!=( const Vector3& lhs, const Vector3& rhs )
{
   return (lhs.x != rhs.x) ||
          (lhs.y != rhs.y) || (lhs.z != rhs.z);
} // operator!=( Vector3
```

Για τελεστές που αυτό το τελευταίο βήμα είναι πιο πολύπλοκο –και υπάρχει μεγάλη πιθανότητα λάθους– μην το κάνεις.

---

[1] Η σύσταση 36 της (ELLEMTEL 1998) λέει: «*When two operators are opposite (such as "==" and "!="), it is appropriate to define both.*»

Prj02.4 Οι Τελεστές "+", "-", "*", "^"

Οι δύο προσθετικοί (δυαδικοί) τελεστές και ο "^" (εξωτερικό γινόμενο) θα επιφορτωθούν με συναρτήσεις της μορφής:

```
Trv operator@( Tl lhs, Tr rhs )
```

Γι ά τις πράξεις αυτές έχουμε:

+: *Vector3* × *Vector3* ↛ *Vector3*

-: *Vector3* × *Vector3* ↛ *Vector3*

^: *Vector3* × *Vector3* ↛ *Vector3*

Τις θεωρούμε μερικές συναρτήσεις επειδή παίρνουμε υπόψη την περίπτωση υπερχείλησης. Στη συνέχεια, στην υλοποίηση, θα τις χειριστούμε σαν ολικές συναρτήσεις· δηλαδή δεν θα ρίχνουμε εξαιρέσεις.

Στις περιπτώσεις αυτές *Trv*, *Tl* και *Tr* είναι ο *Vector3*. Σύμφωνα όμως με αυτά που είπαμε παραπάνω, θα βάλουμε τους *Tl* και *Tr* `const Vector3&`:

```
Vector3 operator+( const Vector3& lhs, const Vector3& rhs )
{
   Vector3 fv;
   fv.x = lhs.x + rhs.x;
   fv.y = lhs.y + rhs.y;
   fv.z = lhs.z + rhs.z;
   return fv;
} // operator+( const Vector3&

Vector3 operator-( const Vector3& lhs, const Vector3& rhs )
{
   Vector3 fv;
   fv.x = lhs.x - rhs.x;
   fv.y = lhs.y - rhs.y;
   fv.z = lhs.z - rhs.z;
   return fv;
} // operator-( const Vector3&

Vector3 operator^( const Vector3& lhs, const Vector3& rhs )
{
   Vector3 fv;
   fv.x = lhs.y*rhs.z - lhs.z*rhs.y;
   fv.y = rhs.x*lhs.z - rhs.z*lhs.x;
   fv.z = rhs.y*lhs.x - rhs.x*lhs.y;
   return fv;
} // operator^( const Vector3
```

Σημείωση:▶

Υπάρχει δυαδικός τελεστής "^" στη C++; Ναι, αλλά θα τον μάθουμε αργότερα. Στην §14.6 λέγαμε «*Δεν αλλάζουμε το νόημα του τελεστή που επιφορτώνουμε.*» Αυτό το σεβόμαστε με την επιφόρτωση που κάνουμε; Όχι, αλλά όπως θα καταλάβεις δεν δημιουργεί οποιαδήποτε σύγχυση.◀

Ο "*" θέλει να σκεφτούμε κάτι παραπάνω. Αν έχουμε δηλώσει:

```
Vector3 v1, v2;
double  a;
```

θα πρέπει να μπορούμε να γράψουμε είτε:

```
v2 = a*v1;
```

είτε:

```
v2 = v1*a;
```

Έχουμε δηλαδή δύο περιπτώσεις:

*: `double` × *Vector3* ↛ *Vector3*

*: *Vector3* × `double` ↛ *Vector3*

Θα κάνουμε λοιπόν διπλή επιφόρτωση του "*":

```
Vector3 operator*( double lhs, const Vector3& rhs )
{
   Vector3 fv;
   fv.x = lhs * rhs.x;
   fv.y = lhs * rhs.y;
   fv.z = lhs * rhs.z;
   return fv;
} // operator*

Vector3 operator*( const Vector3& lhs, double rhs )
{
   Vector3 fv;
   fv.x = rhs * lhs.x;
   fv.y = rhs * lhs.y;
   fv.z = rhs * lhs.z;
   return fv;
} // operator*
```

Ο "`*`" πρέπει να επιφορτωθεί άλλη μια φορά για το εσωτερικό γινόμενο:

`*`: *Vector3* × *Vector3* ↛ `double`

που υλοποιείται με την

```
double operator*( const Vector3& lhs, const Vector3& rhs )
{
   return lhs.x*rhs.x + lhs.y*rhs.y + lhs.z*rhs.z;
} // operator*( const Vector3&
```

Prj02.5 Ο Ενικός Τελεστής "-"

Αν έχουμε δηλώσει:

```
   Vector3 v1, v2;
```

και δώσουμε:

```
   v2 = -v1;
```

το διάνυσμα $\mathbf{v}_2$ είναι το αντίθετο του $\mathbf{v}_1$, δηλαδή: $\mathbf{v}_1 + \mathbf{v}_2 = \mathbf{0}$.

Για τον "`-`" έχουμε:

`-`: *Vector3* → *Vector3*

Στην §14.6.4 λέγαμε ότι επιφορτώνουμε έναν *προθεματικό ενικό* τελεστή "`@`" με μια συνάρτηση

```
   Trv operator@( T rhs )
```

και είδαμε ήδη μια εφαρμογή αυτού του κανόνα στον "`-`" για τον *complex* (§15.5). Τώρα, *Trv* και *T* είναι ο *Vector3*:

```
Vector3 operator-( const Vector3& rhs )
{
   return Vector3( -rhs.x, -rhs.y, -rhs.z );
} // operator-( const Vector3&
```

Prj02.6 Οι Τελεστές Εκχώρησης[2]

Τώρα θα επιφορτώσουμε τους τρεις τελεστές εκχώρησης "`+=`", "`-=`", "`*=`" όπως είπαμε στην §14.6.3. Εκεί είδαμε ότι ο "+=" για τον τύπο *T* επιφορτώνεται ως:

```
T& operator+=( T& lhs, const T& rhs )
```

Προσεξε ότι ο τύπος της πρώτης παραμέτρου δεν έχει `const`.

Στην περίπτωσή μας *T* είναι ο *Vector3* οπότε έχουμε:

```
Vector3& operator+=( Vector3& lhs, const Vector3& rhs )
{
```

[2] Αργότερα θα μάθουμε ότι ο πάγιος τρόπος επιφόρτωσης αυτών των τελεστών είναι διαφορετικός.

```
   lhs.x += rhs.x;
   lhs.y += rhs.y;
   lhs.z += rhs.z;
   return lhs;
} // operator+=( Vector3&
```

Παρομοίως γίνεται η επιφόρτωση και των άλλων δύο τελεστών:

```
Vector3& operator-=( Vector3& lhs, const Vector3& rhs )
{
   lhs.x -= rhs.x;
   lhs.y -= rhs.y;
   lhs.z -= rhs.z;
   return lhs;
} // operator-=( const Vector3&

Vector3& operator*=( Vector3& lhs, double rhs )
{
   lhs.x *= rhs;
   lhs.y *= rhs;
   lhs.z *= rhs;
   return lhs;
} // operator*=( const Vector3&
```

Prj02.7 Ο Τελεστής "<<"

Έχουμε ήδη επιφορτώσει τον "<<" για αρκετους τύπους. Αντιγράφοντας σχεδόν την επιφόρτωση για τον τύπο *complex* (§15.5) έχουμε:

```
ostream& operator<<( ostream& tout, const Vector3& rhs )
{
   return tout << "(" << rhs.x << ", " << rhs.y << ", "
               << rhs.z << ")";
} // operator<<
```

Prj02.8 ... και το Ευκλείδιο Μέτρο

Για το ευκλείδιο μέτρο δεν έχουμε κάποιον βολικό (από οπτική άποψη) τελεστή. Θα γράψουμε λοιπόν μια:

```
double Vector3_abs( const Vector3& lhs )
{
   return sqrt( lhs.x*lhs.x + lhs.y*lhs.y + lhs.z*lhs.z );
} // double Vector3_abs
```

Prj02.9 Το Πρόγραμμα

Έχοντας αυτά τα εργαλεία το πρόγραμμα είναι τετριμμένο. Για να διευκολύνουμε το γράψιμο (και για να κάνουμε οικονομία στις πράξεις) της τελευταίας ερώτησης ορίζουμε μια επιπλέον συνάρτηση:

```
double Vector3_abs2( const Vector3& lhs )
{
   return ( lhs.x*lhs.x + lhs.y*lhs.y + lhs.z*lhs.z );
} // double Vector3_abs2
```

που για ένα διάνυσμα **a** μας δίνει το $|\mathbf{a}|^2$.

Αυτό που περιμένουμε να δούμε είναι ο μηδενισμός της παράστασης:

```
Vector3_abs2(a)*Vector3_abs2(b) - ((a*b)*(a*b)+Vector3_abs2(a^b))
```

Γράφουμε λοιπόν το πρόγραμμα:

```
#include <iostream>
#include <cmath>
```

```
using namespace std;

struct Vector3
{
   double x;
   double y;
   double z;
   Vector3( double ax=0, double ay=0, double az=0 )
   {  x = ax;  y = ay;  z = az;  }
   Vector3( const Vector3& rhs  )
   {  x = rhs.x;  y = rhs.y;  z = rhs.z;  }
}; // Vector3

bool operator==( const Vector3& lhs, const Vector3& rhs );
bool operator!=( const Vector3& lhs, const Vector3& rhs );
Vector3 operator-( const Vector3& rhs );
Vector3 operator+( const Vector3& lhs, const Vector3& rhs );
Vector3& operator+=( Vector3& lhs, const Vector3& rhs );
Vector3 operator-( const Vector3& lhs, const Vector3& rhs );
Vector3& operator-=( Vector3& lhs, const Vector3& rhs );
Vector3 operator*( double lhs, const Vector3& rhs );
Vector3 operator*( const Vector3& lhs, double rhs );
Vector3& operator*=( Vector3& lhs, double rhs );
double operator*( const Vector3& lhs, const Vector3& rhs );
Vector3 operator^( const Vector3& lhs, const Vector3& rhs );
ostream& operator<<( ostream& tout, const Vector3& rhs );
double Vector3_abs( const Vector3& lhs );
double Vector3_abs2( const Vector3& lhs );

int main()
{
   double q( 1e-6 ); // Cb
   Vector3 v( 1e8, 0, 0 );
   Vector3 B( 0, 0, 10 );
   Vector3 FL;

   FL = q*( v ^ B );
   cout << FL << endl;
   cout << FL*v << " " << FL*B << endl;

   Vector3 a( 0.1, 0.2, 0.3 ), b( 0.2, 0.3, 0.4 );

   cout << a << "  " << b << endl;
   cout << B*( a ^ b) << endl;

   cout << Vector3_abs2(a)*Vector3_abs2(b) -
           ((a*b)*(a*b)+Vector3_abs2(a^b)) << endl;
   cout << Vector3_abs2(a)*Vector3_abs2(b) << endl;
} // main
```

Αποτέλεσμα:

```
(0, -1000, 0)
0 0
(0.1, 0.2, 0.3)  (0.2, 0.3, 0.4)
-0.1
-3.27294e-018
0.0406
```

Οι τελευταίες δύο γραμμές χρειάζονται ένα σχόλιο. Η διαφορά που περιμένουμε μηδέν βγαίνει περίπου $-3.3 \cdot 10^{-18}$. Αλλά το γινόμενο $|\mathbf{a}|^2|\mathbf{b}|^2$ έχει τιμή $0.0406 \approx 4 \cdot 10^{-2}$. Όπως βλέπεις, κατ' απόλυτη τιμή, η διαφορά είναι 10^{16} φορές μικρότερη. Μπορούμε λοιπόν να τη θεωρήσουμε μηδέν (0).