Προγραμματισμός Υπολογιστών με C++

ΤΕΛΕΣΤΕΣ

Γεώργιος Παπαϊωάννου (2013-20)
gepap@aueb.gr

Περιεχόμενο Παρουσίασης

- Περιγραφή:
  - Τι είναι οι τελεστές
  - Χρήση τελεστών
  - Υπερφόρτωση τελεστών
  - Chain-linking Τελεστών

- Τελευταία ενημέρωση: Νοέμβριος 2020

Τελεστές

- Τελεστής είναι μια βασική πράξη που μπορούμε να εκτελέσουμε πάνω σε ένα ή περισσότερα δεδομένα και να πάρουμε ένα νέο αποτέλεσμα το οποίο είτε επιστρέφεται ως μια νέα τιμή, είτε αποθηκεύεται στο αρχικό όρισμα (π.χ. ο τελεστής ++)
- Στη C++ οι περισσότεροι από τους γνωστούς τελεστές μπορούν να υπερφορτωθούν

Υπερφόρτωση Τελεστών – Γιατί;

- Πολύ συχνά θέλουμε να αντικαταστήσουμε μια δύσχρηστη μέθοδο ή συνάρτηση με μια πιο αυθόρμητη και απλή πράξη
- Παράδειγμα:

```
vec3 v1(1,2), v2(3,4), v3();
v3 = AddVectors(v1,v2);
```

Θα θέλαμε:

```
v3 = v1+v2;
```

- Εδώ, πρέπει να επαναπροσδιορίσουμε την έννοια του τελεστή + προκειμένου να υλοποιεί άλλους υπολογισμούς όταν οι τελεστέοι είναι vec3

Ποιοι Τελεστές Μπορούν να Υπερφορτωθούν;

- Σχεδόν όλοι!
- Τόσο σε επίπεδο κλάσης (ως μέθοδοι), όσο και σε μορφή ελεύθερων τελεστών (ως συναρτήσεις)
- Π.χ:
  - +,-,*,/,++,-=,=,|,&,^,~,<<,>>,>,<,==,!=, …
  - ->
  - (…)
  - ,
  - Τα καλουπώματα (typeA)a, typeA(a)
  - [ ]
  - Οι τελεστές κλάσεων delete, new, delete[] και new[]

Ποιοι Τελεστές Δε Μπορούν να Υπερφορτωθούν;

- Κάποιες ανακατευθύνσεις: a.b, a.*b
- Ο τριαδικός τελεστής συνθήκης: a?b:c
- Ο τελεστής μέλους: a::b
- Οι τελεστές ελέγχων με βάση τύπο: sizeof(a), typeid(a)

Υπερφόρτωση Τελεστών Κλάσης - Παράδειγμα: +

```
class vec2 {
  float x, y;
public:
  vec2(float xx = 0, float yy = 0) { x = xx; y = yy; }
  void print() const { cout << x << "," << y << endl; }
  vec2 operator+(const vec2& p) const; };


vec2 vec2::operator+(const vec2& p) const {
   vec2 temp(x + p.x, y + p.y);
   return temp;
} // Ή κατευθείαν «return vec2(x + p.x, y + p.y);» χωρίς temp.


int main() {
   vec2 p1(1, 2), p2(3, 4);
   vec2 p3 = p1 + p2;            // Χρήση υπερφορτωμένου τελεστή +.
   vec2 p4 = p1.operator+(p2);   // Ισοδύναμη μορφή – δε χρησιμοποιείται
   p1.print();  p2.print();            // Τυπώνουν «1,2», «3,4».
   p3.print();  p4.print();            // Τυπώνουν «4,6», «4,6».
}
```

Περισσότερα για τους Τελεστές

- Μπορούν να υπερφορτωθούν μόνο **υπάρχοντες τελεστές** της C++ (π.χ., +, ++, >>, <<).
- Δεν είναι δυνατή η αλλαγή της **σύνταξης** των τελεστών (π.χ. ο + απαιτεί δύο ορίσματα).
- Η κλάση τουλάχιστον ενός από τα ορίσματα πρέπει να είναι **ορισμένη από το χρήστη** (π.χ. δεν επιτρέπεται να αλλάξουμε τη σημασία του + για πρόσθεση ακεραίων).

Προσοχή με unary operators – Παράδειγμα: ++ (1)

```
class vec2 {
    float x, y;
public:
    vec2(float xx = 0, float yy = 0) { x = xx; y = yy; }
    void print() const { cout << x << "," << y << endl; }
    void operator++();
};

void vec2::operator++() {
    ++x; ++y;
}

int main() {
    vec2 p1(1, 2), p2;
    ++p1;            // Ισοδύναμα: p1.operator++();
    p1.print();      // Τυπώνει «2,3».
    p2 = ++p1;       // Λάθος. Η κλήση ++p1 δεν επιστρέφει τίποτα.
}
```

Προσοχή με unary operators – Παράδειγμα: ++ (2)

```
class vec2 {
   float x, y;
public:
   vec2(float xx = 0, float yy = 0) { x = xx; y = yy; }
   void print() const { cout << x << "," << y << endl; }
   vec2 operator++();
};

vec2 vec2::operator++() {      // Αλλάζει τα x και y του αντικειμένου
   return vec2(++x, ++y); }    // του οποίου τη μέθοδο καλέσαμε και
                               // επιστρέφει αντίγραφο του αντικειμένου.
int main() {
   vec2 p1(1, 2), p2, p3;
   p2 = ++p1;                  // Το ++p1 είναι ένα αντίγραφο του p1.
   p1.print(); p2.print();     // OK. Τυπώνει «2,3», «2,3».
   p3 = ++(++p1);              // Το αριστερό ++ δεν εφαρμόζεται στο p1,
                               // αλλά στο νέο vec2 που επιστρέφεται.
   p1.print(); p3.print();     // Πρόβλημα: τυπώνει «3,4», «4,5».
}
```

Προσοχή με unary operators – Παράδειγμα: ++ (3)

```
class vec2 {
   float x, y;
public:
   vec2(float xx = 0, float yy = 0) { x = xx; y = yy; }
   void print() const { cout << x << "," << y << endl; }
   vec2& operator++();
};

vec2& vec2::operator++() {
   ++x; ++y;    // Ο δείκτης this == διεύθυνση του αντικειμένου που
                // καλέσαμε τη μέθοδο. *this είναι αυτό το αντικείμενο.
   return *this;
}
void main() {
   vec2 p1(1, 2), p2, p3;
   p2 = ++p1;
   p1.print(); p2.print();       // OK. Τυπώνει «2,3», «2,3».
   p3 = ++(++p1);               // Το ++p1 είναι αναφορά στο p1.
   p1.print(); p3.print(); }    // OK. Τυπώνει «4,5», «4,5».
```

Προθεματικοί και Επιθεματικοί Τελεστές

- Πώς διαφοροποιούνται οι προθεματικοί (prefix) τελεστές από τους μεταθεματικούς (postfix);
  - Π.χ.: ++a, a++

```
class vec2 {
   float x, y;
public:
   vec2(float xx = 0, float yy = 0) { x = xx; y = yy; }
   void print() const { cout << x << "," << y << endl; }
   vec2& operator++();
   vec2& operator++(int);
};
```

Δε χρησιμοποιείται το όρισμα πουθενά, απλά χρησιμεύει για να καταλάβει ο μεταγλωττιστής ότι πρόκειται για την postfix έκδοση του τελεστή

Προβλήματα Επιθεματικών Τελεστών

- Δεδομένου ότι για παράδειγμα ο τελεστής operator++ (π.χ. v++) είναι μια μέθοδος κατά βάση, δεν απαγορεύεται εν γένει να καλέσουμε: ((v++)++)++ …
- Αυτό όμως συντακτικά απαγορεύεται για τους τελεστές βασικών τύπων
- Επίσης, η κλήση vec2 v = (v1++)++ δε θα έκανε αυτό που πιθανά να θέλαμε, δηλαδή να αυξήσει το v1 και μετά να το πάρουμε στο v.
- Ακόμα, τα δεύτερο (δεξιό) ++ που θα εφαρμοζόταν; Δε θα χρησίμευε κάπου.

Επίλυση Προβλημάτων Επιθεματικών Τελεστών

```
const vec2 vec2::operator++(int) {
   vec2 temp(x, y);
   x++; y++;
   return temp;}

int main() {
   vec2 p1(1, 2), p2;
   p2 = (p1++)++;  // Τώρα πλέον δεν επιτρέπεται, γιατί το
                   // αντικείμενο vec2 (αντίγραφο) που
                   // επιστρέφει η κλήση p1++ είναι const.
   p2 = p1++;      // Επιτρέπεται.
   p1.print(); p2.print();    // Τυπώνει «2,3» και «1,2».
}
```

Ελεύθεροι (non-member) Τελεστές

- Όπως μπορώ να ορίσω μεθόδους – τελεστές, έτσι μπορώ να ορίσω τελεστές σε global scope
- Προφανώς, δεν υπάρχει το ίδιο εύρος τελεστών που μπορούν να υπερφορτωθούν (π.χ. δεν ορίζεται ο ->)

Τελεστές Ελεύθεροι ή Μέλη Κλάσης;

- Υπάρχουν γενικοί κανόνες για την επιλογή (αλλά προφανώς και εξαιρέσεις…):
  - Αν είναι μοναδιαίος (unary) τελεστής: μέλος της κλάσης
  - Αν είναι δυαδικός (binary) τελεστής που δεν αλλοιώνει καμία από τις δύο κλάσεις – ορίσματα: ελεύθερος τελεστής
  - Αν είναι δυαδικός τελεστής που μεταβάλλει ένα από τα ορίσματα, τότε τον κάνουμε μέλος της αντίστοιχης κλάσης
  - Οι τελεστές ->, [ ], ( ) είναι υποχρεωτικά μέλη κλάσης

Chain-linking Τελεστών

- Συχνά χρειάζεται να στοιβάζουμε τελεστές στη σειρά
- Οι τυπικοί τελεστές που επενεργούν πάνω σε αντικείμενα και επιστρέφουν αντικείμενα του ίδιου τύπου συνδέονται στη σειρά εύκολα
- Το πρόβλημα προκύπτει όταν στον τελεστή μετέχουν ετερογενή δεδομένα, όπως στην περίπτωση των τελεστών «stream out/in»: <<, >>

Chain-linking Τελεστών – Το Παράδειγμα του << (1)

```
class vec2 {
   float x, y;
public:
   vec2(float xx = 0, float yy = 0) { x = xx; y = yy; }
   friend void operator << (ostream& os, const vec2& p);
};

void operator<<(ostream& os, const vec2& p) {
   os << p.x << ", " << p.y; }

int main() {
   vec2 p1(1, 2);
   cout << p1;                  // Τυπώνει «1, 2». Το cout ανήκει στην κλάση ostream.
   operator<<(cout, p1);        // Ισοδύναμη με την προηγούμενη γραμμή.
   cout << p1 << "test";        // Δεν δουλεύει, γιατί
                                // το cout << p1 δεν επιστρέφει τίποτα.
                                // Ισοδύναμη μορφή: (cout << p1) << "test";
}
```

Chain-linking Τελεστών – Το Παράδειγμα του << (2)

```
class vec2 {
   float x, y;
public:
   vec2(float xx = 0, float yy = 0) { x = xx; y = yy; }
   friend ostream& operator << (ostream& os, const vec2& p);
};
ostream& operator << (ostream& os, const vec2& p) {
   os << p.x << ", " << p.y;
   return os;
}

int main() {
   vec2 p1(1, 2);
   cout << p1 << endl << "test" << endl;    // OK.
   // Ισοδύναμη μορφή:
   (((cout << p1) << endl) << "test") << endl;
}
```