ΑΝΤΙΚΕΙΜΕΝΟΣΤΡΕΦΗΣ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ ΜΕ C++

Προετοιμασία για το εργαστήριο 3

Υπερφόρτωση συναρτήσεων και τελεστών, δείκτες, δέσμευση και αποδέσμευση μνήμης με new και delete, πίνακες, πολυδιάστατοι πίνακες, κατασκευαστής αντιγραφής

`fun(int) / fun(double)` `operator<` `new int[n]` `int a[5][5]` `A(const A &o)`

ΥΠΕΡΦΟΡΤΩΣΗ ΣΥΝΑΡΤΗΣΕΩΝ

Ίδιο όνομα, διαφορετικές παράμετροι

```
void fun(int a) {
    cout << "int: " << a << endl;
}
void fun(double b) {
    cout << "double: " << b << endl;
}
void fun(int a, double b) {
    cout << a << " " << b << endl;
}
```

Κλήση	Έκδοση	Εμφανίζει
`fun(5)`	`fun(int)`	`int: 5`
`fun(2.5)`	`fun(double)`	`double: 2.5`
`fun(5, 2.5)`	`fun(int, double)`	`5 2.5`
`fun('A')`	`fun(int)`	`int: 65`

Επιλογή από τα ορίσματα

Ο compiler κοιτά τον αριθμό και τον τύπο των ορισμάτων. Ο τύπος επιστροφής δεν παίζει ρόλο.

Το 'A' προάγεται σε int

Δεν υπάρχει fun(char): ο χαρακτήρας γίνεται 65 και καλείται η fun(int).

ΥΠΕΡΦΟΡΤΩΣΗ ΣΥΝΑΡΤΗΣΕΩΝ · ΣΥΝΑΡΤΗΣΕΙΣ ΜΕΛΗ

Υπερφόρτωση και σε συναρτήσεις μέλη

```
class A {
public:
    void fun(int a) { cout << a << endl; }
    void fun(double b) { cout << b << endl; }
    void fun(int a, double b) {
        cout << a << " " << b << endl;
    }
};
int main() {
    A obj;
    obj.fun(5);
    obj.fun(2.5);
    obj.fun(5, 2.5);
}
```

Εμφανίζει: 5, 2.5 και 5 2.5, μία τιμή ανά γραμμή.

Ίδιος κανόνας επιλογής

Η obj.fun(5) καλεί τη fun(int). Οι εκδόσεις είναι μέλη της κλάσης A.

Δεν αρκεί ο τύπος επιστροφής

Οι int fun(int) και double fun(int) μαζί δίνουν σφάλμα μεταγλώττισης.

Προσοχή στην ασάφεια

Με g(int, double) και g(double, int), η g(1, 2) ταιριάζει και στις δύο: σφάλμα.

ΥΠΕΡΦΟΡΤΩΣΗ ΤΕΛΕΣΤΩΝ

Ο τελεστής < συγκρίνει, ο << εμφανίζει

```
class A {
    int a, b, c;
public:
    A(int a, int b, int c)
        : a(a), b(b), c(c) {}
    int sum() const {
        return a + b + c;
    }
    bool operator<(const A &o) const {
        return sum() < o.sum();
    }
    friend ostream &operator<<(
        ostream &os, const A &x);
};
```

```
ostream &operator<<(ostream &os,
                    const A &x) {
    os << x.a << " " << x.b << " "
       << x.c << " sum=" << x.sum();
    return os;
}
int main() {
    A p(1, 2, 3), q(4, 0, 1);
    cout << p << endl;
    cout << (p < q) << endl;
}
```

Εμφανίζει `1 2 3 sum=6` `0`

Το άθροισμα του p είναι 6, του q είναι 5, άρα p < q είναι false.

Ισοδύναμα: p < q ≡ p.operator<(q) · cout << p ≡ operator<<(cout, p)

ΥΠΕΡΦΟΡΤΩΣΗ ΤΕΛΕΣΤΩΝ · ΑΣΚΗΣΗ

Σύγκριση πελατών με < και ==

```
class customer {
    string name;
    double balance;
public:
    customer(string n, double b)
        : name(n), balance(b) {}
    bool operator<(const customer &o) const {
        return balance < o.balance;
    }
    bool operator==(const customer &o) const {
        return balance == o.balance;
    }
};
```

```
customer a("Nick", 120.0);
customer b("Mary", 120.0);
customer c("Anna", 80.0);
cout << (c < a) << endl;  // 1
cout << (a == b) << endl; // 1
cout << (a < b) << endl;  // 0
```

Μετράει μόνο το υπόλοιπο

Οι a και b έχουν άλλο όνομα αλλά ίσο υπόλοιπο, άρα a == b.

Μέλη με const

Η a < b σημαίνει a.operator<(b) και δεν αλλάζει το αντικείμενο.

ΔΕΙΚΤΕΣ

Με δείκτες επιστρέφουμε τιμές από συνάρτηση

```
void sum_prod(int a, int b,
              int *s, int *p) {
    *s = a + b;
    *p = a * b;
}
int main() {
    int s, p;
    sum_prod(3, 4, &s, &p);
    cout << s << " " << p;
}
```

Εμφανίζει **7 12**

& και *

Το &x δίνει τη διεύθυνση του x. Το *p διαβάζει ή γράφει εκεί που δείχνει ο p.

Πέρασμα με δείκτη

Η συνάρτηση παίρνει διευθύνσεις και γράφει απευθείας στις μεταβλητές της main.

Ο δείκτης είναι μεταβλητή

Έχει δική του διεύθυνση και μπορεί να δείχνει αλλού ή σε τίποτα (nullptr).

Εναλλακτικά: πέρασμα με αναφορά, void sum_prod(int a, int b, int &s, int &p), χωρίς * και &.

ΔΥΝΑΜΙΚΗ ΜΝΗΜΗ

Με new δεσμεύουμε, με delete αποδεσμεύουμε

```
int *pi = new int;
double *pd = new double;
char *pc = new char;
*pi = 42;
*pd = 3.14;
*pc = 'Z';
cout << *pi << " " << *pd << " " << *pc;
delete pi;
delete pd;
delete pc;
```

Εμφανίζει **42 3.14 Z**

new Τύπος

Δεσμεύει μνήμη στον σωρό (heap) και επιστρέφει τη διεύθυνσή της.

delete δείκτης

Ελευθερώνει τη μνήμη. Χωρίς delete έχουμε διαρροή (memory leak).

new T[n] και delete[]

Για πίνακες η αποδέσμευση γίνεται με delete[] p και όχι με delete p.

Κανόνας: κάθε new χρειάζεται ακριβώς ένα delete, κάθε new[] ένα delete[].

ΔΕΙΚΤΕΣ · ΠΑΓΙΔΕΣ

Μετά το delete, ο δείκτης είναι "ξεκρέμαστος" (dangling pointer)

Λάθος: χρήση μετά το delete

```
int *p = new int(7);
int *q = p;
delete p;
cout << *q;  // σφάλμα
*p = 5;      // σφάλμα
delete q;    // σφάλμα
```

Ο q δείχνει σε μνήμη που ήδη ελευθερώθηκε: απροσδιόριστη συμπεριφορά, ακόμη κι αν το πρόγραμμα μεταγλωττίζεται.

Σωστά: delete και nullptr

```
int *p = new int(7);
int *q = p;
cout << *q << endl;
delete p;
p = nullptr;
q = nullptr;
```

Εμφανίζει **7**

Ο nullptr κάνει το λάθος ορατό: η χρήση του δίνει άμεσα σφάλμα αντί για τυχαία δεδομένα.

Θυμηθείτε: δύο δείκτες στο ίδιο αντικείμενο, ένα delete. Και ο δεύτερος δείκτης είναι ξεκρέμαστος.

ΠΙΝΑΚΕΣ · ΑΣΚΗΣΗ

Αντιγραφή ενός πίνακα x φορές

```
int a[5] = {2, 7, 1, 3, 6};
int x;
cin >> x;
int *d = new int[5 * x];
for (int k = 0; k < x; k++)
    for (int i = 0; i < 5; i++)
        d[k * 5 + i] = a[i];
for (int i = 0; i < 5 * x; i++)
    cout << d[i] << " ";
delete[] d;
```

Για x = 2 **2 7 1 3 6 2 7 1 3 6**

Στατικός πίνακας

Ο int a[5] έχει μέγεθος γνωστό στη μεταγλώττιση και ζει στη στοίβα.

Δυναμικός πίνακας

Ο new int[5 * x] παίρνει μέγεθος κατά την εκτέλεση και ζει στον σωρό.

delete[] d

Αποδεσμεύουμε πριν τον τερματισμό. Και για τους δύο: a[i] ≡ *(a + i).

Προσοχή: το στοιχείο i του αντιγράφου k βρίσκεται στη θέση k * 5 + i.

ΠΙΝΑΚΕΣ · ΙΧΝΗΛΑΤΗΣΗ

Ο δυναμικός πίνακας γεμίζει με x αντίγραφα



k = 0, τα a[0..4] στις θέσεις 0 έως 4

k = 1, τα a[0..4] στις θέσεις 5 έως 9

Έκφραση	Τιμή	Σχόλιο
`x`	`2`	πλήθος αντιγράφων που δίνει ο χρήστης
`new int[5 * x]`	`10 θέσεις`	40 byte στον σωρό (4 byte ο int)
`d[k * 5 + i]`	`a[i]`	το αντίγραφο k ξεκινά στη θέση 5k
`delete[] d`	—	ελευθερώνει και τις 10 θέσεις

Εμφανίζει **2 7 1 3 6 2 7 1 3 6**

ΠΟΛΥΔΙΑΣΤΑΤΟΙ ΠΙΝΑΚΕΣ

Πίνακας 5×5 με άθροισμα δεικτών

```
int a[5][5];
for (int i = 0; i < 5; i++)
    for (int j = 0; j < 5; j++)
        a[i][j] = i + j;
for (int i = 0; i < 5; i++) {
    for (int j = 0; j < 5; j++)
        cout << a[i][j] << " ";
    cout << endl;
}
```

Στη μνήμη γραμμή-γραμμή

Τα 25 στοιχεία είναι διαδοχικά: πρώτα η γραμμή 0, μετά η 1, και ούτω καθεξής.

i \ j	0	1	2	3	4
0	0	1	2	3	4
1	1	2	3	4	5
2	2	3	4	**5**	6
3	3	4	5	6	7
4	4	5	6	7	8

a[2][3] = 2 + 3 = 5

ΠΟΛΥΔΙΑΣΤΑΤΟΙ ΠΙΝΑΚΕΣ · ΣΥΝΑΡΤΗΣΕΙΣ

Δυναμικός δισδιάστατος πίνακας

```
void print(int **m, int r, int c) {
    for (int i = 0; i < r; i++) {
        for (int j = 0; j < c; j++)
            cout << m[i][j] << " ";
        cout << endl;
    }
}
```

Σειρά αποδέσμευσης

Πρώτα delete[] m[i] για κάθε γραμμή, μετά delete[] m για τον πίνακα των δεικτών.

```
int r = 2, c = 3;
int **m = new int *[r];
for (int i = 0; i < r; i++) {
    m[i] = new int[c];
    for (int j = 0; j < c; j++)
        m[i][j] = i * c + j;
}
print(m, r, c);
for (int i = 0; i < r; i++)
    delete[] m[i];
delete[] m;
```

Εμφανίζει `0 1 2` `3 4 5`

Δύο επίπεδα: ο m δείχνει σε r δείκτες και κάθε m[i] δείχνει σε c ακέραιους.

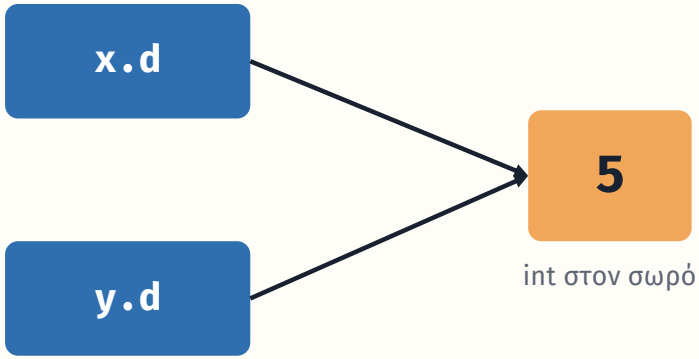
ΚΑΤΑΣΚΕΥΑΣΤΗΣ ΑΝΤΙΓΡΑΦΗΣ

Το προκαθορισμένο αντίγραφο μοιράζει τη μνήμη

```
class A {
    int *d;
public:
    A(int v) { d = new int(v); }
    ~A() { delete d; }
};
int main() {
    A x(5);
    A y = x;   // ίδιος δείκτης
}
```

Στο τέλος της main

Ο ~A() τρέχει για τον y και για τον x: δύο delete στην ίδια διεύθυνση, σφάλμα.



Shallow copy: αντιγράφεται η διεύθυνση, όχι η τιμή.

ΚΑΤΑΣΚΕΥΑΣΤΗΣ ΑΝΤΙΓΡΑΦΗΣ · DEEP COPY

Το deep copy δεσμεύει νέα μνήμη

```
class A {
    int *d;
    int n;
public:
    A(int n) : d(new int[n]), n(n) {
        for (int i = 0; i < n; i++)
            d[i] = i;
    }
    A(const A &o) : d(new int[o.n]), n(o.n) {
        for (int i = 0; i < n; i++)
            d[i] = o.d[i];
    }
    ~A() { delete[] d; }
};
```

```
A x(3);
A y = x;  // κατασκευαστής αντιγραφής
```

Πότε καλείται

Στα A y = x; και A y(x); και στο πέρασμα ή την επιστροφή με τιμή.

Deep copy

Νέα μνήμη για τον y και αντιγραφή των τιμών: οι x.d και y.d διαφέρουν.

Κανόνας των τριών

Με ~A() που κάνει delete[], γράφουμε και κατασκευαστή αντιγραφής.

ΣΥΝΟΨΗ

Επανάληψη

Θέμα	Τι κρατάμε	Παράδειγμα
Υπερφόρτωση συναρτήσεων	Ίδιο όνομα, άλλες παράμετροι	`fun(int), fun(double)`
Υπερφόρτωση τελεστών	Δικός μας ορισμός για κλάσεις	`a < b, cout << a`
Δείκτες	Διεύθυνση και πέρασμα με δείκτη	`*p, &x`
new και delete	Δυναμική μνήμη, ένα delete ανά new	`int *p = new int;`
Πίνακες	Στατικοί στη στοίβα, δυναμικοί στον σωρό	`new int[n], delete[] d`
Πολυδιάστατοι πίνακες	Γραμμές και στήλες, int ** για δυναμικούς	`a[i][j]`
Κατασκευαστής αντιγραφής	Deep copy όταν υπάρχει δείκτης	`A(const A &o)`

`new / delete`

Ένα delete για κάθε new και ένα delete[] για κάθε new[].

`nullptr`

Μετά το delete μηδενίζουμε τον δείκτη για να μη κρεμάει.

`A(const A &o)`

Χρειάζεται όταν η κλάση κρατά δείκτη σε δική της μνήμη.

Στο εργαστήριο: λύστε τις ασκήσεις ξεκινώντας από τα function_overload.cpp, overload1.cpp, dynamic_memory2.cpp, two_dimensional_array1.cpp και deep_copy1.cpp.