

# Κλάσεις στη C++

# Συναρτήσεις πρόσβασης και μεταλλαγής

Πολλές φορές γράφοντας κώδικα χρειάζεται να έχουμε πρόσβαση στα δεδομένα της κλάσης και είτε να δοθεί κάποια τιμή ή να χρησιμοποιηθεί κάποια τιμή. Θα χρειαζόμασταν κάποιες set/get συναρτήσεις

Κανονικά θα γράφατε μια συνάρτηση μέλος της κλάσης που έχει πρόσβαση στα private δεδομένα της κλάσης

Θέλετε επίσης να δώσετε μια λειτουργικότητα στη κλάση σας έστω πολ/σμός δύο λόγων

Πρέπει να είμαστε προσεκτικοί στη χρήση αυτών των συναρτήσεων γιατί δημιουργούν το μέσο πρόσβασης και αλλαγής των δεδομένων μελών της κλάσης

Έστω ότι ορίζουμε μια συνάρτηση μή μέλος της κλάσης για να κάνουμε τα παραπάνω

```
//fraction.cpp
```

```
...
```

```
Fraction mult_fracs(const Fraction & lhs, const Fraction & rhs)
```

```
{
```

```
    Fraction temp;
```

```
    temp.m_Numerator = lhs.m_Numerator * rhs.m_Numerator;
```

```
    ...
```

Αυτή δεν θα γίνει compiled αν γράφαμε απλά `f3=f1*f2` γιατί ο τελεστής «\*» δεν έχει οριστεί για την κλάση που έχουμε και ισχύει μόνο για πολ/σμούς με την κλάση αριθμών.

Για να αποφύγουμε το πρόβλημα, ορίζουμε το αντικείμενο `temp` τύπου `Fraction` εφόσον θέλουμε να επιστρέψουμε κάτι που είναι τύπου `Fraction`.

Θα χτίσουμε το αντικείμενο αυτό αναθέτοντας στη τιμή της μιας μεταβλητής του το γινόμενο των `numerator` του `lhs` και `rhs` αντικειμένου

**Όπως το γράψαμε ΔΕΝ δουλεύει γιατί η `mult_fracs` δεν είναι συνάρτηση μέλος και δεν μπορούμε να έχουμε πρόσβαση στα δεδομένα**

← Προσοχή ότι δίνουμε τις μεταβλητές με αναφορά και τις θεωρούμε σταθερές (`const`) ώστε να μην αλλάξουν με κάποιον τρόπο

# Συναρτήσεις πρόσβασης και μεταλλαγής

Για να αποφύγουμε το πρόβλημα προσθέτουμε κάποιες συναρτήσεις πρόσβασης στην κλάση

```
//fraction.h
...
class Fraction
{
public:
    void readin();
    void print();
    Fraction reciprocal();
    void unreduce(const int m);
    int getNum(); } πρόσβαση
    int getDen(); }
    void setNumer(const int numer); } μεταλλαγή
    bool setDenom(const int denom); }
private:
    int m_Numerator;
    int m_Denominator;
};
```

```
//fraction.cpp
...
int Fraction::getNum()      επιστροφή του
{                            numerator του
    return m_Numerator;     αντικειμένου που
                            καλεί τη συνάρτηση
}

int Fraction::getDen()
{
    return m_Denominator;
}

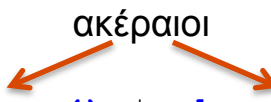
void Fraction::setNumer(const int numer)
{
    m_Numerator = numer;
    return;
}

bool Fraction::setDenom(const int denom){
    bool set = false;
    if(denom != 0){
        set = true;
        m_Denominator = denom; }
    return set;
}
```

# Συναρτήσεις πρόσβασης/μεταλλαγής - Προβλήματα

```
// στο file ορισμού των συναρτήσεων της Fractions
Fraction mult_fracs(const Fraction & lhs, const Fraction & rhs)
{
    Fraction temp;
    temp.setNumer(lhs.getNumer() * rhs.getNumer());
    if (!(temp.setDenom(lhs.getDenom() * rhs.getDenom())))
    {
        cout<<"ERROR: denominator is 0 "<<endl;
        exit(2);
    }
    return temp;
}
```

ακέρατοι



# Σταθερές συναρτήσεις μέλη - Const

Όταν μια συνάρτηση υποτίθεται ότι δεν πρέπει να αλλάξει τα δεδομένα μέλη του καλούντος αντικειμένου, είναι καλή πρακτική να δηλώνεται ως Const

```
//fraction.h
...
class Fraction
{
public:
    void readin();
    void print() const;
    Fraction reciprocal() const;
    void unreduce(const int m);
    int getNum() const;
    int getDen() const;
    void setNumer(const int numer);
    bool setDenom(const int denom);
private:
    int m_Numerator;
    int m_Denominator;
};
```

```
void Fraction::print() const
{
    cout<<"("<<m_Numerator
        <<"/"<<m_Denominator<<")";
    return;
}

Fraction Fraction::reciprocal() const
{
    Fraction returnable;
    returnable.m_Numerator =
m_Denominator;
    returnable.m_Denominator =
m_Numerator;
    return returnable;}

int Fraction::getNum() const
{
    return m_Numerator;}

int Fraction::getDen() const
{
    return m_Denominator;}
```

Πρέπει να τονιστεί η διαφορετική χρήση της επιλογής const (global ή συνάρτησης μέλους)  
Const πρέπει να εισαχθεί στον ορισμό της συνάρτησης

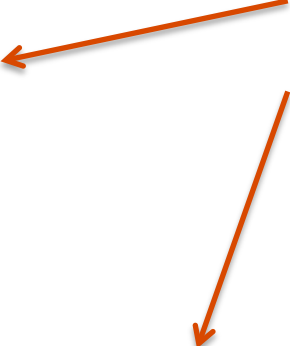
# Φιλική συνάρτηση – **friend** function μιας κλάσης

Είδαμε ότι όταν χρειάστηκε να έχουμε πρόσβαση στα δεδομένα μέλη της κλάσης, χρειάστηκε να δημιουργήσουμε συναρτήσεις πρόσβασης και μεταλλαγής

```
//fraction.h
...
class Fraction
{
    ...
private:
    int m_Numerator;
    int m_Denominator;
};
```

```
//fraction.cpp
...
Fraction mult_frac(const Fraction & lhs, const Fraction & rhs)
{
    Fraction temp;
    temp.m_Numerator = lhs.m_Numerator * rhs.m_Numerator;
    ...
}
```

Ως έχει δεν μπορεί να γίνει compiled γιατί η συνάρτηση `mult_frac` προσπαθεί να αποκτήσει πρόσβαση στις τιμές των δεδομένων της κλάσης, κάτι που απαγορεύεται.



# friend function

Είναι μια μη μέλος συνάρτηση μιας κλάσης στην οποία έχει δοθεί πρόσβαση στα δεδομένα της κλάσης

Το πρόθεμα friend χρησιμοποιείται πάντοτε μέσα στον ορισμό της κλάσης που δίνει δικαιώματα πρόσβασης στην συνάρτηση

Το πρόθεμα friend δεν πρέπει να χρησιμοποιείται ποτέ έξω από τον ορισμό της κλάσης

```
//fraction.h
```

```
...
```

```
class Fraction
```

```
{
```

```
...
```

```
friend Fraction mult_fracs(const Fraction & lhs, const Fraction & rhs);
```

```
private:
```

```
int m_Numerator;
```

```
int m_Denominator;
```

```
};
```

```
//fraction.cpp
```

```
...
```

```
Fraction mult_fracs(const Fraction & lhs, const Fraction & rhs)
```

```
{
```

```
Fraction temp;
```

```
temp.m_Numerator = lhs.m_Numerator * rhs.m_Numerator;
```

```
temp.m_Denominator = lhs.m_Denominator * rhs.m_Denominator;
```

```
return temp; }
```

**Δυο σημεία προσοχής:**

- Η συνάρτηση δεν είναι μέλος της κλάσης Fraction και επομένως δεν δηλώνεται με την χρήση του τελεστή «::»
- Δεν χρησιμοποιείται το πρόθεμα friend στον ορισμό της

# Συνάρτηση κατασκευής - Constructor

Constructors είναι ιδιαίτερες συναρτήσεις που χρησιμοποιούνται για την αρχικοποίηση, initialization, στοιχείων που αποτελούν μέλη ενός αντικειμένου συγκεκριμένης κλάσης

Constructors έχουν πράγματι ιδιαιτερότητες όπως:

- Το όνομά τους είναι πάντοτε το ίδιο με το όνομα της κλάσης
- Δεν έχουν τύπο επιστροφής αποτελέσματος (δεν είναι όμως void) και άρα δεν υπάρχει return
- Καλούνται αυτόματα από τον compiler και πολύ σπάνια από τον χρήστη
- Αν δεν δοθεί constructor ο compiler δίνει αυτόματα constructor. Η διαδικασία αναιρείται μόλις γραφεί κάποιος constructor

Ο compiler θα αναζητήσει τους constructors και θα χρησιμοποιήσει αυτόν που είναι κατάλληλος για τον τρόπο που δηλώθηκε η κλάση

Αν δεν δόθηκε από τον προγραμματιστή ένας constructor, τότε ο compiler θα κατασκευάσει τον εξ'ορισμού και θα τον εφαρμόσει αν συνάδει

Αν είναι η μοναδική επιλογή και δεν συνάδει τότε ο compiler θα επιστρέψει σφάλμα

Ο compiler θα δώσει τον constructor εξ'ορισμού για όλες τις κλάσεις.

Ο default constructor θα καλέσει τους defaults constructors για κάθε δεδομένο μέλος της κλάσης

Αν τα δεδομένα μέλη είναι κοινές μεταβλητές τότε θα πάρετε τη θέση μνήμης με ότι τιμή τυγχάνει να υπάρχει στη διεύθυνση αυτή

Αυτό σημαίνει ότι πρέπει να γράφετε τους δικούς σας constructors

Όταν δοθεί ο constructor, τότε ο compiler δεν θα δώσει το δικό το



# Constructors

Δηλώνοντας ένα αντικείμενο τύπου της κλάσης που ορίσαμε **Fraction f**;  
 μας δίνεται ένα αντικείμενο εξ'ορισμού. Αν τυπώσουμε το αντικείμενο τότε θα πάρουμε μια τιμή  
 3.5939E-34/123212 που είναι τιμές που υπάρχουν στις θέσεις μνήμης των μεταβλητών μελών  
 της κλάσης

Μπορούμε όμως να δηλώσουμε και κατόπιν να ορίσουμε τον constructor:

```
#include "fraction.h"

int main()
{
    float x;
    float y = 6.7;
    float z(7.2);
    Fraction f;
    Fraction g(4, 5);
    ...
}
```

Δεν μπορώ να το κάνω  
γιατί δεν έχω ορίσει  
constructor

```
class Fraction
{
public:
    Fraction(const int num, const int den);
    void readin();
    void print();
    ...
};

Fraction::Fraction(const int num, const int den)
{
    m_Numerator = num;
    m_Denominator = den;
}
```

δεν υπάρχει  
τύπος επιστροφής

δήλωση του  
constructor

ορισμός του  
constructor

default constructor  
Κρατά τα απαραίτητα  
στοιχεία μνήμης αλλά  
δεν υπάρχουν αρχικές  
τιμές

δεν υπάρχει  
return

# Constructors

Από τη στιγμή που δηλώσουμε/ορίσουμε τον constructor τότε θα έχουμε:

```
class Fraction
```

```
{
    public:
        Fraction(const int num,
                const int den);
        void readin();
        void print();
        ...
};
```

```
Fraction::Fraction(const int num,
                    const int den)
```

```
{
    m_Numerator = num;
    m_Denominator = den;
}
```

```
Fraction::Fraction(const int num, const int den)
```

```
{
    setNum(num);
    setDen(den);
}
```

```
#include "fraction.h"
```

```
int main()
{
```

```
    float x;
```

```
    float y = 6.7;
```

```
    float z(7.2);
```

```
    Fraction f;
```

```
    Fraction g(4, 5);
```

```
    Fraction h(4, 0)
```

```
    ...
```

Δεν μπορώ να το κάνω  
γιατί δεν έχω ορίσει  
Constructor αυτού του είδους  
Αυτός που όρισα είναι  
διαφορετικός (2 ορίσματα)

Δημιουργώ το αντικείμενο g  
με τιμή 4/5

Δημιουργώ το αντικείμενο h  
με τιμή 4 και 0!! για παρονομαστή  
Δυστυχώς δεν έχουμε έλεγχο για  
τη τιμή.

Γράφουμε καλύτερο constructor

# Καλύτερη αρχικοποίηση constructor με λίστα

```
// main.cpp
#include "fraction.h"

int main()
{
    float x;
    float y = 6.7;
    float z(7.2);
    Fraction f;
    Fraction g(4, 0);
    ...
}
```

τρόπος αρχικοποίησης ακριβώς με την σειρά προτεραιότητας.  
ανάθεση αρχικής τιμής μόνο με το α(3) και όχι τον τελεστή ανάθεση (=).

```
// fraction.cpp
Fraction::Fraction(const int num, const int den) : m_Numerator(num), m_Denominator(den) {}
```

// Σημειώστε: πρέπει να έχετε το κύριο μέρος της δομής μιας συνάρτησης  
// σε μια λίστα αρχικοποίησης ακόμα και αν είναι άδειο

```
Fraction::Fraction(const int num, const int den) : m_Numerator(num), m_Denominator(den)
{
    if(m_Denominator == 0)
    {
        cout << "error: 0 passed in as denominator" << endl;
        exit(1);
    }
}
```

Το κύριο μέρος της συνάρτησης που διορθώνει το πρόβλημα με το 0 για αρχική τιμή στο παρονομαστή

# Constructor – default constructor

```
#include "fraction.h"
```

```
int main()
```

```
{
```

```
    float x;
```

```
    float y = 6.7;
```

```
    float z(7.2);
```

```
    Fraction f; ← Προβληματικό γιατί ο constructor πρέπει να έχει 2 ορίσματα
```

```
    Fraction g(4, 5); και ο compiler δεν προσφέρει τον default constructor
```

```
    ...
```

```
// fraction.h
```

```
...
```

```
class Fraction
```

```
{
```

```
    Fraction();    // default constructor
```

```
    ...
```

```
// fraction.cpp
```

```
Fraction::Fraction() : m_Numerator(0), m_Denominator(1) {} // defaults to 0/1
```

# Copy constructor

```
#include "fraction.h"

int main()
{
    float x;
    float y = 6.7;
    float z(7.2);
    Fraction f;
    Fraction g(4, 5);
    Fraction h(g); // δημιουργεί ένα fraction που είναι αντίγραφο ενός άλλου
    ...

// fraction.h
...
class Fraction
{
    Fraction(const Fraction & source);
    ...
}
```

Η λίστα του copy constructor είναι πάντα ίδια

Μια const

Το είδος του αντικειμένου που θα αντιγραφεί

Η αναφορά στο αντικείμενο γιατί ουσιαστικά αν περάσουμε το αντικείμενο με τιμή τότε θα πρέπει να δημιουργηθεί αντίγραφο του αντικειμένου. Τότε θα κληθεί και πάλι ο copy constructor και καταλήγουμε σε ατέρμονη κλήση του copy constructor

# Copy constructor

Το προηγούμενο μπορεί να γίνει ως εξής τελικά:

```
// main.cpp
```

```
...
```

```
int main()
```

```
{
```

```
    Fraction g(4, 5);
```

```
    Fraction h(g)
```

```
    ...
```

```
// fraction.h
```

```
...
```

```
class Fraction
```

```
{
```

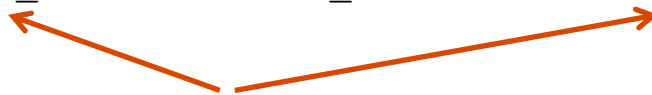
```
    Fraction(const Fraction & source);
```

```
    ...
```

```
// fraction.cpp
```

```
Fraction::Fraction(const Fraction & source) :
```

```
    m_Numerator(source.m_Numerator), m_Denominator(source.m_Denominator) {}
```



τοποθετούμε σαν αρχικοποιημένες τιμές τις τιμές του αντικειμένου source

# Κακή πρακτική – ζητάτε προβλήματα

```
#include "fraction.h"
```

```
int main()
```

```
{
```

```
    fraction f(); ← ποτέ μη το κάνετε αυτό σε αρχικοποίηση
```

```
    ....
```

Ο compiler πιστεύει ότι είναι μια συνάρτηση με μηδενικό όρισμα

Δεν θα πάρετε λάθος στο compilation αλλά όταν θα κληθεί ο compiler θα νομίζει ότι μια συνάρτηση πρέπει να καλέσει μια συνάρτηση που είναι μέλος μιας κλάσης συναρτήσεων  
Κάτι τέτοιο δεν ισχύει

# Σύγκριση struct και class

Όπως έχετε διαπιστώσει μέχρι τώρα η κλάση είναι παρόμοια της struct

Μια struct μπορεί να έχει private και public τμήματα, συναρτήσεις μέλη, μεταβλητές μέλη,, constructors και destructors ακριβώς όπως οι κλάσεις

Η κύρια διαφορά τους είναι το γεγονός ότι στην struct εξορισμού όλα τα στοιχεία είναι public ενώ στην class είναι private

```
struct boot
{
    float m_size;
    float m_heelheight;
};
```

```
struct boot
{
    public:
        float m_size;
        float m_heelheight;
};
```

```
class boot
{
    public:
        float m_size;
        float m_heelheight;
};
```

```
class table
{
    private:
        int m_numLegs;
        float m_topArea;
    public:
        void fold();
};
```

```
class table
{
    int m_numLegs;
    float m_topArea;
    public:
        void fold();
};
```

```
struct table
{
    void fold();
    private:
        int m_numLegs;
        float m_topArea;
};
```