# Σύνοψη-ορισμένα βασικά σημεία της ύλης

- Έμφαση στα ιδιαίτερα χαρακτηριστικά της C++
- Πέρασμα με τιμή/αναφορά
- Διαχείριση μνήμης – object allocation – stack vs heap
- Κληρονομικότητα
  - Σειρά κλήσης constructors, destructors
  - Κλάσης βάσης και παραγόμενων
  - Initialization list, κλήση constructor
- Friend κλάσεις, συναρτήσεις
- Δημιουργία με new – delete
- Πολυμορφισμός
  - Virtual functions, vtables
  - Object slicing
  - Virtual destructor
- Δημιουργία templates
- Operator overload
- Δημιουργία, έγερση και χειρισμός εξαιρέσεων
- Χειρισμός βασικών δομών (array, string, vector)
- Ροές, αρχεία



# 7.2 const (Σταθερά) Αντικείμενα και Μέθοδοι

- Αρχικοποίηση αντικειμένου
  - Αρχικοποίηση με member initializer syntax
    - Μπορεί να χρησιμοποιηθεί
      - Με όλα τα μέλη δεδομένων
    - (Πρέπει να χρησιμοποιηθεί)
      - `Για τα μέλη const`
      - Για όλες τις αναφορές μεταβλητών
  - C++11:
    - **Μπορεί πλέον να αρχικοποιηθεί κανονικά** μέσα στην κλάση

# 7.4 friend Συναρτήσεις και friend Τάξεις

- **`friend`** συναρτήσεις
  - Ορίζονται εκτός εμβέλειας της τάξης
  - Έχουν πρόσβαση σε non-public members
- Δήλωση **`friend`**s
  - Συνάρτηση
    - Προηγείται το keyword **`friend`**
  - Όλες οι συναρτήσεις της τάξης **`classTwo`** ως **`friend`**s της τάξης **`classOne`**
    - Βάζουμε τη δήλωση της μορφής

      **`friend class classTwo;`**

      στον ορισμό της **`classOne`**





# 7.6 Διαχείριση Δυναμικής Μνήμης με χρήση new και delete

- Έστω

  ```
  Time *timePtr;
  timePtr = new Time;
  ```

- **`Τελεστής new`**
  - Δημιουργεί αντικείμενα κατάλληλου μεγέθους για τον τύπο **`Time`**
    - Δίνει λάθος αν δεν υπάρχει χώρος στη μνήμη
  - Επιστρέφει δείκτη στον συγκεκριμένο τύπο
- **Stack vs Heap**
- Με αρχικοποίηση

  ```
  double *ptr = new double( 3.14159 );
  Time *timePtr = new Time( 12, 0, 0 );
  ```

- Δήλωση πίνακα

  ```
  int *gradesArray = new int[ 10 ];
  ```

# 7.6 Διαχείριση Δυναμικής Μνήμης με χρήση new και delete

- Απελευθερώνει τη μνήμη και καταστρέφει τα αντικείμενα
- Έστω
  ```
  delete timePtr;
  ```
- Τελεστής **delete**
  - Καλεί το destructor
  - Η μνήμη μπορεί να χρησιμοποιηθεί με άλλα αντικείμενα
- Deallocating arrays
  ```
  delete [] gradesArray;
  ```
  - Απελευθερώνει το array στο οποίο δείχνει το **gradesArray**
  - Αν είναι δείκτης σε array αντικειμένων
    - Καλείται πρώτα ο destructor για κάθε αντικείμενο του array
    - Μετά απελευθερώνει τη μνήμη





# Constructors και Destructors στις παραγόμενες κλάσεις

- Δημιουργία αντικειμένου παραγόμενης κλάσης
  - Σειρά κλήσης των constructors
    - Ο constructor της παραγόμενης κλάσης **καλεί** τον constructor της κλάσης βάσης
      - **Έμμεσα** (ο default, που δεν έχει *ή μπορεί να κληθεί χωρίς*, ορίσματα)
      - Άμεσα στη **member initialization list**
    - Δεν υπάρχει keyword `super`!
    - Βάση της ιεραρχίας κληρονομικότητας
      - Ο τελευταίος στη σειρά constructor που καλείται και ο πρώτος που ολοκληρώνει την εκτέλεσή του
      - Παράδειγμα: ιεραρχία **Point3/Circle4/Cylinder**
        - **Point3** constructor, καλείται τελευταίος και ολοκληρώνει την εκτέλεσή του πρώτος
    - Αρχικοποίηση των πεδίων
      - Ο constructor κάθε κλάσης βάσης αρχικοποιεί τα πεδία του

# Constructors και Destructors στις παραγόμενες κλάσεις

- Καταστροφή αντικειμένου παραγόμενης κλάσης
  - Σειρά κλήσης των destructors
    - Αντίστροφη σειρά από αυτή της κλήσης των constructors
    - Ο destructor της παραγόμενης κλάσης καλείται πρώτος
    - Ο destructor της επόμενης κλάσης βάσης στην ιεραρχία καλείται στη συνέχεια
    - Συνεχίζουμε προς τα πάνω μέχρι να φθάσουμε στην τελευταία κλάση βάσης της ιεραρχίας
      - Μετά την εκτέλεση του τελευταίου destructor, το αντικείμενο αφαιρείται από τη μνήμη



# Σχέσεις αντικειμένων σε μια ιεραρχία κληρονομικότητας

- Σημείο κλειδί
  - Ένα αντικείμενο μιας παραγόμενης κλάσης μπορεί να το μεταχειριστούμε και ως αντικείμενο της κλάσης βάσης
    - "IS-A" σχέση
    - Όμως, το αντίστροφο δεν ισχύει: ένα αντικείμενο της κλάσης βάσης δεν είναι αντικείμενο της παραγόμενης κλάσης
- Εκχώρηση αντικειμένου παραγόμενης κλάσης σε αντικείμενο τύπου κλάσης βάσης (με τιμή)
  - Object Slicing!

```
class Pet
{
 public:
    string name;
};
class Dog : public Pet
{
public:
    string breed;
};

int main()
{
    Dog dog;
    Pet pet;

    dog.name = "Tommy";
    dog.breed = "Kangal Dog";
    pet = dog;
    cout << pet.breed; //ERROR
```

# Εικονικές Συναρτήσεις (Virtual Functions)

- Πολυμορφισμός
  - Το ίδιο μήνυμα, "print", αποστέλλεται σε πολλά αντικείμενα
    - Τα αντικείμενα προσπελάζονται μέσω δείκτη της κλάσης βάσης
  - Το μήνυμα παίρνει "πολλές μορφές"
- Σύνοψη
  - Δείκτης κλάσης βάσης σε αντικείμενο της κλάσης βάσης, δείκτης παραγόμενης κλάσης σε αντικείμενο της παραγόμενης κλάσης
    - Απλό
  - Δείκτης κλάσης βάσης σε αντικείμενο της παραγόμενης κλάσης
    - Καλεί μόνο μεθόδους της κλάσης βάσης
    - Δεν ισχύει για τις μεταβλητές
  - Δείκτης παραγόμενης κλάσης σε αντικείμενο της κλάσης βάσης
    - Compiler error
    - Επιτρέπεται με downcasting



# Abstract Classes

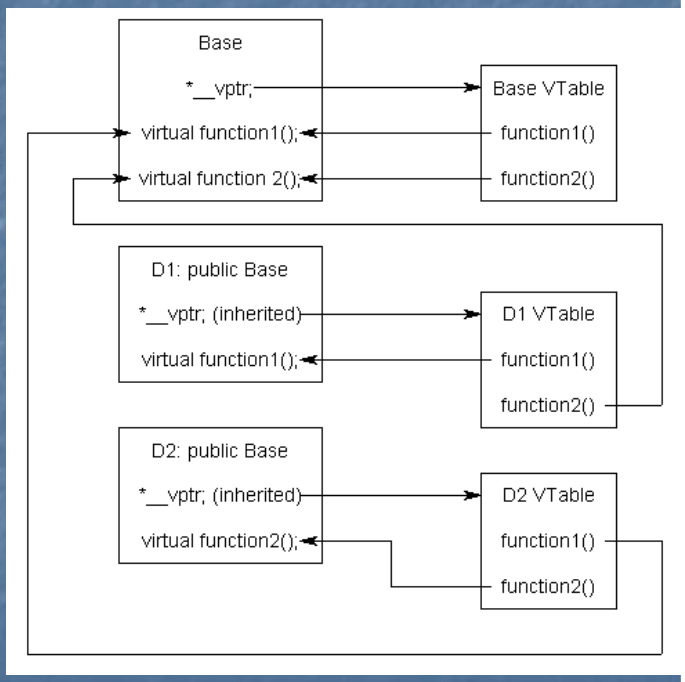
- Είναι χρήσιμες, όχι υποχρεωτικές
- Για να ορίσουμε μια abstract class
  - Θέλουμε μια ή περισσότερες "pure" virtual functions

```
virtual void draw() const = 0;
```

  - Regular virtual functions
    - Έχουν υλοποίηση, η υπερκάλυψη είναι προαιρετική
  - Pure virtual functions
    - Δεν έχουν υλοποίηση, η υπερκάλυψη είναι υποχρεωτική
  - Μια abstract class μπορεί να έχει δεδομένα και υλοποιημένες συναρτήσεις

## Πολυμορφισμός, Εικονικές Συναρτήσεις και Δυναμική Σύνδεση: Υπάρχει κόστος χρήσης?

- Ο πολυμορφισμός έχει overhead
  - Δεν χρησιμοποιείται π.χ. στην STL (Standard Template Library) για λόγους επίδοσης
- **`virtual`** function table (vtable)
  - Κάθε κλάση με μια **`virtual`** function έχει ένα vtable
  - Για κάθε **`virtual`** function, το vtable έχει δείκτη στην κατάλληλη συνάρτηση






# Virtual Destructors

- Δείκτης κλάσης βάσης σε αντικείμενο της παραγόμενης κλάσης
  - Εάν καταστρέψουμε το αντικείμενο με την **`delete`**, η συμπεριφορά είναι απροσδιόριστη
- Απλή λύση
  - Δηλώνουμε τον destructor της base-class ως virtual
  - Όταν καλείται η **`delete`** καλείται και ο κατάλληλος destructor
- Όταν καταστρέφουμε ένα αντικείμενο μιας παραγόμενης κλάσης
  - Πρώτα εκτελείται ο destructor της παραγόμενης κλάσης
  - Μετά εκτελείται ο destructor της base-class
- Οι constructors δεν μπορεί να είναι virtual

# Custom exceptions

```
#include <iostream>
#include <stdexcept>
using namespace std;
class DivideByZeroException : public out_of_range {
  public:
    DivideByZeroException(int n, string m)
              : numerator(n), out_of_range(m){}
    int numerator;
};

double divide( int numerator, int denominator ) {
  if ( denominator == 0 )
    throw DivideByZeroException(numerator,"attempted to divide with zero");
  return static_cast< double >( numerator ) / denominator;
}

int main(){
  int a,b;
  try{
    while ( cin >> a >> b )
      divide(a,b);
  }
  catch (out_of_range e){
    cout << e.what();
  }
}
```

Αν και δεν είναι υποχρεωτικό, εδώ φτιάχνουμε δικό μας τύπο εξαίρεσης, κληρονομώντας από τον υπάρχοντα τύπο out_of_range.
Μέσω του δημιουργού μπορούμε να ορίσουμε την πληροφορία που θα αποθηκεύουμε στην εξαίρεση μέσω των ορισμάτων του. (πχ εδώ μπορούμε να κρατάμε τον αριθμητή της πράξης).
Στον δημιουργό της κλάσης out_of_range μπορούμε να περνάμε μήνυμα σχετικό με το σφάλμα το οποία θα είναι προσπελάσιμο μέσω της μεθόδου what() που έχει η κλάση exception και παράγωγες της.

Με κλήση του δημιουργού, δημιουργούμε εξαίρεση του τύπου που φτιάξαμε και την πετάμε με την εντολή throw.

Στο catch κομμάτι χειριζόμαστε εξαιρέσεις τύπου out_of_range ή παράγωγες της ( όπως αυτή που φτιάξαμε ) . **Πέρασμα με αναφορά;**

```
catch (Exception &e){
    cout << e.what();
  }
```



# Υλοποίηση Στοίβας με Template Κλάση

```
template< class T >
class Stack {
  public:
    Stack( int = 10 );
   ~Stack() {
     delete [] stackPtr;
    }
    bool push( const T& );
    bool pop( T& );
    bool isEmpty() const { return top == -1; }
    bool isFull() const { return (top == size - 1);
}
  private:
    int size;
    int top;
    T *stackPtr; };
```

```
template<  class T >
Stack< T >::Stack( int s ){
  size = s > 0 ? s : 10;
  top = -1;
  stackPtr = new T[ size ];
}


template< class T >
bool Stack< T >::push( const T &pushValue ){
  if ( !isFull() ) {
    stackPtr[ ++top ] = pushValue;
    return true;
  }
  return false;
}

template< class T >
bool Stack< T >::pop( T &popValue ){
  if ( !isEmpty() ) {
    popValue = stackPtr[ top-- ];
    return true;
   }
   return false;
}
```

# Υπερφόρτωση Τελεστή ως μέλος κλάσης ή ως Friend συνάρτηση

- Operator functions
  - Ως μέθοδος (μέλος της κλάσης)
    - Το αριστερό μέλος του τελεστή είναι το αντικείμενο για το οποίο καλείται (μπορούμε να χρησιμοποιήσουμε και τον τελεστή **`this`** για να αναφερθούμε σε αυτό). Προφανώς είναι ιδίου τύπου με την κλάση στην οποία ορίζουμε τον τελεστή.
    - Το δεξιά μέλος του τελεστή είναι όρισμα της συνάρτησης (οτιδήποτε τύπου θέλουμε)
  - Ως συνάρτηση (εκτός της κλάσης, όχι μέθοδος )
    - Πρέπει να δοθούν και οι δύο παράμετροι (πριν και μετά τον τελεστή) ως ορίσματα
    - Πρέπει να είναι friend συνάρτηση (προσθέτουμε σχετική δήλωση στην κλάση) για να προσπελάσει **`private`** ή **`protected`** μέλη
  - (), [], ->, = πρέπει να οριστούν ως μέλη της κλάσης
  - Καλούνται
    - Όταν το αριστερό μέλος του τελεστή (για binary τελεστές) είναι αντικείμενο της κλάσης
    - Το μοναδικό όρισμα (για unary τελεστές) είναι αντικείμενο της κλάσης