Εργαστήριο Υπολογιστικών Συστημάτων & Τεχνολογίας Λογισμικού - CSSE

Τμήμα Εφ. Πληροφορικής

C++: Σημειώσεις

ΠΑΝΕΠΙΣΤΗΜΙΟ ΜΑΚΕΔΟΝΙΑΣ
ΟΙΚΟΝΟΜΙΚΩΝ ΚΑΙ ΚΟΙΝΩΝΙΚΩΝ ΕΠΙΣΤΗΜΩΝ

ΘΕΣΣΑΛΟΝΙΚΗ 2005

Εμφάνιση μηνυμάτων στην οθόνη (σελ 53 - 95)

Ας δούμε ένα παράδειγμα προγράμματος C++

ex1.cpp

```
#include <iostream.h>
void main(void)
 {
   int age = 29;
   cout << "my age is " << age << " years old" << endl;
 }
```

Αναλύοντας το πρόγραμμα έχουμε τα παρακάτω

#include <iostream.h> ⇒ οδηγεί το μεταγλωττιστή (compiler) να βρει τα πρότυπα για τις συναρτήσεις που χρησιμοποιούνται. Το πρότυπο εδώ είναι το <iostream.h>.

void main(void) ⇒ όνομα της κύριας συνάρτησης

{ } ⇒ αγκύλες που ορίζουν το σώμα της συνάρτησης

int age ⇒ εντολή δήλωσης

age = 29 ⇒ εντολή ανάθεσης

cout << "Η ηλικία μου είναι " << age << " χρονών" << endl; ⇒ εντολή συνάρτησης.

Το παραπάνω πρόγραμμα ex1.cpp εμφανίζει στην οθόνη το εξής μήνυμα: «my age is 29 years old»

ex2.cpp

```
#include <iostream.h>
#include <iomanip.h>
void main(void)
 {
  cout << "This is line one\nThis is line two"<<endl;
  cout<<1001<< endl;
  cout << "My favorite number is" << 1001<< endl;
  cout << "My favorite number is" << setw(4) << 1001 << endl;
  cout << "My favorite number is" << setw(6) << 1001 << endl;
 }
```

Όταν εκτελείται το ex2.cpp εμφανίζονται στην οθόνη τα παρακάτω

```
This is line one
This is line two
```

```
1001
My favorite number is1001
My favorite number is1001
My favorite number is  1001
```

Η συνάρτηση « cout » εμφανίζει στην οθόνη μηνύματα και χρησιμοποιεί ως πρότυπο το <iostream.h>.

Μέσα στα " " εισάγουμε τα αλφαριθμητικά που θέλουμε να εμφανίσουμε.

Η συνάρτηση setw( ) ορίζει το πεδίο εμφάνισης και χρησιμοποιεί ως πρότυπο το <iomanip.h>

Ο χαρακτήρας '\n' αλλάζει γραμμή. Το ίδιο κάνει και το 'endl'.

Χαρακτήρες που ελέγχουν τη θέση του κέρσορα

Χαρακτήρας	Σκοπός
\b	Πηγαίνει τον κέρσορα ένα διάστημα πίσω
\n	Νέα γραμμή
\t	Μετακινεί τον κέρσορα ένα διάστημα οριζόντια
\v	Μετακινεί τον κέρσορα ένα διάστημα κάθετα
Endl	Αλλάζει γραμμή

Αποθήκευση πληροφοριών σε μεταβλητές (σελ 53 - 95)

Στη C++ χρησιμοποιείται η ακόλουθη μορφή:

Τύπος_Μεταβλητής Όνομα_Μεταβλητής

Οι μεταβλητές έχουν την παρακάτω μορφή:

Τύπος	**Τιμές που αποθηκεύονται**
char	-128 έως 127, γράμματα του αλφαβήτου
int	-32768 έως 32767, ακέραιοι αριθμοί
unsigned	0 έως 65535
long	-2147483648 έως 2147483647, φυσικοί αριθμοί
float	$-3.4*10^{38}$ έως $3.4*10^{38}$
double	$-1.7*10^{308}$ έως $1.7*10^{308}$

ex3.cpp

```
#include <iostream.h>
void main(void)
 {
   int age = 32;
   float salary = 25000.75;
   long distance_to_the_moon = 238857;
   cout << "The employee is " << age << " years old" << endl;
   cout << "The employee makes $" << salary << endl;
   cout << "The moon is " << distance_to_the_moon << " miles from the earth" << endl;
 }
```

Στην οθόνη εμφανίζονται τα εξής:

```
The employee is 32 years old
The employee makes $25000.75
The moon is 238857 miles from the earth
```

Ανάγνωση δεδομένων από το πληκτρολόγιο (σελ 53 - 95)

Η συνάρτηση που χρησιμοποιείται είναι η παρακάτω:

cin >> μεταβλητή

ex4.cpp

```
#include <iostream.h>
void main(void)
 {
   char letter;  // Το γράμμα διαβάζεται από το πληκτρολόγιο
   int number;   // Ο αριθμός διαβάζεται από το πληκτρολόγιο
   long value;  //  Ο μεγάλος αριθμός διαβάζεται από το πληκτρολόγιο

   cout << "Type any character and press Enter: ";
   cin >> letter;
   cout << "The letter typed was " << letter << endl;

   cout << "Type your favorite number and press Enter: ";
   cin >> number;
   cout << "Your favorite number is " << number << endl;

   cout << "Type a large number and press Enter: ";
   cin >> value;
   cout << "The number you typed was " << value << endl;
 }
```

Στο παραπάνω πρόγραμμα ζητούνται ένα γράμμα, ένας αριθμός, και ένας μεγάλος αριθμός και στη συνέχεια εμφανίζονται στην οθόνη.

Παρατηρήσαμε επίσης τη γραμμή «// Ο αριθμός διαβάζεται από το πληκτρολόγιο ».

Στην πραγματικότητα είναι σχόλιο που εισάγουμε στον κώδικα ώστε να θυμόμαστε τι εντολές έχουμε χρησιμοποιήσει. Πριν από σχόλιο χρησιμοποιούμε τις δύο καθέτους (//).

Εκτέλεση απλών αριθμητικών πράξεων (σελ 53 - 95)

Η C++ χρησιμοποιεί τα παρακάτω σύμβολα για αριθμητικές πράξεις

*	Πρόσθεση
-	Αφαίρεση
*	Πολλαπλασιασμός
/	Διαίρεση

Το επόμενο πρόγραμμα δείχνει μια σειρά απλών πράξεων

ex5.cpp

```
#include <iostream.h>
void main(void)
 {
   float cost = 15.50;         // Το κόστος ενός πράγματος
   float sales_tax = 0.06;     // Ο φόρος είναι 6 τοις εκατό
   float amount_paid = 20.00;  // Το ποσό που πληρώνει ο αγοραστής
   float tax, change, total;   // Φόρος, επιστρεφόμενο ποσό και τελικό πληρωτέο ποσό

   tax = cost * sales_tax;
   total = cost + tax;
   change = amount_paid - total;

   cout << "Item Cost: $" << cost << "\tTax: $" << tax << "\tTotal: $" << total << endl;
   cout << "Customer change: $" << change << endl;
 }
```

Λήψη αποφάσεων από το πρόγραμμα (σελ 99 – 150)

Η C++ χρησιμοποιεί τα παρακάτω σύμβολα για να συγκρίνει μεταβλητές

Σύμβολο	Έλεγχος	Παράδειγμα
==	Δύο μεταβλητές είναι ίσες	Score == 100
!=	Δύο μεταβλητές δεν είναι ίσες	Old != New
>	Η πρώτη μεταβλητή μεγαλύτερη της δεύτερης	Cost>50
<	Η πρώτη μεταβλητή μικρότερη της δεύτερης	Cost<50
>=	Η πρώτη μεταβλητή μεγαλύτερη ή ίση της δεύτερης	Cost>=50
<=	Η πρώτη μεταβλητή μικρότερη ή ίση της δεύτερης	Cost<=50

Η εντολή if

Η σύνταξη της εντολής είναι η ακόλουθη

ıf(η_συνθήκη_είναι_αλήθεια)

εντολή

η εντολή **if** συμπληρώνεται από την εντολή **else** σε περίπτωση που η συνθήκη δεν είναι αληθής.

ıf(η_συνθήκη_είναι_αλήθεια)

εντολή

else

εντολή

Πολλές φορές μετά από μια εντολή if ή else χρειάζεται να δώσουμε ένα συνδυασμό εντολών. Τότε οι εντολές μπαίνουν μέσα σε αγκύλες {}.

ex6.cpp

```
#include <iostream.h>
void main(void)
 {
  int test_score;

  cout << "Type in the test score and press Enter: ";
  cin >> test_score;

  if (test_score >= 90)
   {
     cout << "Congratulations, you got an A!" << endl;
     cout << "Your test score was " << test_score << endl;
```

```
    }
  else
    {
      cout << "You should have worked harder!" << endl;
      cout << "You missed " << 100 - test_score <<
        " points " << endl;
    }
}
```

Μερικές φορές μέσα σε μια εντολή if χρειάζεται να ικανοποιούνται δυο ή περισσότερες συνθήκες ή μία από ένα συνδυασμό. Τότε χρησιμοποιούμε τα σύμβολα && (AND), για την ταυτόχρονη ικανοποίηση των συνθηκών ή τα σύμβολα || (OR), για την ικανοποίηση της μιας ή της άλλης συνθήκης.

Η μορφή των εντολών είναι η ακόλουθη:

If ((συνθήκη_1 == αλήθεια) && (συνθήκη_2 == αλήθεια))

εντολή

If ((συνθήκη_1 == αλήθεια) || (συνθήκη_2 == αλήθεια))

εντολή

Υπάρχει περίπτωση που το πρόγραμμα χρειάζεται να ελέγξει περισσότερες από μία συνθήκες. Τότε χρησιμοποιείται η εντολή **else if.** Ας δούμε ένα παράδειγμα.

ex7.cpp

```
#include <iostream.h>


void main(void)
 {
   int test_score;


   cout << "Type in your test score and press Enter: ";
   cin >> test_score;


   if (test_score >= 90)
     cout << "You got an A!" << endl;
   else if (test_score >= 80)
     cout << "You got a B!" << endl;
   else if (test_score >= 70)
     cout << "You got a C" << endl;
```

```
  else if (test_score >= 60)
    cout << "Your grade was a D" << endl;
  else
    cout << "You failed the test" << endl;
}
```

Το προηγούμενο πρόγραμμα θα μπορούσε να γραφτεί και με την εντολή **switch** ως εξής

ex8.cpp

```
#include <iostream.h>
void main(void)
{
  char grade = 'B';

  switch (grade) {
    case 'A': cout << "Congratulations on your A" << endl;
              break;
    case 'B': cout << "Not bad, a B is ok" << endl;
              break;
    case 'C': cout << "C's are only average" << endl;
              break;
    case 'D': cout << "D's are terrible" << endl;
              break;
    default: cout << "No excuses! Study harder!" << endl;
            break;
  }
}
```

Επανάληψη μίας ή περισσοτέρων εντολών (σελ 99 – 150)

Η εντολή for

Η σύνταξη είναι η εξής

for (Δώσιμο_αρχικής_τιμής; έλεγχος; αύξηση)

εντολή

Στα παρακάτω παραδείγματα βλέπουμε πως λειτουργεί η εντολή

ex9.cpp

```
#include <iostream.h>
void main(void)
 {
   int count;
   int ending_value;

   cout << "Type in the ending value and press Enter: ";
   cin >> ending_value;

   for (count = 0; count <= ending_value; count++)
     cout << count << ' ';
 }
```

ex10.cpp

```
#include <iostream.h>
void main(void)
 {
   int count;
   int total = 0;

   for (count = 1; count <= 100; count++)
     {
       cout << "Adding " << count << " to " << total;
       total = total + count;
       cout << " yields " << total << endl;
```

```
    }
}
```

ex11.cpp

```
#include <iostream.h>
void main(void)
{
  char letter;
  double value;

  for (letter = 'A'; letter <= 'Z'; letter++)
    cout << letter;
  cout << endl;
  for (value = 0.0; value <= 1.0; value += 0.1)
    cout << value << ' ';
  cout << endl;
}
```

Άλλος τρόπος για να επαναλαμβάνονται εντολές είναι η εντολή **while**.

Η σύνταξη της είναι

while (συνθήκη_είναι_αλήθεια)

εντολή

ex12.cpp

```
#include <iostream.h>
void main(void)
{
  int done = 0;    // Ορίζεται σε true όταν πρόκειται για το Y το N
  char letter;

  while (! done)
   {
     cout << "\nType Y or N and press Enter to continue: ";
     cin >> letter;
     if ((letter == 'Y') || (letter == 'y'))
```

```
      done = 1;
    else if ((letter == 'N') || (letter == 'n'))
      done = 1;
    else
        cout << “Type Y or N only” << endl;   // invalid character
  }
 cout << "The letter you typed was " << letter << endl;
}
```

Αν θέλουμε έστω και μια φορά να εκτελεστεί μια εντολή μέσα σε μια εντολή while μπορούμε να χρησιμοποιήσουμε την εξής σύνταξη.

do{

εντολές

} while (συνθήκη_είναι_αληθής)

Εισαγωγή στις συναρτήσεις (σελ 186 - 236)

Οι συναρτήσεις έχουν την ακόλουθη μορφή

Τύπος Όνομα_συνάρτησης (παράμετροι)

```
{
   δηλώσεις_μεταβλητών;
   εντολές;
}
```

Οι συναρτήσεις μπορούν να είναι

- **void show_message(void)**

Η συνάρτηση ούτε επιστρέφει ούτε δέχεται τίποτα

ex13.cpp

```
#include <iostream.h>
void show_message(void)
 {
   cout << "Hello, I like C++" << endl;
 }
void main(void)
 {
   cout << "About to call the function" << endl;
   show_message();
   cout << "Back from the function" << endl;
 }
```

- **void show_number(int value)**

Η συνάρτηση δεν επιστρέφει τίποτα αλλά δέχεται έναν ακέραιο.

ex14.cpp

```
#include <iostream.h>
void show_number(int value)
 {
   cout << "The parameter's value is " << value << endl;
 }
void main(void)
 {
```

```
  show_number(1);
  show_number(1001);
  show_number(-532);
}
```

- **void show_employee(int age, float salary)**

Η συνάρτηση δεν επιστρέφει αλλά δέχεται έναν ακέραιο και έναν φυσικό αριθμό

ex15.cpp

```
#include <iostream.h>

void show_employee(int age, float salary)
{
  cout << "The employee is " << age << " years old" << endl;
  cout << "The employee makes $" << salary << endl;
}

void main(void)
{
  show_employee(32, 25000.00);
}
```

- **float average_value(int a, int b)**

Η συνάρτηση δέχεται δύο ακεραίους και επιστρέφει ένα φυσικό

ex16.cpp

```
#include <iostream.h>
float average_value(int a, int b)
{
  return((a + b) / 2.0);
}

void main(void)
{
   cout << "The average value is: " << average_value(5, 10) << endl;
}
```

Εργασία με πίνακες (σελ 289 – 339)

- Πίνακας είναι μια δομή δεδομένων που επιτρέπει πολλαπλή αποθήκευση τιμών του ιδίου τύπου.
- Όταν δηλώνουμε έναν πίνακα θα πρέπει να καθορίσουμε τον τύπο της μεταβλητής που θα αποθηκευτεί.
- Το πρώτο στοιχείο ενός πίνακα με το όνομα array είναι το array[0], το δεύτερο array[1] κ.λ.π.

Ας δούμε ένα παράδειγμα

ex17.cpp

```
#include <iostream.h>
void main(void)
 {
   int values[5];   // Δήλωση του πίνακα

   values[0] = 100;
   values[1] = 200;
   values[2] = 300;
   values[3] = 400;
   values[4] = 500;

   cout << "The array contains the following values" << endl;
   cout << values[0] << ' ' << values[1] << ' ' << values[2] << ' ' <<
          values[3] << ' ' << values[4] << endl;
 }
```

Μπορούμε στην αρχή να αναθέσουμε τιμές σε πίνακα. Ακόμα μπορούμε περάσουμε πίνακες σε συναρτήσεις όπως και με τις μεταβλητές.

Το πως επιτυγχάνεται, φαίνεται στο επόμενο παράδειγμα.

ex18.cpp

```
#include <iostream.h>
void show_array(int array[], int number_of_elements)
 {
   int i;
```

```
  for (i = 0; i < number_of_elements; i++)
    cout << array[i] << ' ';

  cout << endl;
}
void main(void)
{
  int little_numbers[5] = { 1, 2, 3, 4, 5 };
  int big_numbers[3] = { 1000, 2000, 3000 };

  show_array(little_numbers, 5);
  show_array(big_numbers, 3);
}
```

Μπορούμε να αναθέσουμε τιμές στα στοιχεία ενός πίνακα όπως παρακάτω

ex19.cpp

```
#include <iostream.h>
void get_values(int array[], int number_of_elements)
{
  int i;

  for (i = 0; i < number_of_elements; i++)
   {
     cout << "Enter value " << i << ": ";
     cin >> array[i];
   }
}
void main(void)
{
  int numbers[3];

  get_values(numbers, 3);
```

```
  cout << "The array values are as follows" << endl;

  for (int i = 0; i < 3; i++)
    cout << numbers[i] << endl;
}
```

Εργασία με αλφαριθμητικά χαρακτήρων (σελ 289 - 339)

- Για να δηλώσουμε ένα αλφαριθμητικό χαρακτήρων πρέπει να δηλώσουμε ένα πίνακα τύπου **char**.
- Σε κάθε στοιχείο του πίνακα αντιστοιχεί ένας χαρακτήρας.
- Η C++ δίνει την τιμή NULL (ASCII 0) στο τελευταίο χαρακτήρα της αλυσίδας
- Αλφαριθμητικά χαρακτήρων μπορούν να περαστούν σε συναρτήσεις όπως και κάθε πίνακας

ex20.cpp

```
#include <iostream.h>
void display(char string[])
 {
   for (int i = 0; string[i] != '\0'; i++)
      cout << string[i];
 }

void main(void)
 {
   display("I like C++");
 }
```

Στη συνέχεια θα δούμε συναρτήσεις που μπορούμε να χρησιμοποιούμε και βρίσκονται στη βιβλιοθήκη **string.h** .

Οι συναρτήσεις είναι οι ακόλουθες:

- strupr(), strlwr(). Μετατρέπουν τα πεζά σε κεφαλαία και τα κεφαλαία σε πεζά αντίστοιχα.
- strcpy(). Αντιγράφει ένα αλφαριθμητικό χαρακτήρων σε ένα άλλο.
- strcmp(). Συγκρίνει δυο αλφαριθμητικά χαρακτήρων και επιστρέφει μία ακέραιη τιμή. Αν έχουμε δυο αλφαριθμητικά s1 και s2 η συνάρτηση επιστρέφει

  τιμή < 0 if s1 < s2

  τιμή == 0 if s1 == s2

  τιμή > 0 if s1 > s2
- strcat(). Προσαρτά στο τέλος ενός αλφαριθμητικού ένα άλλο.
- strlen(). Μετρά το μήκος ενός αλφαριθμητικού.

Στα επόμενα παραδείγματα βλέπουμε τη σύνταξη και πως λειτουργούν οι παραπάνω συναρτήσεις.

ex21.cpp

```
#include <iostream.h>
#include <string.h> //Περιέχει πρωτότυπα συναρτήσεων
void main(void)
 {
   char title[] = "I like C++";
   char lesson[] = "Understanding Character Strings";

   cout << "Uppercase: " << strupr(title) << endl;
   cout << "Lowercase: " << strlwr(lesson) << endl;
 }
```

ex22.cpp

```
#include <string.h>
#include <iostream.h>

int main(void)
 {
    char string[10];
    char str1[]= "abcdefghi";
    char str2[]= "Level D" ;

    strcpy(string, str1);
    cout<<string<<endl;

    strcpy(string, str2);
    cout<<string<<endl;
 }
```

ex23.cpp

```
#include <string.h>
#include <iostream.h>
void main(void)
 {
   char buf1[] = "aaa", buf2[] = "bbb", buf3[] = "ccc";
   int ptr;

   ptr = strcmp(buf2, buf1);
   if (ptr > 0)
      cout<<"buffer 2 is greater than buffer 1"<<endl;
   else
      cout<<"buffer 2 is less than buffer 1"<<endl;

   ptr = strcmp(buf2, buf3);
   if (ptr > 0)
      cout<<"buffer 2 is greater than buffer 3"<<endl;
   else
      cout<<"buffer 2 is less than buffer 3"<<endl;


 }
```

ex24.cpp

```
#include <iostream.h>
#include <string.h>
void main(void)
{
  char string[] = " Visual C++ ";

  cout<<strlen(string)<<endl;

}
```

ex25.cpp

```
#include <string.h>
#include <iostream.h>

void main(void)
{
   char destination[25];
   char blank[] = " ", c[] = "C++", turbo[] = "Visual";

   strcpy(destination, turbo);
   strcat(destination, blank);
   strcat(destination, c);

   cout<<destination<<endl;

}
```

Εισαγωγή στους δείκτες (pointers) (σελ 455 – 523)

Μια συνάρτηση πρέπει να χρησιμοποιήσει δείκτες για να μπορέσει να αλλάξει την τιμή μιας παραμέτρου.

Ο τελεστής (&) επιτρέπει στο πρόγραμμα να προσδιορίσει την διεύθυνση στη μνήμη μιας μεταβλητής, ενώ ο τελεστής (*) βοηθάει το πρόγραμμα να προσδιορίσει την μεταβλητή που είναι αποθηκευμένη στην συγκεκριμένη διεύθυνση.

Το επόμενο παράδειγμα δείχνει τα παραπάνω

ex26.cpp

```
#include <iostream.h>
void swap_values(float *a, float *b)
 {
   float temp;

   temp = *a;
   *a = *b;
   *b = temp;
 }

void main(void)
 {
   float big = 10000.0;
   float small = 0.00001;

   swap_values(&big, &small);

   cout << "Big contains " << big << endl;
   cout << "Small contains " << small << endl;
 }
```

Ας δούμε τώρα πως πως μπορούμε να χρησιμοποιήσουμε τους δείκτες με χαρακτήρες αλφαριθμητικών.

Το επόμενο παράδειγμα δείχνει ακριβώς πως γίνεται αυτό.

ex27.cpp

```
#include <iostream.h>
void show_string(char *string)
{
  while (*string != '\0')
    {
      cout << *string;
      string++;
    }
}
void main(void)
{
  show_string("I like C++!");
}
```

Μία συνηθισμένη περίπτωση όπου χρησιμοποιούνται δείκτες με αλφαριθμητικά είναι όταν χρειάζεται να εμφανιστούν μηνύματα λάθους και το πρόγραμμα να τερματιστεί. Το παρακάτω πρόγραμμα υλοποιεί ένα σενάριο όπου υπάρχει πρόβλημα στη διαθέσιμη μνήμη του υπολογιστή.

ex28.cpp

```
#include <iostream.h>
#include <stdlib.h>


void error(char *errmsg)
{
        cerr<<"\nRuntime Error ...\n";
        cerr<<errmsg;
        cerr<<"\n...Quitting Program!\n";
        exit(1);
}
void main(void)
{
        error("allocation failure");
}
```

Όπως χρησιμοποιούσαμε έναν πίνακα για να αποθηκεύσουμε ένα αλφαριθμητικό χαρακτήρων και κάθε στοιχείο του πίνακα αντιπροσώπευε ένα χαρακτήρα, έτσι ένας δείκτης δείχνει τη θέση στη μνήμη κάθε χαρακτήρα. Τους δείκτες μπορούμε να τους χρησιμοποιήσουμε και με άλλους τύπους πινάκων.

ex29.cpp

```
#include <iostream.h>
void show_float(float *array,  int number_of_elements) {
  int i;

  for (i = 0; i < number_of_elements; i++)
    cout << *array++ << endl;
 }
void main(void) {
  float values[5] = {1.1, 2.2, 3.3, 4.4, 5.5};
  show_float(values, 5);
  }
```

Αποθήκευση συσχετιζόμενων δεδομένων σε δομές (σελ 156 – 182)

Οι δομές επιτρέπουν στο πρόγραμμα να ομαδοποιεί πληροφορίες που σχετίζονται μεταξύ τους και διαφέρουν στον τύπο.

Μια δομή αποτελείται από μέλη που μπορεί να είναι διαφορετικού τύπου.

Το επόμενο παράδειγμα δείχνει πως μπορεί να χρησιμοποιηθεί μια δομή (structure).

ex30.cpp

```
#include <iostream.h>
#include <string.h>
void main(void)
 {
   struct employee {
     char name[64];
     long employee_id;
     float salary;
     char phone[10];
     int office_number;
   } worker;

   strcpy(worker.name, "John Doe");
   strcpy(worker.phone, "555-1212");
   worker.employee_id = 12345;
   worker.salary = 25000.00;
   worker.office_number = 102;

   cout << "Employee: " << worker.name << endl;
   cout << "Phone: " << worker.phone << endl;
   cout << "Employee id: " << worker.employee_id << endl;
   cout << "Salary: " << worker.salary << endl;
   cout << "Office: " << worker.office_number << endl;
 }
```

Στο παραπάνω παράδειγμα η δομή ονομάζεται **employee** και περιλαμβάνει διάφορες **μεταβλητές – μέλη**.

Ακόμη βλέπουμε ότι δηλώνεται στο πρόγραμμα μια μεταβλητή που έχει τον τύπο της δομής, **employee worker.**

Η προσπέλαση στα μέλη της δομής γίνεται με τον εξής τρόπο: **όνομα_δομής.μέλος**
π.χ. **worker.salary**.

Έτσι όπως είναι γραμμένο το πρόγραμμα βλέπουμε πως τα μέλη της structure μπορούν να προσπελαστούν μόνο από ένα worker. Αν είχα περισσότερους π.χ. 20 θα μπορούσα να χρησιμοποιήσω ένα πίνακα worker[19]. Δοκιμάστε το.

Ας δούμε κάτι πιο σύνθετο τώρα δηλαδή πως μπορούν να συνεργαστούν δομές και συναρτήσεις.

ex31.cpp

```
#include <iostream.h>
#include <string.h>
struct employee {
  char name[64];
  long employee_id;
  float salary;
  char phone[10];
  int office_number;
 };

void show_employee(employee worker)
 {
   cout << "Employee: " << worker.name << endl;
   cout << "Phone: " << worker.phone << endl;
   cout << "Employee id: " << worker.employee_id << endl;
   cout << "Salary: " << worker.salary << endl;
   cout << "Office: " << worker.office_number << endl;
 }

void main(void)
 {
   employee worker;

   // Copy a name to the string
   strcpy(worker.name, "John Doe");
```

```
  worker.employee_id = 12345;
  worker.salary = 25000.00;
  worker.office_number = 102;

  // Αντίγραφο αριθμού τηλεφώνου στο αλφαριθμητικό
  strcpy(worker.phone, "555-1212");
  show_employee(worker);
 }
```

Όπως βλέπουμε το πρόγραμμα περνάει την δομή με το όνομα worker στη συνάρτηση show_employee η οποία εμφανίζει στην οθόνη τα μέλη της δομής. Παρατηρήστε ότι το πρόγραμμα ορίζει την δομή έξω από το main και πριν από την συνάρτηση show_employee.

Αν η συνάρτηση αλλάζει ένα μέλος της δομής τότε η σύνδεση δομής και συνάρτησης γίνεται με τον τρόπο του παραδείγματος που ακολουθεί.

ex32.cpp

```
#include <iostream.h>
#include <string.h>

struct employee {
  char name[64];
  long employee_id;
  float salary;
  char phone[10];
  int office_number;
 };

void get_employee_id(employee *worker)
 {
   cout << "Type in an employee id: ";
   cin >> worker->employee_id;
 }
```

```
void main(void)
{
  employee worker;

  // Αντίγραφο ανόματος στο αλφαριθμητικό
  strcpy(worker.name, "John Doe");
  get_employee_id(&worker);

  cout << "Employee: " << worker.name << endl;
  cout << "Id: " << worker.employee_id << endl;
}
```

Εργασία με αρχεία (σελ 593 – 654)

Πολλές φορές όταν τα προγράμματα γίνονται πιο περίπλοκα χρειάζεται να αποθηκεύονται και να ανακτώνται δεδομένα από αρχεία.

Για να γράψουμε κάποια δεδομένα σε ένα αρχείο χρησιμοποιούμε την εντολή

ofstream Δείκτης_αρχείου ("όνομα αρχείου").

Ας δούμε ένα παράδειγμα

ex33.cpp

```
#include <fstream.h>


void main(void)
 {
   ofstream book_file("BookInfo.DAT");


   book_file << "I like C++" << endl;
   book_file << "Alfa Press" << endl;
   book_file << "29.95" << endl;
 }
```

Το προηγούμενο παράδειγμα δημιουργεί ένα αρχείο με το όνομα BookInfo.DAT και γράφει τα παρακάτω δεδομένα

I like C++

Alfa Press

29.95

Πρέπει να σημειωθεί ότι αν υπάρχει ήδη το αρχείο τότε τα υπάρχοντα δεδομένα χάνονται και γράφονται τα καινούργια.

Για να διαβάσουμε τα δεδομένα ενός αρχείου χρησιμοποιούμε την εντολή

ifstream δείκτης_αρχείου ("όνομα αρχείου")

Ας δούμε ένα παράδειγμα

ex34.cpp

```
#include <iostream.h>
#include <fstream.h>
```

```
void main(void)
 {
   ifstream input_file("BookInfo.DAT");

   char one[64], two[64], three[64];

   input_file >> one;
   input_file >> two;
   input_file >> three;

   cout << one << endl;
   cout << two << endl;
   cout << three << endl;
 }
```

Όταν εκτελεστεί το παραπάνω πρόγραμμα θα περιμέναμε ίσως να εμφανίσει τις τρεις πρώτες σειρές του αρχείου. Όμως δεν συμβαίνει αυτό. Αυτό που εκτυπώνεται είναι το εξής:

I

like

C++,

Αυτό συμβαίνει γιατί η C++ χρησιμοποιεί το κενό για να ξεχωρίζει που τελειώνει και που αρχίζει μία λέξη ή ένα αλφαριθμητικό.

Αν θέλουμε να διαβάσουμε κάθε φορά μια σειρά θα πρέπει να ακολουθήσουμε τη λογική του παρακάτω παραδείγματος.

ex35.cpp

```
#include <iostream.h>
#include <fstream.h>

void main(void)
 {
   ifstream input_file("BookInfo.DAT");

   char one[64], two[64], three[64];

   input_file.getline(one, sizeof(one));
```

```
  input_file.getline(two, sizeof(two));
  input_file.getline(three, sizeof(three));

  cout << one << endl;
  cout << two << endl;
  cout << three << endl;
}
```

Πολλές φορές θέλουμε να διαβάσουμε τα περιεχόμενα ενός αρχείου. Για να δούμε που τελειώνει το αρχείο χρησιμοποιούμε τη λογική του επόμενου παραδείγματος.

ex36.cpp

```
#include <iostream.h>
#include <fstream.h>

void main(void)
 {
  ifstream input_file("BookInfo.DAT");

  char line[64];

  while (! input_file.eof())
   {
    input_file.getline(line, sizeof(line));

    cout << line << endl;
   }
 }
```

Το επόμενο παράδειγμα διαβάζει λέξη προς λέξη όλα τα δεδομένα του αρχείου

ex37.cpp

```
#include <iostream.h>
#include <fstream.h>
```

```
void main(void)
 {
   ifstream input_file("BookInfo.DAT");

   char word[64];

   while (! input_file.eof())
    {
      input_file >> word;

      cout << word << endl;
    }
 }
```

Το επόμενο παράδειγμα διαβάζει χαρακτήρα προς χαρακτήρα όλα τα δεδομένα του αρχείου.

ex38.cpp

```
#include <iostream.h>
#include <fstream.h>

void main(void)
 {
   ifstream input_file("BookInfo.DAT");

   char letter;

   while (! input_file.eof())
    {
      letter = input_file.get();

      cout << letter;
    }
 }
```

Αν θέλουμε να κλείσουμε ένα αρχείο όταν δεν το χρειαζόμαστε πλέον, τότε μπορούμε να χρησιμοποιήσουμε την εξής εντολή:

input_file.close();

Στα προηγούμενα παραδείγματα είτε ανοίγανε τα αρχεία για γράψιμο, είτε για διάβασμα δεδομένων, όλα ξεκινούσαν από την αρχή του αρχείου. Αν θέλουμε να προσθέσουμε δεδομένα στα ήδη υπάρχοντα τότε χρησιμοποιούμε την εξής εντολή:

ifstream δείκτης_αρχείου("όνομα_αρχείου", ios::app)

Ως τώρα όλες οι εργασίες με αρχεία αφορούσαν χαρακτήρες αλφαριθμητικών. Υπάρχουν φορές που θέλουμε να γράψουμε ή να διαβάσουμε πίνακες ή δομές. Τα παρακάτω παραδείγματα δείχνουν πως γίνεται αυτό.

ex39.cpp

```
#include <iostream.h>
#include <fstream.h>

void main(void)
 {
  struct employee {
    char name[64];
    int age;
    float salary;
  } worker = { "John Doe", 33, 25000.0 };

  ofstream emp_file("Employee.DAT");

  emp_file.write((char *) &worker, sizeof(employee));
 }
```

ex40.cpp

```
#include <iostream.h>
#include <fstream.h>

void main(void)
 {
  struct employee {
    char name[64];
```

```
  int age;
  float salary;
 } worker = { "John Doe", 33, 25000.0 };

 ifstream emp_file("Employee.DAT");

 emp_file.read((char *) &worker, sizeof(employee));

 cout << worker.name << endl;
 cout << worker.age << endl;
 cout << worker.salary << endl;
}
```

ΑΝΤΙΚΕΙΜΕΝΟΣΤΡΕΦΗΣ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ

Εργασία με κλάσεις (σελ 240 – 284)

Η βασικότερη έννοια στον αντικειμενοστρεφή προγραμματισμό είναι η ενσωμάτωση **δεδομένων** (χαρακτηριστικών) και **συναρτήσεων** (συμπεριφορές) σε οντότητες που ονομάζονται **κλάσεις**. Μια κλάση είναι ένας αφηρημένος τύπος δεδομένων, τον οποίο είναι δυνατόν να χρησιμοποιήσει ένας προγραμματιστής χωρίς να ενδιαφέρεται για τον εσωτερικό τρόπο υλοποίησής του. Από μια κλάση είναι δυνατόν να δημιουργηθούν συγκεκριμένα αντικείμενα, όπως ακριβώς από έναν τύπο δεδομένων int είναι δυνατό να δημιουργηθούν συγκεκριμένες μεταβλητές ακεραίων.
Τα δεδομένα μιας κλάσης αναφέρονται και ως **μέλη δεδομένων** της κλάσης ενώ οι συναρτήσεις που δρουν επί αυτών αναφέρονται ως **μέθοδοι** της κλάσης.

Υλοποίηση ενός αφηρημένου τύπου δεδομένων με μια κλάση
Ο ορισμός μιας κλάσης που αφορά ένα σημείο του επιπέδου είναι ο εξής :

```
class Point
{
 public:
        Point();
        void setPoint(int a,int b);
        void printPoint();
 private:
        int x;
        int y;
};
```

Ο ορισμός της κλάσης ξεκινά με τη δεσμευμένη λέξη **class** και το όνομα τύπου της κλάσης και ακολουθεί το σώμα της κλάσης το οποίο περικλείεται σε ζεύγος άγκιστρων και ολοκληρώνεται με ένα ελληνικό ερωτηματικό (;). Η κλάση Point περιλαμβάνει μεθόδους που δίνουν τη δυνατότητα προσδιορισμού των συντεταγμένων του σημείου και εκτύπωσή του, καθώς και δύο μέλη δεδομένων που αφορούν την τετμημένη και τεταγμένη του σημείου.
Οι δεσμευμένες λέξεις public και private ονομάζονται προσδιοριστές προσπέλασης και καθορίζουν τα δικαιώματα προσπέλασης στα μέλη μιας κλάσης. Υπάρχουν τρεις προσδιοριστές προσπέλασης η έννοια των οποίων είναι η παρακάτω.
public: Όλα τα μέλη (δεδομένα ή συναρτήσεις) μετά τον προσδιοριστή public και πριν από οποιονδήποτε άλλο προσδιοριστή είναι προσπελάσιμα από οποιοδήποτε μέρος του προγράμματος.
private: Τα μέλη που ακολουθούν τον προσδιοριστή private μπορεί να χρησιμοποιηθούν μόνο από τις μεθόδους που ανήκουν στην ίδια κλάση ή από συναρτήσεις που δηλώνονται φιλικές προς την κλάση. Δεν είναι δυνατόν εξωτερικές συναρτήσεις (συναρτήσεις δηλωμένες εκτός της κλάσης) να προσπελάσουν private μέλη της κλάσης. Επίσης, τα private μέλη δεν είναι προσπελάσιμα από παράγωγες κλάσεις, δηλαδή κλάσεις που κληρονομούν τη συγκεκριμένη κλάση.
protected: Τα protected μέλη μιας κλάσης όπως και τα private δεν είναι προσπελάσιμα από εξωτερικές συναρτήσεις με τη διαφορά ότι είναι προσπελάσιμα από τις παράγωγες κλάσεις.
Ένας προσδιοριστής βρίσκεται σε ισχύ από το σημείο που εμφανίζεται μέχρι την εμφάνιση του επόμενου προσδιοριστή ή του αγκίστρου που κλείνει τον ορισμό της κλάσης.

Το πρόγραμμα που ακολουθεί χρησιμοποιεί την κλάση Point και υλοποιεί ένα αντικείμενο τύπου Point, δηλαδή ένα σημείο του επιπέδου.

ex41.cpp

```
#include <iostream.h>

class Point
{
public:
        Point(); //constructor
        void setPoint(int a,int b); //καθορισμός συντεταγμένων
        void printPoint(); //εκτύπωση συντεταγμένων
private:
        int x;
        int y;
};

Point::Point()                          //ο constructor του σημείου αρχικοποιεί
{                                       //τα μέλη δεδομένων με την τιμή 0
        x=0;
        y=0;
}
void Point::setPoint(int a, int b)
{
        x=a;
        y=b;
}
void Point::printPoint()
{
        cout<<x<<" "<<y<<"\n";
}
void main()
{
        Point p;
        p.setPoint(5,10);
        p.printPoint();
}
```

Μετά τον ορισμό μιας κλάσης, είναι δυνατόν να οριστούν μεταβλητές που ανήκουν σε τύπο κλάσης, όπως στο ανωτέρω πρόγραμμα η μεταβλητή p που είναι τύπου Point. Κατά τον ίδιο τρόπο είναι δυνατόν να δηλωθούν πίνακες αντικειμένων τύπου Point ή pointers και αναφορές προς αντικείμενα τύπου Point :

```
Point   arrayofPoints[5];          //array of Point objects
Point    *pointertoPoint           //pointer to a Point object
Point   &refPoint=p;               //reference to a Point object
```

Στο ανωτέρω πρόγραμμα, κατά την υλοποίηση ενός αντικειμένου p τύπου Point, καλείται αυτόματα η μέθοδος Point() της κλάσης, που ονομάζεται και κατασκευάστρια μέθοδος (constructor) της κλάσης. Η κατασκευάστρια μέθοδος είναι μια ειδική μέθοδος της κλάσης, που χρησιμοποιείται για τη δημιουργία των αντικειμένων μιας κλάσης. **Το όνομα μιας**

κατασκευάστριας μεθόδου είναι ίδιο με το όνομα της κλάσης στην οποία ανήκει. Μια κατασκευάστρια μέθοδος είναι δυνατόν να λαμβάνει παραμέτρους ή να έχει κενή λίστα παραμέτρων όπως στο προηγούμενο παράδειγμα. Σε περίπτωση που ο προγραμματιστής δεν παρέχει κάποια κατασκευάστρια μέθοδο, ο μεταγλωττιστής του συστήματος δημιουργεί μια προεπιλεγμένη ή εξ ορισμού (default) κατασκευάστρια μέθοδο.
Στο προηγούμενο πρόγραμμα οι μέθοδοι της κλάσης Point παρέχουν τη δυνατότητα προσδιορισμού των συντεταγμένων ενός σημείου καθώς και τη δυνατότητα εκτύπωσής του. Οι μέθοδοι είναι δηλωμένες ως public, γεγονός που παρέχει τη δυνατότητα χρήσης τους έξω από την κλάση. Αυτή η περίπτωση είναι η πιο συνηθισμένη, καθώς οι μέθοδοι παρέχουν τη δυνατότητα επικοινωνίας ενός αντικειμένου με άλλα αντικείμενα (διασύνδεση — interface). Η δήλωση των μεθόδων σε μια κλάση συνήθως πραγματοποιείται δηλώνοντας μόνο το πρωτότυπό τους (επικεφαλίδα) μέσα στην κλάση, ενώ ο πλήρης ορισμός τους (σώμα της μεθόδου) πραγματοποιείται εκτός της κλάσης. Σε αυτή την περίπτωση, κατά τον πλήρη ορισμό των μεθόδων, θα πρέπει να δηλώνεται η κλάση στην οποία ανήκουν χρησιμοποιώντας τον τελεστή διάκρισης (::).
Τα δύο μέλη δεδομένων της κλάσης Point είναι δηλωμένα ως private και κατά συνέπεια ο χειρισμός τους είναι δυνατόν να γίνει μόνο μέσω των μεθόδων της κλάσης. Η φιλοσοφία που υιοθετείται συνήθως στην αντικειμενοστρεφή σχεδίαση, είναι ότι η εσωτερική αναπαράσταση των δεδομένων δεν ενδιαφέρει τα αντικείμενα με τα οποία επικοινωνεί το εν λόγω αντικείμενο. Η αρχή αυτή της απόκρυψης πληροφορίας επαυξάνει τη δυνατότητα επέκτασης και τροποποίησης ενός προγράμματος. Αν για παράδειγμα η εσωτερική υλοποίηση μιας κλάσης τροποποιηθεί, δεδομένου ότι ο τρόπος επικοινωνίας της κλάσης με άλλα αντικείμενα παραμείνει αναλλοίωτος, δεν απαιτείται να τροποποιηθεί και ο κώδικας των υπολοίπων κλάσεων.

Destructors

Η μέθοδος καταστροφής (destructor) μιας κλάσης είναι μια ειδική μέθοδος που εκτελεί το αντίθετο από ότι μια κατασκευάστρια μέθοδος, δηλαδή καθαρίζει τον χώρο μνήμης που δεσμεύεται από ένα αντικείμενο όταν αυτό πλέον δεν χρησιμοποιείται. Το όνομα του destructor είναι η περισπωμένη (~) ακολουθούμενη από το όνομα της κλάσης. Μία κλάση μπορεί να διαθέτει μόνο μία μέθοδο καταστροφής η οποία δεν λαμβάνει παραμέτρους και δεν επιστρέφει κάποια τιμή. Σε περίπτωση που ο προγραμματιστής δεν παρέχει μια μέθοδο καταστροφής, δημιουργείται αυτόματα από το σύστημα μια προεπιλεγμένη μέθοδος καταστροφής (default destructor).

Διαχωρισμός Υλοποίησης και Τρόπου Επικοινωνίας με μία Κλάση

Μία βασική αρχή στην τεχνολογία λογισμικού είναι ο διαχωρισμός του τρόπου επικοινωνίας με μία κλάση και της εσωτερικής υλοποίησής της. Κατ' αυτόν τον τρόπο η τροποποίηση των προγραμμάτων γίνεται ευκολότερη. Για παράδειγμα, σε περίπτωση που κάποια προγράμματα εξυπηρετούνται από μία κλάση, τροποποιήσεις στον εσωτερικό τρόπο υλοποίησης της κλάσης δεν απαιτούν την τροποποίηση των εξυπηρετούμενων προγραμμάτων, όσο ο τρόπος επικοινωνίας με την κλάση (interface) δεν αλλάζει.
Ο ορισμός μιας κλάσης είναι δυνατόν να τοποθετηθεί σε ένα αρχείο κεφαλίδων (header file) το οποίο μπορεί να ενσωματωθεί σε οποιοδήποτε πρόγραμμα πρόκειται να χρησιμοποιήσει την κλάση αυτή. Ο ορισμός της κλάσης παρέχει τον εξωτερικό τρόπο επικοινωνίας με την κλάση. Ο πλήρης ορισμός των μεθόδων της κλάσης που ουσιαστικά αποτελούν την υλοποίηση της κλάσης είναι δυνατόν να τοποθετηθούν σε ένα ξεχωριστό πηγαίο αρχείο (source file). Κατά αυτόν τον τρόπο, παρέχεται η δυνατότητα χρήσης μιας κλάσης χωρίς να υπάρχει απαραίτητα πρόσβαση στον πηγαίο κώδικα της κλάσης. Είναι δηλαδή δυνατόν να χρησιμοποιηθεί μια κλάση έχοντας πρόσβαση μόνο στο αντίστοιχο αρχείο κεφαλίδων και στον αντίστοιχο

αντικείμενο κώδικα που παράγεται από τη μεταγλώττιση του πηγαίου αρχείου που περιέχει τους ορισμούς των μεθόδων της κλάσης.
Για παράδειγμα, στο προηγούμενο πρόγραμμα που αναπτύχθηκε, ο ορισμός της κλάσης είναι δυνατόν να τοποθετηθεί σε ένα ξεχωριστό αρχείο επικεφαλίδων με όνομα point.h:

```
#define POINT_H
class Point
{
public:
        Point(); //constructor
        void setPoint(int a,int b); //ορίζει συντεταγμένες σημείου
        void printPoint(); //εκτυπώνει συντεταγμένες σημείου
private:
        int x;
        int y;
};
```

Το αρχείο κεφαλίδων "point.h" είναι δυνατόν να ενσωματωθεί σε ένα πηγαίο αρχείο μέσω της οδηγίας #include, ώστε να υπάρχει η δυνατότητα χρήσης της συγκεκριμένης κλάσης :

ex42.cpp

```
#include <iostream.h>
#include "point.h"

Point::Point() {x=0;y=0;}

void Point::setPoint(int a, int b) {x=a;y=b;}
void Point::printPoint() {cout<<x<<" "<<y<<"\n";}
void main(){
        Point p;
        p.setPoint(5,10);
        p.printPoint();
}
```

Φιλικές Συναρτήσεις και Κλάσεις
Μία φιλική συνάρτηση (friend function) προς μια κλάση, ορίζεται έξω από το χώρο δράσης της κλάσης, ωστόσο έχει το δικαίωμα πρόσβασης των private μελών της κλάσης. Ομοίως, μια ολόκληρη κλάση μπορεί να δηλωθεί φιλική προς μία άλλη κλάση.
Για να δηλωθεί μια συνάρτηση φιλική προς μια κλάση, στη δήλωση της συνάρτησης μέσα στον ορισμό μιας κλάσης θα πρέπει να προηγείται ο προσδιοριστής **friend**. Για να δηλωθεί η κλάση ClassB ως φιλική της κλάσης ClassA, θα πρέπει να συμπεριληφθεί μια δήλωση:

```
        friend class ClassB;
```

στον ορισμό της κλάσης ClassA.

Θεωρούμε το ακόλουθο πρόγραμμα χρήσης μιας φιλικής προς μια κλάση συνάρτησης:

ex43.cpp

```
#include <iostream.h>
```

```
class Simple
{
	friend void setX(Simple &, int); //δήλωση φιλικής συνάρτησης
public:
	Simple() {x=0;}
	void print() const {cout << x << endl; }
private:
	int x;
};
void setX(Simple &s, int val)
{
	s.x = val; //Επιτρεπτό διότι η setX είναι friend
}
int main()
{
	Simple simpleClass;
	cout << “simpleClass.x after instantiation: “;
	simpleClass.print();
	cout << “simpleClass.x after call of setX: “
	setX(simpleClass, 8);
	simpleClass.print();
	return 0;
}
```

Το πρόγραμμα εκτυπώνει:

```
	simpleClass.x after instantiation: 0
	simpleClass.x after call of setX: 8
```

Στο ανωτέρω πρόγραμμα η συνάρτηση setX έχει δηλωθεί φιλική για να μπορεί να προσπελάσει το μέλος δεδομένων x της κλάσης Simple.
Θα πρέπει να σημειωθεί ότι η συνάρτηση setX είναι μία μεμονωμένη συνάρτηση. Δεν είναι μέθοδος της κλάσης Simple. Για αυτό το λόγο το συγκεκριμένο αντικείμενο της κλάσης για το οποίο καλείται, περνά ως παράμετρος της συνάρτησης setX.

Αφαίρεση Δεδομένων και Απόκρυψη Πληροφορίας

Ο κύριος σκοπός ύπαρξης των κλάσεων είναι η απόκρυψη των εσωτερικών λεπτομερειών υλοποίησής από τους χρήστες της κλάσης, γεγονός που ονομάζεται απόκρυψη πληροφορίας. Θεωρούμε ως παράδειγμα απόκρυψης πληροφορίας μία δομή δεδομένων που ονομάζεται στοίβα. Σε μία στοίβα η εισαγωγή και εξαγωγή των δεδομένων γίνεται με βάση την αρχή (LIFO: last-in, first-out), δηλαδή το τελευταίο στοιχείο που εισάγεται στη δομή είναι το πρώτο στοιχείο που εξάγεται.
Ο προγραμματιστής είναι δυνατόν να δημιουργήσει μία κλάση στοίβας και να αποκρύψει από τους χρήστες της στοίβας τον εσωτερικό τρόπο υλοποίησής της. Αυτό συμβαίνει διότι ο χρήστης μιας στοίβας δεν χρειάζεται να γνωρίζει τις λεπτομέρειες υλοποίησής της. Ο χρήστης απλώς απαιτεί την ύπαρξη δυνατότητας εισαγωγής και εξαγωγής στοιχείων με βάση την αρχή LIFO. Η περιγραφή της λειτουργικότητας μιας κλάσης ανεξάρτητα του τρόπου υλοποίησής της ονομάζεται αφαίρεση δεδομένων και οι κλάσεις στην C++ ορίζουν αφηρημένους τύπους δεδομένων (Abstract Data Types – ADTs). Υπό αυτή την έννοια, η εσωτερική υλοποίηση μιας κλάσης (π.χ. ο τρόπος υλοποίησης της στοίβας) είναι δυνατόν να τροποποιηθεί χωρίς να

απαιτούνται αλλαγές στα υπόλοιπα τμήματα του λογισμικού που συνεργάζονται, δεδομένου ότι ο τρόπος επικοινωνίας με την κλάση μέσω των public μελών της παραμένει αναλλοίωτος.

Κλάσεις και Υπολογιστικά Μαθηματικά

Πολλοί υπολογισμοί εμπεριέχουν τη λύση γραμμικών εξισώσεων. Στη συνέχεια θα παρουσιάσουμε τις κατάλληλες έννοιες για λύση προβλημάτων γραμμικής άλγεβρας και ειδικά τις κλάσεις vector και matrix οι οποίες υλοποιούν διανύσματα και πίνακες.
Το παρακάτω πρόγραμμα διαβάζει το μέγεθος ενός διανύσματος, τα στοιχεία που το αποτελούν και τέλος τα εκτυπώνει.

ex44.cpp

```
#include <iostream.h>
#include <cmath>
#include <stdlib.h>

void error(char *errmsg)
{
        cerr<<"\nRuntime Error ...\n";
        cerr<<errmsg;
        cerr<<"\n...Quitting Program!\n";
        exit(1);
}

class vector {
  friend ostream& operator<<(ostream&, const vector&);
  friend istream& operator>>(istream&, vector&);

  private:
   int size;
   double *vec;

 public:

  // Constructors
  vector(int);

  // Destructor is required because the constructor above uses ``new"
  ~vector();

// Accessors
  double& operator[](int );


};
vector::vector(int n)
{
        size=n;
        vec=new double[size];
        if(!vec)
                error("allocation failure");
}
```

```
vector::~vector()
{
  delete vec;
}

double& vector::operator[](int i)
{
        return vec[i];
}

istream& operator>>(istream &s,  vector &v)
{
        int n=v.size;
        for(int i=0; i<n; i++)
        {
                cout<<"dose to "<<i+1<<" stoixeio :";
                s>>v.vec[i];

        }
        return s;
}

ostream& operator<<(ostream &s, const vector &v)
{
        int n=v.size;
        for(int i=0; i<n; i++)
        {
                s<<v.vec[i];
                cout<<" ";
        }
        return s;
}

int main(){
   int n;
   cout << "Enter size of vector\n";
   cin >> n;

   vector a(n);

   cout << "Enter the components\n";

   cin >> a;

   cout << "Your vector is \n";
   cout << a;

  return 0;
}
```

Ας εξετάσουμε το παραπάνω πρόγραμμα.

```
 private:
  int size;
  double *vec;
```

Οι μεταβλητές size και vec είναι τα αριθμητικά μέλη της κλάσης. Η μεταβλητή size κρατάει το μέγεθος του διανύσματος ενώ η μεταβλητή vec τη διεύθυνση του πρώτου στοιχείου του διανύσματος.

```
vector(int);
```

Πρόκειται για την κατασκευάστρια μέθοδο του διανύσματος. Η κατασκευάστρια μέθοδος καλείται στο κύριο πρόγραμμα με την εντολή **vector a(n);**. Πρέπει να σημειωθεί ότι πριν από την κλήση της κατασκευάστριας μεθόδου πρέπει να γνωρίζουμε το μέγεθος του διανύσματος.

```
~vector();
```

Πρόκειται για τη μέθοδο καταστροφής (destructor) της κλάσης.

```
  double& operator[](int );
```

Ας σημειωθεί ότι στο κύριο πρόγραμμα μπορούμε να χρησιμοποιούμε το **a[i]** για να αναφερθούμε στο **i**-στοιχείο του διανύσματος σαν να έχουμε ένα απλό (μονοδιάστατο) πίνακα. Όμως ο **a** δεν είναι ένας απλός πίνακας αλλά μια περίπτωση όπου χρησιμοποιείται η κλάση vector. Έτσι χρησιμοποιώντας τη συνάρτηση **operator[]** βοηθούμε το μεταγλωττιστή να παίρνει ένα ακέραιο όρισμα **i** (η τιμή ανάμεσα στις αγκύλες), και να επιστρέφει το **i**-στοιχείο του διανύσματος.

```
friend ostream& operator<<(ostream&, const vector&);
friend istream& operator>>(istream&, vector&);
```

Αυτές είναι φιλικές συναρτήσεις οι οποίες μας επιτρέπουν με μία απλή εντολή στο κυριως πρόγραμμα να εισάγουμε και να εκτυπώσουμε τα στοιχεία ενός διανύσματος.

Αντίστοιχα στην περίπτωση που έχω διδιάστατο πίνακα και θέλω να εισάγω και να εκτυπώσω τα στοιχεία του θα πρέπει να εργαστώ ως εξής:

ex45.cpp

```
#include <iostream.h>
#include <cmath>
#include <stdlib.h>

void error(char *errmsg)
{
        cerr<<"\nRuntime Error ...\n";
        cerr<<errmsg;
        cerr<<"\n...Quitting Program!\n";
        exit(1);
}

class matrix;
```

```
class vector {
  friend ostream& operator<<(ostream&, const vector&);
  friend istream& operator>>(istream&, vector&);

  private:
   int size;
   double *vec;

 public:

  // Constructors
  vector(int);

  // Destructor is required because the constructor above uses ``new"
  ~vector();

// Accessors
  double& operator[](int );


};
vector::vector(int n)
{
        size=n;
        vec=new double[size];
        if(!vec)
                error("allocation failure");
}

vector::~vector()
{
  delete vec;
}

double& vector::operator[](int i)
{
        return vec[i];
}

istream& operator>>(istream &s,  vector &v)
{
        int n=v.size;
        for(int i=0; i<n; i++)
        {
                cout<<"dose to "<<i+1<<" stoixeio :";
                s>>v.vec[i];

        }
        return s;
}
```

```
ostream& operator<<(ostream &s, const vector &v)
{
        int n=v.size;
        for(int i=0; i<n; i++)
        {
                s<<v.vec[i];
                cout<<" ";
        }
        return s;
}
class matrix
{
friend ostream& operator<<(ostream&, const matrix&);
friend istream& operator>>(istream&, matrix&);
private:
        int numrows;
        int numcols;
        vector **mat;

public:

        matrix(int, int);
        ~matrix();
        vector& operator[](int i);

};

matrix::matrix(int nrows, int ncols)
{
        int j;

        numrows=nrows;
        numcols=ncols;
        mat=new vector* [numrows];

        if(!mat)
                error("allocation failure for rows");

        for(j=0; j<numrows; j++)
        {
                mat[j]=new vector(numcols);
                if (!mat[j])
                        error("allocation failure for columns");
        }

}
matrix::~matrix()
{
        for(int i=numrows; i>0; i--)
```

```
                delete mat[i-1];
        delete mat;
}

vector& matrix::operator[](int i)
{
        return *mat[i];
}

istream& operator>>(istream &s, matrix &m)
{
        int nr=m.numrows;
        cout<<"doste ta stoixeia tou pinaka ana grammi"<<endl;
        for(int i=0; i<nr; i++)
        {
                cout<<"dose to stoixeio ths "<<i+1<<" grammis"<<endl;
                s>>*m.mat[i];

        }
        return s;
}


ostream& operator<<(ostream &s, const matrix &m)
{
        int nr=m.numrows;
        for(int i=0; i<nr; i++)
        {
                s<<*m.mat[i];
                cout<<endl;
        }
        return s;
}
int main(){
   int nr,nc;
   cout<<"doste ton arithmo ton grammon :";
        cin>>nr;
        cout<<"doste ton arithmo ton stilon :";
        cin>>nc;
        cout<<endl;

        matrix m(nr,nc);
        cin>>m;

        cout<<"o pinakas exei tin akolouthi morfi"<<endl;
        cout<<m;
        cout<<endl;
}
```

ΑΝΤΙΚΕΙΜΕΝΟΣΤΡΕΦΗΣ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ: ΚΛΗΡΟΝΟΜΙΚΟΤΗΤΑ

Οι δύο βασικές ιδιότητες του αντικειμενοστρεφούς προγραμματισμού που υποστηρίζονται από την C++ είναι η *κληρονομικότητα* και η *πολυμορφία.* Στο κεφάλαιο αυτό θα εξετάσουμε την κληρονομικότητα που είναι η δυνατότητα παραγωγής κλάσεων από άλλες υπαρκτές κλάσεις, κληρονομώντας χαρακτηριστικά (μέλη δεδομένων) και συμπεριφορά (μέλη συναρτήσεων) αλλά και τροποποίησης ή επέκτασης της κληρονομούμενης κλάσης.

Απλή Κληρονομικότητα

Κατά τη δημιουργία μιας νέας κλάσης, ο προγραμματιστής έχει τη δυνατότητα αντί να γράψει εξ' ολοκλήρου νέα μέλη δεδομένων και νέες μεθόδους, να ορίσει ότι η νέα κλάση είναι **παράγωγη κλάση** μιας προηγούμενα ορισμένης κλάσης, που αποτελεί την **κλάση βάσης** (ή βασική κλάση).

Για τον ορισμό μιας παράγωγης κλάσης από μία βασική κλάση αναφέρεται το όνομα της βασικής κλάσης στον ορισμό με το διαχωριστικό στήλης (:). Στο παράδειγμα που ακολουθεί η παράγωγη κλάση Circle κληρονομεί την βασική κλάση Point:

```
class Point
{
public:
 Point();                        //constructor
 void setPoint(int a,int b);        //καθορισμός συντεταγμένων
 void printPoint();              //εκτύπωση συντεταγμένων
private:
 int x;
 int y;
};
```

```
class Circle:public Point   //Η κλάση Circle κληρονομεί την Point
{
public:
 Circle();
 Circle(double r, int x, int y);
 void printRadius();
protected:
 double radius;
};
```

Η κλάση Circle κληρονομεί τις μεθόδους της κλάσης Point, ορίζοντας επιπρόσθετα τους δικούς της constructors, μία μέθοδο για τον ορισμό της ακτίνας του κύκλου και ένα protected μέλος δεδομένων, την ακτίνα του κύκλου. Η ύπαρξη του προσδιοριστή public στον ορισμό της κλάσης **class Circle:public Point** προσδιορίζει ότι όλα τα public μέλη της κλάσης Point καθίστανται και public μέλη της κλάσης Circle. Ένα αντικείμενο τύπου Circle μπορεί να χρησιμοποιεί και τις μεθόδους της βασικής κλάσης τύπου Point:

```
#include <iostream.h>

using std::cout;

class Point{
```

```
public:
 Point();                  //constructor
 void setPoint(int a,int b);   //καθορισμός συντεταγμένων
 void printPoint();        //εκτύπωση συντεταγμένων
private:
 int x,y;
};

class Circle:public Point{//Η κλάση Circle
public:                          //κληρονομεί την Point
 Circle();
 Circle(double r, int x, int y);
 void printRadius();
protected:
 double radius;
};

Point::Point()  {x=0;y=0;}

Circle::Circle(){
 radius=0;
 Point::Point();
}

Circle::Circle(double r, int a, int b){
 radius=r;
 Point::setPoint(a,b);
}

void Point::setPoint(int a, int b){
 x=a;
 y=b;
}

void Point::printPoint(){
 cout<<"x= "<<x<<" y= "<<y<<"\n";
}

void Circle::printRadius(){
 cout<<"Radius = "<<radius<<"\n";
}


void main()
{
 Point p;
 Circle c(10,40,50);

 p.setPoint(5,10);
 p.printPoint();
```

```
 c.printPoint();
 c.printRadius();
}
```

Στο ανωτέρω κυρίως πρόγραμμα (main) η μέθοδος printPoint καλείται τόσο από το αντικείμενο p τύπου Point όσο και από το αντικείμενο c τύπου Circle, όπου και εκτυπώνει τις συντεταγμένες του κέντρου του κύκλου. Η μέθοδος Circle(double r, int x, int y) είναι επίσης ένας constructor για την κλάση Circle ο οποίος λαμβάνει τρεις παραμέτρους και δημιουργεί ένα αντικείμενο τύπου Circle αρχικοποιώντας τόσο τις συντεταγμένες του κέντρου του κύκλου όσο και την ακτίνα του. Παρατηρούμε ότι στον ορισμό του constructor καλείται η μέθοδος setPoint της κλάσης Point χρησιμοποιώντας τον τελεστή διάκρισης.

Constructors και Destructors κατά την κληρονομικότητα

Οι constructors και οι destructors μιας βασικής κλάσης δεν κληρονομούνται από την παράγωγή της κλάσης. Οι constructors μιας παράγωγης κλάσης θα πρέπει συνεπώς να παρέχουν τη δυνατότητα αρχικοποίησης και των μελών δεδομένων της βασικής κλάσης. Για το σκοπό αυτό είναι δυνατόν να χρησιμοποιηθεί στον constructor της παράγωγης κλάσης και ο constructor της βασικής κλάσης μέσω του διαχωριστικού στήλης, όπως παρακάτω:

```
derived::derived(int a, int b, int c):base(a, b)
{
   //...
}
```

όπου derived είναι ο constructor της παράγωγης κλάσης και δέχεται τρεις παραμέτρους εκ των οποίων οι δύο είναι παράμετροι που χρησιμοποιούνται για την αρχικοποίηση μελών δεδομένων της βασικής κλάσης, μέσω του constructor base της βασικής κλάσης.

Public, protected και private κληρονομικότητα

Κατά την παραγωγή μιας κλάσης από μία βασική, η βασική κλάση είναι δυνατόν να κληρονομηθεί ως public, protected ή private ανάλογα με τον προσδιοριστή που θα χρησιμοποιηθεί στον ορισμό της. Η χρήση των προσδιοριστών protected και private στην κληρονομικότητα είναι σπάνια και δεν θα εξετασθεί περαιτέρω.

Χρησιμοποιώντας τον προσδιοριστή public κατά την δημιουργία μιας παράγωγης κλάσης από μία βασική, όλα τα public μέλη της βασικής κλάσης καθίστανται public μέλη της παράγωγης κλάσης και τα protected μέλη της βασικής κλάσης, protected μέλη της παράγωγης κλάσης. Αντίθετα, τα private μέλη της βασικής κλάσης δεν είναι απευθείας προσπελάσιμα από την παράγωγη κλάση, αλλά είναι δυνατόν να προσπελασθούν μέσω κλήσεων των public και protected μελών της βασικής κλάσης.

Πολλαπλή Κληρονομικότητα

Μία κλάση είναι δυνατόν να παραχθεί από περισσότερες από μία βασικές κλάσεις. Η δυνατότητα μιας κλάσης να κληρονομήσει μέλη από διάφορες βασικές κλάσεις, ενώ παρέχει τη δυνατότητα χρήσιμων τεχνικών επαναχρησιμοποίησης λογισμικού, είναι συχνά αρκετά επικίνδυνη και μπορεί να δημιουργήσει προβλήματα ασάφειας.

Στο παράδειγμα που ακολουθεί η κλάση Derived κληρονομεί από δύο βασικές κλάσης και χρησιμοποιεί επιπλέον τη μέθοδο getReal για να διαβάσει το private μέλος δεδομένων double real.

```
#include <iostream.h>
```

```
using std::cout;
using std::endl;

class Base1{
public:
 Base1(int x) {value=x;}
 int getData() {return value;}
protected:
 int value;
};

class Base2{
public:
 Base2(char c) {letter=c;}
 char getData() {return letter;}
protected:
 char letter;
};

class Derived:public Base1, public Base2 {
public:
 Derived(int x, char c, double r):Base1(x),Base2(c) {real=r;}
 double getReal() {return real;}
protected:
 double real;
};

void main()
{
 Base1 b1(10);
 Base2 b2('Z');
 Derived d(7,'A',3.5);

cout<<"Object1 contains: "<<b1.getData()<<"\n";
cout<<"Object2 contains: "<<b2.getData()<<"\n";
cout<<"Der.object contains: Int :"<<d.Base1::getData()<<"\n"
      <<"                  Char :"<<d.Base2::getData()<<"\n"
      <<"               Double  :"<<d.getReal()<<"\n";
}
```

Από το παραπάνω παράδειγμα παρατηρείται ότι η πολλαπλή κληρονομικότητα είναι δυνατόν να δηλωθεί πολύ εύκολα εισάγοντας μετά το διαχωριστικό στήλης στον ορισμό της παράγωγης κλάσης, τη λίστα των βασικών κλάσεων διαχωρισμένες με κόμμα. Επίσης φαίνεται ότι ο constructor της παράγωγης κλάσης καλεί τους constructors των δύο βασικών κλάσεων και αρχικοποιεί επιπλέον το μέλος δεδομένων της κλάσης όπου ανήκει.
Στη συνάρτηση main υλοποιούνται τρία αντικείμενα, ένα τύπου Base1, ένα τύπου Base2 και ένα αντικείμενο d της παράγωγης κλάσης Derived. Στη συνέχεια για την εκτύπωση των περιεχομένων των μελών δεδομένων των δύο πρώτων αντικειμένων καλείται η συνάρτηση getData. Παρόλο που υπάρχουν δύο συναρτήσεις getData (μία ορισμένη στην κλάση Base1 και

μια στην κλάση Base2), δεν προκαλείται πρόβλημα ασάφειας γιατί καλούνται μέσω του αντίστοιχου αντικειμένου.
Κατά την εκτύπωση των μελών δεδομένων του αντικειμένου d, υπάρχει πρόβλημα ασάφειας γιατί αυτό κληρονομεί δύο συναρτήσεις getData, μία από κάθε βασική κλάση. Το πρόβλημα επιλύεται με χρήση του ονόματος της αντίστοιχης βασικής κλάσης, του τελεστή διάκρισης και της μεθόδου που καλείται. Το μέλος δεδομένων real της παράγωγης κλάσης εκτυπώνεται χωρίς πρόβλημα με κλήση της μοναδικά ορισμένης μεθόδου getReal.

ΑΝΤΙΚΕΙΜΕΝΟΣΤΡΕΦΗΣ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ: ΠΟΛΥΜΟΡΦΙΣΜΟΣ

Με τη χρήση virtual (υπερβατών) συναρτήσεων και με τον πολυμορφισμό είναι δυνατόν να σχεδιάσουμε προγράμματα τα οποία είναι εύκολα επεκτάσιμα.

Virtual Συναρτήσεις

Θεωρούμε ένα σύνολο από κλάσεις σχημάτων, τις **Circle, Triangle, Rectangle, Square** κτλ. Στην αντικειμενοστρεφή σχεδίαση είναι δυνατόν κάθε κλάση να είναι εφοδιασμένη με τη δυνατότητα να σχεδιάζει το αντίστοιχο σχήμα. Παρόλο που κάθε κλάση μπορεί να έχει μία τέτοια συνάρτηση σχεδίασης (**draw**), η συνάρτηση αυτή είναι διαφορετική για κάθε σχήμα. Θα ήταν ιδανικό αν θα μπορούσαμε να καλούμε τη συνάρτηση draw μιας βασικής κλάσης και να αφήνουμε στο πρόγραμμα να αποφασίζει δυναμικά (κατά την εκτέλεση του προγράμματος) ποια συνάρτηση draw από τις παράγωγες κλάσεις να χρησιμοποιήσει.
Για να καταστήσουμε δυνατή αυτή την συμπεριφορά, δηλώνουμε τη συνάρτηση **draw** στην βασική κλάση ως υπερβατή **(virtual)** και επικαλύπτουμε τη συνάρτηση σε κάθε μία από τις παράγωγες κλάσεις για τη σχεδίαση του αντίστοιχου σχήματος. Μία υπερβατή συνάρτηση δηλώνεται εισάγοντας στο πρωτότυπο της συνάρτησης τον όρο **virtual**, για παράδειγμα:

```
virtual void draw();
```

Όταν μία συνάρτηση δηλώνεται υπερβατή, παραμένει υπερβατή σε όλη την ιεραρχία των παράγωγων κλάσεων, ακόμα και όταν αυτό δεν δηλώνεται σαφώς κατά την επικάλυψη της σε μία κλάση.

Πολυμορφισμός

Η C++ επιτρέπει τον πολυμορφισμό, δηλαδή τη δυνατότητα σε αντικείμενα διαφορετικών κλάσεων που σχετίζονται με κληρονομικότητα, **να αποκρίνονται με διαφορετικό τρόπο στο ίδιο μήνυμα** (π.χ. σε μία κλήση συνάρτησης). Για παράδειγμα, για τις κλάσεις σχημάτων της προηγούμενης παραγράφου, η κλήση μιας συνάρτησης για τον υπολογισμό του εμβαδού θα πρέπει να υλοποιείται με διαφορετικό τρόπο για κάθε σχήμα.
Ο πολυμορφισμός υλοποιείται με υπερβατές συναρτήσεις. Όταν μία αίτηση πραγματοποιείται μέσω ενός δείκτη (ή αναφοράς) της βασικής κλάσης για τη χρήση μιας υπερβατής συνάρτησης, η C++ επιλέγει τη σωστή επικαλυπτόμενη συνάρτηση στην κατάλληλη παράγωγη κλάση.
Αν μια συνάρτηση δεν είναι δηλωμένη ως υπερβατή και επικαλύπτεται σε μία παράγωγη κλάση, τότε κατά την κλήση μιας τέτοιας συνάρτησης μέσω ενός δείκτη της βασικής κλάσης, χρησιμοποιείται η συνάρτηση της βασικής κλάσης. Αν χρησιμοποιηθεί ο δείκτης της παράγωγης κλάσης καλείται η συνάρτηση της παράγωγης κλάσης. Η συμπεριφορά αυτή είναι μή-πολυμορφική.
Έστω για παράδειγμα η βασική κλάση Car και η παράγωγη της Sportscar:

```
#include <iostream.h>

using std::cout;
```

```
using std::endl;

class Car
{
public:
 Car(char *, int);        //Constructor
 void print();       //print type and maxspeed
 ~Car();             //destructor
private:
 char *type;              //dynamically allocated string
 int maxspeed;
};

class Sportscar:public Car
{
public:
 Sportscar(char *, int, int);
 void print();            //overriden base class print
private:
 int horsepower;
};

Car::Car(char *typeName, int topspeed)
{
 type=new char[strlen(typeName)+1];
 strcpy(type, typeName);
 maxspeed=topspeed;
}

Car::~Car()
{
 delete [] type;          //free dynamic memory
}

void Car::print()         //print a car's type and max.speed
{
 cout<<'\n'<<type<<" max.speed= "<<maxspeed;
}

Sportscar::Sportscar(char *typeName, int topspeed, int horsepwr)
        :Car(typeName, topspeed)  //call base class constructor
{
 horsepower=horsepwr;
}

void Sportscar::print()   //print type, speed, horsepower
{
 Car::print();    //call base class print function
 cout<<" horsepower = "<<horsepower;
}
```

```
void main()
{
 Car c("Volvo", 220);
 Sportscar sc("Porsche", 350, 400);
 Car *cPtr=&c;
 Sportscar *scPtr=&sc;
 cPtr->print();      //call base class print function
 scPtr->print();          //call derived class print function
 cPtr=&sc;           //Allowable conversion
 cPtr->print();      //unfortunately calls base class print
}
```

Το ανωτέρω πρόγραμμα τυπώνει:

```
Volvo max.speed= 220
Porsche max.speed= 350 horsepower= 400
Porsche max.speed= 350
```

Η βασική κλάση και η παράγωγή της έχουν ορίσει τη δική τους συνάρτηση εκτύπωσης. Επειδή οι συναρτήσεις δεν δηλώθηκαν virtual και έχουν το ίδιο όνομα, η κλήση της συνάρτησης print μέσω ενός pointer της βασικής κλάσης αντιστοιχεί στην κλήση της Car::print(), ανεξαρτήτως του αν ο pointer δείχνει σε αντικείμενο της βασικής ή της παράγωγης κλάσης.

Αν η συνάρτηση print δηλωθεί virtual στην βασική κλάση :

```
class Car
{
public:
     Car(char *, int);
     virtual void print();    //virtual
     ~Car();
private:
     char *type;
     int maxspeed;
};
```

τότε εκτυπώνεται το ορθό:

```
Volvo max.speed= 220
Porsche max.speed= 350 horsepower= 400
Porsche max.speed= 350 horsepower= 400
```

δηλαδή η κλήση της συνάρτησης print μέσω ενός Pointer της βασικής κλάσης ο οποίος δείχνει σε αντικείμενο τύπου Sportscar, καλεί τη συνάρτηση της παράγωγης κλάσης, ακριβώς διότι η συνάρτηση print δηλώθηκε **virtual**.

Αφηρημένες Κλάσεις Βάσης και Συγκεκριμένες Κλάσεις

Όταν θεωρούμε μία κλάση ως τύπο δεδομένων, υποθέτουμε ότι στη συνέχεια θα υλοποιήσουμε αντικείμενα της κλάσης αυτής. Ωστόσο, υπάρχουν περιπτώσεις όπου είναι χρήσιμο να μην υλοποιηθούν αντικείμενα από μία κλάση. Οι κλάσεις αυτές ονομάζονται **αφηρημένες**. Ο μοναδικός σκοπός ύπαρξης τους είναι να παρέχουν μία κατάλληλη κλάση βάσης από την οποία

οι παράγωγες κλάσεις θα υιοθετήσουν χαρακτηριστικά και συμπεριφορά. Οι κλάσεις από τις οποίες υλοποιούνται αντικείμενα ονομάζονται **συγκεκριμένες**.
Μία κλάση καθίσταται αφηρημένη, δηλώνοντας μία ή περισσότερες από τις υπερβατές συναρτήσεις ως **αμιγή** (pure). Μία αμιγής υπερβατή συνάρτηση είναι αυτή που αρχικοποιείται στη δήλωση της ως ίση με 0, π.χ. :

```
virtual double salary () = 0; //pure virtual
```

Θα πρέπει να σημειωθεί ότι για μία αμιγή υπερβατή συνάρτηση δεν υπάρχει ορισμός.

Θεωρούμε το ακόλουθο παράδειγμα όπου υλοποιούμε μία αφηρημένη κλάση βάσης Employee που αναφέρεται στην γενικευμένη έννοια ενός εργαζομένου σε μία επιχείρηση. Από αυτήν παράγονται δύο κλάσεις, η κλάση HourlyWorker που αναφέρεται σε στελέχη τα οποία πληρώνονται με μισθό ανάλογο των ωρών εργασίας τους ανά εβδομάδα και η κλάση PieceWorker η οποία αναφέρεται σε εργαζόμενους που πληρώνονται ανάλογα με τον αριθμό τεμαχίων που παράγουν ανά εβδομάδα.
Μία συνάρτηση υπολογισμού του μηνιαίου μισθού των εργαζομένων (salary) προφανώς εφαρμόζεται σε όλους τους εργαζόμενους, ωστόσο ο τρόπος υπολογισμού διαφέρει για κάθε κλάση εργαζομένων. Για τον λόγο αυτό η συνάρτηση salary δηλώνεται αμιγής υπερβατή στην βασική κλάση, και κατάλληλες υλοποιήσεις της salary παρέχονται για κάθε μία από τις παράγωγες κλάσεις. Στη συνέχεια, για τον υπολογισμό του μισθού το πρόγραμμα χρησιμοποιεί έναν pointer της βασικής κλάσης (ή μία αναφορά) προς το αντίστοιχο αντικείμενο του εργαζομένου και καλεί τη συνάρτηση salary. Σε ένα πραγματικό σύστημα, τα διάφορα αντικείμενα εργαζομένων μπορεί να δεικτοδοτούνται από έναν πίνακα (λίστα) δεικτών τύπου Employee *. Το πρόγραμμα απλά θα σάρωνε τον πίνακα και μέσω των δεικτών Employee * θα καλούσε την κατάλληλη συνάρτηση salary για κάθε εργαζόμενο.

```
#include <iostream.h>

using std::cout;
using std::endl;

class Employee           //Αφηρημένη Κλάση Βάσης
{
public:
     Employee(const char *, const char *);
     ~Employee();
     const char *getFirstName();
     const char *getLastName();

     virtual double salary() = 0;  //αμιγής υπερβατή
     virtual void print();
private:
     char *firstName;
     char *lastName;
};

Employee::Employee(const char *first, const char *last)
{
     firstName=new char[strlen(first)+1];
     strcpy(firstName, first);
```

```
     lastName=new char[strlen(last)+1];
     strcpy(lastName, last);
}

Employee::~Employee()
{
     delete [] firstName;
     delete [] lastName;
}

//Ο προσδιοριστής const δεν επιτρέπει στο πρόγραμμα που καλεί
//την συνάρτηση να τροποποιήσει private δεδομένα
const char *Employee::getFirstName()
{
     return firstName;
}

const char *Employee::getLastName()
{
     return lastName;
}

void Employee::print()
{
     cout<<firstName<<' '<<lastName;
}

class HourlyWorker:public Employee
{
public:
     HourlyWorker(const char *,const char *,double=0.0, double=0.0);
     void setWage(double);
     void setHours(double);
     virtual double salary();
     virtual void print();
private:
     double wage;
     double hours;
};

HourlyWorker::HourlyWorker(const char *first, const char *last,
                           double w, double h)
                           : Employee(first, last)
{
     setWage(w);
     setHours(h);
}

void HourlyWorker::setWage(double w)
{
```

```
     wage = w>0 ? w : 0;
}

void HourlyWorker::setHours(double h)
{
     hours= h>=0 && h<168 ? h : 0;
}

double HourlyWorker::salary()
{
     if(hours<=40)            //χωρίς υπερωρία
          return wage * hours;
     else                 //η υπερωρία πληρώνεται παραπάνω !!!
          return 40 * wage + (hours-40) * wage * 1.5;
}

void HourlyWorker::print()
{
     cout<<"\n    Hourly worker: ";
     Employee::print();
}

class PieceWorker:public Employee
{
public:
     PieceWorker(const char *, const char *, double=0.0, int=0);
     void setWage(double);
     void setQuantity(int);
     virtual double salary();
     virtual void print();
private:
     double wage;
     int quantity;
};

PieceWorker::PieceWorker(const char *first, const char *last,
                              double w, int q)
                              : Employee(first, last)
{
     setWage(w);
     setQuantity(q);
}

void PieceWorker::setWage(double w)
{
     wage = w>0 ? w : 0;
}

void PieceWorker::setQuantity(int q)
{
     quantity= q>=0 ? q : 0;
```

```
}

double PieceWorker::salary()
{
     return quantity * wage;
}

void PieceWorker::print()
{
     cout<<"\n    Piece worker: ";
     Employee::print();
}

void virtualViaPointer(Employee *);
void virtualViaReference(Employee &);

void main()
{
     HourlyWorker h("John", "Smith", 13.75, 40);
     h.print();                              //static binding
     cout<<" earned $"<<h.salary();          //static binding
     virtualViaPointer(&h);                  //dynamic binding
     virtualViaReference(h);                 //dynamic binding

     PieceWorker p("Bob", "Lewis", 2.5, 200);
     p.print();                              //static binding
     cout<<" earned $"<<p.salary();          //static binding
     virtualViaPointer(&p);                  //dynamic binding
     virtualViaReference(p);                 //dynamic binding
     cout<<endl;
}

void virtualViaPointer(Employee *baseClassPtr)
{
     baseClassPtr->print();
     cout<<" earned $"<<baseClassPtr->salary();
}

void virtualViaReference(Employee &baseClassRef)
{
     baseClassRef.print();
     cout<<" earned $"<<baseClassRef.salary();
}
```

Το ανωτέρω πρόγραμμα τυπώνει:

```
Hourly worker: John Smith earned $550
Hourly worker: John Smith earned $550
Hourly worker: John Smith earned $550
Piece worker: Bob Lewis earned $500
```

```
Piece worker: Bob Lewis earned $500
Piece worker: Bob Lewis earned $500
```

Το ερώτημα είναι γιατί η συνάρτηση salary στην βασική κλάση δηλώθηκε ως αμιγής υπερβατή. Η απάντηση είναι ότι δεν είναι λογικό να παρέχουμε υλοποίηση της συνάρτησης αυτής, δηλαδή τρόπο υπολογισμού του μισθού για την γενικευμένη έννοια ενός υπαλλήλου - θα πρέπει πρώτα να γνωρίζουμε τι είδους υπάλληλος είναι. Δηλώνοντας την συνάρτηση pure virtual υποδηλώνουμε ότι θα υπάρξει υλοποίηση για την συνάρτηση αυτή σε κάθε παράγωγη κλάση αλλά όχι στην ίδια την βασική κλάση.
Στην συνάρτηση main η γραμμή

```
HourlyWorker h("John", "Smith", 13.75, 40);
```

υλοποιεί ένα αντικείμενο h της κλάσης HourlyWorker και παρέχει τις παραμέτρους προς τον constructor, ήτοι το όνομα και επίθετο του εργαζομένου, την αμοιβή ανά ώρα και τον αριθμό ωρών εργασίας που πραγματοποίησε σε μία εβδομάδα.
Η γραμμή

```
h.print();                              //static binding
```

καλεί απευθείας τη συνάρτηση print της κλάσης HourlyWorker χρησιμοποιώντας τον τελεστή διαχωρισμού (.) και το συγκεκριμένο αντικείμενο h. Αυτή είναι μία περίπτωση **στατικής σύνδεσης** (static binding) καθώς ο τύπος του αντικειμένου για το οποίο καλείται η συνάρτηση είναι γνωστός κατά τη διάρκεια της μεταγλώττισης του προγράμματος (at compile time).
Στη συνέχεια η γραμμή

```
cout<<" earned $"<<h.salary();          //static binding
```

καλεί κατά τον ίδιο τρόπο (στατική σύνδεση) τη συνάρτηση salary της κλάσης HourlyWorker.

Αντίθετα, η γραμμή

```
virtualViaPointer(&h);                  //dynamic binding
```

καλεί τη συνάρτηση virtualViaPointer με παράμετρο τη διεύθυνση του αντικειμένου h της παράγωγης κλάσης HourlyWorker. Η συνάρτηση λαμβάνει τη διεύθυνση αυτή στην παράμετρο baseClassPtr, που είναι ορισμένη ως δείκτης σε αντικείμενο τύπου Employee (Employee *). Η περίπτωση αυτή αποτελεί κλασσική εφαρμογή πολυμορφικής συμπεριφοράς.
Η γραμμή

```
baseClassPtr->print();
```

της συνάρτησης virtualViaPointer καλεί τη μέθοδο print του αντικειμένου στο οποίο "δείχνει" ο pointer baseClassPtr. Επειδή η μέθοδος print είναι δηλωμένη virtual στην βασική κλάση, το σύστημα καλεί την μέθοδο print της παράγωγης κλάσης (ακριβώς ότι ονομάζεται πολυμορφική συμπεριφορά). Η κλήση αυτή της συνάρτησης είναι παράδειγμα **δυναμικής σύνδεσης** (dynamic binding), καθώς η υπερβατή συνάρτηση καλείται μέσω pointer της βασικής κλάσης και κατά συνέπεια η απόφαση για το ποιά συνάρτηση θα κληθεί λαμβάνεται κατά τη διάρκεια εκτέλεσης του προγράμματος (at run time).
Η γραμμή

```
cout<<" earned $"<<baseClassPtr->salary();
```

καλεί τη μέθοδο salary του αντικειμένου στο οποίο δείχνει ο pointer baseClassPtr. Επειδή και σε αυτή την περίπτωση η μέθοδος salary είναι δηλωμένη ως virtual στην βασική κλάση, το σύστημα καλεί την μέθοδο salary του αντικειμένου της παράγωγης κλάσης. Και αυτή η περίπτωση αποτελεί παράδειγμα δυναμικής σύνδεσης.
Η συνάρτηση virtualViaReference παρουσιάζει τη δυνατότητα πολυμορφικής συμπεριφοράς με χρήση υπερβατών συναρτήσεων που καλούνται από αναφορές της βασικής κλάσης.

ΔΟΜΕΣ ΔΕΔΟΜΕΝΩΝ

Στο κεφάλαιο αυτό εξετάζονται δυνατότητες υλοποίησης δυναμικών δομών δεδομένων των οποίων το μέγεθος μεταβάλλεται κατά τη διάρκεια της εκτέλεσης των προγραμμάτων που τις χρησιμοποιούν. Τέτοιες δομές είναι οι διασυνδεδεμένες λίστες (linked lists), οι στοίβες (stacks), οι ουρές (queues) και τα δυαδικά δέντρα (binary trees).

Αυτοαναφορικές κλάσεις

Μία αυτοαναφορική κλάση περιέχει έναν pointer ο οποίος "δείχνει" προς ένα αντικείμενο της ιδίας κλάσης. Για παράδειγμα η κλάση :

```
class Node {
public:
  Node(int);
  void setData(int);
  int getData() const;
  void setNextPtr(Node *);
  const Node *getNextPtr() const;
private:
  int data;
  Node *nextPtr;
};
```

δηλώνει ένα **Κόμβο** με δύο private μέλη δεδομένων (έναν ακέραιο και έναν pointer ο οποίος "δείχνει" προς αντικείμενο της ιδίας κλάσης, εξ'ου και η ονομασία αυτοαναφορική κλάση). Το μέλος nextPtr εξασφαλίζει τη "σύνδεση" μεταξύ δύο αντικειμένων ιδίου τύπου. Η κλάση έχει επίσης πέντε μεθόδους, έναν constructor με παράμετρο έναν ακέραιο για την αρχικοποίηση του μέλους δεδομένων data, μία συνάρτηση setData για τον καθορισμό της τιμής data, μία συνάρτηση getData για την επιστροφή της τιμής data, μία συνάρτηση setNextPtr για τον καθορισμό της τιμής του pointer nextPtr και τη συνάρτηση getNextPtr για την επιστροφή της τιμής του pointer nextPtr.
Η δυνατότητα διασύνδεσης των αντικειμένων μιας αυτοαναφορικής κλάσης παρέχει τη δυνατότητα υλοποίησης των διαφόρων δομών δεδομένων.

Δυναμική Διαχείριση Μνήμης

Η υλοποίηση και συντήρηση δομών δεδομένων απαιτεί δυναμική διαχείριση μνήμης (dynamic memory allocation), δηλαδή τη δυνατότητα από το πρόγραμμα να δεσμεύει κατά τη διάρκεια της εκτέλεσης επιπρόσθετο χώρο μνήμης για νέους κόμβους και τη δυνατότητα να απελευθερώνει χώρο μνήμης όταν πλέον ένας κόμβος δεν είναι απαραίτητος.
Οι τελεστές **new** και **delete** της C++ παρέχουν τη δυνατότητα δυναμικής διαχείρισης μνήμης. Ο τελεστής new λαμβάνει ως παράμετρο τον τύπο του αντικειμένου για το οποίο πρόκειται να δεσμευτεί μνήμη και επιστρέφει έναν pointer προς αντικείμενο αυτού του τύπου. Για παράδειγμα η εντολή :

```
Node *newPtr = new Node(10);
```

δεσμεύει αριθμό από bytes ίσο με sizeof(Node), καλεί τον constructor Node με τιμή 10 για το μέλος data και αποθηκεύει **τον pointer προς τη θέση μνήμης που δεσμεύτηκε** στην μεταβλητή pointer newPtr.
Ο τελεστής delete καλεί τον destructor της κλάσης Node και αποδεσμεύει το χώρο μνήμης που δεσμεύτηκε με τον τελεστή new. Για παράδειγμα η εντολή :

```
delete newPtr;
```

απελευθερώνει τη μνήμη που δεσμεύθηκε με την προηγούμενη εντολή. Η μνήμη που απελευθερώθηκε είναι πλέον διαθέσιμη στο σύστημα και είναι δυνατόν να επαναδεσμευτεί στο μέλλον. Σημειώνεται ότι η μεταβλητή newPtr δεν διαγράφεται, μόνο η θέση μνήμης προς την οποία δείχνει ο pointer απελευθερώνεται. Η μεταβλητή συνεχίζει να υπάρχει με τιμή 0 (ο pointer δεν "δείχνει" πουθενά).