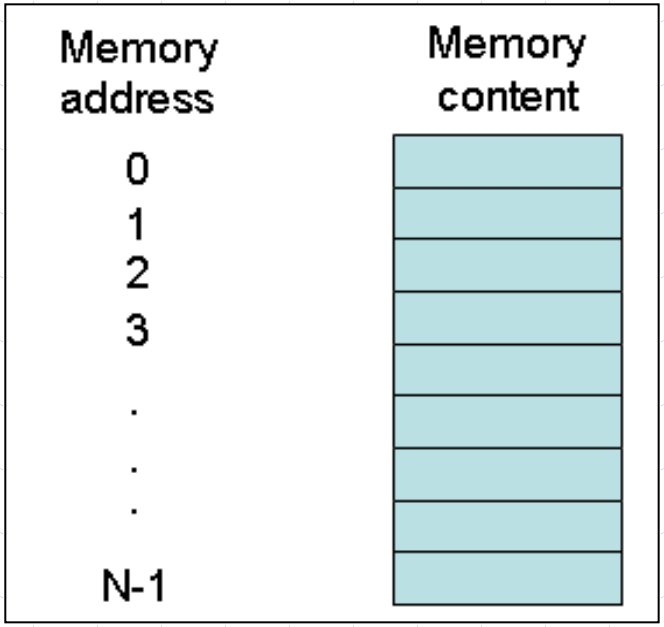
Προγραμματισμός Υπολογιστών C++

Δείκτες
Αλφαριθμητικά

Μνήμη Υπολογιστή

- Η μνήμη RAM (Random Access Memory) ενός υπολογιστή αποτελείται από εκατομμύρια αριθμημένες θέσεις μνήμης με **διαδοχική αρίθμηση**
- Κάθε θέση μπορεί να αποθηκεύσει οκτώ bits πληροφορίας
- Κάθε θέση προσδιορίζεται από έναν μοναδικό αριθμό που ονομάζεται **διεύθυνση** μνήμης
- Π.χ. σε έναν υπολογιστή με **N** θέσεις (οκτάδες) μνήμης η διεύθυνση της κάθε θέσης είναι ένας αύξοντας αριθμός από **0** έως **N-1**



Μνήμη Υπολογιστή και Μεταβλητές

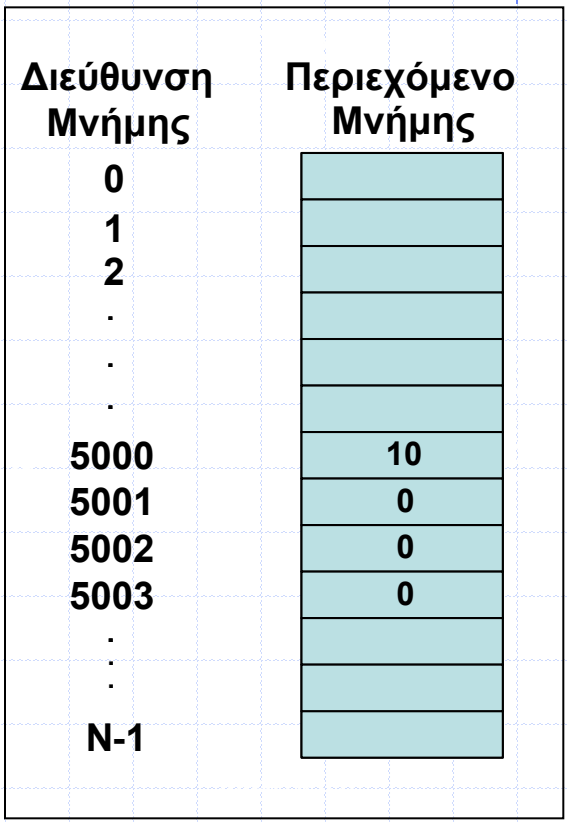
- Όταν δηλώνεται μία μεταβλητή, ο μεταγλωττιστής δεσμεύει τις απαραίτητες **συνεχόμενες** θέσεις (bytes) στη μνήμη, για να αποθηκεύσει την τιμή της

- Όπως ήδη ξέρουμε, κάθε τύπος μεταβλητής απαιτεί συγκεκριμένο χώρο στη μνήμη

- Όταν μία μεταβλητή καταλαμβάνει πολλές θέσεις μνήμης (δηλ. περισσότερα από ένα byte), τότε ως **διεύθυνση της μεταβλητής** θεωρείται **η διεύθυνση της πρώτης θέσης μνήμης** (δηλ. του πρώτου byte από τα bytes που καταλαμβάνει η μεταβλητή)

Παράδειγμα

- Έστω η δήλωση: `int a = 10;`
- Τότε: Ο μεταγλωττιστής ψάχνει και βρίσκει 4 συνεχόμενες θέσεις μνήμης στη RAM, οι οποίες δεν πρέπει να έχουν δεσμευτεί για άλλη μεταβλητή, και τις δεσμεύει για να αποθηκεύσει την τιμή της μεταβλητής `a`
- Στο διπλανό σχήμα θεωρούμε ότι η διεύθυνση της μεταβλητής `a` αρχίζει στη θέση 5000
- Έτσι, η τιμή της `a` (η τιμή 10) θα αποθηκευτεί στις θέσεις μνήμης από 5000 έως και 5003,θεωρώντας ότι η χαμηλότερη οκτάδα της τιμής θα αποθηκευτεί στη χαμηλότερη διεύθυνση (*little-endian* αρχιτεκτονική)
- Ο μεταγλωττιστής συσχετίζει το όνομα κάθε μεταβλητής με τη διεύθυνσή της και αποθηκεύει αυτές τις αντιστοιχίσεις
- Όταν μία μεταβλητή χρησιμοποιείται στο πρόγραμμα, ο μεταγλωττιστής ανακτά τη διεύθυνσή της
- Π.χ. με την εντολή `a = 80;` ο μεταγλωττιστής γνωρίζει ότι η διεύθυνση της `a` είναι η 5000 και θέτει το περιεχόμενό της ίσο με 80



Δήλωση Δείκτη (1)

- Ο **δείκτης** είναι μία **μεταβλητή**, στην οποία μπορεί να αποθηκευτεί μία διεύθυνση μνήμης, όπως, για παράδειγμα, η διεύθυνση μνήμης κάποιας άλλης μεταβλητής

- Για να δηλώσουμε ένα δείκτη γράφουμε:

```
τύπος_δεδομένων *όνομα_δείκτη;
```

- Π.χ. με τη δήλωση `int *ptr;` η μεταβλητή `ptr` είναι δείκτης σε τύπο `int`. Επομένως, στον `ptr` μπορεί να εκχωρηθεί η διεύθυνση κάποιας `int` μεταβλητής
- Γενικά, αν ο `ptr` είναι «δείκτης σε τύπο» `T`, τότε η έκφραση `*ptr` είναι τύπου `T`. Π.χ. στην προηγούμενη δήλωση (`int *ptr;`), η έκφραση `*ptr` είναι τύπου `int`

Δήλωση Δείκτη (2)

- Σημειώστε ότι το * επιτρέπεται να τοποθετηθεί δίπλα στον τύπο (π.χ. `int* ptr`)
- Αν και αρκετοί προγραμματιστές προτιμούν την παραπάνω γραφή, άλλοι προτιμούν την πρώτη σύνταξη γιατί μπορεί να δημιουργηθεί σύγχυση σε μία πολλαπλή δήλωση μεταβλητών. Π.χ. με τη δήλωση:

```
int* p1, p2;
```

μπορεί να δημιουργηθεί η σύγχυση ότι και το `p2` δηλώνεται ως «δείκτης σε ακέραιο» ενώ αυτό είναι ακέραιος. Η δήλωση:

```
int *p1, p2;
```

είναι πιο ξεκάθαρη

Δήλωση Δείκτη (3)

- Όπως και με τις απλές μεταβλητές, όταν δηλώνεται ένας δείκτης, ο μεταγλωττιστής δεσμεύει μνήμη για να αποθηκεύσει την τιμή του
- Οι οκτάδες μνήμης που δεσμεύονται για έναν δείκτη είναι πάντα ίδιες, ανεξάρτητα από τον τύπο δεδομένων στον οποίο δείχνει ο δείκτης
- Αυτή η τιμή εξαρτάται από την αρχιτεκτονική του συστήματος. Συνήθως, είναι τέσσερις ή οκτώ (για 32-bit και 64-bit συστήματα, αντίστοιχα)
- Δηλαδή, είτε γράψουμε `char *ptr;` είτε: `float *ptr;` είτε: `double *ptr;` ο μεταγλωττιστής δεσμεύει τον ίδιο αριθμό οκτάδων για τον `ptr` (π.χ. 4)
- Αν θέλετε να βρείτε το μέγεθος μίας μεταβλητής δείκτη στο σύστημά σας, χρησιμοποιήστε τον τελεστή `sizeof` (π.χ. `cout << sizeof(ptr)`

Αρχικοποίηση Δείκτη (1)

- Αφού δηλώσουμε έναν δείκτη μπορούμε να του εκχωρήσουμε τη διεύθυνση μνήμης κάποιας μεταβλητής
- Για να βρούμε τη διεύθυνση κάποιας μεταβλητής χρησιμοποιούμε τον τελεστή διεύθυνσης **&** πριν από το όνομα της μεταβλητής
- Υπενθυμίζεται ότι η διεύθυνση είναι η θέση της μεταβλητής στη μνήμη και δεν έχει καμία σχέση με την τιμή της μεταβλητής. Π.χ.

```
#include <iostream>
int main()
{
   int *ptr, a;
   ptr = &a;
   std::cout << ptr << ' ' << &ptr << '\n';
   return 0;
}
```

- Με την εντολή `ptr = &a;` η τιμή του δείκτη `ptr` γίνεται ίση με τη διεύθυνση μνήμης της μεταβλητής `a`, συνηθίζεται να λέμε ότι ο `ptr` «δείχνει» στο `a`. Όπως είπαμε, ο μεταγλωττιστής δεσμεύει μνήμη για να αποθηκεύσει την τιμή του `ptr`. Το πρόγραμμα εμφανίζει την διεύθυνση μνήμης του `a` και του `ptr`
- Όπως βλέπετε, χρησιμοποιώντας δείκτες είμαστε πολύ κοντά στο υλικό και τη μνήμη του συστήματος. Γενικά, αν ο τύπος είναι *T* ο τύπος της έκφρασης *&T* είναι «δείκτης σε *T*»

Αρχικοποίηση Δείκτη (2)

- Όταν εκχωρείται η διεύθυνση μίας μεταβλητής σε έναν δείκτη, **ο δείκτης πρέπει να έχει δηλωθεί** σαν δείκτης **στον ίδιο τύπο** με τη **μεταβλητή**. Π.χ.

```
int *ptr;
float a;
ptr = &a; // Λάθος
```

- Η ανάθεση μίας ακέραιας τιμής σε έναν δείκτη είναι πολύ πιθανό να προκαλέσει σφάλμα μεταγλώττισης. Αυτό είναι λογικό να συμβεί, αφού ο δείκτης δεν είναι ακέραιος. Π.χ.

```
int *ptr;
ptr = 1000;
```

Ωστόσο, σε εφαρμογές που χρειάζεται να προσπελάσουν διευθύνσεις μνήμης του υλικού του συστήματος (π.χ. σε ενσωματωμένα συστήματα), η ανάθεση μπορεί να επιτραπεί με προσαρμογή τύπου

Μηδενικοί Δείκτες

- Όπως και με μία απλή μεταβλητή, η αρχική τιμή μίας μεταβλητής δείκτη είναι μία **τυχαία τιμή («σκουπίδι»)**
- Όταν θέλουμε να δηλώσουμε **ρητά** ότι ένας δείκτης **δεν δείχνει πουθενά**, του αναθέτουμε την τιμή `nullptr` (εναλλακτικά την τιμή `0` ή την τιμή `NULL`). Ένας τέτοιος δείκτης ονομάζεται **μηδενικός**
- Επειδή η τιμή `nullptr` συνδέεται αποκλειστικά με δείκτες, συστήνεται η χρήση της έναντι των ακεραίων σταθερών `NULL` και `0`. Π.χ.

```
int *ptr;
ptr = nullptr;
```

- Η τιμή ενός δείκτη μπορεί να συγκριθεί με τις παραπάνω σταθερές, όπως παρακάτω:

```
if(ptr != nullptr) /* Ισοδύναμο με if(ptr) */
if(ptr == nullptr) /* Ισοδύναμο με if(!ptr) */
```

Χρήση Δείκτη (1)

- Για να προσπελάσουμε το περιεχόμενο της μνήμης στην οποία δείχνει κάποιος δείκτης χρησιμοποιούμε τον τελεστή αποαναφοροποίησης ή έμμεσης αναφοράς * (dereference ή indirection operator) πριν από το όνομα του δείκτη. Π.χ.

```
#include <iostream>
int main()
{
        int *ptr, a = 10;

        ptr = &a;
        std::cout << *ptr << '\n';
        *ptr = 20; // Ισοδύναμο με a = 20.
        std::cout << a << '\n';
        return 0;
}
```

- Η έκφραση *ptr ισοδυναμεί με το περιεχόμενο της διεύθυνσης στην οποία δείχνει ο δείκτης ptr
- Αφού ο ptr δείχνει στη διεύθυνση του a, το *ptr είναι ίσο με το a, οπότε, το πρόγραμμα εμφανίζει πρώτα 10 και μετά 20

Χρήση Δείκτη (2)

- **Προσοχή**. Πριν αποαναφοροποιηθεί ένας δείκτης πρέπει να δείχνει σε κάποια έγκυρη διεύθυνση, όπως η διεύθυνση κάποιας μεταβλητής
- Για παράδειγμα, το επόμενο πρόγραμμα θα εμφανίσει μήνυμα λάθους κατά την εκτέλεσή του, γιατί ο δείκτης `ptr` δεν έχει αρχικοποιηθεί πριν χρησιμοποιηθεί στην εντολή `a = *ptr;`

```
#include <iostream>
int main()
{
        int *ptr, a, b = 10;

        a = *ptr; // Λάθος
        std::cout << a << '\n';
        return 0;
}
```

- Αν, για παράδειγμα, είχε προηγηθεί η εντολή `ptr = &b;` το `a` θα γινόταν ίσο με το `b` και το πρόγραμμα θα εμφάνιζε `10`
- Συνήθως, σε Unix/Linux περιβάλλον το παραπάνω λάθος υποδεικνύεται με το μήνυμα *"Segmentation fault"*

Παράδειγμα (1)

- Ποια είναι η έξοδος των παρακάτω προγραμμάτων;

```
#include <iostream>
int main()
{
  int *ptr, i = 0;

  ptr = &i;
  *ptr = 2;
  for(; i < 5; i++)
            std::cout << *ptr << ' ';
  return 0;
}
```

- ```
  #include <iostream>
  int main()
  {
    int *ptr1, *ptr2, *ptr3, i = 10, j = 20, k = 30;

    ptr1 = &i;
    i = 100;

    ptr2 = &j;
    j = *ptr2 + *ptr1;

    ptr3 = &k;
    k = *ptr3 + *ptr2;
    std::cout << *ptr1 << ' ' << *ptr2 << ' ' << *ptr3;
    return 0;
  }
  ```
  ```

Παράδειγμα (1)

```
#include <iostream>
int main()
{
  int *ptr1, *ptr2, *ptr3, i = 10, j = 20, k = 30;

  ptr1 = &i;
  ptr2 = &j;
  ptr3 = &k;
  *ptr1 = *ptr2 = 100;
  k = i+j;
  std::cout << *ptr3 << '\n';
  return 0;
}
```

```
#include <iostream>
int main()
{
  int *ptr1, *ptr2, i = 10, j = 20;

  ptr1 = &i;
  ptr2 = &j;

  ptr2 = ptr1;
  *ptr1 = *ptr1 + *ptr2;
  *ptr2 *= 2;
  std::cout << *ptr1 + *ptr2 << '\n';

  return 0;
}
```

Παράδειγμα (1)

Έξοδος:

```
2  3  4
100 120 150
200
80
```

Παράδειγμα (2)

- Να συμπληρώσετε το παρακάτω πρόγραμμα χρησιμοποιώντας τον δείκτη `p` και έναν `while` βρόχο, ώστε να εμφανίζει τους ακεραίους από το `1` έως και το `10`. Η μεταβλητή `i` να χρησιμοποιηθεί μόνο μία φορά

```
#include <iostream>
int main()
{
 int *p, i;
 ...
 return 0;
}
```

Παράδειγμα (2)

```
#include <iostream>
int main()
{
 int *p, i;

 p = &i; /* Πριν χρησιμοποιήσουμε τον δείκτη, πρέπει οπωσδήποτε να τον έχουμε αρχικοποιήσει. */
 *p = 1;
 while(*p <= 10)
 {
        std::cout << *p << '\n';
        (*p)++; /* Πρέπει να χρησιμοποιήσουμε παρενθέσεις για λόγους προτεραιότητας. */
 }
 return 0;
}
```

Ο Δείκτης void* (1)

- Υπάρχουν περιπτώσεις που μπορεί να χρειαστεί να αποθηκεύσουμε ή να μεταβιβάσουμε σε μία συνάρτηση τη διεύθυνση μνήμης μίας μεταβλητής της οποίας δεν γνωρίζουμε τον τύπο. Τότε, χρησιμοποιείται ο τύπος **void***
- Ένας **void*** δείκτης είναι ένας γενικός δείκτης, με την έννοια ότι μπορεί να δείξει σε μία μεταβλητή οποιουδήποτε τύπου
- Ένας **void*** δείκτης επιτρέπεται να εκχωρηθεί σε έναν άλλο **void*** δείκτη, καθώς και να του εκχωρηθεί ένας δείκτης με μη **void*** τύπο
- Σημειώστε ότι, αν δεν είναι δείκτης σε **void**, για να γίνει εκχώρηση ενός δείκτη σε άλλο δείκτη με διαφορετικό τύπο πρέπει να γίνει προσαρμογή. Π.χ.

```
char *p1;
int *p2;
...
p2 = p1; // Λάθος.
p2 = (int*)p1; // Τώρα, μεταγλωττίζεται.
```

Γενικά, είναι καλό να αποφεύγονται τέτοιου είδους προσαρμογές, αφού τα διαφορετικά μεγέθη μνήμης των τύπων, αν, για παράδειγμα, γίνει αποαναφοροποίηση, μπορεί να προκαλέσουν πρόβλημα στο πρόγραμμα

Ο Δείκτης void* (2)

- Για να προσπελάσουμε τη μεταβλητή με χρήση ενός void* δείκτη πρέπει να προσαρμόσουμε τον τύπο του στον τύπο της μεταβλητής, ώστε ο μεταγλωττιστής να γνωρίζει το αντίστοιχο μέγεθος. Π.χ.

```
void *ptr;
int i;
ptr = &i;
*ptr += 20; /* Λάθος μεταγλώττισης. Πρέπει να γίνει προσαρμογή τύπου, όπως φαίνεται παρακάτω. */
*(int*)ptr += 20; // Σωστό.
```

- Ένας δείκτης μπορεί να μετατραπεί σε τύπο void και πάλι πίσω στον αρχικό τύπο χωρίς απώλεια πληροφορίας

Χρήση της Λέξης const στη Δήλωση ενός Δείκτη

- Χρησιμοποιούμε τη δεσμευμένη λέξη const κατά τη δήλωση του δείκτη, όταν επιθυμούμε μία μεταβλητή-δείκτης:
  - είτε να μην μπορεί να αλλάξει την τιμή της μεταβλητής στην οποία δείχνει (χρήση της λέξης const πριν τον τύπο δεδομένων)
  - είτε να μην μπορεί να δείξει σε κάποια άλλη μεταβλητή (χρήση της λέξης const πριν το όνομα του δείκτη)

- Δείτε λοιπόν, τι επιτρέπεται και τι όχι, στα παρακάτω παραδείγματα

```
int j, i = 10;
const int *ptr;
ptr = &i;
*ptr = 30; /* Μη επιτρεπτή ενέργεια. */
ptr = &j; /* Επιτρεπτή ενέργεια. */
```

Ο δείκτης ptr δεν μπορεί να αλλάξει την τιμή της μεταβλητής στην οποία δείχνει (της i). Ωστόσο, επιτρέπεται να "δείξει" σε κάποια άλλη μεταβλητή ίδιου τύπου (εδώ της j)

```
int i, j;
int* const ptr = &i;
ptr = &j; /* Μη επιτρεπτή ενέργεια. */
*ptr = 30; /* Επιτρεπτή ενέργεια.
              Η τιμή του i γίνεται 30. */
```

Ο δείκτης ptr δεν μπορεί να "δείξει" σε άλλη μεταβλητή (όπως π.χ. εδώ στην j), παρά μόνο στην i (ωστόσο, επιτρέπεται να αλλάξει την τιμή του i). Όπως και με τις απλές const μεταβλητές, ο δείκτης πρέπει να αρχικοποιηθεί όταν δηλωθεί

Χρήση της Λέξης const στη Δήλωση ενός Δείκτη

- Χρησιμοποιώντας δύο φορές τη λέξη const κατά τη δήλωση του δείκτη, μπορούμε να τον αναγκάσουμε και να μην μπορεί να αλλάξει την τιμή της μεταβλητής στην οποία δείχνει και (ταυτόχρονα) να μην μπορεί να δείξει σε κάποια άλλη μεταβλητή

- Π.χ.

```
int i, j;
const int* const ptr = &i; /* Initialize pointer. */
ptr = &j; /* Illegal action. */
*ptr = 30; /* Illegal action. */
```

Αριθμητική Δεικτών

- Η αριθμητική δεικτών αφορά στην εκτέλεση αριθμητικών πράξεων με δείκτες
- Οι επιτρεπτές πράξεις είναι:
  - Η πρόσθεση ακεραίου σε δείκτη
  - Η αφαίρεση ακεραίου από δείκτη
  - Η αφαίρεση δύο δεικτών και
  - Η σύγκριση δύο δεικτών

Δείκτες και Ακέραιοι (Αύξηση Δεικτών)

- Η πρόσθεση ή η αφαίρεση ενός ακεραίου από ένα δείκτη παράγει αξιόπιστα αποτελέσματα μόνο όταν ο δείκτης δείχνει σε έναν πίνακα, αλλιώς το αποτέλεσμα είναι απροσδιόριστο
- Θεωρώντας ότι ο δείκτης `ptr` δείχνει σε ένα στοιχείο ενός πίνακα, η πρόσθεση ενός θετικού ακέραιου `n` στον `ptr`, π.χ.:

```
ptr = ptr + n;
```

αυξάνει την τιμή του δείκτη κατά `n * μέγεθος` του τύπου στον οποίο δείχνει ο `ptr` και **τον κάνει να δείχνει στη διεύθυνση του n-οστού στοιχείου μετά από αυτό που έδειχνε**

- Με τις παραδοχές που κάναμε για τα μεγέθη των τύπων, αν ο `ptr` έχει δηλωθεί σαν δείκτης σε έναν πίνακα `int` ή `float`, η τιμή του αυξάνει κατά `n*4`, αφού το μέγεθος και των δύο τύπων είναι τέσσερις οκτάδες. Αν είναι πίνακας `char`, η τιμή του αυξάνει κατά `n*1 = n`, ενώ αν είναι πίνακας `double`, η τιμή του αυξάνει κατά `n*8`
- Μετά την πρόσθεση του ακεραίου στο δείκτη, ο δείκτης πρέπει να δείχνει σε στοιχείο του πίνακα ή στην πρώτη θέση μετά το τέλος του πίνακα, η οποία θεωρείται έγκυρο αποτέλεσμα. Αν δεν ισχύει αυτό, το αποτέλεσμα της πρόσθεσης, καθώς και η αποαναφοροποίηση του δείκτη, δημιουργεί απροσδιόριστη συμπεριφορά

Παράδειγμα

```
#include <iostream>
int main()
{
 int *ptr, arr[] = {1, 2, 3};

 ptr = &arr[0];
 *ptr = 10;
 std::cout << "Addr: " << ptr << ' ' << arr[0] << '\n';

 ptr = ptr+2;
 *ptr = 20;
 std::cout << "Addr: " << ptr << ' ' << arr[2] << '\n';
 return 0;
}
```

Αφού ο ptr αρχικά δείχνει στο arr[0], η τιμή του arr[0] γίνεται 10. Μετά, επειδή ο ptr έχει δηλωθεί σαν δείκτης σε int, η τιμή του αυξάνεται κατά 8 (8 = 2*sizeof(int)) και όχι κατά 2. Έτσι, αφού ο ptr δείχνει στο arr[2], η τιμή του arr[2] γίνεται 20. Άρα, το πρόγραμμα εμφανίζει δύο διευθύνσεις με τη δεύτερη να είναι μεγαλύτερη κατά 8 θέσεις από την πρώτη και τις τιμές 10 και 20

Δείκτες και Ακέραιοι (Μείωση Δεικτών)

- Όμοια με την πρόσθεση, η αφαίρεση ενός θετικού ακέραιου από έναν δείκτη σε μία ανάθεση όπως: `ptr = ptr - n;` μειώνει την τιμή του δείκτη κατά `n * μέγεθος` του τύπου στον οποίο δείχνει ο δείκτης και κάνει τον δείκτη να δείχνει στη διεύθυνση του n-οστου στοιχείου πριν από αυτό που έδειχνε. Π.χ.

```
ptr = &arr[2];
cout << "Addr: " << ptr << '\n';
ptr = ptr-2;
cout << "Addr: " << ptr << '\n';
```

Ο `ptr` δείχνει στο `arr[0]` και η δεύτερη διεύθυνση είναι 8 θέσεις μικρότερη από την πρώτη

- Όπως και με την πρόσθεση, το αποτέλεσμα θεωρείται έγκυρο αν ο δείκτης δείχνει σε στοιχείο που ανήκει στον πίνακα ή στην επόμενη θέση μετά το τελευταίο. Π.χ.

```
int arr[] = {10, 20, 30};
int *ptr = arr+3; /* Έγκυρο, δείκτης στο στοιχείο μετά το τελευταίο. */
```

Αν και η ανάθεση με την διεύθυνση αμέσως μετά το τέλος του πίνακα είναι έγκυρη, όπως στην περίπτωση του `ptr`, μην επιχειρήσετε να προσπελάσετε τη συγκεκριμένη διεύθυνση γιατί είναι εκτός των ορίων. Π.χ. μην γράψετε `*ptr = 2;` η συμπεριφορά του προγράμματος είναι απρόβλεπτη

Δείκτες και Τελεστές ++/--

- Αν ο δείκτης δεν δείχνει σε στοιχείο πίνακα, η εφαρμογή των τελεστών ++/-- είναι έγκυρη και προκαλεί την αύξηση/μείωση του δείκτη κατά το μέγεθος του τύπου στον οποίο δείχνει
- Η συνδυαστική χρήση των τελεστών ++ και -- με τον τελεστή * είναι πολύ συνηθισμένη στη διαχείριση πινάκων με χρήση δείκτη
- Το αποτέλεσμα της έκφρασης βασίζεται στην προτεραιότητα των τελεστών
- Ας δούμε τις διαφορετικές περιπτώσεις συνδυαστικής χρήσης του τελεστή * με τον τελεστή ++ (αντίστοιχα είναι και για τον --)

`i = (*ptr)++;` πρώτα εκχωρείται η τιμή του `*ptr` στο `i` και μετά αυξάνεται κατά ένα η τιμή του `*ptr`

`i = *ptr++;` πρώτα εκχωρείται η τιμή του `*ptr` στο `i` και μετά αυξάνεται η τιμή του `ptr`, ανάλογα με τον τύπο του

`i = ++*ptr;` πρώτα αυξάνεται κατά ένα η τιμή του `*ptr` και μετά αυτή εκχωρείται στο `i`

`i = *++ptr;` πρώτα αυξάνεται η τιμή του `ptr` και μετά εκχωρείται στο `i` η τιμή του `*ptr`

Αφαίρεση Δεικτών

- Το αποτέλεσμα της αφαίρεσης δύο δεικτών είναι έγκυρο αν και οι δύο δείκτες δείχνουν σε στοιχεία του ίδιου πίνακα ή στην αμέσως επόμενη θέση από το τέλος του πίνακα
- Το αποτέλεσμα είναι ο αριθμός των στοιχείων που υπάρχουν μεταξύ τους
- Αν η τιμή του δείκτη που αφαιρείται είναι μεγαλύτερη, τότε το αποτέλεσμα είναι το ίδιο, απλά με αρνητικό πρόσημο
- Π.χ. αν ο `p1` δείχνει στο δεύτερο στοιχείο και ο `p2` στο πέμπτο στοιχείο του ίδιου πίνακα, το αποτέλεσμα της πράξης `p2-p1` είναι `3`, ενώ της πράξης `p1-p2` είναι `-3`

Σύγκριση Δεικτών

- Το αποτέλεσμα της σύγκρισης δύο δεικτών με τους τελεστές == και != είναι **αξιόπιστο**

- Με τους τελεστές <, <=, > και >= το αποτέλεσμα θεωρείται αξιόπιστο αν οι δείκτες δείχνουν σε μέλη του ίδιου αντικειμένου (π.χ. δομή) ή σε στοιχεία του ίδιου πίνακα συμπεριλαμβανόμενης και της επόμενης θέσης μετά το τέλος του, αλλιώς είναι απροσδιόριστο

- Π.χ. αν θέλουμε να ελέγξουμε αν δύο δείκτες `p1` και `p2` δείχνουν στην ίδια διεύθυνση μνήμης (ή όχι) μπορούμε να γράψουμε:

  `if(p1 == p2)` ή αντίστοιχα: `if(p1 != p2)`

- Π.χ. αν ο `p1` δείχνει στο δεύτερο στοιχείο και ο `p2` στο πέμπτο στοιχείο του ίδιου πίνακα, το αποτέλεσμα της πράξης `p2 > p1` είναι `1`, ενώ της πράξης `p1 > p2` είναι `0`

Παρατηρήσεις

- Εκτός από τις λειτουργίες που ήδη περιγράφησαν, καμία άλλη αριθμητική πράξη δεν επιτρέπεται να εκτελεστεί με τη συμμετοχή κάποιου δείκτη

- Π.χ. για τη δήλωση `double *ptr, *ptr1, *ptr2;`

οι εντολές:

πολλαπλασιασμού	`ptr *= 2;`
πρόσθεσης δεκαδικού	`ptr += 7.5;`
πρόσθεσης δεικτών	`ptr1 + ptr2;`

δεν είναι επιτρεπτές εκφράσεις ακόμα κι αν οι δείκτες αυτοί δείχνουν σε στοιχεία του ίδιου πίνακα

Παράδειγμα (1)

- Ποια είναι η έξοδος των παρακάτω προγραμμάτων;

-
```
#include <iostream> // 1
int main()
{
  int *ptr1, *ptr2, i = 20;

  ptr1 = ptr2 = &i;
  *ptr2 += 40;
  i += *ptr1;
  std::cout << *ptr2 << '\n';
  return 0;
}
```

-
```
#include <iostream> // 2
int main()
{
  int *ptr, i = 10, j = 20, k = 30;

  ptr = &i;
  *ptr = 40;

  ptr = &j;
  *ptr += i;

  ptr = &k;
  *ptr += i+j;

  std::cout << k << '\n';
  return 0;
}
```

-
```
#include <iostream> // 3
int main()
{
  int *ptr, i = 0;
  for(ptr = &i; *ptr < 5; i++)
  {
            (*ptr)++;
            ++*ptr;
            std::cout << i << ' ';
  }
  return 0;
}
```

-
```
#include <iostream> // 4
int main()
{
  int *ptr1, *ptr2, i = 10, j = 20;

  ptr1 = &i;
  *ptr1 = 150;

  ptr2 = &j;
  *ptr2 = 50;

  ptr2 = ptr1;
  *ptr2 = 250;

  ptr1 = ptr2;
  *ptr1 += *ptr2;
  std::cout << i+j << '\n';
  return 0;
}
```

Παράδειγμα (1)

Έξοδος:

```
120
130
2   5
550
```

Παράδειγμα (2)

- Να συμπληρώσετε το παρακάτω πρόγραμμα χρησιμοποιώντας τους δείκτες p1 και p2, για να εμφανίσετε το γινόμενο των άρτιων αριθμών από το 10 μέχρι και το 20. Οι μεταβλητές i και mul να χρησιμοποιηθούν μόνο μία φορά.

```
#include <iostream>
int main()
{
 int *p1, *p2, i, mul; /* Δεν επιτρέπεται να δηλώσετε
άλλες μεταβλητές. */
 ...
 return 0;
}
```

Παράδειγμα (2)

```
#include <iostream>
int main()
{
 int *p1, *p2, i, mul;

 p1 = &i; // Δεν ξεχνάμε την αρχικοποίηση των δεικτών.
 p2 = &mul;
 for(*p1 = 10, *p2 = 1; *p1 <= 20; (*p1)+=2)
        *p2 = *p2 * *p1;
 std::cout << "Mul = " << *p2 << '\n';
 return 0;
}
```

Δείκτες και Πίνακες (1)

- **Τα στοιχεία ενός πίνακα** αποθηκεύονται σε **διαδοχικές θέσεις μνήμης**, με το πρώτο στοιχείο στη χαμηλότερη διεύθυνση
- Ο τύπος του πίνακα καθορίζει την απόσταση μεταξύ των στοιχείων. Π.χ. σε έναν `char` πίνακα, τα στοιχεία απέχουν μία οκτάδα, ενώ σε έναν `int` πίνακα απέχουν το μέγεθος του `int` τύπου (π.χ. 4 οκτάδες)
- Η στενή σχέση πίνακα και δείκτη βασίζεται στο ότι το **όνομα ενός πίνακα** μπορεί να χρησιμοποιηθεί ως **δείκτης στο πρώτο του στοιχείο**
- Παρόμοια, το `arr+1` μπορεί να χρησιμοποιηθεί σαν δείκτης στο δεύτερο στοιχείο του πίνακα, το `arr+2` σαν δείκτης στο τρίτο, κ.ο.κ.

Δείκτες και Πίνακες (2)

- Στη γενική περίπτωση, οι ακόλουθες εκφράσεις είναι ισοδύναμες:

το arr είναι ισοδύναμο με &arr[0]
το arr+1 είναι ισοδύναμο με &arr[1]
...
το arr+n είναι ισοδύναμο με &arr[n]

- Έτσι, το παρακάτω πρόγραμμα εμφανίζει την ίδια τιμή δύο φορές:

```
#include <iostream>
int main()
{
        int *ptr, arr[5];

        ptr = arr;
        std::cout << ptr << ' ' << &arr[0] << '\n';
        return 0;
}
```

Δείκτες και Πίνακες (3)

- Αφού το όνομα ενός πίνακα μπορεί να χρησιμοποιηθεί σαν δείκτης στη διεύθυνση του πρώτου στοιχείου του, το περιεχόμενό του θα είναι ίσο με την τιμή του πρώτου στοιχείου
- Δηλαδή, η τιμή του `*arr` είναι ίση με `arr[0]`
- Παρόμοια, αφού το `arr+1` είναι δείκτης στο δεύτερο στοιχείο του πίνακα, τότε ισχύει ότι `*(arr+1)` είναι ίσο με `arr[1]`, κ.ο.κ.
- Δηλαδή, γενικά ισχύει ότι (**προσοχή στις παρενθέσεις**):

  το `*arr` είναι ισοδύναμο με `arr[0]`

  το `*(arr+1)` είναι ισοδύναμο με `arr[1]`

  ...

  το `*(arr+n)` είναι ισοδύναμο με `arr[n]`
- Προσοχή, οι παρενθέσεις είναι απαραίτητες, γιατί ο τελεστής `*` έχει υψηλότερη προτεραιότητα απ΄τον τελεστή `+`
- Επομένως, **οι εκφράσεις** `*(arr+n)` **και** `*arr+n` **δεν είναι ισοδύναμες** (αφού δεν αποτιμώνται με τον ίδιο τρόπο)

Παράδειγμα (1)

- Το επόμενο πρόγραμμα εμφανίζει τις διευθύνσεις μνήμης και τις τιμές των στοιχείων του πίνακα **arr** με δύο διαφορετικούς τρόπους

```
#include <iostream>
int main()
{
  int i, arr[5] = {10, 20, 30, 40, 50};

  std::cout << "***** Using array notation *****\n";
  for(i = 0; i < 5; i++)
     std::cout << "Addr: " << &arr[i] << " Val: " << arr[i] << '\n';
  std::cout << "\n***** Using pointer notation *****\n";
  for(i = 0; i < 5; i++)
     std::cout << "Addr: " << arr+i << " Val: " << *(arr+i) << '\n';
  return 0;
}
```

Παράδειγμα (2)

- Το επόμενο πρόγραμμα χρησιμοποιεί μία μεταβλητή δείκτη για να εμφανίσει τις διευθύνσεις μνήμης και τις τιμές των στοιχείων του πίνακα **arr**

```
#include <iostream>
int main()
{
        int *ptr, i, arr[5] = {10, 20, 30, 40, 50};

        ptr = arr;
        for(i = 0; i < 5; i++)
        {
                std::cout << "Addr: " << ptr << " Val: " << *ptr <<
'\n';
                ptr++; /* Η τιμή του ptr γίνεται ίση με τη διεύθυνση
του επόμενου στοιχείου του πίνακα. Ισοδύναμα, μπορούμε να γράψουμε
ptr = &arr[i]; */
        }
        return 0;
}
```

Παρατηρήσεις (1)

- Όταν το όνομα ενός πίνακα χρησιμοποιείται σαν δείκτης, η C++ τον χειρίζεται σαν `const` δείκτη. Επομένως, **δεν επιτρέπεται** να αλλάξει η τιμή του για να δείξει κάπου αλλού. Ένας πίνακας **δεν είναι** τροποποιήσιμη *lvalue*, η τιμή του είναι **πάντα** ίση με τη διεύθυνση του πρώτου στοιχείου του
- Δηλαδή, αν γράψουμε `arr++;` ή `arr = &i;` ο μεταγλωττιστής θα εμφανίσει μήνυμα λάθους για μη επιτρεπτή ενέργεια
- Τώρα, μπορείτε να καταλάβετε γιατί συμβαίνει αυτό που είπαμε στο Κ.7, ότι δεν μπορείτε να γράψετε `b = a;` για να αντιγράψετε τα στοιχεία του πίνακα `a` στον πίνακα `b`
- Ωστόσο, μπορούμε να εκχωρήσουμε την τιμή του σε έναν δείκτη, όπως κάναμε με την εντολή `ptr = arr;` στο παράδειγμα και στη συνέχεια να χρησιμοποιήσουμε αυτόν τον δείκτη για να προσπελάσουμε τα στοιχεία του πίνακα

Παρατηρήσεις (2)

- Αν και οι πίνακες και οι δείκτες σχετίζονται στενά μεταξύ τους, να ξεκαθαρίσουμε, ότι ένας πίνακας δεν είναι δείκτης και το αντίστροφο
- Π.χ. οι δηλώσεις `int a[50];` και `int *a;` είναι διαφορετικές
  - Με την πρώτη δήλωση ο μεταγλωττιστής δεσμεύει μνήμη για 50 ακεραίους και το όνομα `a` αναφέρεται πάντα στην ίδια θέση μνήμης. Όπως είπαμε, δεν μπορεί να αλλάξει η τιμή του, δηλαδή δεν μπορούμε να γράψουμε `a = arr;`
  - Με τη δεύτερη δήλωση ο μεταγλωττιστής δεσμεύει μνήμη για να αποθηκευτεί η τιμή του δείκτη (π.χ. 4 bytes). Σε αντίθεση με τον πίνακα, η τιμή του επιτρέπεται να αλλάξει και να δείξει κάπου αλλού, δηλαδή, μπορούμε να γράψουμε `a = arr;`
- Απλά να θυμάστε ότι, όταν το όνομα ενός πίνακα χρησιμοποιείται σε μία έκφραση, ο μεταγλωττιστής το μετατρέπει σε δείκτη στο πρώτο του στοιχείο.
- Πάντα; Όχι πάντα, υπάρχουν κάποιες εξαιρέσεις. Π.χ. όταν αποτελεί τον τελεστέο του `sizeof`. Το `sizeof` υπολογίζει το μέγεθος όλου του πίνακα

Παρατηρήσεις (3)

- Αν και ένας δείκτης δεν είναι πίνακας, μπορεί να χρησιμοποιηθεί με σημειογραφία πίνακα. Π.χ. το επόμενο πρόγραμμα χρησιμοποιεί τον δείκτη **ptr** σαν να ήταν πίνακας, για να εμφανίσει τις διευθύνσεις μνήμης και τις τιμές των στοιχείων του πίνακα **arr**

```
#include <iostream>
int main()
{
        int *ptr, i, arr[5] = {10, 20, 30, 40, 50};

        ptr = arr;
        for(i = 0; i < 5; i++)
                std::cout << "Addr: " << &ptr[i] << " Val: "
<< ptr[i] << '\n';
        return 0;
}
```

Παράδειγμα (1)

- Ποια είναι η έξοδος των παρακάτω προγραμμάτων;

```
#include <iostream>
int main()
{
  int *ptr, arr[] = {10, 20, 30, 40, 50};

  ptr = arr;
  *ptr = 3;

  ptr += 2;
  *ptr = 5;

  std::cout << arr[0]+arr[2];
  return 0;
}
```

```
#include <iostream>
int main()
{
  int i, *ptr1, *ptr2, arr[] = {10, 20, 30, 40, 50, 60, 70};

  ptr1 = &arr[2];
  ptr2 = &arr[4];
  for(i = ptr2-ptr1; i < 5; i+=2)
    std::cout << ptr1[i] << '\n';

  return 0;
}
```

Παράδειγμα (1)

Έξοδος:

```
8
50   70
```

Παράδειγμα (2)

- Να συμπληρώσετε το παρακάτω πρόγραμμα χρησιμοποιώντας τον δείκτη **ptr** για να διαβάσετε 50 ακεραίους και να αποθηκεύσετε στον πίνακα **arr** όσους έχουν τιμή στο [30, 40]. Το πρόγραμμα να εμφανίζει τον αριθμό των στοιχείων που αποθηκεύτηκαν στον πίνακα. Στα υπόλοιπα στοιχεία του πίνακα να αποθηκευτεί η τιμή **-1**. Η διαχείριση του πίνακα να γίνει με σημειογραφία δείκτη και η μεταβλητή **arr** να χρησιμοποιηθεί μέχρι **3** φορές

```
#include <iostream>
int main()
{
        const int SIZE = 50;
        int *ptr, i, arr[SIZE];
        ...
        return 0;
}
```

Παράδειγμα (2)

```
#include <iostream>
int main()
{
        const int SIZE = 50;
        int *ptr, i, arr[SIZE];

        ptr = arr;
        for(i = 0; i < SIZE; i++)
        {
                std::cout << "Enter number: ";
                std::cin >> *ptr;

                if(*ptr >= 30 && *ptr <= 40)
                        ptr++;
        }
        std::cout << ptr-arr << " elements are stored\n";
        for(; ptr < arr+SIZE; ptr++)
                *ptr = -1;
        return 0;
}
```

Πίνακας Δεικτών

- Ένας πίνακας δεικτών είναι ένας πίνακας, όπου τα στοιχεία του είναι δείκτες στον ίδιο τύπο δεδομένων
- Για να δηλώσουμε έναν πίνακα δεικτών χρησιμοποιούμε τον τελεστή * πριν από το όνομα του πίνακα. Π.χ. `int *p[3];`

  Δήλωση ενός πίνακα δεικτών με όνομα `p`, ο οποίος περιέχει 3 στοιχεία και το καθένα από αυτά είναι ένας δείκτης σε μία ακέραια μεταβλητή τύπου `int`
- Όταν δηλώνετε έναν πίνακα δεικτών, μην περικλείετε το όνομά του σε παρενθέσεις. Π.χ. με τη δήλωση: `int (*p)[3];` η μεταβλητή `p` δηλώνεται σαν δείκτης προς έναν πίνακα τριών ακεραίων και όχι σαν πίνακας τριών δεικτών
- Τα στοιχεία ενός πίνακα δεικτών τα χειριζόμαστε με τον ίδιο τρόπο, όπως και τους απλούς δείκτες

Παράδειγμα (1)

- Ποια είναι η έξοδος του παρακάτω προγράμματος;

```
#include <iostream>
int main()
{
        int *p[3], i = 10, j = 30;

        p[0] = &i;
        *p[0] = 20;
        p[1] = &j;
        p[2] = p[0];
        std::cout << i+*p[1]+*p[2] << '\n';
        return 0;
}
```

Παράδειγμα (1)

- Απάντηση. Με την εντολή `p[0] = &i;` ο δείκτης `p[0]` δείχνει στη διεύθυνση του `i`, επομένως το `*p[0]` είναι ισοδύναμο με το `i`. Άρα, το `i` γίνεται `20`. Παρόμοια, το `*p[1]` είναι ίσο με το `j`. Ο δείκτης `p[2]` δείχνει εκεί που δείχνει ο `p[0]`, δηλαδή, στο `i`. Άρα, το `*p[2]` είναι ίσο με το `i`. Επομένως, το πρόγραμμα εμφανίζει `70`

Παράδειγμα (2)

- Ποια είναι η έξοδος του παρακάτω προγράμματος;

```
#include <iostream>
int main()
{
        int *p[3], i, num;

        for(i = 0; i < 3; i++)
        {
                std::cout << "Enter number: ";
                std::cin >> num;
                p[i] = &num;
        }
        for(i = 0; i < 3; i++)
                std::cout << *p[i] << '\n';
        return 0;
}
```

Παράδειγμα (2)

- Απάντηση. Με την εντολή `p[i] = &num;` κάθε στοιχείο-δείκτης του πίνακα `p` δείχνει στη διεύθυνση της μεταβλητής `num`. Άρα, αφού και οι τρεις δείκτες δείχνουν στην ίδια διεύθυνση, το περιεχόμενό τους θα είναι ίσο με την τιμή του τρίτου ακεραίου που εισήγαγε ο χρήστης. Επομένως, ο δεύτερος βρόχος θα εμφανίσει τρεις φορές την τελευταία τιμή που εισήγαγε ο χρήστης

Δείκτης σε Δείκτη

- Όταν δηλώνεται ένας δείκτης, ο μεταγλωττιστής, όπως κάνει για οποιαδήποτε μεταβλητή, δεσμεύει τις απαραίτητες θέσεις μνήμης για να αποθηκεύσει την τιμή του
- Επομένως, μπορούμε να δηλώσουμε έναν άλλον δείκτη που να δείχνει σε αυτή τη διεύθυνση
- Για να δηλώσουμε έναν δείκτη σε κάποιον άλλον δείκτη χρησιμοποιούμε δύο φορές τον τελεστή *
- Π.χ. με την εντολή `int **pp;` η μεταβλητή `pp` δηλώνεται σαν δείκτης προς κάποιον άλλον δείκτη, ο οποίος με τη σειρά του μπορεί να δείξει στη διεύθυνση μίας `int` μεταβλητής
- Αν έχουμε δηλώσει έναν δείκτη σε έναν δεύτερο δείκτη, τότε με ένα * αποκτούμε πρόσβαση στη διεύθυνση του δεύτερου δείκτη, ενώ με το διπλό ** αποκτούμε πρόσβαση στο περιεχόμενο της διεύθυνσης που δείχνει ο δεύτερος δείκτης

Παράδειγμα

- Ποια είναι η έξοδος των παρακάτω προγραμμάτων;

```
#include <iostream>
int main()
{
  int *p, **pp, i = 20;

  p = &i;
  pp = &p;
  std::cout << **pp << '\n';
   return 0;
}
```

```
#include <iostream>
int main()
{
  int *p, **pp, i = 10, j = 20;

  p = &i;
  pp = &p;
  **pp = j;

  p = &j;
  **pp += 10;
  std::cout << i+j << '\n';
  return 0;
}
```

Παράδειγμα

- Απάντηση (1). Αφού ο pp δείχνει στη διεύθυνση του p, το *pp είναι ίσο με p. Αφού το p δείχνει στη διεύθυνση του i, το **pp είναι ίσο με i και το πρόγραμμα εμφανίζει 20
- Απάντηση (2). Αφού ο pp δείχνει στον p και αυτός στο i, το *pp είναι ίσο με i. Επομένως, η εντολή *pp = j; ισοδυναμεί με I = j = 20. Με την εντολή p = &j, ο p δείχνει στο j, άρα το **pp γίνεται ίσο με j. Επομένως, η εντολή **pp += 10; ισοδυναμεί με j = j+10 = 20+10 = 30. Τελικά, το πρόγραμμα εμφανίζει 50

Δείκτες και Διδιάστατοι Πίνακες (1)

- Θυμηθείτε ότι η C++ χειρίζεται έναν διδιάστατο πίνακα σαν μονοδιάστατο, όπου το κάθε στοιχείο του είναι πίνακας
- Για να χειριστούμε έναν διδιάστατο πίνακα `arr[N][M]` με αριθμητική δεικτών, χρησιμοποιούμε το κάθε όνομα πίνακα `arr[0]`, `arr[1]`, `...`, `arr[N-1]` σαν δείκτη στο πρώτο στοιχείο της αντίστοιχης γραμμής
- Π.χ. με την εντολή:

```
int arr[2][3];
```

το `arr[0]` μπορεί να χρησιμοποιηθεί σαν δείκτης προς έναν μονοδιάστατο πίνακα 3 ακεραίων που περιέχει τα στοιχεία της πρώτης γραμμής, δηλαδή τα `arr[0][0]`, `arr[0][1]` και `arr[0][2]`

Δείκτες και Διδιάστατοι Πίνακες (2)

- Συγκεκριμένα, το `arr[0]` είναι δείκτης στο πρώτο στοιχείο του πίνακα, δηλαδή στο `arr[0][0]`, άρα, η τιμή του `*arr[0]` είναι ίση με το `arr[0][0]`
- Αφού χρησιμοποιούμε το `arr[0]` σαν δείκτη στο πρώτο στοιχείο της γραμμής:
  - το `arr[0]+1` είναι δείκτης στο δεύτερο στοιχείο του πίνακα, δηλαδή στο `arr[0][1]`
  - το `arr[0]+2` είναι δείκτης στο τρίτο στοιχείο του πίνακα, δηλαδή στο `arr[0][2]`, κ.ο.κ.
  - ...
  - συνεπώς, στη γενική περίπτωση ισχύει ότι το `arr[0]+j` είναι δείκτης στο στοιχείο `arr[0][j]` της πρώτης γραμμής του διδιάστατου πίνακα

- Δηλαδή, ισχύει ότι:
  - το `arr[0]+j` είναι ισοδύναμο με `&arr[0][j]`
  - η τιμή του `*(arr[0]+j)` είναι ίση με `arr[0][j]`

Δείκτες και Διδιάστατοι Πίνακες (3)

- Αντίστοιχα, το `arr[1]` μπορεί να χρησιμοποιηθεί σαν δείκτης προς έναν πίνακα 3 ακεραίων που περιέχει τα στοιχεία της δεύτερης γραμμής, δηλαδή τα `arr[1][0]`, `arr[1][1]` και `arr[1][2]`

- Συγκεκριμένα:
  - το `arr[1]` είναι δείκτης στο πρώτο στοιχείο του πίνακα, δηλαδή στο `arr[1][0]`
  - Άρα, η τιμή του `*arr[1]` είναι ίση με το `arr[1][0]`

- Παρομοίως με πριν, ισχύει ότι το `arr[1]+j` είναι δείκτης στο στοιχείο `arr[1][j]` της δεύτερης γραμμής του διδιάστατου πίνακα

- Δηλαδή, ισχύει ότι:
  - το `arr[1]+j` είναι ισοδύναμο με `&arr[1][j]`
  - η τιμή του `*(arr[1]+j)` είναι ίση με `arr[1][j]`

Δείκτες και Διδιάστατοι Πίνακες (3)

- Επομένως, στη γενική περίπτωση έχουμε ότι:
  - το `arr[i]+j` είναι ισοδύναμο με `&arr[i][j]`
  - η τιμή του `*(arr[i]+j)` είναι ίση με `arr[i][j]`
- και επειδή, όπως είδαμε, το `arr[i]` μεταφράζεται σε `*(arr+i)` έχουμε ότι:
  - το `*(arr+i)+j` είναι ισοδύναμο με `&arr[i][j]`
  - η τιμή του `*(*(arr+i)+j)` είναι ίση με `arr[i][j]`
- Στην πραγματικότητα, όταν γράφουμε `arr[i][j]` ο μεταγλωττιστής το μετατρέπει σε `*(*(arr+i)+j)`

Παρατηρήσεις (1)

- Έστω ότι έχουμε τις ακόλουθες δηλώσεις:

```
int a[2][3], b[10];
```

- Ερώτηση: Όπως λέμε ότι το b μπορεί να χρησιμοποιηθεί σαν δείκτης στο b[0], είναι σωστό να πούμε ότι και το a είναι δείκτης στο a[0][0];

  Απάντηση: Επειδή η C++ χειρίζεται τον a σαν μονοδιάστατο πίνακα, η απάντηση είναι **όχι**, το a είναι δείκτης στο πρώτο στοιχείο του που είναι το a[0]. Άρα, αν θέλαμε να εκχωρήσουμε το a σε έναν δείκτη, ποιος θα έπρεπε να είναι ο τύπος του δείκτη; Θα ήταν **δείκτης σε πίνακα**, π.χ.:

```
int (*p)[3]; /* Οι παρενθέσεις είναι απαραίτητες, γιατί αλλιώς ο
μεταγλωττιστής θα μετέφραζε το p σαν πίνακα τριών δεικτών σε
ακεραίους. */
p = a;
```

  και τώρα το p δείχνει στο πρώτο στοιχείο της πρώτης γραμμής του a, δηλαδή στο a[0][0]

Παρατηρήσεις (2)

- Να θυμάστε: αν έχουμε τη δήλωση `int x[5]` το `x` μπορεί να «υποβιβαστεί» σε δείκτη, ενώ αν έχουμε τη δήλωση `int y[5][3]` το `y` δεν «υποβιβάζεται» σε δείκτη σε δείκτη, αλλά σε δείκτη σε πίνακα

- Επίσης σημειώστε: με δεδομένες τις δηλώσεις `int x[5]` και `int y[5][3]` ο τύπος της έκφρασης `&x` είναι δείκτης σε έναν πίνακα 5 ακεραίων, ενώ ο τύπος της έκφρασης `&y` δείκτης σε έναν πίνακα 5 πινάκων που ο καθένας έχει 3 ακεραίους

Παράδειγμα

- Τι κάνουν οι παρακάτω κώδικες;

1.
```
int i, arr[2][5] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
for(i = 0; i < 2; i++)
  *(arr[i]+3) = 0;
```

2.
```
int *ptr, arr[2][5] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
for(ptr = arr[1]+2; ptr < arr[1]+5; ptr++)
  *ptr = 0;
```

Παράδειγμα

1. Σε κάθε επανάληψη, το `arr[i]` δείχνει στο πρώτο στοιχείο της γραμμής `i`. Η έκφραση `arr[i]+3` είναι ένας δείκτης στο τέταρτο στοιχείο της γραμμής `i`. Συνεπώς, το `*(arr[i]+3)` είναι ισοδύναμο με `arr[i][3]`, οπότε, το πρόγραμμα μηδενίζει τα στοιχεία της τέταρτης στήλης του πίνακα, δηλαδή τα `arr[0][3]` και `arr[1][3]` γίνονται `0`

2. Με την εντολή `ptr = arr[1]+2;` ο `ptr` δείχνει στη διεύθυνση του `arr[1][2]`. Επομένως, το `*ptr` είναι ίσο με το `arr[1][2]`. Άρα, η εντολή `*ptr = 0` ισοδυναμεί με `arr[1][2] = 0`. Με την εντολή `ptr++`, ο `ptr` δείχνει στο επόμενο στοιχείο της συγκεκριμένης γραμμής. Για παράδειγμα, όταν αυξηθεί για πρώτη φορά θα δείχνει στη διεύθυνση του στοιχείου `arr[1][3]` και με την επόμενη αύξηση στη διεύθυνση του `arr[1][4]`. Επομένως, ο κώδικας μηδενίζει τα τρία τελευταία στοιχεία της δεύτερης γραμμής

Δείκτης προς Συνάρτηση

- **Σύσταση:** Για να γίνει κατανοητή η συγκεκριμένη υποενότητα, θα πρέπει να έχει διδαχθεί η ενότητα των Συναρτήσεων
- Όταν ορίζεται μία συνάρτηση, ο μεταγλωττιστής δεσμεύει μνήμη για να αποθηκεύσει τον κώδικά της
- Ένας δείκτης προς μία συνάρτηση δείχνει στη διεύθυνση μνήμης όπου βρίσκεται αποθηκευμένος ο κώδικας της συνάρτησης
- Αυτό που μπορούμε να κάνουμε με αυτόν τον δείκτη είναι να καλέσουμε τη συνάρτηση, δεν επιτρέπεται να την τροποποιήσουμε
- Η γενική μορφή της δήλωσης ενός δείκτη προς μία συνάρτηση είναι:

```
τύπος_επιστροφής (*όνομα_δείκτη) (τύπος_παραμ_1 όνομα_1,
τύπος_παραμ_2 όνομα_2, ..., τύπος_παραμ_ν όνομα_ν);
```

- Ο `τύπος_επιστροφής` καθορίζει τον τύπο επιστροφής της συνάρτησης, ενώ οι μεταβλητές `όνομα_1`, `όνομα_2`,..., `όνομα_ν` αποτελούν τις παραμέτρους της συνάρτησης (εφόσον η συνάρτηση δέχεται παραμέτρους)

Παραδείγματα Δήλωσης Δείκτη προς Συνάρτηση

```
int (*ptr)(int arr[], int size); /* Η μεταβλητή ptr
δηλώνεται σαν δείκτης προς μία συνάρτηση, η οποία
δέχεται σαν παραμέτρους έναν πίνακα ακεραίων και
έναν ακέραιο και επιστρέφει μία ακέραια τιμή. */

void (*ptr)(double *arr[]); /* Η μεταβλητή ptr
δηλώνεται σαν δείκτης προς μία συνάρτηση, η οποία
δέχεται σαν παράμετρο έναν πίνακα δεικτών σε
πραγματικούς και δεν επιστρέφει τίποτα. */

int test(void (*ptr)(int a)); /* Η συνάρτηση test()
επιστρέφει μία ακέραια τιμή και δέχεται σαν
παράμετρο έναν δείκτη προς μία άλλη συνάρτηση, η
οποία δέχεται μία ακέραια παράμετρο και δεν
επιστρέφει τίποτα. */
```

Παρατηρήσεις

- Το όνομα του δείκτη **πρέπει** να βρίσκεται ανάμεσα σε **παρενθέσεις**, γιατί ο τελεστής κλήσης της συνάρτησης `()` έχει υψηλότερη προτεραιότητα από τον τελεστή `*`

- Π.χ. αν γράψουμε:

```
int *ptr(int a);
```

αντί για

```
int (*ptr)(int a);
```

τότε δηλώνεται μία συνάρτηση με όνομα `ptr`, η οποία δέχεται μία ακέραια παράμετρο και επιστρέφει έναν δείκτη σε μία ακέραια μεταβλητή

- Το όνομα μίας συνάρτησης μπορεί να χρησιμοποιηθεί σαν δείκτης στη διεύθυνσή της

Χρήση Δείκτη προς Συνάρτηση

- Για να δείξει ένας δείκτης σε μία συνάρτηση πρέπει η δήλωση του δείκτη να ταιριάζει με το πρωτότυπο της συνάρτησης. Το παρακάτω πρόγραμμα εμφανίζει την τιμή 20

```
#include <iostream>

void test(int a);

int main()
{
        void (*ptr)(int a); /* Η μεταβλητή ptr δηλώνεται σαν δείκτης προς μία συνάρτηση, η οποία δέχεται μία ακέραια παράμετρο και δεν επιστρέφει κάτι. */
        ptr = test; /* Ο δείκτης ptr δείχνει στη διεύθυνση μνήμης της συνάρτησης test(). Εναλλακτικά, μπορούμε να γράψουμε ptr = &test; */
        (*ptr)(10); /* Κλήση της συνάρτησης στην οποία δείχνει ο δείκτης ptr. Εναλλακτικά, μπορούμε να γράψουμε ptr(10). */
        return 0;
}

void test(int a)
{
        std::cout << 2*a << '\n';
}
```

Πίνακας Δεικτών σε Συναρτήσεις

- Ένας πίνακας δεικτών σε συναρτήσεις είναι ένας πίνακας, του οποίου κάθε στοιχείο είναι δείκτης σε κάποια συνάρτηση
- Όλες οι συναρτήσεις πρέπει να έχουν το ίδιο πρωτότυπο
- Η δήλωσή του είναι παρόμοια με τη δήλωση ενός δείκτη σε συνάρτηση με τη διαφορά ότι αντί για ένας δείκτης δηλώνεται ένας πίνακας δεικτών
- Π.χ. η εντολή: `void (*ptr[20])(int a);`

  δηλώνει τον πίνακα `ptr` με 20 στοιχεία, όπου το κάθε στοιχείο του είναι δείκτης σε μία συνάρτηση που δέχεται μία ακέραια παράμετρο και δεν επιστρέφει κάτι

Παράδειγμα

- Στο παρακάτω πρόγραμμα κάθε στοιχείο του πίνακα `ptr` είναι δείκτης σε μία συνάρτηση που δέχεται δύο ακέραιες παραμέτρους και επιστρέφει μία ακέραια τιμή

```
#include <iostream>

int f1(int a, int b);
int f2(int a, int b);
int f3(int a, int b);

int main()
{
  int (*ptr[3])(int a, int b);
  int i, j, k;

  ptr[0] = f1; /* Ο δείκτης ptr[0] δείχνει στη διεύθυνση της συνάρτησης f1. */
  ptr[1] = f2;
  ptr[2] = f3;
  std::cout << "Enter numbers: ";
  std::cin >> i >> j;

  if(i > 0 && i < 10)
            k = ptr[0](i, j); /* Κλήση της συνάρτησης στην οποία δείχνει ο δείκτης
ptr[0]. Θα μπορούσαμε να γράψουμε k = (*ptr[0])(i, j). */
  else if(i >= 10 && i < 20)
            k = ptr[1](i, j); /* Κλήση της συνάρτησης στην οποία δείχνει ο δείκτης
ptr[1]. */
  else
            k = ptr[2](i, j); /* Κλήση της συνάρτησης στην οποία δείχνει ο δείκτης
ptr[2]. */
  std::cout << k << '\n';
  return 0;
}
```

Παράδειγμα

```
/* Συνέχεια Προγράμματος. */

int f1(int a, int b)
{
  return a+b;
}

int f2(int a, int b)
{
  return a-b;
}

int f3(int a, int b)
{
  return a*b;
}
```

Το πρόγραμμα διαβάζει δύο ακεραίους και ανάλογα με την τιμή του πρώτου χρησιμοποιεί κάποιον δείκτη για να καλέσει την αντίστοιχη συνάρτηση. Το πρόγραμμα εμφανίζει την τιμή επιστροφής της συνάρτησης.

Παράδειγμα

```
int test_1(int a, int b)
{
      return a+b;
}

int test_2(int a, int b)
{
      return a-b;
}

int test_3(int a, int b)
{
      return a*b;
}
```

Το πρόγραμμα ανάλογα με την τιμή του **i** καλεί την αντίστοιχη συνάρτηση μέσω του δείκτη που δείχνει στη διεύθυνσή της.
Η τιμή επιστροφής της συνάρτησης εμφανίζεται στην οθόνη.

Αλφαριθμητικά - Εισαγωγή

- Ένα αλφαριθμητικό είναι μία ακολουθία χαρακτήρων αποθηκευμένοι σε διαδοχικές θέσεις μνήμης
- Η C++ υποστηρίζει δύο τρόπους για την διαχείριση αλφαριθμητικών
- Ο πρώτος είναι όπως και στη C, δηλαδή, με πίνακα χαρακτήρων (C-style string). Ένα αλφαριθμητικό C-μορφής πρέπει να τελειώνει με έναν ειδικό χαρακτήρα, που ονομάζεται τερματικός χαρακτήρας (*null character*) και προσδιορίζει το τέλος του αλφαριθμητικού. Είναι ο πρώτος χαρακτήρας στο ASCII σύνολο, έχει ASCII τιμή `0` και συμβολίζεται με `'\0'`
- Μην συγχέετε τους χαρακτήρες `'\0'` και `'0'`. Ο πρώτος είναι ο τερματικός χαρακτήρας με ASCII τιμή `0` (μηδέν), ενώ ο δεύτερος είναι ο χαρακτήρας-ψηφίο μηδέν με ASCII τιμή `48`
- Επίσης, θα μιλήσουμε για την πρότυπη κλάση `string`, η οποία αποτελεί μία ευέλικτη και ασφαλέστερη προσέγγιση για την διαχείριση αλφαριθμητικών

Κυριολεκτικά Αλφαριθμητικά (1)

- Μία ακολουθία χαρακτήρων μέσα σε διπλά εισαγωγικά ονομάζεται κυριολεκτικό αλφαριθμητικό
- Ένα κυριολεκτικό αλφαριθμητικό είναι σταθερά. Τα εισαγωγικά δεν αποτελούν μέρος του, χρησιμοποιούνται μόνο για την οριοθέτησή του. Από τεχνική άποψη, η C++ το χειρίζεται σαν έναν ανώνυμο σταθερό πίνακα χαρακτήρων
- Συγκεκριμένα, όταν ο μεταγλωττιστής συναντήσει ένα κυριολεκτικό αλφαριθμητικό, δεσμεύει μνήμη για να αποθηκεύσει τους χαρακτήρες του και τον τερματικό χαρακτήρα
- Για παράδειγμα, αν ο μεταγλωττιστής συναντήσει στο πρόγραμμα το αλφαριθμητικό `"message"` δεσμεύει για αυτό οκτώ θέσεις μνήμης, ώστε να αποθηκεύσει τους επτά χαρακτήρες του και τον τερματικό χαρακτήρα (`'\0'`)
- Ένα κυριολεκτικό αλφαριθμητικό μπορεί να είναι κενό. Για παράδειγμα, με το `""` δεσμεύεται μνήμη μόνο για τον τερματικό χαρακτήρα

Κυριολεκτικά Αλφαριθμητικά (2)

- Για να είναι σαφές, οι εκφράσεις `"a"` και `'a'` είναι διαφορετικές. Η έκφραση `"a"` είναι ένα κυριολεκτικό αλφαριθμητικό, το οποίο είναι αποθηκευμένο στη μνήμη σαν ένας πίνακας δύο χαρακτήρων, το `'a'` και το `'\0'`. Ουσιαστικά, αντιπροσωπεύεται από έναν δείκτη σε αυτή τη μνήμη. Η έκφραση `'a'` είναι απλά ο χαρακτήρας `'a'` και αντιπροσωπεύεται από την ASCII τιμή του

Αποθήκευση Αλφαριθμητικών (1)

- Για να αποθηκεύσουμε ένα αλφαριθμητικό C-μορφής χρησιμοποιούμε έναν πίνακα χαρακτήρων
- Επειδή ένα αλφαριθμητικό C-μορφής πρέπει να τελειώνει με τον τερματικό χαρακτήρα, το μέγεθος του πίνακα για την αποθήκευση ενός αλφαριθμητικού Ν χαρακτήρων πρέπει να είναι Ν+1 θέσεις τουλάχιστον
- Για παράδειγμα, για να αποθηκεύσουμε ένα αλφαριθμητικό μέχρι 4 χαρακτήρες γράφουμε:

```
char str[5];
```

Αποθήκευση Αλφαριθμητικών (2)

- Όταν δηλώνεται ένας πίνακας μπορεί να αρχικοποιηθεί με ένα αλφαριθμητικό. Π.χ. με τη δήλωση: `char str[5] = "text";`

  Η τιμή του `str[0]` γίνεται `'t'`, η τιμή του `str[1]` γίνεται `'e'`, και η τιμή του τελευταίου στοιχείου `str[4]` γίνεται `'\0'`
- Ένας πιο ευέλικτος και ασφαλής τρόπος είναι να μην δηλώσουμε το μέγεθός του και να αφήσουμε τον μεταγλωττιστή να το υπολογίσει. Δηλαδή, `char str[] = "text";`
- Όπως με όλους τους πίνακες, αν το πλήθος των χαρακτήρων είναι μικρότερο από το μέγεθος του πίνακα, οι τιμές των υπολοίπων στοιχείων αρχικοποιούνται με `0`. Αν είναι μεγαλύτερο, προκύπτει λάθος μεταγλώττισης
- Π.χ. με τη δήλωση `char str[5] = "te";` το `str[0]` γίνεται `'t'`, το `str[1]` γίνεται `'e'` και τα υπόλοιπα στοιχεία ίσα με `0` ή ισοδύναμα ίσα με `'\0'`
- Όταν δηλώνετε έναν πίνακα χαρακτήρων για να αποθηκεύσετε ένα αλφαριθμητικό C-μορφής, μην ξεχνάτε να δεσμεύσετε μία επιπλέον θέση για την αποθήκευση του τερματικού χαρακτήρα
- Αν ο τερματικός χαρακτήρας δεν αποθηκευτεί, και το πρόγραμμα χρησιμοποιήσει κάποια συνάρτηση βιβλιοθήκης για να χειριστεί τον πίνακα (π.χ. `strlen()`), είναι πολύ πιθανό η συνάρτηση να μην λειτουργήσει σωστά

Εμφάνιση Αλφαριθμητικών

- Υπάρχουν πολλοί τρόποι με τους οποίους μπορούμε να εμφανίσουμε ένα αλφαριθμητικό C-μορφής. Π.χ. μπορούμε να χρησιμοποιήσουμε το `cout` και να μεταβιβάσουμε ένα δείκτη στο αλφαριθμητικό
- Ο παρακάτω κώδικας χρησιμοποιεί το όνομα του πίνακα σαν δείκτη στον πρώτο χαρακτήρα του αλφαριθμητικού

```
char str[] = "Print a message";
cout << str; // Εμφάνιση του αλφαριθμητικού
```

Ο κώδικας εμφανίζει τους χαρακτήρες του αλφαριθμητικού από τον πρώτο χαρακτήρα στον οποίο δείχνει ο δείκτης μέχρι να συναντήσει τον τερματικό χαρακτήρα

- Αν δεν θέλουμε να εμφανίσουμε από την αρχή το αλφαριθμητικό κάνουμε τον δείκτη να δείχνει στην επιθυμητή θέση. Π.χ., αν γράψουμε: `cout << str+8;` ή ισοδύναμα: `cout << &str[8];` εμφανίζεται το τμήμα του αλφαριθμητικού από τον ένατο χαρακτήρα και μετά, δηλαδή το `message`
- Η εμφάνιση των χαρακτήρων σταματάει όταν βρεθεί ο τερματικός χαρακτήρας. Π.χ. αν προσθέσουμε την εντολή: `str[5] = '\0';` πριν από την έξοδο, ο κώδικας θα εμφανίσει `Print`

Παράδειγμα

1. Είναι σωστά γραμμένο το παρακάτω πρόγραμμα;

```
#include <iostream>
int main()
{
        char str1[] = "abc", str2[] = "efg";
        str2[sizeof(str1)] = 'w';
        std::cout << str1[0] << '\n';
        return 0;
}
```

2. Ο σκοπός του παρακάτω προγράμματος είναι να αντιμεταθέσει τα περιεχόμενα των πινάκων. Είναι σωστά γραμμένο;

```
#include <iostream>
int main()
{
        char tmp[100], str1[100] = "Let see", str2[100] = "Is
everything OK?";
        tmp = str1;
        str1 = str2;
        str2 = tmp;
        std::cout << str1 << ' ' << str2 << '\n';
        return 0;
}
```

Παράδειγμα

Απάντηση (1). Το πρόγραμμα μεταγλωττίζεται κανονικά. Ας δούμε την εκτέλεσή του. Αφού το μέγεθος του `str1` είναι `4`, η εντολή εκχώρησης ισοδυναμεί με `str2[4] = 'w'`. Επειδή το μέγεθος του `str2` είναι `4`, αυτή η εντολή υπερεγγράφει τα δεδομένα σε μία θέση μνήμης που είναι εκτός των ορίων του `str2`, με αποτέλεσμα την απρόβλεπτη συμπεριφορά του προγράμματος. Για παράδειγμα, το πρόγραμμα μπορεί να εμφανίσει `a` μπορεί όμως και να εμφανίσει `w`, αν ο πίνακας `str1` έχει αποθηκευτεί στη μνήμη αμέσως μετά τον `str2`. **Όπως έχουμε αναφέρει, προσοχή στην υπέρβαση του ορίου ενός πίνακα**

Απάντηση (2). Θυμηθείτε από το Κ.8, όταν το όνομα ενός πίνακα χρησιμοποιείται σαν δείκτης είναι const δείκτης, δηλαδή δεν επιτρέπεται να αλλάξει η τιμή του και να δείξει σε κάποια άλλη διεύθυνση. Επομένως, η εντολή `tmp = str1;` δεν είναι αποδεκτή. Το ίδιο ισχύει και για τις επόμενες δύο εντολές, άρα το πρόγραμμα δεν θα μεταγλωττιστεί

Δείκτες και Κυριολεκτικά Αλφαριθμητικά

- Ένας βολικός τρόπος για να χειριστούμε ένα κυριολεκτικό αλφαριθμητικό είναι να δηλώσουμε έναν δείκτη σε αυτό. Π.χ.

```
#include <iostream>
int main()
{
        const char *ptr = "This is text";
        int i;
        for(i = 0; ptr[i] != '\0'; i++)
                std::cout << *(ptr+i) << ' ' << ptr[i] << '\n';
        return 0;
}
```

- Με την εντολή `ptr = "This is text";` ο μεταγλωττιστής δεσμεύει μνήμη για να αποθηκεύσει τους χαρακτήρες του κυριολεκτικού αλφαριθμητικού και τον τερματικό χαρακτήρα ενώ, στη συνέχεια, ο δείκτης `ptr` δείχνει σε αυτή τη μνήμη και συγκεκριμένα στη διεύθυνση του πρώτου χαρακτήρα
- Τους χαρακτήρες του αλφαριθμητικού μπορούμε να τους χειριστούμε είτε με σημειογραφία δείκτη είτε χρησιμοποιώντας τον δείκτη σαν πίνακα. Το πρόγραμμα εμφανίζει έναν-έναν τους χαρακτήρες και με τους δύο τρόπους. Ο βρόχος εκτελείται μέχρι να συναντήσουμε τον τερματικό χαρακτήρα

Παραδείγματα

1. Είναι σωστά γραμμένο το παρακάτω πρόγραμμα;

```
#include <iostream>
int main()
{
        char *ptr;
        ptr[0] = 'a';
        ptr[1] = 'b';
        ptr[2] = '\0';
        std::cout << ptr << '\n';
        return 0;
}
```

2. Ποια είναι η έξοδος του παρακάτω προγράμματος;

```
#include <iostream>
int main()
{
        char str1[] = "test", str2[] =
"test";
        if(str1 == str2)
                std::cout << "One\n";
        else
                std::cout << "Two\n";
        return 0;
}
```

3. Ποια είναι η έξοδος του παρακάτω προγράμματος;

```
#include <iostream>
int main()
{
        char *p, *q, s[] = "play";

        p = s+1;
        q = s;
        p[1] = 'x';
        *s = 'a';
std::cout << *q+2 << " " << *(q+2) << '\n'

        return 0;
}
```

Παραδείγματα

Απάντηση (1). Όχι βέβαια. Αφού ο `ptr` δεν έχει αρχικοποιηθεί, οι χαρακτήρες `'a'`, `'b'` και `'\0'` γράφονται σε τυχαίες διευθύνσεις μνήμης, με αποτέλεσμα την απρόβλεπτη συμπεριφορά του προγράμματος. Αν είχαμε δηλώσει έναν πίνακα χαρακτήρων και αρχικοποιήσει τον δείκτη `ptr` με τη διεύθυνσή του, όπως:

```
char str[3], *ptr;
ptr = str;
```

τότε το πρόγραμμα θα λειτουργούσε σωστά. **Μην ξεχνάτε ότι είναι πολύ σοβαρό λάθος να χρησιμοποιήσετε (αποαναφοροποιήσετε) έναν δείκτη που δεν έχει αρχικοποιηθεί**

Απάντηση (2). Αφού τα ονόματα των πινάκων χρησιμοποιούνται σαν δείκτες, η έκφραση `str1 == str2` ελέγχει αν οι δύο δείκτες δείχνουν στην ίδια διεύθυνση μνήμης, και όχι αν οι αντίστοιχοι πίνακες έχουν το ίδιο περιεχόμενο. Δείχνουν οι `str1` και `str2` στην ίδια θέση μνήμης;

Όχι βέβαια. Ναι μεν το περιεχόμενο των `str1` και `str2` πινάκων είναι το ίδιο, αλλά οι θέσεις μνήμης τους είναι διαφορετικές. Άρα, το πρόγραμμα θα εμφανίσει `Two`. Άν γράψουμε `if(*str1 == *str2)`, το πρόγραμμα θα εμφανίσει `One`, γιατί τα `*str1` και `*str2` είναι ίσα με `'t'`

Παραδείγματα

Απάντηση (3). Αφού ο **p** δείχνει στη διεύθυνση του δεύτερου στοιχείου του πίνακα **s**, η τιμή του **s[2]** αλλάζει σε 'x'. Επίσης, το **s[0]** αλλάζει σε 'a'. Επειδή ο q δείχνει στο **s[0]** και ο τελεστής * έχει μεγαλύτερη προτεραιότητα από τον τελεστή +, έχουμε *q+2 = 'a'+2. Άρα, το πρόγραμμα θα εμφανίσει την ASCII τιμή του χαρακτήρα που βρίσκεται δύο θέσεις μετά από τον 'a'. Ο χαρακτήρας αυτός είναι ο 'c' και το πρόγραμμα εμφανίζει 99. Αφού η τιμή της έκφρασης *(q+2) είναι ίση με το **s[2]**, το πρόγραμμα θα εμφανίσει **x**

Διάβασμα Αλφαριθμητικών

- Για το διάβασμα ενός αλφαριθμητικού υπάρχουν αρκετοί τρόποι

- Ένας τρόπος είναι να χρησιμοποιήσουμε την συνάρτηση `getline()`, η οποία είναι συνάρτηση μέλος της κλάσης `istream`

- Όπως θα δούμε στο Κ.17, για να καλέσουμε μία συνάρτηση που είναι μέλος μίας κλάσης γράφουμε το όνομα του αντικειμένου (π.χ. `cin`) και τον τελεστή τελεία `.`

Η Συνάρτηση `getline()` (1)

- Το πρώτο όρισμα της `getline()` είναι δείκτης στη μνήμη που θα αποθηκευτούν οι χαρακτήρες, ενώ το δεύτερο καθορίζει το μέγιστο πλήθος των χαρακτήρων που θα διαβαστούν. Ο αριθμός αυτός δεν πρέπει να είναι μεγαλύτερος από το μέγεθος της δεσμευμένης μνήμης, για να μην συμβεί υπερχείλιση
- Η `getline()` σταματάει να διαβάζει χαρακτήρες όταν διαβάσει τον χαρακτήρα νέας γραμμής ή όταν διαβάσει `size-1` χαρακτήρες, όποιο συμβεί πρώτο. Στο τέλος των χαρακτήρων που αποθηκεύτηκαν, προσθέτει τον τερματικό χαρακτήρα. Π.χ.

```
char str[30];
cin.getline(str, sizeof(str));
```

Η `getline()` θα διαβάσει το πολύ 29 χαρακτήρες, ώστε να μείνει μία θέση για τον τερματικό χαρακτήρα. Οι χαρακτήρες θα αποθηκευτούν στον πίνακα `str`. Αν συναντηθεί νωρίτερα ο χαρακτήρας νέας γραμμής, το διάβασμα θα σταματήσει

- Η `getline()` εξάγει τον χαρακτήρα νέας γραμμής από το ρεύμα εισόδου και δεν τον αποθηκεύει στον πίνακα. Αν ο χρήστης εισάγει παραπάνω χαρακτήρες, οι πλεονάζοντες θα παραμείνουν στην ουρά εισόδου
- Για να λειτουργεί το πρόγραμμά σας ανεξάρτητα από το μέγεθος του πίνακα, να χρησιμοποιείτε τον τελεστή `sizeof` αντί μίας τιμής (π.χ. 30)

Η Συνάρτηση getline() (2)

- Σημειώστε ότι υπάρχει μία έκδοση της getline(), η οποία μας επιτρέπει να καθορίσουμε τον χαρακτήρα που θα τερματίσει το διάβασμα. Π.χ.

```
cin.getline(str, sizeof(str), '#');
```

Τώρα, αν η getline() συναντήσει τον χαρακτήρα νέας γραμμής δε θα σταματήσει να διαβάζει. Έτσι, μπορούμε να διαβάσουμε πολλές γραμμές κειμένου. Θα σταματήσει, όταν διαβαστεί ο χαρακτήρας #

- Αν θέλουμε να βρούμε πόσοι χαρακτήρες έχουν διαβαστεί μπορούμε να χρησιμοποιήσουμε τη συνάρτηση gcount(). Π.χ.

```
char str[30];
cin.getline(str, sizeof(str));
cout << cin.gcount() << '\n';
```

Αν ο χρήστης εισάγει τη λέξη test, ο κώδικας θα εμφανίσει 5

- Ο δείκτης που μεταβιβάζεται στη getline() πρέπει να δείχνει σε μία μνήμη που έχει ήδη δεσμευτεί για την αποθήκευση του αλφαριθμητικού. Για παράδειγμα, είναι σωστά γραμμένος ο παρακάτω κώδικας;

```
char *p, s[10];
p = s;
cin.getline(p, 10);
```

Ναι, γιατί ο δείκτης p δείχνει σε δεσμευμένη μνήμη. Αν έλειπε η p = s; ο κώδικας θα ήταν λανθασμένος

Παράδειγμα (1)

- Μία **απρόσμενη** συμπεριφορά που κάνει πολλούς να αναρωτιούνται γιατί το πρόγραμμά τους δεν λειτουργεί σωστά συμβαίνει όταν το πρόγραμμα διαβάζει ένα αλφαριθμητικό μετά από το διάβασμα κάποιας αριθμητικής τιμής. Είδαμε μία παρόμοια περίπτωση σε παράδειγμα στο Κ.9. Για παράδειγμα, ας υποθέσουμε ότι ο χρήστης εισάγει έναν ακέραιο και μετά επιλέγει *Enter*. Ποια θα είναι η έξοδος του παρακάτω προγράμματος;

```
#include <iostream>
using std::cout;
using std::cin;

int main()
{
        char str[100];
        int num;

        cout << "Enter number: ";
        cin >> num;

        cout << "Enter text: ";
        cin.getline(str, sizeof(str));
        cout << num << ' ' << str;
        return 0;
}
```

Το πρόγραμμα διαβάζει τον ακέραιο και τον αποθηκεύει στη μεταβλητή `num`. Ο χαρακτήρας νέας γραμμής που δημιουργείται με το πάτημα του πλήκτρου *Enter* αποθηκεύεται στην ουρά εισόδου. Αφού η εκτέλεση της `getline()` τερματίζει όταν διαβαστεί ο χαρακτήρας νέας γραμμής, ο χρήστης δεν έχει τη δυνατότητα να εισάγει άλλους χαρακτήρες. Άρα, το πρόγραμμα εμφανίζει μόνο τον αριθμό που εισήγαγε ο χρήστης. Μία λύση είναι να χρησιμοποιήσουμε την `cin.get()` πριν από την `getline()`, ώστε να διαβαστεί ο χαρακτήρας νέας γραμμής.

Παράδειγμα (2)

- Να συμπληρώσετε το παρακάτω πρόγραμμα χρησιμοποιώντας τον δείκτη **p** και έναν **while** βρόχο, ώστε το πρόγραμμα να διαβάζει ένα αλφαριθμητικό μέχρι 100 χαρακτήρες και να εμφανίζει τον αριθμό των χαρακτήρων του, τον αριθμό των εμφανίσεων του χαρακτήρα **'b'**, καθώς και το αλφαριθμητικό, αφού πρώτα αντικαταστήσει κάθε κενό χαρακτήρα με τον χαρακτήρα νέας γραμμής και τα **'a'** με **'p'**.

```
#include <iostream>
int main()
{
        char *p, str[100];
        int cnt;
        ...
        return 0;
}
```

Παράδειγμα (2)

```
#include <iostream>
using std::cout;
using std::cin;
int main()
{
        char *p, str[100];
        int cnt;

        p = str;
        cout << "Enter text: ";
        cin.getline(p, sizeof(str));

        cnt = 0;
        while(*p != '\0')
        {
                if(*p == ' ')
                        *p = '\n';
                else if(*p == 'a')
                        *p = 'p';
                else if(*p == 'b')
                        cnt++;
                p++;
        }
        cout << "Len:" << p-str << " Times:" << cnt << "\nText:" << str << '\n';
        return 0;
}
```

Όταν τερματίσει η εκτέλεση του βρόχου, ο **p** δείχνει στον τερματικό χαρακτήρα. Με αριθμητική δεικτών βρίσκουμε το μήκος του αλφαριθμητικού

Συναρτήσεις Αλφαριθμητικών C-μορφής

- Η βιβλιοθήκη της C++ έχει κληρονομήσει τις συναρτήσεις αλφαριθμητικών της C
- Μην ξεχνάτε ότι ένα αλφαριθμητικό C-μορφής τελειώνει με τον τερματικό χαρακτήρα
- Αν λείπει, η συμπεριφορά τους είναι απρόβλεπτη
- Σε αυτή την ενότητα παρουσιάζονται συνοπτικά μερικές από τις πιο χρήσιμες συναρτήσεις αλφαριθμητικών

Η Συνάρτηση strlen()

- Η συνάρτηση strlen() δηλώνεται στο αρχείο cstring ως εξής:

```
size_t  strlen(const char *str);
```

- Ο τύπος size_t είναι δηλωμένος στην C++ βιβλιοθήκη σαν συνώνυμο κάποιου απρόσημου ακεραίου τύπου (π.χ. unsigned int)

- Η strlen() επιστρέφει τον αριθμό των χαρακτήρων του αλφαριθμητικού στο οποίο δείχνει ο δείκτης str, χωρίς να μετράει τον τερματικό χαρακτήρα. Π.χ.

```
int len;
len = strlen("Text"); /* Το len γίνεται 4. */

char str[] = "New text";
len = strlen(str); /* Το len γίνεται 8. */
```

Η Συνάρτηση strcpy()

- Ένα πολύ συνηθισμένο **λάθος** είναι η χρήση του τελεστή = για την αντιγραφή αλφαριθμητικών. Π.χ.

```
char str[10];
str = "something"; /* Λάθος */
```

- Ένας τρόπος για να αντιγράψουμε ένα αλφαριθμητικό είναι με τη συνάρτηση strcpy() που δηλώνεται στο αρχείο cstring, ως εξής:

```
char *strcpy(char *dest, const char *src);
```

- Η strcpy() αντιγράφει το αλφαριθμητικό στο οποίο δείχνει ο δείκτης src στη μνήμη που δείχνει ο δείκτης dest. Όταν αντιγραφεί και ο τερματικός χαρακτήρας, η strcpy() τερματίζει και επιστρέφει τον δείκτη dest. Π.χ.

```
char str[100];
strcpy(str, "something");
```

- Επειδή η strcpy() δεν ελέγχει αν η μνήμη προορισμού στην οποία θα αντιγραφεί το αλφαριθμητικό χωράει όλους τους χαρακτήρες του, πρέπει να έχετε εξασφαλίσει ότι το μέγεθός της θα είναι αρκετά μεγάλο, ώστε να αποφευχθεί η υπερεγγραφή μνήμης

- Και ναι, είναι σοβαρό λάθος να μεταβιβάσετε στην strcpy() έναν δείκτη που δεν έχει αρχικοποιηθεί. Για παράδειγμα, μην γράψετε κάτι τέτοιο:

```
char *str;
strcpy(str, "something"); // Λάθος, ο str δεν έχει αρχικοποιηθεί
```

Η Συνάρτηση strcat()

- Η συνάρτηση strcat() δηλώνεται στο αρχείο cstring ως εξής:

```
char *strcat(char *dest, const char *src);
```

- Η strcat() προσθέτει το αλφαριθμητικό στο οποίο δείχνει ο δείκτης src στο τέλος του αλφαριθμητικού που δείχνει ο δείκτης dest

- Η strcat() προσθέτει τον τερματικό χαρακτήρα και επιστρέφει τον δείκτη dest, ο οποίος δείχνει στο νέο, ενωμένο, αλφαριθμητικό. Π.χ.

```
char str1[20] = "One", str2[] = "Two";
strcat(str1, str2);
cout << str1; // Ο κώδικας εμφανίζει OneTwo
```

- Επειδή η strcat() δεν ελέγχει αν η μνήμη στην οποία θα προστεθεί το αλφαριθμητικό χωράει όλους τους χαρακτήρες του, το μέγεθος της μνήμης που έχει δεσμευτεί για το πρώτο αλφαριθμητικό πρέπει να είναι αρκετά μεγάλο για να χωράει τους χαρακτήρες και των δύο αλφαριθμητικών. Αν δεν είναι, οι πλεονάζοντες χαρακτήρες θα εγγραφούν σε θέσεις μνήμης μετά από το επιτρεπτό όριο, υπερεγγράφοντας τα δεδομένα που υπάρχουν εκεί με απρόβλεπτες συνέπειες στη λειτουργία του προγράμματος

Η Συνάρτηση strcmp() (1)

- Η συνάρτηση strcmp() δηλώνεται στο αρχείο cstring, ως εξής:

```
int strcmp(const char *str1, const char *str2);
```

- Η strcmp() συγκρίνει το αλφαριθμητικό στο οποίο δείχνει ο δείκτης str1 με το αλφαριθμητικό στο οποίο δείχνει ο δείκτης str2

- Αν τα δύο αλφαριθμητικά είναι ακριβώς ίδια, η strcmp() επιστρέφει 0. Αν το πρώτο αλφαριθμητικό είναι μικρότερο από το δεύτερο, επιστρέφει μία αρνητική τιμή ενώ, αν είναι μεγαλύτερο, επιστρέφει μία θετική τιμή

Η Συνάρτηση `strcmp()` (2)

- Ένα αλφαριθμητικό θεωρείται μικρότερο από κάποιο άλλο, αν ισχύει μία από τις δύο παρακάτω περιπτώσεις:

  α) οι πρώτοι `n` χαρακτήρες των αλφαριθμητικών είναι ίδιοι, αλλά η τιμή του `n+1` χαρακτήρα στο πρώτο αλφαριθμητικό είναι μικρότερη από την τιμή του `n+1` χαρακτήρα στο δεύτερο αλφαριθμητικό

  β) οι χαρακτήρες τους είναι οι ίδιοι, αλλά το μήκος του πρώτου αλφαριθμητικού είναι μικρότερο από το δεύτερο

- Π.χ. η εντολή: `strcmp("onE", "one");`

  επιστρέφει μία αρνητική τιμή, γιατί η ASCII τιμή του πρώτου μη κοινού χαρακτήρα `'E'` είναι μικρότερη από την τιμή του `'e'`

  Η εντολή: `strcmp("w", "many");` επιστρέφει μία θετική τιμή αφού η ASCII τιμή του πρώτου μη κοινού χαρακτήρα `'w'` είναι μεγαλύτερη από την αντίστοιχη του `'m'`

  Η εντολή: `strcmp("some", "something");` επιστρέφει μία αρνητική τιμή διότι μπορεί οι πρώτοι τέσσερις χαρακτήρες των δύο αλφαριθμητικών να είναι ίδιοι, αλλά το μήκος του πρώτου αλφαριθμητικού είναι μικρότερο από το μήκος του δεύτερου

Παρατηρήσεις

Όπως είδαμε και σε προηγούμενο παράδειγμα, ένα πολύ συνηθισμένο **λάθος** είναι η χρήση του τελεστή == για τη σύγκριση αλφαριθμητικών. Άρα, η παρακάτω **if** συνθήκη:

```
char str[20];
cin.getline(str, sizeof(str));
if(str == "test") /* Λανθασμένη απόπειρα σύγκρισης αλφαριθμητικών. */
```

δεν συγκρίνει τα αλφαριθμητικά, αλλά τις τιμές δύο δεικτών, του δείκτη **str** και του δείκτη στο κυριολεκτικό αλφαριθμητικό. Επειδή οι τιμές τους είναι διαφορετικές, η συνθήκη είναι πάντα ψευδής, ανεξάρτητα από αυτό που θα εισάγει ο χρήστης. Παρόμοια είναι και τα παρακάτω **λάθη**:

```
char str1[20], str2[20];
...
if(str1 > str2)
...
if(str1 < str2)
...
if(str1 == str2)
```

Όλες οι παραπάνω συνθήκες συγκρίνουν τις τιμές των δεικτών και όχι τα αλφαριθμητικά που περιέχονται στους πίνακες

Θυμόμαστε λοιπόν ότι για τη σύγκριση αλφαριθμητικών χρησιμοποιούμε την **strcmp()** και **όχι** τον τελεστή ==

Διδιάστατοι Πίνακες και Αλφαριθμητικά C-μορφής

- Οι διδιάστατοι πίνακες χρησιμοποιούνται πολύ συχνά για την αποθήκευση αλφαριθμητικών

  Π.χ., με την εντολή:

```
char str[3][40];
```

  δηλώνεται ο πίνακας **str**, ο οποίος περιέχει 3 γραμμές και σε κάθε γραμμή του πίνακα μπορεί να αποθηκευτεί ένα αλφαριθμητικό μέχρι 40 χαρακτήρες

- Μπορούμε να αποθηκεύσουμε κυριολεκτικά αλφαριθμητικά σε έναν διδιάστατο πίνακα ταυτόχρονα με τη δήλωσή του

  Π.χ. με τη δήλωση:

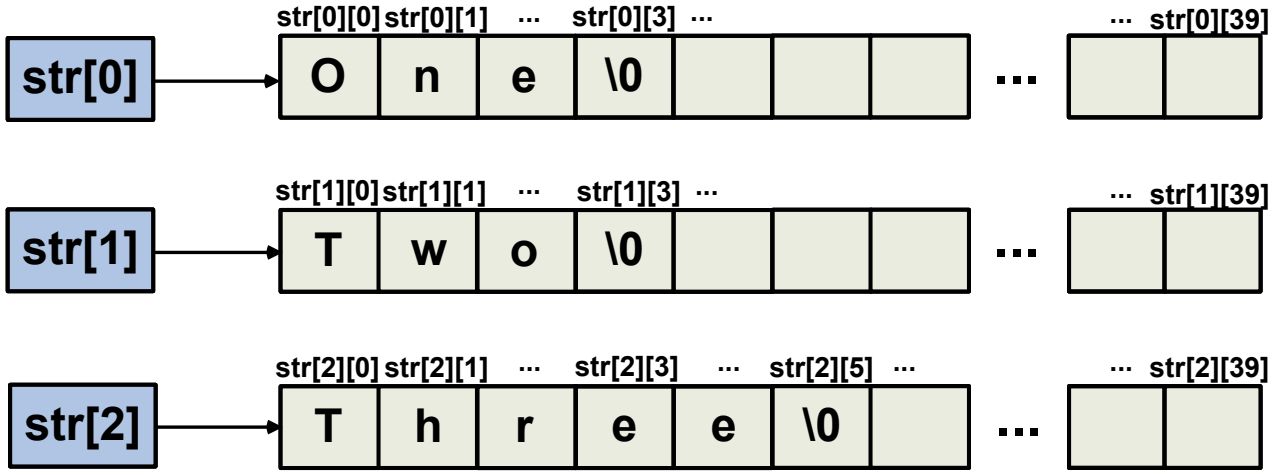
```
char str[3][40] = {"One", "Two", "Three"};
```

  οι χαρακτήρες του **"One"** αποθηκεύονται στην πρώτη γραμμή του πίνακα **str**, του **"Two"** στη δεύτερη και του **"Three"** στην τρίτη. Αφού το μήκος των αλφαριθμητικών είναι μικρότερο από το μέγεθος της κάθε γραμμής, προστίθενται τερματικοί χαρακτήρες

Διδιάστατοι Πίνακες και Αλφαριθμητικά C-μορφής

- Θυμηθείτε από την ενότητα των «Δείκτες και Διδιάστατοι Πίνακες» στο Κ.8 ότι μπορούμε να χειριστούμε το καθένα από τα `N` στοιχεία `str[0]`, `str[1]`, ..., `str[N-1]` ενός διδιάστατου πίνακα, έστω `str[N][M]`, σαν δείκτη σε πίνακα που περιέχει τα `M` στοιχεία της αντίστοιχης γραμμής.

- Άρα, στο προηγούμενο παράδειγμα, το `str[0]` μπορεί να χρησιμοποιηθεί σαν δείκτης σε έναν πίνακα 40 χαρακτήρων, ο οποίος περιέχει το αλφαριθμητικό `"One"`, ενώ με παρόμοιο τρόπο μπορούμε να χειριστούμε και τα στοιχεία `str[1]` και `str[2]`



Διδιάστατοι Πίνακες και Αλφαριθμητικά C-μορφής

- Θυμηθείτε επίσης από την ενότητα «Πίνακας Δεικτών» ότι για εξοικονόμηση μνήμης, αντί για διδιάστατο πίνακα, μπορούμε να χρησιμοποιήσουμε έναν πίνακα δεικτών:

```
const char *str[] = {"One", "Two", "Three"};
```

και για το κάθε αλφαριθμητικό δεσμεύεται όση μνήμη ακριβώς χρειάζεται, ενώ, όπως γνωρίζουμε, τον πίνακα μπορούμε να τον διαχειριστούμε σαν διδιάστατο για να έχουμε πρόσβαση σε κάθε χαρακτήρα των αλφαριθμητικών, π.χ.:

```
for(i = 0; i < 3; i++)
    if(str[i][0] == 'T')
        printf("%s\n", str[i]);
```

Ο παραπάνω βρόχος εμφανίζει τα αλφαριθμητικά που αρχίζουν από **T**

Παράδειγμα

- Ποια είναι η έξοδος του παρακάτω προγράμματος;

```
#include <iostream>
int main()
{
	char arr[7][10] = {"Monday", "Tuesday", "Wednesday",
"Thursday", "Friday", "Saturday", "Sunday"};
	int i;

	for(i = 0; i < 7; i++)
		if(arr[i][2] == 'n' && *(arr[i]+3) == 'd' &&
*(*(arr+i)+4) == 'a')
			std::cout << arr[i] << " is No." << i+1 <<
" week day\n";

	return 0;
}
```

Παράδειγμα

Απάντηση. Οι χαρακτήρες του "Monday" αποθηκεύονται στην πρώτη γραμμή του πίνακα **arr**, οι χαρακτήρες του "Tuesday" στη δεύτερη γραμμή, κ.ο.κ. Θυμηθείτε ότι το *(arr[i]+3) είναι ισοδύναμο με arr[i][3] και το *(*(arr+i)+4) ισοδύναμο με *(arr[i]+4), δηλαδή arr[i][4]. Άρα, ο βρόχος ελέγχει κάθε γραμμή του πίνακα και εμφανίζει τα αλφαριθμητικά των οποίων ο τρίτος, ο τέταρτος και ο πέμπτος χαρακτήρας είναι οι 'n', 'd' και 'a', αντίστοιχα. Επομένως, το πρόγραμμα εμφανίζει:

```
Monday is No.1 week day
Sunday is No.7 week day
```

Η Πρότυπη Κλάση `string`

- Για μεγαλύτερη ευελιξία και ευκολία στη διαχείριση αλφαριθμητικών, η C++ παρέχει την κλάση `string`
- Συγκεκριμένα, αντί να χρησιμοποιήσουμε έναν πίνακα χαρακτήρων, μπορούμε να χρησιμοποιήσουμε ένα αντικείμενο της κλάσης `string` για να χειριστούμε το αλφαριθμητικό
- Η κλάση `string` ορίζεται στο χώρο ονομάτων `std`
- Οι συναρτήσεις της κλάσης string παρέχουν μεγάλη ευκολία στη διαχείριση των αλφαριθμητικών, όπως στην αντιγραφή, πρόσθεση και σύγκριση αλφαριθμητικών, και κυρίως στη διαχείριση της μνήμης που καταλαμβάνουν

Παραδείγματα Χρήσης

- Ας δούμε μερικά παραδείγματα χρήσης της κλάσης:

```
string s1, s2; /* Καλείται ο εξ'ορισμού κατασκευαστής (default
constructor) και δημιουργούνται δύο άδεια string αντικείμενα. */
string s3("Test"); /* Δημιουργείται ένα string αντικείμενο που
αρχικοποιείται με ένα αλφαριθμητικό. Οι χαρακτήρες του Test
αντιγράφονται στο s3. */
s1 = s3; // Αντιγραφή του s3 στο s1.
s2 = s1+s3; // Εκχώρηση των ενωμένων αλφαριθμητικών στο s2.
s2 += "More"; // Πρόσθεση του αλφαριθμητικού στο τέλος του s2.
string s4(s1); /* Καλείται ο κατασκευαστής δόμησης αντιγράφου (copy
constructor) και το s4 αντικείμενο αρχικοποιείται με το s1. */
s3 += '?'; // Πρόσθεση του χαρακτήρα ? στο τέλος του s3.
s4 += s3; // Πρόσθεση του s3 στο τέλος του s4.
if(s1 == s2) // Σύγκριση αλφαριθμητικών.
if(s1 != s2) // Σύγκριση αλφαριθμητικών.
if(s1 == "Test") // Σύγκριση αλφαριθμητικών.
if(s1 > s2 && s2 <= s3) // Σύγκριση αλφαριθμητικών.
```

- Όπως φαίνεται, ένα string αντικείμενο δηλώνεται όπως μία μεταβλητή και όχι σαν πίνακας χαρακτήρων. Το αποτέλεσμα αυτών των ενεργειών είναι το `s1` να γίνει ίσο με `Test`, το `s2` ίσο με `TestTestMore`, το `s3` ίσο με `Test?` και το `s4` ίσο με `TestTest?`. Με αυτά τα παραδείγματα βλέπουμε πόσο πιο εύκολο είναι να αντιγράψουμε, να προσθέσουμε ή να συγκρίνουμε αλφαριθμητικά από ότι να χρησιμοποιούσαμε C-μορφής αλφαριθμητικά και αντίστοιχες συναρτήσεις

Παρατηρήσεις

- Εκτός από την ευκολία στις ενέργειες μεταξύ αλφαριθμητικών, το μεγάλο πλεονέκτημα όταν χρησιμοποιούμε ένα `string` αντικείμενο είναι ότι δεν χρειάζεται να ανησυχούμε για θέματα μνήμης
- Το μέγεθος της μνήμης προσαρμόζεται ανάλογα με τις απαιτήσεις δυναμικά, δηλαδή, κατά την εκτέλεση του προγράμματος
- Αντίθετα, για τα αλφαριθμητικά C-μορφής το μέγεθος ενός πίνακα χαρακτήρων καθορίζεται στο στάδιο της μεταγλώττισης και δεν μπορεί να μεταβληθεί
- Για παράδειγμα, αν και στην αρχή το μήκος του `s1` ήταν μηδέν, όταν έγινε η ανάθεση `s1 = s3;` το μήκος του άλλαξε για να αποθηκευτεί το `s3`. Το ίδιο συμβαίνει και με το μήκος του `s2` στην εντολή `s2 = s1+s3;`

Διάβασμα Αλφαριθμητικών

- Ας δούμε πως μπορούμε να διαβάσουμε χαρακτήρες και να τους αποθηκεύσουμε σε ένα `string` αντικείμενο

```
string s;
cin >> s; /* Πρώτα απομακρύνονται όλοι οι λευκοί χαρακτήρες που μπορεί να προηγούνται. Μετά, διαβάζονται και αποθηκεύονται χαρακτήρες στο αντικείμενο μέχρι να συναντηθεί κάποιος λευκός χαρακτήρας. Για παράδειγμα, αν ο χρήστης εισάγει τη φράση multiple words μόνο η λέξη multiple θα αποθηκευτεί στο s. */
getline(cin, s); /* Διαβάζονται και αποθηκεύονται χαρακτήρες στο s μέχρι να συναντηθεί ο χαρακτήρας νέας γραμμής. Το \n εξάγεται και δεν αποθηκεύεται στο s. */
getline(cin, s, '?'); /* Όπως πριν, με την διαφορά ότι ο χαρακτήρας τερματισμού του διαβάσματος είναι ο ? και όχι ο '\n'. */
```

- Παρατηρούμε ότι όταν χρησιμοποιούμε έναν πίνακα χαρακτήρων, χρησιμοποιούμε την `getline()` που είναι μέλος της `istream`. Όταν χρησιμοποιούμε ένα `string` αντικείμενο χρησιμοποιούμε μία διαφορετική `getline()` που δεν αποτελεί μέλος κάποιας κλάσης
- Προσέξτε επίσης ότι δεν υπάρχει παράμετρος που να καθορίζει τον μέγιστο αριθμό των χαρακτήρων που θα διαβαστούν, γιατί, όπως είπαμε, το μέγεθος του `string` αντικειμένου προσαρμόζεται αυτόματα για να χωρέσει το αλφαριθμητικό

Παράδειγμα (1)

- Η κλάση **string** περιέχει ένα μεγάλο πλήθος συναρτήσεων με τις οποίες μπορούμε να διαχειριστούμε το αλφαριθμητικό. Ας δούμε ένα παράδειγμα:

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
          int i, len;
          string s;

          cout << "Enter string: ";
          getline(cin, s);

          s.append("fin");
          len = s.size(); // Μήκος αλφαριθμητικού.
          cout << "L:" << len << " S:" << s << '\n';
          i = s.find('a');
          if(i == -1)
                    cout << "Not found\n";
          else
                    cout << "Found in position:" << i+1 << '\n';
          for(i = 0; i < len; i++)
          {
                    if(s[i] == 'p')
                              s[i] = 'w';
          }
          cout << s << '\n';
          s.resize(2*len, s[0]);
          s.erase(0, 3);
          cout << s << '\n';
          return 0;
}
```

Παράδειγμα (1)

- Το πρόγραμμα διαβάζει ένα αλφαριθμητικό, προσθέτει στο τέλος του το **fin**, εμφανίζει το μήκος του και την πρώτη θέση εμφάνισης του **a**. Μετά, αλλάζει όλα τα **p** με **w** και το εμφανίζει. Στη συνέχεια, διπλασιάζει το μέγεθός του και θέτει τους επιπλέον χαρακτήρες ίσους με τον πρώτο χαρακτήρα του. Τέλος, διαγράφει τους πρώτους τρεις χαρακτήρες και το εμφανίζει. Για την πρόσβαση στους χαρακτήρες του χρησιμοποιούμε τον τελεστή **[]**, όπως και με τους πίνακες

Παράδειγμα (2)

- Να γραφεί ένα πρόγραμμα το οποίο να διαβάζει και να αποθηκεύει ένα αλφαριθμητικό σε ένα `string` αντικείμενο. Στη συνέχεια, το πρόγραμμα να ελέγχει αν είναι «παλίνδρομο», δηλαδή, αν μπορεί να διαβαστεί με τον ίδιο τρόπο και από το τέλος προς την αρχή (π.χ. η λέξη level είναι παλίνδρομο, γιατί διαβάζεται με τον ίδιο τρόπο και από τις δύο κατευθύνσεις)

Παράδειγμα (2)

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
          int i, diff, len;
          string str;

          cout << "Enter text: ";
          getline(cin, str);
          len = str.size();

          diff = 0;
          for(i = 0; i < len/2; i++)
          {
                    if(str[i] != str[len-1-i]) /* Αν δύο χαρακτήρες δεν είναι ίδιοι ο βρόχος
τερματίζεται. */
                    {
                              diff = 1;
                              break;
                    }
          }
          if(diff == 1)
                    cout << str << " is not a palindrome\n";
          else
                    cout << str << " is a palindrome\n";
          return 0;
}
```

Σχόλια: Στον βρόχο γίνεται σύγκριση των χαρακτήρων από την αρχή του αλφαριθμητικού μέχρι τον μεσαίο χαρακτήρα με τους αντίστοιχους χαρακτήρες από το τέλος μέχρι τη μέση. Γι'αυτό το λόγο ο βρόχος εκτελείται από 0 μέχρι len/2. Ο τελευταίος χαρακτήρας του αλφαριθμητικού βρίσκεται στη θέση str[len-1].

Παράδειγμα (3)

- Να γραφεί ένα πρόγραμμα το οποίο να διαβάζει και να αποθηκεύει ένα αλφαριθμητικό σε ένα `string` αντικείμενο. Στη συνέχεια, να το εμφανίζει, αφού πρώτα αντικαταστήσει όλους τους 'a' χαρακτήρες που βρίσκονται στην αρχή ή και στο τέλος του με τον κενό χαρακτήρα. Για παράδειγμα, αν ο χρήστης εισάγει το "aabcdaa" το πρόγραμμα να εμφανίζει " bcd ", ενώ αν εισάγει το "bcdaa" να εμφανίζει "bcd "

Παράδειγμα (3)

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
          string str;
          int i, len;

          cout << "Enter text: ";
          getline(cin, str);
          len = str.size();
          for(i = 0; i < len; i++)
          {
                    if(str[i] == 'a')
                              str[i] = ' ';
                    else
                              break;
          }
          for(i = len-1; i >= 0; i--)
          {
                    if(str[i] == 'a')
                              str[i] = ' ';
                    else
                              break;
          }
          cout << "New text: " << str << '\n';
          return 0;
}
```

Σχόλια: Ο πρώτος βρόχος ελέγχει αν οι χαρακτήρες που βρίσκονται στην αρχή του αλφαριθμητικού είναι 'a'. Αν είναι, αντικαθίστανται με τον κενό χαρακτήρα, αλλιώς ο βρόχος τερματίζεται. Παρόμοια, ο δεύτερος βρόχος ελέγχει αν υπάρχουν 'a' χαρακτήρες στο τέλος του αλφαριθμητικού και τους αντικαθιστά με τον κενό χαρακτήρα.