C++: Από τη Θεωρία στην Εφαρμογή

Κεφάλαιο 1ο

Εισαγωγή στη Γλώσσα C++

Σύντομη Αναδρομή

- Η C++ προέρχεται από τη γλώσσα C, η οποία αναπτύχθηκε στις αρχές της δεκαετίας του 1970
- Η C++ άρχισε να αναπτύσσεται από τον Bjarne Stroustrup και συναδέλφους του, στις αρχές της δεκαετίας του 1980
- Η C++ υποστηρίζει *διαδικασιακό* προγραμματισμό με τη χρήση δομών ελέγχου, εντολών επανάληψης και συναρτήσεων, *αντικειμενοστρεφή* προγραμματισμό με την υλοποίηση και χρήση ιεραρχιών κλάσεων και αντικειμένων και *γενικευμένο* προγραμματισμό με την υλοποίηση και την χρήση προτύπων και γενικών αλγορίθμων
- Η C++ τυποποιήθηκε αρχικά το 1998 και από τότε συνεχίζει να εξελίσσεται με την παραγωγή νέων προτύπων
- Ένας μεγάλος αριθμός δημοφιλών εφαρμογών που χρησιμοποιούμε στην καθημερινή μας ζωή είναι γραμμένες σε C++

Κύκλος Δημιουργίας Προγράμματος

- Συγγραφή προγράμματος. Για τη συγγραφή του κώδικα μπορεί να χρησιμοποιηθεί οποιοσδήποτε διαθέσιμος κειμενογράφος
- Μεταγλώττιση προγράμματος. Ο μεταγλωττιστής (compiler) δημιουργεί ένα αρχείο, το οποίο περιέχει τον κώδικα του προγράμματος μεταφρασμένο στη γλώσσα που κατανοεί ο υπολογιστής, η οποία ονομάζεται γλώσσα μηχανής (machine language). Αυτό το αρχείο ονομάζεται αρχείο αντικειμένου (object file)
- Σύνδεση προγράμματος. Ο συνδέτης (linker) αναλαμβάνει να συνδέσει τον κώδικα αντικειμένου με τον κώδικα βιβλιοθήκης (library code) που χρησιμοποιεί το πρόγραμμα και άλλη απαραίτητη πληροφορία. Ο κώδικας βιβλιοθήκης περιέχει τον κώδικα μηχανής κλάσεων και συναρτήσεων βιβλιοθήκης που χρησιμοποιεί το πρόγραμμα
- Εκτέλεση προγράμματος. Ο συνδέτης παράγει το εκτελέσιμο αρχείο (π.χ. .exe στα Windows), το οποίο μπορούμε να εκτελέσουμε

Το Πρώτο Πρόγραμμα

```
#include <iostream>
int main()
{
        std::cout << "Hey Ho, Let's Go\n";
        return 0;
}
```

Η οδηγία #include

- #include <iostream>
  - Με την οδηγία #include <όνομα_αρχείου> ο μεταγλωττιστής υποχρεώνεται να συμπεριλάβει (include) το περιεχόμενο του αρχείου στον κώδικα του προγράμματος
  - Το αρχείο iostream περιέχει πληροφορία για κλάσεις και συναρτήσεις που είναι απαραίτητες για το διάβασμα δεδομένων (π.χ. από το πληκτρολόγιο) και την εμφάνιση δεδομένων (π.χ. στην οθόνη)

Παρατηρήσεις

- Το αρχείο `iostream` συμπεριλαμβάνεται σχεδόν πάντα σε ένα C++ πρόγραμμα, αφού περιέχει την απαραίτητη πληροφορία για την εμφάνιση και διάβασμα δεδομένων

- Οι οδηγίες (π.χ. `#include`) ξεκινούν πάντα με τον χαρακτήρα # και στο τέλος της οδηγίας δεν προστίθεται το ερωτηματικό ; ή κάποιος άλλος χαρακτήρας

- Ο προγραμματιστής μπορεί να δημιουργήσει και δικά του αρχεία και να τα συμπεριλάβει στο πρόγραμμά του με την οδηγία `#include`

- Κατά τη μεταγλώττιση του προγράμματος, ο μεταγλωττιστής ψάχνει να βρει σε ποιο σημείο του δίσκου είναι αποθηκευμένο το αρχείο για να το συμπεριλάβει στο πρόγραμμα

Η Συνάρτηση `main()`

- `int main()`
  - Ένα πρόγραμμα μπορεί να περιέχει πολλές συναρτήσεις
    - πρέπει όμως οπωσδήποτε να περιέχει τη συνάρτηση `main()`, η οποία αποτελεί την «κύρια συνάρτηση του προγράμματος»
  - Η `main()` καλείται αυτόματα από το λειτουργικό σύστημα όταν εκτελεστεί το πρόγραμμα
  - Η εκτέλεση του προγράμματος αρχίζει με την πρώτη εντολή της `main()` και τελειώνει με την τελευταία εντολή της, εκτός αν κληθεί νωρίτερα μία εντολή εξόδου, όπως η `return`
  - Οι εντολές της συνάρτησης `main()` ή αλλιώς το «σώμα της συνάρτησης» πρέπει να περιέχονται μέσα σε άγκιστρα `{}`

Η Συνάρτηση `main()`

- `int main()`
  - Η ειδική λέξη `int` σημαίνει ότι η `main()` πρέπει να επιστρέψει έναν ακέραιο αριθμό στο λειτουργικό σύστημα όταν τερματίσει
  - Αυτός ο ακέραιος επιστρέφεται με την εντολή `return`. Η τιμή 0 δηλώνει τον ομαλό τερματισμό του προγράμματος. Συνήθως, μία μη-μηδενική τιμή υποδεικνύει το λάθος που συνέβη

Το cout Αντικείμενο

- `std::cout << "Hey Ho, Let's Go\n";`
  - Οι εντολές γράφονται συνήθως μία σε κάθε γραμμή και σχεδόν πάντα τελειώνουν με το ελληνικό ερωτηματικό (;)
  - Το `cout` είναι ένα αντικείμενο το οποίο δηλώνεται στον χώρο ονομάτων `std` και συνδέεται με την προκαθορισμένη έξοδο (π.χ. οθόνη)
  - Το `cout` ξέρει πώς να εμφανίζει διαφορετικούς τύπους δεδομένων, όπως αλφαριθμητικά, αριθμούς και χαρακτήρες
  - Η σημειογραφία `<<` δηλώνει ότι το αλφαριθμητικό αποστέλλεται στο `cout`
  - Ο χαρακτήρας `'\n'` δημιουργεί μία νέα γραμμή μετά την εμφάνιση του μηνύματος στην οθόνη
    - Ισοδυναμεί δηλ. με το πάτημα του πλήκτρου Enter

Χώροι Ονομάτων (1)

- Ο χώρος ονομάτων είναι ένα τμήμα του προγράμματος, στο οποίο δηλώνονται ορισμένα ονόματα (π.χ. ονόματα μεταβλητών). Αυτά τα ονόματα δεν είναι γνωστά (ορατά) έξω από αυτόν τον χώρο
- Για παράδειγμα, όλα τα ονόματα της καθιερωμένης βιβλιοθήκης ορίζονται σε ένα χώρο ονομάτων που ονομάζεται `std`
- Για να έχουμε πρόσβαση σε ένα στοιχείο του χώρου πρέπει να γράψουμε το όνομα του χώρου ακολουθούμενο από τον τελεστή επίλυσης εμβέλειας `::`. Έτσι, γράφοντας `std::cout` έχουμε πρόσβαση στο `cout` αντικείμενο που δηλώνεται στον `std` χώρο

Χώροι Ονομάτων (2)

- Αν θέλουμε να είναι διαθέσιμα όλα τα ονόματα του `std` χώρου γράφουμε: `using namespace std;` Τώρα, δεν χρειάζεται να προσθέτουμε το πρόθεμα `std::` πριν από τα ονόματα που χρησιμοποιούμε. Για παράδειγμα, μπορούμε να γράψουμε: `cout << "Hey Ho, Let's Go\n";`
- Εναλλακτικά, μπορούμε να κάνουμε διαθέσιμα μόνο τα ονόματα που χρησιμοποιούνται στο πρόγραμμά μας. Αυτό επιτυγχάνεται με αντίστοιχες `using` δηλώσεις. Για παράδειγμα: `using std::cout;` Τώρα, δεν χρειάζεται να προσθέτουμε το πρόθεμα `std::` όταν χρησιμοποιούμε το `cout`, ενώ πρέπει με άλλα ονόματα του `std`

Εισαγωγή Σχολίων σε Πρόγραμμα (1)

- **Τι είναι τα «Σχόλια»;**
  - Είναι κείμενο το οποίο εισάγεται από τον προγραμματιστή στον κώδικα προκειμένου να καταστήσει τον ίδιο τον κώδικα περισσότερο ευανάγνωστο και σαφή
  - Τα σχόλια εισάγονται είτε για επεξήγηση προς τρίτους (που πιθανόν διαβάσουν τον κώδικά μας) είτε και για μας τους ίδιους (ιδίως όταν πρόκειται να διαβάσουμε τον κώδικά μας μετά από αρκετό χρονικό διάστημα)
  - Περιέχεται μεταξύ /* και */ (και αγνοείται από τον υπολογιστή)
  - Επίσης σχόλιο είναι και το κείμενο στην ίδια γραμμή μετά από //

Εισαγωγή Σχολίων σε Πρόγραμμα (2)

- π.χ.

```
/* Ο σκοπός του πρώτου προγράμματος είναι να εμφανίσει ένα μήνυμα στην
οθόνη. */
#include <iostream>
using std::cout; // Θα χρησιμοποιήσουμε το αντικείμενο cout.

int main()
{
        cout << "Hey Ho, Let's Go\n";
        return 0;
}
```

Λάθη Μεταγλώττισης (1)

- Τα πιο συνηθισμένα λάθη μεταγλώττισης, ιδίως από αρχάριους προγραμματιστές, είναι τα συντακτικά
- Η C++ είναι μία γλώσσα προγραμματισμού που κάνει διάκριση μεταξύ πεζών και κεφαλαίων γραμμάτων (case sensitive)
  - Δηλαδή, αν γράψουμε `Cout` αντί για `cout` ο μεταγλωττιστής θα εμφανίσει μήνυμα λάθους, γιατί ο χαρακτήρας `'C'` είναι διαφορετικός από τον χαρακτήρα `'c'`

- Μερικές φορές το λάθος που υποδεικνύει ένας μεταγλωττιστής μπορεί να μη βρίσκεται στη γραμμή που υποδεικνύει, αλλά σε κάποια προηγούμενη, με πιθανότερη την αμέσως προηγούμενη

- Αν ο μεταγλωττιστής εμφανίζει πολλά μηνύματα λάθους, τότε διορθώστε μόνο το πρώτο λάθος ή μαζί και κάποια άλλα προφανή λάθη και μεταγλωττίστε πάλι το πρόγραμμά σας. Είναι πολύ πιθανό, η νέα μεταγλώττιση να εμφανίσει πολύ λιγότερα ή και καθόλου λάθη

Λάθη Μεταγλώττισης (2)

- Ο μεταγλωττιστής μπορεί να μην εμφανίσει μηνύματα λάθους, αλλά να εμφανίσει μηνύματα προειδοποίησης (warnings). Ένα μήνυμα προειδοποίησης μπορεί να ενημερώνει τον προγραμματιστή για ενδεχόμενη δυσλειτουργία του προγράμματός του, οπότε καλό είναι να μην τα αγνοείτε

- Ο μεταγλωττιστής ανιχνεύει λανθασμένη εφαρμογή των κανόνων της γλώσσας που τον εμποδίζει να μεταγλωττίσει το πρόγραμμα σωστά (π.χ. ορθογραφικά λάθη, συντακτικά λάθη, αδήλωτες μεταβλητές, ...) και όχι λάθη λογικής που ενδέχεται να περιέχονται στο πρόγραμμά σας

- Δηλαδή, ο μεταγλωττιστής δεν είναι «μέσα στο κεφάλι σας» για να ξέρει ποιος είναι ο σκοπός του προγράμματός σας

- Άρα, αν το πρόγραμμά σας μεταγλωττίζεται σωστά, δεν σημαίνει απαραίτητα ότι θα παράγει και σωστά αποτελέσματα

Λάθη Μεταγλώττισης (3)

- Π.χ. αν θέλετε το πρόγραμμά σας να εμφανίζει τη λέξη Less αν η τιμή του a είναι μικρότερη από 5 και γράψετε:

```
if(a > 5) // Λογικό λάθος.
        cout << "Less\n";
```

τότε, αν και αυτός ο κώδικας μεταγλωττίζεται επιτυχημένα, το πρόγραμμα δεν θα λειτουργεί όπως θα θέλατε

- Λάθη σαν και αυτό ονομάζονται **λογικά λάθη** (bugs) και δεν εντοπίζονται από τον μεταγλωττιστή. Αποτελεί υποχρέωση του προγραμματιστή να τα εντοπίσει και να τα διορθώσει
- Η διαδικασία ανίχνευσης λαθών ονομάζεται αποσφαλμάτωση *(debugging)*. Οι μεταγλωττιστές συνήθως συνοδεύονται από ένα περιβάλλον ανίχνευσης λαθών, το οποίο βοηθάει τον προγραμματιστή να εντοπίσει τα λάθη του προγράμματός του

C++: Από τη Θεωρία στην Εφαρμογή

Κεφάλαιο 2ο

Μεταβλητές, Σταθερές και Αριθμητικές Μετατροπές

Μεταβλητές

Σχέση Μνήμης Υπολογιστή και Μεταβλητών

- Η μνήμη (RAM) ενός υπολογιστή αποτελείται από πολλά εκατομμύρια θέσεις αποθήκευσης δεδομένων που έχουν διαδοχική αρίθμηση
- Το μέγεθος κάθε θέσης μνήμης είναι μία οκτάδα (byte)
- Π.χ. σκεφτείτε ότι ένας παλιός υπολογιστής με μόνο 16MB μνήμης έχει μνήμη:

  16 * 1.024 = 16.384 kbytes

  16.384 * 1.024 = 16.777.216 θέσεις μνήμης (bytes)
- Κάθε θέση μνήμης μπορεί να έχει ένα όνομα και ένα περιεχόμενο
- Μία **μεταβλητή** ονομάζεται μία **θέση μνήμης** που έχει ένα συγκεκριμένο **όνομα** και **περιεχόμενο** (π.χ. `i`)
- Η **τιμή** μίας **μεταβλητής** είναι **το περιεχόμενο** αυτής της θέσης μνήμης (π.χ. 10)

Ονόματα Μεταβλητών

- **Απαράβατοι κανόνες κατά τη δήλωση του ονόματος μίας μεταβλητής**
  - Το όνομα μπορεί να περιέχει γράμματα, ψηφία και τον χαρακτήρα υπογράμμισης _
  - Ο πρώτος χαρακτήρας πρέπει να είναι γράμμα ή ο χαρακτήρας υπογράμμισης _
  - Η γλώσσα C++ είναι case sensitive (δηλ. κάνει διάκριση μεταξύ των πεζών και κεφαλαίων γραμμάτων)
    - Συνεπώς, η μεταβλητή με το όνομα **sum** είναι διαφορετική από τη μεταβλητή με το όνομα **Sum**
  - Οι δεσμευμένες λέξεις της C++ απαγορεύεται να χρησιμοποιηθούν ως ονόματα μεταβλητών

Δεσμευμένες Λέξεις

```
and       const_cast  friend      or_eq       template    volatile
and_eq    continue    goto        private     this        wchar_t
asm       default     if          protected   throw       while
auto      delete      inline      public      true        xor
bitand    do          int         register    try         xor_eq
bitor     double      long        reinterpret_cast
bool      dynamic_cast            return      typedef
break     else        mutable     short       typeid
case      enum        namespace   signed      typename
catch     explicit    new         sizeof      union
char      extern      not         static      unsigned
class     false       not_eq      static_cast using
compl     float       operator    struct      virtual
const     for         or          switch      void
```

Η C++11 δεσμεύει και τις ακόλουθες λέξεις:

```
alignas alignof  constexpr  char16_t char32_t decltype  noexcept
nullptr static_assert thread_local
```

Παρατηρήσεις

- Το όνομα που επιλέγετε να δώσετε σε μία μεταβλητή είναι χρήσιμο να περιγράφει όσο το δυνατόν καλύτερα τον σκοπό της μεταβλητής μέσα στο πρόγραμμα, ώστε ο κώδικας να είναι πιο ευανάγνωστος
  - Π.χ. το όνομα μίας μεταβλητής που υπολογίζει το άθροισμα κάποιων αριθμών είναι προτιμότερο να είναι `sum` αντί για `var`

- Αν το όνομα που επιλέξατε για μία μεταβλητή αποτελείται από δύο ή και περισσότερες λέξεις, μία συνήθης πρακτική είναι να διαχωρίζονται μεταξύ τους με τον χαρακτήρα '_', έτσι ώστε να διευκολύνεται η ερμηνεία τους
  - Π.χ. το όνομα μίας μεταβλητής που υπολογίζει τον αριθμό των βιβλίων σε μία βιβλιοθήκη είναι προτιμότερο να είναι `books_number` αντί για `booksnumber`

- Μην δίνετε παρόμοια ονόματα σε μεταβλητές (π.χ. `more` και `More`), γιατί είναι πολύ εύκολο να κάνετε λάθος και να χρησιμοποιήσετε την μία μεταβλητή στη θέση της άλλης
- Τα ονόματα απλών μεταβλητών συνηθίζεται να γράφονται με πεζά γράμματα, ενώ τα ονόματα σταθερών και μακροεντολών (τις οποίες θα δούμε στη συνέχεια) με κεφαλαία

Δήλωση Μεταβλητών

- Για να χρησιμοποιήσετε μία μεταβλητή μέσα σε ένα πρόγραμμα πρέπει πρώτα να τη δηλώσετε
- Μία μεταβλητή μπορεί να δηλωθεί σε όποιο σημείο του προγράμματος επιθυμούμε
- Η δήλωση μίας μεταβλητής γίνεται με τον ακόλουθο τρόπο:

```
τύπος_δεδομένων όνομα_μεταβλητής;
```

- Το `όνομα_μεταβλητής` είναι το τυχαίο όνομα που επιλέγει ο προγραμματιστής σύμφωνα με τους κανόνες και τις παρατηρήσεις που είπαμε προηγουμένως
- Ο `τύπος_δεδομένων` είναι ένας από τους τύπους δεδομένων που υποστηρίζει η γλώσσα C++
  - Π.χ. η δεσμευμένη λέξη `int` χρησιμοποιείται για τη δήλωση ακέραιων μεταβλητών, δηλαδή μεταβλητών που μπορούν να έχουν μόνο ακέραιες τιμές ενώ η δεσμευμένη λέξη `float` χρησιμοποιείται για τη δήλωση πραγματικών μεταβλητών, δηλαδή μεταβλητών που μπορούν να έχουν τιμές με κλασματικό μέρος

Τύποι Δεδομένων

Τύπος	Συνηθισμένο μέγεθος (bytes)	Εύρος τιμών (min-max)
bool	1	false/true
char	1	-128 ... 127
wchar_t	2	-32.768 ... 32.767
short int	2	-32.768 ... 32.767
int	4	-2.147.483.648...2.147.483.647
long int	4	-2.147.483.648...2.147.483.647
float	4	Μικρότερη θετική τιμή: $1.17*10^{-38}$ Μεγαλύτερη θετική τιμή: $3.4*10^{38}$
double	8	Μικρότερη θετική τιμή: $2.2*10^{-308}$ Μεγαλύτερη θετική τιμή: $1.8*10^{308}$
long double	12, 16	
unsigned char	1	0 ... 255
unsigned short int	2	0 ... 65535
unsigned int	4	0 ... 4.294.967.295
unsigned long int	4	0 ... 4.294.967.295

Η C++11 προσθέτει τους παρακάτω τύπους:

char16_t: Χρησιμοποιείται για την αποθήκευση συνόλων χαρακτήρων 16 bit, όπως το UTF-16.

char32_t: Χρησιμοποιείται για την αποθήκευση συνόλων χαρακτήρων 32 bit, όπως το UTF-32.

long long int: Χρησιμοποιείται για πολύ μεγάλους ακεραίους (π.χ. τουλάχιστον 64 bit). Ισχύει `sizeof(long) <= sizeof(long long)`.

Δημιουργία Συνωνύμων Τύπων (1)

- Υπάρχουν πολλές περιπτώσεις όπου η δημιουργία συνωνύμων για τύπους μπορεί να κάνει το πρόγραμμά μας πιο ευέλικτο και πιο εύκολο να διαβαστεί
- Για να δημιουργήσουμε συνώνυμο για έναν υπάρχοντα τύπο δεδομένων μπορούμε να χρησιμοποιήσουμε την δεσμευμένη λέξη `typedef` και με την C++11 την λέξη `using`
- Π.χ., η δήλωση

```
typedef short int short_t;
```

κάνει το όνομα `short_t` συνώνυμο του τύπου `short int`. Άρα, οι δηλώσεις:

```
short int i;  και  short_t i;  είναι ισοδύναμες
```

- Η σύνταξη μοιάζει με τη δήλωση μεταβλητής (π.χ. όπως θα δηλώναμε τη `short_t`), απλά προηγείται η λέξη `typedef`
- Εναλλακτικά, με την `using` γράφουμε `using short_t = short int`
- Προσοχή, η `typedef` (και το ίδιο ισχύει και με την `using`) δεν δημιουργεί μεταβλητή (δηλαδή, το `short_t` δεν είναι μεταβλητή), ούτε ένα νέο τύπο, απλά ένα νέο όνομα

Δημιουργία Συνωνύμων Τύπων (2)

- Η δυνατότητα να δημιουργούμε ένα συνώνυμο για έναν υπάρχοντα τύπο δεδομένων μπορεί να αποβεί πολύ χρήσιμη. Για παράδειγμα, αν η εφαρμογή σας πρόκειται να εκτελεστεί σε συστήματα που οι τύποι δεδομένων εξαρτώνται από το σύστημα, η δημιουργία συνωνύμων παρέχει μεγάλη ευελιξία
- Για παράδειγμα, με την προηγούμενη δήλωση οι μεταβλητές τύπου `short_t` είναι `short int`. Αν όμως σε κάποιο άλλο σύστημα αυτές οι μεταβλητές πρέπει να είναι `int`, απλά αλλάζετε το `short int` σε `int` και οι μεταβλητές γίνονται `int`
- Βλέπουμε λοιπόν ότι η χρήση των συνωνύμων μας βοηθάει να προσαρμόσουμε τους τύπους μας πολύ εύκολα σε συστήματα με διαφορετικές απαιτήσεις
- Συνήθως, η δημιουργία συνωνύμων τοποθετείται σε κάποιο αρχείο επικεφαλίδας, ώστε να μπορούμε να το συμπεριλάβουμε σε όποιο αρχείο κώδικα επιθυμούμε και να χρησιμοποιήσουμε εκεί αυτά τα συνώνυμα

Παρατηρήσεις (1)

- Πολλές μεταβλητές του ίδιου τύπου μπορούν να δηλωθούν στην ίδια γραμμή, αρκεί να διαχωρίζονται μεταξύ τους με κόμμα (,)
  - Δηλαδή, αντί να δηλώσετε τις μεταβλητές `a`, `b` και `c` σε τρεις ξεχωριστές γραμμές:

```
int a;
int b;
int c;
```

    μπορείτε να τις δηλώσετε σε μία γραμμή ως εξής:

```
int a, b, c;
```

- Η δεσμευμένη λέξη `int` μπορεί να παραληφθεί στους ακέραιους τύπους
  - Π.χ., `short` αντί για `short int`

- Οι δεσμευμένες λέξεις μπορούν να βρίσκονται σε οποιαδήποτε σειρά κατά τη δήλωση μιας μεταβλητής
  - Π.χ., `unsigned long int a;`

    είναι το ίδιο με:

```
int long unsigned a;
```

Παρατηρήσεις (2)

- Όταν δηλώνεται μία μεταβλητή, ο μεταγλωττιστής δεσμεύει τόσα bytes όσα χρειάζονται για να είναι σε θέση να αποθηκεύσει την τιμή της
- Το μέγεθος της μνήμης που δεσμεύει ένας τύπος δεδομένων μπορεί να διαφέρει από υπολογιστή σε υπολογιστή
  - Δηλαδή, ο τύπος `int` μπορεί να δεσμεύει 8 bytes σε κάποιον υπολογιστή και όχι 4 bytes
  - Για να μάθετε πόσες οκτάδες δεσμεύει ένας τύπος δεδομένων στον υπολογιστή που εργάζεστε πρέπει να χρησιμοποιήσετε τον τελεστή `sizeof` → Θα τον δούμε παρακάτω
- Με τη δήλωση μιας μεταβλητής, ο μεταγλωττιστής γνωρίζει το όνομά της και τη διεύθυνση μνήμης στην οποία βρίσκεται η μεταβλητή αυτή, έτσι, όταν η μεταβλητή χρησιμοποιείται στο πρόγραμμα ο μεταγλωττιστής χρησιμοποιεί το όνομά της και γνωρίζοντας την αντίστοιχη διεύθυνση μνήμης έχει πρόσβαση στο περιεχόμενο της μεταβλητής
- Η C++ υποστηρίζει στατικό έλεγχο τύπων *(statically typed language)* με την έννοια ότι ο έλεγχος της εγκυρότητας της χρήσης των τύπων γίνεται κατά τη μεταγλώττιση και όχι κατά την εκτέλεση του προγράμματος

Παρατηρήσεις (3)

- Να χρησιμοποιείτε τον τύπο `float` μόνο όταν η ακρίβεια των δεκαδικών ψηφίων δεν είναι τόσο σημαντική στο πρόγραμμά σας
- Σε περίπτωση που χρειάζεστε υψηλή ακρίβεια των δεκαδικών ψηφίων, να χρησιμοποιείτε τον τύπο `double`
- Π.χ. ποια πιστεύετε θα είναι η έξοδος του παρακάτω προγράμματος;

```
#include <iostream>
int main()
{
        float a = 3.1;

        if(a == 3.1)
                std::cout << "Yes\n";
        else
                std::cout << "No\n";
        return 0;
}
```

Παρατηρήσεις (3 - Συνέχεια)

Μεγάλη ήττα...!!! Περιμέναμε η έξοδος του προγράμματος να είναι Yes, αλλά εμφανίστηκε No;

Και ο λόγος οφείλεται στο περιορισμένο μέγεθος του τύπου `float` να αναπαραστήσει ακριβώς τον αριθμό 3.1, δηλαδή, η τιμή που έχει αποθηκευτεί στο a δεν είναι ακριβώς το 3.1, αλλά μία παραπλήσια τιμή

Να θυμάστε ότι, όταν χρησιμοποιείτε μία πραγματική μεταβλητή σε συγκρίσεις, είναι ασφαλέστερο να την έχετε δηλώσει σαν `double` ή `long double` και όχι `float`. Δηλαδή, αν αντί για `float a;` είχαμε γράψει `double a;` το πρόγραμμα πιθανότατα θα εμφάνιζε Yes

Γενικά, όταν ελέγχετε την τιμή μίας πραγματικής μεταβλητής για ισότητα, δεν μπορείτε να είστε απόλυτα σίγουροι για το αποτέλεσμα της σύγκρισης, αφού υπάρχει το ενδεχόμενο ο αριθμός να μην μπορεί να αναπαρασταθεί με την ακρίβεια που υποστηρίζει ο τύπος. Αν μπορεί να αναπαρασταθεί, το αποτέλεσμα της σύγκρισης είναι έγκυρο

Για παράδειγμα, αν αντί με το 3.1 συγκρίνουμε με το 0.5, η σύγκριση είναι επιτυχής και το πρόγραμμα εμφανίζει Yes, επειδή ο αριθμός 0.5 μπορεί να αναπαρασταθεί με την ακρίβεια του τύπου `float` χωρίς να χαθούν δεκαδικά ψηφία

Παρατηρήσεις (3 - Συνέχεια)

Στη γενική περίπτωση λοιπόν, αν πρέπει να συγκρίνετε για ισότητα δύο πραγματικές τιμές a και b μην γράψετε if(a == b), **δεν** είναι ασφαλές. Ένας σχετικά απλός και σίγουρα πιο ασφαλής τρόπος είναι να ελέγξετε όχι αν οι δύο τιμές είναι ίδιες, αλλά αν η διαφορά τους είναι πολύ μικρή. Για παράδειγμα, μπορούμε να γράψουμε:

```
if(fabs(a-b) <= accuracy)
```

Η `fabs()` είναι συνάρτηση βιβλιοθήκης που δηλώνεται στο cmath και υπολογίζει την απόλυτη τιμή του ορίσματος, δηλαδή, της διαφοράς των a και b. Για τιμή της `accuracy` να επιλέξετε αυτήν που θεωρείτε ότι προσεγγίζει την ισότητα τους

Εκχώρηση Τιμών σε Μεταβλητές (1)

- Η εκχώρηση μίας τιμής σε μία μεταβλητή γίνεται είτε μαζί με τη δήλωση της μεταβλητής είτε αργότερα
- Π.χ. με την πρώτη εντολή δηλώνεται μία ακέραια μεταβλητή (`int`) με όνομα `a` και μετά της εκχωρείται η τιμή `100`

```
int a;
a = 100;
```

- Εναλλακτικά, θα μπορούσαμε να γράψουμε την εκχώρηση τιμής μαζί με τη δήλωση:

`int a = 100;` ή ακόμα και `int a = {100};` ή ακόμα και `int a{100};`

- Επίσης, επιτρέπεται η απόδοση αρχικών τιμών σε περισσότερες από μία μεταβλητές ίδιου τύπου μαζί με τη δήλωσή τους, π.χ.

```
int a = 100, b = 200, c = 300;
```

Εκχώρηση Τιμών σε Μεταβλητές (2)

- Για την εκχώρηση μίας πραγματικής τιμής χρησιμοποιείται η τελεία (.) για το δεκαδικό μέρος και όχι το κόμμα (,) π.χ.

```
float a = 1.24;
```

- Αν μπροστά από μία ακέραια τιμή υπάρχει το ψηφίο 0, τότε αυτή η τιμή ερμηνεύεται σαν οκταδικός αριθμός
  - Π.χ. με την παρακάτω εντολή η τιμή που εκχωρείται στη μεταβλητή a δεν είναι 100, αλλά 64

```
int a = 0100;
```

- Παρομοίως, αν μπροστά από μία ακέραια τιμή υπάρχει το 0x ή το 0X, τότε αυτή η τιμή ερμηνεύεται σαν δεκαεξαδικός αριθμός
  - Π.χ. με την παρακάτω εντολή η τιμή της μεταβλητής a γίνεται 16.

```
int a = 0x10;
```

- Παρομοίως, αν μπροστά από μία ακέραια τιμή υπάρχει το 0b ή το 0B, τότε αυτή η τιμή ερμηνεύεται σαν δυαδικός αριθμός
  - Π.χ. με την παρακάτω εντολή η τιμή της μεταβλητής a γίνεται 15.

```
int a = 0b1111;
```

Παρατηρήσεις (1)

- Η τιμή που εκχωρείται σε μία μεταβλητή πρέπει να συμβαδίζει με τον τύπο της μεταβλητής
  - Π.χ. με την εντολή:

```
int a = 10.9;
```

η τιμή της μεταβλητής a γίνεται 10, γιατί η μεταβλητή a δηλώνεται σαν ακέραια μεταβλητή και όχι σαν πραγματική και το δεκαδικό μέρος αποκόπτεται (Προσοχή!! Δεν στρογγυλοποιείται)

- Η τιμή που εκχωρείται σε μία μεταβλητή πρέπει να είναι μέσα στο επιτρεπτό εύρος τιμών
  - Π.χ. με την εντολή:

```
char ch = 130;
```

η τιμή της μεταβλητής ch δεν γίνεται 130, γιατί το εύρος τιμών μίας μεταβλητής τύπου char είναι από -128 έως 127. Άρα, η τιμή 130 είναι μία τιμή εκτός των επιτρεπτών ορίων

Παρατηρήσεις (2)

- Η τιμή μίας πραγματικής μεταβλητής μπορεί να γραφεί και με επιστημονική σημειογραφία (συνήθως χρησιμοποιείται όταν η τιμή είναι πολύ μικρή ή πολύ μεγάλη)
  - Π.χ. αντί για

```
a = 0.085;
```

μπορούμε να γράψουμε

```
a = 85E-3;
```

- Το γράμμα `E` ή `e` αναπαριστά το 10, ενώ ο αριθμός που το ακολουθεί είναι η θετική ή αρνητική δύναμη του 10.

- Δηλαδή, η έκφραση `85E-3` αντιστοιχεί στον αριθμό $85*10^{-3}$

Σταθερές (1)

- Σταθερά ονομάζεται μία μεταβλητή που η τιμή της δεν μπορεί να αλλάξει μέσα στο πρόγραμμα
- Για μεγαλύτερη ευελιξία, μία καλή πρακτική είναι η χρήση σταθερών στη θέση τιμών που εμφανίζονται πολλές φορές στο πρόγραμμα. Αν στο μέλλον χρειαστεί να αλλάξουμε την τιμή της σταθεράς την αλλάζουμε σε ένα μόνο σημείο
- Για να δηλωθεί μία μεταβλητή σαν σταθερά, χρησιμοποιούμε τη λέξη `const`
- Μία `const` μεταβλητή πρέπει να αρχικοποιηθεί όταν δηλωθεί και η τιμή της δεν επιτρέπεται να αλλάξει στη συνέχεια του προγράμματος
  - Π.χ. με την επόμενη εντολή η ακέραια μεταβλητή `a` δηλώνεται σαν σταθερά και της εκχωρείται (μόνιμα) η τιμή 10

```
const int a = 10;
```

Αν σε κάποιο σημείο του προγράμματος επιχειρήσουμε να της αλλάξουμε τιμή, π.χ. να γράψουμε:

```
a = 100;
```

τότε ο μεταγλωττιστής θα εμφανίσει μήνυμα λάθους για μη επιτρεπτή ενέργεια

Σταθερές (2)

- Εναλλακτικός τρόπος για τη δήλωση μίας σταθεράς είναι η χρήση της οδηγίας **#define**, η οποία χρησιμοποιείται για τη δήλωση μακροεντολών
- Συνήθως, μία μακροεντολή αντιστοιχίζει ένα συμβολικό όνομα με κάποια αριθμητική τιμή
- Για τη δήλωση μακροεντολών, η οδηγία **#define** χρησιμοποιείται ως εξής:

```
#define όνομα_μακροεντολής τιμή
```

- Π.χ. η εντολή:

```
#define NUM 100
```

δηλώνει τη μακροεντολή με όνομα **NUM** και τιμή **100**

- Η **NUM** μπορεί να χρησιμοποιηθεί οπουδήποτε μέσα στο πρόγραμμα
- Ο μεταγλωττιστής όταν συναντάει τη **NUM** μέσα στο πρόγραμμα την αντικαθιστά με την τιμή **100**

Ρητή Μετατροπή Τύπου (type cast)

- Υπάρχουν περιπτώσεις όπου ο προγραμματιστής επιθυμεί να μετατρέψει τον τύπο δεδομένων μίας έκφρασης σε κάποιον άλλο τύπο δεδομένων
- Η γενική μορφή μίας τέτοιας μετατροπής είναι:

`(τύπος_δεδομένων)έκφραση ή τύπος_δεδομένων(έκφραση)`

- Π.χ. έστω η δήλωση: `double a, b, c = 1.23;`

  τότε η εντολή: `a = (int)c;`

  προσαρμόζει **προσωρινά** τον τύπο της `c` από `double` σε `int` και η τιμή του `a` γίνεται `1`
- Η προσαρμογή δεν αλλάζει την τιμή του `c`, δηλαδή, παραμένει ίση με 1.23. Λέγοντας **προσωρινά**, εννοούμε ότι στη συνέχεια του προγράμματος ο τύπος της μεταβλητής `c` συνεχίζει να είναι `double`
- Εκτός από τους παραπάνω τρόπους ρητής προσαρμογής, η C++ υποστηρίζει και άλλους τρόπους με τη χρήση ειδικών τελεστών. Για παράδειγμα, μπορούμε να γράψουμε:

  `a = static_cast<int>(c); // double σε int.`

Παραδείγματα

- Ποια είναι η έξοδος των παρακάτω προγραμμάτων;

```
#include <iostream>
int main()
{
      int i = 100;
      i = i+i;
      i = 2*i;
      std::cout << i+i << ' ' << i << '\n';
      return 0;
}
```

```
#include <iostream>
int main()
{
      int k;
      float i = 3.9, j = 1.2;

      k = i + (int)j;
      std::cout << k - (int)((int)i + j) << '\n';
      return 0;
}
```

Παραδείγματα

```
Έξοδος: 800 400
```

```
Έξοδος: 0
```

C++: Από τη Θεωρία στην Εφαρμογή

Κεφάλαιο 3ο

Είσοδος/Έξοδος Δεδομένων

Έξοδος Δεδομένων με το cout

- Το cout είναι ένα αντικείμενο της κλάσης ostream
- Εξ'ορισμού, το cout συνδέεται με την προκαθορισμένη έξοδο (π.χ. οθόνη)
- Το cout ακολουθείται από τον τελεστή << και την παράσταση που θέλουμε να εμφανιστεί
- Η C++ υποστηρίζει αρκετούς τρόπους για τη μορφοποίηση της πληροφορίας

Ακολουθίες Διαφυγής

- Για την αναπαράσταση μη εκτυπώσιμων χαρακτήρων (π.χ. Enter) και χαρακτήρων με ειδική σημασία (π.χ. "), η C++ παρέχει τις ακολουθίες διαφυγής (escape sequences)
- Μία ακολουθία διαφυγής αποτελείται από τον χαρακτήρα \ και έναν ειδικό χαρακτήρα. Ο παρακάτω πίνακας εμφανίζει κάποιες συνηθισμένες ακολουθίες διαφυγής

Ακολουθία διαφυγής	Σημασία
`\a`	Χρησιμοποιείται για τη δημιουργία ηχητικού σήματος.
`\b`	Χρησιμοποιείται για τη διαγραφή του τελευταίου χαρακτήρα, όπως το πλήκτρο backspace.
`\n`	Χρησιμοποιείται για την αλλαγή γραμμής, όπως το πλήκτρο Enter.
`\r`	Χρησιμοποιείται για την επαναφορά του δρομέα στην αρχή της τρέχουσας γραμμής.
`\t`	Χρησιμοποιείται για τη μετακίνηση του δρομέα σε μία απόσταση ίση με το μήκος του tab, όπως το πλήκτρο tab.
`\\`	Χρησιμοποιείται για την εμφάνιση της ανάστροφης κεκλιμένης (\).
`\"`	Χρησιμοποιείται για την εμφάνιση των διπλών εισαγωγικών (").

Χειριστές

- Ένας βολικός τρόπος για τη μορφοποίηση της πληροφορίας είναι με τη χρήση χειριστών (manipulators). Οι χειριστές δηλώνονται στον std χώρο ονομάτων. Ο παρακάτω πίνακας εμφανίζει κάποιους συνηθισμένους χειριστές

Χειριστής	Λειτουργία
`endl`	Προκαλεί αλλαγή γραμμής. Το `endl` αδειάζει άμεσα την ενδιάμεση μνήμη εξόδου.
`dec`	Οι τιμές διαβάζονται/εμφανίζονται σε δεκαδική μορφή.
`hex`	Οι τιμές διαβάζονται/εμφανίζονται σε δεκαεξαδική μορφή.
`oct`	Οι τιμές διαβάζονται/εμφανίζονται σε οκταδική μορφή.
`left`	Η έξοδος στοιχίζεται αριστερά με την εισαγωγή χαρακτήρων συμπλήρωσης στο τέλος
`right`	Η έξοδος στοιχίζεται δεξιά με την εισαγωγή χαρακτήρων συμπλήρωσης στην αρχή.
`fixed`	Οι δεκαδικές τιμές εμφανίζονται σε μορφή σταθερής υποδιαστολής.
`scientific`	Οι δεκαδικές τιμές εμφανίζονται σε επιστημονική μορφή.
`showbase`	Προσθέτει τη βάση του αριθμητικού συστήματος (`0`, `0x`) πριν από την τιμή.
`showpos`	Προσθέτει `+` πριν από θετικές τιμές.
`skipws`	Τα λευκά διαστήματα (π.χ. κενά, στηλοθέτες) που μπορεί να προηγούνται των δεδομένων εισόδου αγνοούνται.
`setfill(c)`	Αν η τιμή δε χωράει στο πλάτος του πεδίου, οι υπόλοιπες θέσεις συμπληρώνονται με το χαρακτήρα συμπλήρωσης `c`.
`setprecision(n)`	Ο ακέραιος `n` καθορίζει τα ψηφία της ακρίβειας. Η ακρίβεια παραμένει σε ισχύ για τις επόμενες πράξεις εξόδου.
`setw(n)`	Ο ακέραιος `n` καθορίζει το πλάτος του επόμενου πεδίου. Αν το `n` είναι μικρότερο από τους χαρακτήρες που απαιτούνται για την εμφάνιση του πεδίου δεν λαμβάνεται υπόψη. Το πλάτος παραμένει σε ισχύ μόνο για το επόμενο πεδίο.
`uppercase`	Εμφανίζει τα `e` και `x` με κεφαλαία.

Παράδειγμα (1)

- Ποια είναι η έξοδος του παρακάτω προγράμματος;

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

int main()
{
     int i = 100;
     double j = 100.536;

     cout << i << oct << '\t' << i << hex << '\t' << i << endl;
     cout << scientific << j << '\t' << fixed << j << endl;
     cout << setprecision(2) << j << '\t' << setprecision(0) << '\t' << j
<< endl;
     cout << showbase << uppercase << i << '\t' << 1.4999 << '\t' <<
showpos << dec << i << endl;
     return 0;
}
```

Παράδειγμα (1)

```
100       144       64
1.005360e+002       100.536000
100.54              101
0x64      1         +100
```

Παράδειγμα (2)

- Ποια είναι η έξοδος του παρακάτω προγράμματος;

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

int main()
{
     int i = 200;
     double j = 1.23497654;

     cout << setw(10) << setfill('*') << i << '\n';
     cout << setw(2) << i << ' ' << setw(7) << fixed << j << '\n';
     cout << setw(8) << setprecision(3) << left << j << '\n';
     cout << setw(7) << showpos << setfill('#') << internal << i << '\n';
     return 0;
}
```

Παράδειγμα (2)

```
********200
200 1.234977
1.235***
+200###
```

Είσοδος Δεδομένων με το `cin`

- Το `cin` είναι ένα αντικείμενο της κλάσης `istream`
- Εξ'ορισμού, το `cin` συνδέεται με την προκαθορισμένη είσοδο (π.χ. πληκτρολόγιο)
- Το `cin` ακολουθείται από τον τελεστή `>>` και την μεταβλητή στην οποία θα αποθηκευτεί η τιμή που θα διαβαστεί
- Όπως και με το `cout`, μπορούμε να χρησιμοποιήσουμε στην ίδια `cin` εντολή πολλές φορές τον τελεστή `>>` για διάβασμα τιμών. Για παράδειγμα, μπορούμε να γράψουμε `cin >> i >> j;`

Παράδειγμα

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

int main()
{
      int i;
      double j;

      cout << "Enter number: ";
      cin >> i; // Διάβασμα ακεραίου και αποθήκευσή του στο i.
      cout << "Enter number: ";
      cin >> j;
      cout << i << '\t' << fixed << j << '\n';
      return 0;
}
```

- Όταν εκτελείται η **cin** εντολή, το πρόγραμμα περιμένει τον χρήστη να πληκτρολογήσει μία τιμή. Συνήθως, πριν από την **cin** εντολή υπάρχει μία **cout** εντολή, που υποδεικνύει στον χρήστη τι δεδομένα πρέπει να εισάγει
- Όταν ο χρήστης εισάγει μία τιμή και πατήσει *Enter*, αυτή η τιμή θα αποθηκευτεί στην αντίστοιχη μεταβλητή

Είσοδος Δεδομένων με Συγκεκριμένη Μορφή

- Υπάρχουν περιπτώσεις, όπου μπορεί να ζητηθεί από τον χρήστη να εισάγει τα δεδομένα με μία συγκεκριμένη μορφή. Για παράδειγμα, ένα πρόγραμμα που απαιτεί η εισαγωγή της ημερομηνίας να γίνεται με τη μορφή ημέρα/μήνας/χρόνος

- Ένας τρόπος για να εξάγουμε τις τιμές που μας ενδιαφέρουν και να τις εκχωρήσουμε σε αντίστοιχες μεταβλητές είναι να χρησιμοποιήσουμε μεταβλητή(ές) με σκοπό την εξαγωγή των χαρακτήρων που δεν μας ενδιαφέρουν. Για παράδειγμα, η μεταβλητή `ch` χρησιμοποιείται για την εξαγωγή του /

```
char ch;
int d, m, y;
cout << "Enter date in the form d/m/y: ";
cin >> d >> ch >> m >> ch >> y;
```

- Βέβαια, προϋπόθεση για να λειτουργήσει σωστά ο παραπάνω κώδικας είναι ο χρήστης να εισάγει την ημερομηνία με τον ίδιο τρόπο (π.χ. 19/6/2030).

C++: Από τη Θεωρία στην Εφαρμογή

Κεφάλαιο 4ο
Τελεστές

Ο Τελεστής Εκχώρησης =

- Ο τελεστής = χρησιμοποιείται για την εκχώρηση τιμής σε μία μεταβλητή. Π.χ. με την εντολή: `a = 10;` η τιμή του `a` γίνεται ίση με `10`, ενώ με την εντολή `a = b;` η τιμή του `a` γίνεται ίση με την τιμή του `b`

- Αν χρησιμοποιηθεί διαδοχικά σε μία εντολή εκχώρησης, τότε η τελική τιμή εκχώρησης αποθηκεύεται σε όλες τις μεταβλητές (ο τελεστής εφαρμόζεται από δεξιά προς τα αριστερά). Π.χ. με την εντολή: `a = b = c = 10;` οι τιμές των μεταβλητών `a`, `b` και `c` γίνονται ίσες με `10`

- Αν ο τύπος της μεταβλητής και της εκχωρούμενης τιμής δεν είναι ίδιος, τότε, η τιμή πρώτα μετατρέπεται στον τύπο της μεταβλητής, εφόσον αυτό είναι δυνατό, και μετά εκχωρείται σε αυτήν. Π.χ.

```
int a;
float b;
b = a = 10.96;
```

Ποιες τιμές αποθηκεύονται στις μεταβλητές `a` και `b`;

Παρατηρήσεις

- Ο δεξιός τελεστέος είναι μία έκφραση που έχει τιμή, όπως μία μεταβλητή ή μία σταθερά. Ο αριστερός τελεστέος πρέπει να είναι μία *lvalue (left value – αριστερή τιμή)*. Μία *lvalue* αναφέρεται σε μία οντότητα που είναι αποθηκευμένη στη μνήμη (π.χ. μεταβλητή) στην οποία μπορούμε να αποθηκεύσουμε κάτι. Π.χ, δεν επιτρέπεται να είναι σταθερά ή το αποτέλεσμα ενός υπολογισμού:

```
20 = a; // Λάθος
b-a = 30; // Λάθος
```

Στις παραπάνω περιπτώσεις ο μεταγλωττιστής θα εμφανίσει μήνυμα λάθους της μορφής: `"error '=' : left operand must be lvalue"`

- Επίσης, η *lvalue* πρέπει να είναι τροποποιήσιμη. Για παράδειγμα, θα δούμε στη συνέχεια ότι το όνομα πίνακα δεν είναι μία τροποποιήσιμη *lvalue*

- Το συμπληρωματικό της *lvalue* έννοιας είναι η *rvalue (right value – δεξιά τιμή)*. Σαν γενικός κανόνας, μια *rvalue* είναι μια έκφραση που δεν είναι *lvalue*, όπως μια σταθερή τιμή (π.χ. 10) ή μία έκφραση (π.χ. a+b)

Αριθμητικοί Τελεστές (1/3)

- Οι μαθηματικοί τελεστές +, -, *, / χρησιμοποιούνται για την εκτέλεση των γνωστών μαθηματικών πράξεων
- Αν και οι δύο τελεστέοι είναι ακέραιοι, τότε ο τελεστής / αποκόπτει το δεκαδικό μέρος. Π.χ.,

```
int a = 3, b = 2, c;
c = a/b;
```

η τιμή της έκφρασης a/b που ανατίθεται στη μεταβλητή c είναι 1 και όχι 1.5

- Αν ένας από τους δύο τελεστέους είναι πραγματική σταθερά ή μεταβλητή, το δεκαδικό μέρος δεν αποκόπτεται. Π.χ.:

```
int a = 8;
float b = 5;
```

το αποτέλεσμα a/b είναι 1.6, αφού η μεταβλητή b είναι float και το αποτέλεσμα a/10.0 είναι 0.8

Αριθμητικοί Τελεστές (2/3)

- Ο τελεστής % χρησιμοποιείται για τον υπολογισμό **του υπολοίπου της διαίρεσης** δύο ακεραίων τελεστέων. Π.χ. στο επόμενο παράδειγμα:

```
int a, b, c, d;
a = 11;
b = 3;
c = a%b;
d = a%c%b;
```

η τιμή της μεταβλητής `c` είναι `2` και η τιμή της μεταβλητής `d` είναι `1`

- **Προσοχή!** Ο τελεστής % μπορεί να εφαρμοστεί μόνο μεταξύ ακεραίων αριθμών, αλλιώς ο μεταγλωττιστής εμφανίζει μήνυμα λάθους

Αριθμητικοί Τελεστές (3/3)

- Οι πέντε αυτοί τελεστές λέγονται και «δυαδικοί» (binary), επειδή απαιτούν δύο τελεστέους
- Η C++ παρέχει και τους μοναδιαίους (unary) τελεστές + και – που απαιτούν μόνο έναν τελεστέο
  - Ο μοναδιαίος τελεστής + δεν έχει κάποια επίδραση στον τελεστέο
  - Το αποτέλεσμα με τον μοναδιαίο – είναι η αντίθετη τιμή του τελεστέου
- Όταν σε κάποια έκφραση περιέχονται και μοναδιαίοι και δυαδικοί τελεστές + και –, ο μεταγλωττιστής αντιλαμβάνεται το πώς εφαρμόζονται. Π.χ.:

```
int a = +20; // Ο μοναδιαίος τελεστής + δεν έχει κάποια επίδραση.
a = -10; // Το - είναι εδώ ο μοναδιαίος τελεστής.
int b = -(a-5); /* Το «μέσα -» αντιστοιχεί στον δυαδικό τελεστή ενώ
   το «έξω -» στον μοναδιαίο, αντίστοιχα. */
-a; /* Δεν επιφέρει κάποιο αποτέλεσμα στη μεταβλητή a (δεν γίνεται
   κάποια ανάθεση), άρα η τιμή της a παραμένει ως έχει.*/
```

Η μεταβλητή b αποκτά την τιμή 15 και η a την τιμή -10

Ο Τελεστής Αύξησης ++ (1/2)

- Ο τελεστής αύξησης ++ εισάγεται πριν ή μετά από το όνομα του τελεστέου
- Σε κάθε περίπτωση η τιμή του τελεστέου αυξάνεται κατά ένα
- Ο τελεστέος πρέπει να είναι μία τροποποιήσιμη «αριστερή-τιμή» *(lvalue)*, όπως μία μεταβλητή. Π.χ.

```
int a = 10;
a++; // Ισοδύναμη με a = a+1;
++a; // Ισοδύναμη με a = a+1;
(a+10)++; // Λάθος
```

Η τιμή του a αυξάνεται δύο φορές κατά ένα και γίνεται ίση με 12.

Ο Τελεστής Αύξησης ++ (2/2)

- Στην επιθεματική μορφή, η αύξηση πραγματοποιείται αφού πρώτα χρησιμοποιηθεί η τιμή της μεταβλητής στην έκφραση. Π.χ., με τις εντολές:

```
int a = 10, b;
b = a++;
```

πρώτα αποθηκεύεται στη μεταβλητή b η τρέχουσα τιμή του a και μετά η τιμή του a αυξάνεται. Επομένως, το a θα γίνει 11 και το b ίσο με 10

- Στην προθεματική μορφή, πρώτα αυξάνεται η τιμή της μεταβλητής και μετά αυτή χρησιμοποιείται στην έκφραση. Π.χ., με τις εντολές:

```
int a = 10, b;
b = ++a;
```

πρώτα αυξάνεται η τιμή του a και μετά αυτή αποθηκεύεται στο b. Επομένως, και οι δύο μεταβλητές θα γίνουν ίσες με 11

Ο Τελεστής Μείωσης --

- Ο τελεστής μείωσης -- εισάγεται πριν ή μετά από το όνομα μίας μεταβλητής
- Σε κάθε περίπτωση η τιμή της μεταβλητής μειώνεται κατά ένα
- Σε μία έκφραση, ο τελεστής -- συμπεριφέρεται όπως και ο τελεστής ++. Π.χ.

```
double a = 1.23;
cout << ++a << '\n';
cout << a-- << '\n';
```

Πρώτα η τιμή του a αυξάνεται και ο κώδικας εμφανίζει 2.23. Μετά, πρώτα ο κώδικας εμφανίζει την τιμή του a, δηλαδή 2.23, και μετά αυτή μειώνεται

Παρατηρήσεις

- Αν εφαρμόσετε τους τελεστές ++ και -- για να αλλάξετε την τιμή μιας μεταβλητής σε μία έκφραση, μην χρησιμοποιήσετε ξανά τη μεταβλητή στην ίδια έκφραση, επειδή η σειρά αποτίμησης είναι απροσδιόριστη

- Π.χ., το αποτέλεσμα της έκφρασης:

```
a * a++;
```

είναι απροσδιόριστο, εξαρτάται από τον μεταγλωττιστή αν πρώτα αυξήσει την τιμή του a και μετά κάνει τον πολ/σμό, ή όχι.

Σχεσιακοί Τελεστές (1/2)

- Οι σχεσιακοί τελεστές >, >=, <, <=, !=, ==, χρησιμοποιούνται για τη σύγκριση των τιμών δύο τελεστέων

- Συνήθως χρησιμοποιούνται σε εντολές ελέγχου (π.χ. στην εντολή `if`) και σε επαναληπτικούς βρόχους (π.χ. στην εντολή `for`)

- Π.χ.
  `if(a > 5)` ελέγχουμε αν η τιμή του a είναι μεγαλύτερη από το 5.
  `if(a >= 5)` ελέγχουμε αν η τιμή του a είναι μεγαλύτερη ή ίση από το 5.
  `if(a < 5)` ελέγχουμε αν η τιμή του a είναι μικρότερη από το 5.
  `if(a <= 5)` ελέγχουμε αν η τιμή του a είναι μικρότερη ή ίση από το 5.
  `if(a != 5)` ελέγχουμε αν η τιμή του a είναι διαφορετική από το 5.
  `if(a == 5)` ελέγχουμε αν η τιμή του a είναι ίση με το 5.

Σχεσιακοί Τελεστές (2/2)

- Μία έκφραση χαρακτηρίζεται αληθής (true), όταν η τιμή της είναι διαφορετική από το μηδέν, ενώ - αν είναι μηδέν- χαρακτηρίζεται ψευδής (false)

- Το αποτέλεσμα που παράγουν οι σχεσιακοί τελεστές είναι τύπου `bool`, δηλαδή, `true` ή `false`. Π.χ. το αποτέλεσμα της έκφρασης (`a > 10`) είναι `true` μόνο αν η τιμή της μεταβλητής `a` είναι μεγαλύτερη από το 10, αλλιώς είναι `false`

- Ποια είναι η έξοδος του διπλανού προγράμματος:

```
#include <iostream>
int main()
{
        int a = 4, b = 6, c;

        a = (a <= (b-2)) + (b > (a+1));
        b = (a == 2) > ((b-3) < 3);
        c = (b != 0);
        std::cout << a << ' ' << b << ' ' << c << '\n';
        return 0;
}
```

Σχεσιακοί Τελεστές (2/2)

```
Έξοδος: 2  1  1
```

Παρατηρήσεις

- Μην συγχέετε τον τελεστή **ελέγχου ισότητας** (==) με τον τελεστή **εκχώρησης** (=)
- Ο τελεστής == χρησιμοποιείται για να ελέγξουμε αν δύο εκφράσεις έχουν την ίδια τιμή, ενώ ο τελεστής = χρησιμοποιείται για την εκχώρηση τιμής
- Π.χ., η τιμή της έκφρασης:

```
a == 5
```

είναι `true`, μόνο η τιμή της μεταβλητής `a` είναι `5`, αλλιώς είναι `false`

ενώ η τιμή της έκφρασης:

```
a = 5
```

εκχωρεί την τιμή `5` στο `a`

Συνδυαστικοί Τελεστές

- Οι συνδυαστικοί τελεστές χρησιμοποιούνται για να γραφούν εκφράσεις με συμπτυγμένο τρόπο, βάσει του παρακάτω τύπου:

```
exp1 op= exp2;
```

όπου συνήθως ο τελεστής op είναι κάποιος από τους αριθμητικούς τελεστές +, -, *, %, / ή κάποιος από τους τελεστές bit που θα δούμε παρακάτω (&, ^, |, <<, >>). Η παραπάνω έκφραση είναι ισοδύναμη με:

```
exp1 = exp1 op (exp2);
```

- Π.χ. η έκφραση:

```
a += b;
```

είναι ισοδύναμη με:

```
a = a + b;
```

ενώ η έκφραση:

```
a *= 3;
```

είναι ισοδύναμη με:

```
a = a * 3;
```

Παράδειγμα

- Ποια είναι η έξοδος του παρακάτω προγράμματος;

```
#include <iostream>
int main()
{
    int a = 1, b = 2;

    a -= 2;
    a *= 1-b;
    a += b+3;
    a /= b+2;
    a %= b;
    std::cout << a << '\n';
    return 0;
}
```

Παράδειγμα

```
Έξοδος: 1
```

Λογικοί Τελεστές (1)

Ο τελεστής !

- Ο τελεστής ! είναι μοναδιαίος, δηλαδή εφαρμόζεται σε έναν μόνο τελεστέο
- Αν μία έκφραση exp είναι αληθής (δηλαδή έχει μη μηδενική τιμή), τότε το αποτέλεσμα της πράξης !exp είναι false
- Αν μία έκφραση exp είναι ψευδής (δηλαδή έχει μηδενική τιμή), τότε το αποτέλεσμα της πράξης !exp είναι true
- Συνήθως, ο τελεστής ! χρησιμοποιείται σε συνθήκες ελέγχου στην εντολή if. Π.χ.

  η εντολή: `if(!a)` είναι ισοδύναμη με `if(a == 0)`

  και η εντολή: `if(a)` είναι ισοδύναμη με `if(a != 0)`

Λογικοί Τελεστές (2)

Ο τελεστής &&

- Ο τελεστής && εκτελεί τη λογική πράξη AND μεταξύ δύο τελεστέων
- Η τιμή της έκφρασης είναι **true**, μόνο αν και οι δύο τελεστέοι είναι αληθείς. Αλλιώς, η τιμή της είναι **false**
- Αν ο όρος που αποτιμάται σε μία έκφραση με τον τελεστή && είναι ψευδής, ο μεταγλωττιστής δεν αποτιμά τους επόμενους όρους *(short circuit evaluation)* και θέτει κατευθείαν την τιμή της συνολικής έκφρασης ίση με **false**. Π.χ. στο επόμενο πρόγραμμα η τιμή του **b** δεν αυξάνεται, γιατί ο πρώτος όρος (a < 2) έχει ψευδή τιμή:

```
#include <iostream>
int main()
{
    int a = 5, b = 8, c;
    c = (a < 2) && (++b > 3);
    std::cout << c << ' ' << b << '\n';
    return 0;
}
```

Επομένως, το πρόγραμμα εμφανίζει: 0 8

Λογικοί Τελεστές (3)

Ο τελεστής ||

- Ο τελεστής || εκτελεί τη λογική πράξη OR μεταξύ δύο τελεστέων
- Η τιμή της έκφρασης είναι `true`, αν έστω και ένας από τους δύο τελεστέους είναι αληθής. Αν και οι δύο τελεστέοι είναι ψευδείς, η τιμή της έκφρασης είναι `false`
- Όπως και ο τελεστής &&, ο τελεστής || εφαρμόζει *short circuit* αποτίμηση. Αν ο όρος που αποτιμάται είναι αληθής, ο μεταγλωττιστής δεν αποτιμά τους επόμενους όρους και θέτει κατευθείαν την τιμή της συνολικής έκφρασης ίση με `true`. Π.χ., στο επόμενο πρόγραμμα η τιμή του `b` δεν αυξάνεται, γιατί ο πρώτος όρος `(a > 2)` είναι αληθής:

```
#include <iostream>
int main()
{
    int a = 5, b = 8, c;
    c = (a > 2) || (++b > 3);
    std::cout << c << ' ' << b << '\n';
    return 0;
}
```

Επομένως, το πρόγραμμα εμφανίζει: `1 8`

Ο Τελεστής κόμμα ,

- Ο τελεστής κόμμα (,) χρησιμοποιείται για τη σύνδεση εκφράσεων, ώστε να σχηματιστεί μία μοναδική έκφραση:

```
exp1, exp2, exp3,...
```

- Επειδή ο τελεστής κόμμα (,) έχει αριστερή συσχέτιση, πρώτη εκτελείται η exp1, μετά η exp2, μετά η exp3, έως και την τελευταία έκφραση. Π.χ., αφού οι εκφράσεις εκτελούνται από αριστερά προς τα δεξιά ο παρακάτω κώδικας θα εμφανίσει 30:

```
int b;
b = 10, b += 20, cout << b;
```

- Ο τύπος και η τιμή της συνολικής έκφρασης είναι ίδιοι με τον τύπο και την τιμή της τελευταίας έκφρασης

Παράδειγμα

- Ποια είναι η έξοδος του παρακάτω προγράμματος?

```
#include <iostream>
int main()
{
    int a, b, c;

    a = (b = 30, b = b/8, ++b);
    std::cout << a << ' ' << b << '\n';
    c = (b != 4);
    if(a, b, c)
        std::cout << "One\n";

    return 0;
}
```

Παράδειγμα

- Απάντηση. Με τις δύο πρώτες εκφράσεις η τιμή του b γίνεται 3. Με την έκφραση ++b πρώτα θα αυξηθεί η τιμή του b και μετά αυτή θα αποθηκευτεί στο a. Άρα, το πρόγραμμα θα εμφανίσει: 4 4. Αφού το b είναι 4, η συνθήκη είναι ψευδής και το c γίνεται 0. Στη συνέχεια, επειδή η τιμή της συνολικής έκφρασης είναι η τιμή του c και επειδή αυτή είναι 0, το πρόγραμμα δε θα εμφανίσει τίποτα.

Παρατηρήσεις

- Επειδή η χρήση του τελεστή κόμμα μπορεί να παράξει δυσνόητο και δυσκολότερο να ελεγχθεί κώδικα, δεν χρησιμοποιείται συχνά
- Η συνηθέστερη χρήση του είναι στην εντολή `for`, π.χ.:

```
int a, b;
for(a = 1, b = 2; b < 10; a++, b++)
```

- Στο παραπάνω κομμάτι κώδικα στην πρώτη έκφραση εντός της παρένθεσης, πρώτα η τιμή της `a` γίνεται `1` και μετά η τιμή της `b` γίνεται `2`, ενώ, στην τρίτη έκφραση εντός της παρένθεσης πρώτα εκτελείται η εντολή `a++` και μετά η `b++`
- Σημειώστε ότι το κόμμα που χρησιμοποιούμε για να ξεχωρίζουν μεταβλητές σε δηλώσεις, ορίσματα συναρτήσεων και τιμές σε λίστες αρχικοποίησης δεν είναι ο τελεστής, αλλά το απλό διαχωριστικό

Ο Τελεστής sizeof

- Ο τελεστής sizeof υπολογίζει το μέγεθος, σε οκτάδες, κάποιου τύπου δεδομένων, μίας μεταβλητής, σταθεράς ή έκφρασης. Π.χ., το επόμενο πρόγραμμα χρησιμοποιεί τον τελεστή sizeof για να εμφανίσει πόσες οκτάδες δεσμεύουν οι μεταβλητές του προγράμματος:

```
#include <iostream>
int main()
{
    char c;
    int i;
    double d;

    std::cout << sizeof(c) << ' ' << sizeof(i) << ' ' <<
sizeof(d) << '\n';
    return 0;
}
```

Ο Τύπος enum (1)

- Ο τύπος απαρίθμησης enum *(enumeration type)* χρησιμοποιείται για να οριστεί ένα σύνολο ακεραίων με συγκεκριμένα ονόματα και σταθερές τιμές
- Τα ονόματα πρέπει να διαφέρουν μεταξύ τους καθώς και με ονόματα μεταβλητών στην ίδια εμβέλεια, αλλά οι τιμές τους δεν χρειάζεται να είναι διαφορετικές

- Συνήθως, δηλώνεται ως εξής:

```
enum όνομα {λίστα απαρίθμησης};
```

- Το αναγνωριστικό όνομα είναι προαιρετικό και δηλώνει το όνομα της απαρίθμησης, π.χ. η εντολή:

```
enum Seasons {AUTUMN, WINTER, SPRING, SUMMER};
```

δηλώνει τον τύπο απαρίθμησης Seasons και τις ακέραιες σταθερές AUTUMN, WINTER, SPRING και SUMMER

- Θυμηθείτε ότι με την οδηγία #define ή με το προσδιοριστικό const μπορούμε επίσης να δηλώσουμε σταθερές. Η κύρια διαφορά τους με τον τύπο enum είναι ότι ο τύπος enum ομαδοποιεί τις σταθερές, ώστε να φαίνεται ότι χαρακτηρίζουν ένα σύνολο τιμών

Ο Τύπος enum (2)

- Εξ'ορισμού, κατά τη δήλωση ενός τύπου απαρίθμησης, η τιμή της πρώτης σταθεράς αρχικοποιείται με 0
- Αν σε κάποια σταθερά δεν αποδίδεται τιμή, η τιμή της γίνεται ίση με την τιμή της προηγούμενης σταθεράς αυξημένη κατά ένα
- Επομένως, στο προηγούμενο παράδειγμα που δεν αποδίδονται τιμές στις σταθερές, οι τιμές των σταθερών **AUTUMN**, **WINTER**, **SPRING** και **SUMMER** γίνονται 0, 1, 2 και 3, αντίστοιχα
- Σε περίπτωση που στο προηγούμενο παράδειγμα θα θέλαμε να αποδώσουμε συγκεκριμένες τιμές στις σταθερές, θα μπορούσαμε να δηλώσουμε τον τύπο απαρίθμησης π.χ. ως εξής:

```
enum Seasons {AUTUMN=10, WINTER, SPRING=30, SUMMER};
```

- Στο παράδειγμα αυτό οι τιμές των σταθερών **AUTUMN**, **WINTER**, **SPRING** και **SUMMER** γίνονται 10, 11, 30 και 31, αντίστοιχα

Δήλωση Μεταβλητής Τύπου enum

- Για να δηλώσουμε μία μεταβλητή σύμφωνα με έναν ήδη δηλωμένο τύπο απαρίθμησης γράφουμε:

```
όνομα_τύπου λίστα_μεταβλητών;
```

- Π.χ. με την εντολή:

```
Seasons s1, s2;
```

δηλώνουμε τις s1 και s2 σαν μεταβλητές απαρίθμησης του τύπου Seasons του προηγούμενου παραδείγματος

- Εναλλακτικά, μπορούμε να δηλώσουμε τις μεταβλητές μαζί με τη δήλωση του τύπου απαρίθμησης, π.χ.:

```
enum Seasons {AUTUMN, WINTER, SPRING, SUMMER} s1, s2;
```

Οι Τελεστές bit

- Οι τελεστές bit χρησιμοποιούνται για το χειρισμό των bits μίας ακέραιας μεταβλητής ή σταθεράς
- Η τιμή ενός bit μπορεί να είναι 0 ή 1
- Ο υπολογισμός της τιμής μίας έκφρασης που περιέχει τελεστές bit γίνεται με την εφαρμογή τους στα αντίστοιχα bits των τελεστέων
- Οι τελεστές bit είναι οι εξής:
  - Ο τελεστής AND &
  - Ο τελεστής OR |
  - Ο τελεστής XOR ^
  - Ο τελεστής NOT ~
- Όταν εκτελείτε πράξεις με τελεστές bit είναι ασφαλέστερο να τους εφαρμόζετε σε `unsigned` μεταβλητές, αλλιώς, να λαμβάνετε υπόψη σας το bit προσήμου, όταν κάνετε τους υπολογισμούς σας

Ο Τελεστής &

- Ο τελεστής & εφαρμόζει τη λογική πράξη AND (λογική πράξη ΚΑΙ) στα bits των δύο τελεστέων και θέτει το bit εξόδου στο 1 μόνο αν τα αντίστοιχα bits και στους δύο τελεστέους είναι 1, αλλιώς, το bit εξόδου τίθεται στο 0
- Π.χ. το αποτέλεσμα της πράξης 19 & 2 είναι 2

```
        00010011 (19)
&       00000010 (2)
        ------------
        00000010 (2)
```

- Ο τελεστής & χρησιμοποιείται συχνά για να μηδενίσει, δηλ. να θέσει ίσα με το μηδέν (0), μια σειρά από bits
  - Π.χ., η εντολή: `a = a & 3;` κάνει 0 όλα τα bits της a εκτός από τα δύο «λιγότερα σημαντικά» bits
- Μην συγχέετε τους τελεστές && και & (π.χ., αν a=1 και b=2, το αποτέλεσμα της έκφρασης `a && b` είναι 1, ενώ το αποτέλεσμα της έκφρασης `a & b` είναι 0)

Ο Τελεστής |

- Ο τελεστής | εφαρμόζει τη λογική πράξη OR (λογική πράξη Ή) στα bits των δύο τελεστέων και θέτει το bit εξόδου στο 0 μόνο αν τα αντίστοιχα bits και στους δύο τελεστέους είναι 0, αλλιώς, το bit εξόδου τίθεται στο 1
- Π.χ. το αποτέλεσμα της πράξης 19 | 6 είναι 23

```
    00010011 (19)
|   00000110 (6)
    ------------
    00010111 (23)
```

- Ο τελεστής | χρησιμοποιείται συχνά για να θέσει ίσα με το ένα (1) μια σειρά από bits
  - Π.χ., η εντολή: a = a | 3; κάνει τα δύο «λιγότερα σημαντικά» bits της μεταβλητής a ίσα με ένα
- Μην συγχέετε τους τελεστές || και | (π.χ., αν a=1 και b=2, το αποτέλεσμα της έκφρασης a || b είναι 1, ενώ το αποτέλεσμα της έκφρασης a | b είναι 3)

Ο Τελεστής ^

- Ο τελεστής ^ εφαρμόζει τη λογική πράξη XOR (eXclusive OR , αποκλειστικό Ή) στα bits των δύο τελεστέων και θέτει το bit εξόδου στο 1 μόνο αν τα αντίστοιχα bits και στους δύο τελεστέους είναι διαφορετικά μεταξύ των, αλλιώς, το bit τίθεται στο 0

- Π.χ. το αποτέλεσμα της πράξης 19 ^ 6 είναι 21

```
      00010011 (19)
^     00000110 (6)
      -----------
      00010101 (21)
```

Ο Τελεστής ~

- Ο τελεστής συμπληρώματος ~ είναι μοναδιαίος, δηλαδή εφαρμόζεται σε έναν τελεστέο και εφαρμόζει τη λογική πράξη NOT (λογική πράξη ΔΕΝ)

- Συγκεκριμένα, αντιστρέφει κάθε bit στον τελεστέο του, αλλάζοντας όλα τα 0 σε 1, και αντιστρόφως

- Π.χ. σε ένα 32-bit σύστημα το αποτέλεσμα της πράξης ~19 είναι $(2^{32} - 1) - 19$

```
~ 00000000 00000000 00000000 00010011 (19) =
  11111111 11111111 11111111 11101100
```

Οι Τελεστές Ολίσθησης

- Οι τελεστές ολίσθησης (>> και <<) μετατοπίζουν τα bits μίας ακέραιας μεταβλητής ή σταθεράς κατά ένα συγκεκριμένο αριθμό θέσεων, όπως δείχνουν τα «νοητά βέλη»

- Ο τελεστής >> μετατοπίζει τα bits της μεταβλητής προς τα δεξιά, όπως δηλαδή δείχνουν τα «νοητά βέλη»

- Ο τελεστής << μετατοπίζει τα bits της μεταβλητής προς τα αριστερά, όπως δηλαδή δείχνουν τα «νοητά βέλη»

Ο Τελεστής >>

- Η έκφραση `i >> n` μετατοπίζει τα bits της μεταβλητής `i` κατά `n` θέσεις δεξιά. Αν η τιμή του `i` είναι θετική ή ο τύπος του είναι unsigned, οι τιμές των n υψηλότερης τάξης bits που εισάγονται στα αριστερά είναι μηδέν
- Επειδή η θέση ενός bit αντιστοιχεί σε μία δύναμη του 2, όταν ένας θετικός ακέραιος ολισθαίνει n θέσεις δεξιά η τιμή του διαιρείται με 2^n
- Π.χ. ποια θα είναι η τιμή της μεταβλητής `a` κατά την εκτέλεση του παρακάτω κώδικα;

```
unsigned int a, b = 35;
a = b >> 2;
```

`a = 8`, διότι: `00100011 >> 2 = 00001000`

Συγκεκριμένα, χάθηκαν τα τελευταία bits `1` και `1` του αρχικού αριθμού (35) και τοποθετήθηκαν τα bits `0` και `0` στην έβδομη και όγδοη θέση, αντίστοιχα

- Και ποια είναι η τιμή της μεταβλητής `b`;

Ο Τελεστής <<

- Η έκφραση `i << n` μετατοπίζει τα bits της μεταβλητής `i` κατά `n` θέσεις αριστερά και τοποθετεί μηδενικά στα `n` χαμηλότερης τάξης bits της μεταβλητής
- Αν δεν συμβεί υπερχείλιση, όταν ένας θετικός ακέραιος ολισθαίνει n θέσεις αριστερά η τιμή του πολ/ζεται με 2^n
- Π.χ. ποια θα είναι η τιμή της μεταβλητής `a` κατά την εκτέλεση του παρακάτω κώδικα;

```
unsigned int a, b = 35;
a = b << 2;
```

`a = 140, διότι: 00100011 << 2 = 00100011`**`00`**

Συγκεκριμένα, τα bits του αρχικού αριθμού (35) ολίσθησαν δύο θέσεις αριστερά και τοποθετήθηκαν τα bits 0 και 0 στην πρώτη και στη δεύτερη θέση, αντίστοιχα

Παρατηρήσεις

- Όταν χρησιμοποιείται ο τελεστής << και το αποτέλεσμα αποθηκεύεται σε μία μεταβλητή, ο τύπος δεδομένων της μεταβλητής πρέπει να είναι τέτοιος ώστε να μπορεί να αποθηκευτεί η τελική τιμή

- Π.χ. ποια θα είναι η τιμή της μεταβλητής a κατά την εκτέλεση του παρακάτω κώδικα;

```
unsigned char a = 32;
a <<= 3; // Ισοδύναμη με a = a << 3;
```

Παρατηρήσεις

- Απάντηση. Η εντολή `a <<= 3` ολισθαίνει την τιμή του `a` τρεις θέσεις αριστερά και εισάγει τρία μηδενικά bits στα δεξιά. Άρα, το αποτέλεσμα της ολίσθησης είναι 100000000. Όμως, αφού ο τύπος του `a` είναι `unsigned char`, αποθηκεύεται μόνο η τιμή που υπάρχει στα οκτώ τελευταία bits. Άρα, η τιμή του `a` γίνεται 0

- Τελειώνοντας τη συζήτησή μας για τους τελεστές `<<` και `>>` είναι λογικό να σας έχει δημιουργηθεί η απορία, πώς ακριβώς χρησιμοποιούνται με τα `cout` και `cin` αντικείμενα; Η απάντηση είναι ότι εκεί δε λειτουργούν σαν τελεστές ολίσθησης. Όπως θα μάθουμε αργότερα, η κλάση `ostream` τους υπερφορτώνει και αλλάζει η συμπεριφορά τους. Ανάλογα με την έκφραση, ο μεταγλωττιστής αντιλαμβάνεται τον τρόπο με τον οποίου πρέπει να τους

Εναλλακτικές Αναπαραστάσεις Τελεστών

- Η γλώσσα παρέχει ένα σύνολο δεσμευμένων λέξεων για την εναλλακτική αναπαράσταση των παρακάτω τελεστών:

```
&&    &=      &       |     ~     !    !=      ||    |=     ^     ^=
and   and_eq  bitand  bitor compl not  not_eq  or    or_eq  xor   xor_eq
```

Για παράδειγμα, οι παραστάσεις:

```
bool a = (b && c) || !d;
int a = (b & c) | ~d;
```

είναι ισοδύναμες με:

```
bool a = (b and c) or not d;
int a = (b bitand c) bitor compl d;
```

Προτεραιότητα Τελεστών

- Κάθε τελεστής χαρακτηρίζεται από μία **προτεραιότητα**
- Σε μία έκφραση που περιέχονται περισσότεροι του ενός τελεστές, οι πράξεις εκτελούνται σύμφωνα **με τη σειρά προτεραιότητας** του κάθε τελεστή
- Π.χ. το αποτέλεσμα της πράξης:

  `7 + 5 * 3 - 1` είναι 21,

  γιατί ο τελεστής * έχει μεγαλύτερη προτεραιότητα από τους τελεστές + και -, οπότε πρώτα εκτελείται η πράξη `5*3 = 15` και όχι οι πράξεις `7+5` ή `3-1`
- Αν μία έκφραση περιέχει διαδοχικούς τελεστές με την **ίδια** προτεραιότητα, τότε οι πράξεις εκτελούνται σύμφωνα **με τη συσχέτισή τους** (associativity), δηλαδή από αριστερά προς τα δεξιά ή αντίστροφα
- Π.χ. το αποτέλεσμα της πράξης:

  `7 * 4 / 2 * 5` είναι 70,

  γιατί, αφού οι τελεστές * και / έχουν την ίδια προτεραιότητα και συσχέτιση από αριστερά προς τα δεξιά, τότε:

  πρώτα εκτελείται ο πολλαπλασιασμός `7*4 = 28`, μετά η διαίρεση `28/2 = 14` και μετά ο πολλαπλασιασμός `14*5 = 70`

Παρατηρήσεις

- Δεν χρειάζεται να απομνημονεύσετε την προτεραιότητα και τη συσχέτιση των τελεστών. Όταν δεν είστε σίγουροι για την προτεραιότητα των τελεστών να χρησιμοποιείτε παρενθέσεις
- Επίσης, συστήνεται η χρήση παρενθέσεων ( ) ακόμα και όταν δεν χρειάζονται, έτσι ώστε ο κώδικας να διαβάζεται πιο εύκολα και να γίνεται σαφέστερη στον αναγνώστη η σειρά αποτίμησης των εκφράσεων. Π.χ. είναι πιο σαφές να γράψουμε:

  `a = b-(c/d)+d;` αντί για `a = b-c/d+d;`

C++: Από τη Θεωρία στην Εφαρμογή

Κεφάλαιο 5ο

Έλεγχος Ροής Προγράμματος

Η Εντολή if

- Η εντολή **if** ελέγχει τη ροή ενός προγράμματος ανάλογα με την τιμή μίας συνθήκης
- Στην πιο απλή μορφή της συντάσσεται ως εξής:

```
if(συνθήκη)
{
        ... /* ομάδα εντολών */
}
```

- Αν η συνθήκη είναι **αληθής (true)**, τότε εκτελούνται οι εντολές που περικλείονται στα άγκιστρα **{...}**
- Αν η συνθήκη είναι **ψευδής (false)**, τότε το μπλοκ των εντολών που περικλείεται στα άγκιστρα παρακάμπτεται και συνεπώς δεν εκτελείται
- Αν το μπλοκ εντολών περιέχει μόνο μία εντολή, τότε τα άγκιστρα μπορούν να παραλειφθούν
- Η έκφραση: **if(x)** είναι ισοδύναμη με **if(x != 0)**
- Η έκφραση: **if(!x)** είναι ισοδύναμη με **if(x == 0)**

Παρατηρήσεις (1)

- **Προσοχή!** Ένα πολύ συνηθισμένο λάθος είναι να προστίθεται το ; στο τέλος της `if` εντολής, όπως κάνουμε με τις απλές εντολές
- Το ; θεωρείται ξεχωριστή εντολή, η οποία απλά δεν κάνει τίποτα *(null statement)*, με αποτέλεσμα ο μεταγλωττιστής να τερματίζει την εκτέλεση της `if`
- Π.χ. τί εμφανίζει ο παρακάτω κώδικας;

```
int x = 10;
if(x < 0); // Προσοχή.
      cout << "Yes\n";
```

Το ; τερματίζει την εκτέλεση της `if`, άρα η `cout` εντολή δε συνδέεται μαζί της. Επομένως, αυτός ο κώδικας εμφανίζει `Yes` ανεξάρτητα από την τιμή του `x`

Παρατηρήσεις (2)

- **Προσοχή!** Ένα ακόμα συνηθισμένο λάθος είναι να συγχέεται ο τελεστής εκχώρησης = με τον τελεστή ελέγχου ισότητας ==
- Π.χ. τί εμφανίζει ο παρακάτω κώδικας;

```
int x = 10;
if(x = 30)
        cout << "Yes\n";
```

ο κώδικας δεν ελέγχει αν το **x** είναι 30, αλλά εκχωρεί την τιμή 30 στο **x**. Άρα, η συνθήκη είναι αληθής και ο κώδικας εμφανίζει **Yes**. Παρόμοια, ο παρακάτω κώδικας δεν εμφανίζει τίποτα:

```
int x = 0;
if(x = 0)
        cout << "Yes\n";
```

γιατί με την εκχώρηση του 0 στο **x**, η συνθήκη γίνεται ψευδής

Η Εντολή if-else

- Η εντολή **if** μπορεί προαιρετικά να συμπληρώνεται με την εντολή **else**:

```
if(συνθήκη)
{
      ... /* ομάδα εντολών A */
}
else
{
      ... /* ομάδα εντολών B */
}
```

Όταν η **συνθήκη** είναι **αληθής** (**true**), τότε εκτελείται η ομάδα εντολών A (δηλ. οι εντολές που περιέχονται ανάμεσα στα άγκιστρα του **if**), ενώ όταν η συνθήκη είναι **ψευδής** (**false**), τότε εκτελείται η ομάδα εντολών B (δηλ. οι εντολές που περιέχονται ανάμεσα στα άγκιστρα του **else**)

Παρατηρήσεις

- **Προσοχή!** Όπως και στην περίπτωση του `if`, ένα συνηθισμένο λάθος είναι να προστεθεί το ; στο τέλος του `else`
- Π.χ. τί εμφανίζει ο παρακάτω κώδικας;

```
int i = 20, j = 10, max;
if(i > j)
        max = i;
else; // Προσοχή.
        max = j;
cout << max;
```

Αρχικά το `max` γίνεται 20, μετά όμως, επειδή το ; τερματίζει την εκτέλεση του `else`, γίνεται 10. Έτσι, ο κώδικας εμφανίζει 10

Ένθετες if Εντολές

- Μία if εντολή μπορεί να περιέχει ένθετες if και else εντολές, οι οποίες με τη σειρά τους μπορεί να περιέχουν και άλλες, κ.ο.κ. Για παράδειγμα, ποια είναι η έξοδος του παρακάτω προγράμματος;

```
#include <iostream>
int main()
{
        int a = -1, b = 0, c = 1;

        if(a < b && b < c)
        {
                if(!b)
                {
                        if(-a == c)
                                std::cout << "One\n";
                        else
                                std::cout << "Two\n";
                }
        }
        else
                std::cout << "Three\n";
        return 0;
}
```

Ένθετες if Εντολές

- Απάντηση. Αφού η συνθήκη στην πρώτη `if` είναι αληθής εκτελείται η ένθετη `if`. Αφού το `b` είναι `0`, η συνθήκη είναι αληθής και εκτελείται η επόμενη ένθετη `if`. Αφού η συνθήκη είναι αληθής το πρόγραμμα εμφανίζει `One`

Ένθετες if Εντολές

- Επειδή η **else** είναι προαιρετική, η παράλειψή της σε μία αλληλουχία από ένθετες **if** εντολές μπορεί να προκαλέσει σύγχυση. Αυτή η περίπτωση συχνά αναφέρεται ως αιωρούμενη (dangling) **else**
- Για να ταιριάζετε **if** και **else** εντολές, ο κανόνας είναι ότι κάθε **else** συνδέεται με την πιο κοντινή **if** που υπάρχει στην ίδια ομάδα εντολών και η οποία δεν σχετίζεται με άλλη **else**. Για παράδειγμα, ποια είναι η έξοδος του παρακάτω προγράμματος;

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

        if(a != 10)
                if(a < 30)
                        std::cout << "1\n";
        else
                std::cout << "2\n";
        return 0;
}
```

Ένθετες if Εντολές

- Απάντηση. Σύμφωνα με τον παραπάνω κανόνα, η **else** συνδέεται με την πιο κοντινή **if** που ανήκει στην ίδια ομάδα εντολών. Αυτή είναι η δεύτερη **if**. Άρα, αφού η συνθήκη στην πρώτη **if** είναι ψευδής το πρόγραμμα δεν εμφανίζει τίποτα

- Σημειώστε ότι επίτηδες χρησιμοποίησα αυτή τη στοίχιση για να σας παραπλανήσω και να νομίσετε ότι η **else** συνδέεται με την πρώτη **if**. Στοιχίζοντας κάθε **else** με την αντίστοιχη **if**, γράφοντας τον κώδικα σε εσοχές και χρησιμοποιώντας άγκιστρα, ακόμα και όταν αυτά δεν χρειάζονται, ο κώδικας διαβάζεται και ελέγχεται πιο εύκολα. Για παράδειγμα, δείτε πόσο πιο εύκολα διαβάζεται και ελέγχεται το προηγούμενο πρόγραμμα:

```
#include <iostream>
int main()
{
        int a = 10;
        if(a != 10)
        {
                if(a < 30)
                        std::cout << "1\n";
                else
                        std::cout << "2\n";
        }
        return 0;
}
```

Έλεγχος Σειράς Συνθηκών

- Όταν θέλουμε να ελέγξουμε μία σειρά από συνθήκες, συνήθως χρησιμοποιείται η ακόλουθη σύνταξη:

```
if(συνθήκη_A)
{
      ... /* ομάδα εντολών A */
}
else if(συνθήκη_B)
{
      ... /* ομάδα εντολών B */
}
else if(συνθήκη_C)
{
      ... /* ομάδα εντολών C */
}
.
.
.
else
{
      ... /* ομάδα εντολών N */
}
... /* επόμενες εντολές του προγράμματος. */
```

- Όταν βρεθεί μία συνθήκη που να είναι αληθής, τότε εκτελείται η ομάδα εντολών που σχετίζεται με αυτή και οι υπόλοιπες **if** συνθήκες αγνοούνται
- Η εκτέλεση του κώδικα συνεχίζει με την πρώτη εντολή που υπάρχει μετά την τελευταία **else** εντολή
- Η τελευταία **else** είναι προαιρετική. Αν υπάρχει, το τμήμα εντολών της εκτελείται μόνο αν όλες οι συνθήκες είναι ψευδείς

Παράδειγμα

- Το παρακάτω πρόγραμμα διαβάζει δύο ακεραίους και εμφανίζει το αποτέλεσμα της σύγκρισής τους:

```
#include <iostream>
using std::cout;
using std::cin;
int main()
{
    int i, j;

    cout << "Enter numbers: ";
    cin >> i >> j;
    if(i < j)
        cout << i << " < " << j << '\n';
    else if(i > j)
        cout << i << " > " << j << '\n';
    else
        cout << i << " = " << j << '\n';
    return 0;
}
```

Παράδειγμα

Να συμπληρώσετε τα κενά στο πρόγραμμα (δεν επιτρέπεται να προσθέσετε άλλες εντολές ή μεταβλητές) ώστε να υλοποιούνται τα παρακάτω:

α. Να εμφανίζει τον μεγαλύτερο από τους δύο ακεραίους.

β. Αν το i είναι μεγαλύτερο από 10, τότε το j να γίνεται 2, αλλιώς να γίνεται 1.

γ. Η παρακάτω εντολή να γραφεί χωρίς τη χρήση λογικών τελεστών:

```
if(i < j && i > 10 && j != 20)
        i = 5;.
```

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

        std::cout << "Enter numbers: ";
        std::cin >> i >> j;
        /* Πρώτη σχέση. */
        if(i ____ j)
                ____;
        std::cout << i;
        /* Δεύτερη σχέση. */
        j = ____;
        /* Τρίτη σχέση. Χρησιμοποιήστε όσες γραμμές θέλετε για να
γράψετε τον κώδικά σας. */

        ____

        return 0;
}
```

Παράδειγμα

- Απάντηση. Για την πρώτη σχέση, στο πρώτο κενό η απάντηση είναι < και στο δεύτερο είναι i = j.

Για τη δεύτερη σχέση, η απάντηση είναι (i > 10) + 1.

Για την τρίτη σχέση μπορούμε να γράψουμε μία σειρά από ένθετες if:

```
if(i < j)
    if(i > 10)
        if(j != 20)
            i = 5;
```

Ο Τελεστής ?: (1)

- Ο τελεστής ?: επιτρέπει την εκτέλεση μίας από δύο ενέργειες, ανάλογα με την τιμή μίας έκφρασης και η σύνταξή του είναι:

```
exp1 ? exp2 : exp3;
```

- Σε μία εντολή με τον τελεστή ?: αν η έκφραση **exp1** είναι αληθής, τότε θα εκτελεστεί η έκφραση που ακολουθεί το ερωτηματικό ? (δηλαδή η **exp2**), αλλιώς θα εκτελεστεί η έκφραση που ακολουθεί την άνω-κάτω τελεία : (δηλαδή η **exp3**). Π.χ., ο παρακάτω κώδικας εμφανίζει δύο τιμές σε αύξουσα σειρά:

```
(i < j) ? (cout << i << ' ' << j) : (cout << j << ' ' << i);
```

- Η τιμή μίας έκφρασης με τον τελεστή ?: είναι ίση με την τιμή της έκφρασης που εκτελείται τελευταία. Η τιμή της έκφρασης μπορεί να αποθηκευτεί σε μία μεταβλητή. Π.χ., στον παρακάτω κώδικα αν το **k** ανήκει στο [5, 10], καταχωρείται στο **i**. Αλλιώς, το **i** γίνεται **0**

```
i = (k >= 5 && k <= 10) ? k : 0;
```

Ο Τελεστής ?: (2)

- Συνήθως, ο τελεστής ?: χρησιμοποιείται για να υποκαταστήσει την εντολή **if-else**, όταν αυτή έχει απλή μορφή. Π.χ., η επόμενη **if-else** εντολή:

```
if(a > b)
        max = a;
else
        max = b;
```

μπορεί να αντικατασταθεί με: `max = (a > b)? a : b;`

- Γενικότερα, η έκφραση: `exp1 ? exp2 : exp3;`

Είναι ισοδύναμη με:

```
if(exp1)
    exp2;
else
    exp3;
```

Ο Τελεστής ?: (3)

- Η έκφραση μετά την την άνω-κάτω τελεία : (δηλαδή η exp3) μπορεί να αντικατασταθεί από άλλη έκφραση που χρησιμοποιεί τον τελεστή ?:

- Π.χ.

```
k = exp1 ? exp2 : add1 ? add2 : add3;
```

Η παραπάνω έκφραση είναι ισοδύναμη με:

```
if(exp1)
        k = exp2;
else if(add1)
        k = add2;
else
        k = add3;
```

Η Εντολή switch (1)

- Η εντολή ελέγχου **switch** χρησιμοποιείται εναλλακτικά έναντι της **if-else-if** δομής, όταν θέλουμε να ελέγξουμε την τιμή μίας έκφρασης έναντι ενός σύνολου τιμών και να χειριστούμε την κάθε περίπτωση ξεχωριστά

- Μία συνηθισμένη σύνταξη της εντολής **switch**:

```
switch(έκφραση)
{
      case σταθερά_1:
      /* ομάδα εντολών που θα εκτελεστεί αν η τιμή της
      έκφρασης είναι ίση με τη σταθερά_1. */
      break;

      case σταθερά_2:
      /* ομάδα εντολών που θα εκτελεστεί αν η τιμή της
      έκφρασης είναι ίση με τη σταθερά_2. */
      break;
      ...
      case σταθερά_n:
      /* ομάδα εντολών που θα εκτελεστεί αν η τιμή της
      έκφρασης είναι ίση με τη σταθερά_n. */
      break;

      default:
      /* ομάδα εντολών που θα εκτελεστεί αν η τιμή της
      έκφρασης δεν είναι ίση με καμία από τις προηγούμενες
      σταθερές. */
      break;
}
```

Η Εντολή switch (2)

- Η έκφραση που ελέγχεται πρέπει να είναι ακέραιη μεταβλητή ή έκφραση
- Οι τιμές των σταθερά_1, σταθερά_2, ... , σταθερά_n πρέπει και αυτές να είναι ακέραιες σταθερές με διαφορετικές τιμές μεταξύ των
- Τα «βήματα» κατά την εκτέλεση της εντολής **switch**:

1. Η τιμή της έκφρασης συγκρίνεται διαδοχικά με κάθε μία από τις σταθερά_1, σταθερά_2, ..., σταθερά_n
   - Αν βρεθεί μία ίδια τιμή, τότε εκτελούνται οι εντολές που ακολουθούν το αντίστοιχο **case** και στη συνέχεια γίνεται τερματισμός της εντολής **switch** μέσω της εντολής **break** (λεπτομέρειες για την εντολή **break** σε επόμενη διάλεξη...)
   - Αν δεν βρεθεί ίδια τιμή, τότε εκτελούνται οι εντολές που ακολουθούν το **default** και στη συνέχεια γίνεται τερματισμός της εντολής **switch** μέσω της εντολής **break**
2. Και στις δύο περιπτώσεις, η εκτέλεση του κώδικα συνεχίζει με την πρώτη εντολή που υπάρχει μετά το άγκιστρο κλεισίματος της **switch** εντολής

Παράδειγμα

```
#include <iostream>
using std::cout;
using std::cin;
int main()
{
	int a;

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

	switch(a)
	{
		case 1:
			cout << "One\n";
		break;

		case 2:
			cout << "Two\n";
		break;

		default:
			cout << "Other\n";
		break;
	}
	cout << "End\n";
	return 0;
}
```

Παρατηρήσεις (1)

- Η ύπαρξη της **default** περίπτωσης είναι **προαιρετική**
- Η **default** περίπτωση μπορεί να βρίσκεται οπουδήποτε μέσα σε μία εντολή **switch** (π.χ. να είναι πρώτη ή να βρίσκεται ανάμεσα στα **case**), αν όμως υπάρχει, προτείνεται να βρίσκεται στο τέλος, δηλ. μετά από κάθε **case**, ώστε να ξεχωρίζει
- Σε περίπτωση που δεν υπάρχει η **default** περίπτωση και η τιμή της έκφρασης δεν είναι ίση με κάποια από τις τιμές των σταθερά_1, σταθερά_2, ..., σταθερά_n, τότε γίνεται τερματισμός της εντολής **switch**, χωρίς να γίνει κάποια άλλη ενέργεια
  - Δηλαδή, η ροή του προγράμματος συνεχίζει με την εκτέλεση της πρώτης εντολής μετά το **switch**

Παρατηρήσεις (2)

- Η ύπαρξη της **break** σε κάθε **case** δεν είναι υποχρεωτική. Αν η **break** λείπει από την **case** που ταιριάζει, το πρόγραμμα συνεχίζει με την εκτέλεση των εντολών που υπάρχουν στις επόμενες **case**
- Συνήθως, η αθέλητη απουσία της **break** αποτελεί την πιο συνηθισμένη αιτία για την μη επιθυμητή συμπεριφορά της **switch**. Π.χ., στο προηγούμενο πρόγραμμα, αν ο προγραμματιστής δεν έχει προσθέσει την εντολή **break** στην περίπτωση του αριθμού 1 και ο χρήστης εισάγει το 1, τότε το πρόγραμμα θα εμφανίσει **One** και **Two**
- Συνήθως, η έλλειψη της **break** αποτελεί πιθανό bug. Όταν την παραλείπετε σκόπιμα, να προσθέτετε κάποιο σχετικό σχόλιο, ώστε αν κάποιος διαβάσει τον κώδικά σας ή ακόμα και εσείς μετά από κάποιο χρονικό διάστημα να μην αναρωτιέται αν η παράλειψή της ήταν σκόπιμη ή όχι

Παρατηρήσεις (3)

- Αν θέλουμε να εκτελεστεί η ίδια ομάδα εντολών σε περισσότερες από μία **case** περιπτώσεις, μπορούμε να τις ενώσουμε

- Π.χ. αν τα μπλοκ εντολών για τις περιπτώσεις των σταθερά_1, σταθερά_2 και σταθερά_3 είναι κοινά, τότε τα αντίστοιχα **case** συνενώνονται ως εξής (έχουν, όπως βλέπουμε, κοινή **break**)

```
case σταθερά_1:
case σταθερά_2:
case σταθερά_3:
/* μπλοκ εντολών που θα εκτελεστεί αν η τιμή της έκφρασης
είναι ίση με σταθερά_1 ή σταθερά_2 ή σταθερά_3. */
break;
```

Παρατηρήσεις (4)

- Κάθε **switch** εντολή μπορεί να γραφτεί ισοδύναμα με χρήση πολλαπλών εντολών **if-else-if**
  - Σε περιπτώσεις, όμως, πολλαπλών **if-else-if** εντολών, η χρήση της **switch** μπορεί να παράξει πιο ευανάγνωστο κώδικα

- **Μειονεκτήματα** της **switch** έναντι της **if**:
  1. Η εντολή **switch** διαφέρει από την εντολή **if** στο ότι η **switch** κάνει έλεγχο μόνο για ισότητα (δηλαδή, για τιμές της έκφρασης που να είναι **ίσες** με σταθερές **case**), ενώ με την εντολή **if** μπορούμε να κάνουμε οποιαδήποτε σύγκριση
  2. Οι τιμές της **έκφρασης** της **switch** και των συγκρινόμενων σταθερών πρέπει υποχρεωτικά να είναι ακέραιες

Παράδειγμα

```
#include <iostream>
using std::cout;
using std::cin;
int main()
{
	char sign;
	int i, j;

	cout << "Enter math sign and two integers: ";
	cin >> sign >> i >> j;
	switch(sign)
	{
		case '+':
			cout << "Sum: " << i+j << '\n';
		break;

		case '-':
			cout << "Diff: " << i-j << '\n';
		break;

		case '*':
			cout << "Product: " << i*j << '\n';
		break;

		case '/':
			if(j != 0)
				cout << "Div: " << (float)i/j << '\n';
			else
				cout << "Second number should not be 0\n";
		break;

		default:
			cout << "Unacceptable operation\n";
		break;
	}
	return 0;
}
```

C++: Από τη Θεωρία στην Εφαρμογή

Κεφάλαιο 6ο
Βρόχοι Επανάληψης

Η Εντολή for

- Η εντολή **for** είναι μία από τις τρεις εντολές επανάληψης και χρησιμοποιείται για τη δημιουργία **επαναληπτικών βρόχων** στη C
- Επαναληπτικός βρόχος καλείται το τμήμα του κώδικα μέσα σε ένα πρόγραμμα, το οποίο εκτελείται από την αρχή και επαναλαμβάνεται όσο μία συνθήκη παραμένει **αληθής (true)**

- Γενική σύνταξη της εντολής **for**:

```
for(αρχική_έκφραση; συνθήκη; τελική_έκφραση)
{
/* ομάδα εντολών (ή αλλιώς "σώμα" του βρόχου) που
   εκτελείται όσο η συνθήκη παραμένει αληθής. */
}
```

Τα Βήματα Εκτέλεσης της for

1. Εκτελείται η αρχική_έκφραση
   - Η αρχική_έκφραση εκτελείται μόνο μία φορά, όταν αρχίζει η εκτέλεση της for εντολής
   - Συνήθως, είναι μία εντολή εκχώρησης που αρχικοποιεί κάποια μεταβλητή, η οποία θα χρησιμοποιηθεί από τις άλλες δύο εκφράσεις
2. Γίνεται έλεγχος της τιμής της συνθήκης
   - Η συνθήκη είναι συνήθως μία σχεσιακή έκφραση
   - Αν είναι ψευδής, τότε ο for βρόχος τερματίζεται και η εκτέλεση του προγράμματος συνεχίζει με την πρώτη εντολή που υπάρχει μετά το άγκιστρο κλεισίματος της for εντολής
   - Αν είναι αληθής, τότε εκτελείται η ομάδα των εντολών που ονομάζεται και «σώμα του βρόχου»
3. Εκτελείται η τελική_έκφραση
   - Συνήθως, η τελική_έκφραση αλλάζει την τιμή κάποιας μεταβλητής που χρησιμοποιείται στη συνθήκη
4. Επαναλαμβάνονται συνεχώς τα βήματα (2) και (3), μέχρι η τιμή της συνθήκης να γίνει ψευδής

Παράδειγμα

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

   for(i = 0; i < 5; i++)
   {
       std::cout << i << '\n';
   }
   return 0;
}
```

Έξοδος: 0 1 2 3 4

Παρατηρήσεις (1)

- Γενικά, ο βρόχος τερματίζεται είτε όταν η **συνθήκη** γίνει ψευδής ή αν νωρίτερα εκτελεστεί κάποια εντολή που τερματίζει την εκτέλεσή του, όπως η `break` και η `return`
- Όταν γνωρίζουμε εκ των προτέρων τον αριθμό των επαναλήψεων που επιθυμούμε να εκτελεστούν, τότε χρησιμοποιούμε συνήθως την εντολή `for` και όχι κάποια άλλη επαναληπτική μέθοδο
- Όπως και στην περίπτωση της `if-else` δομής, αν το μπλοκ εντολών περιέχει μόνο μία εντολή, τότε τα άγκιστρα μπορούν να παραλειφθούν

Παρατηρήσεις (2)

- Τα τμήματα της **for** εντολής, αρχική_έκφραση, συνθήκη και τελική_έκφραση μπορεί να αποτελούνται από μία μόνο εντολή, αλλά και από περισσότερες
- Στην περίπτωση που αποτελούνται από περισσότερες από μία εντολές, τότε αυτές χωρίζονται μεταξύ τους με τον τελεστή κόμμα (,)

- Π.χ.:

```
for(i = 0, j = 10; i+j<15; i++, j--)
```

αρχική_έκφραση συνθήκη τελική_έκφραση

Παρατηρήσεις (3)

- Ένα πολύ συνηθισμένο λάθος είναι η χρήση του = για να συγκρίνουμε με την τελική τιμή. Π.χ. αν γράψουμε

```
for(i = 0; i = 5; i++)
        std::cout << i << ' ';
```

η συνθήκη είναι πάντα αληθής και, όπως θα δείτε στη συνέχεια, δημιουργείται ένας ατέρμονος βρόχος, ο οποίος εμφανίζει συνεχώς 5, υποβαθμίζοντας σημαντικά την απόδοση του υπολογιστή

- Λάθος είναι επίσης και να γράψετε `i == 5`. Αφού το `i` είναι 0, η συνθήκη είναι ψευδής και ο βρόχος δεν εκτελείται. Άρα, δεν εμφανίζεται τίποτα στην οθόνη

Παρατηρήσεις (4)

- Όπως και με την **if** εντολή, ένα συνηθισμένο λάθος είναι η αθέλητη πρόσθεση του ερωτηματικού ; στο τέλος της **for** εντολής
- Π.χ. ο παρακάτω κώδικας:

```
for(i = 16; i > 10; i-=3);
        std::cout << i << ' ';
```

κάνει τρεις επαναλήψεις και εμφανίζει μία φορά την τελική τιμή του i που είναι 10

- Συνήθως, **for** βρόχοι με «κενή ομάδα εντολών» χρησιμοποιούνται σαν βρόχοι εισαγωγής χρονικής καθυστέρησης, δηλαδή «για να περάσει η ώρα» μέχρι να γίνει κάποια ενέργεια
- Όταν θέλετε το σώμα του βρόχου να είναι άδειο, συστήνεται το ; να εισάγεται σε ξεχωριστή γραμμή, ώστε να φαίνεται ξεκάθαρα η πρόθεσή σας. Π.χ.

```
for(a = 0; a < 1000; a++)
;
```

Παρατηρήσεις (5)

- Σε μία **for** εντολή μπορεί να λείπουν κάποια από τα 3 τμήματά της ή ακόμη και όλα. Π.χ. στην εντολή:

```
for(; a < 5; a++)
```

Λείπει η αρχική_έκφραση

- Ωστόσο, το ελληνικό ερωτηματικό ; πρέπει να υπάρχει και να λειτουργεί σαν διαχωριστικό μεταξύ των τμημάτων
- Αν η αρχική_έκφραση και η τελική_έκφραση λείπουν, ο **for** βρόχος είναι ισοδύναμος με τον αντίστοιχο **while** βρόχο, όπως θα δούμε παρακάτω. Π.χ. ο παρακάτω **for** βρόχος:

```
for(; a < 5;)
```

Είναι ισοδύναμος με τον **while** βρόχο:

```
while(a < 5)
```

Παρατηρήσεις (6)

- Όταν σε μία **for** εντολή **λείπει η συνθήκη** ή **η συνθήκη είναι πάντα αληθής**, τότε αυτός ο **for** βρόχος ονομάζεται **ατέρμονος βρόχος**, δηλαδή, δεν τερματίζεται ποτέ. Π.χ. ο βρόχος:

```
for(a = 0; 0 < 1; a++)
```

είναι ατέρμονος, γιατί η συνθήκη 0 < 1 είναι πάντα αληθής

- Η συνήθης πρακτική για τη δημιουργία ατέρμονου βρόχου είναι να παραλείπονται και οι τρεις εκφράσεις, δηλαδή:

```
for(;;)
```

Παράδειγμα (1)

- Ποια είναι η έξοδος του παρακάτω προγράμματος;

```
#include <iostream>
int main()
{
 int i = 0, j = 5;
 for(i > j; i+j == 5; j < 2)
 {
        std::cout << "One\n";
        i = 4;
        j = 2;
 }
 std::cout << i << ' ' << j;
 return 0;
}
```

Παράδειγμα (1)

- Απάντηση. Η τιμή της έκφρασης `i > j` είναι `0`, αλλά αυτό δεν επηρεάζει την εκτέλεση του βρόχου. Ας δούμε αναλυτικά τις επαναλήψεις του.

  1η επανάληψη. Αφού το `i` είναι `0`, η συνθήκη `i+j == 5` (`0+5 = 5 == 5`) είναι αληθής και επομένως εμφανίζεται το `One`. Στη συνέχεια, οι τιμές των `i` και `j` γίνονται `4` και `2`, αντίστοιχα

  2η επανάληψη. Η τιμή της έκφρασης `j < 2` είναι `0`, αλλά δεν επηρεάζει την εκτέλεση του βρόχου. Η συνθήκη `i+j == 5` (`4+2 = 6 == 5`) γίνεται ψευδής και τερματίζεται η εκτέλεση του βρόχου

  Το πρόγραμμα εμφανίζει τις τιμές των `i` και `j` που είναι `4` και `2`, αντίστοιχα.

Παράδειγμα (2)

- Ποια είναι η έξοδος του παρακάτω προγράμματος;

```
#include <iostream>
int main()
{
 int i;
 unsigned int j;

 for(i = 12; i > 2; i -= 5)
        std::cout << i << ' ';
        std::cout << "End = " << i << '\n';

 for(j = 2; j >= 0; j--)
        std::cout << "Test\n";

 return 0;
}
```

Παράδειγμα (2)

- Απάντηση. Αφού η `for` εντολή δεν έχει άγκιστρα, ο μεταγλωττιστής θεωρεί ότι το σώμα του βρόχου αποτελείται από μία μόνο εντολή, δηλαδή από την πρώτη `cout`. Η δεύτερη `cout`, παρά την παραπλανητική στοίχισή της, εκτελείται μόνο μία φορά, μετά το τέλος του βρόχου. Άρα, αρχικά εμφανίζεται `12 7 End = 2`

Πάμε στην επόμενη παγίδα. Αφού το `j` έχει δηλωθεί σαν απρόσημος ακέραιος, η τιμή του δεν θα γίνει ποτέ αρνητική. Το `-1` θα μετατραπεί σε μία unsigned τιμή (π.χ. σε 32-bit σύστημα είναι 4294967295). Επομένως, ο δεύτερος βρόχος είναι ατέρμονος και το πρόγραμμα εμφανίζει συνέχεια `Test`

Παράδειγμα (3)

- Να γραφεί ένα πρόγραμμα το οποίο να διαβάζει έναν ακέραιο και, αν αυτός ανήκει στο [10, 20], να εμφανίζει τόσες φορές όσες και η τιμή του αριθμού τη λέξη `One`, αλλιώς να διαβάζει 10 ακεραίους και να εμφανίζει πόσους αρνητικούς αριθμούς εισήγαγε ο χρήστης

Παράδειγμα (3)

```
#include <iostream>
using namespace std;
int main()
{
        int i, num, neg;

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

        if(num >= 10 && num <= 20)
        {
                for(i = 0; i < num; i++)
                        cout << "One\n";
        }
        else
        {
                neg = 0;
                for(i = 0; i < 10; i++)
                {
                        cout << "Enter number: ";
                        cin >> num;
                        if(num < 0)
                                neg++;
                }
                cout << "Negatives = " << neg << '\n';
        }
        return 0;
}
```

Η Εντολή break

- Η εντολή **break** εκτός από τον τερματισμό της **switch** εντολής, μπορεί να χρησιμοποιηθεί για τον άμεσο τερματισμό ενός **for**, **while** ή **do-while** βρόχου

- Στους επαναληπτικούς βρόχους, μετά την εκτέλεση της εντολής **break** το πρόγραμμα **συνεχίζει** με την **εκτέλεση της πρώτης** εντολής **μετά** τον βρόχο

- Όπως θα δούμε στη συνέχεια, η εκτέλεση της εντολής **break** μέσα σε έναν **ένθετο** επαναληπτικό βρόχο προκαλεί τον τερματισμό μόνο του βρόχου στον οποίο η ίδια περιέχεται

- Επίσης, όπως είδαμε στην εντολή **switch**, η εκτέλεση της εντολής **break** μέσα σε μία **switch** προκαλεί επίσης τον άμεσο τερματισμό της λειτουργίας της

Παράδειγμα

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

   for(i = 1; i < 10; i++)
   {
       if(i == 5)
              break;

       std::cout << i << ' ';
   }
   std::cout << "\nOut: " << i << '\n';
   return 0;
}
```

Παράδειγμα

Έξοδος: 1 2 3 4

Out = 5

Η Εντολή continue

- Η εντολή **continue** χρησιμοποιείται μόνο μέσα σε έναν **for**, **while** ή **do-while** επαναληπτικό βρόχο
- Η εκτέλεση της εντολής **continue** τερματίζει την τρέχουσα επανάληψη του βρόχου που την περιέχει και προκαλεί την έναρξη της επόμενης επανάληψης
- Άρα, οι εντολές μετά την εντολή **continue** **δεν εκτελούνται** για την τρέχουσα επανάληψη

Παράδειγμα

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

   for(i = 0; i < 5; i++)
   {
       if(i == 2 || i == 3)
              continue;
       std::cout << i << ' ';
   }
   return 0;
}
```

Παράδειγμα

```
Έξοδος: 0 1 4
```

Ένθετοι Βρόχοι

- Ένας επαναληπτικός βρόχος (π.χ. **for**, **while** ή **do-while**) μπορεί να είναι **ένθετος** στο εσωτερικό κάποιου άλλου
- Οποιοσδήποτε βρόχος μπορεί να είναι ένθετος μέσα σε οποιοδήποτε άλλο είδος βρόχου (π.χ. **while** βρόχος μέσα σε **for** βρόχο)
- Π.χ. στην παρακάτω γενική περίπτωση, βλέπουμε δύο ένθετα **for**, στα οποία για να συμβεί μία επανάληψη του εξωτερικού βρόχου πρέπει πρώτα να τερματίσει η εκτέλεση του εσωτερικού βρόχου

Εξωτερικός for βρόχος

Εσωτερικός for βρόχος

```
for(αρχική_έκφραση_1; συνθήκη_1; τελική_έκφραση_1)
{
   for(αρχική_έκφραση_2; συνθήκη_2; τελική_έκφραση_2)
   {
         /* ομάδα εντολών που θα εκτελείται συνεχώς
         όσο η συνθήκη_2 παραμένει αληθής. */
   }
   /* ομάδα εντολών που θα εκτελείται συνεχώς όσο η
  συνθήκη_1 παραμένει αληθής. */
}
```

Παράδειγμα (1)

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

   for(i = 0; i < 2; i++)
   {
        std::cout << "One ";
        for(j = i+1; j < 2; j++)
             std::cout << "Two ";
   }
   return 0;
}
```

Παράδειγμα (1)

Έξοδος: `One Two One`

Παράδειγμα (2)

- Ποια είναι η έξοδος του παρακάτω προγράμματος;

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

   for(i = 1; i < 3; i++)
   {
        for(j = 0; j < i; j++)
        {
                if(i+j == 1)
                        break;
                std::cout << "Two ";
        }
        std::cout << "One ";
   }
   std::cout << i << ' ' << j << '\n';
   return 0;
}
```

Παράδειγμα (2)

Έξοδος: `One Two Two One 3 2`

Η Εντολή while

- Γενική σύνταξη της εντολής while:

```
while(συνθήκη)
{
/* ομάδα εντολών που θα εκτελείται όσο η
  συνθήκη παραμένει αληθής. */
}
```

Εκτέλεση της Εντολής while

1. Γίνεται έλεγχος της τιμής της συνθήκης
   - Αν η συνθήκη είναι ψευδής (false) τότε ο while βρόχος τερματίζεται και η εκτέλεση του προγράμματος συνεχίζει με την πρώτη εντολή που υπάρχει μετά το άγκιστρο κλεισίματος της while εντολής
   - Αν η συνθήκη είναι αληθής (true) τότε εκτελείται η ομάδα εντολών που υπάρχει ανάμεσα στα άγκιστρα {} και η τιμή της συνθήκης ελέγχεται πάλι
     - Αν η τιμή της συνθήκης γίνει ψευδής (false), τότε ο while βρόχος τερματίζεται
     - Αν όχι, επανεκτελείται η ομάδα των εντολών του βρόχου while

Η παραπάνω διαδικασία επαναλαμβάνεται μέχρι η τιμή της συνθήκης να γίνει ψευδής

Παρατηρήσεις (1)

- Η εντολή **while** χρησιμοποιείται συνήθως όταν **δεν γνωρίζουμε τον ακριβή αριθμό** των επαναλήψεων που θέλουμε να εκτελεστεί η ομάδα των εντολών μας
  - Όταν αντιθέτως **γνωρίζουμε** εκ των προτέρων **τον αριθμό** των επαναλήψεων που επιθυμούμε να εκτελεστούν, τότε συνήθως χρησιμοποιούμε την εντολή **for**

- Όπως και σε προηγούμενες περιπτώσεις (π.χ. εντολές **if-else**, **for**, κτλ), αν η ομάδα εντολών περιέχει μόνο μία εντολή, τότε τα άγκιστρα μπορούν να παραλειφθούν

- Όπως και με την εντολή **for**, το ; στο τέλος της **while** εντολής δηλώνει ότι το σώμα του βρόχου δεν περιέχει καμία εντολή. Π.χ. ο παρακάτω βρόχος γίνεται ατέρμονος

```
int a = 1;
while(a != 10);
 a++; // Δεν θα εκτελεστεί ποτέ
```

Παρατηρήσεις (2)

- Η εντολή `while(x)` είναι ισοδύναμη με την εντολή `while(x != 0)` και η `while(x == 0)` ισοδύναμη με την `while(!x)`

- Αν η συνθήκη είναι **πάντα αληθής**, ο βρόχος θα εκτελείται συνεχώς, εκτός αν περιέχει μία εντολή που να τον τερματίσει (π.χ. `break`)

- Π.χ. ο βρόχος `while(1)` είναι ατέρμονος, γιατί η συνθήκη είναι πάντα αληθής, αφού το 1 είναι διαφορετικό από το 0

Παρατηρήσεις (3)

- Οι εντολές `for` και `while` είναι πολύ στενά συνδεδεμένες
- Ουσιαστικά, οι εντολές είναι ισοδύναμες (εκτός αν ο `for` βρόχος περιέχει την εντολή `continue`):

```
for(exp1; exp2; exp3)
{
  /* εντολές */
}
```

```
exp1;
while(exp2)
{
    /* εντολές */
    exp3;
}
```

Παράδειγμα

- Να γραφεί ένα πρόγραμμα το οποίο να διαβάζει συνεχώς έναν ακέραιο και να εμφανίζει τη λέξη `Hi` τόσες φορές όσες και η τιμή του ακεραίου. Αν ο χρήστης εισάγει αρνητικό αριθμό, η εισαγωγή των ακεραίων να σταματάει και το πρόγραμμα να εμφανίζει τον συνολικό αριθμό των `Hi` που εμφανίστηκαν στην οθόνη. Χρησιμοποιήστε μόνο `while` βρόχους

Παράδειγμα

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

int main()
{
	int i, num, times;

	times = 0;
	while(1)
	{
		cout << "Enter number: ";
		cin >> num;
		if(num < 0)
			break;
		i = 0;
		while(i < num)
		{
			cout << "Hi\n";
			i++;
		}
		times += num;
	}
	cout << "Total number is: " << times << '\n';
	return 0;
}
```

Η Εντολή do-while

- Αντίθετα με τις εντολές **for** και **while** όπου η συνθήκη ελέγχου ελέγχεται πριν από την εκτέλεση των εντολών, η εντολή **do-while** ελέγχει τη συνθήκη ελέγχου μετά την εκτέλεση των εντολών
- Επομένως, ο βρόχος **do-while** εκτελείται τουλάχιστον μία φορά
- Γενική σύνταξη της εντολής **do-while**:

```
do
{
/* ομάδα εντολών που εκτελείται αρχικά μία φορά
   και στη συνέχεια κατ' επανάληψη όσο η συνθήκη
   παραμένει αληθής. */
} while(συνθήκη);
```

Τα Βήματα Εκτέλεσης της do-while

1. Εκτελείται η ομάδα εντολών που υπάρχει ανάμεσα στα άγκιστρα {}

2. Γίνεται έλεγχος της τιμής της συνθήκης
   - Αν η συνθήκη είναι ψευδής (false) τότε ο do-while βρόχος τερματίζεται και η εκτέλεση του προγράμματος συνεχίζει με την πρώτη εντολή που υπάρχει μετά το άγκιστρο κλεισίματος της do-while εντολής
   - Αν η συνθήκη είναι αληθής (true) τότε επανεκτελείται η ομάδα εντολών που υπάρχει ανάμεσα στα άγκιστρα {}
   - Το βήμα αυτό επαναλαμβάνεται μέχρι η τιμή της συνθήκης να γίνει ψευδής

Παρατηρήσεις (1)

- Γενικά, ο βρόχος **do-while** χρησιμοποιείται λιγότερο από τους **for** και **while** βρόχους, αφού μπορεί να αντικατασταθεί από αυτούς
- Μία πολύ συνηθισμένη χρήση του είναι για έλεγχο εγκυρότητας των τιμών που εισάγει ο χρήστης
- Ο βρόχος **do-while** **πρέπει να τελειώνει** με το ελληνικό ερωτηματικό (;)

Παρατηρήσεις (2)

- Σε περίπτωση που ο βρόχος **do-while** περιέχει μόνο μία εντολή, τα άγκιστρα μπορούν, και στην **do-while** εντολή, να παραλειφθούν

- Παρόλα αυτά, προτείνεται **να χρησιμοποιείτε πάντοτε τα άγκιστρα σε εντολές do-while**, διότι η παράλειψή τους πιθανότατα να παραπλανήσει τους αναγνώστες του προγράμματός σας (στο «κάτω» κομμάτι της εντολής), κάνοντάς τους να νομίζουν ότι χρησιμοποιείτε **while** εντολή

Π.χ.:

```
do
  std::cout << i << '\n';
while(++i <= 10);
```

Παράδειγμα

- Να γραφεί ένα πρόγραμμα το οποίο να διαβάζει συνεχώς έναν ακέραιο και να εμφανίζει τη λέξη `Hi` τόσες φορές όσες και η τιμή του ακεραίου. Το πρόγραμμα να υποχρεώνει τον χρήστη να εισάγει έναν θετικό ακέραιο. Αν ο χρήστης εισάγει διαδοχικά την ίδια τιμή το πρόγραμμα να τερματίζει. Χρησιμοποιήστε μόνο `do-while` βρόχους

Παράδειγμα (2)

```
#include <iostream>
using std::cout;
using std::cin;
int main()
{
    int i, num, last;

    do
    {
        cout << "Enter number: ";
        cin >> num;
    } while(num <= 0);

    do
    {
        last = num;
        i = 1;
        do
        {
            cout << "Hi\n";
            i++;
        } while(i <= num);

        do
        {
            cout << "Enter number: ";
            cin >> num;
        } while(num <= 0);
    } while(last != num);
    return 0;
}
```

Η Εντολή goto

- Με την εντολή goto μπορούμε να μεταφέρουμε τον έλεγχο του προγράμματος σε κάποια άλλη εντολή μέσα στην ίδια συνάρτηση, με την προϋπόθεση ότι η εντολή έχει μία ετικέτα

- Γενική σύνταξη της εντολής goto:

```
goto label;
```

- Όταν εκτελείται η εντολή goto η εκτέλεση του προγράμματος μεταβαίνει άμεσα στην εντολή που ακολουθεί τη θέση που έχει δηλωθεί με το όνομα label

- Η θέση με το όνομα label ονοματίζεται όπως και μία μεταβλητή και πρέπει να είναι μοναδική μέσα στη συνάρτηση όπου χρησιμοποιείται η εντολή goto

- Το όνομά της πρέπει να ακολουθείται από την άνω κάτω τελεία :

Παράδειγμα

- Αν ο χρήστης εισάγει την τιμή -1 η εκτέλεση του προγράμματος μεταβαίνει στη θέση START και ο for βρόχος εκτελείται πάλι από την αρχή

```
#include <iostream> // Παράδειγμα 6.12
int main()
{
  int i, num;
START:
  for(i = 0; i < 5; i++)
  {
   std::cout << "Enter number: ";
   std::cin >> num;
   if(num == -1)
          goto START; /* Θα μπορούσαμε να γράψουμε i=-1 και
  να έχουμε το ίδιο αποτέλεσμα, απλά είναι παράδειγμα για
  τη χρήση της goto. */
  }
  return 0;
}
```

Παρατηρήσεις (1)

- Γενικά, δε συστήνεται η χρήση της goto, γιατί η μετάβαση της εκτέλεσης του προγράμματος από ένα σημείο σε κάποιο άλλο και μετά σε κάποιο άλλο δημιουργεί δυσνόητο κώδικα που δεν είναι καλά οργανωμένος και άρα δύσκολα διαβάζεται και ελέγχεται
- Πολλοί μάλιστα είναι εντελώς αντίθετοι στη χρήση της goto, υποστηρίζοντας ότι δεν έχει καμία θέση μέσα σε ένα καλά δομημένο πρόγραμμα
- Ωστόσο, υπάρχουν περιπτώσεις που η goto μπορεί να φανεί χρήσιμη, όπως για την αποφυγή επανάληψης του ίδιου κώδικα με την ομαδοποίηση κοινών εντολών. Π.χ.

```
if(error_1)
        goto ERROR;
...
if(error_2)
        goto ERROR;
...
ERROR:
... // Εντολές διαχείρισης ανεπιθύμητων καταστάσεων.
```

Παρατηρήσεις (2)

- Σαν ένα ακόμα παράδειγμα, επειδή η **break** τερματίζει μόνο το **switch** ή τον βρόχο στον οποίο περιέχεται, η **goto** μπορεί να φανεί χρήσιμη για την άμεση έξοδο από ένα ένθετο **switch**, ή από ένα **switch** μέσα σε ένα βρόχο, ή από έναν ένθετο βρόχο. Π.χ.

```
for(i = 0; i < 10; i++)
        for(j = 0; j < 20; j++)
                for(k = 0; k < 30; k++)
                {
                        if(συνθήκη)
                                goto NEXT;
                }
NEXT:
...
```

- Αν και γενικά πρέπει να αποφεύγετε την χρήση της, να θυμάστε ότι κάποιες φορές μπορεί να φανεί χρήσιμη και να οδηγήσει σε απλούστερο και πιο ευανάγνωστο κώδικα. Άλλωστε, και οι εντολές **break**, **continue** και **return** είναι παραλλαγές της **goto**

C++: Από τη Θεωρία στην Εφαρμογή

Κεφάλαιο 7ο
Πίνακες

Πίνακες

- Ένας πίνακας είναι **μία δομή δεδομένων** η οποία περιέχει έναν συγκεκριμένο αριθμό στοιχείων **του ίδιου τύπου** (π.χ. πίνακας ακεραίων αριθμών, πίνακας πραγματικών αριθμών, πίνακας χαρακτήρων, ...)

- Κάθε στοιχείο μπορεί να προσπελαστεί μέσω της θέσης του στον πίνακα

- Όλοι οι πίνακες **δεσμεύουν συνεχόμενες θέσεις στη μνήμη** (στην περιοχή μνήμης που ονομάζεται **στοίβα** ή **stack**) του υπολογιστή

- Συνηθέστερα είδη είναι οι μονοδιάστατοι και οι διδιάστατοι πίνακες

Δήλωση Μονοδιάστατου Πίνακα

- Για να δηλώσουμε έναν μονοδιάστατο πίνακα πρέπει να καθορίσουμε το όνομα του πίνακα, τον τύπο δεδομένων των στοιχείων του πίνακα και το πλήθος των στοιχείων του πίνακα

- Η γενική περίπτωση δήλωσης ενός μονοδιάστατου πίνακα είναι:

```
τύπος_δεδομένων  όνομα_πίνακα[πλήθος_στοιχείων_πίνακα];
```

Παρατηρήσεις

- Το όνομα_πίνακα πρέπει να είναι μοναδικό (να μην υπάρχει άλλη μεταβλητή στο πρόγραμμα με το ίδιο όνομα)
- Το πλήθος_στοιχείων_πίνακα αλλιώς και μήκος, μέγεθος ή διάσταση του πίνακα, δηλώνεται από μία θετική σταθερή ακέραια έκφραση μέσα σε αγκύλες [ ]
- Όλα τα στοιχεία του πίνακα έχουν τον ίδιο τύπο δεδομένων, ο οποίος μπορεί να είναι οποιοσδήποτε τύπος (π.χ. `int`, `float`, `char`, …)

Δέσμευση Μνήμης (1)

- Όταν δηλώνεται ένας πίνακας, ο μεταγλωττιστής δεσμεύει ένα τμήμα μνήμης για να αποθηκεύσει τις τιμές των στοιχείων του
  - Αυτές οι τιμές αποθηκεύονται η μία μετά την άλλη, σε διαδοχικές θέσεις μνήμης
  - Τυπικά, αυτό το κομμάτι μνήμης δεσμεύεται από συγκεκριμένο μέρος της μνήμης που ονομάζεται στοίβα (stack), και αποδεσμεύεται όταν τερματιστεί η λειτουργία της συνάρτησης μέσα στην οποία έχει δηλωθεί ο πίνακας

- Π.χ., με την παρακάτω δήλωση ο μεταγλωττιστής δεσμεύει 40 bytes για να αποθηκεύσει τις τιμές των 10 ακεραίων στοιχείων του

```
int arr[10];
```

- Για την εύρεση του μεγέθους της δεσμευμένης μνήμης από έναν πίνακα, μπορούμε να χρησιμοποιήσουμε τον τελεστή `sizeof` (π.χ., `sizeof(arr)`)

Δέσμευση Μνήμης (2)

- Το μέγιστο μέγεθος μνήμης που μπορεί να δεσμευτεί με τη δήλωση ενός πίνακα εξαρτάται από τη διαθέσιμη μνήμη στη στοίβα
- Π.χ. το επόμενο πρόγραμμα μπορεί να μην εκτελεστεί σε κάποιον υπολογιστή, αν δεν υπάρχει η διαθέσιμη μνήμη στη στοίβα για την αποθήκευση των τιμών

```
#include <iostream>
int main()
{
        double arr[500000];
        return 0;
}
```

Θα δούμε στη συνέχεια ότι μπορούμε να δεσμεύσουμε μνήμη και από μία μεγαλύτερη περιοχή που ονομάζεται σωρός (heap)

- Όταν η μνήμη είναι πολύτιμη, μη δηλώνετε πίνακες με μέγεθος μεγαλύτερο από ότι χρειάζεται, ώστε να αποφεύγεται η άσκοπη σπατάλη της

Στοιχεία Μονοδιάστατου Πίνακα

- Για να αναφερθούμε σε κάποιο στοιχείο του πίνακα γράφουμε το **όνομα του πίνακα** συνοδευόμενο από τον **δείκτη θέσης** του στοιχείου μέσα σε αγκύλες []
- **Ο δείκτης θέσης** πρέπει να είναι μία ακέραια σταθερά, μεταβλητή ή έκφραση, η οποία **προσδιορίζει τη θέση** του συγκεκριμένου στοιχείου στον πίνακα
- Το πρώτο στοιχείο ενός πίνακα με μέγεθος n στοιχεία αποθηκεύεται στη θέση [0] του πίνακα, το δεύτερο στοιχείο στη θέση [1], το τρίτο στη θέση [2], ... κ.ο.κ., με αποτέλεσμα το τελευταίο στοιχείο να αποθηκεύεται στη θέση [n-1]
- Κάθε στοιχείο μπορεί να χρησιμοποιηθεί όπως μία απλή μεταβλητή. Π.χ:

```
int i, j, a[10], b[10];
a[0] = 2; // Η τιμή του πρώτου στοιχείου γίνεται 2.
b[9] = a[0]; // Η τιμή του τελευταίου στοιχείου γίνεται 2.
i = j = b[9]+1; // Οι τιμές των i και j γίνονται 3.
a[i+j] = 100; /* Αφού i+j=3+3=6, η τιμή του έβδομου στοιχείου γίνεται
100. */
b[2*i-1] = a[i%j]; /* Η τιμή του b[5] γίνεται ίση με του a[0], δηλαδή
2. */
```

Παρατηρήσεις (1)

- Το πλήθος των στοιχείων του πίνακα δεν μπορεί να αλλάξει κατά την εκτέλεση του προγράμματος. Δηλαδή, δεν μπορείτε να προσθέσετε νέα στοιχεία στον πίνακα ή να διαγράψετε στοιχεία από αυτόν
- Αν το μήκος του πίνακα χρησιμοποιείται πολλές φορές μέσα στο πρόγραμμα, μία καλή πρακτική είναι να το αντικαταστήσετε με μία σταθερά. Αν στο μέλλον χρειαστεί να αλλάξετε το μήκος του, απλά αλλάζετε την τιμή της σταθεράς στο σημείο της δήλωσής της

Παρατηρήσεις (2)

- Να αποφεύγετε παρενέργειες στη δεικτοδότηση των στοιχείων, π.χ. μη γράψετε κάτι σαν αυτό:

```
b[i] = a[i++];
```

  - Εξαρτάται από τον μεταγλωττιστή πότε θα γίνει η αύξηση του i, δηλαδή πριν ή μετά την ανάθεση

- Μην ξεχνάτε ότι η αρίθμηση των στοιχείων ενός πίνακα n στοιχείων ξεκινά απ' το 0 (όχι απ' το 1) και φτάνει ως και το n-1. Κάποτε είχα διαβάσει αυτό:

"My girlfriend told me that I care about programming more than about her. I told her that in an array of my interests she is [1] - she was satisfied"

- Για την αντιγραφή ενός πίνακα (π.χ. `int a[10]`) σε έναν άλλον (π.χ. `int b[10]`), δεν επιτρέπεται να γράψετε κάτι σαν το παρακάτω:

```
b = a;
```

παρόλο που φαίνεται αρκετά «κομψό». Θα καταλάβετε γιατί, όταν εξηγήσουμε τη στενή σχέση μεταξύ πινάκων και δεικτών στην επόμενη διάλεξη

Παρατηρήσεις (3)

- Ένα ακόμα λάθος συμβαίνει όταν θέλουμε να ελέγξουμε αν οι δύο πίνακες έχουν τις ίδιες τιμές. Φαίνεται λογικό να γράψουμε `if(b == a)`. Όχι, αυτή η σύγκριση δεν λειτουργεί όπως θα περιμένατε, δεν συγκρίνονται τα στοιχεία των πινάκων. Θα δούμε στην επόμενη διάλεξη τι πραγματικά συγκρίνεται. Και το χειρότερο είναι ότι αυτή η παραπλανητική συνθήκη μεταγλωττίζεται και μας δίνει την εσφαλμένη εντύπωση ότι πράγματι οι πίνακες συγκρίνονται. Προσοχή, είναι ένα πολύ συνηθισμένο λάθος

Παρατηρήσεις (4)

- **Προσοχή!!!** Η C++ δεν ελέγχει αν κάνετε υπέρβαση των ορίων των θέσεων του πίνακα, αλλά αφήνει τον προγραμματιστή υπεύθυνο γι' αυτό... Σε περίπτωση, όμως, που γίνει υπέρβαση των ορίων του πίνακα, η συμπεριφορά του προγράμματος είναι απρόβλεπτη
- Π.χ. δείτε το διπλανό πρόγραμμα

```
#include <iostream>
int main()
{
        int i, j = 20, arr[3];

        for(i = 0; i < 4; i++)
                arr[i] = 100;

        std::cout << j;
        return 0;
}
```

- Ο `arr` περιέχει τρία στοιχεία και οι επιτρεπτές τιμές των δεικτών θέσης είναι από 0 έως 2
- Στην τελευταία επανάληψη (`i=3`) η έκφραση `arr[3] = 100;` αναθέτει μία τιμή σε ένα στοιχείο που δεν ανήκει στον πίνακα και συγκεκριμένα την τιμή 100 που υπερεγγράφει (overwrites) το περιεχόμενο της μνήμης ακριβώς μετά το στοιχείο `arr[2]`
- Αν η συγκεκριμένη μνήμη έχει δεσμευτεί για τη μεταβλητή `j`, το `j` αλλάζει τιμή και το πρόγραμμα θα εμφανίσει 100 αντί για 20...!!!

Δήλωση Πίνακα και Απόδοση Αρχικών Τιμών (1)

- Τη δήλωση του πίνακα την ακολουθεί ο τελεστής = και οι τιμές των στοιχείων διαχωρίζονται με κόμμα (,) μέσα σε άγκιστρα {}. Π.χ. με τη δήλωση `int arr[4] = {10, 20, 30, 40};` οι τιμές των `arr[0]`, `arr[1]`, `arr[2]`, `arr[3]` γίνονται `10, 20, 30, 40` αντίστοιχα
- Αν η λίστα των τιμών είναι μικρότερη από το πλήθος των στοιχείων του πίνακα, τα υπόλοιπα στοιχεία αρχικοποιούνται με την προεπιλεγμένη τιμή που έχει ο τύπος του στοιχείου. Η προεπιλεγμένη τιμή για ακέραιους είναι το 0 και για πραγματικούς τύπους το 0.0.
- Π.χ. με τη δήλωση `int arr[4] = {10, 20};` οι τιμές των `arr[0]` και `arr[1]`, γίνονται `10` και `20`, ενώ οι τιμές των `arr[2]` και `arr[3]` γίνονται `0`
- Η λίστα των τιμών δεν επιτρέπεται να είναι μεγαλύτερη από το πλήθος των στοιχείων του πίνακα

Δήλωση Πίνακα και Απόδοση Αρχικών Τιμών (2)

- Αν το μήκος του πίνακα παραληφθεί, ο μεταγλωττιστής δημιουργεί έναν πίνακα με μέγεθος ίσο με το πλήθος των τιμών στη λίστα. Π.χ. με τη δήλωση `int arr[4] = {10, 20, 30, 40};` ο μεταγλωττιστής δημιουργεί έναν πίνακα τεσσάρων ακεραίων και θέτει τις τιμές 10, 20, 30 και 40 στα στοιχεία του
- Αν θέλουμε οι τιμές ενός πίνακα, μονοδιάστατου ή πολυδιάστατου, να παραμένουν οι ίδιες κατά την εκτέλεση του προγράμματος, δηλώνουμε τον πίνακα σαν `const`. Ένας `const` πίνακας πρέπει να αρχικοποιηθεί όταν δηλώνεται
- Π.χ. `const int arr[4] = {10, 20, 30, 40};` Ο μεταγλωττιστής θα εμφανίσει μήνυμα λάθους σε οποιαδήποτε προσπάθεια αλλαγής της τιμής κάποιου στοιχείου, όπως π.χ. με την εντολή: `arr[0] = 80;`

Παράδειγμα (1)

- Να γραφεί ένα πρόγραμμα το οποίο να δηλώνει έναν πίνακα 5 πραγματικών και με χρήση επαναληπτικού βρόχου να θέτει τις τιμές 1.1, 1.2, 1.3, 1.4 και 1.5 στα στοιχεία του. Στη συνέχεια, να εμφανίζει τα στοιχεία του πίνακα με αντίστροφη σειρά, δηλαδή από το τελευταίο προς το πρώτο

Παράδειγμα (1)

```
#include <iostream>
int main()
{
   int i;
   double arr[5];

   for(i = 0; i < 5; i++)
       arr[i] = 1.1 + (i*0.1);

   for(i = 4; i >= 0; i--)
       std::cout << arr[i] << '\n';
   return 0;
}
```

Παράδειγμα (2)

- Ποια είναι η έξοδος του παρακάτω προγράμματος;

```
#include <iostream>
int main()
{
        int i, a[] = {20, 10, 0, -10, -20};
        for(i = 0; a[i]; i++)
                if(a[i] < 0)
                        std::cout << a[i] << ' ';
        return 0;
}
```

- Ποια είναι τα περιεχόμενα του πίνακα a στον παρακάτω κώδικα;

```
int i, a[] = {4, 2, 0}, b[] = {2, 3, 4};
for(i = 0; i < 3; i++)
        a[b[i]-a[2-i]]++;
```

Παράδειγμα (2)

1. Η συνθήκη `a[i]` στην `for` εντολή ισοδυναμεί με `a[i] != 0`. Αφού η τιμή του τρίτου στοιχείου είναι 0, ο βρόχος θα τερματιστεί, άρα το πρόγραμμα δεν εμφανίζει τίποτα

2. Τα `a[0]`, `a[1]`, και `a[2]` γίνονται 5, 3, και 1, αντίστοιχα

Παράδειγμα (3)

- Να γραφεί ένα πρόγραμμα το οποίο να δηλώνει δύο πίνακες 10 ακεραίων και να διαβάζει τις τιμές των στοιχείων τους. Στη συνέχεια, να ελέγχει αν υπάρχουν κοινά στοιχεία στους δύο πίνακες και, αν ναι, να εμφανίζει την τιμή του κάθε κοινού στοιχείου. Αλλιώς, να εμφανίζει ένα μήνυμα ότι δεν υπάρχουν κοινά στοιχεία

Παράδειγμα (3)

```
#include <iostream>
using std::cout;
using std::cin;
int main()
{
    const int SIZE = 10;
    int i, j, cnt, arr1[SIZE], arr2[SIZE];

    for(i = 0; i < SIZE; i++)
    {
        cout << "Enter number_" << i+1 << " for the 1st array: ";
        cin >> arr1[i];
        cout << "Enter number_" << i+1 << " for the 2nd array: ";
        cin >> arr2[i];
    }
    cnt = 0; /* Μεταβλητή που μετράει τα κοινά στοιχεία. */
    for(i = 0; i < SIZE; i++)
    {
        for(j = 0; j < SIZE; j++) /* Αυτός ο βρόχος ελέγχει αν το στοιχείο του πρώτου πίνακα
    υπάρχει στον δεύτερο. */
        {
            if(arr1[i] == arr2[j])
            {
                cnt++;
                cout << arr1[i] << '\n';
            }
        }
    }
    if(cnt == 0)
        cout << "No common elements were found\n";
    return 0;
}
```

Η for εύρους Εντολή

- Η C++11 εισάγει μία πιο απλή μορφή της **for** εντολής για την προσπέλαση μίας ακολουθίας τιμών, όπως σε έναν πίνακα ή σε έναν αποδέκτη (π.χ. vector). Αυτή η μορφή της **for** εντολής ονομάζεται **for** εύρους (**range-for**)
- Η σύνταξή της είναι: **for**(δήλωση : έκφραση)

όπου η έκφραση μετά την : αντιπροσωπεύει μία ακολουθία τιμών και η δήλωση καθορίζει τη μεταβλητή που θα χρησιμοποιηθεί για να αποθηκευτούν οι τιμές των στοιχείων της ακολουθίας. Π.χ.

```
int arr[] = {10, 20, 30, 40};
for(auto i : arr)
        std::cout << i << '\n';
```

Σε κάθε επανάληψη, από το πρώτο μέχρι και το τελευταίο στοιχείο του πίνακα, το **i** γίνεται ίσο με το αντίστοιχο στοιχείο του πίνακα, δηλαδή, η τιμή του αντιγράφεται στο **i**, και αυτή εμφανίζεται στην οθόνη. Ο βρόχος διασχίζει όλα τα στοιχεία του πίνακα. Δηλαδή, το **i** γίνεται διαδοχικά ίσο με 10, 20, 30, 40

- Θα μπορούσαμε να δηλώσουμε το **i** σαν **int**, απλά με την λέξη **auto**, η επανάληψη γράφεται με γενικό τρόπο που να μην εξαρτάται από τον τύπο των στοιχείων του πίνακα. Είναι πιο ευέλικτο και βολικό να αφήσουμε τον μεταγλωττιστή να συμπεράνει τον τύπο

Η Πρότυπη Κλάση `vector` (1)

- Η πρότυπη κλάση `vector` είναι ένας αποδέκτης που χρησιμοποιείται για την αποθήκευση στοιχείων του ίδιου τύπου σε συνεχόμενες θέσεις μνήμης
- Η κλάση `vector` είναι μία από τις πιο χρήσιμες και ευρέως χρησιμοποιούμενες κλάσεις της Standard Template Library (STL)
- Σε αντίθεση με τους πίνακες, μπορούμε να καθορίσουμε το αρχικό μέγεθος ενός `vector` αντικειμένου δυναμικά, και το σημαντικότερο, το μέγεθός του μπορεί να αυξηθεί δυναμικά ανάλογα με τις απαιτήσεις του προγράμματος
- Δηλαδή, η κλάση `vector` υποστηρίζει δυναμική διαχείριση μνήμης που επιτρέπει την προσθήκη ή τη διαγραφή στοιχείων κατά την εκτέλεση του προγράμματος
- Γενικά, η διαχείριση της κλάσης `vector` είναι σχετικά απλή και παρέχει πολλές δυνατότητες, μεγαλύτερη ασφάλεια και ευελιξία από ότι ο πίνακας

Η Πρότυπη Κλάση vector (2)

- Για να χρησιμοποιήσουμε την κλάση vector πρέπει να συμπεριλάβουμε το αρχείο vector. Για να δημιουργήσουμε ένα vector αντικείμενο πρέπει να καθορίσουμε τον τύπο των στοιχείων που θα περιέχει. Π.χ, ας δούμε μερικά παραδείγματα δήλωσης vector αντικειμένων

```
vector<int> arr; // Δημιουργία άδειου διανύσματος ακεραίων στοιχείων.
vector<int>  arr(3,  20);  /* Δημιουργία διανύσματος τριών ακεραίων
στοιχείων με αρχική τιμή 20. */
vector<int>  arr{1,  2,  3,  4};  /* Δημιουργία διανύσματος τεσσάρων
ακεραίων στοιχείων με τις αντίστοιχες αρχικές τιμές. */
int num;
cin >> num;
vector<double>  arr(num);  /* Δημιουργία διανύσματος num πραγματικών
στοιχείων. */
```

Παράδειγμα

- Το παρακάτω πρόγραμμα διαβάζει 10 αριθμούς και αποθηκεύει σε ένα vector αντικείμενο, που αρχικά είναι άδειο, τους θετικούς αριθμούς. Στη συνέχεια εμφανίζει τις τιμές των στοιχείων που έχουν αποθηκευτεί

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
        int i, num;
        vector<int> vec;
        for(i = 0; i < 10; i++)
        {
                cout << "Enter number: ";
                cin >> num;
                if(num > 0)
                        vec.push_back(num);
        }
        for(i = 0; i < vec.size(); i++)
                cout << vec[i] << '\n';
        return 0;
}
```

Διδιάστατοι Πίνακες

- Οι **διδιάστατοι πίνακες** μοιάζουν με τους γνωστούς μαθηματικούς πίνακες δύο διαστάσεων της άλγεβρας και αποτελούνται και αυτοί από **γραμμές** και **στήλες**

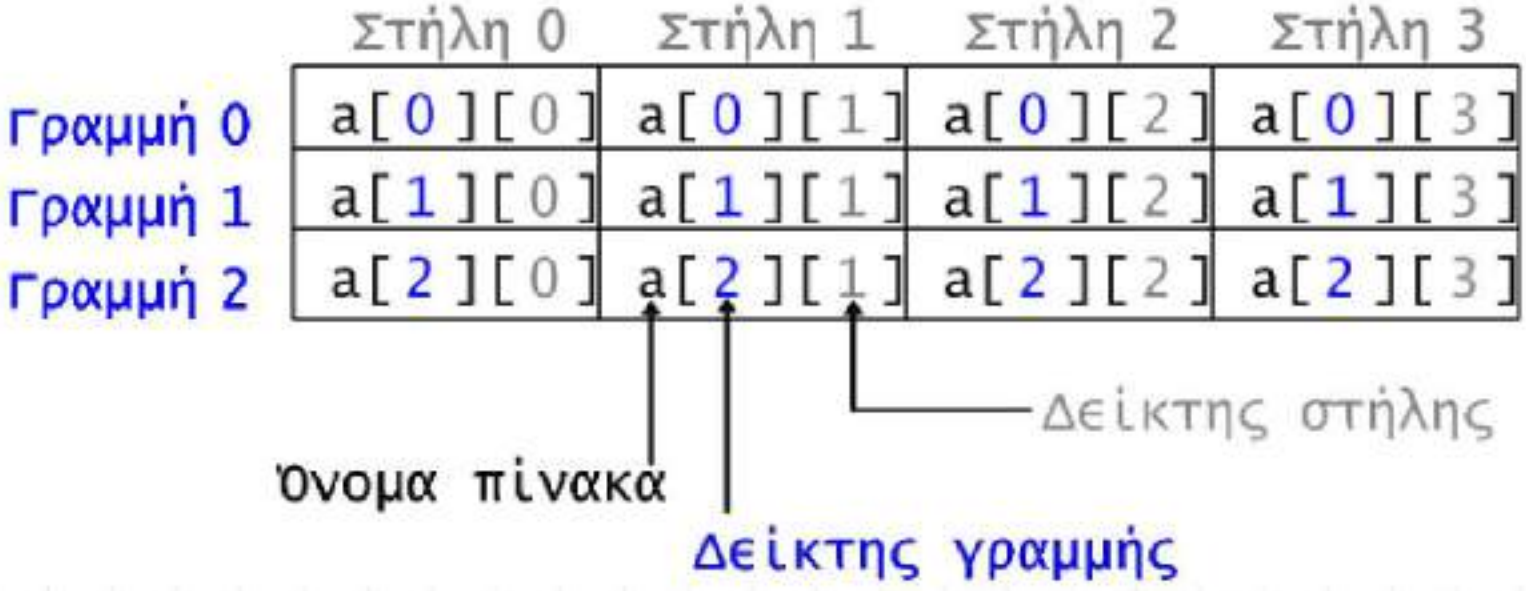
- Για να δηλώσουμε έναν διδιάστατο πίνακα πρέπει να καθορίσουμε το όνομα του πίνακα, τον τύπο δεδομένων των στοιχείων του πίνακα, καθώς και το πλήθος των γραμμών και των στηλών του

```
τύπος_δεδομένων όνομα[πλήθος_γραμμών][πλήθος_στηλών]
```

- Το **πλήθος των στοιχείων** ενός διδιάστατου πίνακα είναι ίσο με το **γινόμενο** του πλήθους **των γραμμών** του επί το πλήθος **των στηλών** του

Στοιχεία Διδιάστατου Πίνακα

- Π.χ. η εντολή `int a[3][4];` δηλώνει έναν διδιάστατο πίνακα με όνομα `a`, ο οποίος περιέχει 12 στοιχεία και καθένα από αυτά τα στοιχεία είναι ένας ακέραιος αριθμός (`int`)
- Για να αναφερθούμε σε κάποιο στοιχείο γράφουμε το όνομα του πίνακα και τους δείκτες γραμμής και στήλης μέσα σε διπλές αγκύλες `[][]`. Όπως και με τους μονοδιάστατους πίνακες, η αρίθμηση των δεικτών θέσης γραμμής και στήλης αρχίζει από το μηδέν. Π.χ.



Διδιάστατοι Πίνακες και Μνήμη (1)

- Όπως και με τους μονοδιάστατους πινάκες, όταν δηλώνεται ένας διδιάστατος πίνακας, ο μεταγλωττιστής δεσμεύει ένα τμήμα μνήμης από τη στοίβα για να αποθηκεύσει τα στοιχεία του
- Τα στοιχεία αποθηκεύονται στη μνήμη διαδοχικά ανά γραμμή, αρχίζοντας πρώτα με τα στοιχεία της πρώτης γραμμής, μετά της δεύτερης, της τρίτης, κ.ο.κ.
- Αν και μιλάμε για πολυδιάστατους πίνακες, στην πραγματικότητα, η C++ υποστηρίζει μόνο μονοδιάστατους πίνακες. Απλά, επειδή το στοιχείο ενός πίνακα μπορεί και αυτό να είναι πίνακας μπορούμε να προσομοιώσουμε πολυδιάστατους πίνακες
- Π.χ., τα στοιχεία του πίνακα `a` είναι τα `a[0]`, `a[1]` και `a[2]`, όπου το καθένα από αυτά είναι πίνακας τεσσάρων ακεραίων

Διδιάστατοι Πίνακες και Μνήμη (2)

- Θεωρήστε έναν διδιάστατο πίνακα, έστω `int a[ROWS][COLS]`
- Αφού τα στοιχεία του πίνακα αποθηκεύονται σε διαδοχικές θέσεις μνήμης, ο μεταγλωττιστής για να βρει τη διεύθυνση μνήμης ενός στοιχείου (και δεδομένου επίσης ότι το όνομα του πίνακα ισούται με τη διεύθυνση του πρώτου στοιχείου του πίνακα, όπως θα δούμε σε επόμενη διάλεξη), ο μεταγλωττιστής για να βρει τη διεύθυνση μνήμης του στοιχείου `a[i][j]`

  α) υπολογίζει το μέγεθος σε bytes της γραμμής, δηλ: `row_size = COLS × sizeof(int)` και το πολλαπλασιάζει με το `i`

  β) πολλαπλασιάζει το `j` με το μέγεθος σε bytes του ενός στοιχείου και

  γ) προσθέτει τα δύο γινόμενα στη διεύθυνση του πρώτου στοιχείου του πίνακα, άρα:

```
memory_address = a + (i × row_size) + (j × sizeof(int))
```

- Παρατηρήστε ότι για τον υπολογισμό της διεύθυνσης μνήμης ενός στοιχείου, μόνο ο αριθμός των στηλών είναι απαραίτητος

Αρχικοποίηση Διδιάστατου Πίνακα (1)

- Όπως και ένας μονοδιάστατος πίνακας, ένας διδιάστατος πίνακας μπορεί να αρχικοποιηθεί μαζί με τη δήλωσή του. Οι τιμές εκχωρούνται στα στοιχεία του πίνακα ανά γραμμή, ξεκινώντας από τα στοιχεία της πρώτης γραμμής, μετά της δεύτερης, κ.ο.κ. Π.χ. με τη δήλωση:

```
int arr[3][3] = {{10, 20, 30}, {40, 50, 60}, {70, 80, 90}};
```

η τιμή του `arr[0][0]` γίνεται 10, η τιμή του `arr[0][1]` γίνεται 20, η τιμή του `arr[0][2]` γίνεται 30 κ.ο.κ. Εναλλακτικά, μπορούμε να παραλείψουμε τα εσωτερικά άγκιστρα και να γράψουμε:

```
int arr[3][3] = {10, 20, 30, 40, 50, 60, 70, 80, 90};
```

- Με τα εσωτερικά άγκιστρα, αν η λίστα των τιμών είναι μικρότερη από το πλήθος των στοιχείων μίας γραμμής, ο μεταγλωττιστής εκχωρεί την τιμή 0 στα υπόλοιπα στοιχεία της. Αν είναι μεγαλύτερη, είναι λάθος. Π.χ. με τη δήλωση:

```
int arr[3][3] = {{10, 20}, {40, 50}, {70}};
```

οι τιμές των `arr[0][2]` , `arr[1][2]` , `arr[2][1]` και `arr[2][2]` αρχικοποιούνται με 0

Αρχικοποίηση Διδιάστατου Πίνακα (2)

- Αν παραλείψουμε την αρχικοποίηση κάποιας γραμμής, ο μεταγλωττιστής εκχωρεί την τιμή 0 στα στοιχεία της. Π.χ. με τη δήλωση:

  ```
  int arr[3][3] = {{10, 20, 30}};
  ```

  τα στοιχεία της δεύτερης και της τρίτης γραμμής γίνονται 0

- Αν παραλείπονται τα εσωτερικά άγκιστρα και η λίστα των τιμών είναι μικρότερη από το πλήθος των στοιχείων του πίνακα, ο μεταγλωττιστής αποδίδει την τιμή 0 στα υπόλοιπα στοιχεία. Π.χ. με τη δήλωση:

  ```
  int arr[3][3] = {10, 20};
  ```

  οι τιμές των `arr[0][0]` και `arr[0][1]` γίνονται 10 και 20 αντίστοιχα, ενώ όλων των υπολοίπων στοιχείων γίνονται 0

- Όταν δηλώνεται ένας διδιάστατος πίνακα, ο αριθμός των στηλών πρέπει υποχρεωτικά να καθοριστεί. Ο αριθμός των γραμμών είναι προαιρετικός. Αν δεν δηλωθεί, ο μεταγλωττιστής θα δημιουργήσει έναν διδιάστατο πίνακα με βάση τη λίστα αρχικοποίησης. Π.χ. με τη δήλωση:

  ```
  int arr[][3] = {10, 20, 30, 40, 50, 60};
  ```

  αφού ο πίνακας έχει τρεις στήλες και οι αρχικές τιμές είναι έξι, ο μεταγλωττιστής θα δημιουργήσει έναν διδιάστατο πίνακα ακεραίων με δύο γραμμές και τρεις στήλες

Αρχικοποίηση Διδιάστατου Πίνακα (3)

- Αν η αρχική τιμή μπορεί εύκολα να παραχθεί ή είναι η ίδια για όλα τα στοιχεία, ένας συνήθης τρόπος αρχικοποίησης του πίνακα είναι με τη χρήση διπλών επαναληπτικών βρόχων. Π.χ. ο επόμενος κώδικας αρχικοποιεί τα στοιχεία του πίνακα με την τιμή 60

```
int row, col, arr[5][10];
for(row = 0; row < 5; row++)
      for(col = 0; col < 10; col++)
            arr[row][col] = 60;
```

Παράδειγμα

- Να γραφεί ένα πρόγραμμα το οποίο να διαβάζει οκτώ ακεραίους, να τους αποθηκεύει σε έναν πίνακα 2*4 και να εμφανίζει τα στοιχεία του πίνακα αντίστροφα, δηλαδή ξεκινώντας από το «κάτω-δεξιά» στοιχείο και καταλήγοντας στο «πάνω-αριστερά»

```
int row, col, arr[5][10];
for(row = 0; row < 5; row++)
      for(col = 0; col < 10; col++)
            arr[row][col] = 60;
```

Παράδειγμα

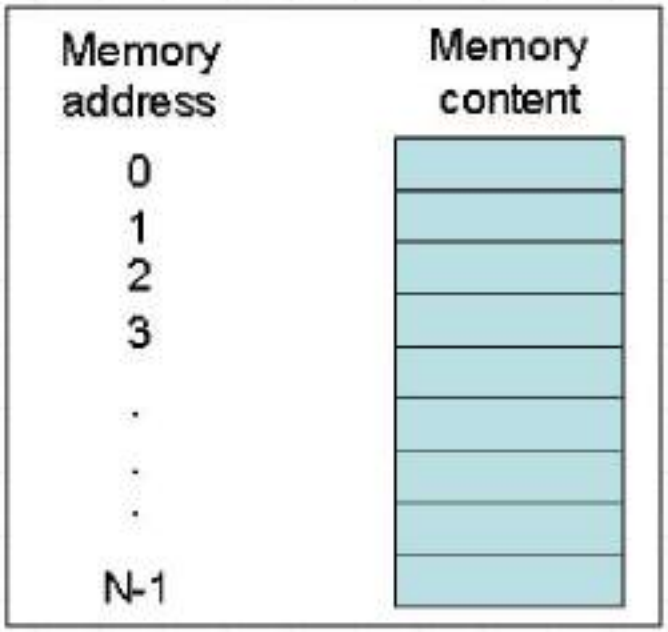
```
#include <iostream>
using namespace std;
int main()
{
    const int ROWS = 2;
    const int COLS = 4;
    int i, j, arr[ROWS][COLS];

    for(i = 0; i < ROWS; i++)
    {
            for(j = 0; j < COLS; j++)
            {
                    cout << "Enter arr[" << i << "][" << j << "]: ";
                    cin >> arr[i][j];
            }
    }
    cout << "\nArray elements\n";
    cout << "--------------\n";
    for(i = ROWS-1; i >= 0; i--)
    {
            for(j = COLS-1; j >= 0; j--)
                    cout << arr[i][j] << ' ';
            cout << '\n';
    }
    return 0;
}
```

C++: Από τη Θεωρία στην Εφαρμογή

Κεφάλαιο 8ο
Δείκτες

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

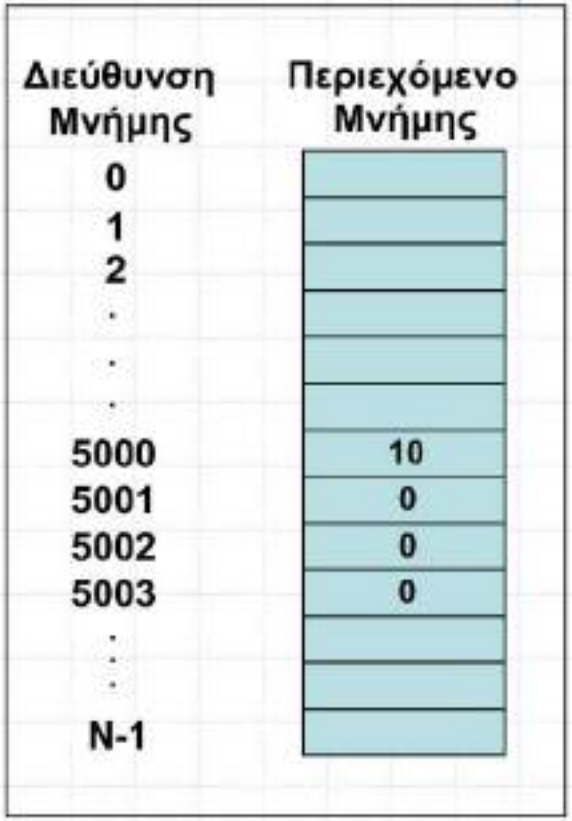
- Όταν δηλώνεται μία μεταβλητή, ο μεταγλωττιστής δεσμεύει τις απαραίτητες **συνεχόμενες** θέσεις (bytes) στη μνήμη, για να αποθηκεύσει την τιμή της

- 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 <u>τον κάνει να δείχνει στη διεύθυνση του n-οστού στοιχείου μετά από αυτό που έδειχνε</u>

- 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; /* Εγκυρο, 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.

C++: Από τη Θεωρία στην Εφαρμογή

Κεφάλαιο 9ο
Χαρακτήρες

Χαρακτήρες

- Έως τώρα έχουμε κατά κύριο λόγο χρησιμοποιήσει τους αριθμητικούς τύπους δεδομένων `int`, `float` και `double`
- Ο τύπος δεδομένων `char` είναι κι αυτός **αριθμητικός**
- Για τη διαχείριση των χαρακτήρων (και των αλφαριθμητικών στο επόμενο κεφάλαιο), θα θεωρήσουμε ότι το σύνολο των χαρακτήρων που υποστηρίζει ο υπολογιστής είναι κωδικοποιημένο σύμφωνα με το πιο διαδεδομένο πρότυπο, τον **κώδικα ASCII**, που **αντιστοιχίζει** κάθε χαρακτήρα σε μία **αριθμητική τιμή**
- Ο **ASCII κώδικας** αντιστοιχίζει (κωδικοποιεί) ένα σύνολο χαρακτήρων που αποτελείται από γράμματα, αριθμούς, σημεία στίξης, κτλ... με ακέραιες τιμές ανάμεσα στο 0 και το 255
- Π.χ. η ASCII τιμή του χαρακτήρα `'C'` είναι το 67, ενώ η ASCII τιμή του χαρακτήρα `'c'` είναι το 99

Πίνακας ASCII - Βασικοί χαρακτήρες (0-127)

Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex
(nul)	0	0x00	(sp)	32	0x20	@	64	0x40	`	96	0x60
(soh)	1	0x01	!	33	0x21	A	65	0x41	a	97	0x61
(stx)	2	0x02	"	34	0x22	B	66	0x42	b	98	0x62
(etx)	3	0x03	#	35	0x23	C	67	0x43	c	99	0x63
(eot)	4	0x04	$	36	0x24	D	68	0x44	d	100	0x64
(enq)	5	0x05	%	37	0x25	E	69	0x45	e	101	0x65
(ack)	6	0x06	&	38	0x26	F	70	0x46	f	102	0x66
(bel)	7	0x07	'	39	0x27	G	71	0x47	g	103	0x67
(bs)	8	0x08	(	40	0x28	H	72	0x48	h	104	0x68
(ht)	9	0x09	)	41	0x29	I	73	0x49	i	105	0x69
(nl)	10	0x0a	*	42	0x2a	J	74	0x4a	j	106	0x6a
(vt)	11	0x0b	+	43	0x2b	K	75	0x4b	k	107	0x6b
(np)	12	0x0c	,	44	0x2c	L	76	0x4c	l	108	0x6c
(cr)	13	0x0d	-	45	0x2d	M	77	0x4d	m	109	0x6d
(so)	14	0x0e	.	46	0x2e	N	78	0x4e	n	110	0x6e
(si)	15	0x0f	/	47	0x2f	O	79	0x4f	o	111	0x6f
(dle)	16	0x10	0	48	0x30	P	80	0x50	p	112	0x70
(dc1)	17	0x11	1	49	0x31	Q	81	0x51	q	113	0x71
(dc2)	18	0x12	2	50	0x32	R	82	0x52	r	114	0x72
(dc3)	19	0x13	3	51	0x33	S	83	0x53	s	115	0x73
(dc4)	20	0x14	4	52	0x34	T	84	0x54	t	116	0x74
(nak)	21	0x15	5	53	0x35	U	85	0x55	u	117	0x75
(syn)	22	0x16	6	54	0x36	V	86	0x56	v	118	0x76
(etb)	23	0x17	7	55	0x37	W	87	0x57	w	119	0x77
(can)	24	0x18	8	56	0x38	X	88	0x58	x	120	0x78
(em)	25	0x19	9	57	0x39	Y	89	0x59	y	121	0x79
(sub)	26	0x1a	:	58	0x3a	Z	90	0x5a	z	122	0x7a
(esc)	27	0x1b	;	59	0x3b	[	91	0x5b	{	123	0x7b
(fs)	28	0x1c	<	60	0x3c	\	92	0x5c	|	124	0x7c
(gs)	29	0x1d	=	61	0x3d	]	93	0x5d	}	125	0x7d
(rs)	30	0x1e	>	62	0x3e	^	94	0x5e	~	126	0x7e
(us)	31	0x1f	?	63	0x3f	_	95	0x5f	(del)	127	0x7f

Πίνακας ASCII - Επιπλέον χαρακτήρες (128-255)

Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex
Ç	128	0x80	á	160	0xa0	└	192	0xc0	α	224	0xe0
ü	129	0x81	í	161	0xa1	┴	193	0xc1	ß	225	0xe1
é	130	0x82	ó	162	0xa2	┬	194	0xc2	Γ	226	0xe2
â	131	0x83	ú	163	0xa3	├	195	0xc3	π	227	0xe3
ä	132	0x84	ñ	164	0xa4	─	196	0xc4	Σ	228	0xe4
à	133	0x85	Ñ	165	0xa5	┼	197	0xc5	σ	229	0xe5
å	134	0x86	ª	166	0xa6	╞	198	0xc6	µ	230	0xe6
ç	135	0x87	º	167	0xa7	╟	199	0xc7	τ	231	0xe7
ê	136	0x88	¿	168	0xa8	╚	200	0xc8	Φ	232	0xe8
ë	137	0x89	⌐	169	0xa9	╔	201	0xc9	Θ	233	0xe9
è	138	0x8a	¬	170	0xaa	╩	202	0xca	Ω	234	0xea
ï	139	0x8b	½	171	0xab	╦	203	0xcb	δ	235	0xeb
î	140	0x8c	¼	172	0xac	╠	204	0xcc	∞	236	0xec
ì	141	0x8d	¡	173	0xad	═	205	0xcd	φ	237	0xed
Ä	142	0x8e	«	174	0xae	╬	206	0xce	ε	238	0xee
Å	143	0x8f	»	175	0xaf	╧	207	0xcf	∩	239	0xef
É	144	0x90	░	176	0xb0	╨	208	0xd0	≡	240	0xf0
æ	145	0x91	▒	177	0xb1	╤	209	0xd1	±	241	0xf1
Æ	146	0x92	▓	178	0xb2	╥	210	0xd2	≥	242	0xf2
ô	147	0x93	│	179	0xb3	╙	211	0xd3	≤	243	0xf3
ö	148	0x94	┤	180	0xb4	╘	212	0xd4	⌠	244	0xf4
ò	149	0x95	╡	181	0xb5	╒	213	0xd5	⌡	245	0xf5
û	150	0x96	╢	182	0xb6	╓	214	0xd6	÷	246	0xf6
ù	151	0x97	╖	183	0xb7	╫	215	0xd7	≈	247	0xf7
ÿ	152	0x98	╕	184	0xb8	╪	216	0xd8	°	248	0xf8
Ö	153	0x99	╣	185	0xb9	┘	217	0xd9	∙	249	0xf9
Ü	154	0x9a	║	186	0xba	┌	218	0xda	·	250	0xfa
¢	155	0x9b	╗	187	0xbb	█	219	0xdb	√	251	0xfb
£	156	0x9c	╝	188	0xbc	▄	220	0xdc	ⁿ	252	0xfc
¥	157	0x9d	╜	189	0xbd	▌	221	0xdd	²	253	0xfd
Pts	158	0x9e	╛	190	0xbe	▐	222	0xde	■	254	0xfe
ƒ	159	0x9f	┐	191	0xbf	▀	223	0xdf		255	0xff

Ο Τύπος char (1)

- Για να αποθηκεύσουμε ένα χαρακτήρα σε μία μεταβλητή χρησιμοποιούμε τον τύπο **char**
- Βέβαια, επειδή ο χαρακτήρας κωδικοποιείται σαν ακέραιος, μπορούμε να χρησιμοποιήσουμε και κάποιον άλλο ακέραιο τύπο (π.χ. **int**)
- Στο επόμενο παράδειγμα δηλώνεται μία μεταβλητή τύπου **char** με το όνομα **ch** και αποθηκεύεται ο χαρακτήρας **'a'** σε αυτήν

```
char ch;
ch = 'a';
```

- Όταν χρησιμοποιείται κάποιος σταθερός χαρακτήρας πρέπει να περικλείεται σε **μονές αποστρόφους** (' ') και **όχι** σε διπλά εισαγωγικά

Ο Τύπος char (2)

- Όταν αποθηκεύεται ένας χαρακτήρας σε μία μεταβλητή τύπου **char**, στην πραγματικότητα αποθηκεύεται η *ASCII* τιμή του χαρακτήρα. Δηλαδή, στο προηγούμενο παράδειγμα στη μεταβλητή **ch** αποθηκεύτηκε η τιμή **97**

- Επομένως, οι εντολές: **ch = 'a';** και **ch = 97;** είναι ισοδύναμες

- Βέβαια, είναι προτιμότερο να αποφεύγετε τη χρήση της αριθμητικής τιμής, όχι μόνο γιατί το πρόγραμμά σας θα διαβάζεται πιο εύκολα, αλλά και γιατί δεν θα εξαρτάται από το σύνολο χαρακτήρων που χρησιμοποιείται

- Παρατηρήστε ότι οι πιο συνηθισμένοι χαρακτήρες, όπως **γράμματα**, **ψηφία** και **σημεία στίξης** αντιστοιχίζονται σε αριθμητικές τιμές ανάμεσα στο **0** και το **127**

- Οι χαρακτήρες με τιμές από **128** έως **255** αποτελούν το εκτεταμένο *ASCII* σύνολο και αντιστοιχίζονται σε εξεζητημένα γράμματα και ειδικά σύμβολα

Χρήση Χαρακτήρα

- Για να εμφανίσουμε την ASCII τιμή με το cout προσαρμόζουμε τον τύπο της μεταβλητής σε int. Π.χ.

```
char ch = 'a';
cout << ch << ' ' << (int)ch;
```

Ο κώδικας εμφανίζει a 97

- Ουσιαστικά, όταν ένας χαρακτήρας περιέχεται σε μία έκφραση, είτε είναι σταθερά είτε μεταβλητή, η C++ τον χειρίζεται σαν ακέραιο και χρησιμοποιεί την ASCII τιμή του

- Αφού η C++ χειρίζεται τους χαρακτήρες σαν ακεραίους, μπορούμε να τους χρησιμοποιήσουμε σε αριθμητικές εκφράσεις. Π.χ.

```
char ch = 'c';
int i;
ch++; /* Η μεταβλητή ch γίνεται 'd'. */
ch = 68; /* Η μεταβλητή ch γίνεται 'D'. */
i = ch-3; /* Η μεταβλητή i γίνεται 'A' δηλ. 65. */
```

Παρατηρήσεις

- Επειδή η μέγιστη τιμή που μπορεί να πάρει μία μεταβλητή `char` είναι το 127 (θυμηθείτε ότι το εύρος τιμών του `char` είναι -128...127) σε περίπτωση που θέλουμε να αποθηκεύσουμε σε μία μεταβλητή `char` ένα χαρακτήρα με ASCII τιμή μεγαλύτερη από 127, πρέπει να χρησιμοποιήσουμε κάποιον άλλον τύπο, όπως π.χ. `unsigned char` ή `int`

- Ένα συνηθισμένο σημείο που «μπερδεύει» στην αρχή είναι η διαχείριση χαρακτήρων που αντιστοιχούν σε ψηφία. Για παράδειγμα, η ASCII τιμή του χαρακτήρα 2 δεν είναι 2, αλλά 50. Αν θέλουμε να χρησιμοποιήσουμε τον χαρακτήρα 2 γράφουμε `'2'`, όχι 2. Παρόμοια, για να ελέγξουμε αν ένας χαρακτήρας αντιστοιχεί σε ψηφίο 0-9 γράφουμε:

```
if(ch >= '0' && ch <= '9') όχι if(ch >= 0 && ch <= 9)
```

Διάβασμα Χαρακτήρων

- Υπάρχουν πολλοί τρόποι για να διαβαστεί ένας χαρακτήρας από το ρεύμα εισόδου
- Ο πιο συνηθισμένος τρόπος είναι με την `get()`, η οποία είναι συνάρτηση μέλος της `istream` κλάσης
- Η `get()` αρχίζει να διαβάζει χαρακτήρες όταν ο χρήστης πατήσει *Enter*. Αν εκτελεστεί επιτυχημένα, επιστρέφει τον χαρακτήρα που διαβάστηκε σαν `int`
- Αν δεν υπάρχουν άλλοι διαθέσιμοι χαρακτήρες για διάβασμα, επιστρέφει μία ειδική σταθερά που δηλώνεται στο `iostream` με το όνομα `EOF` (*End Of File*)
- Για παράδειγμα, αν ο χρήστης πληκτρολογήσει τους χαρακτήρες `abc` και πατήσει *Enter*, η πρώτη κλήση της get() επιστρέφει το `'a'`, η δεύτερη το `'b'`, η τρίτη το `'c'` και η τέταρτη το `'\n'`. Αν κληθεί πάλι, το πρόγραμμα «κολλάει» μέχρι ο χρήστης να εισάγει νέο(υς) χαρακτήρα(ες) και πατήσει πάλι *Enter*

Παράδειγμα (1)

- Το παρακάτω πρόγραμμα εμφανίζει και μετράει τους χαρακτήρες που εισάγει ο χρήστης μέχρι να πατήσει *Enter*

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

int main()
{
  int ch, sum;

  cout << "Enter characters: ";
  sum = 0;
  while((ch = cin.get()) != '\n' && ch != EOF) /* Επειδή ο τελεστής != έχει
μεγαλύτερη προτεραιότητα από τον = χρησιμοποιούμε παρενθέσεις. */
  {
          sum++;
          cout << (char)ch; /* Προσαρμόζουμε τον τύπο για την εμφάνιση του
χαρακτήρα. */
  }
  cout << '\n' << sum << " characters are read\n";
  return 0;
}
```

Η `get()` επιστρέφει έναν-έναν τους χαρακτήρες που εισήγαγε ο χρήστης μέχρι να συναντήσει τον `'\n'` χαρακτήρα. Κάθε φορά, τον αποθηκεύει στην `ch` και τον συγκρίνει με την `EOF` τιμή για να ελέγξει ότι έχει έγκυρη τιμή

Παράδειγμα (2)

- Μία πολύ συνηθισμένη περίπτωση που κάνει πολλούς να αναρωτιούνται γιατί το πρόγραμμά τους δεν λειτουργεί σωστά είναι όταν μετά το διάβασμα αριθμητικών τιμών ακολουθεί διάβασμα χαρακτήρων. Το παρακάτω πρόγραμμα αποτελεί ένα τέτοιο παράδειγμα. Εντοπίζετε κάποια **δυσλειτουργία** όταν ο χρήστης εισάγει έναν ακέραιο και πατήσει *Enter*;

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

int main()
{
  int i, ch;

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

  cout << "Enter character: ";
  ch = cin.get();
  cout << i << ' ' << (char)ch << '\n';
  return 0;
}
```

Παράδειγμα (2)

- Απάντηση. Όταν ο χρήστης εισάγει έναν ακέραιο και πατήσει *Enter*, ο χαρακτήρας `'\n'` που δημιουργείται αποθηκεύεται στην ουρά εισόδου. Η `get()` τον παίρνει από εκεί, τον εκχωρεί στη μεταβλητή `ch` και έτσι δεν αφήνει τον χρήστη να εισάγει κάποιο χαρακτήρα. Άρα, το πρόγραμμα εμφανίζει μόνο τον ακέραιο. Υπάρχουν διάφοροι τρόποι για να ξεφορτωθούμε τον χαρακτήρα `'\n'`. Ένας τρόπος είναι να προσθέσουμε άλλη μία εντολή `get()` πριν από τη δεύτερη `cout`. Δηλαδή, `cin.get();` Επίσης, αν αντιστρέψουμε τη σειρά διαβάσματος, δηλαδή, πρώτα διαβάσουμε τον χαρακτήρα και μετά τον ακέραιο, το πρόγραμμα θα εκτελεστεί σωστά, αφού οι λευκοί χαρακτήρες, όπως ο `'\n'`, παραλείπονται πριν από μία αριθμητική τιμή

Παράδειγμα (3)

- Να γραφεί ένα πρόγραμμα το οποίο να εμφανίζει όλα τα πεζά γράμματα του λατινικού αλφαβήτου σε μία γραμμή, τα κεφαλαία γράμματα σε μία δεύτερη γραμμή και τους χαρακτήρες που αντιστοιχούν στα ψηφία 0-9 σε μία τρίτη γραμμή. Να χρησιμοποιήσετε μόνο έναν `for` βρόχο

Παράδειγμα (3)

```
#include <iostream>
int main()
{
 char ch, end_ch;

 end_ch = 'z';
 for(ch = 'a'; ch <= end_ch; ch++)
 {
        std::cout << ch << ' ';
        if(ch == 'z')
        {
                ch = 'A'-1; /* Αφαιρούμε 1, ώστε στην επόμενη
επανάληψη του βρόχου με την εντολή ch++ να γίνει ίση με 'A'. */
                end_ch = 'Z'; /* Αλλάζουμε τον τερματικό χαρακτήρα,
ώστε ο βρόχος να εμφανίσει τα κεφαλαία γράμματα. */
                std::cout << '\n';
        }
        else if(ch == 'Z')
        {
                ch = '0'-1;
                end_ch = '9';
                std::cout << '\n';
        }
 }
 return 0;
}
```

C++: Από τη Θεωρία στην Εφαρμογή

Κεφάλαιο 10ο
Αλφαριθμητικά

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 N χαρακτήρων πρέπει να είναι N+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:

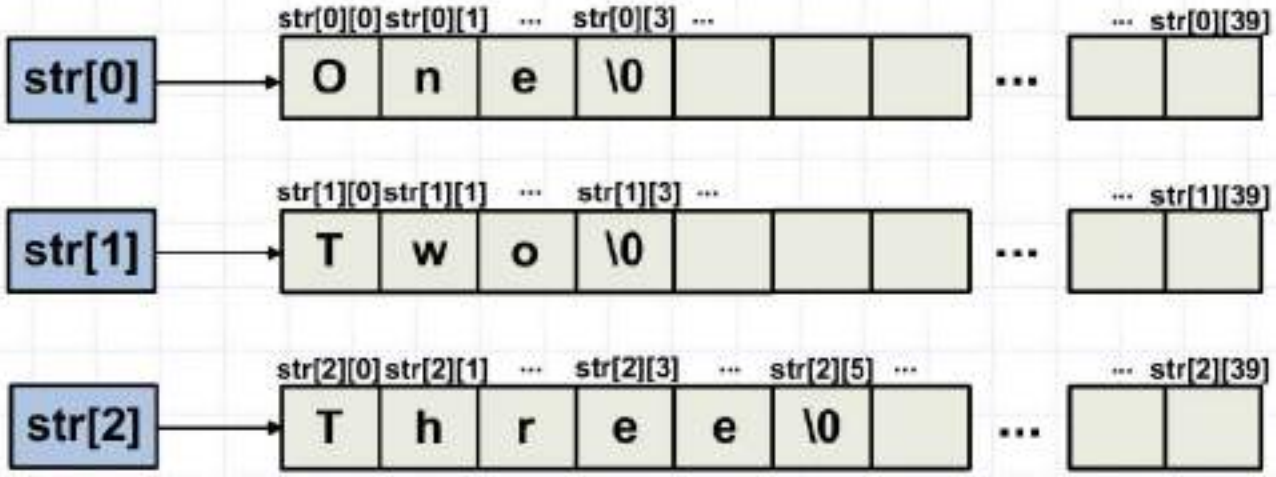
  ```
  char str[3][40] = {"One", "Two", "Three"};
  ```

  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.

C++: Από τη Θεωρία στην Εφαρμογή

Κεφάλαιο 11ο
Συναρτήσεις

Συναρτήσεις

- Μία **συνάρτηση** είναι ένα ανεξάρτητο τμήμα κώδικα που έχει όνομα
- Όταν κληθεί εκτελείται ο κώδικάς της και όταν τελειώσει μπορεί **προαιρετικά** να επιστρέψει μία τιμή
- Η συγγραφή προγραμμάτων με χρήση συναρτήσεων που εκτελούν ανεξάρτητες εργασίες αποτελεί τη βάση του λεγόμενου **δομημένου προγραμματισμού**
- Με τη χρήση συναρτήσεων ένα πρόγραμμα χωρίζεται σε μικρότερα τμήματα, άρα ο κώδικας αναπτύσσεται, διαβάζεται, τροποποιείται και ελέγχεται πιο εύκολα
- Επίσης, επειδή μία συνάρτηση μπορεί να κληθεί όσες φορές θέλουμε μέσα σε ένα πρόγραμμα, αποφεύγουμε τη συγγραφή του ίδιου κώδικα και έτσι μειώνεται το μέγεθος του προγράμματος
- Γενικά, η χρήση συναρτήσεων (δεν αναφέρομαι σε μικρά και απλά προγράμματα) είναι απαραίτητη για την καλύτερη οργάνωση και διαχείριση του προγράμματος
- Μέχρι τώρα, η μοναδική συνάρτηση που έχουμε γράψει είναι η συνάρτηση `main()`, ενώ τώρα θα μάθετε να γράφετε δικές σας συναρτήσεις και να τις χρησιμοποιείτε στα προγράμματά σας

Δήλωση Συνάρτησης

- Η **δήλωση** (ή αλλιώς **πρωτότυπο**) μίας συνάρτησης καθορίζει το **όνομα** της συνάρτησης, τον **τύπο επιστροφής** της και **μία λίστα παραμέτρων**
- Η γενική περίπτωση δήλωσης μίας συνάρτησης έχει την παρακάτω μορφή:

```
τύπος_επιστροφής όνομα_συνάρτησης(λίστα_παραμέτρων);
```

- Το όνομα_συνάρτησης πρέπει να είναι μοναδικό μέσα στο πρόγραμμα, δηλαδή να μην υπάρχει άλλη μεταβλητή ή συνάρτηση με το ίδιο όνομα
- Η δήλωση της συνάρτησης πρέπει να τελειώνει πάντοτε με το ελληνικό ερωτηματικό ;
- Το πρωτότυπο της συνάρτησης ενημερώνει τον μεταγλωττιστή για τον τύπο των παραμέτρων, τον αριθμό τους, τον τύπο επιστροφής, ώστε να μπορεί να ελέγξει αν ο τρόπος που καλείται στο πρόγραμμα είναι σύμφωνα με αυτό
- Προσπαθήστε να επιλέγετε περιγραφικά ονόματα για τις συναρτήσεις σας, ώστε αυτός που διαβάζει τον κώδικα της συνάρτησης να αντιλαμβάνεται γρήγορα τον σκοπό της

Πού γράφουμε τη Δήλωση (Πρωτότυπο) μίας Συνάρτησης:

- Συνήθως, οι δηλώσεις των συναρτήσεων περιέχονται σε διαφορετικό αρχείο από τον κώδικα τους. Για παράδειγμα, οι δηλώσεις των συναρτήσεων βιβλιοθήκης περιέχονται σε διάφορα αρχεία επικεφαλίδας

- Όταν θέλουμε να χρησιμοποιήσουμε κάποια συνάρτηση που η δήλωσή της περιέχεται σε ξεχωριστό αρχείο, χρησιμοποιούμε την οδηγία `#include` για να συμπεριλάβουμε το αρχείο που περιέχει τη δήλωσή της. Π.χ. για να χρησιμοποιήσουμε την `sqrt()` συμπεριλαμβάνουμε το αρχείο `cmath`

- Όπως θα δούμε στη συνέχεια, αν η συνάρτηση οριστεί πριν από το σημείο κλήσης της μπορούμε να παραλείψουμε το πρωτότυπό της. Σε αυτή την περίπτωση, ο ορισμός παρέχει στον μεταγλωττιστή την πληροφορία που χρειάζεται για να χειριστεί την κλήση της συνάρτησης

- Γενικά, αυτό που συνηθίζεται είναι οι δηλώσεις συναρτήσεων να τοποθετούνται σε ένα αρχείο επικεφαλίδας, το οποίο να συμπεριλαμβάνεται με την οδηγία `#include` σε όποιο αρχείο κώδικα την χρειάζεται

- Για απλότητα, θα δηλώνουμε τις συναρτήσεις που γράφουμε στο ίδιο αρχείο με τη `main()`

Επιστροφή Συνάρτησης

- Μία συνάρτηση μπορεί να επιστρέψει το πολύ μία τιμή
- Ο `τύπος_επιστροφής` μίας συνάρτησης καθορίζει τον τύπο της τιμής επιστροφής
- Μία συνάρτηση μπορεί να επιστρέψει οποιοδήποτε τύπο δεδομένων, όπως `char`, `int`, `double`, ... ή δείκτη σε κάποιον τύπο
- **Μοναδικός περιορισμός** είναι ότι ο τύπος επιστροφής δεν επιτρέπεται να είναι πίνακας ή συνάρτηση. Επιτρέπεται, όμως, να επιστρέψει δείκτη σε πίνακα ή συνάρτηση
- Ο τύπος επιστροφής `void` δηλώνει ότι η συνάρτηση **δεν επιστρέφει** κάποια τιμή
- Με την C++11 η συνάρτηση μπορεί να επιστρέψει μία λίστα τιμών μέσα σε άγκιστρα. Η λίστα χρησιμοποιείται για την αρχικοποίηση της προσωρινής μεταβλητής που χρησιμοποιείται για τιμή επιστροφής. Π.χ.

```
vector<int> f()
{
        return {4, 5, 6};
}
int main()
{
        vector<int> v = f(); /* Το v αρχικοποιείται με τις τιμές 4,
5, και 6. */
}
```

Παράμετροι Συνάρτησης

- Μία συνάρτηση μπορεί **προαιρετικά** να δέχεται μία λίστα παραμέτρων (`λίστα_παραμέτρων`), που χωρίζονται μεταξύ τους με κόμμα (,)
- Κάθε παράμετρος χαρακτηρίζεται από τον τύπο της
- Η παράμετρος μίας συνάρτησης είναι ουσιαστικά μία μεταβλητή της συνάρτησης, η οποία θα αρχικοποιηθεί με μία τιμή, όταν θα γίνει η κλήση της
- Αν η συνάρτηση **δεν δέχεται παραμέτρους** χρησιμοποιούμε κενές παρενθέσεις ()

Παραδείγματα Δήλωσης Συναρτήσεων

```
void show(); /* Δήλωση μίας συνάρτησης με όνομα show, η
οποία δεν δέχεται παραμέτρους και δεν επιστρέφει κάτι. */

double show(char ch, int a, float b); /* Δήλωση μίας
συνάρτησης με όνομα show, η οποία δέχεται ένα χαρακτήρα,
μία ακέραια και μία πραγματική παράμετρο τύπου float και
επιστρέφει μία πραγματική τιμή τύπου double. */

double *show(int *p1, int *p2); /* Δήλωση μίας συνάρτησης
με όνομα show, η οποία δέχεται σαν παραμέτρους δύο δείκτες
σε ακεραίους και επιστρέφει έναν δείκτη σε τύπο double.*/
```

Παρατηρήσεις

- Σημειώστε ότι τα ονόματα των παραμέτρων δεν είναι υποχρεωτικά, αρκεί ο τύπος τους, π.χ., θα μπορούσαμε να γράψουμε:

```
double show(char, int, float);
```

- Η συνήθης προτίμηση είναι να προστίθενται τα ονόματα, ώστε αυτός που απλά διαβάζει το πρωτότυπο της συνάρτησης ή σκοπεύει να χρησιμοποιήσει τη συνάρτηση να παίρνει μία ιδέα για την πληροφορία που της διοχετεύεται, με ποια σειρά και τον σκοπό της κάθε παραμέτρου

- Ο τύπος της κάθε παραμέτρου πρέπει να καθορίζεται ακόμα και αν όλοι οι παράμετροι έχουν τον ίδιο τύπο. Π.χ. είναι **λάθος** να γράψουμε

```
void test(int a, b, c);
```

αντί για:

```
void test(int a, int b, int c);
```

Ορισμός Συνάρτησης (1)

- Ο **ορισμός** μίας συνάρτησης (function definition) περιέχει τον κώδικα της συνάρτησης. Η γενική μορφή του ορισμού μίας συνάρτησης είναι:

```
τύπος_επιστροφής όνομα_συνάρτησης(λίστα_παραμέτρων)
{
      /* Σώμα Συνάρτησης */
}
```

- Ο ορισμός μίας συνάρτησης δεν επιτρέπεται να μοιράζεται σε διαφορετικά αρχεία. Επίσης, κάθε συνάρτηση πρέπει να έχει ένα μοναδικό ορισμό

- Η πρώτη γραμμή πρέπει να ταιριάζει με τη δήλωση της συνάρτησης, αλλά δεν χρειάζεται να είναι ίδια. Για παράδειγμα, τα ονόματα των παραμέτρων δεν χρειάζεται να είναι τα ίδια με τα ονόματα στη δήλωση της συνάρτησης. Επίσης, σημειώστε ότι δεν προστίθεται το ερωτηματικό ; στο τέλος της

- Ο **κώδικας** ή, αλλιώς, το **σώμα** της συνάρτησης είναι ότι περιέχεται ανάμεσα στα άγκιστρα. Το σώμα της συνάρτησης μπορεί να είναι άδειο

- Ο κώδικας μίας συνάρτησης εκτελείται μόνο όταν αυτή κληθεί από κάποιο σημείο του προγράμματος

- Η εκτέλεση μίας συνάρτησης τερματίζει αν κληθεί μία εντολή τερματισμού (π.χ. `return`) ή όταν εκτελεστεί η τελευταία εντολή της

Παράδειγμα Χρήσης Συνάρτησης

```
#include <iostream>

void test(); /* Πρωτότυπο συνάρτησης.*/

int main(void)
{
    test(); /* Κλήση συνάρτησης. */
    return 0;
}

void test()/* Ορισμός συνάρτησης. */
{
    /* Σώμα συνάρτησης. */
    std::cout << "In\n";
}
```

Το πρόγραμμα εμφανίζει In

Δήλωση Συνάρτησης σε Ξεχωριστό Αρχείο (1)

- Για να δηλώσετε μία συνάρτηση σε ξεχωριστό αρχείο:
  - Δημιουργήστε ένα αρχείο, π.χ., prototype.h
  - Αντιγράψτε το πρωτότυπο (τη δήλωση) μέσα στο αρχείο αυτό
  - Αφαιρέστε τη δήλωση απ' το αρχείο του πηγαίου κώδικα όπου την είχατε έως τώρα
  - Χρησιμοποιήστε την οδηγία #include για να συμπεριλάβετε το αρχείο prototype.h όπως παρακάτω:

Έστω ότι το αρχείο έχει όνομα program.cpp

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

int main()
{
    ...
}
void test()
{
    ...
}
```

Το αρχείο prototype.h περιέχει μόνο τη δήλωση (πρωτότυπο) της test() δηλ.:

```
void test();
```

Παρατηρήσεις

- Αν ο ορισμός της συνάρτησης γίνει πριν από το πρώτο σημείο κλήσης της, η δήλωσή της μπορεί να παραλειφθεί. Π.χ.

```
#include <iostream>

void test()
{
   std::cout << "In\n";
}

int main()
{
   test();
   return 0;
}
```

- Αν και τα προγράμματά μας είναι μικρά και θα μπορούσαμε να ορίζουμε τις συναρτήσεις μας πριν από την **main()**, στη γενική περίπτωση μία τέτοια πρακτική δεν είναι πάντα εφαρμόσιμη
- Για παράδειγμα, σε μεγάλες εφαρμογές που ο κώδικας μοιράζεται σε πολλά αρχεία, η συνάρτηση που θέλουμε να καλέσουμε σε ένα αρχείο μπορεί να ορίζεται σε ένα διαφορετικό αρχείο. Τότε, η ύπαρξη πρωτοτύπου είναι υποχρεωτική, ώστε να γνωρίζει ο μεταγλωττιστής τον τρόπο που θα την καλέσει
- Επιπλέον, αν υπάρχουν πολλές συναρτήσεις η τοποθέτησή τους σε σωστή σειρά ώστε κάθε μία από αυτές να ορίζεται πριν κληθεί από κάποια άλλη απαιτεί χρόνο ή μπορεί να είναι και αδύνατη
- Επίσης, η ύπαρξη πρωτοτύπων επιτρέπει σε κάποιον που τα διαβάζει να αποκτήσει γρήγορα μια γενική εικόνα της λειτουργικότητας των συναρτήσεων
- Άρα, μαθαίνουμε να χρησιμοποιούμε πρωτότυπα

Σύνθετες Δηλώσεις (1)

- Αφού είδαμε πώς ορίζουμε συναρτήσεις, ας κάνουμε μία παρένθεση για να μάθουμε **πώς αναλύουμε σύνθετες δηλώσεις**
- Αναλύουμε κάθε δήλωση **πάντα «από μέσα προς τα έξω»**
- Βρίσκουμε το όνομα της μεταβλητής και, **αν εσωκλείεται σε παρενθέσεις** (), **αρχίζουμε την ανάλυση από εκεί** και συνεχίζουμε με τον επόμενο τελεστή, όπου η προτεραιότητα από την υψηλότερη στη χαμηλότερη είναι:
  - α) οι **μεταθεματικοί τελεστές**, δηλ. οι παρενθέσεις () που υποδεικνύουν συνάρτηση και οι αγκύλες [] που υποδεικνύουν πίνακα
  - β) ο **προθεματικός τελεστής** * υποδεικνύει «δείκτη σε...»

Σύνθετες Δηλώσεις (2)

- Π.χ., έστω η δήλωση: `int *p[5];`

  Το όνομα p δεν περικλείεται σε παρενθέσεις κι επειδή οι [] προηγούνται του * πάμε δεξιά και έχουμε ότι το p είναι «πίνακας πέντε...», πάμε αριστερά στο * και η δήλωση γίνεται «πίνακας πέντε δεικτών σε...», προσθέτουμε και τον τύπο και καταλήγουμε στη δήλωση: «πίνακας πέντε δεικτών σε ακεραίους»

- Άλλο ένα, έστω η δήλωση: `int (*p)[5];`

  Τώρα το p περικλείεται σε παρενθέσεις, κι επειδή υπάρχει μέσα στην παρένθεση το * έχουμε ότι το p είναι «δείκτης σε...», πάμε δεξιά τώρα στις [] και η δήλωση γίνεται «δείκτης σε πίνακα πέντε...», πάμε και αριστερά και καταλήγουμε στη δήλωση: «δείκτης σε πίνακα πέντε ακεραίων»

- Άντε κι άλλο ένα, έστω η δήλωση: `int *(*p)(int);`

  Ξεκινάμε όμοια με πριν από την παρένθεση ότι το p είναι «δείκτης σε...», οι () προηγούνται του *, άρα πάμε δεξιά και έχουμε «δείκτης σε συνάρτηση με όρισμα ακέραιο...», πάμε αριστερά στον τύπο επιστροφής και καταλήγουμε στη δήλωση «δείκτης σε συνάρτηση με όρισμα ακέραιο που επιστρέφει δείκτη σε ακέραιο»

Η Εντολή return (1)

- Η εντολή **return** χρησιμοποιείται για τον **άμεσο τερματισμό** μίας συνάρτησης, δηλ. αν η εκτέλεση του κώδικα της συνάρτησης φτάσει σε μία εντολή **return**, τότε η συνάρτηση **τερματίζεται**
- Άρα, αν εκτελεστεί η εντολή **return** μέσα στη συνάρτηση **main()**, τότε **το πρόγραμμα τερματίζει**. Π.χ.

```
#include <iostream>
int main()
{
   int num;
   while(1)
   {
          std::cout << "Enter number: ";
          std::cin >> num;

          if(num == 2)
                 return 0; // Τερματισμός προγράμματος.
          else /* Το else δε χρειάζεται, απλά για ευκολία στο παράδειγμα. */
                 std::cout << num << '\n';
   }
   return 0;
}
```

- Το παραπάνω πρόγραμμα τερματίζει, αν ο χρήστης εισάγει την τιμή 2, αλλιώς εμφανίζει την εισαγόμενη τιμή (παρατηρήστε ότι η τελευταία **return** δεν θα εκτελεστεί ποτέ, αφού μόνο η πρώτη **return** είναι αυτή που μπορεί να τερματίσει το πρόγραμμα, λόγω του ατέρμονου βρόχου)
- Η τιμή που επιστρέφει η **main()** δηλώνει τον τρόπο τερματισμού του προγράμματος. Η τιμή 0 δηλώνει τον ομαλό τερματισμό του προγράμματος, ενώ μία μη μηδενική τιμή δηλώνει συνήθως μία εσφαλμένη συνθήκη τερματισμού

Η Εντολή return (2)

- Αν η συνάρτηση δεν έχει οριστεί να επιστρέφει κάποια τιμή (δηλ. αν ο επιστρεφόμενος τύπος της είναι `void`), τότε - για να τερματίσουμε άμεσα σε κάποιο σημείο τη συνάρτηση - γράφουμε απλά `return;`
- Επίσης, σε αυτή την περίπτωση, η `return` στο τέλος της συνάρτησης (πριν το άγκιστρο «κλεισίματος» }) δεν είναι απαραίτητη, αφού η συνάρτηση θα επιστρέψει αυτόματα, δεδομένου ότι τελειώνει το «σώμα» της
- Αν, όμως, η συνάρτηση έχει οριστεί να επιστρέφει κάποια τιμή, τότε η εντολή `return` **πρέπει να ακολουθείται** από κάποια τιμή. Αυτή η τιμή επιστρέφεται στο σημείο κλήσης της συνάρτησης. Η καλούσα συνάρτηση είναι ελεύθερη να χρησιμοποιήσει ή να αγνοήσει την επιστρεφόμενη τιμή. Π.χ.

```
int f(int a, int b)
{
   if(a == b)
          return 0;
   cout << (a+b)/2.0;
   return 1;
}
```

- Ο τύπος της τιμής που επιστρέφεται πρέπει να **ταιριάζει** με τον τύπο που ορίστηκε να επιστρέφει η συνάρτηση στη δήλωσή της. Αν δεν ταιριάζουν, ο μεταγλωττιστής θα μετατρέψει την επιστρεφόμενη τιμή, αν αυτό είναι δυνατό, στον τύπο επιστροφής της συνάρτησης. Π.χ:

```
int test()
{
   return 4.9;
}
```

Αφού η `test()` έχει δηλωθεί να επιστρέφει `int`, η τιμή επιστροφής θα μετατραπεί σε `int`. Επομένως, θα επιστραφεί η τιμή **4**

Κλήση Συνάρτησης

- Όταν καλείται μία συνάρτηση, η εκτέλεση του προγράμματος συνεχίζει με την εκτέλεση του κώδικα της συνάρτησης
- Όταν τερματίζεται η συνάρτηση, η εκτέλεση του προγράμματος **επιστρέφει στο σημείο κλήσης** της συνάρτησης και συνεχίζει με την εκτέλεση της επόμενης εντολής
- Μία συνάρτηση μπορεί να κληθεί **όσες φορές είναι απαραίτητο** για τους σκοπούς του προγράμματος
- Όταν γίνεται η κλήση μίας συνάρτησης, ο μεταγλωττιστής **δεσμεύει μνήμη** για να αποθηκεύσει τις μεταβλητές που δηλώνονται στη λίστα παραμέτρων της συνάρτησης, καθώς και αυτές που δηλώνονται μέσα στο σώμα της
- Συνήθως, αυτή η μνήμη **δεσμεύεται** από ένα συγκεκριμένο τμήμα μνήμης που παρέχει το λειτουργικό σύστημα στο πρόγραμμα και ονομάζεται **στοίβα** (**stack**), χωρίς αυτό να είναι υποχρεωτικό
- Η **αποδέσμευση** αυτής της μνήμης **γίνεται αυτόματα** όταν τερματιστεί η εκτέλεση της συνάρτησης

Παρατηρήσεις

- Όταν χρησιμοποιείτε μία συνάρτηση βιβλιοθήκης, θα πρέπει να συμπεριλάβετε το αρχείο που περιέχει τη δήλωσή της με την οδηγία `#include`

- Για παράδειγμα, για να χρησιμοποιηθεί η συνάρτηση `strlen()` (δηλ. για να μπορέσετε να καλέσετε τη συγκεκριμένη συνάρτηση σε ένα πρόγραμμά σας), πρέπει να προστεθεί το αρχείο `cstring` με την οδηγία:

```
#include <cstring>
```

αλλιώς ο μεταγλωττιστής θα εμφανίσει μήνυμα λάθους για αδήλωτη συνάρτηση

Κλήση Συνάρτησης χωρίς Παραμέτρους

- Η κλήση μίας συνάρτησης που δεν δέχεται παραμέτρους γίνεται γράφοντας το όνομά της ακολουθούμενο από κενές παρενθέσεις ()
- Στη συνάρτηση δεν μεταβιβάζεται πληροφορία
- Στο επόμενο παράδειγμα, η καλούσα συνάρτηση, δηλαδή η main(), καλεί δύο φορές τη συνάρτηση test()

```
#include <iostream>
void test();
int main()
{
  std::cout << "Call_1 ";
  test(); /* Κλήση συνάρτησης. Οι παρενθέσεις είναι κενές, γιατί
  η συνάρτηση δεν δέχεται παραμέτρους. */
  std::cout << "Call_2 ";
  test(); // Δεύτερη κλήση.
  return 0;
}
void test()
{
  // Σώμα συνάρτησης.
  for(int i = 0; i < 2; i++)
        std::cout << "In ";
}
```

Το πρόγραμμα εμφανίζει `Call_1 In In Call_2 In In`

Παράδειγμα Κλήσης Συνάρτησης Χωρίς Παραμέτρους που Επιστρέφει Τιμή

```
#include <iostream>

int test();

int main()
{
  int sum;

  sum = test(); /* Κλήση συνάρτησης. Η τιμή που
  επιστρέφει η test() αποθηκεύεται στη sum. */
  std::cout << sum << '\n';
  return 0;
}

int test()
{
  int i = 10, j = 20;
  return i+j;
}
```

Το πρόγραμμα εμφανίζει 30

Κλήση Συνάρτησης με Παραμέτρους (1)

- Η κλήση μίας συνάρτησης που δέχεται παραμέτρους σημαίνει ότι στη συνάρτηση μεταβιβάζεται πληροφορία μέσω των ορισμάτων της
- Η διαφορά μεταξύ παραμέτρου και ορίσματος είναι ότι ο όρος παράμετρος αναφέρεται στις μεταβλητές που εμφανίζονται στον ορισμό της συνάρτησης, ενώ ο όρος όρισμα αναφέρεται στις εκφράσεις που εμφανίζονται στην κλήση της συνάρτησης. Π.χ.

```
#include <iostream>

int test(int x, int y);

int main()
{
  int sum, a = 10, b = 20;

  sum = test(a, b); /* Οι μεταβλητές a και b είναι τα ορίσματα της
  συνάρτησης. */
  std::cout << sum << '\n';
  return 0;
}

int test(int x, int y) /* Οι μεταβλητές x και y είναι οι παράμετροι της
  συνάρτησης. */
{
  return x+y;
}
```

Κλήση Συνάρτησης με Παραμέτρους (2)

- Η κλήση της συνάρτησης γίνεται γράφοντας **το όνομά της** και μέσα σε παρενθέσεις **τη λίστα ορισμάτων**
- Ένα όρισμα μπορεί να είναι μία οποιαδήποτε έγκυρη έκφραση, όπως μία σταθερά, μία μεταβλητή, μία μαθηματική ή λογική έκφραση ή ακόμη και μία άλλη συνάρτηση με τιμή επιστροφής
- Το **πλήθος** των ορισμάτων και οι **τύποι** τους <u>πρέπει να ταιριάζουν</u> με τη δήλωση της συνάρτησης
- Αν το πλήθος είναι μικρότερο, ο μεταγλωττιστής θα εμφανίσει μήνυμα λάθους
- Αν οι τύποι των ορισμάτων δεν ταιριάζουν, ο μεταγλωττιστής θα προσπαθήσει να μετατρέψει τους τύπους των ασύμβατων ορισμάτων στους τύπους των αντίστοιχων παραμέτρων
- Αν τα καταφέρει, μπορεί να εμφανίσει μήνυμα προειδοποίησης (warning) για να ενημερώσει τον προγραμματιστή για τη μετατροπή. Αν δεν τα καταφέρει, θα εμφανίσει μήνυμα λάθους
- Όταν καλείται μία συνάρτηση οι τιμές της λίστας των ορισμάτων εκχωρούνται **μία-προς-μία** στις παραμέτρους της συνάρτησης

Επεξήγηση Παραδείγματος

- Ας επανέλθουμε στο πρόγραμμα. Όταν εκτελείται, ο μεταγλωττιστής δεσμεύει οκτώ θέσεις μνήμης (όπως έχουμε πει, θεωρούμε ότι ο `int` τύπος χρειάζεται τέσσερις οκτάδες) για την αποθήκευση των τιμών των `a` και `b`

- Όταν καλείται η `test()`, ο μεταγλωττιστής δεσμεύει άλλες οκτώ θέσεις για την αποθήκευση των τιμών των `x` και `y`

- Στη συνέχεια, αντιγράφει τις τιμές των ορισμάτων `a` και `b` (δηλ. 10 και 20) στις αντίστοιχες θέσεις μνήμης των παραμέτρων `x` και `y`

- Ουσιαστικά, κάθε παράμετρος μίας συνάρτησης είναι μία μεταβλητή που δημιουργείται όταν καλείται η συνάρτηση. Η τιμή με την οποία αρχικοποιείται είναι η τιμή του αντίστοιχου ορίσματος

- Όταν τερματιστεί η εκτέλεση της συνάρτησης `test()` γίνεται αυτόματη αποδέσμευση της μνήμης που είχε δεσμευτεί για τις παραμέτρους `x` και `y`

Κλήση Μέσω Τιμής

- Αφού οι διευθύνσεις μνήμης των **x** και **y** είναι διαφορετικές από τις διευθύνσεις μνήμης των **a** και **b**, οποιαδήποτε αλλαγή γίνει στις τιμές των **x** και **y** δεν επηρεάζει τις τιμές των **a** και **b**

- Αυτός ο τρόπος μεταβίβασης τιμών ονομάζεται **κλήση κατ'αξία** ή **κλήση μέσω τιμής** (*call by value*)

- Δηλαδή, η καλούμενη συνάρτηση δέχεται τις τιμές των ορισμάτων της σε αντίγραφα των πρωτότυπων μεταβλητών. Επομένως, μπορεί να αλλάξει μόνο την τιμή του αντιγράφου της και όχι της πρωτότυπης μεταβλητής

- Όσον αφορά την απόδοση, το κόστος αυτού του τρόπου μεταβίβασης είναι η μνήμη και ο χρόνος που χρειάζεται για την αντιγραφή των ορισμάτων

- Όπως θα δούμε στη συνέχεια, εξαίρεση σε αυτόν τον κανόνα αποτελούν οι πίνακες. Όταν ένας πίνακας μεταβιβάζεται σε μία συνάρτηση, δεν δημιουργείται αντίγραφο του πίνακα, αλλά μεταβιβάζεται η διεύθυνση του πρώτου του στοιχείου. Η συνάρτηση μπορεί να αλλάξει την τιμή οποιουδήποτε στοιχείου του πίνακα

Παράδειγμα

- Πριν συνεχίσουμε στην επόμενη ενότητα, ας κάνουμε μία άσκηση. Βρείτε τα λάθη στο παρακάτω πρόγραμμα:

```
#include <iostream> // Πρόγραμμα με λάθη μεταγλώττισης.

void f(int a, int b);

int main()
{
  int i, j, k;

  k = f(i, &j);
  return 0;
}

void f(int a, int b)
{
  if(a > 0)
        return b;
}
```

Παράδειγμα

Απάντηση. Η κλήση της `f()` είναι λάθος, γιατί ο τύπος της έκφρασης `&j` είναι δείκτης σε ακέραιο, και όχι ακέραιος όπως θα έπρεπε να είναι σύμφωνα με το πρωτότυπο. Η ανάθεση τιμής στο `k` είναι λάθος, γιατί η `f()` δεν επιστρέφει τιμή. Στην `f()`, η εντολή `return b;` είναι λάθος, γιατί η `f()` δεν επιστρέφει τιμή.

Αλλαγή Πρωτότυπων Τιμών

- Αν επιθυμούμε μία συνάρτηση να μπορεί να αλλάξει την τιμή κάποιου ορίσματος, πρέπει να μεταβιβάσουμε στη συνάρτηση τη διεύθυνση μνήμης του ορίσματος και όχι την τιμή του

- Επομένως, αφού η συνάρτηση θα έχει πρόσβαση στη διεύθυνση μνήμης του ορίσματος, θα μπορεί να μεταβάλλει την τιμή του

- Η αντίστοιχη παράμετρος πρέπει να δηλωθεί σαν δείκτης και η συνάρτηση μπορεί να χρησιμοποιήσει τον δείκτη για να αποκτήσει πρόσβαση στην μεταβλητή

Παράδειγμα

```
#include <iostream>

void test(int *p, int a);

int main()
{
  int i = 10, j = 20;

  test(&i, j);
  std::cout << i << ' ' << j << '\n';
  return 0;
}

void test(int *p, int a)
{
  *p = 30;
  a = 40;
}
```

Αφού ο τύπος της έκφρασης &i είναι δείκτης σε ακέραιο, η κλήση της test() ταιριάζει με το πρωτότυπο της συνάρτησης. Όταν γίνεται η κλήση της έχουμε p = &i. Αφού ο δείκτης p δείχνει στο i, η test() μπορεί να αλλάξει την τιμή του i. Ειδικότερα, το *p είναι ισοδύναμο με το i και η εντολή *p = 30; αλλάζει την τιμή του i από 10 σε 30. Όπως φαίνεται, όταν γίνεται κλήση μίας συνάρτησης, επιτρέπεται να γίνει συνδυασμός στη μεταβίβαση των ορισμάτων, δηλαδή, για κάποια ορίσματα να διοχετευθούν οι διευθύνσεις τους και για κάποια άλλα οι τιμές τους. Έτσι, η τιμή του j δεν αλλάζει και το πρόγραμμα εμφανίζει 30 και 20

Παρατηρήσεις

- Αφού μία συνάρτηση δεν μπορεί να επιστρέψει περισσότερες από μία τιμές, η μεταβίβαση των διευθύνσεων των ορισμάτων αποτελεί έναν ευέλικτο τρόπο για την αλλαγή των τιμών τους. Αυτή είναι μία μεγάλη χρησιμότητα των δεικτών
- Όσον αφορά την απόδοση, όταν μεταβιβάζουμε απλές μεταβλητές (π.χ. `int`) τις μεταβιβάζουμε όπως έχουμε δει, ενώ όταν μεταβιβάζουμε μεγάλες οντότητες (π.χ. ένα vector που περιέχει χιλιάδες στοιχεία) είναι πιο αποδοτικό να μεταβιβάσουμε την διεύθυνση μνήμης της οντότητας, ώστε να αποφύγουμε τον χρόνο και τη μνήμη που απαιτείται για την αντιγραφή της οντότητας
- Στο Κ.16 θα μιλήσουμε για έναν εναλλακτικό τρόπο για να μεταβιβάσουμε την διεύθυνση, μέσω αναφοράς (*call by reference*)
- Αν θέλουμε μόνο να χρησιμοποιήσουμε την τιμή του ορίσματος και να αποτρέψουμε τυχόν αλλαγές κάνουμε τον δείκτη `const`. Π.χ.

```
void test(const int *p)
{
  if(*p == 10) // Σωστό.
       *p = 30; // Λάθος.
  ...
}
```

Παράδειγμα

- Ποια είναι η έξοδος του παρακάτω προγράμματος;

```
#include <iostream>

void test(int *arg);
int var = 100;

int main()
{
  int *ptr, i = 30;

  ptr = &i;
  test(ptr);
  std::cout << *ptr << '\n';
  return 0;
}

void test(int *arg)
{
  arg = &var;
}
```

Παράδειγμα

Απάντηση. Επειδή η `test()` δέχεται σαν όρισμα την τιμή του `ptr` και όχι τη διεύθυνσή του, δεν μπορεί να αλλάξει την τιμή του. Άρα, οποιαδήποτε μεταβολή στην τιμή του `arg` δεν επηρεάζει την τιμή του `ptr` και το πρόγραμμα εμφανίζει `30`. Για την εμβέλεια της `var` θα μιλήσουμε σε λίγο. Αν θέλουμε μία συνάρτηση να μπορεί να αλλάξει την τιμή ενός δείκτη μεταβιβάζουμε στη συνάρτηση όρισμα **δείκτη σε δείκτη**. Δείτε το επόμενο παράδειγμα

Παράδειγμα Μεταβίβασης Δείκτη σε Δείκτη

Αν θέλουμε μία συνάρτηση να μπορεί να αλλάξει την τιμή ενός δείκτη μεταβιβάζουμε στη συνάρτηση όρισμα δείκτη σε δείκτη. Π.χ.

```
#include <iostream>

void test(int **arg);
int var = 100;

int main()
{
  int *ptr, i = 30;

  ptr = &i;
  test(&ptr); /* Η τιμή του &ptr είναι η διεύθυνση μνήμης του ptr, ο οποίος με τη
  σειρά του δείχνει στη διεύθυνση του i. Άρα, ο τύπος του ορίσματος είναι δείκτης
  σε δείκτη σε ακέραιο και συμφωνεί με την int** δήλωση της συνάρτησης. */
  std::cout << *ptr << '\n';
  return 0;
}

void test(int **arg)
{
  *arg = &var;
}
```

Αφού στην `test()` μεταβιβάζεται η διεύθυνση της μεταβλητής `ptr`, η `test()` μπορεί να αλλάξει την τιμή της. Συγκεκριμένα, όταν καλείται η test() έχουμε arg = &ptr, άρα *arg = ptr. Επομένως, η έκφραση *arg = &var ισοδυναμεί με ptr = &var, δηλαδή, η τιμή του `ptr` αλλάζει και δείχνει στη διεύθυνση της `var`. Άρα, το πρόγραμμα εμφανίζει 100

Παράδειγμα (1)

- Δημιουργήστε μία συνάρτηση που να δέχεται σαν παράμετρο έναν ακέραιο και να επιστρέφει το τετράγωνο του αριθμού και άλλη μία που να επιστρέφει τον κύβο του αριθμού. Να γραφεί ένα πρόγραμμα το οποίο να διαβάζει έναν ακέραιο και να εμφανίζει το άθροισμα του τετραγώνου και του κύβου του αριθμού με χρήση των συναρτήσεων

Παράδειγμα (1)

```
#include <iostream>

int square(int a);
int cube(int a);

int main()
{
  int i, j, k;

  std::cout << "Enter number: ";
  std::cin >> i;

  j = square(i);
  k = cube(i);
  std::cout << j+k << '\n'; /* Θα μπορούσαμε να μην χρησιμοποιήσουμε τις
  μεταβλητές j, k και να γράψουμε κατευθείαν cout << square(i)+cube(i); */
  return 0;
}

int square(int a)
{
   return a*a;
}

int square(int a)
{
   return a*a*a;
}
```

Παράδειγμα (2)

- Ποια είναι η έξοδος του παρακάτω προγράμματος;

```
#include <iostream>

int f(int a);

int main()
{
  int i = 10;

  std::cout << f(3-f(2*f(i+1))) << '\n';
  return 0;
}

int f(int a)
{
   return a+1;
}
```

Παράδειγμα (2)

- Απάντηση. Κάθε φορά που καλείται η `f()` επιστρέφει την τιμή του ορίσματος αυξημένη κατά ένα. Οι κλήσεις της `f()` εκτελούνται από μέσα προς τα έξω. Άρα, η πρώτη κλήση επιστρέφει την τιμή `12` και το όρισμα της δεύτερης κλήσης είναι η τιμή `24`. Η δεύτερη κλήση επιστρέφει την τιμή `25` και το όρισμα της τρίτης κλήσης είναι η τιμή `-22`. Επομένως, το πρόγραμμα θα εμφανίσει `-21`

Παράδειγμα (3)

- Δημιουργήστε μία συνάρτηση που να δέχεται σαν παραμέτρους έναν ακέραιο και έναν χαρακτήρα και να εμφανίζει τον χαρακτήρα τόσες φορές όσες και η τιμή του ακεραίου. Επίσης, η συνάρτηση με χρήση της εντολής `switch` να επιστρέφει τον ίδιο τον χαρακτήρα αν αυτός είναι `'a'`, `'b'` ή `'c'`, αλλιώς τον επόμενο χαρακτήρα στο ASCII σύνολο. Να γραφεί ένα πρόγραμμα το οποίο να διαβάζει έναν ακέραιο και έναν χαρακτήρα, να καλεί τη συνάρτηση και να εμφανίζει την τιμή επιστροφής

Παράδειγμα (3)

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

char show(int num, char c);

int main()
{
   char c;
   int i;

   cout << "Enter character: ";
   cin >> c;
   cout << "Enter number: ";
   cin >> i;

   c = show(i, c);
   cout << '\n' << c << '\n'; /* Θα μπορούσαμε να μην καλέσουμε σε ξεχωριστή εντολή τη
   show() και στη θέση του c να γράψουμε show(i, c) */
   return 0;
}

char show(int num, char c)
{
   for(int i = 0; i < num; i++)
          cout << c;

   switch(c)
   {
          case 'a':
          case 'b':
          case 'c':
          return c;

          default:
          return c+1;
   }
}
```

Παράδειγμα (4)

- Δημιουργήστε μία συνάρτηση που να δέχεται σαν παραμέτρους δύο δείκτες σε πραγματικούς και να αντιμεταθέτει τα περιεχόμενά τους. Να γραφεί ένα πρόγραμμα το οποίο να διαβάζει δύο πραγματικούς και να τους αντιμεταθέτει με χρήση της συνάρτησης

Παράδειγμα (4)

```
#include <iostream>

void swap(float *ptr1, float *ptr2);

int main()
{
  float i, j;

  std::cout << "Enter numbers: ";
  std::cin >> i >> j;

  swap(&i, &j);
  std::cout << i << ' ' << j << '\n';
  return 0;
}

void swap(float *ptr1, float *ptr2)
{
  float m;

  m = *ptr1; // Ισοδύναμο με m = i;
  *ptr1 = *ptr2; // Ισοδύναμο με i = j;
  *ptr2 = m; // Ισοδύναμο με j = m;
}
```

Εμβέλεια Μεταβλητών

- **Εμβέλεια** μίας μεταβλητής καλείται το τμήμα του προγράμματος, στο οποίο η μεταβλητή είναι **προσβάσιμη**, ή αλλιώς λέμε ότι **είναι «ορατή»** από τα υπόλοιπα τμήματα του προγράμματος

- Η εμβέλεια μίας μεταβλητής εξαρτάται από το σημείο δήλωσής της μέσα στο πρόγραμμα

- Τα διαφορετικά είδη μεταβλητών βάσει της εμβέλειάς των είναι:
  - Οι **τοπικές** μεταβλητές (**local** variables)
  - Οι **καθολικές** μεταβλητές (**global** variables)

Τοπικές Μεταβλητές

- Μία μεταβλητή που δηλώνεται στο σώμα μίας συνάρτησης ονομάζεται **τοπική**
- Η εμβέλειά της περιορίζεται μέσα στη συνάρτηση από το σημείο της δήλωσής της μέχρι το τέλος του τμήματος που περικλείει τη δήλωση και ονομάζεται εμβέλεια τμήματος. Αυτό σημαίνει ότι οι υπόλοιπες συναρτήσεις του προγράμματος **δεν** έχουν πρόσβαση σε αυτή
- Αφού μία τοπική μεταβλητή δεν είναι ορατή έξω από τη συνάρτηση στην οποία δηλώνεται, μπορούμε να δηλώσουμε μεταβλητές **με το ίδιο όνομα** σε άλλες συναρτήσεις
- Σημειώστε ότι οι παράμετροι που εμφανίζονται στη δήλωση μίας συνάρτησης θεωρούνται και αυτές τοπικές μεταβλητές της συνάρτησης
- Κάθε φορά που καλείται μία συνάρτηση ο μεταγλωττιστής δεσμεύει μνήμη για να δημιουργήσει τις τοπικές μεταβλητές της. Αυτή η μνήμη αποδεσμεύεται αυτόματα όταν τελειώσει η εκτέλεση της συνάρτησης, άρα, μία τοπική μεταβλητή δεν διατηρεί την τιμή της μεταξύ διαδοχικών κλήσεων

Παράδειγμα

- Στο επόμενο πρόγραμμα, η τοπική μεταβλητή i που δηλώνεται στη main() είναι διαφορετική από την τοπική μεταβλητή i που δηλώνεται στην test(), παρόλο που έχουν το ίδιο όνομα

```
#include <iostream>

void test();

int main()
{
   int i = 10;

   test();
   std::cout << "I_main = " << i << '\n';
   return 0;
}

void test()
{
   int i = 200;
   std::cout << "I_test = " << i << '\n';
}
```

Αφού είναι διαφορετικές μεταβλητές, η κάθε μία έχει τη δική της τιμή και το πρόγραμμα εμφανίζει: I_test = 200 και I_main = 10. Τι θα συνέβαινε αν δεν δηλώναμε την i σαν int στην test() και γράφαμε i = 200;

Αφού οι δύο μεταβλητές i είναι ασυσχέτιστες μεταξύ τους, ο μεταγλωττιστής θα εμφάνιζε μήνυμα λάθους ότι η μεταβλητή i στην test() δεν έχει δηλωθεί.

Δήλωση Μεταβλητών μέσα σε Τμήμα

- Επιτρέπεται να δηλώσουμε μεταβλητές μετά το { μίας σύνθετης εντολής (π.χ. if)
- Η εμβέλεια μίας μεταβλητής που δηλώνεται σε ένα τμήμα εκτείνεται μέχρι το αντίστοιχο }, άρα δεν σχετίζεται με τυχόν ομώνυμες μεταβλητές έξω από το τμήμα
- Ο μεταγλωττιστής δεσμεύει μνήμη για αυτήν όταν εισέρχεται στο τμήμα και την αποδεσμεύει όταν εξέρχεται. Π.χ.

```
#include <iostream>
int main()
{
  int i = 20, num = 10;

  if(num == 10)
  {
        int i; // Δέσμευση μνήμης για τη νέα i.
        i = 50;
  } // Αποδέσμευση μνήμης.
  std::cout << i << '\n';
  return 0;
}
```

Επειδή η i που δηλώνεται στην if εντολή δεν έχει καμία σχέση με την πρώτη i, το πρόγραμμα εμφανίζει 20. Ένας λόγος για να δηλώσουμε μεταβλητές μέσα σε ένα τμήμα και όχι στην αρχή της συνάρτησης είναι ότι θα δεσμευτεί μνήμη για αυτές μόνο αν χρειαστεί. Αυτό μπορεί να είναι χρήσιμο για ανάπτυξη εφαρμογών σε συστήματα με περιορισμένη μνήμη

Στατικές Μεταβλητές

- Όπως συζητήσαμε, η μνήμη που δεσμεύεται για μία τοπική μεταβλητή αποδεσμεύεται όταν τελειώσει η εκτέλεση της συνάρτησης
- Επομένως, αν κληθεί πάλι η συνάρτηση, δεν υπάρχει καμία εγγύηση ότι θα έχει διατηρήσει την τιμή της
- Αν θέλουμε μία τοπική μεταβλητή να διατηρεί την τιμή της, πρέπει να δηλωθεί με την λέξη `static`
- Η μνήμη για μία `static` μεταβλητή δεσμεύεται μόνο την πρώτη φορά που θα κληθεί η συνάρτηση. Αυτή η μνήμη δεν αποδεσμεύεται όταν τερματιστεί η συνάρτηση, αλλά όταν τερματιστεί το πρόγραμμα
- Άρα, μία στατική μεταβλητή διατηρεί την τελευταία τιμή της
- Μία στατική μεταβλητή αρχικοποιείται μόνο την πρώτη φορά που καλείται η συνάρτηση. Αν δεν της ανατεθεί κάποια τιμή, αρχικοποιείται με 0. Στις επόμενες κλήσεις της συνάρτησης, η μεταβλητή διατηρεί την τελευταία τιμή της και δεν αρχικοποιείται πάλι

Παράδειγμα

- Ποια είναι η έξοδος του παρακάτω προγράμματος;

```
#include <iostream>

void test();

int main()
{
  test();
  test();
  test();
  return 0;
}

void test()
{
  static int i = 100, arr[1];
  int j = 0;

  i++;
  arr[0] += 2;
  j++;
  std::cout << i << ' ' << arr[0] << ' ' << j << '\n';
}
```

Παράδειγμα

- Απάντηση. Στην πρώτη κλήση της `test()` η τιμή του i γίνεται 101. Αφού το i έχει δηλωθεί σαν στατική μεταβλητή, η τιμή του διατηρείται και με την επόμενη κλήση της `test()` γίνεται 102. Παρομοίως, με την τρίτη κλήση γίνεται 103. Αφού τα στοιχεία ενός στατικού πίνακα αρχικοποιούνται εξ'ορισμού με 0, η τιμή του arr[0] γίνεται 2, 4 και 6, αντίστοιχα. Αντίθετα, η j είναι μία αυτόματη μεταβλητή που δεν διατηρεί την τιμή της ανάμεσα στις κλήσεις της `test()`. Επομένως, το πρόγραμμα εμφανίζει:

```
101 2 1
102 4 1
103 6 1
```

Παράδειγμα

- Αφού η μνήμη μίας απλής τοπικής μεταβλητής αποδεσμεύεται, μία συνάρτηση **δεν πρέπει** να επιστρέφει τη διεύθυνσή της, π.χ.

```
#include <iostream>

int *test();

int main()
{
  int *ptr, j;

  ptr = test();
  std::cout << *ptr << '\n';

  j = *ptr;
  std::cout << j << '\n';
  return 0;
}

int *test()
{
  int i = 10;
  return &i; /* Σοβαρό λάθος, μην το
  κάνετε. */
}
```

Όταν καλείται η **test()**, ο μεταγλωττιστής δεσμεύει μνήμη για την τοπική μεταβλητή **i**. Η εντολή **return &i;** επιστρέφει τη διεύθυνση μνήμης της.

Όμως, αυτή η μνήμη αποδεσμεύεται όταν τερματίζει η εκτέλεση της συνάρτησης.

Επομένως, μπορεί να αποθηκευτούν νέα δεδομένα σε αυτή τη μνήμη και να χαθεί η τιμή 10.

Άρα, το πρόγραμμα αντί για τις τιμές 10 και 10 μπορεί να εμφανίσει 10 και μία άλλη τυχαία τιμή ή ακόμα και δύο τυχαίες τιμές.

Η συμπεριφορά του προγράμματος είναι **απρόβλεπτη**

Παρατηρήσεις

- Αν, στο προηγούμενο παράδειγμα, η τοπική μεταβλητή `i` είχε δηλωθεί ως `static`, η συνάρτηση θα μπορούσε να επιστρέφει τη διεύθυνση της `i`, δεδομένου ότι η μνήμη που έχει δεσμευτεί για μία `static` μεταβλητή δεν αποδεσμεύεται όταν τερματιστεί η συνάρτηση

- **Προσοχή Λοιπόν!!!**

  Μην επιστρέφετε τη διεύθυνση μίας τοπικής μεταβλητής, εκτός αν έχει δηλωθεί ως `static`

Καθολικές Μεταβλητές

- Μία μεταβλητή που δηλώνεται έξω από οποιαδήποτε συνάρτηση και τύπους με εμβέλεια τμήματος, όπως απαρίθμηση, κλάση και χώρος ονομάτων, ονομάζεται **καθολική (global)** μεταβλητή

- Η εμβέλειά της εκτείνεται από το σημείο της δήλωσής της μέχρι το τέλος του αρχείου στο οποίο δηλώνεται. Επομένως, όλες οι συναρτήσεις που ορίζονται μετά από το σημείο δήλωσής της έχουν πρόσβαση σε αυτήν

- Μία καθολική μεταβλητή έχει μόνιμη διάρκεια ζωής και διατηρεί την τελευταία τιμή που της εκχωρήθηκε. Δεσμεύεται μνήμη για αυτήν όταν αρχίζει το πρόγραμμα και αποδεσμεύεται όταν τερματίσει η εκτέλεσή του

Παρατηρήσεις

- Όταν η ίδια μεταβλητή χρησιμοποιείται σε πολλές συναρτήσεις, πολλοί προγραμματιστές συνηθίζουν να τη δηλώνουν σαν καθολική, αντί να τη μεταβιβάζουν σαν όρισμα στις κλήσεις των συναρτήσεων
- Ουσιαστικά, μία καθολική μεταβλητή μπορεί να χρησιμοποιηθεί σαν εναλλακτικός τρόπος για τη μεταβίβαση πληροφορίας σε μία συνάρτηση
- Για καλύτερο έλεγχο και κατανόηση του προγράμματος, είναι πολύ βοηθητικό το όνομα που θα επιλέξετε για μία καθολική μεταβλητή να περιγράφει τον ρόλο της. Για παράδειγμα, μην χρησιμοποιείτε ονόματα τα οποία συνήθως δίνονται σε τοπικές μεταβλητές (π.χ. `i`)
- Εξ'ορισμού, ο μεταγλωττιστής αρχικοποιεί μία καθολική μεταβλητή ή τα μέλη ενός σύνθετου τύπου (π.χ. πίνακας) με 0
- Αν και η χρήση καθολικών μεταβλητών μπορεί να φαίνεται ότι διευκολύνει την δουλειά του προγραμματιστή, μάλλον είναι καλύτερα να αποφεύγετε τη χρήση τους ή τουλάχιστον τη συχνή χρήση τους. Ένας λόγος είναι ότι δυσκολεύει η ανίχνευση προβληματικών καταστάσεων. Για παράδειγμα, όταν πολλές συναρτήσεις έχουν πρόσβαση στην ίδια μεταβλητή και κάποια από αυτές της εκχωρήσει μία ανεπιθύμητη τιμή, πρέπει να βρούμε και να ελέγξουμε όλες τις συναρτήσεις που την χρησιμοποιούν για να εντοπιστεί η «ένοχη» συνάρτηση

Παράδειγμα

- Ποια είναι η έξοδος του παρακάτω προγράμματος;

```
#include <iostream>

void add();
void sub();

int glob = 0;/* Αν και ο μεταγλωττιστής την αρχικοποιεί με 0, προτιμώ να το κάνω ξεκάθαρα. */

int main()
{
   add();
   glob += 2;
   sub();
   std::cout << glob << '\n';
   return 0;
}

void add()
{
   glob++;
}

void sub()
{
   glob--;
}
```

- Όλες οι συναρτήσεις έχουν πρόσβαση στην καθολική μεταβλητή **glob**, αφού η δήλωσή της γίνεται πριν από τους ορισμούς τους. Το πρόγραμμα εμφανίζει 2

Παρατηρήσεις

- Μία τοπική μεταβλητή μπορεί να έχει το ίδιο όνομα με μία καθολική. Ο κανόνας εμβέλειας λέει ότι όταν μία μεταβλητή σε ένα τμήμα κώδικα ονοματίζεται το ίδιο με μία άλλη που είναι ήδη ορατή, η νέα δήλωση «κρύβει» την παλιά στο συγκεκριμένο τμήμα

- Π.χ. στο παρακάτω πρόγραμμα η τοπική μεταβλητή a που δηλώνεται στην test() είναι διαφορετική από την καθολική μεταβλητή a. Το πρόγραμμα εμφανίζει 100

```
#include <iostream>

void test();

int a = 100;

int main()
{
  test();
  std::cout << a << '\n';
  return 0;
}
void test()
{
  int a;
  a = 2000;
}
```

Εξωτερικές Καθολικές Μεταβλητές (extern)

- Μία καθολική μεταβλητή μπορεί να γίνει ορατή σε περισσότερα από ένα αρχεία
- Αυτό είναι πολύ χρήσιμο, ιδιαίτερα στις περιπτώσεις μεγάλων προγραμμάτων, που είναι πολύ πιθανό διαφορετικά αρχεία να χρειάζεται να χρησιμοποιήσουν την ίδια μεταβλητή
- Η λέξη **extern** είναι ένα προσδιοριστικό κατηγορίας αποθήκευσης, η οποία επιτρέπει μία μεταβλητή να μπορεί να γίνει ορατή σε διαφορετικά αρχεία
- Συνήθως, μία τέτοια μεταβλητή δηλώνεται σαν καθολική στο αρχείο που χρησιμοποιείται περισσότερο και στα υπόλοιπα αναφέρεται ως **extern**
- Π.χ. η εντολή: `extern int size;`

  ενημερώνει τον μεταγλωττιστή ότι η ακέραια μεταβλητή με όνομα **size** δεν ορίζεται σε αυτό το αρχείο, αλλά σε κάποιο άλλο αρχείο του προγράμματος
- Η **size** μπορεί να χρησιμοποιηθεί και, αν χρειαστεί, να τροποποιηθεί η τιμή της σε οποιοδήποτε από τα αρχεία στα οποία εμφανίζεται σαν **extern**

Συνάρτηση με Παράμετρο Πίνακα

- Όταν η παράμετρος μίας συνάρτησης είναι ένας μονοδιάστατος πίνακας, γράφουμε το όνομα του πίνακα ακολουθούμενο από κενές αγκύλες. Π.χ. `void test(int arr[]);`
- Όταν μεταβιβάζεται ένας πίνακας σε μία συνάρτηση, γράφουμε μόνο το όνομα του πίνακα, χωρίς τις αγκύλες. Π.χ. `test(arr);`
- Όταν το όνομα ενός πίνακα μεταβιβάζεται σε μία συνάρτηση, χρησιμοποιείται σαν δείκτης. Ουσιαστικά, μεταβιβάζεται ένας δείκτης στο πρώτο του στοιχείο και όχι ένα αντίγραφο του πίνακα. Επομένως, αφού δεν γίνεται αντιγραφή του πίνακα, ο χρόνος που απαιτείται για να μεταβιβαστεί ένας πίνακας σε μία συνάρτηση δεν εξαρτάται από το μέγεθός του
- Να θυμόμαστε λοιπόν, όταν μία απλή μεταβλητή μεταβιβάζεται σε μία συνάρτηση, η συνάρτηση εργάζεται με αντίγραφό της. Όμως, όταν μεταβιβάζεται πίνακας, δεν δημιουργείται αντίγραφό του, η συνάρτηση εργάζεται με τον πρωτότυπο πίνακα
- Και γιατί λήφθηκε αυτή η απόφαση; Για καλύτερη απόδοση, για να γλιτώσουμε την μνήμη και τον χρόνο που απαιτείται για την αντιγραφή όλων των στοιχείων του. Απλά, μεταβιβάζεται ο δείκτης στο πρώτο του στοιχείο και η συνάρτηση μπορεί να προσπελάσει όποιο στοιχείο επιθυμεί

Παράδειγμα

- Ποια είναι η έξοδος του παρακάτω προγράμματος;

```
#include <iostream>

void test(int arr[]);

int main()
{
  int i, a[5] = {10, 20, 30, 40, 50};

  test(a); // Ισοδύναμο με test(&a[0]);
  for(i = 0; i < 5; i++)
       std::cout << a[i] << ' ';
  return 0;
}

void test(int arr[])
{
  arr[0] = arr[1] = 0;
}
```

Παράδειγμα

- Απάντηση. Σημειώστε ότι στην `test()` θα μπορούσαμε να χρησιμοποιήσουμε σαν όνομα παραμέτρου το όνομα `a` αντί για `arr`, αφού, ήδη γνωρίζετε ότι, οι τοπικές μεταβλητές διαφορετικών συναρτήσεων δεν σχετίζονται μεταξύ τους, ακόμα και αν έχουν το ίδιο όνομα. Όταν καλείται η `test()` έχουμε `arr = a = &a[0]`. Επομένως, οι εντολές `arr[0] = 0;` και `arr[1] = 0;` αλλάζουν τις τιμές των δύο πρώτων στοιχείων του πίνακα και το πρόγραμμα εμφανίζει: `0 0 30 40 50`

Παρατηρήσεις (1)

- Μία παράμετρος πίνακας μπορεί επίσης να δηλωθεί σαν δείκτης. Για παράδειγμα, οι δηλώσεις:

  `void test(int arr[]);` και `void test(int *arr);`

  είναι ισοδύναμες. Ο μεταγλωττιστής τις χειρίζεται με τον ίδιο τρόπο και μεταβιβάζει στη συνάρτηση τον δείκτη. Άρα, η δεύτερη δήλωση είναι πιο ακριβής, αφού δηλώνει ξεκάθαρα ότι στη συνάρτηση μεταβιβάζεται δείκτης και όχι αντίγραφο του πίνακα. Σαν προτίμηση, προτιμώ τον πρώτο τρόπο, ώστε να φαίνεται ξεκάθαρα η πρόθεση του προγραμματιστή να μεταβιβάσει στη συνάρτηση πίνακα. Με τη δεύτερη δήλωση ο αναγνώστης του κώδικα δεν μπορεί να καταλάβει αν η συνάρτηση θα δεχτεί ένα πλήθος τιμών ή έναν δείκτη σε μία μόνο τιμή

- Για ασφάλεια, αν δεν θέλουμε μία συνάρτηση να μπορεί να αλλάξει τις τιμές των στοιχείων του πίνακα, δηλώνουμε την παράμετρο σαν `const`. Π.χ. με την παρακάτω δήλωση η συνάρτηση `test()` μπορεί να προσπελάσει τα στοιχεία του πίνακα `arr`, αλλά δεν μπορεί να αλλάξει τις τιμές τους

```
void test(const int arr[]);
```

Παρατηρήσεις (2)

- Εναλλακτικά, θα μπορούσαμε να χρησιμοποιήσουμε σημειογραφία δείκτη για να προσπελάσουμε τα στοιχεία του πίνακα. Π.χ:

```
void test(int arr[])
{
  *arr = 0; // Ισοδύναμο με arr[0] = 0.
  arr++;
  *arr = 0;
}
```

- Αυτό που έχει ενδιαφέρον είναι η εντολή **arr++**. Μπορούμε να αλλάξουμε την τιμή μίας μεταβλητής πίνακα; Όχι βέβαια, όπως έχουμε πει στο Κ.8, όταν το όνομα ενός πίνακα χρησιμοποιείται σαν δείκτης είναι **const** δείκτης. Επομένως, γιατί ο μεταγλωττιστής επιτρέπει την παραπάνω εντολή; Γιατί, όπως είπαμε, η παράμετρος **arr** μπορεί να δηλώνεται σαν πίνακας, αλλά στην πραγματικότητα είναι δείκτης. Άρα, μπορούμε να του εκχωρήσουμε μία νέα τιμή

Προσδιορισμός Τέλους Πίνακα (1)

- Επειδή λοιπόν στη συνάρτηση μεταβιβάζεται ο δείκτης, δεν παίζει κανένα ρόλο αν μέσα στις αγκύλες προσθέσουμε το μήκος του πίνακα

  (π.χ. `void test(int arr[5])`)
- Ο μεταγλωττιστής το αγνοεί, δεν πρόκειται να ελέγξει αν ο πίνακας έχει πράγματι το μήκος που δηλώνεται (π.χ. 5)
- Στην πραγματικότητα, αυτό που μεταβιβάζεται είναι ένας δείκτης στο πρώτο στοιχείο ενός πίνακα αγνώστου μήκους
- Αυτός είναι ο λόγος που οι αγκύλες συνηθίζεται να είναι κενές, ώστε να μην δημιουργείται η εσφαλμένη εντύπωση ότι ο μεταγλωττιστής απαιτεί ο πίνακας που μεταβιβάζεται να έχει ένα συγκεκριμένο πλήθος στοιχείων. Όχι, ο μεταγλωττιστής μεταγλωττίζει το πρόγραμμα ανεξάρτητα από το μήκος του πίνακα

Προσδιορισμός Τέλους Πίνακα (2)

- Ένας εύκολος τρόπος για να μάθει η συνάρτηση τον αριθμό των στοιχείων του πίνακα είναι να το μεταβιβάσουμε σαν όρισμα. Π.χ:

  ```
  void test(int arr[], int size);
  ```

  και για να την καλέσουμε να γράψουμε: `test(a, 5);`
- Εναλλακτικά, μπορούμε να δηλώσουμε μία σταθερά όπως,

  `const int SIZE = 5;` και να τη χρησιμοποιούμε όπου χρειάζεται, αντί να μεταβιβάζουμε ένα πρόσθετο όρισμα
- Σημειώστε ότι αν και μπορούμε να χρησιμοποιήσουμε τον τελεστή `sizeof` για να υπολογίσουμε το μέγεθος ενός πίνακα, δεν μπορούμε να τον χρησιμοποιήσουμε για να υπολογίσουμε το μέγεθος ενός πίνακα που δηλώνεται σαν παράμετρος σε μία συνάρτηση. Αφού, όπως είπαμε, στη συνάρτηση μεταβιβάζεται δείκτης, ο τελεστής `sizeof` υπολογίζει το μέγεθος της μνήμης που δεσμεύει μία μεταβλητή δείκτης (π.χ. 4) και όχι τη μνήμη που δεσμεύει ο πίνακας

Μεταβίβαση Τμήματος Πίνακα

- Σε μία συνάρτηση μπορούμε να μεταβιβάσουμε ένα τμήμα του πίνακα. Για παράδειγμα, ποια είναι η έξοδος του παρακάτω προγράμματος;

```
#include <iostream>

void test(int ptr[]);

int main()
{
  int i, arr[6] = {1, 2, 3, 4, 5, 6};

  test(arr+3); // Εναλλακτικά, test(&arr[3]).
  for(i = 0; i < 6; i++)
        std::cout << arr[i] << ' ';
  return 0;
}

void test(int ptr[])
{
  int i, tmp[3] = {10, 20, 30};

  for(i = 0; i < 3; i++)
        ptr[i] = tmp[i];

  *ptr = *(ptr-1);
}
```

Μεταβίβαση Τμήματος Πίνακα

- Απάντηση. Με την κλήση της test() έχουμε ptr = arr+3, δηλαδή, μεταβιβάζουμε στην test() το τμήμα του πίνακα arr από το τέταρτο στοιχείο και μετά. Αφού χρησιμοποιούμε τον δείκτη ptr σαν πίνακα, το ptr[0] αντιστοιχεί στο arr[3], το ptr[1] στο arr[4] και το ptr[2] στο arr[5]. Επομένως, με τον βρόχο, οι τιμές των arr[3], arr[4] και arr[5] γίνονται 10, 20 και 30, αντίστοιχα. Αφού ο ptr δείχνει στο arr[3], η εντολή *ptr = *(ptr-1); είναι ισοδύναμη με arr[3] = arr[2]; Άρα, το πρόγραμμα εμφανίζει: 1 2 3 3 20 30

Μεταβίβαση `vector` Αντικειμένου

- Και βέβαια, σε μία συνάρτηση μπορούμε να μεταβιβάσουμε ένα `vector` αντικείμενο. Για παράδειγμα, δείτε την παρακάτω άσκηση.

  Δημιουργήστε μία συνάρτηση που να δέχεται σαν παραμέτρους ένα `vector` αντικείμενο με βαθμούς φοιτητών και δύο βαθμούς (π.χ. `a` και `b`) και να επιστρέφει τον μέσο όρο των βαθμών που ανήκουν στο `[a, b]`. Να γραφεί ένα πρόγραμμα το οποίο να διαβάζει τον αριθμό των φοιτητών, να δημιουργεί ένα `vector` αντικείμενο, να διαβάζει τους βαθμούς τους και να τους καταχωρεί σε αυτό. Μετά, να διαβάζει τους βαθμούς `a` και `b`, να καλεί την συνάρτηση και να εμφανίζει τον μέσο όρο. Το πρόγραμμα να υποχρεώνει τον χρήστη η τιμή του `a` να είναι μικρότερη ή ίση του `b`.

Παράδειγμα (1)

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

float avg_arr(const vector<float>& v, float min, float max);

int main()
{
   int i, num;
   float a, b, k;

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

   vector<float> grd_v(num);
   for(i = 0; i < num; i++)
   {
          cout << "Enter grade: ";
          cin >> grd_v[i];
   }
   do
   {
          cout << "Enter min and max grades: ";
          cin >> a >> b;
   } while(a > b);

   k = avg_arr(grd_v, a, b);
   if(k == -1)
          cout << "None grade in the indicated set\n";
   else
          cout << "Avg = " << k << '\n';
   return 0;
}
```

Παράδειγμα (1)

```
float avg_arr(const vector<float>& v, float min, float max)
{
  int i, cnt = 0;
  float sum = 0;

  for(i = 0; i < v.size(); i++)
  {
        if(v[i] >= min && v[i] <= max)
        {
              cnt++;
              sum += v[i];
        }
  }
  if(cnt == 0)
        return -1; // Ειδική τιμή.
  else
        return sum/cnt;
}
```

Παράδειγμα (2)

- Δημιουργήστε μία συνάρτηση η οποία να δέχεται σαν παραμέτρους δύο αλφαριθμητικά και με χρήση δεικτών να επιστρέφει 0 αν είναι ίδια ή τη διαφορά των πρώτων χαρακτήρων τους που δεν είναι ίδιοι. Να γραφεί ένα πρόγραμμα το οποίο να διαβάζει δύο αλφαριθμητικά μέχρι 100 χαρακτήρες και να εμφανίζει το αποτέλεσμα της σύγκρισής τους με χρήση της συνάρτησης

Παράδειγμα (2)

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

int str_cmp(const char *str1, const char *str2);

int main()
{
   char str1[100], str2[100];
   int i;

   cout << "Enter first text: ";
   cin.getline(str1, sizeof(str1));

   cout << "Enter second text: ";
   cin.getline(str2, sizeof(str2));

   i = str_cmp(str1, str2);
   if(i == 0)
         cout << str1 << " = " << str2 << '\n';
   else if(i < 0)
         cout << str1 << " < " << str2 << '\n';
   else
         cout << str1 << " > " << str2 << '\n';
   return 0;
}
int str_cmp(const char *str1, const char *str2)
{
   while(*s1 == *s2)
   {
         if(*s1 == '\0')
                  return 0;
         s1++;
         s2++;
   }
   return *s1-*s2;
}
```

Παράδειγμα (3)

- Δημιουργήστε μία συνάρτηση που να δέχεται σαν παράμετρο έναν πίνακα και να ελέγχει αν περιέχει τιμές που να επαναλαμβάνονται. Αν ναι, η συνάρτηση να επιστρέφει έναν δείκτη στο στοιχείο που επαναλαμβάνεται τις περισσότερες φορές, αλλιώς, να επιστρέφει `nullptr`. Να γραφεί ένα πρόγραμμα το οποίο να διαβάζει 100 ακεραίους, να τους αποθηκεύει σε έναν πίνακα, να καλεί τη συνάρτηση και να χρησιμοποιεί την τιμή επιστροφής της για να εμφανίσει την τιμή του στοιχείου με τις περισσότερες εμφανίσεις. (Σημ. Αν υπάρχουν παραπάνω από ένα στοιχεία με τον ίδιο αριθμό μέγιστων εμφανίσεων, το πρόγραμμα να εμφανίζει το πρώτο από αυτά)

Παράδειγμα (3)

```
#include <iostream>

int *find(int arr[]);
const int SIZE = 100;

int main()
{
  int *ptr, i, arr[SIZE];

  for(i = 0; i < SIZE; i++)
  {
        std::cout << "Enter number: ";
        std::cin >> arr[i];
  }
  ptr = find(arr);
  if(ptr == nullptr)
        std::cout << "No duplicated
  value is found\n";
  else
        std::cout << "The number " <<
  *ptr <<  " appears the most times\n";
  return 0;
}
```

```
int *find(int arr[])
{
  int i, j, cnt, max, pos;

  max = 0;
  for(i = 0; i < SIZE; i++)
  {
        cnt = 0;
        for(j = i+1; j < SIZE; j++)
        {
              if(arr[i] == arr[j])
                    cnt++;
        }
        if(cnt > max)
        {
              max = cnt;
              pos = i;
        }
  }
  if(max == 0)
        return nullptr;
  else
        return arr+pos;
}
```

Δήλωση Συνάρτησης με Παράμετρο Διδιάστατο Πίνακα

- Ο πιο συνηθισμένος τρόπος για να δηλώσουμε μία συνάρτηση που να δέχεται σαν παράμετρο έναν διδιάστατο πίνακα είναι να γράψουμε το όνομα του πίνακα ακολουθούμενο από τις διαστάσεις του μέσα σε αγκύλες
- Για παράδειγμα, η `test()` δέχεται σαν παράμετρο έναν διδιάστατο πίνακα ακεραίων με 5 γραμμές και 10 στήλες:

```
void test(int arr[5][10]);
```

- Επειδή, όπως είδαμε στο Κ.7, ο μεταγλωττιστής δεν χρειάζεται να γνωρίζει τον αριθμό των γραμμών για να υπολογίσει τη θέση μνήμης ενός στοιχείου, μπορούμε να παραλείψουμε την τιμή της πρώτης διάστασης. Αν την δηλώσετε, ο μεταγλωττιστής θα την αγνοήσει. Π.χ: `void test(int arr[][10]);`
- Πρέπει, όμως, να δηλώσουμε τον αριθμό των στηλών. Στη γενική περίπτωση, όταν μεταβιβάζεται ένας πολυδιάστατος πίνακας σε μία συνάρτηση η πρώτη διάσταση μπορεί να παραλειφθεί. Οι άλλες πρέπει να δηλωθούν και να είναι ίδιες με αυτές στη δήλωση του πίνακα
- Όπως και στην περίπτωση του μονοδιάστατου πίνακα, για να μεταβιβάσουμε έναν διδιάστατο πίνακα σαν όρισμα συνάρτησης, γράφουμε μόνο το όνομα του πίνακα, χωρίς τις αγκύλες

Παρατηρήσεις (1)

- Αφού, όπως είπαμε, η C++ χειρίζεται έναν διδιάστατο πίνακα σαν έναν πίνακα μονοδιάστατων πινάκων, στην πραγματικότητα, ο μεταγλωττιστής μεταφράζει τον διδιάστατο πίνακα σε έναν «δείκτη σε πίνακα». Επομένως, θα μπορούσαμε ισοδύναμα να γράψουμε:

```
void test(int (*arr)[10]);
```

- Επειδή οι αγκύλες [] έχουν μεγαλύτερη προτεραιότητα από το *, οι παρενθέσεις είναι απαραίτητες (αλλιώς, ο arr θα μεταφραστεί ως πίνακας 10 δεικτών σε ακεραίους αντί για δείκτης σε πίνακα 10 ακεραίων)

- Αν και αυτή η δήλωση είναι πιο ακριβής, προτιμώ την προηγούμενη δήλωση και να παραλείπω την πρώτη διάσταση, ώστε να φαίνεται ξεκάθαρα ότι η παράμετρος είναι διδιάστατος πίνακας

- Το γράφω πάλι, ο μεταγλωττιστής μεταφράζει τον διδιάστατο πίνακα σε δείκτη σε πίνακα, όχι σε δείκτη σε δείκτη, όπως ίσως να νομίζατε. Δηλαδή, είναι λάθος να γράψετε:

```
void test(int **arr);
```

Παρατηρήσεις (2)

- Ας συνοψίσουμε πώς μεταφράζει ο μεταγλωττιστής ορίσματα που είναι πίνακες:

Μονοδιάστατος: `arr[10]` μεταφράζεται σε δείκτη `*arr`

Διδιάστατος: `arr[][20]` μεταφράζεται σε δείκτη σε πίνακα `(*arr)[20]`

Πίνακας δεικτών: `*arr[50]` μεταφράζεται σε δείκτη σε δείκτη `**arr`

Παράδειγμα

- Δημιουργήστε μία συνάρτηση που να δέχεται σαν παράμετρο έναν 3×4 διδιάστατο πίνακα και να επιστρέφει έναν μονοδιάστατο πίνακα όπου το κάθε στοιχείο του θα είναι ίσο με το άθροισμα των στοιχείων της αντίστοιχης γραμμής του διδιάστατου πίνακα, καθώς και έναν δεύτερο μονοδιάστατο πίνακα όπου το κάθε στοιχείο του θα είναι ίσο με το άθροισμα των στοιχείων της αντίστοιχης στήλης του διδιάστατου πίνακα. Να γραφεί ένα πρόγραμμα το οποίο να διαβάζει 12 ακεραίους, να τους αποθηκεύει σε έναν διδιάστατο πίνακα 3×4 και να εμφανίζει το άθροισμα των στοιχείων της κάθε γραμμής και στήλης του με χρήση της συνάρτησης

Παράδειγμα

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

const int ROWS = 3;
const int COLS = 4;

void find_sums(int arr1[][COLS], int arr2[], int arr3[]);

int main()
{
  int i, j, arr1[ROWS][COLS], arr2[ROWS], arr3[COLS];

  for(i = 0; i < ROWS; i++)
        for(j = 0; j < COLS; j++)
        {
                cout << "arr1[" << i << "][" << j << "] = ";
                cin >> arr1[i][j];
        }

  find_sums(arr1, arr2, arr3);
  for(i = 0; i < ROWS; i++)
        cout << "sum_line_" << i << " = " << arr2[i] << '\n';
  for(i = 0; i < COLS; i++)
        cout << "sum_col_" << i << " = " << arr3[i] << '\n';
  return 0;
}
```

Παράδειγμα

```
void find_sums(int arr1[][COLS], int arr2[], int arr3[])
{
  int i, j, sum;

  for(i = 0; i < ROWS; i++)
  {
        sum = 0;
        for(j = 0; j < COLS; j++)
              sum += arr1[i][j];
        arr2[i] = sum;
  }
  for(i = 0; i < COLS; i++)
  {
        sum = 0;
        for(j = 0; j < ROWS; j++)
              sum += arr1[j][i];
        arr3[i] = sum;
  }
}
```

Σχόλια: Αφού μία συνάρτηση δεν μπορεί να επιστρέψει πίνακα, οι πίνακες δηλώνονται στη **main()** και μεταβιβάζονται στη συνάρτηση

Διοχέτευση Πληροφορίας στη Συνάρτηση `main()` (1)

- Όταν εκτελούμε ένα πρόγραμμα από τη γραμμή εντολών (command line) μπορούμε να του μεταβιβάσουμε πληροφορία
- Για παράδειγμα, ας υποθέσουμε ότι το εκτελέσιμο αρχείο `hello.exe` είναι αποθηκευμένο στον φάκελο `C:\programs`. Τότε, αν γράψουμε στη γραμμή εντολών: `C:\programs>hello 100 200` θα εκτελεστεί το πρόγραμμα `hello` και στη `main()` θα διοχετευθούν οι τιμές `100` και `200`

Διοχέτευση Πληροφορίας στη Συνάρτηση `main()` (2)

- Όμως, για να είναι σε θέση η συνάρτηση `main()` ενός προγράμματος να πάρει τις τιμές των παραμέτρων από τη γραμμή εντολών θα πρέπει να έχει δηλωθεί ως:

```
int main(int argc, char *argv[])
```

Αν και μπορείτε να χρησιμοποιήσετε όποια ονόματα επιθυμείτε, η τυπική επιλογή είναι τα `argc` (*argument count*) και `argv` (*argument vector*). Και επειδή, όπως είπαμε, ο μεταγλωττιστής μεταφράζει μία παράμετρο πίνακα σε δείκτη πολλοί προτιμούν, αντί για `*argv[]`, να γράφουν `**argv`

α) Η παράμετρος `argc` είναι ένας ακέραιος που δηλώνει το πλήθος των ορισμάτων της γραμμής εντολών, συμπεριλαμβανομένου του ονόματος του προγράμματος. Για παράδειγμα, στην προηγούμενη γραμμή εντολών, η τιμή του `argc` είναι ίση με 3. Τα ορίσματα πρέπει να διαχωρίζονται μεταξύ τους με κενά, ώστε να ξεχωρίζουν οι τιμές

Διοχέτευση Πληροφορίας στη Συνάρτηση `main()` (3)

β) Η παράμετρος `argv` είναι ένας πίνακας δεικτών στα ορίσματα της γραμμής εντολών, τα οποία αποθηκεύονται σαν αλφαριθμητικά C-μορφής

Ο δείκτης `argv[0]` δείχνει στο όνομα του προγράμματος, ενώ οι υπόλοιποι δείκτες μέχρι τον `argv[argc-1]` δείχνουν στα υπόλοιπα ορίσματα. Το τελευταίο στοιχείο του πίνακα `argv` είναι το `argv[argc]`, του οποίου η τιμή είναι πάντα ίση με `nullptr`

Στο παράδειγμά μας, τα ορίσματα `hello`, `100` και `200` μεταβιβάζονται στην `main()` σαν αλφαριθμητικά. Ο δείκτης `argv[0]` δείχνει στο αλφαριθμητικό `"hello"`, ο `argv[1]` στο αλφαριθμητικό `"100"` και ο `argv[2]` στο `"200"`. Η τιμή του `argv[3]` είναι ίση με `nullptr`

Παράδειγμα

- Το επόμενο πρόγραμμα ελέγχει αν ο χρήστης εισήγαγε το σωστό όνομα χρήστη και συνθηματικό στη γραμμή εντολών. Θεωρήστε ότι αυτά είναι τα **user** και **pswd**, αντίστοιχα

```
#include <iostream>
#include <cstring>

int main(int argc, char *argv[])
{
  if(argc == 1)
        std::cout << "Error: missing user name and password\n";
  else if(argc == 2)
        std::cout << "Error: missing password\n";
  else if(argc == 3)
  {
        if(strcmp(argv[1], "user") == 0 && strcmp(argv[2], "pswd") == 0)
                std::cout << "Valid user. The program " << argv[0] << " will
  be executed ...\n";
        else
                std::cout << "Wrong input\n";
  }
  else
        std::cout << "Error: too many parameters\n";
  return 0;
}
```

- Οι **if** συνθήκες ελέγχουν την τιμή της παραμέτρου **argc**. Αν η τιμή της είναι 3, το πρόγραμμα ελέγχει αν ο χρήστης εισήγαγε έγκυρο όνομα χρήστη και συνθηματικό. Αν δεν είναι 3, το πρόγραμμα εμφανίζει ανάλογο μήνυμα

Συναρτήσεις με Μεταβλητό Αριθμό Παραμέτρων

- Μία συνάρτηση μπορεί να δεχτεί μεταβλητό αριθμό παραμέτρων
- Ένας τρόπος δήλωσης μίας τέτοιας συνάρτησης, που προέρχεται από τη C, είναι να γράψουμε πρώτα τις σταθερές παραμέτρους, δηλαδή, αυτές που πρέπει να υπάρχουν πάντα στην κλήση της συνάρτησης, και στο τέλος τα αποσιωπητικά ...
- Π.χ. η συνάρτηση: `void test(int num, char *str, ...);`
  είναι μία συνάρτηση που δέχεται δύο σταθερές παραμέτρους (μία ακέραια μεταβλητή και έναν δείκτη σε χαρακτήρα) και μπορεί να ακολουθήσει ένας μεταβλητός αριθμός παραμέτρων
- Μία συνάρτηση που δέχεται έναν μεταβλητό αριθμό παραμέτρων πρέπει να έχει τουλάχιστον μία σταθερή παράμετρο
- Για να καλέσουμε μία τέτοια συνάρτηση, πρώτα γράφουμε τις τιμές των σταθερών ορισμάτων και μετά τις τιμές των προαιρετικών ορισμάτων. Για παράδειγμα, μία κλήση της `test()` θα μπορούσε να είναι:
  `test(3, "keimeno", 5, 8.9, "sample");`
- Οι σταθερές παράμετροι αυτής έχουν τιμές 3 και `"keimeno"` αντίστοιχα
- Οι τύποι δεδομένων των προαιρετικών ορισμάτων αυτής της συνάρτησης είναι `int`, `double` και `char*` με τιμές 5, 8.9 και `"sample"`, αντίστοιχα

Αναδρομικές Συναρτήσεις

- Μία συνάρτηση μπορεί να καλεί μέσα στο σώμα της οποιαδήποτε άλλη συνάρτηση, ακόμα και τον εαυτό της
- Μία συνάρτηση που μέσα στο σώμα της **καλεί τον εαυτό της** ονομάζεται **αναδρομική συνάρτηση**
- Για να εξηγήσουμε πως λειτουργεί μία αναδρομική συνάρτηση θα εξετάσουμε τη συνάρτηση `show()` του επόμενου παραδείγματος
- Η συνάρτηση `show()` βλέπουμε ότι είναι αναδρομική, αφού μέσα στο σώμα της καλεί τον εαυτό της

Παράδειγμα

```
#include <iostream>

void show(int num);

int main()
{
  int i;

  std::cout << "Enter number: ";
  std::cin >> i;
  show(i);
  return 0;
}

void show(int num)
{
  if(num > 1)
        show(num-1);

  std::cout << "val = " << num << '\n';
}
```

Επεξήγηση Παραδείγματος

- Για να δούμε πώς λειτουργεί η αναδρομή, ας υποθέσουμε ότι ο χρήστης εισάγει έναν αριθμό μεγαλύτερο από το 1, π.χ. το 3, ώστε να κληθεί πάλι η **show()**
  - α) Στην πρώτη κλήση της **show()**, αφού num = 3 > 1, η **show()** καλεί τον εαυτό της με όρισμα num-1 = 3-1 = 2 και η cout δεν θα εκτελεστεί. Αφού δεν τερματίζεται η εκτέλεση της **show()**, η μνήμη που δεσμεύτηκε για τη μεταβλητή **num** με τιμή 3, καθώς και πρόσθετη πληροφορία που σχετίζεται με την κλήση της συνάρτησης δεν αποδεσμεύεται
  - β) Στη δεύτερη κλήση της **show()** δεσμεύεται νέα μνήμη για τη **num**. Αφού **num = 2 > 1**, η **show()** καλεί πάλι τον εαυτό της με όρισμα num-1 = 2-1 = 1. Παρόμοια με πριν, η **cout** δεν θα εκτελεστεί και η μνήμη που δεσμεύτηκε για την νέα **num** με τιμή 2 δεν αποδεσμεύεται
  - γ) Όπως και πριν, στην τρίτη κλήση της **show()** δεσμεύεται νέα μνήμη για τη **num**. Τώρα όμως δεν θα γίνει νέα κλήση της **show()**, γιατί η τιμή της **num** δεν είναι μεγαλύτερη από το 1. Άρα, θα εκτελεστεί η **cout** και θα εμφανιστεί val = 1. Επίσης, αποδεσμεύεται η μνήμη για τη συγκεκριμένη **num**
- Στη συνέχεια, όλες οι ανεκτέλεστες **cout** θα εκτελεστούν διαδοχικά, ξεκινώντας από την τελευταία. Σε κάθε τερματισμό της **show()**, η μνήμη που είχε δεσμευτεί για την αντίστοιχη num και γενικότερα για τη συγκεκριμένη κλήση της συνάρτησης αποδεσμεύεται. Επομένως, το πρόγραμμα θα εμφανίσει:

```
val = 1
val = 2
val = 3
```

Παρατηρήσεις (1)

- Μία αναδρομική συνάρτηση πρέπει να περιέχει μία **συνθήκη τερματισμού**, αλλιώς θα εκτελείται συνεχώς με αποτέλεσμα την εξάντληση των πόρων του υπολογιστή. Στο προηγούμενο παράδειγμα, αυτή η συνθήκη ήταν η εντολή: `if(num > 1)`

- Κάθε φορά που μία αναδρομική συνάρτηση καλεί τον εαυτό της δεσμεύεται νέα μνήμη από τη στοίβα για τις αυτόματες μεταβλητές της. Για τις στατικές μεταβλητές δεν δεσμεύεται νέα μνήμη, δηλαδή, όλες οι κλήσεις διαχειρίζονται τις ίδιες στατικές μεταβλητές. Η πληροφορία για το ποιο τμήμα του κώδικα δεν εκτελέστηκε αποθηκεύεται και αυτή στη στοίβα

- Όμως, το μέγεθος της στοίβας μπορεί να μην είναι αρκετά μεγάλο για να αποθηκευτεί η πληροφορία που σχετίζεται με την κάθε κλήση της συνάρτησης. Για παράδειγμα, αν ο χρήστης εισάγει μία μεγάλη τιμή, π.χ. `100000`, είναι πολύ πιθανό να τερματιστεί η εκτέλεση του προγράμματος και να εμφανιστεί το μήνυμα "Stack overflow", που σημαίνει ότι δεν υπάρχει άλλος διαθέσιμος χώρος στη στοίβα

- Να προσέχετε όταν χρησιμοποιείτε μία αναδρομική συνάρτηση, γιατί, αν καλεί πολλές φορές τον εαυτό της, ο χρόνος εκτέλεσής της μπορεί να γίνει υπερβολικά μεγάλος, καθώς και οι διαδοχικές κλήσεις της μπορεί να οδηγήσουν στην εξάντληση της μνήμης της στοίβας

Παρατηρήσεις (2)

- Συνήθως, η αναδρομή χρησιμοποιείται στην ανάπτυξη μαθηματικών αλγορίθμων ή σε λειτουργίες σε δομές δεδομένων
- Γενικά, όταν μία επιμέρους εργασία είναι μία μικρότερη έκδοση της αρχικής εργασίας που πρέπει να πραγματοποιηθεί είναι υποψήφια να υλοποιηθεί με χρήση αναδρομικής συνάρτησης
- Ωστόσο, αν μπορείτε να γράψετε ισοδύναμο κώδικα χωρίς μεγάλη δυσκολία, είναι μάλλον καλύτερη ιδέα. Για παράδειγμα, να χρησιμοποιήσετε στη θέση της κάποιον επαναληπτικό βρόχο
- Αν και ο αναδρομικός κώδικας μπορεί να διαβάζεται και να γράφεται πιο εύκολα, η απόδοση του βρόχου μάλλον θα είναι καλύτερη αφού αποφεύγονται οι διαδοχικές κλήσεις της συνάρτησης και οι δεσμεύσεις μνήμης στη στοίβα

Παράδειγμα (1)

- Ποια είναι η έξοδος του παρακάτω προγράμματος;

```
#include <iostream>

int unknown(int arr[], int num);

int main()
{
  int arr[] = {10, 20, 30, 40};
  std::cout << unknown(arr, 4) << '\n';
  return 0;
}

int unknown(int arr[], int num)
{
  if(num == 1)
        return arr[0];
  else
        return arr[num-1] + unknown(arr, num-1);
}
```

Παράδειγμα (1)

- Απάντηση: Όταν καλείται η unknown() επιστρέφει:

arr[4-1 = 3] + unknown(arr, 4-1 = 3) =
arr[3] + (arr[3-1] + unknown(arr, 3-1)) =
arr[3] + arr[2] + (arr[2-1] + unknown(arr, 2-1)) =
arr[3] + arr[2] + arr[1] + unknown(arr, 1)

Η τελευταία κλήση της unknown(arr, 1) επιστρέφει arr[0], γιατί num = 1. Επομένως, η τελική τιμή επιστροφής είναι: arr[3]+arr[2]+arr[1]+arr[0], δηλαδή, η συνάρτηση υπολογίζει με αναδρομικό τρόπο το άθροισμα των στοιχείων ενός πίνακα και το πρόγραμμα εμφανίζει 100.

Παράδειγμα (2)

- Δημιουργήστε μία αναδρομική συνάρτηση που να δέχεται σαν παράμετρο έναν ακέραιο αριθμό `n` και να επιστρέφει το παραγοντικό του, χρησιμοποιώντας τον τύπο `n! = n*(n-1)!`. Το παραγοντικό ενός ακεραίου `n`, όπου `n >= 1`, είναι το γινόμενο των ακεραίων από το `1` μέχρι και το `n`, δηλαδή `1*2*3*...*n`. Το παραγοντικό του `0` είναι `1` (`0!=1`). Επειδή τα παραγοντικά αριθμών αυξάνονται πολύ γρήγορα και μπορεί να ξεπεραστεί το εύρος του μεγαλύτερου ακεραίου τύπου μην εισάγετε μεγάλο ακέραιο

Παράδειγμα (2)

```
#include <iostream>

unsigned long long int fact(int num);

int main()
{
  int num;
  do
  {
        std::cout << "Enter a positive integer: ";
        std::cin >> num;
  } while(num < 0);

  std::cout << "Factorial of " << num << " is " << fact(num) << '\n';
  return 0;
}
unsigned long long int fact(int num)
{
  if((num == 0) || (num == 1))
        return 1;
  else
        return num * fact(num-1);
}
```

- Σχόλια: Παρατηρήστε ότι για μεγάλες τιμές της num οι κλήσεις της fact() αυξάνουν σημαντικά και επομένως και ο χρόνος υπολογισμού της τελικής τιμής. Σε αυτή την περίπτωση, η λύση με τον for βρόχο στην αντίστοιχη άσκηση του βιβλίου στο Κ.6 υπολογίζει το παραγοντικό του αριθμού πιο γρήγορα

C++: Από τη Θεωρία στην Εφαρμογή

Κεφάλαιο 12ο

Αναζήτηση/Ταξινόμηση Πίνακα

Γραμμική Αναζήτηση (Linear Search)

- Η γραμμική αναζήτηση (linear search) είναι ο πιο απλός τρόπος αναζήτησης μίας τιμής μέσα σε έναν μη ταξινομημένο πίνακα
- Η αναζήτηση γίνεται σειριακά, ξεκινώντας από το πρώτο στοιχείο του πίνακα μέχρι το τελευταίο
- Σε έναν πίνακα `n` στοιχείων, το μέγιστο πλήθος αναζητήσεων μίας τιμής είναι `n` και συμβαίνει όταν η προς αναζήτηση τιμή είναι είτε το τελευταίο στοιχείο είτε δεν υπάρχει στον πίνακα

Παράδειγμα

- Δημιουργήστε μία συνάρτηση που να αναζητά έναν αριθμό σε έναν πίνακα πραγματικών αριθμών. Αν ο αριθμός υπάρχει στον πίνακα, η συνάρτηση να επιστρέφει το πλήθος των εμφανίσεών του και τη θέση της πρώτης εμφάνισής του, αλλιώς -1. Να γραφεί ένα πρόγραμμα το οποίο να διαβάζει μέχρι 100 πραγματικούς και να τους αποθηκεύει σε έναν πίνακα. Αν ο χρήστης εισάγει την τιμή -1, η εισαγωγή των αριθμών να σταματάει. Στη συνέχεια, το πρόγραμμα να διαβάζει έναν πραγματικό αριθμό και να εμφανίζει το πλήθος των εμφανίσεών του και τη θέση της πρώτης εμφάνισής του με χρήση της συνάρτησης.

Παράδειγμα

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

const int SIZE = 100;

int linear_search(const double arr[], int size, double num, int *t);

int main()
{
        int i, times, pos;
        double num, arr[SIZE];

        for(i = 0; i < SIZE; i++)
        {
                cout << "Enter number: ";
                cin >> num;
                if(num == -1)
                        break;
                arr[i] = num;
        }
        cout << "Enter number to search: ";
        cin >> num;

        pos = linear_search(arr, i, num, &times); /* Η μεταβλητή i δηλώνει τον αριθμό των
στοιχείων του πίνακα. */
        if(pos == -1)
                cout << num << "isn't found\n";
        else
                cout << num << " appears " << times << " times (first pos = " << pos <<
")\n";
        return 0;
}
```

Παράδειγμα

```
int linear_search(const double arr[], int size, double num,
int *t)
{
        int i, pos;

        pos = -1;
        *t = 0;
        for(i = 0; i < size; i++)
        {
                if(arr[i] == num)
                {
                        (*t)++;
                        if(pos == -1) /* Αποθήκευση της πρώτης
εμφάνισης. */
                                pos = i;
                }
        }
        return pos;
}
```

Σχόλια: Αφού μία συνάρτηση δεν μπορεί να επιστρέψει δύο τιμές, επιλέξαμε η δεύτερη τιμή να επιστραφεί μέσω δείκτη

Δυαδική Αναζήτηση (Binary Search)

- Ο αλγόριθμος της δυαδικής αναζήτησης (binary search) χρησιμοποιείται για την αναζήτηση μίας τιμής σε έναν ταξινομημένο πίνακα κατά αύξουσα ή φθίνουσα σειρά
- Εν συντομία, αν θεωρήσουμε ότι έχουμε έναν ταξινομημένο πίνακα, με τη δυαδική αναζήτηση ο ταξινομημένος πίνακας χωρίζεται σε ίσα τμήματα κάθε φορά και συγκρίνουμε την τιμή που ψάχνουμε να βρούμε με την τιμή του μεσαίου στοιχείου του κάθε τμήματος επαναλαμβάνοντας τη συγκεκριμένη διαδικασία αναζήτησης στο αντίστοιχο τμήμα του πίνακα
- Η εκτέλεση του αλγορίθμου τερματίζεται είτε στην περίπτωση που βρεθεί η τιμή στον πίνακα είτε στην περίπτωση που η τιμή της μεταβλητής που χρησιμοποιούμε για να δείξουμε το «κάτω όριο» του πίνακα στον οποίο κάνουμε την αναζήτηση γίνει μεγαλύτερη από την τιμή της αντίστοιχης μεταβλητής του «πάνω ορίου» (γεγονός που σημαίνει ότι η τιμή που αναζητούμε, δεν υπάρχει)
- Το μέγιστο πλήθος αναζητήσεων μίας τιμής σε έναν ταξινομημένο πίνακα **n** στοιχείων με τον αλγόριθμο της δυαδικής αναζήτησης είναι $\log_2 n$

Δυαδική Αναζήτηση Βήμα προς Βήμα (1)

- Για να δούμε πώς λειτουργεί ο αλγόριθμος, ας θεωρήσουμε ότι αναζητούμε μία τιμή σε έναν ταξινομημένο πίνακα κατά αύξουσα σειρά:

- **Βήμα 1:**
  - Χρησιμοποιούμε δύο ακέραιες μεταβλητές, έστω με τα ονόματα `start` και `end`, οι οποίες δηλώνουν αντίστοιχα την αρχή και το τέλος του τμήματος του πίνακα, στο οποίο αναζητούμε τη συγκεκριμένη τιμή

  - Στη συνέχεια, υπολογίζουμε τη μεσαία θέση του πίνακα, η οποία είναι: `middle = (start + end)/2`

    Π.χ. αν έχουμε έναν ταξινομημένο πίνακα 100 ακεραίων, τότε η τιμή της μεταβλητής `start` είναι `0`, της `end` είναι `99` και της `middle` είναι `49`

Δυαδική Αναζήτηση Βήμα προς Βήμα (2)

- Βήμα 2: Συγκρίνουμε την τιμή που αναζητούμε με την τιμή του στοιχείου που βρίσκεται στη μεσαία θέση:
  - Αν είναι ίσες, η τιμή βρέθηκε και η αναζήτηση τερματίζεται
  - Αν είναι μεγαλύτερη, η αναζήτηση συνεχίζεται στο τμήμα του πίνακα που αρχίζει από τη θέση `middle` μέχρι τη θέση `end`. Η τιμή της `start` γίνεται `start = middle+1` και η εκτέλεση του αλγορίθμου μεταβαίνει στο Βήμα 1
  - Αν είναι μικρότερη, η αναζήτηση συνεχίζεται στο τμήμα του πίνακα που αρχίζει από τη θέση `start` μέχρι τη θέση `middle`. Η τιμή της `end` γίνεται `end = middle-1` και η εκτέλεση του αλγορίθμου μεταβαίνει στο Βήμα 1
- Συνοπτικά, ο αλγόριθμος χωρίζει τον ταξινομημένο πίνακα σε δύο τμήματα. Η τιμή που ψάχνουμε συγκρίνεται με την τιμή του μεσαίου στοιχείου και η αναζήτηση συνεχίζεται στο κατάλληλο τμήμα του πίνακα. Ο αλγόριθμος τερματίζεται όταν βρεθεί η τιμή που ψάχνουμε ή όταν η τιμή της `start` γίνει μεγαλύτερη από την τιμή της `end`, που σημαίνει ότι η τιμή που ψάχνουμε δεν περιέχεται στον πίνακα
- Ο αλγόριθμος λειτουργεί με παρόμοιο τρόπο για την αναζήτηση μίας τιμής όταν ο πίνακας είναι ταξινομημένος κατά φθίνουσα σειρά. Η μοναδική διαφορά είναι στο δεύτερο βήμα, κατά τη σύγκριση της τιμής που αναζητούμε με την τιμή του στοιχείου που βρίσκεται στη μεσαία θέση, στο οποίο κάνουμε τις αντίθετες ακριβώς συγκρίσεις σε σχέση με αυτές που περιγράψαμε παραπάνω

Παράδειγμα

- Δημιουργήστε μία συνάρτηση που να αναζητά έναν αριθμό σε έναν ταξινομημένο πίνακα ακεραίων. Αν ο αριθμός υπάρχει στον πίνακα, η συνάρτηση να επιστρέφει τη θέση της πρώτης εμφάνισής του, αλλιώς -1. Να γραφεί ένα πρόγραμμα το οποίο να δηλώνει έναν πίνακα ακεραίων με τιμές ταξινομημένες κατά αύξουσα σειρά. Το πρόγραμμα να διαβάζει έναν ακέραιο και να εμφανίζει τη θέση του στον πίνακα με χρήση της συνάρτησης

Παράδειγμα

```
#include <iostream>

int binary_search(const int arr[], int size, int num);

int main()
{
        int num, pos, arr[] = {10, 20, 30, 40, 50, 60, 70};

        std::cout << "Enter number to search: ";
        std::cin >> num;

        pos = binary_search(arr, sizeof(arr)/sizeof(int), num);
/* Η δεύτερη παράμετρος υπολογίζει με γενικό τρόπο (δηλαδή,
αντί να γράψουμε 7) τον αριθμό των στοιχείων του πίνακα. */
        if(pos == -1)
                std::cout << num << " isn't found\n";
        else
                std::cout << num << " is found in position " <<
pos << '\n';
        return 0;
}
```

Παράδειγμα

```
int binary_search(const int arr[], int size, int num)
{
        int start, end, middle;

        start = 0;
        end = size-1;
        while(start <= end)
        {
                middle = (start+end)/2;

                if(num < arr[middle])
                        end = middle-1;
                else if(num > arr[middle])
                        start = middle+1;
                else
                        return middle;
        }
        return -1; /* Αν η εκτέλεση του κώδικα φτάσει σε αυτό
το σημείο σημαίνει ότι ο αριθμός δεν βρέθηκε και η συνάρτηση
επιστρέφει -1. */
}
```

Ταξινόμηση Πίνακα με τον Αλγόριθμο Επιλογής

- Για να περιγράψουμε τον αλγόριθμο, έστω ότι θέλουμε να ταξινομήσουμε τα στοιχεία ενός πίνακα κατά αύξουσα σειρά
- Αρχικά, βρίσκουμε το στοιχείο του πίνακα με τη μικρότερη τιμή και αντιμεταθέτουμε την τιμή του με την τιμή του πρώτου στοιχείου του πίνακα. Άρα, η ελάχιστη τιμή του πίνακα αποθηκεύεται στην πρώτη θέση του
- Στη συνέχεια, βρίσκουμε τη νέα ελάχιστη τιμή μεταξύ των υπολοίπων στοιχείων του πίνακα, δηλαδή, χωρίς το πρώτο στοιχείο. Όπως προηγουμένως, αντιμεταθέτουμε αυτή την τιμή με την τιμή του δεύτερου στοιχείου. Άρα, η δεύτερη μικρότερη τιμή του πίνακα αποθηκεύεται στη δεύτερη θέση του
- Η διαδικασία αυτή επαναλαμβάνεται για τα υπόλοιπα στοιχεία του πίνακα. Ο αλγόριθμος τερματίζεται όταν γίνει η σύγκριση και των δύο τελευταίων στοιχείων του
- Για την ταξινόμηση του πίνακα κατά φθίνουσα σειρά, η μοναδική διαφορά είναι ότι βρίσκουμε κάθε φορά τη μέγιστη τιμή αντί της ελάχιστης

Παράδειγμα

- Δημιουργήστε μία συνάρτηση που να δέχεται σαν παράμετρο έναν πίνακα πραγματικών και να τον ταξινομεί κατά αύξουσα σειρά σύμφωνα με τον αλγόριθμο της ταξινόμησης με επιλογή. Να γραφεί ένα πρόγραμμα το οποίο να διαβάζει τους βαθμούς 10 φοιτητών, να τους αποθηκεύει σε έναν πίνακα, να καλεί τη συνάρτηση και να εμφανίζει τον ταξινομημένο πίνακα

Παράδειγμα

```
#include <iostream>

void sel_sort(double arr[]);

const int SIZE = 10;

int main()
{
        int i;
        double grd[SIZE];

        for(i = 0; i < SIZE; i++)
        {
                std::cout << "Enter grade of stud_" << i+1 << ": ";
                std::cin >> grd[i];
        }
        sel_sort(grd);

        std::cout << "\n***** Sorted array *****\n";
        for(i = 0; i < SIZE; i++)
                std::cout << grd[i] << '\n';
        return 0;
}
```

Παράδειγμα

```
void sel_sort(double arr[])
{
        int i, j;
        double tmp;

        for(i = 0; i < SIZE-1; i++)
        {
                for(j = i+1; j < SIZE; j++)
                {
                        if(arr[i] > arr[j])
                        {
                                // Αντιμετάθεση τιμών.
                                tmp = arr[i];
                                arr[i] = arr[j];
                                arr[j] = tmp;
                        }
                }
        }
}
```

Σχόλια: Σε κάθε επανάληψη του εσωτερικού βρόχου, το arr[i] συγκρίνεται με τα στοιχεία του πίνακα από τη θέση i+1 μέχρι και τη θέση SIZE-1. Αν κάποιο στοιχείο έχει μικρότερη τιμή, γίνεται αντιμετάθεση των τιμών τους. Για παράδειγμα, στην πρώτη επανάληψη του εξωτερικού βρόχου (i = 0), η ελάχιστη τιμή των στοιχείων από τη θέση 1 μέχρι τη θέση SIZE-1 αποθηκεύεται στο στοιχείο arr[0]. Για να ταξινομήσουμε τον πίνακα κατά φθίνουσα σειρά, απλά αλλάζουμε την if συνθήκη σε: if(arr[i] < arr[j])

Ταξινόμηση Πίνακα με τον Αλγόριθμο Εισαγωγής (1)

- Για την περιγραφή του, έστω ότι θέλουμε να ταξινομήσουμε τα στοιχεία ενός πίνακα κατά αύξουσα σειρά
- Ο αλγόριθμος στηρίζεται στις διαδοχικές συγκρίσεις κάθε στοιχείου του πίνακα (ξεκινώντας από το δεύτερο και καταλήγοντας στο τελευταίο του στοιχείο) με τα στοιχεία που κάθε φορά βρίσκονται στα αριστερά του και αποτελούν τον ήδη «ταξινομημένο υποπίνακα»
- Τα στοιχεία στα δεξιά αποτελούν τον «μη-ταξινομημένο υποπίνακα»
- Κάθε φορά, συγκρίνουμε το αριστερότερο στοιχείο (τρέχον στοιχείο) του «μη-ταξινομημένου υποπίνακα» με τα στοιχεία του «ταξινομημένου υποπίνακα», σύμφωνα με την εξής λογική:

Ταξινόμηση Πίνακα με τον Αλγόριθμο Εισαγωγής (2)

- Συγκρίνουμε το τρέχον στοιχείο διαδοχικά με κάθε ένα από τα στοιχεία του ήδη «ταξινομημένου υποπίνακα», τον οποίο διατρέχουμε από δεξιά προς τα αριστερά

- Κατά τη σύγκριση, κάθε φορά:
  - αν το τρέχον στοιχείο είναι ίσο ή μεγαλύτερο από το δεξιότερο στοιχείο του «ταξινομημένου υποπίνακα» παραμένει στη θέση του και ο αλγόριθμος συνεχίζεται με τις συγκρίσεις του επόμενου στοιχείου του πίνακα
  - Αν είναι μικρότερο, μετακινούμε κατά μία θέση δεξιά το στοιχείο του «ταξινομημένου υποπίνακα» και συγκρίνουμε το τρέχον στοιχείο με το προτελευταίο στοιχείο του «ταξινομημένου υποπίνακα» (σχ.β και γ)
    - Αν το τρέχον στοιχείο είναι μεγαλύτερο ή ίσο, εισάγεται στη θέση του τελευταίου στοιχείου που μετακινήσαμε προς τα δεξιά
    - Αν είναι και πάλι μικρότερο, επαναλαμβάνουμε την ίδια διαδικασία (μετακινούμε το δεύτερο δεξιότερο στοιχείο του «ταξινομημένου υποπίνακα» κατά μία θέση δεξιά και συγκρίνουμε το τρέχον στοιχείο με το προ-προτελευταίο στοιχείο) (σχ. γ και δ)

Ταξινόμηση Πίνακα με τον Αλγόριθμο Εισαγωγής (3)

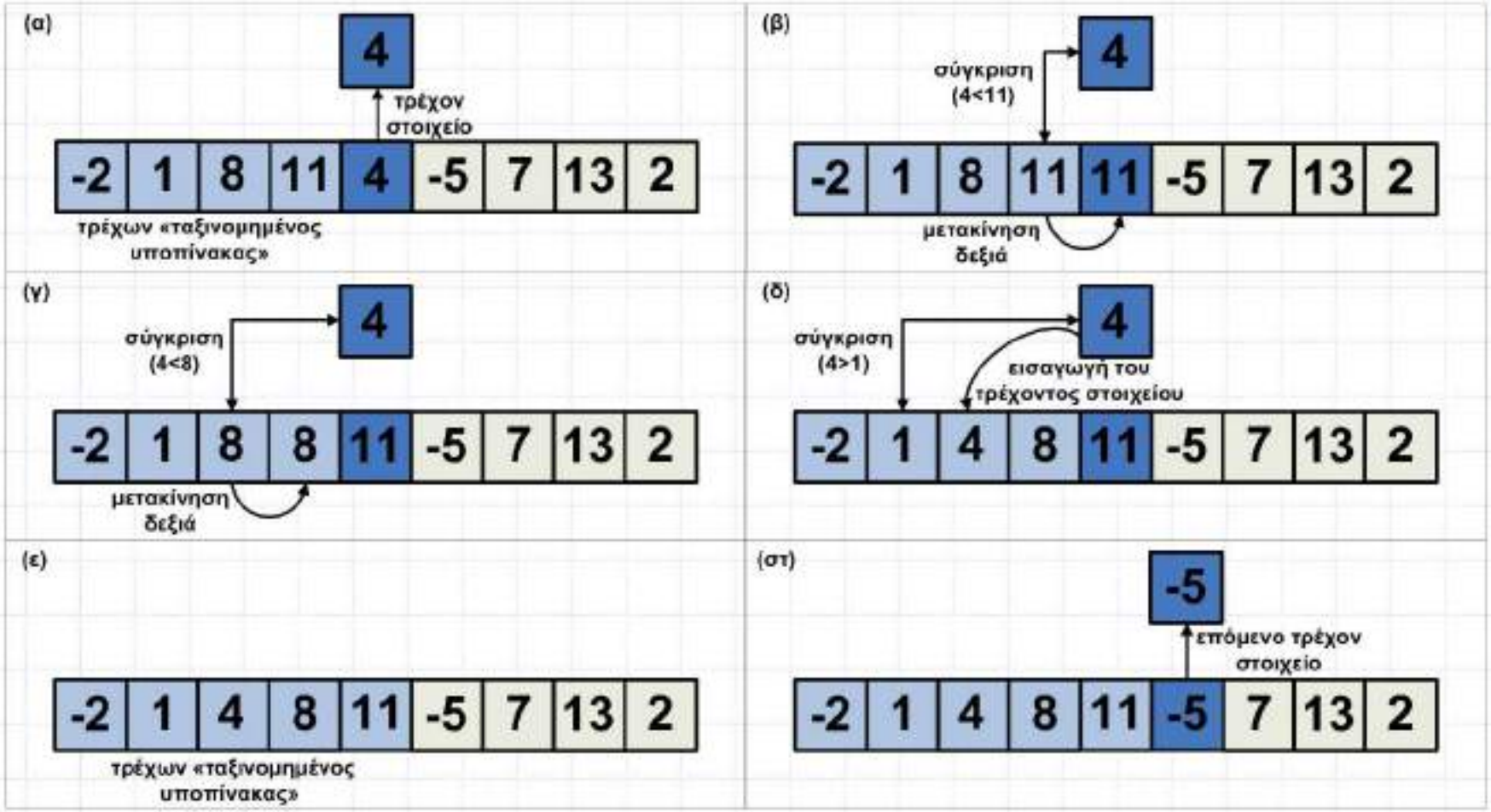
- Οι συγκρίσεις με τα στοιχεία του «ταξινομημένου υποπίνακα» συνεχίζονται μέχρι το τρέχον στοιχείο να μην είναι μικρότερο από κάποιο στοιχείο του (σχ. δ και ε) ή αν φτάσουμε στην αρχή του «ταξινομημένου υποπίνακα».

- Στη συνέχεια, ο αλγόριθμος συνεχίζεται με τις συγκρίσεις του επόμενου αριστερότερου στοιχείου (σχ. ε και στ)

- Όταν ελεγχθεί και το δεξιότερο στοιχείο του «μη-ταξινομημένου υποπίνακα», ο αλγόριθμος τερματίζει

Ταξινόμηση Πίνακα με τον Αλγόριθμο Εισαγωγής (4)

- Το παρακάτω σχήμα αποτυπώνει με ένα παράδειγμα τη λογική που ακολουθείται κατά την εκτέλεση του αλγορίθμου



Ταξινόμηση Πίνακα με τον Αλγόριθμο Εισαγωγής (5)

- Ένας εύκολος τρόπος για να κατανοήσετε καλύτερα και να θυμάστε πιο εύκολα τον συγκεκριμένο αλγόριθμο, είναι να σκεφτείτε ότι η εκτέλεση του αλγορίθμου μοιάζει πολύ με τον τρόπο που οι περισσότεροι άνθρωποι ταξινομούν τα χαρτιά που τους έχουν μοιραστεί σε ένα παιχνίδι τράπουλας
- Θεωρήστε ότι τα χαρτιά είναι όλα κλειστά μπροστά σας στο τραπέζι (αυτός είναι ο πίνακας που θέλετε να ταξινομήσετε) και θέλετε, σηκώνοντας ένα-ένα χαρτί με το δεξί σας χέρι, να καταλήξουν ταξινομημένα στο αριστερό σας χέρι (εκεί βρίσκεται ο «ταξινομημένος υποπίνακας»)
- Κάθε φορά που σηκώνετε ένα χαρτί, το συγκρίνετε με τα χαρτιά που ήδη κρατάτε στο αριστερό σας χέρι και το τοποθετείτε στη σωστή θέση, μετακινώντας όλα τα μεγαλύτερα χαρτιά κατά μία θέση δεξιά
- Έτσι, όταν τελειώσουν όλα τα κλειστά χαρτιά από μπροστά σας, στο αριστερό σας χέρι θα βρίσκονται ταξινομημένα τα χαρτιά που σας μοίρασαν στη συγκεκριμένη παρτίδα
- Για την ταξινόμηση του πίνακα κατά φθίνουσα σειρά, η μοναδική διαφορά είναι ότι το στοιχείο του «ταξινομημένου υποπίνακα» μετακινείται κατά μία θέση δεξιά αν έχει μικρότερη τιμή από το τρέχον στοιχείο

Παράδειγμα

- Δημιουργήστε μία συνάρτηση που να δέχεται σαν παράμετρο έναν πίνακα ακεραίων και να τον ταξινομεί κατά αύξουσα σειρά σύμφωνα με τον αλγόριθμο της ταξινόμησης με εισαγωγή. Να γραφεί ένα πρόγραμμα το οποίο να διαβάζει 5 ακεραίους, να τους αποθηκεύει σε έναν πίνακα, να καλεί τη συνάρτηση και να εμφανίζει τον ταξινομημένο πίνακα

Παράδειγμα

```
#include <iostream>

const int SIZE = 5;

void insert_sort(int arr[]);

int main()
{
        int i, a[SIZE];

        for(i = 0; i < SIZE; i++)
        {
                std::cout << "Enter number: ";
                std::cin >> a[i];
        }
        insert_sort(a);
        std::cout << "\n***** Sorted array *****\n";
        for(i = 0; i < SIZE; i++)
                std::cout << a[i] << '\n';
        return 0;
}
```

Παράδειγμα

```
void insert_sort(int arr[])
{
        int i, j, tmp;

        for(i = 1; i < SIZE; i++)
        {
                tmp = arr[i];
                j = i;
                while((j > 0) && (arr[j-1] > tmp))
                {
                        arr[j] = arr[j-1]; /* Μετακίνηση του στοιχείου
κατά μία θέση δεξιά. */
                        j--;
                }
                arr[j] = tmp;
        }
}
```

Σχόλια: Ο for βρόχος διατρέχει τον πίνακα από το δεύτερο έως και το τελευταίο στοιχείο του. Σε κάθε επανάληψη, στη μεταβλητή tmp αποθηκεύεται το τρέχον στοιχείο του πίνακα. Με τον while βρόχο μετακινούμε κατά μία θέση προς τα δεξιά όλα τα στοιχεία που βρίσκονται αριστερά του τρέχοντος στοιχείου και είναι μεγαλύτερά του

Ταξινόμηση Πίνακα με τον Αλγόριθμο της Φυσαλίδας (1)

- Σύμφωνα με τον αλγόριθμο, γίνονται διαδοχικές συγκρίσεις των τιμών των γειτονικών στοιχείων του πίνακα και οι τιμές «ανεβαίνουν» σαν φυσαλίδες προς τα επάνω και αποθηκεύονται στις αντίστοιχες θέσεις του πίνακα

- Έστω ότι θέλουμε να ταξινομήσουμε έναν πίνακα κατά αύξουσα σειρά

- Αρχικά, συγκρίνεται η τιμή του τελευταίου στοιχείου του πίνακα με την τιμή του προτελευταίου στοιχείου του. Αν είναι μικρότερη, γίνεται αντιμετάθεση των τιμών και η μικρότερη τιμή «ανεβαίνει» σαν φυσαλίδα προς τα επάνω

- Στη συνέχεια, συγκρίνεται η νέα τιμή του προτελευταίου στοιχείου με την τιμή του προ-προτελευταίου στοιχείου του. Αν είναι πάλι μικρότερη, οι τιμές αντιμετατίθενται και αυτή η τιμή συνεχίζει να «ανεβαίνει» προς τα επάνω

- Οι συγκρίσεις συνεχίζονται μέχρι την αρχή του πίνακα και, τελικά, με αυτόν τον τρόπο η ελάχιστη τιμή θα «ανέβει» προς τα επάνω και θα αποθηκευτεί στην πρώτη θέση του πίνακα

Ταξινόμηση Πίνακα με τον Αλγόριθμο της Φυσαλίδας (2)

- Παρομοίως, συνεχίζουμε με την εύρεση της ελαχίστης τιμής στο τμήμα του πίνακα από τη δεύτερη μέχρι και την τελευταία θέση του. Με διαδοχικές συγκρίσεις των γειτονικών στοιχείων και, αν χρειαστεί, αντιμετάθεση των τιμών τους, αυτή η τιμή θα «ανέβει» προς τα επάνω και θα αποθηκευτεί στη δεύτερη θέση του πίνακα

- Η διαδικασία αυτή επαναλαμβάνεται για τα υπόλοιπα τμήματα του πίνακα. Όταν καμία τιμή δεν «ανέβει» προς τα επάνω σημαίνει ότι ο πίνακας ταξινομήθηκε και ο αλγόριθμος τερματίζεται

- Παρόμοια λογική εφαρμόζεται και για την ταξινόμηση του πίνακα κατά φθίνουσα σειρά, μόνο που σε αυτή την περίπτωση η τιμή που «ανεβαίνει» είναι η μεγαλύτερη

Παράδειγμα

- Δημιουργήστε μία συνάρτηση που να δέχεται σαν παράμετρο έναν πίνακα ακεραίων και να τον ταξινομεί κατά αύξουσα σειρά σύμφωνα με τον αλγόριθμο της φυσαλίδας. Να γραφεί ένα πρόγραμμα το οποίο να διαβάζει 5 ακεραίους, να τους αποθηκεύει σε έναν πίνακα, να καλεί τη συνάρτηση και να εμφανίζει τον ταξινομημένο πίνακα

Παράδειγμα

```
#include <iostream>

const int SIZE = 5;

void bubble_sort(int arr[]);

int main()
{
        int i, a[SIZE];

        for(i = 0; i < SIZE; i++)
        {
                std::cout << "Enter number: ";
                std::cin >> a[i];
        }
        bubble_sort(a);

        std::cout << "\n***** Sorted array *****\n";
        for(i = 0; i < SIZE; i++)
                std::cout << a[i] << '\n';
        return 0;
}
```

Παράδειγμα

```
void bubble_sort(int arr[])
{
        bool reorder;
        int i, j, tmp;

        for(i = 1; i < SIZE; i++)
        {
                reorder = 0;
                for(j = SIZE-1; j >= i; j--)
                {
                        if(arr[j] < arr[j-1])
                        {
                                // Αντιμετάθεση τιμών.
                                tmp = arr[j];
                                arr[j] = arr[j-1];
                                arr[j-1] = tmp;
                                reorder = 1;
                        }
                }
                if(reorder == 0)
                        return;
        }
}
```

Σχόλια: Η μεταβλητή reorder χρησιμοποιείται για να ελέγξουμε αν η ταξινόμηση ολοκληρώθηκε, ώστε να αποφύγουμε περιττές επαναλήψεις. Αν γίνει αντιμετάθεση τιμών, η τιμή της γίνεται 1. Αν παραμείνει 0, σημαίνει ότι ο πίνακας ταξινομήθηκε και η συνάρτηση τερματίζεται

Ο Αλγόριθμος Γρήγορης Ταξινόμησης (1)

- Ο αλγόριθμος γρήγορης ταξινόμησης αποτελεί έναν αλγόριθμο του τύπου «διαίρει και βασίλευε» (divide and conquer)
- Είναι ιδιαίτερα δημοφιλής και αποτελεί την καλύτερη μέθοδο ταξινόμησης για μία μεγάλη ποικιλία εφαρμογών ταξινόμησης
- Η λειτουργία του βασίζεται στη διαμέριση του πίνακα σε δύο μέρη και την ταξινόμηση του κάθε τμήματος ξεχωριστά
- Αρχικά, επιλέγεται το στοιχείο της διαμέρισης (π.χ. `a[i]`). Ο πίνακας αναδιατάσσεται έτσι ώστε να ισχύουν τα παρακάτω:

  α) Το στοιχείο `a[i]` να βρίσκεται στην τελική του θέση

  β) Κανένα από τα στοιχεία του αριστερού υποπίνακα `a[l]`, … `a[i-1]` να μην είναι μεγαλύτερο από το `a[i]`

  γ) Κανένα από τα στοιχεία του δεξιού υποπίνακα `a[i+1]`, … `a[r]` να μην είναι μικρότερο από το `a[i]`
- Αυτή η διαδικασία επαναλαμβάνεται αναδρομικά για κάθε έναν από τους υποπίνακες, ώσπου τα τμήματα να γίνουν μεμονωμένα στοιχεία

Ο Αλγόριθμος Γρήγορης Ταξινόμησης (2)

- Η απόδοση του αλγορίθμου εξαρτάται από την επιλογή του στοιχείου διαμέρισης
- Αν υπάρχει πληροφόρηση για τα περιεχόμενα του πίνακα, η καλύτερη επιλογή είναι ένα στοιχείο που να διαιρεί τον πίνακα κοντά στο μέσο
- Σαν παράδειγμα υλοποίησης του αλγορίθμου ας ταξινομήσουμε έναν πίνακα σε αύξουσα σειρά
- Για την καλύτερη απόδοση του αλγορίθμου, έχουν προταθεί διάφορες μέθοδοι για τον τρόπο επιλογής του στοιχείου διαμέρισης. Για απλότητα στην υλοποίηση, ας επιλέξουμε το δεξιότερο στοιχείο

Ο Αλγόριθμος Γρήγορης Ταξινόμησης (3)

- Σαρώνουμε τον πίνακα από το αριστερό του άκρο μέχρι να βρούμε κάποιο στοιχείο μεγαλύτερο ή ίσο από το στοιχείο διαμέρισηςον πίνακα
- Μετά, σαρώνουμε τον πίνακα από το δεξί του άκρο προς τα πίσω μέχρι να βρούμε κάποιο στοιχείο μικρότερο από το στοιχείο διαμέρισης
- Τα δύο στοιχεία στα οποία θα σταματήσουν οι σαρώσεις δεν είναι σε σωστές θέσεις, οπότε τα αντιμεταθέτουμε
- Συνεχίζοντας έτσι, το τελικό αποτέλεσμα είναι να μην υπάρχει κανένα στοιχείο στον αριστερό υποπίνακα με τιμή μεγαλύτερη από το στοιχείο διαμέρισης και κανένα στοιχείο στον δεξιό υποπίνακα με τιμή μικρότερη από το στοιχείο διαμέρισης
- Για την αριστερή σάρωση χρησιμοποιούμε τον δείκτη σάρωσης `i`, και για τη δεξιά τον δείκτη `j`
- Όταν συναντηθούν οι δείκτες, η διαμέριση ολοκληρώνεται με την αντιμετάθεση του δεξιότερου στοιχείου με το στοιχείο στο οποίο δείχνει ο αριστερός δείκτης `i`
- Για την υλοποίηση του αλγορίθμου θα χρησιμοποιήσουμε αναδρομή

Παράδειγμα

```
#include <iostream>

int partition(int a[], int l, int r);
void quick_sort(int a[], int l, int r);

const int SIZE = 7;

int main()
{
        int i, a[SIZE];

        for(i = 0; i < SIZE; i++)
        {
                std::cout << "Enter number: ";
                std::cin >> a[i];
        }
        quick_sort(a, 0, SIZE-1); // Υλοποίηση αλγορίθμου.
        std::cout << "\n***** Sorted array *****\n";
        for(i = 0; i < SIZE; i++)
                std::cout << a[i] << '\n';
        return 0;
}

void quick_sort(int a[], int l, int r)
{
        int i;

        if(r <= l)
                return;
        i = partition(a, l, r);
        quick_sort(a, l, i-1);
        quick_sort(a, i+1, r);
}
```

Παράδειγμα

```
int partition(int a[], int l, int r)
{
        int i, j, v, tmp;

        i = l;
        j = r-1;
        v = a[r];
        while(1)
        {
                while(a[i] < v)
                        i++;
                while(a[j] >= v)
                {
                        if(j == l) /* Ελέγχουμε την περίπτωση το στοιχείο
διαμέρισης να είναι το μικρότερο στοιχείο του τμήματος που εξετάζουμε. */
                                break;
                        j--;
                }
                if(i >= j)
                        break;
                tmp = a[i];
                a[i] = a[j];
                a[j] = tmp;
        }
        tmp = a[i];
        a[i] = a[r];
        a[r] = tmp;
        return i;
}
```

C++: Από τη Θεωρία στην Εφαρμογή

Κεφάλαιο 13ο
Δομές & Ενώσεις

Δομές

- Η δομή είναι μία συλλογή πεδίων, που χρησιμοποιούνται για την ομαδοποίηση πληροφορίας που περιγράφει μία λογική οντότητα

Π.χ. μία δομή μπορεί να περιέχει πληροφορίες για μία εταιρεία, όπως την επωνυμία της, το έτος ίδρυσης, το Α.Φ.Μ, τη διεύθυνσή της, το αντικείμενο εργασιών της, τον αριθμό των υπαλλήλων της, στοιχεία επικοινωνίας και άλλα δεδομένα ή ακόμα και συναρτήσεις που να διαχειρίζονται τα αποθηκευμένα δεδομένα της

Δήλωση Δομής (1)

- Η δήλωση μίας δομής αρχίζει με τη δεσμευμένη λέξη `struct` και στη γενική της μορφή έχει την ακόλουθη σύνταξη:

```
struct ετικέτα_δομής
{
        Δηλώσεις πεδίων;
        Δηλώσεις συναρτήσεων;
} λίστα_μεταβλητών;
```

- Μία `struct` δήλωση δημιουργεί έναν τύπο, ο οποίος ορίζεται από τον προγραμματιστή
- Αν και η `ετικέτα_δομής` είναι προαιρετική, εμείς θα ονοματίζουμε κάθε τύπο δομής που δημιουργούμε, ώστε να μπορούμε να χρησιμοποιούμε την ετικέτα, όποτε θέλουμε, για να δηλώνουμε αντίστοιχες μεταβλητές
- Τα πεδία ή μέλη της δομής, χρησιμοποιούνται όπως και οι μεταβλητές του αντίστοιχου τύπου
- Μία δομή μπορεί να αποτελείται από πεδία διαφορετικού τύπου. Η λογική συσχέτισή τους είναι ότι περιέχουν την πληροφορία που απαιτείται για την περιγραφή μίας συγκεκριμένης οντότητας
- Όπως φαίνεται στη δήλωση, μπορούμε προαιρετικά να δηλώσουμε μία `λίστα_μεταβλητών` αυτού του τύπου

Δήλωση Δομής (2)

- Μία δομή μπορεί να περιέχει συναρτήσεις. Για απλότητα, θα χρησιμοποιήσουμε δομές που περιέχουν μόνο μεταβλητές και όχι συναρτήσεις, γιατί, όπως θα δούμε στο Κ.17, ο τύπος που συνηθίζεται να επιλέγεται για την προσθήκη συναρτήσεων είναι κλάση και όχι δομή
- Η συνήθης πρακτική είναι οι δηλώσεις δομών μαζί με άλλες δηλώσεις, όπως πρωτότυπα συναρτήσεων και μακροεντολές, να αποθηκεύονται σε ένα ξεχωριστό αρχείο επικεφαλίδας το οποίο να συμπεριλαμβάνεται με την οδηγία `#include` όπου χρειάζεται
- Για απλότητα, στα επόμενα παραδείγματα θα δηλώνουμε τον κάθε τύπο δομής με καθολική εμβέλεια στην αρχή του αρχείου, ώστε όλες οι συναρτήσεις να μπορούν να τον χρησιμοποιήσουν

Παράδειγμα Δήλωσης

- Για παράδειγμα, για να αποθηκεύσουμε πληροφορία για μία εταιρεία θα μπορούσαμε να δηλώσουμε τον τύπο Company:

```
struct Company
{
        string name;
        int start_year;
        int field;
        int tax_num;
        int num_empl;
        string addr;
        float balance;
};
```

- Το πρώτο γράμμα της ετικέτας συνηθίζεται να είναι κεφαλαίο
- Αν και τα πεδία του ίδιου τύπου επιτρέπεται να δηλώνονται στην ίδια γραμμή, προτιμώ να τα δηλώνω ξεχωριστά, ώστε να φαίνεται πιο εύκολα η αντιστοίχιση με την πληροφορία που θέλουμε να αποθηκεύσουμε
- Τα πεδία μίας δομής αποθηκεύονται σε θέσεις μνήμης που αυξάνονται, με το πρώτο να αρχίζει από τη διεύθυνση της δομής
- Η δήλωση μίας δομής που δεν ακολουθείται από λίστα μεταβλητών, όπως εδώ της Company, δεν προκαλεί δέσμευση μνήμης. Απλά, περιγράφει τη μορφή της δομής

Δήλωση Μεταβλητών

- Αν η δομή έχει ετικέτα, μπορούμε να τη χρησιμοποιήσουμε για δηλώσεις μεταβλητών. Για παράδειγμα: `Company c1, c2;`
- Για να είναι ξεκάθαρο, η δήλωση της δομής `Company` ορίζει έναν τύπο δεδομένων με αυτό το όνομα, δεν προκαλεί δέσμευση μνήμης και ούτε η `Company` είναι μεταβλητή
- Η `Company` απλά περιγράφει τη μορφή που θα έχουν οι μεταβλητές της δομής όταν αυτές δηλωθούν
- Οι μεταβλητές `c1` και `c2` δηλώνονται σαν δομές τύπου `Company`, η κάθε μία έχει τα δικά της πεδία και ο μεταγλωττιστής δεσμεύει μνήμη για αυτές
- Εναλλακτικά, μπορούμε να δηλώσουμε μεταβλητές στη `λίστα_μεταβλητών`. Π.χ.

```
struct Book
{
        string title;
        int year;
        float price;
} b1, b2;
```

- Οι μεταβλητές `b1` και `b2` δηλώνονται σαν δομές τύπου `Book`

Παρατηρήσεις (1)

- Όπως με κάθε μεταβλητή, όταν δηλώνεται μία δομή ο μεταγλωττιστής δεσμεύει μνήμη για την αποθήκευση των πεδίων της με τη σειρά που εμφανίζονται. Π.χ. το επόμενο πρόγραμμα εμφανίζει πόσες οκτάδες δεσμεύτηκαν για τη δομή d:

```
#include <iostream>
struct Date
{
        int day;
        int month;
        int year;
};
int main()
{
        Date d;

        std::cout << sizeof(d) << '\n';
        return 0;
}
```

- Αφού η d είναι δομή τύπου Date, αποτελείται από τρία ακέραια πεδία, άρα η μνήμη που δεσμεύτηκε είναι 3*4 = 12 οκτάδες. Αν και σε αυτό το παράδειγμα το μέγεθος της δεσμευμένης μνήμης είναι ίσο με το άθροισμα της μνήμης που δεσμεύουν οι τύποι των πεδίων, μπορεί να υπάρχουν περιπτώσεις που να είναι μεγαλύτερο. Για παράδειγμα, αν αλλάξουμε τον τύπο του πεδίου day από int σε char, το πιθανότερο είναι ότι το πρόγραμμα θα εμφανίσει πάλι 12 και όχι 9, όπως θα ήταν το αναμενόμενο
- Αυτό μπορεί να συμβεί, αν ο μεταγλωττιστής, για πιο γρήγορη πρόσβαση στα πεδία της δομής, απαιτεί κάθε πεδίο να αποθηκεύεται σε μία διεύθυνση που να είναι πολλαπλάσια κάποιου αριθμού (συνήθως του 4). Αν υποθέσουμε ότι το πεδίο month πρέπει να αποθηκευτεί σε μία διεύθυνση πολλαπλάσια του 4, ο μεταγλωττιστής θα δεσμεύσει τρεις πρόσθετες οκτάδες αμέσως μετά το πεδίο day
- Για τον υπολογισμό της μνήμης που καταλαμβάνει μία δομή να χρησιμοποιείτε πάντα τον τελεστή sizeof και να **μην** προσθέτετε τη μνήμη που δεσμεύουν ξεχωριστά τα πεδία της

Αρχικοποίηση Πεδίων Δομής (1)

- Ο πιο συνηθισμένος τρόπος για να προσπελάσουμε ένα πεδίο μίας δομής είναι να γράψουμε το όνομά της, να προσθέσουμε τον τελεστή τελεία (.) και μετά το όνομα του πεδίου
- Το επόμενο πρόγραμμα αρχικοποιεί και εμφανίζει τις τιμές των πεδίων της b

```
#include <iostream>
#include <string>

struct Book
{
        std::string title;
        int year;
        float price;
};
int main()
{
        Book b;

        b.title = "Literature";
        b.year = 2023;
        b.price = 10.85;
        std::cout << b.title << ' '  << b.year << ' ' << b.price <<'\n';
        return 0;
}
```

- Εκτός από τον τελεστή . θα δούμε στη συνέχεια ότι μπορούμε να χρησιμοποιήσουμε και τον τελεστή -> για να προσπελάσουμε τα πεδία μίας μεταβλητής δομής

Αρχικοποίηση Πεδίων Δομής (2)

- Ένας εναλλακτικός τρόπος απόδοσης αρχικών τιμών στα πεδία μίας μεταβλητής δομής είναι μαζί με τη δήλωσή της. Σε αυτή την περίπτωση, τη δήλωση της δομής την ακολουθεί μία λίστα τιμών που περικλείεται σε άγκιστρα και πρέπει να ταιριάζει με τη σειρά δήλωσης των πεδίων, καθώς και με τους τύπους τους. Οι τιμές της λίστας χωρίζονται με κόμμα και δεν πρέπει να είναι περισσότερες από τα πεδία. Π.χ. με τη δήλωση:

```
Book b = {"Literature", 2023, 10.85};
```

η τιμή του `b.title` γίνεται `Literature` το `b.year` ίσο με 2023 και το `b.price` ίσο με `10.85`

- Όπως τα στοιχεία ενός πίνακα, έτσι και τα πεδία μίας δομής που δεν αρχικοποιούνται αποκτούν μηδενικές τιμές. Π.χ. με τη δήλωση:

```
Book b = {"Literature"};
```

η τιμή του `b.title` γίνεται `Literature` και τα `b.year` και `b.price` ίσα με `0`

- Αν οι {} είναι κενές, τα πεδία αρχικοποιούνται με τις προεπιλεγμένες τιμές των τύπων τους. Π.χ. με τη δήλωση:

```
Book b = {};
```

η τιμή του `b.title` γίνεται `""` και τα `b.year` και `b.price` ίσα με `0`

Αρχικοποίηση Πεδίων Δομής (3)

- Επίσης, μία μεταβλητή που δηλώνεται στη `λίστα_μεταβλητών` μπορεί ταυτόχρονα να αρχικοποιηθεί. Π.χ.

```
struct Book
{
        string title;
        int year;
        float price;
} b = {"Literature", 2023, 10.8};
```

- Με την C++11, μπορούμε να παραλείψουμε το = για τον καθορισμό της αρχικής τιμής. Π.χ.

```
Book b1{"Literature", 2023, 10.85};
Book b2{}; /* Τα μέλη της δομής αρχικοποιούνται με τις προεπιλεγμένες
αρχικές τιμές σύμφωνα με τον τύπο τους. Δηλαδή, τα αριθμητικά πεδία με
0 και το title με "". */
```

Δείκτης σε Πεδίο Δομής

- Όπως χρησιμοποιούμε έναν δείκτη σε μία μεταβλητή, μπορούμε να τον χρησιμοποιήσουμε και σε ένα πεδίο μίας δομής. Π.χ. ο επόμενος κώδικας χρησιμοποιεί δείκτες για να εμφανίσει τα πεδία της δομής `b`:

```
int main()
{
        string *p1;
        int *p2;
        float *p3;
        Book b = {"Literature", 2023, 10.8};

        p1 = &b.title;
        p2 = &b.year;
        p3 = &b.price;

        cout << *p1 << ' ' << *p2 << ' ' << *p3 << '\n';
        return 0;
}
```

- Για να δείξει κάποιος δείκτης στο πεδίο μίας δομής πρέπει ο τύπος του να είναι συμβατός με τον τύπο του αντίστοιχου πεδίου. Π.χ. αφού ο τύπος του πεδίου `year` είναι `int`, ο δείκτης `p2` δηλώνεται σαν `int*`

Λειτουργίες μεταξύ Δομών

- Αν και δεν μπορούμε να χρησιμοποιήσουμε τον τελεστή = για να αντιγράψουμε έναν πίνακα σε έναν άλλο, μπορούμε να τον χρησιμοποιήσουμε για να αντιγράψουμε μία δομή σε μία άλλη
- Βέβαια, οι δομές πρέπει να είναι ίδιου τύπου. Π.χ.

```
struct Student
{
        int code;
        float grd;
};

int main()
{
        Student s1, s2;
        s1.code = 1234;
        s1.grd = 6.7;
        s2 = s1; // Αντιγραφή δομής.
        ...
}
```

Παρατηρήσεις (1)

- Με την εντολή:

```
s2 = s1;
```

οι τιμές των πεδίων της δομής **s1** αντιγράφονται στα αντίστοιχα πεδία της δομής **s2**

Δηλαδή, η παραπάνω εντολή είναι ισοδύναμη με:

```
s2.code = s1.code;
s2.grd = s1.grd;
```

- Αν οι **s1** και **s2** δεν ήταν μεταβλητές του ίδιου τύπου δομής, η εντολή `s2 = s1;` δεν θα μεταγλωττιζόταν, ακόμα κι αν τα δύο διαφορετικά πρότυπα δομής περιείχαν τα ίδια ακριβώς πεδία και σε αριθμό και σε τύπο δεδομένων

Παρατηρήσεις (2)

- Σημειώστε, επίσης, ότι αν μία δομή περιέχει πίνακα (π.χ. `name`), αν και, όπως γνωρίζετε, δεν μπορούμε να γράψουμε

```
s2.name = s1.name;
```

με την αντιγραφή της δομής (π.χ. `s2 = s1`) αντιγράφεται και ο πίνακας της `s1` στην `s2`

- Επίσης, αν η δομή περιέχει πεδίο δείκτη, μετά την αντιγραφή και οι δύο δείκτες θα δείχνουν στο ίδιο σημείο, γεγονός που μπορεί να αποβεί πολύ επικίνδυνο

Παρατηρήσεις (3)

- Εκτός από την ανάθεση καμία άλλη ενέργεια δεν επιτρέπεται μεταξύ δομών

- Π.χ. οι τελεστές == και != δεν μπορούν να χρησιμοποιηθούν για τον έλεγχο της ισότητας δύο δομών

  Δηλαδή, δεν επιτρέπεται να γράψετε:

```
if(s1 == s2)
```

ή

```
if(s1 != s2)
```

- Αν πρέπει να γίνει έλεγχος αν δύο δομές περιέχουν τα ίδια ακριβώς δεδομένα, πρέπει αναγκαστικά να λάβει χώρα σύγκριση όλων των πεδίων των δομών ένα προς ένα

  Δηλαδή:

```
if((s1.code == s2.code) && (s1.grd == s2.grd))
```

Παράδειγμα Δομής που Περιέχει Πίνακες

- Ποια είναι η έξοδος του παρακάτω προγράμματος;

```
#include <iostream>
#include <cstring>

struct Student
{
        char name[50];
        float grd[2];
};

int main()
{
        Student s1, s2;
        strcpy(s1.name, "somebody");
        s1.grd[0] = 8.5;
        s1.grd[1] = 7.5;
        std::cout << s1.name << ' ' << s1.name[0] << ' ' <<
*s1.name << '\n';
        s2 = s1;
        std::cout << s2.name << ' ' << s2.grd[0] << ' ' <<
s2.grd[1] << '\n';
        return 0;
}
```

Παράδειγμα Δομής που Περιέχει Πίνακες

Έξοδος:
```
somebody s s
somebody 8.5 7.5
```

Παράδειγμα Δομής που Περιέχει Δείκτες

- Ποια είναι η έξοδος του παρακάτω προγράμματος;

```
#include <iostream>
#include <cstring>

struct Student
{
        const char *name;
        float *avg_grd;
};

int main()
{
        float grd = 8.5;
        Student s1, s2;

        s1.name = "somebody";
        s1.avg_grd = &grd;
        std::cout << s1.name+3 << ' ' << *s1.avg_grd << '\n';

        s2.name = "else";
        s2 = s1;
        grd = 3.4;
        std::cout << s2.name << ' ' << *s2.avg_grd << '\n';
        return 0;
}
```

Παράδειγμα Δομής που Περιέχει Δείκτες

- Με την εντολή `s1.name = "somebody";` ο μεταγλωττιστής αρχικά δεσμεύει μνήμη για να αποθηκεύσει το αλφαριθμητικό `"somebody"` και στη συνέχεια ο δείκτης `name` δείχνει στην αρχή αυτής της μνήμης. Με την εντολή `s1.avg_grd = &grd;` ο δείκτης `avg_grd` δείχνει στη διεύθυνση της `grd`. Για να προσπελάσουμε το περιεχόμενο της μνήμης που δείχνει ένα πεδίο δείκτης, ο τελεστής `*` πρέπει να προηγείται του ονόματος της δομής. Επομένως, το πρόγραμμα εμφανίζει: `ebody 8.5`

- Με την `s2 = s1;` οι δείκτες της `s2` δείχνουν εκεί που δείχνουν οι δείκτες της `s1`. Επομένως, το πρόγραμμα εμφανίζει: `somebody 3.4` (προσοχή, όχι `else`). Ο λόγος που έβαλα αυτή την εντολή είναι για να σας επιστήσω την προσοχή όταν γίνεται αντιγραφή δομής που περιέχει δείκτη. Η τρέχουσα τιμή του δείκτη της δομής στην οποία γίνεται η εκχώρηση (π.χ. `s2`) θα χαθεί. Προσοχή, αυτό μπορεί να αποτελέσει σοβαρό πρόβλημα

- Προσοχή στην αντιγραφή δομής όταν περιέχει δείκτη

Ένθετες Δομές

- Μία δομή μπορεί να περιέχει μία ή περισσότερες δομές, οι οποίες ονομάζονται ένθετες δομές
- Μία ένθετη δομή πρέπει να ορίζεται πριν από τον ορισμό της δομής στην οποία περιέχεται, αλλιώς ο μεταγλωττιστής θα εμφανίσει μήνυμα λάθους
- Για να προσπελάσουμε τα πεδία μίας δομής που περιέχεται μέσα σε μία άλλη δομή χρησιμοποιούμε δύο φορές τον τελεστή τελεία (.)
- Την πρώτη φορά ανάμεσα στο όνομα της εξωτερικής και της ένθετης δομής και τη δεύτερη φορά ανάμεσα στο όνομα της ένθετης δομής και το όνομα του πεδίου που μας ενδιαφέρει (το οποίο ανήκει στην ένθετη δομή)

Παράδειγμα Δομής που Περιέχει Δομή

- Ποια είναι η έξοδος του παρακάτω προγράμματος;

```
#include <iostream>
#include <string>

struct Date
{
        int day;
        int month;
        int year;
};
struct Product /* Αφού ο τύπος Date έχει οριστεί, μπορεί να χρησιμοποιηθεί για να δηλώσουμε ένθετες
δομές. */
{
        std::string name;
        double price;
        Date s_date;
        Date e_date;
};
int main()
{
        Product prod;

        prod.name = "product";
        prod.s_date.day = 1;
        prod.s_date.month = 9;
        prod.s_date.year = 2023;

        prod.e_date.day = 1;
        prod.e_date.month = 9;
        prod.e_date.year = 2025;

        prod.price = 7.5;
        std::cout << "The product's life is " << prod.e_date.year - prod.s_date.year << " years\n";
        return 0;
}
```

Παράδειγμα Δομής που Περιέχει Δομή

Έξοδος: `The product's life is 2 years`

Πεδία Δομής με Μέγεθος bit (1)

- Ένα πεδίο δομής μπορεί να περιέχει πεδία, των οποίων το μέγεθος να δηλώνεται σαν ένας συγκεκριμένος αριθμός από bits
- Ένα τέτοιο πεδίο ονομάζεται **πεδίο bit** και δηλώνεται με τον ακόλουθο τρόπο:

```
τύπος_δεδομένων όνομα_πεδίου_bit : αριθμός_bits;
```

- Τα bit πεδία μπορούν να χρησιμοποιηθούν όπως και τα απλά πεδία μίας δομής
- Ο τύπος ενός bit πεδίου πρέπει να είναι ακέραιος
- Επειδή η μνήμη για ένα bit πεδίο δεν δεσμεύεται όπως με τις συνηθισμένες μεταβλητές, δεν επιτρέπεται να εφαρμοστεί ο τελεστής διεύθυνσης & σε αυτό

Πεδία Δομής με Μέγεθος bit (2)

Παράδειγμα:

```
struct Person
{
        unsigned int sex : 1;
        unsigned int married : 1;
        unsigned int children : 4;
};
```

Μέγεθος bit: 1
Εύρος τιμών: 0 – 1

Μέγεθος bit: 4
Εύρος τιμών: 0 – 15

Πεδία Δομής με Μέγεθος bit (3)

- Για την αποθήκευση των τιμών του προηγούμενου παραδείγματος απαιτούνται συνολικά: 1 + 1 + 4 = 6 bits
- Αφού ο τύπος δεδομένων των πεδίων είναι `unsigned int`, τότε ο μεταγλωττιστής **δεσμεύει** 4 byte μνήμης, και 4×8-6 = 26 bits **μένουν αχρησιμοποίητα**
- Το **κύριο πλεονέκτημα** της χρήσης των πεδίων bit είναι **η εξοικονόμηση μνήμης**
- Π.χ., στο προηγούμενο πρόγραμμα ο μεταγλωττιστής δεσμεύει τέσσερις οκτάδες αντί για δώδεκα που θα δέσμευε αν δεν χρησιμοποιούσαμε bit πεδία
- Επομένως, αφού για τα στοιχεία ενός ανθρώπου εξοικονομούμε οκτώ οκτάδες, αν έπρεπε να αποθηκεύσουμε τα στοιχεία 200.000 ανθρώπων θα εξοικονομούσαμε 1.600.000 οκτάδες

Δείκτης σε Δομή

- Ένας δείκτης σε μία δομή χρησιμοποιείται με τον ίδιο τρόπο όπως κάθε άλλος δείκτης. Π.χ.

```
#include <iostream>
#include <string>

struct Product
{
        std::string name;
        float grd;
};
int main()
{
        Student *ptr, stud;

        ptr = &stud;
        (*ptr).name = "somebody";
        (*ptr).grd = 6.7;
        std::cout << stud.name << ' ' << stud.grd << '\n';
        return 0;
}
```

- Το πρόγραμμα εμφανίζει somebody 6.7. Η μεταβλητή ptr δηλώνεται σαν δείκτης σε δομή τύπου Student. Με την εντολή ptr = &stud; ο ptr δείχνει στη διεύθυνση μνήμης της δομής stud. Συγκεκριμένα, δείχνει στη διεύθυνση μνήμης του πρώτου πεδίου της δομής. Αφού ο ptr δείχνει στη διεύθυνση μνήμης της δομής stud, το *ptr είναι ισοδύναμο με τη δομή stud και με τον τελεστή . μπορούμε να αποκτήσουμε πρόσβαση στα πεδία της δομής. Η έκφραση *ptr πρέπει να περικλείεται σε παρενθέσεις, γιατί ο τελεστής . έχει μεγαλύτερη προτεραιότητα από τον τελεστή *

Ο Τελεστής ->

- Ένας εναλλακτικός τρόπος για να αποκτήσουμε πρόσβαση στα πεδία μίας δομής με χρήση δείκτη είναι χρησιμοποιώντας τον τελεστή -> αντί του τελεστή τελεία (.). Για παράδειγμα, δείτε πώς μπορούμε να αλλάξουμε το προηγούμενο πρόγραμμα:

```
int main() // Χρήση δείκτη για την προσπέλαση των πεδίων.
{
        Student *ptr, stud;

        ptr = &stud;
        ptr->name = "somebody";
        ptr->grd = 6.7;
        std::cout << ptr->name << ' ' << ptr->grd << '\n';
        return 0;
}
```

- Η έκφραση `ptr->name` είναι ισοδύναμη με `(*ptr).name` και η έκφραση `ptr->grd` ισοδύναμη με `(*ptr).grd`
- Μεταξύ των δύο τρόπων χρήσης του δείκτη για πρόσβαση στα πεδία μίας δομής, θεωρώ ότι είναι πιο εκφραστικό και απλό να χρησιμοποιούμε τον τελεστή ->

Πίνακας Δομών

- Ένας πίνακας δομών είναι ένας πίνακας με στοιχεία δομές
- Συνήθως, ένας πίνακας δομών χρησιμοποιείται σε εφαρμογές που απαιτείται αποθήκευση πληροφορίας για πολλές οντότητες, όπως μία εφαρμογή καταχώρησης των εμπορευμάτων μίας αποθήκης, των φοιτητών μίας σχολής ή των υπαλλήλων μίας εταιρείας
- Ουσιαστικά, ένας πίνακας δομών μπορεί να χρησιμοποιηθεί σαν μία απλή βάση δεδομένων. Π.χ. με την εντολή:

```
Student stud[100];
```

η μεταβλητή **stud** δηλώνεται σαν ένας πίνακας 100 στοιχείων, όπου κάθε στοιχείο του είναι μία δομή τύπου **Student**

Αρχικοποίηση Πίνακα Δομών (1)

- Ένας πίνακας δομών μπορεί να αρχικοποιηθεί μαζί με τη δήλωσή του. Π.χ. για τη δομή:

```
struct Student
{
        string name;
        int code;
        float grd;
};
```

με την παρακάτω αρχικοποίηση:

```
struct Student stud[3] = {{"nick stergiou", 150, 7.3},
                          {"john nikas", 160, 5.8},
                           {"peter karras", 170, 6.7}};
```

η τιμή του πεδίου `stud[0].name` γίνεται `"nick stergiou"`

η τιμή του πεδίου `stud[1].code` γίνεται `160`

η τιμή του πεδίου `stud[2].grd` γίνεται `6.7`

- Όπως συνήθως, αν παραληφθεί η διάσταση στις αγκύλες `[]`, ο μεταγλωττιστής θα υπολογίσει τον αριθμό των στοιχείων

Αρχικοποίηση Πίνακα Δομών (2)

- Επίσης, μπορούμε να δηλώσουμε και να αρχικοποιήσουμε έναν πίνακα δομών μαζί με τη δήλωση της δομής. Π.χ:

```
struct Student
{
   string name;
   int code;
   float grd;
} stud[] = {{"nick stergiou", 150, 7.3},
            {"john nikas", 160, 5.8},
                  {"peter karras", 170, 6.7}};
```

- Και με τους 2 τρόπους αρχικοποίησης, τα εσωτερικά άγκιστρα δεν είναι απαραίτητα, ωστόσο αν εισάγονται ξεχωρίζει η αρχικοποίηση της κάθε δομής

Παρατηρήσεις

- Όπως και με έναν απλό πίνακα, μπορούμε να χρησιμοποιήσουμε σημειογραφία δείκτη για να προσπελάσουμε τα στοιχεία του

  Π.χ. για την προηγούμενη δομή `stud` (τύπου `Student`) αφού το όνομα ενός πίνακα είναι δείκτης στη διεύθυνση του 1ου στοιχείου του, τότε:
  - το `*stud` είναι ισοδύναμο με `stud[0]`
  - το `*(stud+1)` είναι ισοδύναμο με `stud[1]`
  - το `*(stud+2)` είναι ισοδύναμο με `stud[2]` κ.ο.κ.

- Άρα, αν θέλουμε να προσπελάσουμε το πεδίο `grd` του τρίτου φοιτητή, οι εκφράσεις `stud[2].grd` και `(*(stud + 2)).grd` είναι ισοδύναμες (οι παρενθέσεις στη δεύτερη περίπτωση είναι απαραίτητες λόγω των προτεραιοτήτων)

- Όπως και στην περίπτωση των απλών πινάκων, είναι πιο βολικό να χρησιμοποιούμε τη θέση του στοιχείου και όχι σημειογραφία δείκτη, ώστε ο κώδικας να είναι πιο απλός και ευανάγνωστος

Παράδειγμα

- Το παρακάτω πρόγραμμα χρησιμοποιεί έναν επαναληπτικό βρόχο, για να αποθηκεύσει τα στοιχεία 100 φοιτητών σε έναν πίνακα δομών τύπου Student

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

const int SIZE = 100;

struct Student
{
   string name;
   int code;
   float grd;
};
int main()
{
          int i;
          Student stud[SIZE];
          for(i = 0; i < SIZE; i++)
          {
                    cout << "\nEnter name: ";
                    getline(cin, stud[i].name);

                    cout << "Enter code: ";
                    cin >> stud[i].code;

                    cout << "Enter grade: ";
                    cin >> stud[i].grd;

                  cout << stud[i].name << ' ' << stud[i].code << ' ' << stud[i].grd << '\n';
                    cin.get(); /* Διαβάζουμε τον χαρακτήρα αλλαγής γραμμής που αποθηκεύτηκε
στο cin μετά την εισαγωγή του βαθμού. */
          }
          return 0;
}
```

Παράδειγμα

- Και όπως ξέρουμε αντί για έναν απλό πίνακα μπορούμε να χρησιμοποιήσουμε ένα **vector** αντικείμενο. Π.χ.

```
int main()
{
        vector<Student> stud(SIZE);
        ... // Ο υπόλοιπος κώδικας είναι ίδιος.
}
```

Συνάρτηση με Παράμετρο Δομή

- Μία δομή μπορεί να μεταβιβαστεί σε μία συνάρτηση όπως οποιαδήποτε άλλη μεταβλητή, δηλαδή **είτε η ίδια η δομή είτε η διεύθυνση μνήμης της δομής**, ενώ επίσης, μία συνάρτηση μπορεί να επιστρέφει δομή
- Υπενθυμίζεται ότι, όταν μεταβιβάζεται **η τιμή** μιας παραμέτρου σε συνάρτηση, τότε στη συνάρτηση διοχετεύονται **αντίγραφα** των παραμέτρων του προγράμματος που την καλεί, επομένως, οποιαδήποτε αλλαγή γίνει στις τιμές των πεδίων της δομής μέσα στη συνάρτηση **δεν επηρεάζει** τις αντίστοιχες τιμές των πεδίων της δομής που διοχετεύθηκε στη συνάρτηση, γιατί οι τυχόν αλλαγές γίνονται σε αντίγραφό της
- **Αντίθετα**, σε περίπτωση που μεταβιβάζεται **η διεύθυνση** της παραμέτρου σε συνάρτηση, στη συνάρτηση διοχετεύονται **οι διευθύνσεις μνήμης** των παραμέτρων του προγράμματος που την καλεί και όχι αντίγραφά τους, όπως προηγουμένως, επομένως, αφού η συνάρτηση έχει πρόσβαση στη διεύθυνση της δομής του προγράμματος που την κάλεσε, τότε **μπορεί να μεταβάλλει** τις τιμές των πεδίων της

Παράδειγμα Μεταβίβασης Δομής

```
#include <iostream>
#include <string>

struct Student
{
   std::string name;
   int code;
   float grd;
};

Student test(Student s);

int main()
{
          Student st = {"somebody", 20, 5};
          st = test(st); /* Σύμφωνα με τη δήλωση της συνάρτησης, πρέπει να κληθεί με όρισμα
μία δομή τύπου Student. */
          std::cout << st.name << ' ' << st.code << ' ' << st.grd << '\n';
          return 0;
}

Student test(Student s)
{
          s.name = "new";
          s.code = 30;
          s.grd = 7;
          return s;
}
```

- Όταν καλείται η `test()` οι τιμές των πεδίων της `st` αντιγράφονται στα αντίστοιχα πεδία της `s`. Αφού οι μεταβλητές `s` και `st` αποθηκεύονται σε διαφορετικές θέσεις μνήμης, οποιαδήποτε αλλαγή γίνει στα πεδία της `s` δεν επηρεάζει τις αντίστοιχες τιμές των πεδίων της `st`. Μετά, η δομή που επιστρέφει η `test()` αντιγράφεται στην `st`. Έτσι, το πρόγραμμα εμφανίζει:
  `new 30 7`

Παράδειγμα Μεταβίβασης Διεύθυνσης Δομής

- Αντίθετα, αν μεταβιβάσουμε τη διεύθυνση μνήμης της δομής η συνάρτηση μπορεί να αλλάξει τις τιμές των πεδίων της. Για παράδειγμα, ας τροποποιήσουμε την `test()`, ώστε να μπορεί να αλλάξει τις τιμές των πεδίων της δομής `st`

```
void test(Student *ptr);

int main()
{
        Student st = {"somebody", 20, 5};
        test(&st);
        std::cout << st.name << ' ' << st.code << ' ' << st.grd << '\n';
        return 0;
}

void test(Student *ptr) /* Τώρα, μπορούμε να αλλάξουμε τα πεδία της
δομής. */
{
        ptr->name = "new";
        ptr->code = 30;
        ptr->grd = 7;
}
```

- Όταν καλείται η `test()` έχουμε `ptr = &st`. Άρα, αφού ο `ptr` δείχνει στη διεύθυνση μνήμης της `st`, η συνάρτηση μπορεί να αλλάξει τις τιμές των πεδίων της. Επομένως, ο κώδικας εμφανίζει: `new 30 7`

Παρατηρήσεις

- Όταν μία δομή μεταβιβάζεται σε μία συνάρτηση γίνεται αντιγραφή των πεδίων της στα πεδία της αντίστοιχης παραμέτρου της συνάρτησης. Σημειώστε ότι αν το μέγεθος της δομής είναι μεγάλο ή αν η συνάρτηση καλείται αρκετές φορές, αυτή η διαδικασία αντιγραφής μπορεί να προσθέσει κάποια χρονική καθυστέρηση στην εκτέλεση του προγράμματος

- Αντίθετα, όταν μεταβιβάζεται η διεύθυνση της δομής, μέσω δείκτη ή αναφοράς, δεν γίνεται αντιγραφή πεδίων

- Για καλύτερη απόδοση, να μεταβιβάζετε σε μία συνάρτηση τη διεύθυνση της δομής, ακόμα και αν η συνάρτηση δεν πρόκειται να αλλάξει τις τιμές των πεδίων της

- Αν δεν θέλετε η συνάρτηση να μπορεί να αλλάξει τις τιμές των πεδίων της δομής, να δηλώνετε τον δείκτη ή την αναφορά ως `const`. Π.χ. `void test(const Student *ptr);`

Ενώσεις (Unions)

- Όπως και η δομή, μία **ένωση** αποτελείται από ένα ή περισσότερα πεδία που μπορεί να έχουν διαφορετικούς τύπους
- Οι ιδιότητες της ένωσης είναι παρόμοιες με αυτές των δομών με τις ίδιες λειτουργίες να επιτρέπονται όπως και στις δομές
- Η διαφορά τους είναι ότι τα πεδία μίας δομής αποθηκεύονται σε διαφορετικές διευθύνσεις μνήμης, ενώ τα πεδία μίας ένωσης αποθηκεύονται στην ίδια διεύθυνση μνήμης
- Αφού ο μεταγλωττιστής δεν δεσμεύει ξεχωριστή μνήμη για κάθε πεδίο, ο κύριος λόγος χρήσης μίας ένωσης είναι η εξοικονόμηση μνήμης, όπως για παράδειγμα σε εφαρμογές ενσωματωμένων συστημάτων, όπου η μνήμη είναι περιορισμένη

Δήλωση Ένωσης

- Μία ένωση δηλώνεται όπως και μία δομή, με τη διαφορά ότι, αντί της λέξης `struct` χρησιμοποιείται η λέξη `union`
- Όταν δηλώνεται μία ένωση, ο μεταγλωττιστής δεσμεύει μνήμη για την αποθήκευση του **μεγαλύτερου** πεδίου της, και όχι για όλα τα πεδία της ένωσης
- Επομένως, τα πεδία μίας ένωσης αποθηκεύονται σε έναν **κοινό** χώρο μνήμης και όχι σε ξεχωριστή μνήμη, όπως στην περίπτωση μίας δομής

Παράδειγμα

- Στο επόμενο πρόγραμμα η μνήμη που δεσμεύεται για τη μεταβλητή s είναι οκτώ οκτάδες, αφού το μεγαλύτερο πεδίο της είναι τύπου double

```
#include <iostream>

union Sample
{
  char ch;
  int i;
  double d;
};

int main()
{
        Sample s;
        std::cout << sizeof(s) << '\n';
        return 0;
}
```

Πρόσβαση Πεδίων Ένωσης

- Τα πεδία μίας ένωσης μπορούμε να τα προσπελάσουμε με τους ίδιους τρόπους που προσπελαύνουμε τα πεδία μίας δομής
- Ωστόσο, επειδή όλα τα πεδία αποθηκεύονται στην **ίδια** μνήμη, μόνο το **τελευταίο** πεδίο στο οποίο εκχωρήθηκε μία τιμή έχει έγκυρη τιμή
- Όταν γίνεται ανάθεση τιμής σε ένα πεδίο μίας ένωσης, η τιμή που τελευταία είχε αποθηκευτεί σε κάποιο άλλο πεδίο υπερεγγράφεται

Παράδειγμα

- Το επόμενο πρόγραμμα θέτει μία τιμή σε κάποιο πεδίο της ένωσης **s** και εμφανίζει τις τιμές των υπολοίπων πεδίων

```
#include <iostream>

union Sample
{
   char ch;
   int i;
   double d;
};

int main()
{
          Sample s;

          s.ch = 'a';
          std::cout << s.ch << ' ' << s.i << ' ' << s.d << '\n';

          s.i = 64;
          std::cout << s.ch << ' ' << s.i << ' ' << s.d << '\n';

          s.d = 12.48;
          std::cout << s.ch << ' ' << s.i << ' ' << s.d << '\n';
          return 0;
}
```

- Αρχικά, το πρόγραμμα εμφανίζει **a** και χωρίς νόημα τιμές για τα **s.i** και **s.d**. Μετά, εκχωρείται η τιμή **64** στο πεδίο **i**. Αφού αυτή η τιμή αποθηκεύεται στον κοινό χώρο μνήμης, η τιμή του **s.ch** υπερεγγράφεται. Άρα, εμφανίζεται **64** και χωρίς νόημα τιμές για τα **s.ch** και **s.d**. Τέλος, εκχωρείται η τιμή **12.48** στο πεδίο **d**. Η τιμή του **s.i** υπερεγγράφεται και το πρόγραμμα εμφανίζει **12.48** και χωρίς νόημα τιμές για τα **s.ch** και **s.i**

C++: Από τη Θεωρία στην Εφαρμογή

Κεφάλαιο 14ο
Διαχείριση Μνήμης και Δομές Δεδομένων

Διαχείριση Μνήμης

- Η διαχείριση της μνήμης αφορά κυρίως τις διαδικασίες **δέσμευσης** της μνήμης και **αποδέσμευσης** αυτής (όταν πλέον δεν μας χρειάζεται)

- Η δέσμευση μνήμης μπορεί να γίνει με δύο τρόπους:
  - είτε **στατικά**
  - είτε **δυναμικά**

- Στα προγράμματα μέχρι τώρα, έχουμε χρησιμοποιήσει μόνο τον στατικό τρόπο δέσμευσης μνήμης. Ένα παράδειγμα στατικής δέσμευσης είναι ο πίνακας, όπου τα στοιχεία του αποθηκεύονται σε μνήμη που το μέγεθός της δεν μπορεί να μεταβληθεί

- Σε αυτή την ενότητα, θα δούμε με ποιον τρόπο μπορούμε να δεσμεύουμε δυναμικά μνήμη κατά την εκτέλεση του προγράμματος (run time) για τη δημιουργία δυναμικών δομών δεδομένων, όπως στοίβες, ουρές, συνδεδεμένες λίστες και δέντρα

Κατανομή Μνήμης

- Όταν ένα πρόγραμμα εκτελείται, ζητάει πόρους μνήμης από το λειτουργικό σύστημα του υπολογιστή και, συνήθως, η μνήμη που αποδίδεται χωρίζεται στα ακόλουθα τέσσερα τμήματα:

1) Στο τμήμα κώδικα (*code segment*), το οποίο χρησιμοποιείται για την αποθήκευση του μεταγλωττισμένου κώδικα του προγράμματος

2) Στο τμήμα δεδομένων (*data segment*), το οποίο χρησιμοποιείται για την αποθήκευση των καθολικών και `static` μεταβλητών του προγράμματος

3) Στο τμήμα που ονομάζεται στοίβα (*stack segment*), το οποίο χρησιμοποιείται για την αποθήκευση των δεδομένων των συναρτήσεων (π.χ. τοπικές μεταβλητές)

4) Στο τμήμα που ονομάζεται σωρός (*heap*), το οποίο χρησιμοποιείται για δυναμική δέσμευση μνήμης

Παρατηρήσεις

- Σημειώστε ότι το παραπάνω μοντέλο κατανομής μνήμης αποτελεί μία συνήθης επιλογή, το κάθε σύστημα μπορεί να καθορίσει το δικό του μοντέλο

- Ο κάθε μεταγλωττιστής μπορεί να εφαρμόσει τη δική του πολιτική βελτιστοποίησης, π.χ., οι παράμετροι μίας συνάρτησης μπορεί να μην αποθηκεύονται στη στοίβα, αλλά σε καταχωρητές (registers) του συστήματος, για πιο γρήγορη πρόσβαση

Στατική Δέσμευση Μνήμης (1)

- Με τη στατική δέσμευση, η μνήμη δεσμεύεται από τη στοίβα
- Το μέγεθος της μνήμης που θα δεσμευτεί πρέπει να είναι γνωστό κατά τη συγγραφή του προγράμματος και δεν μπορεί να αλλάξει κατά τη διάρκεια εκτέλεσής του
- Για παράδειγμα, με τη δήλωση:

```
float grades[1000];
```

ο μεταγλωττιστής, όταν μεταγλωττίζει το πρόγραμμα, δεσμεύει στατικά μνήμη `1000×sizeof(float)` bytes για την αποθήκευση των βαθμών 1000 φοιτητών

- Το μέγεθος του πίνακα `grades` δεν μπορεί να αλλάξει κατά τη διάρκεια εκτέλεσης του προγράμματος, ακόμα και αν χρειαστεί να αποθηκευτούν βαθμοί για περισσότερους φοιτητές
- Επίσης, αν οι φοιτητές είναι λιγότεροι από 1000, τότε γίνεται σπατάλη μνήμης, αφού δεσμεύεται περισσότερη απ' την απαιτούμενη
- Ο μοναδικός τρόπος για να αλλάξει το μέγεθος του πίνακα είναι να ξαναγραφεί το πρόγραμμα και να γίνει νέα μεταγλώττιση

Στατική Δέσμευση Μνήμης (2)

Τι συμβαίνει όταν γίνεται κλήση μίας συνάρτησης;

- Όταν καλείται μία συνάρτηση, ο μεταγλωττιστής δεσμεύει μνήμη στη στοίβα για να αποθηκευτούν τα δεδομένα της συνάρτησης, θεωρώντας – για απλότητα – ότι δεν χρησιμοποιούνται και καταχωρητές του συστήματος, παρόλο που κάτι τέτοιο είναι δυνατό
- Π.χ., όταν καλείται μία συνάρτηση που επιστρέφει τιμή, δέχεται παραμέτρους και χρησιμοποιεί τοπικές μεταβλητές, ο μεταγλωττιστής δεσμεύει μνήμη για να αποθηκεύσει:
  - τις τιμές των παραμέτρων
  - τις τοπικές μεταβλητές
  - την τιμή επιστροφής
  - τη διεύθυνση μνήμης της εντολής που θα εκτελεστεί μετά τον τερματισμό της συνάρτησης

Στατική Δέσμευση Μνήμης (3)

Τι συμβαίνει όταν τερματίζει μία συνάρτηση;

- Όταν τερματίσει η συνάρτηση (έστω η συνάρτηση του περιγράφηκε προηγουμένως) συμβαίνουν τα ακόλουθα:
  - Αν η τιμή επιστροφής της συνάρτησης εκχωρείται σε κάποια μεταβλητή, εξάγεται από τη στοίβα και αποθηκεύεται σε αυτήν
  - Η διεύθυνση μνήμης της επόμενης εντολής εξάγεται επίσης από τη στοίβα, ώστε το πρόγραμμα να συνεχίσει με την εκτέλεσή της
  - Η μνήμη που δεσμεύτηκε για την αποθήκευση των δεδομένων της συνάρτησης αποδεσμεύεται, εκτός από τη μνήμη για τις στατικές μεταβλητές

Παράδειγμα

```
#include <iostream>

void test(int i, int j);

int main()
{
    float a[500], b[10];
    test(10, 20);
    return 0;
}

void test(int i, int j)
{
    int arr[200];
}
```

- Όταν καλείται η `test()` ο μεταγλωττιστής δεσμεύει `202×sizeof(int)` bytes στη στοίβα, για να αποθηκευτούν οι τιμές των παραμέτρων `i` και `j` και των στοιχείων του πίνακα `arr`
- Όταν τερματίζει η εκτέλεση της `test()`, η μνήμη αυτή αποδεσμεύεται
- Παρομοίως, η μνήμη των `510×sizeof(float)` bytes που έχει δεσμευτεί για τις τοπικές μεταβλητές της `main()` αποδεσμεύεται, όταν τερματιστεί η εκτέλεση του προγράμματος

Παρατηρήσεις

- Αν δεν υπάρχει διαθέσιμος χώρος στη στοίβα για την αποθήκευση των δεδομένων μίας συνάρτησης, η εκτέλεση του προγράμματος θα τερματιστεί ανώμαλα και είναι πιθανό να εμφανιστεί το μήνυμα `"Stack overflow"` *(υπερχείλιση στοίβας)*

- Κάτι τέτοιο μπορεί να συμβεί αν μία συνάρτηση δεσμεύει αρκετή μνήμη και καλεί άλλες ένθετες συναρτήσεις, οι οποίες επίσης δεσμεύουν αρκετή μνήμη

- Για παράδειγμα, μία αναδρομική συνάρτηση που καλεί πολλές φορές τον εαυτό της μπορεί επίσης να οδηγήσει σε εξάντληση της διαθέσιμης μνήμης της στοίβας

Δυναμική Δέσμευση Μνήμης (1)

- Με τη δυναμική δέσμευση, η μνήμη δεσμεύεται από τον σωρό (heap) δυναμικά, δηλαδή κατά την εκτέλεση του προγράμματος
- Σε αντίθεση με τη στατική δέσμευση, το μέγεθος της μνήμης που θα δεσμευτεί δεν χρειάζεται να είναι γνωστό πριν την εκτέλεση του προγράμματος (δηλ. κατά τη μεταγλώττισή του), αλλά μπορεί να καθοριστεί δυναμικά κατά την εκτέλεσή του
- Επίσης, το μέγεθος της μνήμης που δεσμεύεται με δυναμικό τρόπο μπορεί να μεγαλώσει ή να μικρύνει δυναμικά, ανάλογα με τις απαιτήσεις του προγράμματος
- Για παράδειγμα, η κλάση `vector` δεσμεύει δυναμικά τη μνήμη που χρειάζεται για να αποθηκεύσει τα στοιχεία της
- Η δυναμική δέσμευση είναι πολύ συνηθισμένη σε εφαρμογές όπου ο χρήστης επικοινωνεί με το πρόγραμμα. Για παράδειγμα, όταν το πρόγραμμα δεν γνωρίζει τον αριθμό των φοιτητών, να ζητάει από τον χρήστη να τον εισάγει, και μετά να δεσμεύει μνήμη για έναν πίνακα δομών, στον οποίο να αποθηκεύονται τα στοιχεία τους

Δυναμική Δέσμευση Μνήμης (2)

- Όπως είπαμε, με την στατική δέσμευση η μνήμη δεσμεύεται από την στοίβα. Συνήθως, το προκαθορισμένο μέγεθος της στοίβας δεν είναι μεγάλο. Για παράδειγμα, το επόμενο πρόγραμμα μπορεί να μην εκτελεστεί λόγω έλλειψης διαθέσιμης μνήμης στη στοίβα

```
#include <iostream>
int main()
{
        int arr[10000000]; // Στατική δέσμευση μνήμης
        return 0;
}
```

- Αντιθέτως, το μέγεθος του σωρού είναι συνήθως αρκετά μεγαλύτερο από αυτό της στοίβας. Για παράδειγμα, αν στο προηγούμενο πρόγραμμα γινόταν δυναμική δέσμευση μνήμης (όπως παρακάτω) η απαιτούμενη μνήμη θα δεσμευτεί από τον σωρό και το πρόγραμμα θα εκτελεστεί χωρίς πρόβλημα

```
#include <iostream>
int main()
{
        int *arr;
        arr = new int[10000000]; // Δυναμική δέσμευση μνήμης
        delete[] arr; // Αποδέσμευση μνήμης
        return 0;
}
```

- Η διαχείριση δυναμικά δεσμευμένης μνήμης συχνά προκαλεί την εμφάνιση σοβαρών λαθών κατά την εκτέλεση του προγράμματος. Για παράδειγμα, όπως θα δούμε στη συνέχεια, η διαρροή μνήμης και η προσπέλαση μνήμης που έχει ήδη αποδεσμευτεί αποτελούν πολύ συνηθισμένα **λάθη**. Χρειάζεται λοιπόν ιδιαίτερη προσοχή στον τρόπο χρήσης της.

Ο Τελεστής new

- Με τον τελεστή **new** επιλέγουμε τον τύπο για τον οποίον θέλουμε να δεσμεύσουμε μνήμη, ο new βρίσκει την αντίστοιχη μνήμη, τη δεσμεύει, επιστρέφει τη διεύθυνση της, την οποία και εκχωρούμε σε αντίστοιχο δείκτη για να τη διαχειριστούμε

- Για παράδειγμα, ο παρακάτω κώδικας δεσμεύει μνήμη για έναν ακέραιο, διαβάζει μία τιμή και την αποθηκεύει στη μνήμη:

```
int *p;
p = new int;
cin >> *p; /* Η τιμή του *p θα γίνει ίση με την τιμή
που θα εισάγει ο χρήστης. */
```

- Γενικά, δεν υπάρχει λόγος να χρησιμοποιήσουμε τον τελεστή **new** για να δεσμεύσουμε ένα στοιχείο (π.χ. έναν ακέραιο) αφού μπορούμε να δηλώσουμε μία αντίστοιχη μεταβλητή (π.χ. **int** i) και να χρησιμοποιήσουμε τη μεταβλητή

Ο Τελεστής new[]

- Για τη δέσμευση μνήμης για πολλά στοιχεία χρησιμοποιούμε τον τελεστή new[], όπου ο αριθμός των στοιχείων δηλώνεται μετά τον τύπο τους μέσα σε αγκύλες
- Αυτός ο αριθμός μπορεί να καθοριστεί και κατά την εκτέλεση του προγράμματος. Π.χ:

```
int num;
cin >> num;
int *p = new int[num];
```

- Ο τελεστής new[] δεσμεύει ένα τμήμα μνήμης για num ακεραίους και, αν η δέσμευση είναι επιτυχημένη, ο p δείχνει στην αρχή αυτής της μνήμης. Ο αριθμός στις [] πρέπει να είναι ακέραιος
- Οι αρχικές τιμές των στοιχείων είναι τυχαίες. Αν θέλουμε να τις αρχικοποιήσουμε με 0 προσθέτουμε κενές παρενθέσεις ή άγκιστρα. Π.χ. `int *p = new int[100]();`
- Με τη C++11 μπορούμε να προσδιορίσουμε μία λίστα αρχικών τιμών. Π.χ: `int *p = new int[100]{10, 20, 30};` Τα τρία πρώτα στοιχεία αρχικοποιούνται με τις αντίστοιχες τιμές και τα υπόλοιπα με 0
- Παρόμοια, για να δεσμεύσουμε μνήμη για έναν πίνακα 100 δομών τύπου Student γράφουμε: `Student *p = new Student[100];`

Παρατηρήσεις (1)

- Σημειώστε ότι ο μεταγλωττιστής δεν μας υποχρεώνει να προσπελαύνουμε με τον δείκτη μόνο τα στοιχεία για τα οποία έχει δεσμευτεί μνήμη
- Π.χ., αν ο δείκτης `p` δείχνει σε μία μνήμη που δεσμεύσαμε για 100 ακεραίους ο μεταγλωττιστής μας επιτρέπει να γράψουμε:

```
p[200] = 10;  ή ακόμα και p[-10] = 20;
```

και να προσπελάσουμε μνήμη εκτός του έγκυρου εύρους με καταστροφικές συνέπειες για τη λειτουργία του προγράμματος. Ο μεταγλωττιστής δεν θα απαγορεύσει αυτήν την ενέργεια. Μας εμπιστεύεται ότι ξέρουμε τι κάνουμε και θα μας αφήσει να προσπελάσουμε αυτή τη μνήμη

- Αυτό το είδος σφάλματος είναι ιδιαίτερα δύσκολο να βρεθεί και το χειρότερο είναι ότι, κάθε φορά που εκτελείται το πρόγραμμα, μπορεί να έχει διαφορετική λανθασμένη συμπεριφορά, γεγονός που καθιστά ακόμα πιο δύσκολο τον εντοπισμό του σφάλματος

Παρατηρήσεις (2)

- Μην ξεχνάμε, ότι όπως έχουμε δει σε παραδείγματα, μπορούμε αντί για `new[]` να χρησιμοποιήσουμε ένα `vector` αντικείμενο. Τότε, μπορούμε να προσπελάσουμε με ασφάλεια τα στοιχεία του διανύσματος και να μην ασχολούμαστε με την δέσμευση και αποδέσμευση μνήμης. Π.χ:

```
int num;
cin >> num;
vector<int> v(num);
```

Παρατηρήσεις (3)

- Είναι προτιμότερο να μην δεσμεύετε δυναμικά μνήμη για τη δημιουργία τοπικών μεταβλητών, εφόσον μπορείτε να το κάνετε με στατικό τρόπο. Π.χ. ο παρακάτω κώδικας είναι επιρρεπής σε σφάλματα:

```
void f()
{
        int *p = new int[1000];
        ...
        delete[] p;
}
```

- Π.χ., προσθέτοντας ο προγραμματιστής μία εντολή `return` και ξεχνώντας να καλέσει πρώτα την `delete[]` για να αποδεσμεύσει τη μνήμη, αν εκτελεστεί αυτή η `return`, θα προκληθεί διαρροή μνήμης
- Είναι πολύ πιο ασφαλές να χρησιμοποιήσετε μία συνηθισμένη μεταβλητή (π.χ. `int p[1000]`), για την οποία η μνήμη θα αποδεσμευτεί αυτόματα όταν τερματιστεί η συνάρτηση
- Γενικά, σε ένα πρόγραμμα που εκτελείται για ένα μεγάλο χρονικό διάστημα ή και για πάντα, οι επαναλαμβανόμενες διαρροές μνήμης μπορεί να προκαλέσουν σοβαρή υποβάθμιση της απόδοσης και πιθανώς την κατάρρευση του προγράμματος

Ο Τελεστής delete

- Για να αποδεσμεύσουμε τη μνήμη που έχει δεσμευτεί για ένα στοιχείο με τον τελεστή new χρησιμοποιούμε τον τελεστή delete. Π.χ:

```
int *p = new int;
...
delete p;
```

- Η απόπειρα αποδέσμευσης μνήμης που έχει ήδη αποδεσμευτεί ή η προσπέλαση μνήμης που έχει ήδη αποδεσμευτεί είναι λανθασμένη ενέργεια. Π.χ:

```
int *p = new int;
...
delete p;
*p = 10; // Λάθος
delete p; // Λάθος
```

- Αν ο δείκτης είναι μηδενικός, είναι ασφαλές να χρησιμοποιήσουμε τον delete. Αφού ο μηδενικός δείκτης δεν δείχνει κάπου, η εντολή δεν έχει καμία επίδραση, τίποτα δεν θα συμβεί. Π.χ:

```
int *p = nullptr;
delete p; // Σωστό
```

Ο Τελεστής delete[]

- Για να αποδεσμεύσουμε τη μνήμη που έχει δεσμευτεί για πολλά στοιχεία με τον τελεστή new[] χρησιμοποιούμε τον τελεστή delete[]. Π.χ:

```
Student *p = new Student[100];
...
delete[] p;
```

- Οι αγκύλες υποδεικνύουν στον μεταγλωττιστή ότι πρέπει να αποδεσμευτεί όλη η μνήμη και όχι μόνο η μνήμη για το στοιχείο που δείχνει ο δείκτης. Αν ξεχάσουμε τις αγκύλες η συμπεριφορά του προγράμματος είναι απροσδιόριστη
- Θυμόμαστε, χρησιμοποιούμε τον delete για αποδέσμευση μνήμης που έχει δεσμευτεί με τον new και τον delete[] για αποδέσμευση μνήμης που έχει δεσμευτεί με τον new[]

Παρατηρήσεις

- Προσέξτε ότι όταν αποδεσμεύετε την μνήμη δεν σημαίνει ότι ο δείκτης θα γίνει μηδενικός. Για παράδειγμα, η τιμή του μπορεί να είναι ακόμα ίση με τη διεύθυνση της δεσμευμένης μνήμης. Γενικά, ιδιαίτερα σε μεγάλες εφαρμογές, όπου η ίδια μνήμη μπορεί να χρησιμοποιείται σε διαφορετικά σημεία του προγράμματος, συστήνεται, για μεγαλύτερη ασφάλεια και καλύτερο έλεγχο του προγράμματος, να εκχωρείτε την τιμή `nullptr` στον αντίστοιχο δείκτη όταν αποδεσμεύετε την μνήμη. Για παράδειγμα:

```
delete[] p; /* Αν είναι μηδενικός δείκτης δεν υπάρχει
πρόβλημα. */
p = nullptr;
```

Επίσης, αυτή η πρακτική είναι πολύ βοηθητική για την αποσφαλμάτωση προγραμμάτων με προβλήματα στη διαχείριση μνήμης

Διαρροές Μνήμης (1)

- Όταν δεν χρειάζεστε άλλο τη δεσμευμένη μνήμη, μην ξεχνάτε να την απελευθερώσετε, ώστε να μη δημιουργούνται διαρροές μνήμης που μπορεί να οδηγήσουν στην εξάντληση της μνήμης
- Ας δούμε ένα παράδειγμα διαρροής μνήμης:

```
void f()
{
        int *ptr = new int[10000];
        ...
        ptr = new int[200];
        ...
        delete[] ptr;
}
```

Σε αυτό το παράδειγμα ξεχνάμε να αποδεσμεύσουμε το πρώτο τμήμα μνήμης πριν από τη νέα δέσμευση. Αυτή είναι μια διαρροή μνήμης, η οποία προκαλείται κάθε φορά που καλείται η `f()`

- Όταν δεν χρειάζεστε άλλο τη δεσμευμένη μνήμη, μην ξεχνάτε να την απελευθερώσετε, ώστε να μη δημιουργούνται διαρροές μνήμης που μπορεί να οδηγήσουν στην εξάντληση της μνήμης

Διαρροές Μνήμης (2)

- Μία ακόμα συνηθισμένη περίπτωση διαρροής μνήμης συμβαίνει όταν εκχωρούμε ένα δείκτη σε κάποιον άλλο, ξεχνώντας όμως να αποδεσμεύσουμε τη μνήμη στην οποία αρχικά έδειχνε ο δείκτης. Για παράδειγμα, βρείτε τα λάθη στο παρακάτω πρόγραμμα και να τα διορθώσετε:

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

        p1 = new char[10];
        p2 = new char[10];
        p1 = p2;
        delete[] p1;

        p2[0] = 'a';
        delete[] p2;
        return 0;
}
```

Διαρροές Μνήμης (2)

- Ας δούμε τα προβλήματα:

α) Με την εντολή `p1 = p2;` οι δείκτες `p1` και `p2` δείχνουν στην ίδια μνήμη. Η μνήμη αυτή αποδεσμεύεται με τη `delete[] p1;`, επομένως, η επόμενη εντολή θέτει τιμή σε μη δεσμευμένη μνήμη.

β) Η εντολή `delete[] p2;` αποδεσμεύει μνήμη που έχει ήδη αποδεσμευτεί

γ) Η αρχική μνήμη στην οποία έδειχνε ο δείκτης `p1` δεν αποδεσμεύτηκε ποτέ

Αν βάλουμε την εντολή `delete[] p1;` πριν από την `p1 = p2;` το πρόγραμμα λειτουργεί κανονικά

Συναρτήσεις Διαχείρισης Μνήμης

- Στη συνέχεια, θα περιγράψουμε συνοπτικά τις συναρτήσεις βιβλιοθήκης `memcpy()`, `memmove()` και `memcmp()` που χρησιμοποιούνται αρκετά συχνά για τη διαχείριση μνήμης

Η Συνάρτηση memcpy()

- Η συνάρτηση `memcpy()` χρησιμοποιείται για την αντιγραφή οποιουδήποτε τύπου δεδομένων από μία περιοχή μνήμης σε μία άλλη
- Το πρωτότυπό της δηλώνεται στο αρχείο `cstring` ως εξής:

```
void *memcpy(void *dest, const void *src, size_t size);
```

- Η `memcpy()` αντιγράφει `size` bytes από την περιοχή μνήμης στην οποία δείχνει ο δείκτης `src` στην περιοχή μνήμης στην οποία δείχνει ο δείκτης `dest`
- Αν οι περιοχές μνημών επικαλύπτονται, η συμπεριφορά της συνάρτησης είναι ακαθόριστη

Παρατηρήσεις

- Όταν η `memcpy()` χρησιμοποιείται για την αντιγραφή αλφαριθμητικών μοιάζει με τη `strcpy()` με την εξής όμως διαφορά:
  - Με την `strcpy()` η αντιγραφή αλφαριθμητικών ολοκληρώνεται όταν συναντηθεί ο τερματικός χαρακτήρας (`'\0'`)
  - Αντίθετα, η `memcpy()` συνεχίζει μέχρι να αντιγραφεί ο αριθμός των καθορισμένων οκτάδων

Π.χ. αν έχουμε:

```
char str2[6] = {0}, str1[] = {'a', 'b', 'c', '\0', 'd', 'e'};
```

και γράψουμε: `memcpy(str2, str1, 6);`

το περιεχόμενο του πίνακα `str2` θα γίνει το ίδιο με του `str1`

Αντίθετα, αν γράψουμε: `strcpy(str2, str1);`

το περιεχόμενο του `str2` θα γίνει ίσο με

`{'a', 'b', 'c', '\0', '\0', '\0'}`

Η Συνάρτηση memmove()

- Η συνάρτηση memmove() είναι παρόμοια με την memcpy(), με τη διαφορά ότι η memmove() εξασφαλίζει τη σωστή αντιγραφή των δεδομένων, ακόμα και αν οι δύο περιοχές μνήμης επικαλύπτονται

- Επειδή ακριβώς η memcpy() δεν ελέγχει αν οι δύο περιοχές επικαλύπτονται, εκτελείται πιο γρήγορα από τη memmove()

Παρατηρήσεις

- Κατά τη χρήση της **memcpy()** ή της **memmove()**, για τη μνήμη προορισμού **πρέπει** να έχουν δεσμευτεί τουλάχιστον **size** bytes, σε διαφορετική περίπτωση, τα πλεονάζοντα bytes θα εγγραφούν σε **μη δεσμευμένη μνήμη** και τα δεδομένα εκεί θα υπερεγγραφούν
- Π.χ. η επόμενη αντιγραφή **δεν είναι σωστή**, γιατί το μέγεθος της μνήμης προορισμού είναι 3 bytes, ενώ τα bytes που θα αντιγραφούν είναι 6

```
char str1[3], str2[] = "abcde";
memcpy(str1, str2, sizeof(str2));
```

- Η **memcpy()** είναι πολύ **χρήσιμη**, γιατί συνήθως υλοποιείται με τέτοιο τρόπο ώστε η αντιγραφή μεγάλου όγκου δεδομένων από μία περιοχή μνήμης σε μία άλλη να ολοκληρώνεται πιο γρήγορα από ότι με έναν επαναληπτικό βρόχο
- Για παράδειγμα, ο επόμενος κώδικας χρησιμοποιεί τη **memcpy()** για την αντιγραφή ενός πίνακα σε έναν άλλο:

```
int arr1[SIZE], arr2[SIZE];
...
memcpy(arr2, arr1, sizeof(arr1)); /* Χρησιμοποιούμε τη
memcpy() αντί για έναν επαναληπτικό βρόχο, όπως:
for(int i = 0; i < SIZE; i++)
        arr2[i] = arr1[i]; */
```

Η Συνάρτηση memcmp()

- Η συνάρτηση memcmp() χρησιμοποιείται για τη σύγκριση των δεδομένων που περιέχονται σε μία περιοχή μνήμης με τα δεδομένα που περιέχονται σε μία άλλη περιοχή μνήμης

- Το πρωτότυπό της δηλώνεται στο αρχείο cstring ως εξής:

```
int memcmp(const void *ptr1, const void *ptr2, size_t size);
```

- Η memcmp() συγκρίνει size bytes από την περιοχή μνήμης στην οποία δείχνει ο δείκτης ptr1 με τα αντίστοιχα bytes που περιέχονται στην περιοχή μνήμης στην οποία δείχνει ο δείκτης ptr2

- Αν οι δύο περιοχές μνήμης περιέχουν τα ίδια δεδομένα, τότε η συνάρτηση memcmp() επιστρέφει 0, αλλιώς μια μη μηδενική τιμή, όπως κάνει και η strcmp()

Παρατηρήσεις

- Όταν η `memcmp()` χρησιμοποιείται για τη σύγκριση αλφαριθμητικών μοιάζει με τη `strcmp()` με την εξής όμως διαφορά:
  - Με την `strcmp()` η σύγκριση αλφαριθμητικών ολοκληρώνεται όταν συναντηθεί ο τερματικός χαρακτήρας (`'\0'`)
  - Αντίθετα, η `memcmp()` δεν σταματάει τη σύγκριση όταν συναντηθεί ο τερματικός χαρακτήρας (`'\0'`)

Παράδειγμα (1)

```
#include <iostream>
#include <cstring>
using std::cout;
int main()
{
    char str1[] = {'a', 'b', 'c', '\0', 'd', 'e'};
    char str2[] = {'a', 'b', 'c', '\0', 'd', 'f'};

    if(strcmp(str1, str2) == 0)
            cout << "Same\n";
    else
            cout << "Different\n";

    if(memcmp(str1, str2, sizeof(str1)) == 0)
            cout << "Same\n";
    else
            cout << "Different\n";
    return 0;
}
```

Επειδή η `strcmp()` σταματάει τη σύγκριση όταν συναντήσει τον τερματικό χαρακτήρα, το πρόγραμμα θα εμφανίσει `Same`. Αντίθετα, η `memcmp()` συγκρίνει όλες τις οκτάδες και το πρόγραμμα θα εμφανίσει `Different`

Παράδειγμα (2)

- Δημιουργήστε μία εκδοχή της συνάρτησης memcmp(). Να γραφεί ένα πρόγραμμα το οποίο να διαβάζει δύο αλφαριθμητικά μέχρι 100 χαρακτήρες, τον αριθμό των χαρακτήρων που θα συγκριθούν και να εμφανίζει το αποτέλεσμα της σύγκρισής τους με χρήση της συνάρτησης

Παράδειγμα (2)

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

int mem_cmp(const void *ptr1, const void *ptr2, size_t size);

int main()
{
	char str1[100], str2[100];
	int num;

	cout << "Enter first text: ";
	cin.getline(str1, sizeof(str1));

	cout << "Enter second text: ";
	cin.getline(str2, sizeof(str2));

	cout << "Enter characters to compare: ";
	cin >> num;

	cout << mem_cmp(str1, str2, num) << '\n';
	return 0;
}
int mem_cmp(const void *ptr1, const void *ptr2, size_t size)
{
	char *p1, *p2;

	p1 = (char*)ptr1;
	p2 = (char*)ptr2;
	while(size != 0)
	{
		if(*p1 != *p2)
			return *p1 - *p2;
		p1++;
		p2++;
		size--;
	}
	return 0;
```

Παράδειγμα (2)

- Σχόλια: Επειδή συγκρίνουμε χαρακτήρες, προσαρμόζουμε τον τύπο `void*` σε `char*`. Σε κάθε επανάληψη του βρόχου, η `mem_cmp()` συγκρίνει τους χαρακτήρες στους οποίους δείχνουν οι `ptr1` και `ptr2`. Αν όλοι οι χαρακτήρες είναι ίδιοι, η `mem_cmp()` επιστρέφει `0`, αλλιώς τη διαφορά των δύο πρώτων διαφορετικών χαρακτήρων

Στατικές Δομές Δεδομένων

- Οι **δομές δεδομένων** χρησιμοποιούνται για την αποθήκευση και επεξεργασία πληθώρας δεδομένων με εύκολο και γρήγορο τρόπο
- Π.χ. ο **πίνακας** είναι μία **δομή δεδομένων**, ο οποίος χρησιμοποιείται για την αποθήκευση δεδομένων ίδιου τύπου
- Παρομοίως, οι **δομές** (**structs**) και οι **ενώσεις** (**unions**) είναι **δομές δεδομένων**, οι οποίες χρησιμοποιούνται για την αποθήκευση δεδομένων οποιουδήποτε τύπου
- Αυτές οι δομές δεδομένων είναι **στατικές**, με την έννοια ότι η μνήμη που έχει δεσμευτεί για αυτές είναι στατική και **δεν μπορεί να αλλάξει** κατά την εκτέλεση του προγράμματος
- Π.χ. όταν δηλώνεται έναν πίνακας με μία συγκεκριμένη διάσταση (π.χ. `int arr[100]`), η διάστασή του, δηλαδή το `100`, δεν μπορεί να αλλάξει κατά την εκτέλεση του προγράμματος

Δυναμικές Δομές Δεδομένων

- Υπάρχουν όμως περιπτώσεις που αντί να χρησιμοποιήσουμε μία **στατική** δομή δεδομένων, όπως είναι ο πίνακας, να είναι πιο αποδοτικό να χρησιμοποιήσουμε μία **δυναμική δομή δεδομένων**

- Αντίθετα με τη στατική δομή, το μέγεθος μίας δυναμικής δομής δεδομένων **μπορεί να αυξομειώνεται κατά την εκτέλεση του προγράμματος** με τη δέσμευση και την αποδέσμευση αντίστοιχης μνήμης

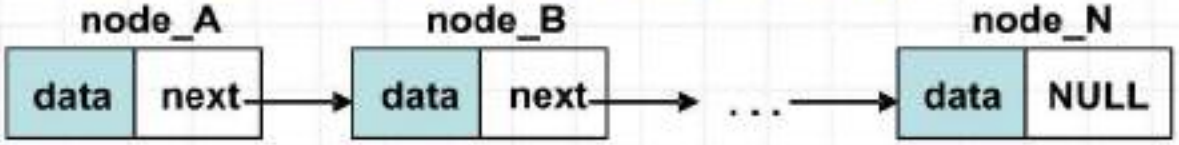
- Μία δυναμική δομή δεδομένων αποτελείται από ένα ή περισσότερα συνδεδεμένα στοιχεία, τα οποία συνήθως ονομάζονται **κόμβοι**

- Κάθε **κόμβος** συνήθως αντιπροσωπεύεται από μία **δομή**, η οποία περιέχει τα δεδομένα του κόμβου κι ένα πεδίο που είναι δείκτης στον επόμενο κόμβο

- Στη συνέχεια θα περιγράψουμε την πιο συνηθισμένη δυναμική δομή δεδομένων, την **απλά συνδεδεμένη λίστα** καθώς και δύο συγκεκριμένες υποπεριπτώσεις αυτής, τη **στοίβα** και την **ουρά**

Απλά Συνδεδεμένη Λίστα

- Μία συνηθισμένη δυναμική δομή δεδομένων είναι η **απλά συνδεδεμένη λίστα**
- Στο σχήμα φαίνεται ότι κάθε κόμβος μίας τέτοιας λίστας περιέχει τα **δεδομένα του κόμβου** (π.χ. το πεδίο `data`) και **έναν δείκτη** (π.χ. το πεδίο `next`) που «δείχνει» στον επόμενο κόμβο



- Ο **πρώτος κόμβος** της λίστας ονομάζεται **κεφαλή (head)** της λίστας και ο τελευταίος κόμβος ονομάζεται **ουρά (tail)**
- Το πεδίο-δείκτης του τελευταίου κόμβου **πρέπει** να έχει την τιμή `nullptr`, ώστε να προσδιορίζεται το τέλος της λίστας
- Ο χειρισμός μίας απλά συνδεδεμένης λίστας γίνεται συνήθως με τη χρήση **δύο δεικτών**, με τον πρώτο να δείχνει στη διεύθυνση μνήμης της **κεφαλής** της λίστας και τον δεύτερο στη διεύθυνση μνήμης της **ουράς** της λίστας

Εισαγωγή Κόμβου σε Απλά Συνδεδεμένη Λίστα (1)

- Για να εισάγουμε έναν νέο κόμβο στη λίστα, εξετάζουμε τις ακόλουθες περιπτώσεις:

  1) Αν η λίστα είναι κενή (δηλ. δεν περιέχει κανέναν κόμβο) τότε ο κόμβος εισάγεται στη λίστα και αποτελεί ταυτόχρονα την κεφαλή και την ουρά της λίστας, ενώ η τιμή του δείκτη του γίνεται `nullptr`, αφού δεν υπάρχει επόμενος κόμβος

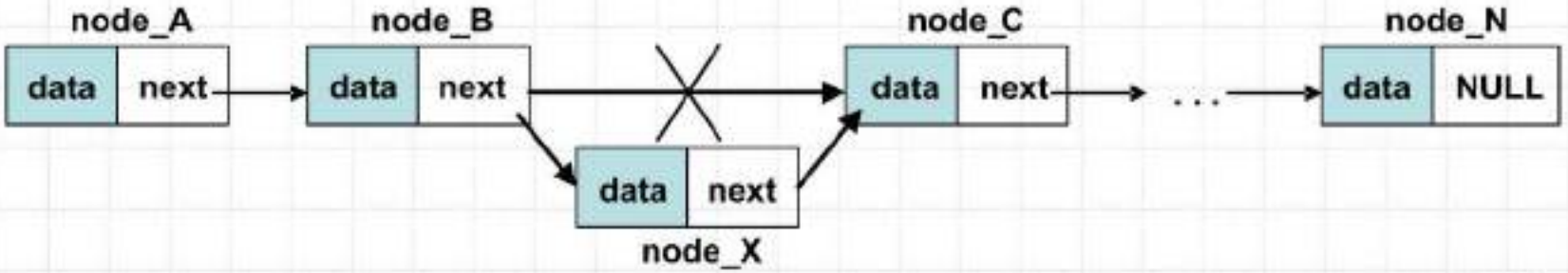
  2) Αν η λίστα δεν είναι κενή τότε εξετάζουμε τις ακόλουθες υπο-περιπτώσεις:

     2α) Αν επιθυμούμε ο νέος κόμβος να τοποθετηθεί στην αρχή της λίστας, τότε ο νέος κόμβος γίνεται η νέα κεφαλή της λίστας και ο δείκτης του δείχνει στην παλιά κεφαλή, που τώρα γίνεται ο δεύτερος κόμβος της λίστας

Εισαγωγή Κόμβου σε Απλά Συνδεδεμένη Λίστα (2)

2β) Αν επιθυμούμε ο νέος κόμβος να τοποθετηθεί στο **τέλος** της λίστας, τότε ο νέος κόμβος γίνεται η **νέα ουρά** της λίστας και η τιμή του δείκτη του γίνεται `nullptr`, ενώ ο κόμβος που ήταν προηγουμένως η ουρά της λίστας γίνεται ο προτελευταίος κόμβος της λίστας με την τιμή του δείκτη του να αλλάζει από `nullptr` και να δείχνει στον νέο κόμβο

2γ) Για να εισάγουμε έναν νέο κόμβο **μετά** από έναν ενδιάμεσο κόμβο μίας λίστας, τότε κάνουμε τον δείκτη αυτού του κόμβου να δείχνει στον νέο κόμβο και τον δείκτη του νέου κόμβου να δείχνει στον κόμβο που έδειχνε ο ενδιάμεσος κόμβος (όπως φαίνεται στο σχήμα, με τον κόμβο X να εισάγεται μεταξύ των κόμβων B και C)



Διαγραφή Κόμβου από Απλά Συνδεδεμένη Λίστα (1)

- Για να διαγράψουμε έναν κόμβο από μία λίστα, εξετάζουμε τις ακόλουθες περιπτώσεις:

1) Αν επιθυμούμε να διαγράψουμε τον κόμβο που είναι η **αρχή** της λίστας, τότε εξετάζουμε τις ακόλουθες υπο-περιπτώσεις:

1α) Αν υπάρχει επόμενος κόμβος, τότε αυτός ο κόμβος γίνεται η **νέα κεφαλή** της λίστας

1β) Αν δεν υπάρχει επόμενος κόμβος, τότε η λίστα γίνεται **κενή**

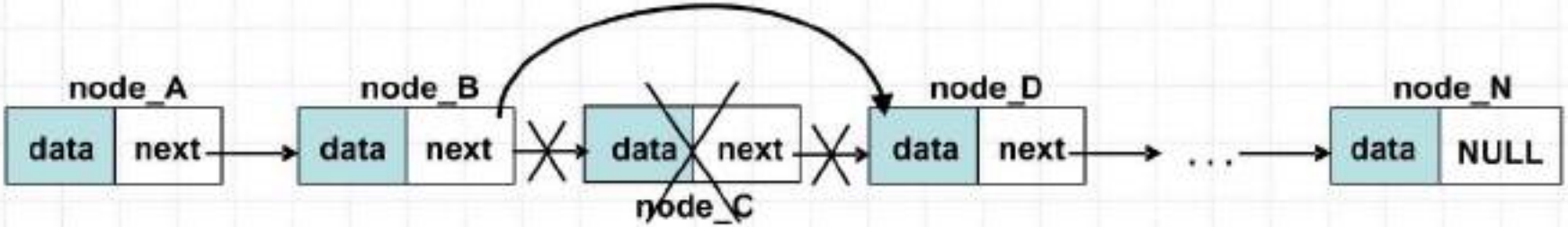
2) Αν επιθυμούμε να διαγράψουμε τον **τελευταίο** κόμβο της λίστας, τότε εξετάζουμε τις ακόλουθες υπο-περιπτώσεις:

2α) Αν υπάρχει προηγούμενος κόμβος, τότε αυτός ο κόμβος γίνεται η **νέα ουρά** της λίστας και η τιμή του δείκτη του γίνεται `nullptr`

2β) Αν δεν υπάρχει προηγούμενος κόμβος, τότε η λίστα γίνεται **κενή**

Διαγραφή Κόμβου από Απλά Συνδεδεμένη Λίστα (2)

3) Για να διαγράψουμε ένα κόμβο που βρίσκεται ανάμεσα σε δύο κόμβους μίας λίστας, τότε κάνουμε τον δείκτη του προηγούμενου κόμβου να δείχνει στον επόμενο κόμβο από αυτόν που θέλουμε να διαγράψουμε και αποδεσμεύουμε τη μνήμη που καταλαμβάνει (όπως φαίνεται στο σχήμα, όπου διαγράφεται ο κόμβος C)



Παρατηρήσεις

- Σε αντίθεση με τους πίνακες, που το μέγεθός τους δεν μπορεί να αλλάξει, μία συνδεδεμένη λίστα είναι πιο ευέλικτη γιατί το μέγεθός της μπορεί να μεταβληθεί δυναμικά, ανάλογα με τις ανάγκες του προγράμματος
- Επίσης, επειδή μπορούμε εύκολα να προσθέτουμε και να αφαιρούμε κόμβους οπουδήποτε στη λίστα, είναι πολύ πιο εύκολη η δημιουργία και διατήρηση μίας ταξινομημένης διάταξης
- Από την άλλη, η πρόσβαση σε ένα στοιχείο του πίνακα είναι γρήγορη και εύκολη, αφού ένας πίνακας υποστηρίζει τυχαία πρόσβαση χρησιμοποιώντας τη θέση του σαν δείκτη. Αντίθετα, για την πρόσβαση σε έναν κόμβο πρέπει να ξεκινήσουμε από την κεφαλή και να διασχίσουμε τη λίστα σειριακά
- Κατά συνέπεια, ο χρόνος πρόσβασης ενός κόμβου εξαρτάται από τη θέση του στη λίστα. Αν είναι στην αρχή της λίστας είναι μικρός, μεγαλύτερος αν είναι στο τέλος της

Στοίβα (stack)

- Η στοίβα (stack) αποτελεί μία ειδική περίπτωση λίστας, με τους ακόλουθους περιορισμούς:

  1) Η εισαγωγή ενός νέου κόμβου γίνεται μόνο στην αρχή της στοίβας, δηλαδή, κάθε νέος κόμβος στη στοίβα γίνεται η νέα κεφαλή της στοίβας

  2) Ο μόνος κόμβος που επιτρέπεται να διαγράψουμε από τη στοίβα είναι η κεφαλή της στοίβας

- Μία τέτοια στοίβα ονομάζεται LIFO (Last In First Out), με την έννοια ότι ο κόμβος που εισάγεται τελευταίος εξάγεται πρώτος

- Μπορείτε να σκεφτείτε την υλοποίηση μίας στοίβας ως μία στοίβα από άπλυτα πιάτα, όπου το κάθε νέο (άπλυτο) πιάτο το τοποθετούμε στην κορυφή της στοίβας και το πιάτο που θέλουμε να πλύνουμε το αφαιρούμε από την κορυφή της

- Μπορείτε να δείτε ένα παράδειγμα υλοποίησης στοίβας στην αντίστοιχη ενότητα του βιβλίου

Ουρά (queue)

- Η ουρά (queue) αποτελεί μία ειδική περίπτωση λίστας, με τους ακόλουθους περιορισμούς:

  1) Η εισαγωγή ενός νέου κόμβου γίνεται μόνο στο τέλος της ουράς, δηλαδή, κάθε νέος κόμβος γίνεται η νέα «ουρά» της ουράς

  2) Ο μόνος κόμβος που επιτρέπεται να διαγράψουμε από την ουρά είναι η κεφαλή της ουράς

- Μία τέτοια ουρά ονομάζεται FIFO (First In First Out), με την έννοια ότι ο κόμβος που εισάγεται πρώτος εξάγεται και πρώτος

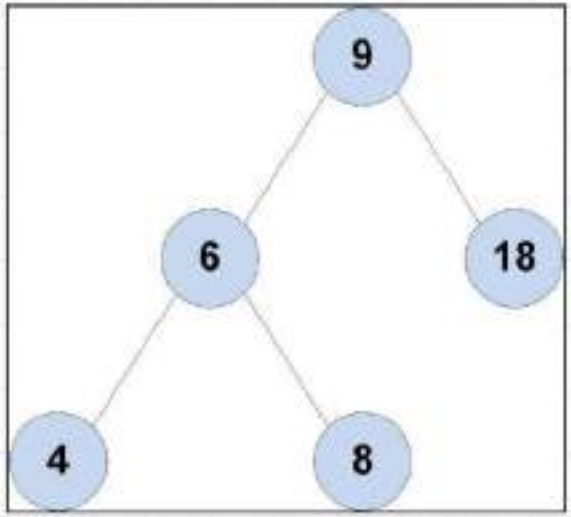
- Μπορείτε να δείτε ένα παράδειγμα υλοποίησης ουράς στην αντίστοιχη ενότητα του βιβλίου

Δυαδικά Δέντρα Αναζήτησης (1)

- Για την οργάνωση των δεδομένων σε ιεραρχική διάταξη συνήθως χρησιμοποιείται μία ευέλικτη δομή δεδομένων που ονομάζεται δένδρο
- Η βασική έννοια του δένδρου μας είναι γνωστή από διάφορα παραδείγματα, όπως το οικογενειακό δένδρο, τα οργανογράμματα οργανισμών ή το σύστημα οργάνωσης φακέλων σε έναν υπολογιστή
- Υπάρχουν πολλοί τύποι δένδρων, εμείς θα παρουσιάσουμε τις βασικές λειτουργίες στο δυαδικό δένδρο αναζήτησης (binary search tree)
- Ένα δένδρο (με ρίζα) είναι μία συλλογή από κόμβους που συνδέονται με ακμές
- Υπάρχει ένας μόνο κόμβος στον οποίο δεν καταλήγει καμία ακμή και ονομάζεται ρίζα
- Στους υπόλοιπους κόμβους καταλήγει μία και μόνο ακμή
- Κάθε κόμβος, εκτός της ρίζας, συνδέεται με ακριβώς έναν κόμβο πάνω από αυτόν που ονομάζεται γονικός
- Οι κόμβοι με τους οποίους συνδέεται ακριβώς κάτω από αυτόν ονομάζονται παιδιά του

Δυαδικά Δέντρα Αναζήτησης (2)

- Ένα δένδρο του οποίου **κάθε κόμβος έχει το πολύ δύο παιδιά** ονομάζεται **δυαδικό δέντρο**
- Στα δύο τμήματα του κάθε κόμβου αναφερόμαστε ως **αριστερό** και **δεξιό υποδένδρο**, αντίστοιχα
- **Δυαδικό δέντρο αναζήτησης** είναι ένα δυαδικό δένδρο, του οποίου κάθε κόμβος **σχετίζεται με ένα κλειδί** και ισχύει ότι:
  - α. το κλειδί του είναι μεγαλύτερο (ή ίσο) από τα κλειδιά όλων των κόμβων στο αριστερό του υποδένδρο και
  - β. το κλειδί του είναι μικρότερο (ή ίσο) από τα κλειδιά όλων των κόμβων στο δεξιό του υποδένδρο
- Ο σημαντικότερος λόγος δημιουργίας ενός τέτοιου δένδρου είναι ότι επιτρέπει την ανάπτυξη αλγορίθμων με **υψηλές επιδόσεις** κατά την εκτέλεση κρίσιμων λειτουργιών, όπως της εισαγωγής και αναζήτησης ενός στοιχείου



- Μπορείτε να δείτε ένα παράδειγμα υλοποίησης κάποιων βασικών λειτουργιών σε ένα δυαδικό δένδρο αναζήτησης στην αντίστοιχη ενότητα του βιβλίου

C++: Από τη Θεωρία στην Εφαρμογή

Κεφάλαιο 15ο

Προεπεξεργαστής και Μακροεντολές

Προεπεξεργαστής και Μακροεντολές

- Ο **προεπεξεργαστής** είναι ένα πρόγραμμα λογισμικού, συνήθως ενοποιημένο με τον μεταγλωττιστή, ο οποίος επεξεργάζεται ένα C++ πρόγραμμα πριν αυτό μεταγλωττιστεί

- Οι κυριότερες λειτουργίες του είναι:
  - η **συμπερίληψη αρχείων**
  - η **μακροαντικατάσταση** και
  - η **επιλεκτική μεταγλώττιση**

- Στη συνέχεια θα δούμε πώς μπορούμε να ορίσουμε και να χρησιμοποιήσουμε μακροεντολές, καθώς και τις οδηγίες του προεπεξεργαστή που επιτρέπουν την επιλεκτική μεταγλώττιση του προγράμματος

Απλές Μακροεντολές (1)

- Η συμπεριφορά του προεπεξεργαστή καθορίζεται από **οδηγίες**, που είναι **ειδικές εντολές**, οι οποίες **αρχίζουν με τον χαρακτήρα #** και υποχρεώνουν τον προεπεξεργαστή να ενεργήσει ανάλογα
- Π.χ., όταν ο προεπεξεργαστής συναντήσει την οδηγία `#include` υποχρεώνεται να συμπεριλάβει στο σημείο του προγράμματος που συνάντησε την οδηγία το περιεχόμενο του αντίστοιχου αρχείου
- Οι οδηγίες **μπορεί να εμφανίζονται οπουδήποτε μέσα στο πρόγραμμα**, ενώ μία γραμμή που περιέχει ένα μόνο # είναι αποδεκτή, απλά δεν έχει καμία επίδραση
- Επίσης, έχουμε χρησιμοποιήσει την οδηγία `#define` **για να ορίσουμε μία απλή μακροεντολή**, δηλαδή **ένα συμβολικό όνομα**, το οποίο **αντιστοιχίζεται με μία τιμή**

Απλές Μακροεντολές (2)

- Μία απλή μακροεντολή ορίζεται ως εξής:

```
#define  όνομα_μακροεντολής  ακολουθία_χαρακτήρων
```

- Τα ονόματα ή αναγνωριστικά των μακροεντολών πρέπει να ακολουθούν τους ίδιους κανόνες ονοματοδοσίας, όπως και οι μεταβλητές
- Συνήθως, ονοματίζονται με κεφαλαία γράμματα, ώστε να ξεχωρίζουν από τις μεταβλητές του προγράμματος
- Η εμβέλεια του ονόματος αρχίζει από το σημείο του ορισμού του μέχρι το τέλος του αρχείου

Απλές Μακροεντολές (3)

Στο παρακάτω παράδειγμα, ο προεπεξεργαστής αντικαθιστά κάθε εμφάνιση του LEN με την τιμή 200

```
#include <iostream>
#define LEN 200
int main()
{
        int i, arr[LEN];
        for(i = 0; i < LEN; i++)
                arr[i] = i + LEN;
        return 0;
}
```

Αν γράφαμε:

```
#define LEN = 200
```

η δήλωση του πίνακα arr θα μεταφραζόταν σε arr[= 200] και η μεταγλώττιση θα αποτύγχανε

Απλές Μακροεντολές (4)

- Αν η μακροεντολή περιέχει τελεστές, να την περικλείετε σε παρενθέσεις
- Π.χ. δείτε τι συμβαίνει αν παραλείψουμε τις παρενθέσεις:

```
#define  NUM 2*5 /* Αντί για (2*5). */
```

Μία εντολή, όπως η:

```
double j = 3.0/NUM;
```

μεταφράζεται σε:

```
double j = 3.0/2*5 = 7.5;
```

δηλαδή, εκχωρεί την τιμή 7.5 και όχι την τιμή 0.3 στην j, αφού η διαίρεση εκτελείται πριν από τον πολλαπλασιασμό

Απλές Μακροεντολές (5)

- Μία μακροεντολή μπορεί να οριστεί οπουδήποτε στο πρόγραμμα, π.χ. μέσα σε μία συνάρτηση
- Η συνήθης πρακτική είναι όλες οι μακροεντολές να ορίζονται μαζί με καθολική εμβέλεια στην αρχή του προγράμματος ή σε κάποιο αρχείο επικεφαλίδας, το οποίο να συμπεριλαμβάνεται στο πρόγραμμα με την οδηγία `#include` όπου χρειάζεται
- Π.χ., έστω ότι έχουμε δημιουργήσει το αρχείο `test.h` με περιεχόμενο:

```
#include <iostream>
#define LEN 200
```

Ερώτηση: Μπορούμε να γράψουμε το προηγούμενο πρόγραμμα ως εξής:

```
#include "test.h"
int main()
{
        ...
}
```

Ναι, φυσικά! Δηλαδή, αν και σε όλα τα προγράμματά μας συμπεριλαμβάνουμε άμεσα το `iostream`, δεν είναι υποχρεωτικό. Θα μπορούσαμε να συμπεριλάβουμε ένα άλλο αρχείο, όπως εδώ με το `test.h`, και να περιέχεται το `iostream` μέσα σε αυτό.

Απλές Μακροεντολές (6)

- Αν το όνομα μίας μακροεντολής περιέχεται στο όνομα μίας μεταβλητής ή ενός κυριολεκτικού αλφαριθμητικού δεν αντικαθίσταται, π.χ.:

```
#include <iostream>
#define LEN 200

int main()
{
        int BUF_LEN; /* Δεν γίνεται αντικατάσταση. */
        std::cout << "LEN is not used\n"; /* Δεν γίνεται αντικατάσταση. */
        return 0;
}
```

- Αν και οι απλές μακροεντολές χρησιμοποιούνται κυρίως για να συσχετίσουμε συμβολικά ονόματα με αριθμούς, μπορούν να χρησιμοποιηθούν και για άλλους σκοπούς, π.χ. :

```
#include <iostream>
#define TEST std::cout << "example\n"
int main()
{
        TEST;
        return 0;
}
```

Ο προεπεξεργαστής αντικαθιστά τη μακροεντολή **TEST** και το πρόγραμμα εμφανίζει **example**

Απλές Μακροεντολές (7)

- Σημειώστε ότι επιτρέπεται να ορίσουμε μία μακροεντολή χωρίς τιμή αντικατάστασης και, όπως θα δούμε παρακάτω, μία τέτοια μακροεντολή χρησιμοποιείται συνήθως για επιλεκτική μεταγλώττιση, π.χ.:

```
#define LABEL
```

- Το όνομα μίας μακροεντολής μπορεί να χρησιμοποιηθεί στον ορισμό μίας άλλης μακροεντολής, π.χ.:

```
#define SIZE 200
#define NUM  SIZE
```

- Επίσης, υπάρχουν μεταγλωττιστές που επιτρέπουν να ορίσουμε μία μακροεντολή, όταν μεταγλωττίζεται το πρόγραμμα, π.χ.:

```
gcc  -DVERSION=3  test.cpp
```

- Με την επιλογή -D, ορίζεται η μακροεντολή VERSION με τιμή 3 και αν δεν οριστεί τιμή, η τιμή της γίνεται 1

- Μία τέτοια δυνατότητα είναι πολύ χρήσιμη, αφού δεν χρειάζεται να τροποποιήσουμε κάποιο αρχείο για να την ορίσουμε ή για να αλλάξουμε την τιμή της

Απλές Μακροεντολές (8)

- Αν θέλουμε να εκτείνουμε μία μακροεντολή σε παραπάνω από μία γραμμές (π.χ. για καλύτερη αναγνωσιμότητα), προσθέτουμε τον χαρακτήρα \ στο τέλος κάθε γραμμής

Οπότε, τι θα εμφανίσει το παρακάτω πρόγραμμα;

```
#include <iostream>
#define NUM \
    10 + \
    20 + \
    30
int main()
{
        std::cout << NUM;
        return 0;
}
```

Έξοδος: 60

Απλές Μακροεντολές (9)

- Στη C++, οι παρακάτω μακροεντολές είναι προκαθορισμένες. Όπως φαίνεται, τα ονόματά τους περικλείονται σε διπλές κάτω παύλες (underscore) και ο κύριος σκοπός τους είναι να παρέχουν πληροφορίες για τη μεταγλώττιση του προγράμματος

Όνομα	Περιγραφή
__DATE__	Αντικαθίσταται με την ημερομηνία μεταγλώττισης.
__FILE__	Αντικαθίσταται με το όνομα του αρχείου που μεταγλωττίζεται.
__LINE__	Αντικαθίσταται με τον αριθμό της γραμμής του αρχείου που μεταγλωττίζεται.
__TIME__	Αντικαθίσταται με την ώρα μεταγλώττισης.
__func__	Αντικαθίσταται με ένα μήνυμα που ορίζεται από την υλοποίηση και περιέχει το όνομα της συνάρτησης

- Π.χ.

```
#include <iostream>
int main()
{
   std::cout << "Line " << __LINE__ << " of file " << __FILE__ << " in function " << _func__
         << " is compiled on " << __DATE__ << " at " << __TIME__ << '\n';
   return 0;
}
```

Έξοδος: `Line 4 of file c:\edu\projects\test.cpp compiled on May 8 2023 at 9:27:48`

Η Μακροεντολή assert

- Για διαγνωστικούς σκοπούς, η C++ παρέχει τη μακροεντολή `assert()`, που δηλώνεται στο αρχείο `cassert`, ως εξής:

```
void assert(int exp);
```

- Αν η τιμή της έκφρασης `exp` είναι αληθής, η `assert()` δεν εμφανίζει τίποτα, αλλιώς, εμφανίζει ένα μήνυμα λάθους στο `cerr` και καλεί τη συνάρτηση `std::abort()`, η οποία τερματίζει άμεσα το πρόγραμμα

- Το μήνυμα που εμφανίζεται ορίζεται από την υλοποίηση, όμως θα περιέχει τουλάχιστον το όνομα του αρχείου κώδικα και τον αριθμό γραμμής της `assert()`. Π.χ. :

```
assert(a != 0); /* Πριν γίνει η διαίρεση ελέγχουμε την τιμή του
                   παρονομαστή. */
c = b/a;
```

Παράδειγμα

Ποια είναι τα λάθη στο παρακάτω πρόγραμμα;

```
#include <iostream>

#define LEN20
#define TEST std::cout << "example\n";

int main()
{
        int arr[LEN] = {10};
        if(arr[0] == 10)
                TEST;
        else
                std::cout << "Not 10\n";
        return 0;
}
```

Παράδειγμα

Απάντηση. Επειδή δεν υπάρχει κενό μεταξύ του **LEN** και του 20, ο μεταγλωττιστής θα εμφανίσει μήνυμα λάθους ότι το **LEN** δεν έχει δηλωθεί. Το δεύτερο λάθος μεταγλώττισης προκαλείται από το ; στο τέλος της **TEST**. Όταν γίνει αντικατάσταση της **TEST** το ; δεν αφήνει την **else** να συνδεθεί με την **if**

Μακροεντολές με Παραμέτρους (1)

- Μία μακροεντολή, εκτός από την απλή μορφή της, μπορεί να δέχεται παραμέτρους και να μοιάζει με μία συνάρτηση, όπως στο παράδειγμα

```
#include <iostream>

#define MIN(a, b) ((a) < (b) ? (a) : (b))

int main()
{
        int i, j, min;

        std::cout << "Enter numbers: ";
        std::cin >> i >> j;
        min = MIN(i, j);
        std::cout << min << '\n';
        return 0;
}
```

Η μακροεντολή **MIN** δέχεται δύο παραμέτρους **a** και **b**. Όταν ο προεπεξεργαστής τη συναντάει στο πρόγραμμα, αντικαθιστά το **a** με το **i**, το **b** με το **j** και τη μεταφράζει σε:

```
min = ((i) < (j) ? (i) : (j));
```

Μακροεντολές με Παραμέτρους (2)

- Λόγω της προτεραιότητας των τελεστών, να περικλείετε πάντα τις παραμέτρους μίας μακροεντολής καθώς και τον ορισμό της μακροεντολής σε παρενθέσεις

Π.χ. δείτε τι συμβαίνει αν παραλείψουμε τις παρενθέσεις στην παρακάτω μακροεντολή:

```
#define MUL(a, b)  (a*b) /* Αντί για ((a)*(b)). */
```

Αν μία εντολή του προγράμματος ήταν η:

```
j = MUL(9+1, 8+2);
```

θα μεταφραζόταν σε:

```
j = 9+1*8+2 = 19;
```

η οποία εκχωρεί στη μεταβλητή j την τιμή 19 και όχι την τιμή 100, αφού ο πολλαπλασιασμός εκτελείται πριν από την πρόσθεση

Μακροεντολές με Παραμέτρους (3)

- Αν μία μακροεντολή δέχεται παραμέτρους, μην αφήνετε κενό διάστημα μεταξύ του ονόματος και της αριστερής παρένθεσης, γιατί ο προεπεξεργαστής θα χειριστεί την αριστερή παρένθεση σαν την αρχή του ορισμού μίας απλής μακροεντολής

Π.χ. στην προηγούμενη μακροεντολή

```
#define MUL(a, b) ((a)*(b))
```

αν αφήσουμε ένα κενό μεταξύ του ονόματός της (MUL) και της αριστερής παρένθεσης

```
#define MUL (a, b) ((a)*(b))
```

η μεταγλώττιση θα αποτύχει, αφού το MUL θα αντικατασταθεί με το τμήμα της εντολής που αρχίζει μετά το πρώτο κενό

Μακροεντολές με Παραμέτρους (4)

- Όταν μεταβιβάζετε ορίσματα, μην εφαρμόζετε τελεστές που μπορεί να αλλάξουν την τιμή τους

Π.χ. αν είχατε δηλώσει τη μακροεντολή:

```
#define MIN(a, b) ((a) < (b) ? (a) : (b))
```

και στο πρόγραμμά σας χρησιμοποιούσατε την παρακάτω εκχώρηση:

```
x = MIN(i++, j);
```

ο επεξεργαστής τη μεταφράζει σε

```
x = ((i++) < (j) ? (i++) : (j));
```

οπότε, αν το `i` είναι μικρότερο από το `j`, η τιμή του `i` θα αυξηθεί (κακώς) δύο φορές και θα εκχωρηθεί λάθος τιμή στη `x`

Παρατηρήσεις (1)

- Εκτός από τα θέματα σύνταξης και τους ενδεχόμενους κινδύνους, η χρήση μίας μακροεντολής στη θέση συνάρτησης έχει **σημαντικά μειονεκτήματα**
- Για παράδειγμα, επειδή οι μακροεντολές βασίζονται στην αντικατάσταση κειμένου, είναι πολύ πιθανό να δημιουργηθούν ανεπιθύμητες αντικαταστάσεις στην περίπτωση που η μακροεντολή περιέχει πολύπλοκο κώδικα
- Επίσης, είναι πολύ πιο δύσκολη η κατανόηση, η διαχείριση και η αποσφαλμάτωση μίας μακροεντολής που περιέχει πολλές εντολές
- Μία μακροεντολή δεν μπορεί να υπερφορτωθεί, ούτε μπορεί να είναι αναδρομική
- Ακόμα, δεν μπορούμε να δηλώσουμε έναν δείκτη σε μακροεντολή, γιατί ο προεπεξεργαστής απομακρύνει τον κώδικα της μακροεντολής πριν τη μεταγλώττιση

Παρατηρήσεις (2)

- Ένα ακόμα μειονέκτημα είναι ότι, αν μία μακροεντολή χρησιμοποιείται πολλές φορές και αποτελείται από πολλές γραμμές, η εμβόλιμη πρόσθεση του κώδικά της στο πρόγραμμα θα μεγαλώσει το μέγεθός του, λόγω της επανάληψης του κώδικα
- Επίσης, όταν καλείται μία συνάρτηση, ο μεταγλωττιστής ελέγχει τους τύπους των ορισμάτων της. Αν ο τύπος κάποιου ορίσματος δεν ταιριάζει με τον τύπο που έχει δηλωθεί στη συνάρτηση, ο μεταγλωττιστής θα προσπαθήσει να τον μετατρέψει και, αν δεν τα καταφέρει, θα εμφανίσει μήνυμα λάθους. Αντίθετα, ο προεπεξεργαστής δεν ελέγχει τους τύπους των ορισμάτων που μεταβιβάζονται σε μία μακροεντολή, επομένως μπορεί να μεταβιβαστούν ανεπιθύμητες τιμές
- Η C++ παρέχει πιο κατάλληλους και ασφαλέστερους μηχανισμούς που μπορούμε να χρησιμοποιήσουμε στη θέση τέτοιων μακροεντολών, όπως οι `inline` και οι πρότυπες συναρτήσεις που θα δούμε στο Κ.16

Ο Τελεστής Προεπεξεργαστή

- Ο τελεστής # μπροστά από το όρισμα μίας μακροεντολής, υποχρεώνει τον προεπεξεργαστή να δημιουργήσει ένα αλφαριθμητικό με το όνομα του ορίσματος

Π.χ.

```
#include <iostream>

#define f(s) std::cout << #s << " = " << s << '\n';

int main()
{
        const char *str = "text";
        f(str);
        return 0;
}
```

Όταν ο προεπεξεργαστής μεταφράζει τη μακροεντολή f, αντικαθιστά το #s με το όνομα του ορίσματος, που είναι το str, οπότε, το πρόγραμμα εμφανίζει:

```
str = text
```

Ο Τελεστής Προεπεξεργαστή

- Ο τελεστής ## χρησιμοποιείται σε μία μακροεντολή για την ένωση συμβόλων (π.χ. χαρακτήρες). Δεν επιτρέπεται να εμφανίζεται στην αρχή ή στο τέλος της ακολουθίας και αν κάποιο σύμβολο είναι παράμετρος μακροεντολής, ο προεπεξεργαστής πρώτα το αντικαθιστά με την τιμή του αντίστοιχου ορίσματος και μετά γίνεται η συνένωση. Π.χ.

```
#include <iostream>

#define f(a) s##u##m##a

int suml(int a, int b);
int main()
{
    int i, j;

    std::cout << "Enter numbers: ";
    std::cin >> i >> j;
    std::cout << f(1)(i, j) << '\n';
    return 0;
}
int suml(int a, int b)
{
    return a+b;
}
```

Όταν ο προεπεξεργαστής μεταφράζει τη μακροεντολή `f`, ενώνει τους χαρακτήρες `s`, `u`, `m` και, επειδή ο χαρακτήρας `a` είναι ίδιος με το όνομα της παραμέτρου, τον αντικαθιστά με την τιμή του ορίσματος, που είναι `1`.

Άρα, η `f(1)(i, j)` μεταφράζεται σε `suml(i, j)`.

Άρα, το πρόγραμμα διαβάζει δύο ακεραίους και εμφανίζει την τιμή επιστροφής της `suml()`, δηλαδή το άθροισμά τους.

Οδηγίες Προεπεξεργαστή για Επιλεκτική Μεταγλώττιση

- Στη συνέχεια θα περιγράψουμε τις οδηγίες του προεπεξεργαστή που επιτρέπουν την επιλεκτική μεταγλώττιση τμημάτων του προγράμματος
- Η επιλεκτική μεταγλώττιση είναι **ιδαίτερα χρήσιμη** σε περιπτώσεις, όπως για παράδειγμα, όταν πρέπει να διατηρούνται πολλές εκδόσεις του ίδιου προγράμματος, για την παρακολούθηση της εκτέλεσης του προγράμματος ή για την αποσφαλμάτωση του προγράμματος

Οι Οδηγίες #if και #endif

- Οι οδηγίες **#if** και **#endif** χρησιμοποιούνται για να καθορίσουν τα τμήματα του κώδικα που θα μεταγλωττιστούν (αρχή και τέλος αντίστοιχα), ανάλογα με την τιμή μίας έκφρασης

Η γενική σύνταξη είναι:

```
#if έκφραση
        ... /* ομάδα εντολών */
#endif
```

- Αν η τιμή της είναι αληθής, ο προεπεξεργαστής κρατάει την ομάδα των εντολών για μεταγλώττιση. Αλλιώς, τις απομακρύνει και ο μεταγλωττιστής δεν αντιλαμβάνεται την ύπαρξή τους

- Η **έκφραση** είναι μία ακέραια σταθερά παράσταση που δεν μπορεί να περιέχει τον τελεστή **sizeof**, προσαρμογές τύπου ή **enum** σταθερές απαρίθμησης

Η Οδηγία #else

- Η οδηγία **#else** χρησιμοποιείται μαζί με την **#if** οδηγία για να δηλώσει μία ομάδα εντολών, η οποία θα μεταγλωττιστεί, αν η τιμή της **έκφρασης** είναι ψευδής και η γενική σύνταξη είναι:

```
#if έκφραση
        ... /* ομάδα εντολών A */
#else
        ... /* ομάδα εντολών B */
#endif
```

Π.χ. τι εμφανίζει το παρακάτω πρόγραμμα?

```
#include <iostream>
#define NUM 10
int main()
{
        #if NUM < 0
                std::cout << "Seg_1\n";
        #else
                std::cout << "Seg_2\n";
        #endif
        return 0;
}
```

Θα μεταγλωττιστεί το τμήμα **#else**, αφού η τιμή της **NUM** είναι μεγαλύτερη από 0. Άρα, το πρόγραμμα εμφανίζει: **Seg_2**

Η Οδηγία #elif

- Η οδηγία **#elif** μπορεί να χρησιμοποιηθεί μαζί με τις **#if**, **#ifdef** και **#ifndef** για να καθορίσει πολλαπλές επιλογές μεταγλώττισης

Η γενική σύνταξη είναι:

```
#if έκφραση_A
        ... /* ομάδα εντολών A */
#elif έκφραση_B
        ... /* ομάδα εντολών B */
.
.
#else
        ... /* ομάδα εντολών N */
#endif
```

Δηλ. η δεύτερη ομάδα εντολών (ομάδα εντολών B) θα μεταγλωττιστεί μόνο αν η τιμή της πρώτης έκφρασης (έκφραση_A) είναι ψευδής και της δεύτερης (έκφραση_B) αληθής, ενώ η τελευταία ομάδα εντολών (ομάδα εντολών N) θα μεταγλωττιστεί μόνο αν όλες οι τιμές των προηγούμενων εκφράσεων είναι ψευδείς (η τελευταία οδηγία **#else** είναι προαιρετική)

Παράδειγμα

- Ποια είναι η έξοδος του παρακάτω προγράμματος;

```
#include <iostream>

#define VER_2 1

int main()
{
    int cnt;

    #if VER_1
      cnt = 1;
      std::cout << "Version_1\n";
    #elif VER_2
      cnt = 2;
      std::cout << "Version_2\n";
    #else
      cnt = 3;
      std::cout << "Version_3\n";
    #endif

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

Αφού η VER_1 δεν είναι ορισμένη και η τιμή της VER_2 είναι 1 (αληθής), η τιμή της μεταβλητής cnt γίνεται 2 και το πρόγραμμα εμφανίζει:

```
Version_2
Cnt = 2
```

Αν αλλάξουμε την τιμή της VER_2 σε 0, η #elif έκφραση γίνεται ψευδής και εκτελούνται οι εντολές της #else οδηγίας. Επομένως, το πρόγραμμα εμφανίζει:

```
Version_3
Cnt = 3
```

Η Οδηγία #ifdef

- Η οδηγία **#ifdef**, χρησιμοποιείται για να ελέγξουμε αν ένα αναγνωριστικό **έχει οριστεί** ως μακροεντολή. Η γενική σύνταξη είναι:

```
#ifdef όνομα_αναγνωριστικού
        ... /* ομάδα εντολών */
#endif
```

- Η διαφορά με την οδηγία **#if** είναι ότι η **#ifdef** ελέγχει μόνο αν το αναγνωριστικό έχει οριστεί σαν μακροεντολή και δεν εξετάζει την τιμή του. Π.χ.

```
#include <iostream>
#define VER_1 0
int main()
{
        #ifdef VER_1
            std::cout << "Version_1\n";
        #else
            std::cout << "Version_2\n";
        #endif
        return 0;
}
```

Δεδομένου ότι η **#ifdef** ελέγχει μόνο αν το αναγνωριστικό έχει οριστεί (χωρίς να εξετάζει την τιμή του) και αφού η **VER_1** έχει οριστεί, το πρόγραμμα εμφανίζει:
Version_1

Η Οδηγία #ifndef (1)

- Η οδηγία **#ifndef** χρησιμοποιείται για να ελέγξουμε αν ένα αναγνωριστικό **δεν έχει οριστεί** σαν μακροεντολή. Η γενική της σύνταξη είναι:

```
#ifndef όνομα_αναγνωριστικού
        ... /* ομάδα εντολών */
#endif
```

- Μία συνηθισμένη χρήση της **#ifndef** είναι για να αποφύγουμε πολλαπλές συμπεριλήψεις του ίδιου αρχείου

- Π.χ. ένα μεγάλο πρόγραμμα αποτελείται από πολλά αρχεία κώδικα και αρχεία επικεφαλίδας και είναι πολύ πιθανό, το ίδιο αρχείο επικεφαλίδας να συμπεριλαμβάνεται σε περισσότερα από ένα αρχεία κώδικα

Η Οδηγία #ifndef (2)

- Ας υποθέσουμε ότι το αρχείο test.cpp συμπεριλαμβάνει τα παρακάτω αρχεία

```
/* test.cpp */
#include "test.h"
#include "one.h"
#include "two.h" /* Υποθέστε ότι και
το two.h συμπεριλαμβάνει το αρχείο test.h */
```

- Η παραδοσιακή τεχνική για να αποφύγουμε την πολλαπλή συμπερίληψη του ίδιου αρχείου (π.χ. test.h) είναι να προσθέσουμε τις επόμενες γραμμές στην αρχή του, π.χ. θεωρήστε το αρχείο test.h:

```
/* test.h */
#ifndef SOME_TAG /* Χρησιμοποιούμε ένα όνομα που δεν έχει
χρησιμοποιηθεί αλλού. */
#define SOME_TAG
/* Περιεχόμενα του test.h */
#endif
```

Η Οδηγία #ifndef (3)

- Όταν ο προεπεξεργαστής συναντήσει την πρώτη οδηγία που συμπεριλαμβάνει το `test.h`, θα συμπεριλάβει το περιεχόμενο του, αφού η `SOME_TAG` δεν έχει οριστεί
- Όμως, όταν ο προεπεξεργαστής συναντήσει πάλι την ίδια οδηγία για να συμπεριλάβει το `test.h` (π.χ. στο `two.h`), δεν θα συμπεριλάβει πάλι το περιεχόμενό του, γιατί τώρα η `SOME_TAG` είναι ορισμένη και, επομένως, η `#ifndef` συνθήκη γίνεται ψευδής
- Στη θέση της `SOME_TAG` **μπορείτε να επιλέξετε όποιο όνομα επιθυμείτε**, συνήθως επιλέγεται όνομα που να ταιριάζει με το όνομα του αρχείου και με μερικούς ακόμη χαρακτήρες για την αποφυγή σύγκρουσης του ονόματος (π.χ. `test_file_h`)

Η Οδηγία #ifndef (4)

- Μπορούμε να χρησιμοποιήσουμε την #ifndef, ώστε αν το test.h περιέχει ορισμούς τύπων (π.χ. κάποιας δομής), η πολλαπλή μεταγλώττιση του test.h θα αποτύχει, αφού δεν επιτρέπεται ο ορισμός του ίδιου τύπου παραπάνω από μία φορές

- Π.χ., ας υποθέσουμε ότι το test.h περιέχει τον ακόλουθο τύπο:

```
struct St
{
        ...
};
```

και το two.h περιλαμβάνει το test.h.

- Αν λείπει η #ifndef, όταν μεταγλωττιστεί το test.cpp, ο ορισμός της St θα συμπεριληφθεί δύο φορές, το οποίο δεν είναι αποδεκτό και προκαλείται σφάλμα μεταγλώττισης

Η Οδηγία #ifndef (5)

- Επίσης, με τη χρήση της οδηγίας `#ifndef` μπορούμε να αποφύγουμε ατέρμονες συμπεριλήψεις
- Μία τέτοια περίπτωση μπορεί να συμβεί όταν ένα αρχείο (π.χ. `one.h`) συμπεριλαμβάνει ένα δεύτερο (π.χ. `sec.h`) και αυτό το δεύτερο συμπεριλαμβάνει το πρώτο

Η Οδηγία #undef

- Η οδηγία #undef, χρησιμοποιείται για την κατάργηση μίας μακροεντολής, που έχει οριστεί νωρίτερα. Αν δεν έχει οριστεί, η #undef δεν έχει επίδραση

```
#include <iostream>

#define NUM 100
int main()
{
        int arr[NUM];
#undef NUM
        std::cout << "Array contains" << NUM << " elements\n";
return 0;
}
```

Αφού η οδηγία #undef NUM
καταργεί τον ορισμό της NUM, ο μεταγλωττιστής θα εμφανίσει μήνυμα λάθους στη cout

Ο Τελεστής defined

- Ένας εναλλακτικός τρόπος για να ελέγξουμε αν ένα αναγνωριστικό είναι ορισμένο ως μακροεντολή είναι με χρήση του τελεστή defined

Π.χ.:

```
#if defined(VER_1) /* Ισοδύναμο με #ifdef VER_1 */
...
#endif
```

Αντίστοιχα, με χρήση του τελεστή defined, μπορούμε να ελέγξουμε και αν ένα αναγνωριστικό δεν είναι ορισμένο ως μακροεντολή

Π.χ.:

```
#if !defined(VER_1) /* Ισοδύναμο με #ifndef VER_1 */
...
#endif
```

Παρατηρήσεις

- Σημειώνουμε ότι η χρήση παρενθέσεων στον τελεστή **defined** δεν είναι απαραίτητη
- Το πλεονέκτημα της χρήσης του τελεστή **defined** έναντι των **#ifdef** και **#ifndef** είναι ότι με αυτές μπορούμε να ελέγξουμε αν μόνο μία μακροεντολή έχει οριστεί ή όχι, ενώ με τον τελεστή **defined** και την οδηγία **#if** μπορούμε να ελέγξουμε πολλαπλές μακροεντολές, π.χ. :

```
#if defined(VER_1) && !defined(VER_2) && defined(VER_3)
...
#endif
```

ή να χρησιμοποιήσουμε τον τελεστή || και να ελέγξουμε αν κάποια μακροεντολή είναι ορισμένη, π.χ.:

```
#if defined(VER_1) || defined(VER_2)
...
#endif
```

Παράδειγμα (1)

- Δημιουργήστε μία μακροεντολή που να υπολογίζει την απόλυτη τιμή ενός αριθμού. Να γραφεί ένα πρόγραμμα το οποίο να διαβάζει έναν ακέραιο και να εμφανίζει την απόλυτη τιμή του με χρήση της μακροεντολής

Παράδειγμα (1)

```
#include <iostream>

#define abs(a) ((a) >= 0 ? (a) : -(a))

int main()
{
        int i;

        std::cout << "Enter number: ";
        std::cin >> i;
        std::cout << abs(i) << '\n';
        return 0;
}
```

Παράδειγμα (2)

- Αν το περιεχόμενο του test.h είναι το εξής:

```
#include <iostream>
#define TEST
#define f() std::cout << "One ";
#undef TEST
#endif
```

- Ποια θα είναι η έξοδος του παρακάτω προγράμματος;

```
#include "test.h"
int main()
{
        f();
        #ifdef TEST
                f();
        #endif
        f();
        return 0;
}
```

Παράδειγμα (2)

- Απάντηση: Ο προεπεξεργαστής αντικαθιστά την πρώτη εμφάνιση της `f()` με την εντολή `cout` και ακυρώνει τον ορισμό της `TEST`. Αφού η `TEST` δεν είναι πλέον ορισμένη, ο προεπεξεργαστής δεν μεταφράζει τη δεύτερη `f()` και συνεχίζει με την τρίτη. Επομένως, το πρόγραμμα εμφανίζει: `One One`

C++: Από τη Θεωρία στην Εφαρμογή

Κεφάλαιο 16ο

Περισσότερα για Συναρτήσεις

Προκαθορισμένα Ορίσματα

- Ένα προκαθορισμένο όρισμα (default argument) είναι μία τιμή που θα χρησιμοποιηθεί αν λείπει το αντίστοιχο όρισμα στην κλήση της συνάρτησης. Π.χ:

```
#include <iostream>

int g = 20;
void test(int a, int b = 10, int *p = &g);

int main()
{
        int i = 25;
        test(5);
        test(5, 15);
        test(5, 15, &i);
        return 0;
}
void test(int a, int b = 10, int *p = &g)
{
        std::cout << a+b+(*p) << '\n';
}
```

- Στην πρώτη κλήση της `test()` επειδή δεν μεταβιβάζονται τιμές για το δεύτερο και τρίτο όρισμα, χρησιμοποιούνται οι προκαθορισμένες. Δηλαδή, το `b` αρχικοποιείται με `10` και το `p` δείχνει στη διεύθυνση του `g`. Άρα, η `test()` εμφανίζει `35`. Στη δεύτερη κλήση χρησιμοποιείται προκαθορισμένη τιμή μόνο για το τρίτο όρισμα και η `test()` εμφανίζει `40`. Στην τρίτη κλήση δεν χρησιμοποιείται καμία προκαθορισμένη τιμή και η `test()` εμφανίζει `45`.

Παρατηρήσεις

- Οι προκαθορισμένες τιμές χρησιμοποιούνται **μόνο** όταν λείπουν τα αντίστοιχα ορίσματα στην κλήση της συνάρτησης
- Συνήθως, οι προκαθορισμένες τιμές χρησιμοποιούνται όταν τις περισσότερες φορές η συνάρτηση καλείται με τις ίδιες τιμές
- Οι τιμές καθορίζονται στο πρωτότυπο, ώστε ο μεταγλωττιστής να έχει ενημερωθεί πριν συναντήσει την κλήση της συνάρτησης
- Τον ορισμό τον γράφουμε με τον ίδιο τρόπο όπως αν δεν υπήρχαν προκαθορισμένες τιμές. Δηλαδή, οι προκαθορισμένες τιμές καθορίζονται **μόνο** στο πρωτότυπο
- Επιτρέπεται όλες οι παράμετροι σε μία συνάρτηση να έχουν προκαθορισμένες τιμές
- Αν μία παράμετρος δηλωθεί με προκαθορισμένη τιμή, **πρέπει όλες οι επόμενες παράμετροι στα δεξιά** να δηλωθούν και αυτές με προκαθορισμένες τιμές. Για παράδειγμα, οι παρακάτω δηλώσεις δεν επιτρέπονται:

```
void test(int a, int b = 10, int *p); // Λάθος
void test(int a = 20, int b, int *p = &g); // Λάθος
```

Εμβόλιμες Συναρτήσεις

- Οι εμβόλιμες συναρτήσεις (inline functions) είναι ένα χαρακτηριστικό που παρέχει η C++ με σκοπό τη βελτίωση της απόδοσης του προγράμματος
- Όπως είπαμε στο Κ.14, όταν καλείται μία συνάρτηση, ο μεταγλωττιστής αποθηκεύει τη διεύθυνση μνήμης της εντολής στην οποία θα επιστρέψει το πρόγραμμα όταν θα τερματιστεί η συνάρτηση, αν χρειάζεται δεσμεύει μνήμη από τη στοίβα για την αντιγραφή των ορισμάτων, ο έλεγχος του προγράμματος μεταφέρεται στη διεύθυνση μνήμης που βρίσκεται ο κώδικας της συνάρτησης, εκτελείται ο κώδικας της συνάρτησης, και, όταν ολοκληρωθεί, ο έλεγχος του προγράμματος επιστρέφει στη διεύθυνση που αποθηκεύτηκε όταν έγινε η κλήση της
- Αυτές οι λειτουργίες επιφέρουν μία χρονική καθυστέρηση στην εκτέλεση του προγράμματος κάθε φορά που καλείται η συνάρτηση
- Όμως, αν η συνάρτηση δηλωθεί σαν εμβόλιμη μπορούμε να αποφύγουμε αυτή τη χρονική καθυστέρηση και να βελτιώσουμε την απόδοση του προγράμματος
- Συγκεκριμένα, ο μεταγλωττιστής αντικαθιστά κάθε κλήση της συνάρτησης στο πρόγραμμα με τον κώδικά της
- Για τον χαρακτηρισμό μίας συνάρτησης σαν εμβόλιμης πρέπει να χρησιμοποιηθεί η λέξη `inline`

Παράδειγμα

- Έστω ο παρακάτω κώδικας:

```
inline void test()
{
        for(int i = 0; i < 10; i++)
                cout << "In\n";
}
int main()
{
        ...
        test();
        test();
        ...
}
```

ο μεταγλωττιστής αντικαθιστά κάθε κλήση της `test()` με τον κώδικά της:

```
int main()
{
    ...
    {
       for(int i = 0; i < 10; i++)
          cout << "In\n";
    }
    {
       for(int i = 0; i < 10; i++)
          cout << "In\n";
    }
    ...
}
```

Παράδειγμα

- Αφού αποφεύγονται οι κλήσεις, η απόδοση του προγράμματος βελτιώνεται. Όπως είπαμε στο Κ.15, οι μακροεντολές εξυπηρετούν παρόμοιο σκοπό. Θυμηθείτε όμως τις αδυναμίες και τα προβλήματα που μπορεί να προκύψουν εξαιτίας της λανθασμένης χρήσης τους. Οι εμβόλιμες συναρτήσεις είναι μία πολύ καλύτερη εναλλακτική. Είναι πιο ασφαλείς αφού παρέχουν έλεγχο τύπων και η χρήση τους και η σύνταξή τους είναι απλή, όπως με μία συνηθισμένη συνάρτηση

Παρατηρήσεις

- Αφού σε κάθε κλήση μίας εμβόλιμης συνάρτησης εισάγεται ο κώδικάς της, το μέγεθος του εκτελέσιμου προγράμματος αυξάνεται
- Γενικά, όταν μία συνάρτηση γίνεται `inline` μην νομίζετε ότι η απόδοση του προγράμματος θα βελτιωθεί οπωσδήποτε, μπορεί ναι μπορεί και όχι, εξαρτάται από την εφαρμογή
- Συνήθως, οι υποψήφιες συναρτήσεις για να γίνουν `inline` είναι μικρές συναρτήσεις που καλούνται συχνά στο πρόγραμμα
- Επίσης, αν δηλώσετε μία συνάρτηση σαν `inline` δεν σημαίνει ότι ο μεταγλωττιστής θα ικανοποιήσει το αίτημά σας και θα κάνει την συνάρτηση εμβόλιμη
- Το προσδιοριστικό `inline` είναι απλά μία υπόδειξη προς τον μεταγλωττιστή. Δηλαδή, μπορεί να αγνοήσει την υπόδειξη και να τη χειριστεί όπως μία συνηθισμένη συνάρτηση
- Για παράδειγμα, αν ο μεταγλωττιστής θεωρήσει ότι ο κώδικάς της είναι αρκετά μεγάλος ή πολύπλοκος μπορεί να αγνοήσει την `inline` υπόδειξη και να χρησιμοποιήσει τον συμβατικό τρόπο για να καλέσει την συνάρτηση
- Και ναι, ένας έξυπνος μεταγλωττιστής μπορεί να κρίνει ότι είναι πιο αποδοτικό και να κάνει μία συνάρτηση εμβόλιμη ακόμα και αν εσείς δεν το έχετε δηλώσει

Αναφορικές Μεταβλητές (1)

- Μία αναφορική μεταβλητή (reference variable) αποτελεί συνώνυμο μίας υπάρχουσας μεταβλητής
- Όπως γνωρίζουμε, η C++ χρησιμοποιεί τον τελεστή & για να εξάγει την διεύθυνση κάποιας μεταβλητής
- Η C++ προσδίδει μία επιπλέον ιδιότητα στον τελεστή & για τη δήλωση αναφορικών μεταβλητών
- Ο μεταγλωττιστής ελέγχει την έκφραση για να συμπεράνει τον τρόπο που χρησιμοποιείται

Αναφορικές Μεταβλητές (2)

- Συγκεκριμένα, για να δηλώσουμε μία αναφορική μεταβλητή προσθέτουμε το & πριν από το όνομά της και την αρχικοποιούμε με μία υπάρχουσα μεταβλητή. Για παράδειγμα:

```
int i = 10;
int& r = i;
r = 20; // Ισοδύναμο με i = 20.
i = 30; // Ισοδύναμο με r = 30.
```

Στη δήλωση της αναφορικής μεταβλητής r, το & δεν χρησιμοποιείται σαν τελεστής διεύθυνσης, αλλά για τη δήλωση του τύπου. Δηλαδή, ο τύπος του r είναι int&, που σημαίνει αναφορά σε ακέραιο. Ουσιαστικά, η μεταβλητή r αποτελεί συνώνυμο, ένα δεύτερο όνομα για το i. Η r θα αναφέρεται πάντα στην τιμή και στη θέση μνήμης του i. Είναι μόνιμα συνδεδεμένη με την i, δηλαδή, δεν επιτρέπεται να αναφερθεί σε άλλη μεταβλητή. Η συνήθης πρακτική είναι το & να προστίθεται δίπλα στον τύπο (π.χ. αντί για int &r = i), ώστε να είναι πιο ξεκάθαρο ότι οι δύο μεταβλητές έχουν την ίδια τιμή, αλλά και να αποφεύγεται η σύγχυση με την εξαγωγή της διεύθυνσης

Παράδειγμα

- Ποια είναι η έξοδος του παρακάτω προγράμματος;

```
#include <iostream>
int main()
{
        int i = 10, j = 20;
        int& r = i;

        r = j;
        if(&r == &j)
                std::cout << "Yes\n";
        else
                std::cout << "No\n";
        if(&i == &r)
                std::cout << "Yes\n";
        else
                std::cout << "No\n";
        std::cout << i+r << '\n';
        return 0;
}
```

Παράδειγμα

- Απάντηση. Η εντολή `r = j;` δεν κάνει το `r` να αναφερθεί στο `j`, απλά αλλάζει την τιμή του `r`. Αφού το `r` με τη δήλωσή του αναφέρεται στο `i`, η εντολή είναι ισοδύναμη με `i = j;`. Άρα, οι τιμές των `i` και `r` γίνονται `20`. Προσέξτε ότι στις `if` εντολές το `&` είναι ο τελεστής διεύθυνσης. Είναι οι διευθύνσεις των `r` και `j` ίδιες; Όχι βέβαια, η διεύθυνση του `r` είναι ίδια με του `i`. Άρα, το πρόγραμμα εμφανίζει `No`, `Yes` και `40`

Κλήση Συνάρτησης με Αναφορικές Παραμέτρους

- Όπως έχουμε μάθει, όταν θέλουμε μία συνάρτηση να αλλάξει την τιμή κάποιου ορίσματος μεταβιβάζουμε δείκτη στην αντίστοιχη μεταβλητή. Ένας εναλλακτικός τρόπος είναι να της μεταβιβαστεί μία αναφορά σε αυτό. Για παράδειγμα:

```
#include <iostream>

void test(int& a, int b);

int main()
{
        int i = 10, j = 50;

        test(i, j);
        std::cout << i << ' ' << j << '\n';
        return 0;
}

void test(int& a, int b)
{
        a = 100;
        b = 200;
}
```

Κλήση Συνάρτησης με Αναφορικές Παραμέτρους

- Όταν σε μία συνάρτηση μεταβιβάζεται μία αναφορά σε μία μεταβλητή, ουσιαστικά μεταβιβάζεται η διεύθυνση μνήμης της μεταβλητής. Έτσι, η συνάρτηση μπορεί να προσπελάσει την πρωτότυπη μεταβλητή
- Για παράδειγμα, όταν καλείται η `test()`, η μεταβλητή `i` μεταβιβάζεται με αναφορά. Αυτή η μέθοδος μεταβίβασης δεδομένων ονομάζεται **κλήση μέσω αναφοράς** (call by reference)
- Προσέξτε ότι ο τελεστής `&` προστίθεται μόνο στη δήλωση της συνάρτησης, όχι στην κλήση της
- Στην `test()`, η μεταβλητή `a` αναφέρεται στην `i`, δηλαδή, όταν χρησιμοποιείται η `a` είναι σαν να χρησιμοποιείται η ίδια η `i`. Με άλλα λόγια, το `a` αποτελεί **συνώνυμο** για το `i`, όχι αντίγραφο του `i`
- Αντίθετα, όπως ήδη γνωρίζουμε, η μεταβλητή `j` μεταβιβάζεται **μέσω τιμής** (call by value), δηλαδή, η `b` αποτελεί αντίγραφο της `j`
- Άρα, η τιμή της `j` δεν αλλάζει και το πρόγραμμα εμφανίζει: `100 50`
- Αν δεν θέλουμε η συνάρτηση να μπορεί να αλλάξει την πρωτότυπη τιμή χρησιμοποιούμε τη λέξη `const`. Για παράδειγμα, αν γράψουμε:

```
void test(const int& a, int b)
```

η εντολή `a = 100;` δεν επιτρέπεται. Η δήλωση ενός ορίσματος-αναφοράς σαν `const` εμποδίζει οποιαδήποτε αλλαγή στην τιμή της αναφερόμενης μεταβλητής

Παρατηρήσεις

- Ίσως για κάποιους, η χρήση αναφορικών παραμέτρων αντί για δείκτες να φαίνεται πιο απλή και κατανοητή. Εσείς τι προτιμάτε;
- Στην ερώτηση πότε να χρησιμοποιώ δείκτη ή αναφορά, αυτό που συνηθίζεται είναι να χρησιμοποιούνται δείκτες όταν στη συνάρτηση μεταβιβάζονται βασικοί τύποι (π.χ. int), δομές και ενώσεις, και, επειδή η C++ συχνά απαιτεί την χρήση αναφορών στη σχεδίαση των κλάσεων, να χρησιμοποιούνται αναφορές όταν στη συνάρτηση μεταβιβάζονται αντικείμενα κλάσεων
- Ανεξάρτητα από την επιλογή σας, το σίγουρο είναι ότι πρέπει να γνωρίζετε και τους δύο τρόπους, γιατί και οι δύο είναι αρκετά δημοφιλείς
- Και όπως ήδη γνωρίζετε από το Κ.13, όταν στη συνάρτηση μεταβιβάζονται μεγάλες οντότητες, όπως δομές και κλάσεις, επιλέγουμε να μεταβιβάζουμε αναφορά ή δείκτη ακόμα και αν η συνάρτηση δεν πρόκειται να τις τροποποιήσει. Και αυτό για να γλιτώσουμε τον χρόνο και την μνήμη που απαιτείται για τη δημιουργία του αντιγράφου της πρωτότυπης οντότητας. Έτσι, η απόδοση του προγράμματος βελτιώνεται. Για απλές μεταβλητές, όπως βασικοί τύποι, η συνήθης επιλογή είναι η κλήση μέσω τιμής, εφόσον βέβαια δεν θέλουμε η συνάρτηση να τροποποιεί την πρωτότυπη τιμή.

Υπερφόρτωση Συνάρτησης

- Η C++ επιτρέπει διαφορετικές συναρτήσεις να έχουν το **ίδιο όνομα**. Αυτό το χαρακτηριστικό ονομάζεται **υπερφόρτωση** συνάρτησης (function overloading) ή **πολυμορφισμός** συνάρτησης (function polymorphism) με την έννοια ότι μία συνάρτηση μπορεί να έχει πολλές μορφές
- Η υπερφόρτωση συνάρτησης αποτελεί μία από τις σημαντικές ενότητες του γενικού προγραμματισμού
- Ένα παράδειγμα της έννοιας της υπερφόρτωσης είναι η χρήση αριθμητικών τελεστών. Για παράδειγμα, ο ίδιος + τελεστής μπορεί να χρησιμοποιηθεί για την πρόσθεση ακεραίων ή πραγματικών αριθμών ή και συνδυασμό τους. Αυτή η ιδέα επεκτάθηκε και στις συναρτήσεις

Υπογραφή Συνάρτησης

- Ο περιορισμός για την ύπαρξη πολλών συναρτήσεων με το ίδιο όνομα είναι ότι κάθε συνάρτηση πρέπει να έχει **διαφορετική υπογραφή** (function signature). Η υπογραφή μίας συνάρτησης **καθορίζεται** από τον αριθμό των παραμέτρων της και τους τύπους τους, τα ονόματα των παραμέτρων δεν παίζουν ρόλο
- Όταν καλείται μία υπερφορτωμένη συνάρτηση, ο μεταγλωττιστής ελέγχει το πλήθος και τον τύπο των ορισμάτων που της μεταβιβάζονται και **επιλέγει** την κατάλληλη έκδοση για να καλέσει
- Δηλαδή, ο πολυμορφισμός επιτυγχάνεται κατά τη **μεταγλώττιση** του προγράμματος (static binding)

Παράδειγμα

- Αν και μπορούμε να υπερφορτώσουμε συναρτήσεις οι οποίες να επιτελούν εντελώς διαφορετικές λειτουργίες, η υπερφόρτωση έχει νόημα όταν χρησιμοποιείται με συναρτήσεις που επιτελούν **παρόμοιες** λειτουργίες. Έτσι, δεν χρειάζεται να ψάχνουμε για διαφορετικά ονόματα συναρτήσεων και να δίνουμε την ψευδή εντύπωση στον αναγνώστη ότι επιιτελούν διαφορετικές λειτουργίες. Το ίδιο όνομα υποδεικνύει στον αναγνώστη ότι η λειτουργία τους είναι παρόμοια, αλλά με διαφορετικούς τύπους δεδομένων. Για παράδειγμα, στο παρακάτω πρόγραμμα η συνάρτηση **abs_v()** υπερφορτώνεται τρεις φορές και ο μεταγλωττιστής επιλέγει την κατάλληλη έκδοση:

```
#include <iostream>

int abs_v(int a);
double abs_v(double a);
void abs_v(int a, int b); /* Εμφάνιση της απόλυτης τιμής του αθροίσματος. */

int main()
{
        int i = -100;
        double j = 1.2345;

        std::cout << abs_v(i) << ' ';
        std::cout << abs_v(j) << ' ';
        abs_v(i, 30);
        return 0;
}
```

Παράδειγμα

```
#include int abs_v(int a)
{
        if(a < 0)
                return -a;
        else
                return a;
}

double abs_v(double a)
{
        if(a < 0)
                return -a;
        else
                return a;
}

void abs_v(int a, int b)
{
        int sum;

        sum = a+b;
        if(sum < 0)
                std::cout << -sum << '\n';
        else
                std::cout << sum << '\n';
}
```

Το πρόγραμμα εμφανίζει: `100 1.2345 70`. Για όσους έχουν εμπειρία με τη *C*, επειδή η *C* δεν υποστηρίζει υπερφόρτωση, για να έχουμε το ίδιο αποτέλεσμα, θα πρέπει να ορίσουμε τρεις συναρτήσεις με διαφορετικά ονόματα όπως `abs()`, `fabs()` και `labs()`. Η σειρά των υπερφορτωμένων συναρτήσεων δεν παίζει ρόλο στην επιλογή της κατάλληλης έκδοσης

Παρατηρήσεις

- Σημειώστε ότι **μόνο** η υπογραφή, και όχι ο τύπος επιστροφής, επιτρέπει την υπερφόρτωση συναρτήσεων
- Να θυμόμαστε λοιπόν ότι για να υπερφορτώσουμε μία συνάρτηση, οι συναρτήσεις πρέπει να έχουν **διαφορετικές υπογραφές**, δηλαδή, διαφορετικό αριθμό παραμέτρων ή τύπων ή και τα δύο
- Ο μεταγλωττιστής **δεν** ελέγχει τους τύπους επιστροφής. Για παράδειγμα, η υπογραφή της τρίτης `abs_v()` είναι (int, int). Επομένως, αν προσθέσουμε την παρακάτω `abs_v()`:

```
int abs_v(int a, int b);
```

ο μεταγλωττιστής θα εμφανίσει μήνυμα λάθους για δύο εκδόσεις με την ίδια υπογραφή

Πρότυπες Συναρτήσεις (1)

- Οι **πρότυπες** συναρτήσεις αποτελούν την βάση του **γενικευμένου προγραμματισμού** (generic programming), δηλαδή, να γράψουμε κώδικα ο οποίος να λειτουργεί για μία **ποικιλία τύπων**
- Ένα πρότυπο συνάρτησης (function template) μας επιτρέπει να ορίσουμε μία συνάρτηση με **γενικό** τρόπο, ώστε να μπορεί να χειριστεί διαφορετικούς τύπους δεδομένων
- Κάθε φορά που καλείται μία πρότυπη συνάρτηση, ο μεταγλωττιστής ελέγχει τον τύπο των ορισμάτων και **δημιουργεί** μία έκδοση της συνάρτησης με τους συγκεκριμένους τύπους των ορισμάτων. Ένα παράδειγμα ορισμού μίας πρότυπης συνάρτησης είναι:

```
template <typename τύπος1, typename τύπος2, ...>
τύπος_επιστροφής όνομα_συνάρτησης(λίστα_παραμέτρων)
```

- Για παράδειγμα, ας υποθέσουμε ότι θέλουμε να δημιουργήσουμε μία συνάρτηση που να επιστρέφει την απόλυτη τιμή ενός αριθμού. Επειδή δεν γνωρίζουμε τον τύπο του αριθμού θα πρέπει να γράψουμε διαφορετικές συναρτήσεις ή να υπερφορτώσουμε μία συνάρτηση για όλους τους πιθανούς τύπους
- Δείτε τώρα την **ευελιξία** που μας παρέχουν τα πρότυπα. Με τα πρότυπα δεν χρειάζεται να γράψουμε διαφορετικές εκδόσεις, αρκεί να γράψουμε **μία** πρότυπη συνάρτηση με **γενικό** τρόπο και αυτή θα μπορεί να λειτουργεί για μία ποικιλία τύπων. Αυτή είναι η έννοια του γενικευμένου προγραμματισμού

Πρότυπες Συναρτήσεις (2)

- Για να δηλώσουμε μία πρότυπη συνάρτηση γράφουμε τη λέξη `template` και μέσα σε `<>` τη λέξη `typename` και τα ονόματα των πρότυπων παραμέτρων
- Αντίθετα από τις παράμετρους μίας συνηθισμένης συνάρτησης οι οποίες χρησιμοποιούνται για τη μεταβίβαση τιμών, οι παράμετροι μίας πρότυπης συνάρτησης συνήθως χρησιμοποιούνται για τη μεταβίβαση τύπων
- Για τα ονόματα των παραμέτρων μπορείτε να επιλέξετε όποια ονόματα θέλετε, αρκεί να ακολουθούν τους κανόνες ονοματοδοσίας. Η συνήθης πρακτική είναι να επιλέγονται μικρά ονόματα, τα οποία αρχίζουν με το γράμμα `T` (π.χ. `T1`, `T2`)
- Όσον αφορά την οργάνωση του προγράμματος, σε μία εφαρμογή με πολλά αρχεία κώδικα τα οποία μπορεί να χρησιμοποιούν το ίδιο πρότυπο, η συνήθης πρακτική είναι να τοποθετείται ο ορισμός της πρότυπης συνάρτησης σε ένα αρχείο επικεφαλίδας, το οποίο να συμπεριλαμβάνεται με την οδηγία `#include` σε όποιο αρχείο κώδικα την χρησιμοποιεί

Παράδειγμα

Το επόμενο πρόγραμμα χρησιμοποιεί την πρότυπη συνάρτηση abs() για να υπολογίζει την απόλυτη τιμή αριθμών με διαφορετικούς τύπους:

```
#include <iostream>

template <typename T> T abs(T a);

int main()
{
        int a = -6;
        float b = -1.2;
        double c = 3.4;

        std::cout << abs(a) << ' ' << abs(b) << ' ' << abs(c) << '\n';
        return 0;
}
template <typename T> T abs(T a)
{
        T k;

        if(a < 0)
                k = -a;
        else
                k = a;
        return k;
}
```

Το πρόγραμμα εμφανίζει: 6 1.2 3.4

Παράδειγμα

- Το πρότυπο δεν αποτελεί ορισμό κάποιας συνάρτησης. Απλά, το πρότυπο παρέχει τις οδηγίες για τον μεταγλωττιστή για το πώς να ορίσει τη συνάρτηση
- Πότε λοιπόν δημιουργείται η συνάρτηση; Κάθε φορά που ο μεταγλωττιστής συναντάει την `abs()` αντικαθιστά τον τύπο `T` με τον τύπο του ορίσματος και δημιουργεί τη συνάρτηση. Για παράδειγμα, στην πρώτη κλήση, αντικαθιστά τον τύπο `T` με `int` και δημιουργεί μία `int` έκδοση της συνάρτησης. Άρα, ο ορισμός που προκύπτει είναι:

```
int abs(int a)
{
        int k;
        if(a < 0)
                k = -a;
        else
                k = a;
        return k;
}
```

- Παρόμοια, στη δεύτερη κλήση, ο μεταγλωττιστής αντικαθιστά τον τύπο `T` με `float` και ο ορισμός της `float` έκδοσης που προκύπτει είναι:

```
float abs(float a)
{
        float k;
        if(a < 0)
                k = -a;
        else
                k = a;
        return k;
}
```

Παρατηρήσεις (1)

- Ο μεταγλωττιστής δημιουργεί έναν ορισμό της πρότυπης συνάρτησης μόνο αν αυτή χρησιμοποιηθεί στο πρόγραμμα. Αν δεν χρησιμοποιηθεί, δεν θα οριστεί
- Σε κάθε κλήση ο μεταγλωττιστής ελέγχει τον τύπο του ορίσματος, αντικαθιστά κάθε εμφάνιση του γενικού τύπου με αυτόν και δημιουργεί την ανάλογη έκδοση
- Αυτός ο τύπος συγκεκριμενοποίησης της συνάρτησης ονομάζεται **έμμεση συγκεκριμενοποίηση** (implicit instantiation), με την έννοια ότι ο μεταγλωττιστής δημιουργεί την συνάρτηση όταν **συναντήσει** την κλήση της και **συμπεράνει** τον **τύπο** των ορισμάτων που μεταβιβάζονται
- Αν κάποια άλλη κλήση της συνάρτησης απαιτεί το **ίδιο** στιγμιότυπο, ο μεταγλωττιστής χρησιμοποιεί την **υπάρχουσα** έκδοση

Παρατηρήσεις (2)

- Αφού ο τύπος των ορισμάτων είναι καθορισμένος και γνωστός, η συγκεκριμενοποίηση των προτύπων γίνεται όταν **μεταγλωττίζεται** το πρόγραμμα
- Άρα, με τα πρότυπα, όπως και στην υπερφόρτωση συναρτήσεων, ο πολυμορφισμός επιτυγχάνεται κατά τη μεταγλώττιση (compile-time polymorphism) και όχι κατά την εκτέλεση του προγράμματος, το οποίο σημαίνει ότι η χρήση προτύπων δεν επιβαρύνει τον χρόνο εκτέλεσης του προγράμματος
- Σημειώστε ότι οι πρότυπες συναρτήσεις δεν κάνουν πιο μικρό το μέγεθος του εκτελέσιμου αρχείου. Για παράδειγμα, στο προηγούμενο πρόγραμμα, το εκτελέσιμο περιέχει τους ορισμούς των τριών συναρτήσεων που παράχθηκαν, όπως αν τους είχαμε γράψει ξεχωριστά

Παρατηρήσεις (3)

- Όπως είπαμε, το πλεονέκτημα των προτύπων είναι ότι δε χρειάζεται να γράψουμε τον ίδιο κώδικα για διαφορετικούς τύπους δεδομένων. Τον γράφουμε μόνο μία φορά με **γενικό** τρόπο
- Έτσι, ο κώδικας διαβάζεται, ελέγχεται και διατηρείται πιο εύκολα, αφού εμφανίζεται μόνο σε ένα σημείο
- Επειδή με τα πρότυπα έχουμε τη δυνατότητα να γράψουμε προγράμματα που να χρησιμοποιούν γενικούς τύπους αντί για συγκεκριμένους τύπους, λέμε ότι η C++ υποστηρίζει τον **γενικό προγραμματισμό**
- Όσον αφορά τις παραμέτρους μίας πρότυπης συνάρτησης υπάρχουν πολλές επιλογές. Για παράδειγμα, δεν είναι απαραίτητο να είναι όλες γενικοί τύποι. Μπορούμε να τις συνδυάσουμε με συγκεκριμένους τύπους, όπως παρακάτω:

```
template <typename T> void f(T a, int b)
```

Πρότυπες Παράμετροι με Διαφορετικούς Τύπους

- Οι πρότυπες παράμετροι μπορεί να έχουν διαφορετικούς τύπους. Για παράδειγμα, ας δημιουργήσουμε μία πρότυπη συνάρτηση που να εμφανίζει τον μεγαλύτερο από δύο αριθμούς:

```
#include <iostream>

template <typename T1, typename T2> void f(T1 a, T2 b);

int main()
{
        int i = 10, j = 20;
        float k = 1.23;
        double p = 5.64;

        f(i, j); // T1=int, T2=int.
        f(p, i); // T1=double, T2=int.
        f(j, k); // T1=int, T2=float.
        return 0;
}

template <typename T1, typename T2> void f(T1 a, T2 b)
{
        if(a < b)
                std::cout << a << '\n';
        else
                std::cout << b << '\n';
}
```

Πρότυπες Παράμετροι με Διαφορετικούς Τύπους

- Για παράδειγμα, στη δεύτερη κλήση, ο μεταγλωττιστής αντικαθιστά τον τύπο `T1` με `double`, τον τύπο `T2` με `int`, και ο ορισμός της έκδοσης που προκύπτει είναι:

```
void f(double a, int b)
{
        if(a < b)
                std::cout << a << '\n';
        else
                std::cout << b << '\n';
}
```

Παράδειγμα

- Δημιουργήστε μία πρότυπη συνάρτηση που να δέχεται σαν παραμέτρο ένα `vector` αντικείμενο με αριθμητικά στοιχεία γενικού τύπου και να επιστρέφει τον μέσο όρο τους. Υπερφορτώστε την συνάρτηση με μία άλλη πρότυπη συνάρτηση με τύπο επιστροφής `void`, η οποία να δέχεται δύο αριθμητικές παραμέτρους που μπορεί να έχουν διαφορετικούς τύπους και να επιστρέφει μέσω δείκτη τον μέσο όρο τους. Να γραφεί ένα πρόγραμμα το οποίο να ελέγχει την λειτουργία των δύο συναρτήσεων.

Παράδειγμα

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

template <typename T> double avg(const vector<T>& v);
template <typename T1, typename T2> void avg(T1 t1, T2 t2, double *p);

int main()
{
        double i;
        vector<int> v = {1, 2, 4};

        avg(1, 2, &i);
        std::cout << "Avg_1:" << avg(v) << '\n';
        std::cout << "Avg_2:" << i << '\n';
        return 0;
}

template <typename T> double avg(const vector<T>& v)
{
        T sum; // Μεταβλητή γενικού τύπου
        int i, size;

        size = v.size();
        sum = 0;
        for(i = 0; i < size; i++)
                sum += v[i];
        return (double)sum/size;
}
template <typename T1, typename T2> void avg(T1 t1, T2 t2, double *p)
{
        *p = (t1+t2)/2.0;
}
```

C++: Από τη Θεωρία στην Εφαρμογή

Κεφάλαιο 17ο

Κλάσεις και Αντικείμενα

Κλάση

- Ο τύπος κλάση (class) είναι παρόμοιος με τον τύπο της δομής
- Όπως και μία δομή, μία κλάση είναι ένας **τύπος δεδομένων** που ορίζεται από τον προγραμματιστή (user-defined type), για να προσδιορίσει τα χαρακτηριστικά και τις λειτουργίες μίας αφηρημένης οντότητας
- Μία κλάση περιέχει ένα σύνολο δεδομένων το οποίο περιγράφει μία οντότητα σύμφωνα με τις ανάγκες της εφαρμογής και μπορεί επίσης να περιέχει ένα σύνολο συναρτήσεων οι οποίες να εφαρμόζονται στα δεδομένα
- Αυτή η συνύπαρξη δεδομένων και συναρτήσεων στην κλάση αναφέρεται σαν **ενθυλάκωση** δεδομένων (data encapsulation)
- Γενικά, μπορούμε να χρησιμοποιήσουμε κλάσεις για να αναπαραστήσουμε όποια οντότητα του πραγματικού κόσμου επιθυμούμε και να γράψουμε προγράμματα που να βασίζονται σε αυτές
- Ουσιαστικά, η κλάση είναι η **καρδιά** του αντικειμενοστρεφούς προγραμματισμού

Δήλωση Κλάσης

- Όπως είπαμε, η κλάση είναι το βασικό δομικό στοιχείο για να αναπαραστήσουμε μία οντότητα στο πρόγραμμα. Για παράδειγμα, για να αναπαραστήσουμε μία τηλεφωνική κλήση θα μπορούσαμε να ορίσουμε μία κλάση, η οποία να περιέχει πληροφορίες για αυτήν, όπως τον αριθμό του καλούντα συνδρομητή, τον αριθμό του καλούμενου, τη διάρκεια της κλήσης και συναρτήσεις για την εγκατάσταση, την αποδέσμευση και χρέωση της κλήσης
- Γενικά, ο προγραμματιστής αποφασίζει το τι θα περιέχει μία κλάση ανάλογα με τον τρόπο που σκοπεύει να χρησιμοποιηθεί η κλάση
- Η γενική μορφή δήλωσης μίας κλάσης είναι:

```
class όνομα_κλάσης
{
        Προσβασιμότητα: Δηλώσεις;
        ...
        Προσβασιμότητα: Δηλώσεις;
};
```

- Η κοινή πρακτική είναι η δήλωση μίας κλάσης να τοποθετείται σε ένα αρχείο επικεφαλίδας, που συνήθως έχει το ίδιο όνομα με το όνομα της κλάσης, ενώ η υλοποίηση των συναρτήσεων της κλάσης σε ένα ή περισσότερα αρχεία κώδικα τα οποία κάνουν `#include` το αρχείο επικεφαλίδας. Με τον διαχωρισμό της δήλωσης της κλάσης από την υλοποίηση της λειτουργικότητας της, ο κώδικας γίνεται πιο εύκολο να γραφεί και να συντηρηθεί

Αντικείμενα Κλάσης

- Παρόμοια με μία ονομαστική δομή, αφού δηλωθεί μία κλάση, μπορούμε να δηλώσουμε μεταβλητές αυτού του τύπου
- Αυτές οι μεταβλητές ονομάζονται **αντικείμενα** (objects) ή **στιγμιότυπα** (instances) της κλάσης και δημιουργούνται με βάση τις προδιαγραφές που ορίζει η κλάση
- Κάθε αντικείμενο αποτελεί μία υπόσταση μίας **συγκεκριμένης** κλάσης και η πληροφορία που περιέχει είναι αποθηκευμένη στα μέλη του
- Η κλάση απλά **περιγράφει** την οντότητα, είναι μία **γενική** έννοια που δεν αναφέρεται σε κάποιο συγκεκριμένο αντικείμενο. Να το ξεκαθαρίσουμε λοιπόν, η **δήλωση μίας κλάσης**, όπως και η **δήλωση μίας δομής**, **δεν δημιουργεί** κάποιο αντικείμενο
- Για παράδειγμα, ένα συγκεκριμένο μοντέλο αυτοκινήτου αποτελεί αντικείμενο της κλάσης που περιγράφει την οντότητα αυτοκίνητο. Μπορούμε να δημιουργήσουμε και να χειριστούμε όσα τέτοια αντικείμενα επιθυμούμε και καθένα από αυτά θα έχει τις δικές του ιδιότητες

Παράδειγμα

- Όπως και με τις δομές, μπορούμε να μεταβιβάσουμε ένα αντικείμενο σε μία συνάρτηση, να ορίσουμε μία συνάρτηση που να επιστρέφει αντικείμενο και να εκχωρήσουμε ένα αντικείμενο σε ένα άλλο της ίδιας κλάσης
- Για παράδειγμα, το παρακάτω πρόγραμμα δηλώνει την κλάση **Student**, διαβάζει τα στοιχεία ενός φοιτητή, τα αποθηκεύει στα αντίστοιχα μέλη ενός αντικειμένου και τα εμφανίζει στην οθόνη

```
// student.h
#ifndef STUDENT_H
#define STUDENT_H

#include <string>

class Student
{
private:
        int code;
public:
        std::string name;
        float grd;

        void set(int c) {code = c;}
        void show();
};

#endif
```

Παράδειγμα

```
#include <iostream>
#include "student.h"
using std::cin;
using std::cout;

void Student::show()
{
        cout << "N:" << name << " C:" << code << " G:" << grd << '\n';
}

int main()
{
        int code;
        class Student s; /* Το s είναι αντικείμενο της κλάσης Student. Η λέξη
class δεν είναι απαραίτητη. */

        cout << "Name: ";
        getline(cin, s.name);

        cout << "Grade: ";
        cin >> s.grd;

        cout << "Code: ";
        cin >> code;
        s.set(code);
        s.show();
        return 0;
}
```

Παράδειγμα

- Όπως και με τα ονόματα δομών, το πρώτο γράμμα της κλάσης συνηθίζεται να είναι κεφάλαιο
- Όταν δηλώνεται ένα αντικείμενο η λέξη `class` δεν χρειάζεται, γιατί το όνομα της κλάσης αποτελεί όνομα τύπου. Για παράδειγμα, το όνομα `Student` αποτελεί το όνομα ενός νέου τύπου το οποίο ορίστηκε από τον προγραμματιστή
- Μία κλάση μπορεί να περιέχει και αντικείμενα άλλων κλάσεων που είτε εμείς έχουμε δημιουργήσει είτε άλλοι προγραμματιστές. Για παράδειγμα, το πεδίο `name` είναι αντικείμενο της κλάσης βιβλιοθήκης `string`. Άρα, είμαστε χρήστες της κλάσης `string` και έχουμε πρόσβαση, όπως θα δούμε στη συνέχεια, στο τμήμα που οι προγραμματιστές της `string` κλάσης αποφάσισαν να κάνουν δημόσιο
- Κάθε φορά που δημιουργείται ένα νέο αντικείμενο, ο μεταγλωττιστής δεσμεύει μνήμη για τις μεταβλητές του. Για παράδειγμα, αν δηλώσουμε τα αντικείμενα `s1` και `s2`, η μνήμη που έχει δεσμευτεί για τα `s1.grd` και `s2.grd` είναι διαφορετική
- Αντίθετα, η μνήμη που δεσμεύεται για τις συναρτήσεις είναι κοινή. Για παράδειγμα, ο κώδικας της `show()` βρίσκεται αποθηκευμένος μόνο σε ένα σημείο, δεν δημιουργούνται αντίγραφά του κάθε φορά που δηλώνεται ένα νέο αντικείμενο. Και αυτό είναι λογικό, αφού ο κώδικας των συναρτήσεων είναι κοινός για όλα τα αντικείμενα. Δηλαδή, οι εντολές `s1.show()` και `s2.show()` εκτελούν τον ίδιο κώδικα και αυτός ο κώδικας εφαρμόζεται στα δεδομένα του αντικειμένου που κάνει την κλήση

Συναρτήσεις Μέλη (1)

- Συναρτήσεις που ανήκουν σε μία κλάση ονομάζονται συναρτήσεις μέλη (member functions)
- Μία συνάρτηση μπορεί να οριστεί **μέσα ή έξω** από την κλάση. Συνήθως, οι συναρτήσεις ορίζονται έξω από την κλάση σε ξεχωριστό(ά) αρχείο(α) κώδικα. Αυτή η διάκριση κάνει ξεκάθαρο το **τι κάνει** η κλάση (δηλώσεις συναρτήσεων) από το **πώς το κάνει** (ορισμοί συναρτήσεων)
- Όπως και με τις δομές, για να προσπελάσουμε τα μέλη ενός αντικειμένου χρησιμοποιούμε τους τελεστές τελεία (.) και ->. Δηλαδή, προσπελαύνουμε τις συναρτήσεις με τον ίδιο τρόπο όπως και τα απλά πεδία
- Επειδή διαφορετικές κλάσεις μπορεί να έχουν συναρτήσεις με το ίδιο όνομα, για να την ορίσουμε χρησιμοποιούμε τον τελεστή **επίλυσης εμβέλειας** : : (scope resolution operator) και το όνομα της κλάσης στην οποία ανήκει. Έτσι, προσδιορίζεται σε ποια κλάση ανήκει η συνάρτηση και ο μεταγλωττιστής μπορεί να μεταγλωττίσει τον κώδικα

Συναρτήσεις Μέλη (2)

- Κάθε συνάρτηση που ορίζεται μέσα στην κλάση θεωρείται **εμβόλιμη** (inline). Έτσι, δεν χρειάζεται να προστεθεί η λέξη `inline` στον ορισμό της. Δηλαδή, η `set()` στο παράδειγμα είναι εμβόλιμη
- Γενικά, η συνήθης πρακτική είναι να ορίζονται μέσα στην κλάση συναρτήσεις που είναι μικρές
- Σημειώστε ότι μία συνάρτηση που έχει οριστεί έξω από την κλάση μπορεί και αυτή να γίνει εμβόλιμη προσθέτοντας τη λέξη `inline` στον ορισμό της. Για παράδειγμα:

```
inline void Student::show()
```

Εμβέλεια Κλάσης

- Κάθε κλάση ορίζει μέσα στα { } τη δική της **εμβέλεια** (class scope)
- Η εμβέλεια των μελών μίας κλάσης περιορίζεται στην κλάση στην οποία ανήκουν. Επομένως, τα ίδια ονόματα μπορούν να χρησιμοποιηθούν και σε άλλες κλάσεις. Για παράδειγμα, μία κλάση C μπορεί να περιέχει τη δική της show():

```
int C::show()
```

- Επειδή ακριβώς τα ονόματα των μελών δεν είναι ορατά έξω από την κλάση πρέπει να χρησιμοποιούμε το όνομα της κλάσης. Με τη χρήση του ονόματος της κλάσης και του τελεστή :: ο ορισμός της κάθε show() συσχετίζεται με την αντίστοιχη κλάση
- Όταν καλείται μία συνάρτηση, τα μέλη αναφέρονται στο αντικείμενο που κάλεσε την συνάρτηση. Για παράδειγμα, όταν καλείται η show(), εμφανίζονται οι τιμές των μελών του αντικειμένου που κάλεσε την show()

Έλεγχος Πρόσβασης (1)

- Τα μέλη μίας κλάσης μπορεί να είναι δημόσια (`public`), ιδιωτικά (`private`) ή προστατευμένα (`protected`)
- Τα **ιδιωτικά** μέλη είναι προσβάσιμα **μόνο** από συναρτήσεις της **ίδιας** κλάσης. Δηλαδή, **δεν επιτρέπεται** να προσπελαστούν από συνάρτηση ή αντικείμενο έξω από την κλάση
- Για παράδειγμα, δεν επιτρέπεται να γράψουμε `s.code = 1`. Αντίθετα η `code = c;` μέσα στη `set()` επιτρέπεται, γιατί μία συνάρτηση μέλος μπορεί να προσπελάσει **οποιοδήποτε** μέλος της ίδιας κλάσης, ανεξάρτητα από το προσδιοριστικό του
- Όσον αφορά τα **δημόσια** μέλη, δεν υπάρχει περιορισμός στην προσπέλασή τους, το οποίο σημαίνει ότι είναι προσπελάσιμα από **οπουδήποτε** μέσα ή έξω από την κλάση
- Τα **προστατευμένα** μέλη συμπεριφέρονται όπως τα ιδιωτικά μέλη με την διαφορά ότι μπορεί να προσπελαστούν από συναρτήσεις παραγώγων κλάσεων, όπως θα δούμε στο Κ.20
- Τα μέλη μίας κλάσης είναι **εξ'ορισμού** ιδιωτικά. Δηλαδή, αν στο παράδειγμά μας έλειπε η λέξη `private` πριν από τη δήλωση του `code`, το `code` θα ήταν και πάλι ιδιωτικό

Έλεγχος Πρόσβασης (2)

- Βλέπουμε λοιπόν ότι μία κλάση όχι μόνο επιτρέπει την αναπαράσταση μίας λογικής οντότητας με την **ενθυλάκωση δεδομένων** σε έναν τύπο (data encapsulation), αλλά παρέχει και τη δυνατότητα στον προγραμματιστή να καθορίσει ποια δεδομένα να είναι **προσβάσιμα** από το εξωτερικό περιβάλλον και ποια να **αποκρύβονται** (data hiding)
- Όπως θα δούμε αργότερα, η απόκρυψη δεδομένων είναι πολύ σημαντική για την **ακεραιότητα** του αντικειμένου
- Με τον όρο εξωτερικό περιβάλλον, εννοούμε τους χρήστες της κλάσης, δηλαδή, προγραμματιστές που την χρησιμοποιούν
- Για παράδειγμα, στο παρακάτω πρόγραμμα, θεωρήστε ότι την κλάση `Test` την γράψατε εσείς και την δώσατε σε κάποιον άλλο προγραμματιστή, ο οποίος την χρησιμοποιεί στο πρόγραμμά του για να δημιουργήσει `Test` αντικείμενα. Βρείτε τα λάθη:

Παράδειγμα

```
#include <iostream>
class Test
{
private: /* Επειδή εξ'ορισμού τα μέλη μίας κλάσης είναι ιδιωτικά, μπορούμε να παραλείψουμε
τη λέξη private. */
            int a;
            void f1();
protected:
            int b;
            void f2();
public:
            int c;
            void f3();
};
void Test::f1()
{
            std::cout << a << ' ' << b << ' ' << c << ' ' << '\n';
}
void Test::f2()
{
            a = 100;
}
void Test::f3()
{
            f1();
}
int main()
{
            Test t;
            t.a = 1;
            t.b = 2;
            t.c = 3;
            t.f1();
            t.f2();
            t.f3();
            return 0;
}
```

Παράδειγμα

- Αφού τα μέλη `a` και `f1()` έχουν δηλωθεί ως `private` δεν επιτρέπεται η πρόσβαση σε αυτά από το εξωτερικό περιβάλλον. Επομένως, οι εντολές `t.a = 1;` και `t.f1();` δεν είναι αποδεκτές. Το ίδιο ισχύει και για τα `protected` μέλη. Άρα, οι εντολές `t.b = 2;` και `t.f2();` δεν είναι αποδεκτές. Αφού η πρόσβαση στα `public` μέλη επιτρέπεται, οι εντολές `t.c = 2;` και `t.f3();` είναι αποδεκτές. Τέλος, αφού οι συναρτήσεις μίας κλάσης έχουν πρόσβαση στα μέλη της, οι `f1()`, `f2()` και `f3()` μεταγλωττίζονται κανονικά
- Να το πούμε και διαφορετικά, η `f1()` μπορεί να χρησιμοποιηθεί άμεσα από τον προγραμματιστή που έγραψε την κλάση, αλλά όχι από τον προγραμματιστή που χρησιμοποιεί την κλάση
- Θυμόμαστε λοιπόν ότι η πρόσβαση στα ιδιωτικά και προστατευμένα μέλη μίας κλάσης επιτρέπεται από συναρτήσεις της κλάσης και, όπως θα δούμε στη συνέχεια, από φιλικές συναρτήσεις

Παρατηρήσεις

- Γενικά, όταν σχεδιάζουμε μία κλάση, τα μέλη και τις συναρτήσεις που δεν θέλουμε να είναι άμεσα προσβάσιμα τα δηλώνουμε **ιδιωτικά** (π.χ. συναρτήσεις που σχετίζονται με την υλοποίηση της κλάσης), ενώ τα μέλη και τις συναρτήσεις που θέλουμε να αποτελούν τη διεπαφή (interface) με το εξωτερικό περιβάλλον τα δηλώνουμε **δημόσια**
- Όπως θα δούμε στο Κ.20, η απόφαση για να κάνουμε ένα μέλος **προστατευμένο** βασίζεται στο αν θέλουμε να επιτρέψουμε την απευθείας πρόσβαση σε αυτό από μία παράγωγη κλάση. Αν ναι, το κάνουμε προστατευμένο, αν όχι, το κάνουμε ιδιωτικό
- Ουσιαστικά, ο χρήστης μίας κλάσης επικοινωνεί με την κλάση μέσω των **δημοσίων** μελών της, τα οποία αποτελούν την διεπαφή της κλάσης
- Έχουμε ήδη συναντήσει παραδείγματα στο Κ.10, όπου σαν χρήστες της κλάσης `string` χρησιμοποιήσαμε δημόσιες συναρτήσεις (π.χ. `size()`) για να την διαχειριστούμε. Το ίδιο κάναμε και σε παραδείγματα με την κλάση `vector`

const Συναρτήσεις Κλάσης

- Αν δεν θέλουμε μία συνάρτηση να μπορεί να αλλάξει τις τιμές του αντικειμένου που την καλεί προσθέτουμε τη λέξη const στο τέλος της δήλωσής της
- Η λέξη const πρέπει να προστεθεί και στον ορισμό της. Για παράδειγμα, η δήλωση της show() γίνεται:

```
void show() const;
```

και ο ορισμός της:

```
void Student::show() const
{
        grd = 2; // Λάθος.
}
```

Αν η show() επιχειρήσει να αλλάξει την τιμή κάποιου πεδίου της κλάσης, όπως το grd, ο μεταγλωττιστής θα εμφανίσει μήνυμα λάθους για μη επιτρεπτή ενέργεια

- Για καλύτερο έλεγχο της λειτουργίας της κλάσης, τις συναρτήσεις που δεν αλλάζουν τις τιμές των μελών της να τις δηλώνετε const

Φιλικές Συναρτήσεις

- Όπως μάθαμε, μία συνάρτηση μέλος μπορεί να έχει πρόσβαση στα ιδιωτικά μέλη μίας κλάσης. Υπάρχει τρόπος μία συνάρτηση που δεν είναι μέλος να έχει και αυτή πρόσβαση στα ιδιωτικά μέλη;
- Ναι, αρκεί να έχει δηλωθεί σαν **φιλική** (friend) της κλάσης. Συγκεκριμένα, μία φιλική συνάρτηση είναι μία συνάρτηση **μη-μέλος** η οποία μπορεί να έχει πρόσβαση στα μέλη μίας κλάσης ανεξάρτητα από το τμήμα στο οποίο δηλώνονται
- Δηλαδή, μία φιλική συνάρτηση έχει τα **ίδια δικαιώματα πρόσβασης** με μία συνάρτηση μέλος της κλάσης
- Για να γίνει μία συνάρτηση φιλική πρέπει να δηλωθεί μέσα στην κλάση με το πρόθεμα `friend`. Η θέση δεν παίζει ρόλο, μπορεί να δηλωθεί στο ιδιωτικό, προστατευμένο ή δημόσιο τμήμα

Παράδειγμα

```
#include <iostream>

class T
{
private:
        int a;
public:
        int show() {std::cout << a << '\n';}
        friend void f(T& t);
};

void f(T& t) // Η λέξη friend δεν προστίθεται στον ορισμό.
{
        t.a = 10; // Είναι λάθος να γράψουμε a = 10.
}

int main()
{
        T t;

        f(t);
        t.show();
        return 0;
}
```

Παράδειγμα

- Επειδή, όπως είπαμε, μία φιλική συνάρτηση δεν αποτελεί μέλος της κλάσης δεν προσθέτουμε το `T::` στον ορισμό της `f()`
- Για να γίνει ακόμα πιο ξεκάθαρο, αν είχαμε δηλώσει το αντικείμενο `t1` και γράφαμε `t1.f(t)`, ο μεταγλωττιστής θα εμφάνιζε μήνυμα λάθους παρόμοιο με «f is not a member of T»
- Για τον ίδιο λόγο, επειδή η `f()` δεν είναι μέλος της `T` δεν μπορούμε να προσπελάσουμε τα μέλη της γράφοντας μόνο τα ονόματά τους
- Σημειώστε ότι η λέξη `friend` δεν προστίθεται στον ορισμό της, εκτός αν δηλωθεί μέσα στην κλάση. Όπως είπαμε, αν και η `f()` δεν είναι συνάρτηση μέλος έχει τα ίδια δικαιώματα πρόσβασης με μία συνάρτηση μέλος
- Έτσι, επιτρέπεται η πρόσβαση στο ιδιωτικό πεδίο `a` και το πρόγραμμα εμφανίζει `10`

Κανόνες Πρόσβασης

- Τώρα που μιλήσαμε για τις φιλικές συναρτήσεις, ας συνοψίσουμε τους κανόνες που ισχύουν για την πρόσβαση στα μέλη μίας κλάσης:

α) Τα ιδιωτικά μέλη είναι προσβάσιμα μόνο από συναρτήσεις-μέλη της κλάσης και φιλικές συναρτήσεις της κλάσης

β) Τα προστατευμένα μέλη είναι προσβάσιμα μόνο από συναρτήσεις-μέλη της κλάσης και φιλικές της συναρτήσεις, καθώς και από συναρτήσεις-μέλη και τις φιλικές συναρτήσεις των κλάσεων που παράγονται από αυτή

γ) Τα δημόσια μέλη είναι προσβάσιμα από οποιαδήποτε συνάρτηση μέσα ή έξω από την κλάση

Κατασκευαστής

- Ο **κατασκευαστής** ή **συνάρτηση δόμησης** (constructor) είναι μία **ειδική** συνάρτηση της κλάσης που καλείται **κάθε φορά** που **δημιουργείται** ένα αντικείμενό της
- Η δήλωση του κατασκευαστή πρέπει να ακολουθεί κάποιους **κανόνες**
- Συγκεκριμένα, πρέπει να έχει το ίδιο όνομα με την κλάση, δεν έχει τύπο επιστροφής, δεν επιτρέπεται να δηλωθεί σαν `const`, μπορεί να δεχτεί παραμέτρους και μπορεί να υπερφορτωθεί, δηλαδή, μία κλάση μπορεί να έχει πολλούς κατασκευαστές
- Όπως και με τις απλές υπερφορτωμένες συναρτήσεις, οι εκδόσεις των κατασκευαστών πρέπει να διαφέρουν στον αριθμό ή τον τύπο των παραμέτρων τους
- Συνήθως, ένας κατασκευαστής χρησιμοποιείται για την απόδοση αρχικών τιμών στα πεδία του αντικείμενου ή για τη δέσμευση μνήμης

Αποδομητής

- Ο **αποδομητής** (destructor) είναι μία **ειδική** συνάρτηση της κλάσης που καλείται αυτόματα όταν **καταστρέφεται** ένα αντικείμενό της (π.χ. όταν βγει εκτός εμβέλειας)
- Πρέπει να έχει το ίδιο όνομα με την κλάση με τη διαφορά ότι προστίθεται ο χαρακτήρας ~ πριν από αυτό, δεν έχει τύπο επιστροφής, δεν δέχεται παραμέτρους και επομένως δεν μπορεί να υπερφορτωθεί. Δηλαδή, μία κλάση μπορεί να έχει έναν μόνο αποδομητή
- Ο λόγος για να ορίσουμε έναν αποδομητή είναι όταν θέλουμε να κάνουμε κάποιες ενέργειες όταν ένα αντικείμενο καταστρέφεται
- Αν δεν χρειάζεται να κάνουμε κάτι, τότε ο εξ'ορισμού αποδομητής, όπως θα δούμε αργότερα, είναι αρκετός. Για παράδειγμα, η πιο συνηθισμένη χρήση του αποδομητή είναι για την απελευθέρωση μνήμης που μπορεί να έχει δεσμεύσει ένα αντικείμενο με δυναμικό τρόπο

Παράδειγμα

- Ποια είναι η έξοδος του παρακάτω προγράμματος;

```
#include <iostream>

class Test
{
public:
        int a, b;

        Test();
        Test(int i, int j);
        ~Test();
        void check();
};
Test::Test()
{
        std::cout << "First\n";
}
Test::Test(int i, int j)
{
        std::cout << "Second\n";
        a = i;
        b = j;
}
Test::~Test()
{
        std::cout << "Out\n";
}

void Test::check()
{
        Test t;
}

int main()
{
        Test t1, t2(5, 10);

        std::cout << t1.a << ' ' <<
t1.b << '\n';
        std::cout << t2.a << ' ' <<
t2.b << '\n';
        t1.check();
        return 0;
}
```

Παράδειγμα

- Όπως είπαμε, οι δηλώσεις του κατασκευαστή και του αποδομητή δεν έχουν τύπο επιστροφής
- Επειδή υπάρχουν δύο υπερφορτωμένοι κατασκευαστές, ο μεταγλωττιστής επιλέγει τον κατάλληλο ανάλογα με τον τρόπο που δημιουργείται το κάθε αντικείμενο. Όπως θα δούμε στη συνέχεια, ένας κατασκευαστής που δεν δέχεται παραμέτρους ονομάζεται εξ'ορισμού κατασκευαστής (default constructor)
- Όταν δημιουργείται το αντικείμενο `t1` καλείται ο πρώτος κατασκευαστής και το πρόγραμμα εμφανίζει `First`. Οι μεταβλητές `a` και `b` του `t1` αρχικοποιούνται με τυχαίες τιμές
- Γενικά, μην ξεχνάτε να αρχικοποιείτε τα πεδία μίας κλάσης, γιατί, αν τα χρησιμοποιήσετε, απροσδόκητα προβλήματα μπορεί να προκύψουν

Παράδειγμα

- Οι κατασκευαστές υπακούουν στους ίδιους κανόνες υπερφόρτωσης με τις συνηθισμένες συναρτήσεις
- Ο μεταγλωττιστής ελέγχει τους τύπους των ορισμάτων για να επιλέξει τον κατάλληλο. Επομένως, όταν δημιουργείται το `t2`, καλείται ο δεύτερος κατασκευαστής, μεταβιβάζονται οι τιμές `5` και `10` και το πρόγραμμα εμφανίζει `Second`
- Στη συνέχεια, το πρόγραμμα εμφανίζει τις τυχαίες τιμές των `a` και `b` του αντικειμένου `t1` και τις τιμές `5` και `10` για τις αντίστοιχες του `t2`
- Όταν καλείται η `check()` δημιουργείται το αντικείμενο `t` και καλείται ο πρώτος κατασκευαστής
- Αμέσως μετά η `check()` τερματίζεται και το `t` παύει να υπάρχει. Επομένως, καλείται αυτόματα ο αποδομητής του `t`, ο οποίος εμφανίζει το μήνυμα `Out`

Παράδειγμα

- Όταν τερματίζει η εκτέλεση του προγράμματος, καλούνται οι αποδομητές των `t1` και `t2` και το πρόγραμμα εμφανίζει άλλες δύο φορές το μήνυμα `Out`
- Ας δούμε με ποια σειρά καλούνται οι αποδομητές. Επειδή η μνήμη για τις αυτόματες μεταβλητές `t1` και `t2` δεσμεύεται στη στοίβα, το αντικείμενο που δημιουργήθηκε τελευταίο είναι το πρώτο που θα καταστραφεί. Άρα, πρώτα θα κληθεί ο αποδομητής του `t2` και μετά του `t1`
- Αν έχουμε έναν πίνακα αντικειμένων, για παράδειγμα, `Test t[10];` η σειρά καταστροφής θα είναι από το τελευταίο προς το πρώτο, δηλαδή, `t[9]...t[0]`.

Εξ'ορισμού Κατασκευαστής και Αποδομητής

- Η ύπαρξη κατασκευαστή είναι απαραίτητη για τη δημιουργία των αντικειμένων
- Τότε όμως, θα μπορούσε να ρωτήσει κάποιος, πώς δημιουργήθηκαν `Student` αντικείμενα στο πρώτο μας παράδειγμα, αφού δεν ορίσαμε κάποιον κατασκευαστή;
- Η απάντηση είναι ότι αν μία κλάση δεν περιέχει κατασκευαστές, τότε ο μεταγλωττιστής παρέχει **αυτόματα** τον **εξ'ορισμού κατασκευαστή** (default constructor), ο οποίος δεν έχει σώμα. Αν δεν υπήρχε αυτός ο κατασκευαστής, δεν θα μπορούσαμε να δημιουργήσουμε `Student` αντικείμενα. Κάθε φορά που δηλώνεται ένα `Student` αντικείμενο, ο μεταγλωττιστής τον καλεί για τη δημιουργία του αντικειμένου
- Ο εξ'ορισμού κατασκευαστής καλεί τους εξ'ορισμού κατασκευαστές των αντίστοιχων κλάσεων για μέλη που είναι αντικείμενα
- Παρόμοια, αν δεν ορίζεται ο αποδομητής, ο μεταγλωττιστής παρέχει τον **εξ'ορισμού αποδομητή** (default destructor) που επίσης δεν έχει σώμα. Για παράδειγμα, ο εξ'ορισμού κατασκευαστής και αποδομητής για την κλάση Student είναι:

  `Student::Student() {}` και `Student::~Student() {}`

  Όπως βλέπετε, και οι δύο δεν δέχονται ορίσματα και δεν κάνουν τίποτα

Παρατηρήσεις

- Ο μεταγλωττιστής παρέχει αυτόματα τον εξ'ορισμού κατασκευαστή **μόνο αν** δεν έχει δηλωθεί άλλος κατασκευαστής
- Για παράδειγμα, αν δηλώσετε τον κατασκευαστή:

  `Student(const string& n, int c, float g);` αποτελεί δική σας ευθύνη να δηλώσετε την δική σας έκδοση του εξ'ορισμού κατασκευαστή, αν βέβαια το επιθυμείτε. Αν δεν το κάνετε, τότε μία δήλωση όπως:

```
Student s;
```

είναι **μη αποδεκτή** και ο μεταγλωττιστής θα εμφανίσει ένα μήνυμα λάθους παρόμοιο με «No default constructor available». Αν βέβαια γράψετε:

```
Student s("P.Lew", 100, 5.5);
```

δεν υπάρχει πρόβλημα, αφού θα κληθεί ο ορισμένος κατασκευαστής

Προκαθορισμένες Συναρτήσεις

- Οι συναρτήσεις που παρέχει εξ'ορισμού ο μεταγλωττιστής για μία κλάση `T` είναι:

  α) Ο προκαθορισμένος κατασκευαστής: `T()`

  β) Μία προκαθορισμένη συνάρτηση αντιγραφής τιμής (copy assignment operator) που δηλώνεται ως: `T& operator=(const T&)`. Για παράδειγμα, όπως θα δούμε στο Κ.18, όταν γράφουμε `t1 = t2;` καλείται η συνάρτηση `operator=`, η οποία αντιγράφει τις τιμές των μη στατικών μελών του `t2` στο `t1`. Αν το μέλος είναι πίνακας γίνεται αντιγραφή των στοιχείων του. Η συνάρτηση επιστρέφει αναφορά στον αριστερό τελεστέο

  γ) Ο προκαθορισμένος αποδομητής: `~T()`

- Αυτές είναι οι συναρτήσεις που εξ'ορισμού θα δημιουργήσει και θα καλέσει ο μεταγλωττιστής, εκτός αν ο προγραμματιστής έχει ορίσει τις δικές του. Στο Κ.19 θα δείτε και άλλες προκαθορισμένες συναρτήσεις
- Σημειώστε ότι υπάρχουν κάποιες ειδικές περιπτώσεις, όπου ο μεταγλωττιστής δεν δημιουργεί τις προκαθορισμένες συναρτήσεις. Για παράδειγμα, αν η κλάση περιέχει αναφορά ή `const` μέλος, ο μεταγλωττιστής δεν δημιουργεί τη συνάρτηση αντιγραφής τιμής

Αποδομητής και Αποδέσμευση Μνήμης

- Ας δούμε ένα παράδειγμα που επιδεικνύει τη συνηθέστερη χρήση ενός αποδομητή, η οποία είναι η αποδέσμευση μνήμης που έχει δεσμευτεί από πεδία-δείκτες του αντικειμένου

```
#include <iostream>

class T
{
private:
        int *data;
public:
        T(int num);
        ~T();
};
T::T(int num)
{
        data = new int[num];
}
T::~T()
{
        delete[] data;
}
int main()
{
        T t(10);
        return 0;
}
```

Ο Δείκτης this

- Κάθε συνάρτηση μίας κλάσης, εκτός από τις στατικές συναρτήσεις όπως θα δούμε στη συνέχεια, έχει πρόσβαση σε έναν κρυμμένο δείκτη που ονομάζεται this
- Όταν καλείται μία μη-στατική συνάρτηση μέλος, ο this δείχνει στο αντικείμενο που κάλεσε τη συνάρτηση. Για παράδειγμα, θεωρήστε ότι η κλάση T περιέχει τις συναρτήσεις f() και g(), ένα ακέραιο μέλος με το όνομα mem και το t είναι ένα αντικείμενο αυτής της κλάσης. Αν γράψουμε:

```
t.f();
```

ο μεταγλωττιστής μεταβιβάζει την διεύθυνση του t αντικειμένου στην f() και ο this δείκτης στην f() αρχικοποιείται με αυτή την διεύθυνση

- Μέσα στη συνάρτηση κάθε απευθείας πρόσβαση σε ένα μέλος της κλάσης θεωρείται ότι είναι μία έμμεση αναφορά στο μέλος μέσω του δείκτη this. Για παράδειγμα:

```
T::f()
{
    mem = 10; // Είναι σαν να έχουμε γράψει this->mem = 10;
    g(); // Είναι σαν να έχουμε γράψει this->g();
}
```

Όταν προσπελαύνεται η mem, ο μεταγλωττιστής προσπελαύνει την mem του αντικείμενου στο οποίο δείχνει ο this. Δηλαδή, ο μεταγλωττιστής, στην πραγματικότητα, μεταφράζει την πρόσβαση σε this->mem

Πίνακας Αντικειμένων

- Ένας πίνακας αντικειμένων είναι ένας πίνακας, όπου το κάθε του στοιχείο είναι αντικείμενο. Για παράδειγμα:

```
#include <iostream>

class Test
{
public:
        int a, b;
        Test() {a = b = 1;}
        Test(int i, int j) {a = i; b = j;}
};
int main()
{
        int i;
        Test t[10] = {Test(1, 2), Test(), Test(5, 6)};

        for(i = 0; i < 10; i++)
                std::cout << t[i].a << ' ' << t[i].b << '\n';
        return 0;
}
```

Για τα t[0] και t[2] καλείται ο δεύτερος κατασκευαστής, ενώ για το t[1] ο εξ'ορισμού κατασκευαστής. Επειδή η λίστα αρχικοποίησης είναι μικρότερη από το πλήθος των στοιχείων, για τα υπόλοιπα επτά αντικείμενα καλείται ο εξ'ορισμού κατασκευαστής. Αν αυτός έλειπε, ο μεταγλωττιστής θα εμφάνιζε μήνυμα λάθους. Το πρόγραμμα εμφανίζει τις τιμές των πεδίων του κάθε αντικειμένου

Στατικά Μέλη Κλάσης

- Όταν σε μία κλάση θέλουμε να δηλώσουμε μία μεταβλητή που να τη **μοιράζονται όλα** τα αντικείμενα της κλάσης τη δηλώνουμε `static`. Μία `static` μεταβλητή μπορούμε να τη δηλώσουμε σε όποιο τμήμα πρόσβασης επιθυμούμε και είναι προσβάσιμη από **οποιοδήποτε** αντικείμενο. Ανήκει στην κλάση ως σύνολο, και όχι σε ένα συγκεκριμένο αντικείμενο. Για παράδειγμα:

```
#include <iostream>

class T
{
private:
        int v;
        double d;
public:
        static inline int num = 5; // Δήλωση της στατικής μεταβλητής (C++17).
};
int main()
{
        T t[10];

        t[2].num = 20; // Εναλλακτικά, T::num = 20;
        std::cout << t[5].num << '\n';
        return 0;
}
```

Ο μεταγλωττιστής δεσμεύει μνήμη για ένα μόνο αντίγραφο της `static` μεταβλητής `num`, ανεξάρτητα από τον αριθμό των αντικειμένων που θα δημιουργηθούν. Επειδή δεσμεύεται μνήμη για μία μόνο `num`, η `num` είναι κοινή για όλα τα αντικείμενα και το πρόγραμμα εμφανίζει την τιμή 20

Στατικές Συναρτήσεις Κλάσης

- Μία συνάρτηση που είναι μέλος μίας κλάσης μπορεί να δηλωθεί στατική χρησιμοποιώντας τη λέξη static. Μία στατική συνάρτηση μέλος δεν συνδέεται με κάποιο αντικείμενο της κλάσης. Η λέξη static πρέπει να υπάρχει στη δήλωση της συνάρτησης αλλά όχι στον ορισμό της, αν αυτή ορίζεται έξω από την κλάση. Για παράδειγμα:

```
#include <iostream>

class T
{
private:
        int code;
public:
        T(int c);
        static inline int cnt = 0;
        static int get_objs();
};
T::T(int c)
{
        code = c;
        cnt++;
}
int T::get_objs() // Δεν προσθέτουμε τη λέξη static.
{
        return cnt;
}
int main()
{
        T t1(5), t2(6);
   std::cout << t1.get_objs() << ' ' << T::get_objs() << '\n'; /* Αν η get_objs() είχε δηλωθε
στο private τμήμα, ο μεταγλωττιστής θα εμφάνιζε μήνυμα λάθους για μη επιτρεπτή πρόσβαση. */
        return 0;
}
```

Στατικές Συναρτήσεις Κλάσης

- Όταν δημιουργείται ένα αντικείμενο η στατική μεταβλητή `cnt` αυξάνεται. Η στατική συνάρτηση `get_objs()` επιστρέφει αυτή την τιμή. Άρα, αφού δημιουργούνται δύο αντικείμενα το πρόγραμμα εμφανίζει `2`
- Όπως φαίνεται, μία στατική συνάρτηση μέλος δεν χρειάζεται να κληθεί από κάποιο αντικείμενο
- Όπως και μία στατική μεταβλητή μέλος, μία στατική συνάρτηση μέλος μπορεί να κληθεί ακόμα και αν δεν έχει δημιουργηθεί κάποιο αντικείμενο της κλάσης (π.χ. `T::get_objs()`)
- Βέβαια, μπορείτε να χρησιμοποιήσετε ένα αντικείμενο για να την καλέσετε (π.χ. `t1`)

Άσκηση

- Προσθέστε στην κλάση T όποια συνάρτηση θεωρείτε σκόπιμο ώστε να λειτουργεί το παρακάτω πρόγραμμα

```
class T
{
private:
        char *s;
        ...
};
int main()
{
        char str[100];

        cout << "Enter text: ";
        cin.getline(str, sizeof(str)); /* Θεωρήστε ότι ο χρήστης θα εισάγει αλφαριθμητικό με λιγότερους από 100 χαρακτήρες. */
        T t1(str); // Το αλφαριθμητικό να αντιγράφεται στο πεδίο s.
        T t2;
        t1.show(); /* Η show() να εμφανίζει το αλφαριθμητικό που εισήγαγε ο χρήστης με αντίστροφη σειρά. Για παράδειγμα, αν ο χρήστης εισάγει abcd το πρόγραμμα να εμφανίζει dcba. Επίσης, υπάρχει ο περιορισμός η μόνη μεταβλητή που επιτρέπεται να δηλώσετε στη show() να είναι δείκτης. */
        return 0;
}
```

Άσκηση

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

class T
{
private:
        char *s;
public:
        T(char str[]);
        T();
        ~T();
        void show() const;
};

T::T()
{
        s = nullptr; /* Αρχικοποιούμε τον δείκτη, ώστε να μην υπάρχει πρόβλημα με την
αποδέσμευση της μνήμης. Για παράδειγμα, για το t2 δεν έχει δεσμευτεί μνήμη. */
}

T::T(char str[])
{
        s = new char[strlen(str)+1];
        strcpy(s, str);
}
T::~T()
{
        delete[] s;
}
void T::show() const
{
        for(char *p = s+strlen(s)-1; p >= s; p--)
                cout << *p;
}
```

Άσκηση

- Σχόλια: Ας δούμε με ποια λογική προστέθηκαν οι παραπάνω συναρτήσεις. Αφού στη `main()` βλέπουμε ότι τα αντικείμενα δημιουργούνται με δύο τρόπους, επιλέγουμε να ορίσουμε δύο κατασκευαστές. Αφού το πεδίο `s` είναι δείκτης, ο κατασκευαστής πρέπει να δεσμεύσει μνήμη για την αντιγραφή του αλφαριθμητικού. Άρα, πρέπει να προστεθεί αποδομητής για την αποδέσμευση της μνήμης.

C++: Από τη Θεωρία στην Εφαρμογή

Κεφάλαιο 18ο

Υπερφόρτωση Τελεστών

Υπερφόρτωση Τελεστών

- Όπως μία συνάρτηση μπορεί να υπερφορτωθεί, ένας τελεστής μπορεί επίσης να υπερφορτωθεί (operator overloading) και να εκτελεί διαφορετικές λειτουργίες
- Για να υπερφορτώσουμε έναν τελεστή δηλώνουμε μία συνάρτηση υπερφόρτωσης χρησιμοποιώντας τη λέξη `operator`, ως εξής:

```
τύπος_επιστροφής operatorop(παράμετροι)
```

- Ο `op` μπορεί να είναι οποιοσδήποτε από τους τελεστές της C++ (π.χ. +, -, ..) με κάποιες εξαιρέσεις
- Όπως κάθε συνηθισμένη συνάρτηση, έχει σώμα, τύπο επιστροφής και μπορεί να δέχεται παραμέτρους
- Ο αριθμός των παραμέτρων είναι ίδιος με τον αριθμό των τελεστέων που έχει τελεστής. Δηλαδή, η συνάρτηση υπερφόρτωσης ενός δυαδικού τελεστή δέχεται δύο παραμέτρους, ενώ ενός μοναδιαίου τελεστή δέχεται μία

Παράδειγμα

- Για παράδειγμα, στο παρακάτω πρόγραμμα ορίζεται η κλάση Rect, η οποία περιέχει μία συνάρτηση που υπερφορτώνει τον τελεστή +:

```
#include <iostream>

class Rect
{
private:
        float length;
        float height;
public:
        Rect(float l = 0, float h = 0); /*
Εξ'ορισμού κατασκευαστής με προκαθορισμένες
αρχικές τιμές. */
        Rect operator+(const Rect& r) const;
        float area() const;
        void show() const;
};

Rect::Rect(float l, float h)
{
        length = l;
        height = h;
}

Rect Rect::operator+(const Rect& r) const
{
        Rect tmp;

        tmp.length = length + r.length;
        tmp.height = height + r.height;
        return tmp;
}

float Rect::area() const
{
        return length * height;
}

void Rect::show() const
{
        std::cout << "L:" << length <<
" H:" << height << '\n';
}
int main()
{
    Rect r1(10, 20), r2(30, 40), r3;

        r3 = r1+r2; /* Ισοδύναμο με
r3 = r1.operator+(r2); */
        r3.show();
        return 0;
}
```

Παράδειγμα

- Όταν ο μεταγλωττιστής συναντήσει την εντολή `r1+r2` εξετάζει τους τύπους των τελεστέων και αντιλαμβάνεται ότι πρέπει να καλέσει την αντίστοιχη `operator+` συνάρτηση
- Συγκεκριμένα, η εντολή `r1+r2` ερμηνεύεται σαν `r1.operator+(r2)`, άρα καλείται η συνάρτηση `operator+` του αντικειμένου `r1` με όρισμα το `r2`
- Γενικά, όταν η συνάρτηση υπερφόρτωσης ενός δυαδικού τελεστή δηλώνεται στην κλάση, ο αριστερός τελεστέος (π.χ. `r1`) είναι το αντικείμενο που την καλεί, ενώ ο δεξιός τελεστέος (π.χ. `r2`) μεταβιβάζεται σαν όρισμα στη συνάρτηση
- Αν δεν είχε δηλωθεί η συνάρτηση `operator+`, ο μεταγλωττιστής θα εμφάνιζε μήνυμα λάθους για μη αποδεκτή ενέργεια, αφού δεν είναι δυνατή η πρόσθεση αντικειμένων
- Εναλλακτικά, θα μπορούσαμε να καλέσουμε ρητά τη συνάρτηση και να γράψουμε `r1.operator+(r2)`
- Απλά, η χρήση του τελεστή είναι μία πιο εύκολη, για να διαβάσουμε και να γράψουμε, συντομογραφία της ρητής κλήσης
- Η συνάρτηση επιστρέφει ένα νέο αντικείμενο με διαστάσεις ίσες με το άθροισμα των αντιστοίχων διαστάσεων των δύο αντικειμένων. Άρα, το πρόγραμμα εμφανίζει `40` και `60`

Παράδειγμα

- Μήπως παρατηρήσατε τίποτα παράξενο με τον αριθμό των παραμέτρων;
- Είπαμε, ότι ο αριθμός των παραμέτρων είναι ίδιος με τον αριθμό των τελεστέων που έχει τελεστής. Όμως, βλέπουμε ότι η `operator+` δέχεται μία παράμετρο, ενώ ο τελεστής `+` δέχεται δύο τελεστέους
- Τι συμβαίνει; Αν η συνάρτηση υπερφόρτωσης είναι μέλος, η διεύθυνση μνήμης του πρώτου τελεστέου (π.χ. `r1`), η οποία ουσιαστικά είναι η τιμή του δείκτη `this`, μεταβιβάζεται έμμεσα σαν ένα «κρυμμένο» όρισμα, όπως είδαμε στο Κ.17
- Αυτός είναι και ο λόγος που οι συναρτήσεις υπερφόρτωσης που είναι μέλη έχουν μία λιγότερη παράμετρο από τον αριθμό των τελεστέων
- Για παράδειγμα, αν η κλάση υπερφορτώνει τον τελεστή `!` που είναι μοναδιαίος γράφουμε `operator!()` χωρίς παραμέτρους
- Αν η συνάρτηση υπερφόρτωσης δεν είναι μέλος κλάσης, τότε δέχεται τον ίδιο αριθμό παραμέτρων με τους τελεστέους του

Αντιγραφή Αντικειμένων που Περιέχουν Δείκτες

- Ας δούμε τώρα την περίπτωση της αντιγραφής αντικειμένων που περιέχουν δείκτη(ες). Δώστε ιδιαίτερη **προσοχή**, ποιο είναι το **λάθος** στο παρακάτω πρόγραμμα;

```
#include <iostream>
#include <cstring>

class T
{
private:
        char *s;
public:
        T(const char str[]);
        ~T();
        void show() const {std::cout << s << '\n';}
};
T::T(const char str[])
{
        s = new char[strlen(str)+1];
        strcpy(s, str);
}
T::~T()
{
         delete[] s;
}
int main()
{
        T t1("Peter"), t2("Mike");
        t2 = t1;
        t2.show();
        return 0;
}
```

Παράδειγμα

- Η εντολή `t2 = t1;` προκαλεί πρόβλημα στη λειτουργία του προγράμματος. Ας δούμε γιατί:
- Με την αντιγραφή των μελών, ο δείκτης `t2.s` δείχνει στην ίδια μνήμη που δείχνει και ο δείκτης `t1.s`. Η μνήμη που είχε δεσμευτεί για τον `t2.s` δεν αποδεσμεύεται
- Όταν το πρόγραμμα τερματίζει καλείται ο αποδομητής του αντικειμένου `t2`, ο οποίος και απελευθερώνει την μνήμη στην οποία τώρα δείχνει ο `t2.s`. Όμως, την ίδια μνήμη επιδιώκει να απελευθερώσει και ο αποδομητής του `t1`, αφού οι δείκτες `t2.s` και `t1.s` δείχνουν στην ίδια μνήμη
- Όπως ξέρουμε, το αποτέλεσμα αυτής της ενέργειας είναι απροσδιόριστο (π.χ. το πρόγραμμα μπορεί να καταρρεύσει)
- Αν η κλάση περιέχει δείκτη(ες) σε μνήμη που έχει δεσμευτεί δυναμικά, το πιθανότερο είναι ότι θα πρέπει να ορίσετε τη δική σας συνάρτηση αντιγραφής, ώστε να αποφύγετε προβλήματα με τη διαχείριση της μνήμης

Αντιγραφή Αντικειμένων που Περιέχουν Δείκτες

- Σε μία τέτοια περίπτωση η δυνατότητα υπερφόρτωσης τελεστών αποδεικνύεται ιδιαίτερα χρήσιμη. Συγκεκριμένα, για να αποφύγουμε ανεπιθύμητες καταστάσεις μπορούμε να υπερφορτώσουμε τον τελεστή =, ώστε να καθορίσουμε τον τρόπο με τον οποίο θα γίνει η αντιγραφή. Για παράδειγμα:

```
class T
{
public:
        ...
        T& operator=(const T& t);
};

T& T::operator=(const T& t)
{
        if(this == &t) /* Ελέγχουμε για αυτοεκχώρηση, όπως με μία εντολή t1 = t1; */
                return *this;

        delete[] s;
        s = new char[strlen(t.s)+1];
        strcpy(s, t.s);
        return *this;
}
```

- Τώρα, με την εντολή `t2 = t1;` η οποία είναι ισοδύναμη με `t2.operator=(t1);` πρώτα αποδεσμεύεται η μνήμη που είχε δεσμευτεί για το `t2.s` και μετά δεσμεύεται νέα μνήμη για να αντιγραφεί σε αυτήν το αλφαριθμητικό στο οποίο δείχνει ο `t1.s`. Η συνάρτηση επιστρέφει το ίδιο το αντικείμενο, δηλαδή, το `t2`. Άρα, τώρα το πρόγραμμα λειτουργεί κανονικά και εμφανίζει **Peter**

Περιορισμοί Υπερφόρτωσης (1)

- Ένας τουλάχιστον τελεστέος του υπερφορτωμένου τελεστή πρέπει να είναι τύπος δηλωμένος από τον προγραμματιστή. Αυτός ο κανόνας εμποδίζει την υπερφόρτωση τελεστών για τους βασικούς τύπους δεδομένων. Για παράδειγμα, δεν μπορείτε να υπερφορτώσετε τον τελεστή + για να επιστρέψετε τη διαφορά δύο ακεραίων αντί για το άθροισμα
- Δεν επιτρέπεται να αλλάξετε τον τρόπο σύνταξης του τελεστή. Για παράδειγμα, δεν επιτρέπεται να χρησιμοποιήσετε τον τελεστή + με έναν τελεστέο
- Δεν μπορείτε να αλλάξετε την προτεραιότητα και συσχέτιση του τελεστή. Για παράδειγμα, σε μία παράσταση `a*b+c` όπου τα `a`, `b` και `c` είναι αντικείμενα και οι τελεστές `*` και `+` έχουν υπερφορτωθεί πρώτα εκτελείται ο πολ/σμός και μετά η πρόσθεση

Περιορισμοί Υπερφόρτωσης (2)

- Δεν επιτρέπεται να δημιουργήσετε νέους τελεστές και στη συνέχεια να τους υπερφορτώσετε. Για παράδειγμα, δεν επιτρέπεται να δηλώσετε τη συνάρτηση `operator^^` για να δημιουργήσετε ύψωση σε δύναμη, αφού δεν υπάρχει ο τελεστής `^^`
- Δεν επιτρέπεται η υπερφόρτωση των παρακάτω τελεστών:

  `.` `.*` `::` `?:` `sizeof` `typeid` `alignas` `noexcept`
- Οι περισσότεροι τελεστές μπορούν να υπερφορτωθούν είτε με τη χρήση συναρτήσεων μελών είτε με συναρτήσεις που δεν είναι μέλη κάποιας κλάσης. Ωστόσο, οι παρακάτω τελεστές μπορούν να υπερφορτωθούν μόνο με μη στατικές συναρτήσεις μέλη:

  `=` `()` `[]` `->`
- Χωρίς αυτό να αποτελεί περιορισμό, όταν υπερφορτώνεται ένας τελεστής είναι ωφέλιμο για την καλύτερη κατανόηση και τον έλεγχο του προγράμματος να επιτελεί μία λειτουργία παρόμοια με τον αρχικό του σκοπό. Για παράδειγμα, μία συνάρτηση υπερφόρτωσης του τελεστή `+` να επιτελεί την πρόσθεση και όχι κάποια άλλη πράξη

Υπερφόρτωση των Τελεστών ++ και --

- Στο παράδειγμα με την κλάση, ας υπερφορτώσουμε τον τελεστή ++, παρόμοια υπερφορτώνεται και ο --, ώστε να αυξάνει τις διαστάσεις του αντικειμένου που τον καλεί κατά ένα
- Όπως γνωρίζουμε, ο τελεστής ++ μπορεί να εφαρμοστεί με δύο τρόπους. Επομένως, θα υλοποιήσουμε δύο εκδόσεις της συνάρτησης, μία για την επιθεματική και μία για την προθεματική αύξηση
- Για παράδειγμα, με την εντολή `++r;` η συνάρτηση να επιστρέφει το αντικείμενο `r` μετά την αύξηση των διαστάσεών του, ενώ με την `r++;` να επιστρέφει το `r` πριν την αύξηση
- Υπάρχει όμως ένα πρόβλημα. Αφού και οι δύο συναρτήσεις έχουν το ίδιο όνομα `operator++` και δεν δέχονται παραμέτρους με ποιο τρόπο ο μεταγλωττιστής θα καταλάβει ποια να καλέσει;
- Για να λυθεί αυτό το πρόβλημα, στην επιθεματική έκδοση προστίθεται μία ακέραια παράμετρος. Είναι μόνο ένα ψευδο-όρισμα που χρησιμοποιεί ο μεταγλωττιστής για να ξεχωρίσει ποια έκδοση να καλέσει
- Επίσης, η προθεματική έκδοση συνήθως επιστρέφει μία αναφορά στο αντικείμενο, ενώ η επιθεματική δημιουργεί και επιστρέφει ένα αντίγραφο του αρχικού αντικειμένου, όχι αναφορά

Υπερφόρτωση των Τελεστών ++ και --

```
Rect& Rect::operator++() /* Προθεματική έκδοση (π.χ. ++r). Η δήλωση
προστίθεται στην κλάση. */
{
        length++;
        height++;
        return *this;
}
Rect Rect::operator++(int) /* Επιθεματική έκδοση (π.χ. r++). Η δήλωση
προστίθεται στην κλάση. */
{
        Rect tmp;

        tmp = *this;
        length++;
        height++;
        return tmp;
}
int main()
{
        Rect r1(10, 20), r2, r3;

        r2 = ++r1; // Ισοδύναμη με r2 = r1.operator++();
        r1.show(); // Το πρόγραμμα εμφανίζει 11 21
        r2.show(); // Το πρόγραμμα εμφανίζει 11 21

        r3 = r1++; // Ισοδύναμη με r3 = r1.operator++(0);
        r1.show(); // Το πρόγραμμα εμφανίζει 12 22
        r3.show(); // Το πρόγραμμα εμφανίζει 11 21
        return 0;
}
```

Παράδειγμα

- Να προσθέσετε στην κλάση `Rect` μία συνάρτηση υπερφόρτωσης του τελεστή `==`, η οποία να δέχεται σαν παράμετρο ένα `Rect` αντικείμενο (π.χ. `r`) και να επιστρέφει `1` αν και οι δύο διαστάσεις του αντικειμένου που την καλεί είναι ίδιες με του `r`, `2` αν κάποια είναι ίδια, αλλιώς `3`. Τροποποιήστε τη `main()`, ώστε να ελέγξετε τη λειτουργία της συνάρτησης

Παράδειγμα

```
class Rect
{
        ...
public:
        int operator==(const Rect& r) const;
        ...
};
int Rect::operator==(const Rect& r) const
{
        if(length == r.length && height == r.height)
                return 1;
        else if(length == r.length || height == r.height)
                return 2;
        else
                return 3;
}
int main()
{
        int a;
        Rect r1(10, 20), r2(10, 0);

        a = (r1 == r2); // Ισοδύναμο με a = (r1.operator==(r2));
        if(a == 1)
                std::cout << "Both the same\n";
        else if(a == 2)
                std::cout << "One is the same\n";
        else
                std::cout << "Unequal\n";
        return 0;
}
```

Παράδειγμα

- Σχόλια. Όπως μάθαμε, η έκφραση (`r1 == r2`) δεν εκτελεί την συνηθισμένη σύγκριση, αλλά προκαλεί την κλήση της συνάρτησης `operator==` του αντικειμένου `r1` με όρισμα το `r2`. Μετά, μπορούμε να ελέγξουμε την τιμή επιστροφής της όπως κάνουμε με κάθε συνάρτηση. Οι παρενθέσεις δεν είναι απαραίτητες, απλά τις προσθέτουμε για να φαίνεται ξεκάθαρα ότι πρώτα θα εκτελεστεί η συνάρτηση

Παράδειγμα

- Βρείτε τα λάθη στο παρακάτω πρόγραμμα:

```
#include <iostream>

class A
{
        int t;
public:
        A(int i) {t = i;}
        void operator+(int i) {t += i;}
};
class B
{
        int t;
public:
        B(int i) {t = i;}
};
int main()
{
        A a1(10), a2;
        B b1(a1.t);

        a2 = a1+5;
        a1+5;
        b1+5;
        5+a1;
        return 0;
}
```

Παράδειγμα

- Αφού στην κλάση A δηλώνεται ένας μη εξ'ορισμού κατασκευαστής, η δήλωση του `a2` είναι λάθος, γιατί δεν έχει δηλωθεί εξ'ορισμού κατασκευαστής
- Η δήλωση `b1(a1.t)` δεν είναι σωστή, γιατί δεν επιτρέπεται η πρόσβαση στο ιδιωτικό πεδίο `t`. Θυμηθείτε ότι η ετικέτα `private` δεν χρειάζεται, αφού τα πεδία μίας κλάσης είναι ιδιωτικά εξ'ορισμού
- Η εντολή `a2 = a1+5;` δεν είναι αποδεκτή, γιατί η συνάρτηση υπερφόρτωσης του τελεστή `+` δεν επιστρέφει κάτι
- Η εντολή `a1+5;` είναι αποδεκτή και προσθέτει το `5` στην μεταβλητή `t` του αντικειμένου `a1`
- Η εντολή `b1+5;` δεν είναι αποδεκτή, γιατί δεν ορίζεται συνάρτηση υπερφόρτωσης του τελεστή `+` στην κλάση `B`
- Τέλος, η εντολή `5+a1;` δεν είναι αποδεκτή, γιατί αριστερά του τελεστή πρέπει να βρίσκεται αντικείμενο της κλάσης `A`. Και για να το καταλάβετε καλύτερα, η `5+a1;` μεταφράζεται σε `5.operator(a1);` η οποία δεν είναι αποδεκτή έκφραση. Θα δούμε πώς να χειριστούμε αυτή την περίπτωση στην επόμενη ενότητα

Υπερφόρτωση με Συναρτήσεις μη Μέλη

- Όπως είπαμε, οι περισσότεροι τελεστές μπορούν να υπερφορτωθούν και με χρήση συναρτήσεων που δεν είναι μέλη κλάσης
- Μάλιστα, υπάρχουν περιπτώσεις που η χρήση τους μπορεί να είναι απαραίτητη. Για παράδειγμα, θεωρήστε την κλάση `A` της προηγούμενης άσκησης. Όπως είδαμε, η εντολή `a1+5;` είναι αποδεκτή, όχι όμως και η `5+a1;`
- Για να επιτύχουμε κάτι τέτοιο πρέπει να δηλώσουμε μία συνάρτηση που να μην είναι μέλος. Μία τέτοια συνάρτηση δεν μπορεί να κληθεί από κάποιο αντικείμενο. Συγκεκριμένα, οι τιμές των τελεστέων μεταβιβάζονται σαν ορίσματα στη συνάρτηση
- Η απαίτηση είναι ένας τουλάχιστον από τους τελεστέους να έχει τύπο κλάσης. Στο παράδειγμά μας, το πρωτότυπο της συνάρτησης είναι:

  ```
  void operator+(int i, A& a);
  ```

  Τώρα, η έκφραση `5+a1;` μεταφράζεται σε `operator+(5, a1);`
- Ας θυμόμαστε λοιπόν, όταν θέλουμε να υπερφορτώσουμε έναν τελεστή, όπου ο πρώτος τελεστέος να είναι κάποιος βασικός τύπος (π.χ. `int`), η συνάρτηση υπερφόρτωσης δεν μπορεί να είναι συνάρτηση μέλος της κλάσης. Αν θέλουμε η συνάρτηση να έχει πρόσβαση στα ιδιωτικά μέλη της κλάσης, την δηλώνουμε φιλική

Υπερφόρτωση με Συναρτήσεις μη Μέλη

```
#include <iostream>

class A
{
        int t;
public:
        A(int i) {t = i;}
        void operator+(int i) {t += i;}
        friend void operator+(int i, A& a);
        void show() const {std::cout << t << '\n';}
};
void operator+(int i, A& a)
{
        a.t += i;
}
int main()
{
        A a(10);

        a+5; // Μεταφράζεται σε a.operator+(5).
        a.show(); // Το πρόγραμμα εμφανίζει 15.
        5+a; // Μεταφράζεται σε operator+(5, a).
        a.show(); // Το πρόγραμμα εμφανίζει 20.
        return 0;
}
```

Υπερφόρτωση του << Τελεστή

- Τώρα που είδατε παραδείγματα υπερφόρτωσης τελεστών μπορείτε να καταλάβετε αυτό που διαβάσατε στο Κ.3, ότι οι τελεστές ολίσθησης << και >> έχουν υπερφορτωθεί για την εμφάνιση και εισαγωγή δεδομένων, αντίστοιχα
- Για παράδειγμα, το cout είναι αντικείμενο της κλάσης ostream, η οποία περιέχει υπερφορτωμένες συναρτήσεις operator<< για κάθε βασικό τύπο
- Ας δούμε πώς μπορούμε να χρησιμοποιήσουμε το cout με έναν δικό μας τύπο, όπως τον Rect. Ο συνηθισμένος τρόπος για να εμφανίσουμε τις τιμές των πεδίων μίας κλάσης είναι να ορίσουμε μία δημόσια συνάρτηση show() και να την καλέσουμε. Η ερώτηση είναι, θα μπορούσαμε, αντί της show(), να γράψουμε κάτι τέτοιο και να εμφανίζουμε τις τιμές τους;

```
Rect r(10, 20);
cout << r;
```

- Βεβαίως, αρκεί να υπερφορτώσουμε τον τελεστή <<. Πράγματι, είναι πολύ συνηθισμένη η υπερφόρτωση του τελεστή << και η χρήση του μαζί με το cout για την εμφάνιση των τιμών του αντικειμένου
- Δείτε πώς λειτουργεί. Αφού ο αριστερός τελεστέος είναι αντικείμενο της κλάσης ostream και ο δεξιός αντικείμενο της κλάσης Rect πρέπει να ορίσουμε μία συνάρτηση που να δέχεται σαν ορίσματα δύο τέτοιους τύπους. Να είναι η συνάρτηση φιλική στην κλάση Rect ή όχι; Αφού η συνάρτηση πρέπει να έχει πρόσβαση στα ιδιωτικά μέλη για να τα εμφανίσει, ναι πρέπει να είναι φιλική. Ας δούμε το πρόγραμμα:

Υπερφόρτωση του << Τελεστή

```
class Rect
{
        ...
public:
   friend ostream& operator<<(ostream& out, const Rect& r);
        ...
};
ostream& operator<<(ostream& out, const Rect& r)
{
        out << r.length << ' ' << r.height << '\n';
        return out;
}
int main()
{
        Rect r(10, 20);
        cout << r; // Μεταφράζεται σε operator<<(cout, r).
        return 0;
}
```

Άσκηση

- Να ορίσετε την κλάση Time με ιδιωτικά πεδία τα ώρες (π.χ. hrs), λεπτά (π.χ. mins) και δευτερόλεπτα (π.χ. secs). Να προσθέσετε κατάλληλες συναρτήσεις ώστε να λειτουργεί το παρακάτω πρόγραμμα.

```
int main()
{
	Time t1, t2(23, 59, 59); /* Υπάρχει ο περιορισμός η κλάση να έχει έναν κατασκευαστή. Τα πεδία του t1 να αρχικοποιούνται με 0. Επίσης, υπάρχει ο περιορισμός οι παράμετροι του κατασκευαστή να έχουν τα ίδια ονόματα με τα ιδιωτικά πεδία. */
	++t2; /* Αυξάνεται ο αριθμός των δευτερολέπτων κατά ένα. Αν η ώρα είναι 23:59:59, όπως στην περίπτωση του t2, η νέα ώρα να γίνει 0:0:0. */
	cout << (t1 == t2) << '\n'; /* Να συγκρίνονται οι δύο ώρες και αν είναι ίδιες, το πρόγραμμα να εμφανίζει 1 αλλιώς 0. Για παράδειγμα, με αυτές τις τιμές των t1 και t2 το πρόγραμμα εμφανίζει 1. */
	cout << t1; /* Να εμφανίζεται η ώρα στη μορφή h:m:s. Για παράδειγμα, 0:0:0. */
	return 0;
}
```

Άσκηση

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

class Time
{
private:
        int hrs;
        int mins;
        int secs;
public:
        Time(int hrs=0, int mins=0, int secs=0);
        bool operator==(const Time& t) const;
        void operator++();
        friend ostream& operator<<(ostream& out, const Time& t);
};
Time::Time(int hrs, int mins, int secs)
{
        this->hrs = hrs; /* Επειδή υπάρχει ο περιορισμός για ίδια ονόματα
χρησιμοποιούμε τον δείκτη this. */
        this->mins = mins;
        this->secs = secs;
}
bool Time::operator==(const Time& t) const
{
        if((hrs == t.hrs) && (mins == t.mins) && (secs == t.secs))
                return true;
        else
                return false;
}
```

Άσκηση

```
void Time::operator++()
{
        secs++;
        if(secs == 60)
        {
                secs = 0;
                mins++;
                if(mins == 60)
                {
                        mins = 0;
                        hrs++;
                        if(hrs == 24)
                                hrs = 0;
                }
        }
}
ostream& operator<<(ostream& out, const Time& t)
{
        out << t.hrs << ':' << t.mins << ':' << t.secs << '\n';
        return out;
}
```

C++: Από τη Θεωρία στην Εφαρμογή

Κεφάλαιο 19ο

Περισσότερα για Κλάσεις

Λίστα Αρχικοποίησης Μελών

- Όπως ξέρουμε, ένας τρόπος για να αρχικοποιήσουμε τα μέλη μίας κλάσης είναι μέσα στο σώμα του κατασκευαστή. Για παράδειγμα:

```
class T
{
private:
        int a;
public:
        double b;
        string s;
        T(int arg1, double arg2);
};

T::T(int arg1, double arg2)
{
        a = arg1;
        b = arg2;
        s = "Text";
}
```

- Εναλλακτικά, τα μέλη μίας κλάσης μπορούν να αρχικοποιηθούν με μία ειδική σύνταξη όταν καλείται ο κατασκευαστής της κλάσης. Για παράδειγμα:

```
T::T(int arg1, double arg2) : s("Text"), a(arg1), b(arg2) /* Λίστα
αρχικοποίησης μελών. */
{
}
```

Λίστα Αρχικοποίησης Μελών

- Αυτή η σύνταξη αρχικοποίησης μελών (member initializer list) μπορεί να χρησιμοποιηθεί μόνο με κατασκευαστές, και όχι με συναρτήσεις μέλη της κλάσης
- Όπως φαίνεται, προστίθεται ένα : και οι αρχικοποιήσεις των μελών χωρίζονται με κόμμα. Πρώτα γράφουμε το όνομα του μέλους και μετά την αρχική τιμή μέσα σε παρενθέσεις
- Έτσι, το `Text` αποθηκεύεται στο `s`, το `a` γίνεται ίσο με `arg1` και το `b` ίσο με `arg2`
- Αν και το λογικό είναι να νομίζετε ότι οι αρχικοποιήσεις γίνονται από αριστερά προς τα δεξιά, δηλαδή, πρώτα το `s` αρχικοπόιείται και μετά τα υπόλοιπα, η πραγματικότητα είναι ότι τα μέλη αρχικοποιούνται με την σειρά με την οποία εμφανίζονται στη δήλωση της κλάσης, δηλαδή, πρώτα το `a`, μετά το `b`, και τελευταίο το `s`

Κατασκευαστής Αντιγράφου (1)

- Ο κατασκευαστής αντιγράφου (copy constructor) είναι μία ειδική συνάρτηση που καλείται **αυτόματα** όταν δημιουργείται ένα νέο αντικείμενο και αρχικοποιείται με ένα υπάρχον. Για παράδειγμα:

```
#include <iostream>

class T
{
private:
        int a;
public:
        int b;
        T(int i, int j) {a=i; b=j;}
        void show() const {std::cout << a << ' ' << b << '\n';}
};
int main()
{
        T t1(1, 2);
        T t2(t1); // Καλείται ο εξ'ορισμού κατασκευαστής αντιγράφου.
        T t3 = t1; /* Καλείται ο εξ'ορισμού κατασκευαστής
αντιγράφου. */
        t2.show();
        t3.show();
        return 0;
}
```

Κατασκευαστής Αντιγράφου (2)

- Όπως ο μεταγλωττιστής παρέχει τον εξ'ορισμού κατασκευαστή αν δεν έχει οριστεί κατασκευαστής, τον εξ'ορισμού αποδομητή αν δεν έχει οριστεί αποδομητής και την εξ'ορισμού συνάρτηση αντιγραφής, παρέχει και τον εξ'ορισμού κατασκευαστή αντιγράφου αν δεν έχει οριστεί κάποιος
- Έτσι, για τη **δημιουργία** των `t2` και `t3` αντικειμένων καλείται ο **εξ'ορισμού** κατασκευαστής αντιγράφου
- Και τι κάνει αυτός ο κατασκευαστής; Ότι λέει το όνομά του, δηλαδή, **αντιγράφει** ένα-προς-ένα τα **μη στατικά** μέλη του **υπάρχοντος** αντικειμένου στο **νέο**. Για παράδειγμα, με την εντολή: `T t2(t1);` ο μεταγλωττιστής δημιουργεί το `t2` αντικείμενο και **καλεί** τον κατασκευαστή αντιγράφου για να **αντιγράψει** τα πεδία του `t1` στο `t2`
- Αυτή η αντιγραφή ονομάζεται και **ρηχή αντιγραφή** (shallow copy). Το πώς θα αντιγραφεί το κάθε μέλος εξαρτάται από τον τύπο του. Οι βασικοί τύποι αντιγράφονται απευθείας, αν είναι πίνακας γίνεται αντιγραφή των στοιχείων του και αν είναι αντικείμενο καλείται ο κατασκευαστής αντιγράφου της κλάσης του για να γίνει η αντιγραφή

Κατασκευαστής Αντιγράφου (3)

- Σημειώστε ότι σε μία συνηθισμένη εντολή εκχώρησης, όπως `t2 = t1`, δεν καλείται ο κατασκευαστής αντιγράφου, αλλά καλείται η εξ'ορισμού συνάρτηση αντιγραφής operator=, η οποία αντιγράφει τα πεδία του `t1` στο `t2`
- Για να είναι σαφές, ο κατασκευαστής αντιγράφου καλείται όταν το αντικείμενο **κατασκευάζεται** (π.χ. `T t2 = t1;`), ενώ όταν το αντικείμενο υπάρχει (π.χ. `t2 = t1;`) καλείται η συνάρτηση αντιγραφής
- Αν η κλάση περιέχει στατικά μέλη αυτά δεν επηρεάζονται αφού ανήκουν στην κλάση και όχι σε κάποιο αντικείμενο
- Το πρόγραμμα εμφανίζει δύο φορές `1 2`

Κατασκευαστής Αντιγράφου και Μέλη Δείκτες

Αν η κλάση περιέχει μέλη που είναι δείκτες, προβλήματα μπορεί να προκύψουν αν χρησιμοποιήσουμε τον εξ'ορισμού κατασκευαστή αντιγράφου (και την εξ'ορισμού συνάρτηση αντιγραφής, όπως είδαμε στο Κ.18). Για παράδειγμα, δείτε το παρακάτω πρόγραμμα και προσπαθήστε να βρείτε ποιο είναι το πρόβλημα

```
#include <iostream>
#include <cstring>
class T
{
public:
        char *s;
        T(const char str[]);
        ~T();
};
T::T(const char str[])
{
        s = new char[strlen(str)+1];
        strcpy(s, str);
}
T::~T()
{
         delete[] s;
}
int main()
{
        T t1("text");
        T t2 = t1; // Καλείται ο εξ'ορισμού κατασκευαστής αντιγράφου.
        t2.s[0] = 'a';
        std::cout << t1.s << '\n';
        return 0;
```

Κατασκευαστής Αντιγράφου και Μέλη Δείκτες

- Το πρόβλημα οφείλεται στην κλήση του εξ'ορισμού κατασκευαστή αντιγράφου όταν δημιουργείται το `t2`
- Συγκεκριμένα, το `t2.s` θα γίνει ίσο με το `t1.s`, δηλαδή, και οι δύο δείκτες θα δείχνουν στην ίδια μνήμη. Άρα, όταν αλλάζει το `t2.s[0]` αλλάζει και το `t1.s[0]`. Έτσι, το πρόγραμμα εμφανίζει `aext`
- Εκτός από αυτό, το ότι και οι δύο δείκτες δείχνουν στην ίδια μνήμη προκαλεί και άλλο λάθος. Ειδικότερα, όταν καταστραφεί το `t2` καλείται ο αποδομητής του και αποδεσμεύεται η μνήμη στην οποία δείχνει ο `t2.s`, η οποία, όπως είπαμε, είναι αυτή που δείχνει και ο `t1.s`. Άρα, όταν καταστραφεί το `t1` ο αποδομητής του επιχειρεί να απελευθερώσει την ίδια μνήμη που είναι λανθασμένη ενέργεια. Και πώς διορθώνονται αυτά τα προβλήματα;
- Η λύση είναι να ορίσουμε τον δικό μας κατασκευαστή αντιγράφου. Δηλαδή, προσθέτουμε στην κλάση την παρακάτω συνάρτηση:

```
class T
{
public:
        ...
        T(const T& t);
};
T::T(const T& t)
{
        s = new char[strlen(t.s)+1];
        strcpy(s, t.s);
}
```

Κατασκευαστής Αντιγράφου και Μέλη Δείκτες

- Τώρα, όταν δημιουργείται το `t2` καλείται αυτός ο κατασκευαστής αντιγράφου και το `t1` μεταβιβάζεται σαν όρισμα
- Δηλαδή, η αναφορά `t` αναφέρεται στο `t1` αντικείμενο. Στη συνέχεια, δεσμεύεται μνήμη για το `t2.s` και αντιγράφεται σε αυτήν το `t1.s`
- Αυτή η αντιγραφή ονομάζεται και **βαθιά αντιγραφή** (deep copy) με την έννοια ότι για κάθε πεδίο-δείκτη του αντικειμένου που δημιουργείται, πρώτα δεσμεύεται μνήμη και μετά γίνεται η αντιγραφή σε αυτήν
- Έτσι, αυτή τη φορά το πρόγραμμα εμφανίζει `text` και όχι `aext` όπως προηγουμένως
- Επίσης, αφού η μνήμη που δείχνει ο `t1.s` είναι διαφορετική από τη μνήμη που δείχνει ο `t2.s`, δεν θα δημιουργηθεί πρόβλημα όταν στο τέλος του προγράμματος κληθεί ο αποδομητής του κάθε αντικειμένου
- Αν η κλάση **περιέχει** δείκτη(ες), θα **πρέπει** να ορίσετε τον **δικό** σας **κατασκευαστή αντιγράφου** και να δεσμεύσετε μνήμη για να αντιγράψετε τα πεδία του πρωτότυπου αντικειμένου
- Για τον ίδιο λόγο, όπως είδαμε σε παρόμοιο παράδειγμα στο προηγούμενο κεφάλαιο, θα πρέπει να ορίσετε και τη **δική** σας **συνάρτηση αντιγραφής** με την υπερφόρτωση του τελεστή =

C++: Από τη Θεωρία στην Εφαρμογή

Κεφάλαιο 20ο
Κληρονομικότητα

Κληρονομικότητα

- Η κληρονομικότητα *(inheritance)* αποτελεί ένα από τα κυριότερα χαρακτηριστικά της C++ και γενικότερα του αντικειμενοστρεφούς προγραμματισμού
- Η κληρονομικότητα επιτρέπει την δημιουργία μίας νέας κλάσης από μία υπάρχουσα. Η νέα κλάση ονομάζεται **παράγωγη** *(derived)*, ενώ η υπάρχουσα ονομάζεται **βασική** *(base)*
- Η παράγωγη κλάση **κληρονομεί** τα χαρακτηριστικά της βασικής κλάσης
- Ο προγραμματιστής μπορεί να προσθέσει νέα χαρακτηριστικά στην παράγωγη κλάση, να χρησιμοποιήσει όπως είναι τα χαρακτηριστικά που κληρονομήθηκαν ή να τους αλλάξει την συμπεριφορά και να τα προσαρμόσει στις ανάγκες της παράγωγης κλάσης
- Η γενική μορφή δήλωσης μίας παράγωγης κλάσης είναι:

```
class όνομα_παράγωγης : τύπος_πρόσβασης όνομα_βασικής
```

- Ένα αντικείμενο μίας παράγωγης κλάσης **περιέχει** ένα αντικείμενο της βασικής κλάσης
- Η παράγωγη κλάση μπορεί να χρησιμοποιηθεί σαν βασική για τη δημιουργία μίας νέας παράγωγης κλάσης και αυτή το ίδιο με τη σειρά της, και έτσι να δημιουργηθεί μία ιεραρχική αλυσίδα κλάσεων. Για παράδειγμα, μία κλάση `C` μπορεί να παράγεται από την κλάση `B`, η οποία με τη σειρά της να παράγεται από την `A`, κ.ο.κ.

Κληρονομικότητα

- Ο `τύπος_πρόσβασης` καθορίζει την πρόσβαση στα μέλη της βασικής κλάσης
- Αν λείπει, ο εξ'ορισμού τύπος πρόσβασης είναι ο `private`
- Όταν ο τύπος πρόσβασης είναι `public`, οι συναρτήσεις και τα αντικείμενα της παράγωγης κλάσης, καθώς και οι φιλικές συναρτήσεις της παράγωγης έχουν πρόσβαση στα δημόσια μέλη της βασικής, αλλά όχι στα ιδιωτικά
- Ειδικότερα, η παράγωγη κλάση μπορεί να προσπελάσει τα ιδιωτικά μέλη της βασικής μόνο μέσω των δημόσιων και των προστατευμένων συναρτήσεων της βασικής κλάσης
- Επίσης, οι συναρτήσεις της παράγωγης κλάσης, καθώς και οι φιλικές συναρτήσεις της μπορούν να προσπελάσουν απευθείας τα προστατευμένα μέλη της βασικής, ενώ τα αντικείμενά της όχι
- Η βασική κλάση περιέχει τα γενικά χαρακτηριστικά μίας οντότητας (π.χ. `Product`). Η παράγωγη κλάση αποτελεί συνήθως μία εξειδίκευση της βασικής (π.χ. `Book`), η οποία κληρονομεί τα μέλη της βασικής και προσθέτει τα δικά της χαρακτηριστικά. Έτσι, ένα αντικείμενο μίας παράγωγης κλάσης, εκτός από τα δικά του μέλη, περιέχει ένα υπο-αντικείμενο της βασικής κλάσης. Για παράδειγμα, δείτε το παρακάτω πρόγραμμα:

Παράδειγμα

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

class Product
{
public:
     string code;
     float prc;

     Product() {cout << "Base Constructor\n";}
     ~Product() {cout << "Base Destructor\n";}
void show() const {cout << "\nC:" << code << '\n';}
};

class Book : public Product
{
public:
     string title;
     string auth;

     Book() {cout << "Derived Constructor\n";}
     ~Book() {cout << "Derived Destructor\n";}
     void display() const {cout << "T:" << title <<
     " A:" << auth << " P:" << prc << '\n';} /* Η
     συνάρτηση έχει πρόσβαση στα μέλη της βασικής
     κλάσης. */
};
```

```
int main()
{
          Book b;

          cout << "Product Details: ";
          cin >> b.code >> b.prc; /*
Προσπέλαση των μελών της βασικής κλάσης. */
          cout << "Book Details: ";
          cin >> b.title >> b.auth;

          b.show(); /* Κλήση συνάρτησης της
βασικής κλάσης. */
          b.display();
          return 0;
}
```

Παράδειγμα

- Το κύριο πλεονέκτημα της κληρονομικότητας είναι ότι μας επιτρέπει να χρησιμοποιήσουμε υπάρχοντα κώδικα, χωρίς να χρειάζεται να τον ξαναγράψουμε
- Αυτό που έχουμε να κάνουμε είναι να γράψουμε τον κώδικα που υλοποιεί την παράγωγη κλάση
- Στο παράδειγμά μας, η `Book` κληρονόμησε όλα τα μέλη και τις συναρτήσεις της `Product`. Δεν χρειάζεται να τα προσθέσουμε πάλι σε αυτήν. Δηλαδή, ένα `Book` αντικείμενο κληρονομεί τα μέλη `code` και `prc` και τη συνάρτηση `show()`. Επίσης, η `Book` περιέχει τα δικά της μέλη και συναρτήσεις
- Όταν δημιουργείται ένα αντικείμενο της παράγωγης κλάσης, **πρώτα** καλείται ο κατασκευαστής της παράγωγης κλάσης. Αυτός με τη σειρά του **καλεί** κατασκευαστή της βασικής κλάσης για να **κατασκευάσει** το **αντικείμενο** της **βασικής** κλάσης που **περιέχεται** στο **παράγωγο** αντικείμενο

Παράδειγμα

- Δηλαδή, όπως είπαμε, το παράγωγο αντικείμενο περιέχει ένα υπο-αντικείμενο της βασικής κλάσης
- Αυτή η σχέση αναφέρεται και ως «**είναι-ένα**» (is-a), με την έννοια ότι κάθε αντικείμενο μίας παράγωγης κλάσης **είναι** επίσης ένα αντικείμενο της βασικής
- Για παράδειγμα, ένα **Book** αντικείμενο είναι επίσης ένα **Product** αντικείμενο
- Ειδικότερα, **πρώτα** πρέπει να δημιουργηθεί το υπο-αντικείμενο της βασικής κλάσης και **μετά** το τμήμα του παράγωγου αντικειμένου
- Άρα, όταν δημιουργείται το **b** ο κατασκευαστής της **Book** **πρώτα** καλεί τον κατασκευαστή της **Product** για να δημιουργήσει το **Product** υπο-αντικείμενο και **μετά** συνεχίζει με την κατασκευή του **b**, δηλαδή, εκτελείται το σώμα του **Book** κατασκευαστή
- Επομένως, το πρόγραμμα αρχικά εμφανίζει:

```
Base Constructor
Derived Constructor
```

- Γενικά, σε μία ιεραρχία κλάσεων (π.χ. **A**, **B**, **C**, **D**), όταν δημιουργείται ένα αντικείμενο της παράγωγης κλάσης (π.χ. **D**), πρώτα καλείται ο κατασκευαστής της υψηλότερης βασικής κλάσης (π.χ. **A**), μετά ο κατασκευαστής της κλάσης που παράγεται από αυτήν (π.χ. **B**), μετά της επόμενης (π.χ. **C**), κ.ο.κ.. Για παράδειγμα, πρώτα δημιουργούνται τα **A**, **B**, **C** υπο-αντικείμενα που περιέχονται στο **D** αντικείμενο και μετά ο **D** κατασκευαστής ολοκληρώνει την κατασκευή του **D** αντικειμένου

Παράδειγμα

- Επειδή η Book παράγεται με δημόσια πρόσβαση, το b μπορεί να προσπελάσει τα δημόσια μέλη της Product
- Το πρόγραμμα διαβάζει την πληροφορία που εισάγει ο χρήστης, την αποθηκεύει στα πεδία της Product και της Book και καλεί συναρτήσεις και από τις δύο κλάσεις για να την εμφανίσει
- Όταν καταστρέφεται το παράγωγο αντικείμενο η αποδόμηση γίνεται σε **αντίστροφη** σειρά από τη σειρά κατασκευής, δηλαδή, το πρόγραμμα **πρώτα** καλεί τον αποδομητή της παράγωγης κλάσης και **μετά** της βασικής
- Επομένως, το πρόγραμμα εμφανίζει:

```
Derived Destructor
Base Destructor
```

Παράδειγμα

- Βρείτε τα λάθη στο παρακάτω πρόγραμμα:

```
#include <iostream>

class A
{
private:
        int a;
        void f();
protected:
        int p;
        void g();
};

class B : public A
{
public:
        int c;
        void h();
};
void A::f()
{
        std::cout << c << '\n';
}
void A::g()
{
        std::cout << a << ' ' << p << '\n';
}
void B::h()
{
        std::cout << p << ' ' << c << '\n';
}
int main()
{
        B b;

        std::cin >> b.a >> b.p;
        b.g();
        b.h();
        return 0;
}
```

Παράδειγμα

- Υπάρχει πρόβλημα που καμία κλάση δεν έχει κατασκευαστή; Όχι βέβαια, θα κληθούν οι εξ'ορισμού κατασκευαστές. Να θυμάστε, **πρώτα** εκτελείται ο εξ'ορισμού κατασκευαστής της βασικής κλάσης και **μετά** της παράγωγης
- Συνεχίζουμε, αφού τα μέλη `a` και `f()` είναι ιδιωτικά, τα αντικείμενα αλλά και οι συναρτήσεις της κλάσης `B` **δεν επιτρέπεται** να έχουν πρόσβαση σε αυτά
- Όσον αφορά τα **προστατευμένα** μέλη, θυμηθείτε ότι για την **ίδια** την κλάση συμπεριφέρονται όπως τα ιδιωτικά. Για **παράγωγη** κλάση, οι συναρτήσεις της **επιτρέπεται** να έχουν πρόσβαση στα προστατευμένα μέλη της βασικής, ενώ τα αντικείμενά της **όχι**. Επομένως, οι ακόλουθες εντολές δεν είναι αποδεκτές:

```
cin >> b.a >> b.p; /* Δεν επιτρέπεται η πρόσβαση στα πεδία a και p. */
b.g();
```

- Η κλάση `A` δεν περιέχει μέλος με όνομα `c`. Το πεδίο `c` έχει δηλωθεί στην παράγωγη κλάση `B`, στην οποία η κλάση `A` δεν έχει πρόσβαση. Άρα, η ακόλουθη εντολή δεν είναι αποδεκτή:

```
cout << c << '\n'; // Στην f().
```

Και αυτό είναι λογικό. Η κληρονομικότητα δεν λειτουργεί ανάποδα, δηλαδή, μία βασική κλάση και τα αντικείμενά της δεν γνωρίζουν τίποτα για κλάσεις που παράγονται από αυτήν και επομένως δεν έχουν πρόσβαση στα μέλη τους

Is-a και Has-a Σχέσεις

- Όπως είπαμε, η κληρονομικότητα κανονικά αναπαριστά στον σχεδιασμό του προγράμματος την σχέση «είναι ένα» (is-a), δηλαδή, ένα αντικείμενο παράγωγης κλάσης θα πρέπει να είναι επίσης ένα αντικείμενο της βασικής κλάσης. Για παράδειγμα, ένα Book αντικείμενο είναι επίσης ένα Product αντικείμενο
- Θεωρήστε ότι έχουμε ορίσει την κλάση Engine, την κλάση Tires, και θέλουμε να ορίσουμε την Car κλάση. Αφού ένα αυτοκίνητο έχει μηχανή και λάστιχα, η ταιριαστή επιλογή είναι να χρησιμοποιήσουμε την σχέση «έχει» (has-a) και να γράψουμε:

```
class Car
{
        ...
        Engine eng;
        Tires tir;
};
```

- Τώρα, υποθέστε ότι έχουμε ορίσει την Shape κλάση και θέλουμε να ορίσουμε την Circle κλάση. Αφού ένας κύκλος είναι σχήμα, η ταιριαστή επιλογή είναι να χρησιμοποιήσουμε την is-a σχέση και να γράψουμε:

```
class Circle : public Shape
{
        ...
};
```

Is-a και Has-a Σχέσεις

- Παρόμοια, υποθέστε ότι έχουμε ορίσει την `BankAccount` κλάση και θέλουμε να ορίσουμε την `CheckingAccount` κλάση. Αφού ένας λογαριασμός όψεως είναι ένας ειδικός τύπος τραπεζικού λογαριασμού, ταιριάζει να χρησιμοποιήσουμε κληρονομικότητα και να παράξουμε την `CheckingAccount` κλάση από την `BankAccount`

- Ποια σχέση θα επιλέγαμε αν είχαμε τις `Animal` και `Cat` κλάσεις; Αφού μία γάτα είναι είδος ζώου, ταιριάζει να χρησιμοποιήσουμε κληρονομικότητα και να παράξουμε την `Cat` κλάση από την `Animal`

- Γενικά, όταν πρέπει να αποφασίσετε αν θα προσθέσετε μια κλάση ως μέλος ή θα χρησιμοποιήσετε κληρονομικότητα, ο γενικός κανόνας είναι να δείτε ποια από τις has-a και is-a σχέσεις ταιριάζει καλύτερα και να ενεργήσετε ανάλογα

Εμβέλεια και Κληρονομικότητα

- Όταν ένα όνομα χρησιμοποιείται σε μία κλάση, ο μεταγλωττιστής ψάχνει για τη δήλωσή του **πρώτα** στην εμβέλεια της κλάσης
- Αν δεν το βρει, **συνεχίζει** την αναζήτηση στις εμβέλειες των βασικών της κλάσεων, σε **όλη** την αλυσίδα από την **απευθείας** βασική της κλάση μέχρι την **κορυφή**
- Για παράδειγμα, έστω μία κλάση `C` παράγεται από την κλάση `B`, η οποία παράγεται από την κλάση `A` και μόνο αυτή περιέχει ένα δημόσιο μέλος με το όνομα `p`. Αν γράψουμε:

```
C c;
c.p = 10;
```

αφού το `c` είναι αντικείμενο της κλάσης `C` ο μεταγλωττιστής ψάχνει για την δήλωση του `p` σε αυτήν. Επειδή το `p` δεν δηλώνεται στη `C`, ο μεταγλωττιστής συνεχίζει την αναζήτηση στη βασική της κλάση, στην `B`. Παρόμοια, επειδή το `p` δεν δηλώνεται στην `B` και επειδή η `B` παράγεται από την `A`, η αναζήτηση συνεχίζεται σε αυτήν, όπου και τελικά βρίσκεται. Αν δεν υπήρχε η δήλωση, ο μεταγλωττιστής θα εμφάνιζε σχετικό μήνυμα λάθους

Εμβέλεια και Κληρονομικότητα

- Μία λογική ερώτηση είναι, και τι θα γίνει αν το όνομα p εμφανίζεται και σε άλλη κλάση; Για παράδειγμα:

```
void g() // Καθολική g().
{
        ...
}

class A
{
public:
        int p;
        ...
};

class B : public A
{
public:
        int p;
        ...
};
```

```
class C : public B
{
public:
    int p;
    void f();
    ...
};
void C::f()
{
    g();
}
```

Εμβέλεια και Κληρονομικότητα

- Όπως ξέρουμε από το Κ.11, τα ονόματα που δηλώνονται σε μία ένθετη εμβέλεια **κρύβουν** τα ονόματα έξω από αυτήν
- Επομένως, με την εντολή `c.p` αναφερόμαστε στο μέλος `p` της κλάσης `C`. Επειδή ο μεταγλωττιστής βρίσκει το `p` στην `C` **σταματάει** την αναζήτηση. Αν όμως το `p` δεν είχε δηλωθεί στην `C`, θα **συνέχιζε** την αναζήτηση στη βασική της κλάση και, άρα, θα αναφερόμασταν στο μέλος `p` της κλάσης `B`
- Και τι κάνουμε αν θέλουμε να αναφερθούμε στο μέλος `p` κάποιας άλλης κλάσης; Χρησιμοποιούμε τον τελεστή επίλυσης εμβέλειας `::`. Για παράδειγμα:

```
C c;
c.A::p = 10;
c.B::p = 20;
c.p = 30;
```

- Σημειώστε ότι ο μεταγλωττιστής εφαρμόζει την **ίδια** διαδικασία αναζήτησης για οποιοδήποτε όνομα, δεν έχει σημασία αν είναι μεταβλητή, συνάρτηση ή κάτι άλλο
- Για παράδειγμα, έστω ότι είναι όνομα συνάρτησης. Αν κληθεί η `f()`, ο μεταγλωττιστής ψάχνει για την δήλωση της `g()` στην εμβέλεια της `C`. Αν δεν την βρει συνεχίζει στην εμβέλεια της `B`, και μετά στην εμβέλεια της `A`. Αν δεν την βρει, συνεχίζει την αναζήτηση για κάποια καθολική `g()`, και αν πάλι αποτύχει εμφανίζει μήνυμα λάθους

Παράγωγες Κλάσεις και Κατασκευαστές

- Ας αλλάξουμε το πρώτο πρόγραμμα, να προσθέσουμε παραμέτρους στους δύο κατασκευαστές και να δούμε με ποιο τρόπο ο κατασκευαστής της παράγωγης κλάσης μπορεί να μεταβιβάσει τιμές στον κατασκευαστή της βασικής

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

class Product
{
private:
        string code;
        float prc;
public:
        Product(const string& c, float p);
        void show() const;
};

class Book : public Product
{
private:
        string title;
        string auth;
public:
        Book(const string& c, float p, const string& t, const string& a);
        void display() const;
};
```

Παράγωγες Κλάσεις και Κατασκευαστές

```
Product::Product(const string& c, float p)
{
    code = c;
    prc = p;
}

void Product::show() const
{
    cout << "C:" << code << " P:" << prc << '\n';
}

Book::Book(const string& c, float p, const string& t, const string& a) : Product(c, p) /* Πρώτα
    θα κληθεί ο κατασκευαστής της βασικής κλάσης και μετά θα εκτελεστούν οι παρακάτω εντολές.
    */
{
    title = t;
    auth = a;
}

void Book::display() const
{
    show();
    cout << "T:" << title << " A:" << auth << '\n';
}

int main()
{
    Book b("AB25", 8.5, "Nice", "Many");

    b.display();
    return 0;
}
```

Παράγωγες Κλάσεις και Κατασκευαστές

- Όπως είπαμε, όταν δημιουργείται ένα αντικείμενο μίας παράγωγης κλάσης, πρέπει πρώτα να δημιουργηθεί το υπο-αντικείμενο της βασικής
- Και πώς θα γίνει αυτό; Απλά, η παράγωγη κλάση πρέπει να καλέσει έναν κατασκευαστή της βασικής κλάσης για να αρχικοποιήσει το υπο-αντικείμενο της βασικής κλάσης
- Όταν λοιπόν δημιουργείται το αντικείμενο `b` καλείται ο κατασκευαστής της κλάσης `Book` και μεταβιβάζονται οι τιμές των ορισμάτων στις αντίστοιχες παραμέτρους. Για παράδειγμα, το `c` θα γίνει ίσο με `AB25` και το `p` ίσο με `8.5`
- Μετά την μεταβίβαση των τιμών, εκτελείται το τμήμα του κώδικα:

```
: Product(c, p)
```

το οποίο καλεί τον κατασκευαστή της βασικής κλάσης χρησιμοποιώντας τη σύνταξη αρχικοποίησης μελών που είδαμε στο Κ.19 και του μεταβιβάζει τις τιμές των `c` και `p`

Παρατηρήσεις

- Θυμόμαστε, όταν το πρόγραμμα δημιουργεί ένα αντικείμενο μίας παράγωγης κλάσης πρώτα δημιουργείται το υπο-αντικείμενο της βασικής
- Ο κατασκευαστής της παράγωγης κλάσης καλεί κατασκευαστή της βασικής
- Ο προγραμματιστής μπορεί να επιλέξει τον κατασκευαστή της βασικής κλάσης που θα κληθεί, αλλιώς, αν δεν επιλέξει κατασκευαστή, το πρόγραμμα θα καλέσει τον εξ'ορισμού κατασκευαστή, εφόσον αυτός υπάρχει
- Αν δεν υπάρχει, ο μεταγλωττιστής θα εμφανίσει μήνυμα λάθους
- Όταν ένα αντικείμενο της παράγωγης κλάσης καταστρέφεται, πρώτα καλείται ο αποδομητής της παράγωγης κλάσης και μετά της βασικής

Ιεραρχία Κλάσεων

- Σε μία αλυσίδα παραγώγων κλάσεων, κάθε κλάση μπορεί να χρησιμοποιήσει τη σύνταξη αρχικοποίησης για να μεταβιβάσει τιμές στον κατασκευαστή της αμέσως προηγούμενης βασικής κλάσης
- **Προσέξτε** όμως, **μόνο** της προηγούμενης βασικής κλάσης και όχι κάποιας άλλης σε υψηλότερο επίπεδο
- **Εξαίρεση** σε αυτόν τον κανόνα αποτελούν οι εικονικές βασικές κλάσεις

Παράδειγμα

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

class A
{
public:
        A(int k) {cout << "A:" << k;}
        ~A() {cout << " ~A\n";}
};

class B : public A
{
public:
        B(int m) : A(m+1) {cout << " B:" << m;}
        ~B() {cout << " ~B";}
};

class C : public B
{
public:
        C(int p) : B(2*p) {cout << " C:" << p << '\n';}
        ~C() {cout << "~C";}
};

int main()
{
        C c(10);
        return 0;
}
```

Ιεραρχία Κλάσεων

- Όταν δημιουργείται το αντικείμενο c καλείται ο κατασκευαστής της C κλάσης, ο οποίος καλεί τον κατασκευαστή της B κλάσης και του μεταβιβάζει την τιμή 20
- Αυτός με τη σειρά του καλεί τον κατασκευαστή της A κλάσης, του μεταβιβάζει την τιμή 21 και εκτελείται το σώμα του. Το πρόγραμμα εμφανίζει A:21
- Μετά, εκτελείται το σώμα του B κατασκευαστή και το πρόγραμμα εμφανίζει B:20
- Πρώτα λοιπόν δημιουργείται το υπο-αντικείμενο της κλάσης A και μετά το υπο-αντικείμενο της κλάσης B
- Δηλαδή, το c αντικείμενο περιέχει δύο υπο-αντικείμενα, ένα της κλάσης A και ένα της κλάσης B. Μετά την κατασκευή αυτών των δύο υπο-αντικειμένων εκτελείται το σώμα του C κατασκευαστή και το πρόγραμμα εμφανίζει C:10
- Γενικά, σε μία ιεραρχία κλάσεων, ένα παράγωγο αντικείμενο περιέχει ένα υπο-αντικείμενο της άμεσης βασικής του κλάσης και ένα υπο-αντικείμενο για κάθε μία από τις έμμεσες βάσεις του
- Οι αποδομητές καλούνται με την αντίστροφη σειρά από αυτήν με την οποία κλήθηκαν οι κατασκευαστές, σε όλη την ιεραρχία των κλάσεων, από την τελευταία παράγωγη κλάση μέχρι τη ρίζα. Τελικά, το πρόγραμμα εμφανίζει:

```
A:21        B:20        C:10
~C          ~B          ~A
```

Ιεραρχία Κλάσεων

- Όπως είπαμε, ο κατασκευαστής της C κλάσης δεν επιτρέπεται να χρησιμοποιήσει την παραπάνω σύνταξη για να καλέσει απευθείας κατασκευαστή της A κλάσης
- Αυτό που επιτρέπεται είναι να καλέσει κατασκευαστή μόνο από την απευθείας βασική του κλάση, δηλαδή, από την B, και ένας B κατασκευαστής να καλέσει κατασκευαστή μόνο από την A κλάση.
- Για παράδειγμα, δεν επιτρέπεται να γράψουμε:

```
C(int p) : A(p) {} // Λάθος μεταγλώττισης.
```

- Αν δεν θέλουμε μία κλάση να κληρονομηθεί χρησιμοποιούμε τη λέξη `final`, η οποία εισήχθηκε με την C++11. Για παράδειγμα:

```
class A final {...}; /* Η κλάση A δεν μπορεί να είναι βασική κλάση */
class B : public A {...}; // Λάθος.
```

Ο μεταγλωττιστής θα εμφανίσει μήνυμα λάθους παρόμοιο με «A is not a direct base of B».

Παράδειγμα

- Ποια είναι τα λάθη στο παρακάτω πρόγραμμα;

```
#include <iostream>

class A
{
private:
        int p;
public:
        A(int a) {p = a;}
};

class B : public A
{
public:
        B(int v) {std::cout << v << '\n';}
        void show(int v) const {std::cout << p+v << '\n';}
};

int main()
{
        B b(10);
        A& r = b;

        r.show(20);
        return 0;
}
```

Παράδειγμα

- Απάντηση: Το πρώτο λάθος είναι στον ορισμό της `show()`. Αφού το `p` είναι ιδιωτικό μέλος, η `show()` δεν έχει άμεση πρόσβαση σε αυτό. Το δεύτερο λάθος είναι στη δημιουργία του `b`. Αφού ο κατασκευαστής της `B` δεν καλεί κατασκευαστή της `A` δεν μπορεί να δημιουργηθεί το `A` αντικείμενο. Για παράδειγμα, αν γράφαμε `B(int v):A(v)` ή αν είχαμε ορίσει τον εξ'ορισμού κατασκευαστή στην `A` δεν θα υπήρχε πρόβλημα. Το `r` μπορεί να αναφερθεί σε `B` αντικείμενο, αλλά επειδή έχει δηλωθεί σαν αναφορά στην κλάση `A` και η `A` δεν περιέχει `show()` είναι λάθος να την καλέσουμε

Μετατροπές από Παραγωγή σε Βασική Κλάση

- Μία ειδική σχέση μεταξύ **βασικής** και **παράγωγης** κλάσης είναι ότι ένας δείκτης ή αναφορά σε βασική κλάση **μπορεί** να δείξει ή να αναφερθεί σε αντικείμενο παράγωγης κλάσης
- Σημειώστε ότι **δεν χρειάζεται** προσαρμογή τύπου. Αυτή η μετατροπή συχνά ονομάζεται παράγωγη-σε-βασική (derived-to-base) μετατροπή. Για παράδειγμα:

```
Book b("AB25", 8.5, "Nice", "Many");
Product *p = &b; /* Η μετατροπή δείκτη από παράγωγη κλάση
(π.χ. &b) σε δείκτη σε βασική (π.χ. p) επιτρέπεται. Ο p δείχνει
στο Product υπο-αντικείμενο του b. */
Product &r = b; /* Το r αναφέρεται στο Product υπο-
αντικείμενο του b. */
p->show();
r.show();
```

- Η μετατροπή που εφαρμόζει ο μεταγλωττιστής είναι **ασφαλής**, γιατί, όπως ξέρουμε, ένα `Book` αντικείμενο περιέχει ένα `Product` υπο-αντικείμενο ή, αλλιώς, ένα `Book` αντικείμενο είναι επίσης ένα `Product` αντικείμενο

Μετατροπές από Παραγωγή σε Βασική Κλάση

- Γενικά, ένα αντικείμενο παράγωγης κλάσης ή μία αναφορά σε αυτό μπορεί να χρησιμοποιηθεί σε εκφράσεις όπου μπορεί να χρησιμοποιηθεί αντικείμενο της βασικής του κλάσης
- Παρόμοια, μπορούμε να χρησιμοποιήσουμε ένα δείκτη σε αντικείμενο της παράγωγης κλάσης σε εκφράσεις όπου μπορεί να χρησιμοποιηθεί δείκτης στη βασική κλάση
- Βέβαια, μία αναφορά ή ένας δείκτης σε βασική κλάση **δεν επιτρέπεται** να προσπελάσει μέλη της παράγωγης κλάσης. Για παράδειγμα, δεν επιτρέπεται να γράψουμε:

```
p->display(); // Λάθος.
r.display(); // Λάθος.
```

Μετατροπές από Παραγωγή σε Βασική Κλάση

- Επειδή ένας δείκτης ή μία αναφορά σε βασική κλάση μπορεί να αναφερθεί σε παράγωγη κλάση (όπως είπαμε, αναφέρεται στο υπο-αντικείμενο της βασικής που υπάρχει στην παράγωγη), μία συνάρτηση που δέχεται σαν όρισμα δείκτη ή αναφορά σε αντικείμενο βασικής κλάσης μπορεί να δεχτεί δείκτη ή αναφορά και σε αντικείμενο παράγωγης κλάσης. Για παράδειγμα:

```
void f(Product& r)
{
        r.show();
}
```

- Αφού η παράμετρος `r` είναι αναφορά σε βασική κλάση, μπορεί να αναφερθεί σε αντικείμενο βασικής ή παράγωγης κλάσης χωρίς καμία ανησυχία για τη λειτουργία της συνάρτησης. Έτσι, μπορούμε να γράψουμε:

```
Product p("One", 1);
Book b("AB25", 8.5, "Nice", "Many");
f(p);
f(b); // Το Book αντικείμενο είναι και Product αντικείμενο.
```

- Σημειώστε ότι αυτή η ιδιότητα είναι μεταβατική. Δηλαδή, αν δημιουργήσουμε την κλάση `ProgrammingBook` που κληρονομεί την `Book`, τότε ένας δείκτης ή αναφορά στην κλάση `Product` μπορεί να αναφερθεί σε ένα `Product` ή `Book` ή `ProgrammingBook` αντικείμενο

Μετατροπές από Παραγωγή σε Βασική Κλάση

- Μπορούμε να εκχωρήσουμε ένα αντικείμενο μίας παράγωγης κλάσης σε ένα αντικείμενο βασικής κλάσης και **μόνο** τα μέλη της βασικής κλάσης θα αντιγραφούν. Για παράδειγμα:

```
Book b("AB25", 8.5, "Nice", "Many");
Product p;
p = b;
```

- Η εκχώρηση `p = b;` αντιγράφει τις τιμές του `Product` υπο-αντικειμένου που περιέχεται στο `b` στα πεδία του `p`. Όπως θα δούμε στη συνέχεια, το αντίστροφο, δηλαδή, το `b = p;` **δεν επιτρέπεται** αυτόματα, αφού ένα `Product` αντικείμενο δεν είναι και `Book` αντικείμενο
- Επίσης, σε μία συνάρτηση που δέχεται σαν παράμετρο αντικείμενο της βασικής κλάσης μπορούμε να μεταβιβάσουμε αντικείμενο παράγωγης κλάσης. Για παράδειγμα:

```
void f(Product p)
{
        p.show();
}
Book b("AB25", 8.5, "Nice", "Many");
f(b); // Το Book αντικείμενο είναι και Product αντικείμενο.
```

- Με την κλήση της `f()` ο κατασκευαστής αντιγράφου της `Product` αναλαμβάνει να αντιγράψει το `Product` τμήμα του `b` στο `p`. Παρόμοια με πριν, το αντίστροφο δεν επιτρέπεται, δηλαδή, **δεν επιτρέπεται** να μεταβιβάσουμε αντικείμενο βασικής κλάσης σε συνάρτηση που δέχεται αντικείμενο παράγωγης κλάσης
- Θυμόμαστε, όταν εκχωρούμε ή αρχικοποιούμε ένα αντικείμενο βασικής κλάσης με ένα αντικείμενο παράγωγης κλάσης **μόνο** τα μέλη του βασικού υπο-αντικειμένου του αντιγράφονται. Το υπόλοιπο τμήμα του παράγωγου αντικειμένου **αγνοείται**

Μετατροπές από Βασική σε Παραγωγή Κλάση

- Η ανάποδη μετατροπή (base-to-derived) **δεν επιτρέπεται** με αυτόματο τρόπο, δηλαδή, μία αναφορά ή ένας δείκτης σε παράγωγη κλάση **δεν επιτρέπεται** να δείξει σε αντικείμενο βασικής κλάσης χωρίς προσαρμογή τύπου
- Και αυτό είναι λογικό, γιατί ένα αντικείμενο βασικής κλάσης **δεν περιέχει** υπο-αντικείμενο της παράγωγης κλάσης, στο οποίο να μπορεί να αναφερθεί ή να δείξει ένας δείκτης σε παράγωγη κλάση
- Μόνο το αντίστροφο ισχύει, όπως είδαμε στην προηγούμενη ενότητα. Για παράδειγμα:

```
Product prod("AB25", 8.5);
Book *p = &prod; /* Λάθος, δεν επιτρέπεται μετατροπή από δείκτη σε βασική κλάση (π.χ. &prod) σε δείκτη σε παράγωγη (π.χ. p). Ενα Product αντικείμενο δεν είναι Book αντικείμενο. */
Book &r = prod; // Λάθος.
```

- Όμως, αν χρησιμοποιήσουμε προσαρμογή τύπου ο κώδικας μεταγλωττίζεται. Για παράδειγμα:

```
Book *p = static_cast<Book*>(&prod);
Book &r = static_cast<Book&>(prod);
```

- Ωστόσο, αυτή η λειτουργία **δεν είναι** ασφαλής, γιατί ένα `Product` αντικείμενο δεν περιέχει ένα `Book` υπο-αντικείμενο. Αν δοκιμάσουμε να προσπελάσουμε μέλη της παράγωγης κλάσης, μπορεί να προκύψουν προβλήματα στην εκτέλεση του προγράμματος. Για παράδειγμα, αν γράψουμε κατά λάθος `p->display();` μπορεί να προκύψουν προβλήματα αφού η `display();` δεν είναι μέλος της `Product` κλάσης

Μετατροπές από Βασική σε Παραγωγή Κλάση

- Είδαμε στην προηγούμενη ενότητα ότι μπορούμε να εκχωρήσουμε ένα αντικείμενο παράγωγης κλάσης σε ένα αντικείμενο βασικής κλάσης. Το ανάποδο δεν επιτρέπεται. Για παράδειγμα:

```
Book b("AB25", 8.5, "Nice", "Many");
Product p;
b = p; // Λάθος.
```

Επανορισμός Συνάρτησης

- Όπως ήδη γνωρίζουμε, ένα αντικείμενο μίας παράγωγης κλάσης μπορεί να χρησιμοποιήσει συναρτήσεις της βασικής κλάσης
- Υπάρχει όμως περίπτωση να θέλουμε μία συνάρτηση της βασικής κλάσης να συμπεριφέρεται διαφορετικά στην παράγωγη κλάση, δηλαδή, να θέλουμε να έχουμε **πολλαπλές** μορφές (polymorphism) της **ίδιας** συνάρτησης
- Έτσι, ο τρόπος που μία συγκεκριμένη συνάρτηση συμπεριφέρεται μπορεί να εξαρτάται από το αντικείμενο που την καλεί
- Για να επιτευχθεί ο **πολυμορφισμός** πρέπει να ορίσουμε πάλι την συνάρτηση στην παράγωγη κλάση
- Υπάρχουν δύο περιπτώσεις για να ελέγξουμε. Η πρώτη είναι όταν οι συναρτήσεις έχουν την **ίδια** υπογραφή και η δεύτερη με **διαφορετικές** υπογραφές

Επανορισμός με Ίδια Υπογραφή

- Ως παράδειγμα, ας χρησιμοποιήσουμε το ίδιο πρόγραμμα με τις Product και Book κλάσεις. Ας αλλάξουμε το όνομα της display() στην κλάση Book σε show() και να γράψουμε τον παρακάτω κώδικα:

```
void Book::show() const
{
        cout << "T:" << title << " A:" << auth << '\n';
}
```

- Τώρα, έχουμε δύο εκδόσεις της show(), μία στη βασική κλάση και μία στην παράγωγη. Αν γράψουμε τον παρακάτω κώδικα:

```
Product p("A", 1);
Book b("B", 2, "Nice", "Many");
p.show(); // Καλείται η Product::show().
b.show(); // Καλείται η Book::show().
```

- Ο μεταγλωττιστής ελέγχει τον τύπο του αντικειμένου για να αποφασίσει ποια από τις δύο εκδόσεις να καλέσει
- Ουσιαστικά, όταν μία συνάρτηση βασικής κλάσης επανορίζεται σε μία παράγωγη κλάση, η συνάρτηση της παράγωγης κλάσης **κρύβει** τη συνάρτηση της βασικής κλάσης

Επανορισμός με Ίδια Υπογραφή

- Και αν θέλουμε να καλέσουμε την show() της βασικής κλάσης από την παράγωγη κλάση υπάρχει τρόπος; Ναι, απλά, χρησιμοποιούμε τον τελεστή επίλυσης εμβέλειας με το όνομα της βασικής κλάσης. Για παράδειγμα:

```
void Book::show()
{
        cout << "T:" << title << " A:" << auth << '\n';
        Product::show(); /* Προσοχή, αν γράψουμε μόνο
show() θα δημιουργηθεί μία ατέρμονη αναδρομή. */
}
```

Επανορισμός με Διαφορετικές Υπογραφές

- Ας δούμε τώρα την περίπτωση που η συνάρτηση της βασικής κλάσης δηλώνεται διαφορετικά στην παράγωγη κλάση. Για παράδειγμα:

```
#include <iostream>

class A
{
public:
        void show() const {std::cout << "A\n";}
};
class B : public A
{
public:
        void show(int k) const {std::cout << k << '\n';}
};
class C : public B
{
public:
        void show(const char *s) const {std::cout << s << '\n';}
};
int main()
{
        B b;
        b.show(10); // Μεταγλωττίζεται κανονικά.
        b.show(); // Προκαλεί λάθος μεταγλώττισης.

        C c;
        c.show("Test"); // Μεταγλωττίζεται κανονικά.
        c.show(10); // Προκαλεί λάθος μεταγλώττισης.
        c.show(); // Προκαλεί λάθος μεταγλώττισης.
        return 0;
}
```

Επανορισμός με Διαφορετικές Υπογραφές

- Ο επανορισμός μίας συνάρτησης **κρύβει** τις δηλώσεις όλων των συναρτήσεων με το **ίδιο** όνομα στις βασικές κλάσεις, ανεξάρτητα από τις **υπογραφές** των συναρτήσεων
- Ο επανορισμός μίας συνάρτησης σε παράγωγη κλάση **δεν αποτελεί** υπερφόρτωση συνάρτησης
- Έτσι, με τη δήλωση της `show()` στην κλάση `B` δεν έχουμε δύο υπερφορτωμένες εκδόσεις της `show()`, αλλά αυτή η δήλωση κρύβει την δήλωση στη βασική κλάση `A`. Ειδικότερα, με την εντολή `b.show();` ο μεταγλωττιστής ψάχνει το όνομα `show` στην `B`. Όταν το βρει, **σταματάει** την αναζήτηση
- Μετά, επειδή βλέπει ότι για την κλήση της `show()` πρέπει να μεταβιβαστεί όρισμα, εμφανίζει μήνυμα λάθους
- Το σημαντικό να καταλάβετε είναι ότι ο μεταγλωττιστής **σταματάει** την αναζήτηση, **δεν συνεχίζει** να ψάχνει για κάποια άλλη έκδοση της `show()` στις βασικές κλάσεις της `B` που να ταιριάζει με την κλήση
- Παρόμοια, η δήλωση της `show()` στην παράγωγη κλάση `C` **κρύβει** τις δηλώσεις της στις βασικές κλάσεις `A` και `B`. Επομένως, ο μεταγλωττιστής θα εμφανίσει άλλα δύο μηνύματα λάθους
- Για να καλέσουμε τη `show()` μίας βασικής κλάσης πρέπει να χρησιμοποιήσουμε τον τελεστή εμβέλειας και το όνομα της κλάσης. Για παράδειγμα, `c.B.show(10);`

Επανορισμός με Διαφορετικές Υπογραφές

- Και αν θέλουμε να έχουμε υπερφορτωμένες εκδόσεις μίας συνάρτησης από την βασική κλάση στην παράγωγη κλάση υπάρχει τρόπος;
- Ναι, χρησιμοποιούμε **using** δηλώσεις και γράφουμε το όνομα των συνάρτησης. Η **using** δήλωση εισάγει την συνάρτηση στον χώρο εμβέλειας της παράγωγης κλάσης, άρα, οι εκδόσεις είναι διαθέσιμες στην παράγωγη κλάση
- Ο μεταγλωττιστής ελέγχει τις υπογραφές τους για να τις ξεχωρίζει. Για παράδειγμα:

```
class C : public B
{
public:
        using B::A::show;
        using B::show;
        void show(const char *s) const {std::cout << s << '\n';}
};
int main()
{
        C c;
        c.show("Test"); // Καλείται η C::show().
        c.show(10); // Καλείται η B::show().
        c.show(); // Καλείται η A::show().
        return 0;
}
```

Στατική Σύνδεση

- Θεωρήστε ότι μία συνάρτηση παράγωγης κλάσης επανορίζει με την ίδια υπογραφή μία συνάρτηση της βασικής κλάσης. Τι συμβαίνει όταν χρησιμοποιούμε δείκτες και αναφορές για να καλέσουμε αυτή την συνάρτηση; Για παράδειγμα, ας χρησιμοποιήσουμε πάλι το ίδιο πρόγραμμα με την παρακάτω show() αντί της display()

```
void Book::show() const
{
        cout << "T:" << title << " A:" << auth << '\n';
}
```

- Τι πιστεύετε ότι θα εμφανίσει ο παρακάτω κώδικας;

```
Product p("A", 1);
Book b("B", 2, "Nice", "Many");

Product *ptr1 = &p;
ptr1->show();
Product *ptr2 = &b;
ptr2->show(); // Ποια show() θα κληθεί;
Product& ref = b;
ref.show(); // Ποια show() θα κληθεί;
```

Στατική Σύνδεση

- Στην περίπτωση της `ptr2->show();` το πιθανότερο είναι να απαντήσατε ότι αφού ο `ptr2` δείχνει στο αντικείμενο `b` θα κληθεί η `show()` της κλάσης `Book`
- **Λάθος**, ο μεταγλωττιστής ελέγχει τον **τύπο** του δείκτη και αποφασίζει ανάλογα
- Έτσι, αφού ο τύπος του `ptr2` είναι δείκτης στην κλάση `Product` θα καλέσει τη δική της `show()`
- Δηλαδή, η απόφαση για το ποια συνάρτηση θα κληθεί λαμβάνεται κατά την **μεταγλώττιση** του προγράμματος, και αυτή η απόφαση **δεν** θα αλλάξει
- Αυτός ο τρόπος επιλογής ονομάζεται **στατική σύνδεση** (static binding)
- Το ίδιο ισχύει και για τις **αναφορές**, δηλαδή, ο μεταγλωττιστής χρησιμοποιεί **στατική σύνδεση** για να επιλέξει την έκδοση της `show()`
- Όπως και με τους δείκτες, η επιλογή γίνεται με βάση τον **τύπο** της αναφοράς και **όχι** τον τύπο του αναφερόμενου αντικειμένου. Άρα, ο κώδικας εμφανίζει:

```
C:A  P:1;
C:B  P:2;
C:B  P:2;
```

Εικονικές Συναρτήσεις

- Ας μιλήσουμε τώρα για τις **εικονικές** συναρτήσεις (virtual functions) και ένα από τα σημαντικότερα χαρακτηριστικά της C++, τον **δυναμικό** πολυμορφισμό
- Αν δηλώσουμε την `show()` σαν εικονική στη βασική κλάση, τα αποτελέσματα είναι διαφορετικά
- Για να την δηλώσουμε εικονική χρησιμοποιούμε τη λέξη `virtual`

```
class Product
{
        ...
        virtual void show() const;
};
class Book : public Product
{
        ...
        virtual void show() const;
};
```

- Όταν μία συνάρτηση σε μία παράγωγη κλάση **επανορίζει** μία εικονική συνάρτηση της βασικής κλάσης με την **ίδια** υπογραφή, ονομάζεται **υποσκέλιση** συνάρτησης (function overriding). Όσον αφορά τον τύπο επιστροφής, γενικά, ο τύπος επιστροφής της εικονικής συνάρτησης στην παράγωγη κλάση θα πρέπει να είναι ο ίδιος με αυτόν στη βασική κλάση

Εικονικές Συναρτήσεις

- Εδώ είναι η ουσία του δυναμικού πολυμορφισμού. Όταν μία εικονική συνάρτηση καλείται μέσω **αναφοράς** ή **δείκτη**, η επιλογή της συνάρτησης που θα κληθεί γίνεται με βάση τον **τύπο** του αντικειμένου στο οποίο **αναφέρεται** η αναφορά ή **δείχνει** ο δείκτης. Άρα, ο προηγούμενος κώδικας εμφανίζει:

```
C:A      P:1;
T:Nice   A:Many;
T:Nice   A:Many;
```

- Δηλαδή, μπορούμε να δηλώσουμε μία αναφορά ή δείκτη σε ένα αντικείμενο βασικής κλάσης, και αν αναφερθεί ή δείξει σε κάποιο παράγωγο αντικείμενο, να κληθεί η συνάρτηση του παράγωγου αντικειμένου, αρκεί αυτή να είναι **εικονική**
- Σκεφτείτε, για παράδειγμα, μία ιεραρχία κλάσεων που να παράγονται από την κλάση `Shape` η οποία να περιέχει μία εικονική συνάρτηση `draw()`. Αφού η `draw()` είναι εικονική μπορούμε να δηλώσουμε μία αναφορά ή δείκτη στη `Shape` και να καλούμε την `draw()` της κάθε κλάσης

Δυναμικός Πολυμορφισμός

- Με τις εικονικές συναρτήσεις η απόφαση για το ποια έκδοση της συνάρτησης θα κληθεί δεν λαμβάνεται κατά τη μεταγλώττιση του προγράμματος, αλλά κατά την **εκτέλεσή** του
- Δηλαδή, με τις εικονικές συναρτήσεις, ο πολυμορφισμός επιτυγχάνεται **δυναμικά** (run-time binding ή run-time polymorphism) με την εκτέλεση του προγράμματος και όχι στατικά με την μεταγλώττιση του προγράμματος
- Αυτό συμβαίνει επειδή ο μεταγλωττιστής **δεν γνωρίζει** όταν μεταγλωττίζει το πρόγραμμα (αναφερόμαστε στη γενική περίπτωση και όχι στον προηγούμενο κώδικα) τον τύπο του αντικειμένου που πρόκειται να αναφερθεί η αναφορά ή να δείχνει ο δείκτης
- Για μη-εικονικές συναρτήσεις η διεύθυνση του κώδικα της συνάρτησης που καλείται είναι γνωστή κατά τη μεταγλώττιση του προγράμματος (στατική σύνδεση), ενώ για εικονικές συναρτήσεις προσδιορίζεται κατά την εκτέλεση του προγράμματος (δυναμική σύνδεση)

Παράδειγμα

- Ποια είναι η έξοδος του παρακάτω προγράμματος;

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

class A
{
public:
        void f() const {cout << "A ";}
        virtual void g() const {cout << "A ";}
        void v() const {cout << "A ";}
};
class B : public A
{
public:
        void f() const {cout << "B ";}
        virtual void g() const {cout << "B ";}
        virtual void v() const {cout << "B ";}
};
int main()
{
        B b;
        A &r = b;

        r.f();
        r.g();
        r.v();
        return 0;
}
```

Παράδειγμα

- Όταν ένας δείκτης ή μία αναφορά χρησιμοποιείται για να κληθεί μία συνάρτηση, **ελέγχουμε** αν η συνάρτηση είναι εικονική ή όχι. Αν δεν είναι, η σύνδεση είναι στατική και κοιτάζουμε τον τύπο του δείκτη ή της αναφοράς. Αν είναι εικονική, η σύνδεση είναι δυναμική και κοιτάζουμε τον τύπο του αντικειμένου που δείχνει ο δείκτης ή αναφέρεται η αναφορά. Ας εφαρμόσουμε αυτόν τον **κανόνα** στο πρόγραμμα
- Αφού η `f()` δεν είναι εικονική, ο μεταγλωττιστής χρησιμοποιεί στατική σύνδεση για να επιλέξει ποια έκδοση θα καλέσει. Επειδή ο τύπος της αναφοράς `r` είναι η κλάση `A`, θα γίνει η κλήση της `A.f()`. Αφού η `g()` είναι εικονική, χρησιμοποιείται δυναμική σύνδεση. Επειδή ο τύπος του αντικειμένου `b` που αναφέρεται η αναφορά `r` είναι η κλάση `B` θα γίνει η κλήση της `B.g()`. Η δήλωση της `v()` ως εικονική στην παράγωγη κλάση και όχι στη βασική δεν έχει κάποια επίδραση. Άρα, η σύνδεση είναι στατική και θα κληθεί η `A.v()`
- Επομένως, το πρόγραμμα εμφανίζει: `A B A`

Παράδειγμα Χρήσης Εικονικών Συναρτήσεων

- Μία πολύ συνηθισμένη χρήση των εικονικών συναρτήσεων σε πραγματικές εφαρμογές είναι σε συναρτήσεις που δέχονται σαν παραμέτρο αναφορά σε αντικείμενο της βασικής κλάσης. Όπως ξέρουμε, μπορούμε να μεταβιβάσουμε ένα αντικείμενο βασικής κλάσης ή ένα αντικείμενο παράγωγης κλάσης και να χρησιμοποιήσουμε την αναφορά για να καλέσουμε μία εικονική συνάρτηση. Η εικονική συνάρτηση που θα κληθεί είναι του αντικειμένου το οποίο θα μεταβιβαστεί. Για παράδειγμα:

```
#include <iostream>

class Shape
{
public:
        virtual void draw() const {std::cout << "Shape ";}
};
class Circle : public Shape
{
public:
        virtual void draw() const {std::cout << "Circle ";}
};
void f(const Shape& s)
{
        s.draw();
}
int main()
{
        Circle c;
        f(c);
        return 0;
}
```

Παράδειγμα Χρήσης Εικονικών Συναρτήσεων

- Όπως ξέρουμε, επιτρέπεται στην `f()` να μεταβιβάσουμε αναφορά ή δείκτη σε αντικείμενο παράγωγης κλάσης. Επειδή η `draw()` είναι εικονική και η `s` αναφέρεται σε `Circle` αντικείμενο θα κληθεί η `Circle::draw()`. Άρα, το πρόγραμμα εμφανίζει `Circle`
- Για να αντιληφθείτε την δύναμη και την ευελιξία του δυναμικού πολυμορφισμού, μέσω των εικονικών συναρτήσεων, θεωρήστε μία πραγματική εφαρμογή που σχεδιάζει σχήματα. Μπορούμε να προσθέσουμε όποιο σχήμα θέλουμε το όποιο να παράγεται από την `Shape`, να ορίσουμε την δική του `draw()` και να καλούμε την `f()`
- Δείτε την ευελιξία τώρα, δεν χρειάζεται να αλλάξουμε την `f()`. Η `f()` δεν έχει καμία ιδέα για το ποιος τύπος αντικειμένου της μεταβιβάζεται πραγματικά, ωστόσο, καλείται η κατάλληλη `draw()`
- Δηλαδή, μπορούμε να επεκτείνουμε το πρόγραμμά μας χωρίς να αλλάξουμε υπάρχοντα κώδικα. Χωρίς αμφιβολία, ο μηχανισμός των εικονικών συναρτήσεων είναι μία πολύ ισχυρή τεχνική που παρέχει μεγάλη προγραμματιστική ευελιξία
- Σημειώστε ότι αν μεταβιβάσουμε ένα δείκτη σε `Circle` αντικείμενο, αντί για αναφορά, θα κληθεί πάλι η `Circle::draw()` με τις απαραίτητες αλλαγές στον κώδικα βέβαια

Παράδειγμα Χρήσης Εικονικών Συναρτήσεων

- Συνεχίζουμε, αν η `draw()` δεν ήταν εικονική το πρόγραμμα θα εμφάνιζε `Shape`
- Και αν αλλάξουμε την `f()` ώστε να δέχεται αντικείμενο, δηλαδή, γράψουμε: `void f(Shape s) {s.draw();}` τι θα εμφανίσει το πρόγραμμα;
- Αφού τώρα το `s` είναι `Shape` αντικείμενο, θα κληθεί η `Shape::draw()` και το πρόγραμμα εμφανίζει `Shape`. Με αυτό το παράδειγμα, να γίνει πάλι σαφές ότι όταν οι εικονικές συναρτήσεις καλούνται από αντικείμενα και όχι μέσω δεικτών ή αναφορών, δεν υπάρχει ειδική μεταχείριση. Δηλαδή, θα κληθεί η συνάρτηση της αντίστοιχης κλάσης
- Σε ένα άλλο παράδειγμα, τι θα εμφανίσει το παρακάτω πρόγραμμα;

```
int main()
{
        Shape sh;
        Circle cir;

        sh.draw();
        cir.draw();
        return 0;
}
```

Παράδειγμα Χρήσης Εικονικών Συναρτήσεων

- Σε κάθε κλήση, καλείται η `draw()` της αντίστοιχης κλάσης. Άρα, το πρόγραμμα εμφανίζει: `Shape Circle`
- Η σύνδεση αντικειμένων (π.χ. `sh`) με κλήσεις συναρτήσεων, είτε αυτές είναι εικονικές (π.χ. `draw()`) είτε όχι, γίνεται **στατικά** κατά την μεταγλώττιση του προγράμματος, ανάλογα με τον **τύπο** του αντικειμένου
- Η διαφορετική συμπεριφορά των εικονικών συναρτήσεων συμβαίνει κατά την **εκτέλεση** του προγράμματος όταν αυτές καλούνται μέσω **δεικτών** ή **αναφορών** σε αντικείμενα παραγώγων κλάσεων και **όχι** όταν καλούνται από τα **ίδια** τα αντικείμενα. Όταν ένα αντικείμενο χρησιμοποιείται για την κλήση συνάρτησης, η κλήση επιλύεται **στατικά** κατά την **μεταγλώττιση**

C++: Από τη Θεωρία στην Εφαρμογή

Κεφάλαιο 21ο

Περισσότερα για Κληρονομικότητα

Αφαιρετικότητα

- Όταν σχεδιάζουμε τις κλάσεις μίας εφαρμογής, υπάρχει περίπτωση κάποιες κλάσεις που φαίνονται ασυσχέτιστες μεταξύ τους να έχουν **κοινά** στοιχεία
- Για παράδειγμα, έστω ότι θέλουμε να αναπτύξουμε μία εφαρμογή που να διαχειρίζεται ορθογώνια τρίγωνα και παραλληλόγραμμα
- Για τη διαχείριση των τριγώνων, μπορούμε να δημιουργήσουμε την κλάση `Triangle`, η οποία να περιέχει δύο πεδία για τις δύο κάθετες πλευρές του τριγώνου. Για τη διαχείριση των παραλληλογράμμων, μπορούμε να δημιουργήσουμε την κλάση `Rectangle` με δύο πεδία για τις δύο κάθετες πλευρές του παραλληλογράμμου
- Επίσης, και οι δύο κλάσεις θα θέλαμε να έχουν μία συνάρτηση που να εμφανίζει τις διαστάσεις του σχήματος (π.χ. `show()`), μία συνάρτηση που να υπολογίζει το εμβαδό του (π.χ. `area()`) και άλλη μία για την περίμετρο (π.χ. `perim()`).
- Μία κομψή προγραμματιστική τεχνική είναι να εφαρμόσουμε τη φιλοσοφία της **αφαιρετικότητας** (abstraction)
- Και ποια είναι αυτή; Ορίζουμε μία κλάση (π.χ. `Shape`), η οποία περιέχει τα **κοινά** πεδία (π.χ. τις δύο κάθετες πλευρές) και συναρτήσεις των δύο κλάσεων (π.χ. `show()`)
- Η `Shape` θα αποτελέσει την βασική κλάση και οι άλλες δύο θα την κληρονομούν

Αφαιρετικότητα

- Τις συναρτήσεις που λειτουργούν **διαφορετικά** σε κάθε κλάση τις δηλώνουμε **εικονικές**
- Για παράδειγμα, θα κάνουμε εικονική την `area()` αφού το εμβαδό του κάθε σχήματος υπολογίζεται διαφορετικά. Αφού η `Shape` δεν αντιπροσωπεύει κάποιο σχήμα δεν χρειάζεται να υλοποιεί την `area()`
- Επιτρέπεται όμως μία κλάση να περιέχει μία συνάρτηση που δεν ορίζεται;
- Ναι, η C++ μας δίνει αυτή τη δυνατότητα με το να τη δηλώσουμε σαν **γνήσια εικονική** (pure virtual). Και πώς την κάνουμε γνήσια εικονική;
- Προσθέτουμε το `= 0` στο τέλος της δήλωσής της

Παράδειγμα

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

class Shape
{
protected:
      float l;
      float h;
public:
      Shape(float len, float hght) {l = len; h =
      hght;}
      void show() {cout << "L:" << l << " H:" << h
      << '\n';}
      virtual void area() const = 0; /* Γνήσια
      εικονική συνάρτηση */
      virtual void perim() const = 0; /* Γνήσια
      εικονική συνάρτηση */
};

class Triangle : public Shape
{
public:
      Triangle(float l, float h) : Shape(l, h) {} /*
      Καλείται ο κατασκευαστής της βασικής κλάσης.
      */
      virtual void area() const {cout << "A:" <<
      l*h/2 << '\n';}
      virtual void perim() const {cout << "P:" <<
      l+h+sqrt(l*l+h*h) << '\n';} /* Για τον
      υπολογισμό της υποτείνουσας εφαρμόζουμε το
      Πυθαγόρειο θεώρημα. */
};
```

```
class Rectangle : public Shape
{
public:
      Rectangle(float l, float h) :
Shape(l, h) {}
      virtual void area() const {cout <<
"A:" << l*h << '\n';}
      virtual void perim() const {cout <<
"P:" << 2*(l+h) << '\n';}
};

int main()
{
      int i;
      vector<Shape*> vec_sh(2);

      vec_sh[0] = new Triangle(1, 2);
      vec_sh[1] = new Rectangle(3, 4);
      for(i = 0; i < 2; i++)
      {
            vec_sh[i]->show();
            vec_sh[i]->area();
            vec_sh[i]->perim();
            delete vec_sh[i];
      }
      return 0;
}
```

Παράδειγμα

- Για ευκολία, τα πεδία `l` και `h` δηλώθηκαν ως προστατευμένα, ώστε οι παράγωγες κλάσεις να έχουν άμεση πρόσβαση σε αυτά
- Για να διαχειριστούμε αντικείμενα και των δύο κλάσεων `Triangle` και `Rectangle` με μία οντότητα και να εφαρμόσουμε **πολυμορφισμό** ένας τρόπος είναι να χρησιμοποιήσουμε ένα διάνυσμα δεικτών στη βασική κλάση `Shape`
- Όπως φαίνεται, μία γνήσια εικονική συνάρτηση **δεν ορίζεται**
- Ο **σκοπός** της είναι να επιτραπεί στις εκδόσεις της συνάρτησης στις παράγωγες κλάσεις να καλούνται με πολυμορφικό τρόπο

Αφηρημένη Κλάση

- Μία κλάση που περιέχει **μία τουλάχιστον** γνήσια εικονική συνάρτηση (π.χ. η Shape) ονομάζεται **αφηρημένη** (abstract class)
- Μία αφηρημένη κλάση μπορεί να περιέχει **μόνο** γνήσιες εικονικές συναρτήσεις
- Σημειώστε ότι **δεν επιτρέπεται** να δημιουργήσουμε **αντικείμενα** μίας αφηρημένης κλάσης. Δηλαδή, αν γράψουμε: `Shape s(1, 2);` ο μεταγλωττιστής θα εμφανίσει μήνυμα **λάθους**
- Η λογική μίας αφηρημένης κλάσης είναι να περιλαμβάνει τις κοινές ιδιότητες των αντίστοιχων οντοτήτων και να χρησιμοποιείται μόνο ως βασική
- Μία γνήσια εικονική συνάρτηση που δεν ορίζεται στην παράγωγη κλάση εξακολουθεί να είναι γνήσια εικονική. Άρα, η παράγωγη κλάση γίνεται επίσης αφηρημένη. Π.χ:

```
class A
{
        ...
        virtual void f() = 0;
        virtual void g() = 0;
};
class B : public A
{
        ...
        virtual void g() {}; // Η f() παραμένει γνήσια εικονική.
};
class C : public B
{
        ...
        virtual void f() {};
};
B b; /* Λάθος, η B είναι αφηρημένη κλάση. Δεν επιτρέπεται η δημιουργία
αντικειμένων της. */
C c; // Σωστό.
```

Παρατηρήσεις

- Το προγραμματιστικό μοντέλο που βασίζεται στην έννοια της αφαιρετικότητας είναι αρκετά συνηθισμένο
- Για να επιστρέψουμε σε αυτό, τα **κοινά** στοιχεία που προκύπτουν από τον σχεδιασμό των κλάσεων τοποθετούνται σε μία **βασική** κλάση, ενώ οι συναρτήσεις που υλοποιούνται διαφορετικά σε κάθε παράγωγη κλάση δηλώνονται ως **γνήσιες** εικονικές
- Με αυτόν τον τρόπο σχεδίασης αποφεύγεται η επανάληψη του ίδιου κώδικα και έτσι διευκολύνεται η συντήρηση και αναβάθμιση του προγράμματος
- Γενικά, οι **αφηρημένες** κλάσεις συνηθίζεται να περιέχουν μόνο γνήσιες εικονικές συναρτήσεις (τα πεδία μεταβλητές δηλώνονται στις παράγωγες κλάσεις) και, επομένως, δεν περιέχουν κατασκευαστή (αφού δεν υπάρχουν πεδία μεταβλητές για να αρχικοποιηθούν, είναι απίθανο να χρειάζεται κατασκευαστής)
- Ουσιαστικά, μία αφηρημένη κλάση αποτελεί ένα είδος **συμβολαίου** για το τι θα πρέπει να παρέχει μία παράγωγη κλάση. Η αφηρημένη κλάση καθορίζει το «**τι πρέπει να υλοποιηθεί**», αφήνοντας την υλοποίηση των εικονικών συναρτήσεων να την αναλάβει ο προγραμματιστής της κάθε παράγωγης κλάσης

Ιδιωτική Κληρονομικότητα

- Εκτός από τη δημόσια κληρονομικότητα που είδαμε στο Κ.20, μία κλάση μπορεί να παραχθεί από μία βασική κλάση με **ιδιωτική** και **προστατευμένη** πρόσβαση
- Με την **ιδιωτική** κληρονομικότητα τα **δημόσια** και **προστατευμένα** μέλη της βασικής κλάσης γίνονται **ιδιωτικά** μέλη της παράγωγης κλάσης
- Αυτό σημαίνει ότι είναι προσπελάσιμα από τις συναρτήσεις της παράγωγης κλάσης, αλλά όχι έξω από την παράγωγη κλάση
- Για να δηλώσουμε την ιδιωτική κληρονομικότητα χρησιμοποιούμε τη λέξη `private` στη δήλωση της παράγωγης κλάσης

Παράδειγμα

```
#include <iostream>

class A
{
protected:
        int r;
        void sub(int i) {v -= i;}
public:
        int v;
        A() : v(1), r(2) {};
        void add(int i) {r += i;}
};

class B : private A
{
public:
        void show();
};

void B::show()
{
        /* Όλες οι παρακάτω ενέργειες
είναι επιτρεπτές. */
        add(3);
        sub(4);
        std::cout << v+r << '\n';
}
```

```
int main()
{
        B b;

        b.show();
        b.add(5); // Μη επιτρεπτή ενέργεια.
        b.sub(6); // Μη επιτρεπτή ενέργεια.
        std::cout << b.v << '\n'; /* Μη
        επιτρεπτή ενέργεια. */
        return 0;
}
```

Προστατευμένη Κληρονομικότητα

- Για να παράξουμε μία κλάση με προστατευμένη πρόσβαση χρησιμοποιούμε τη λέξη `protected` στη δήλωσή της
- Τα **δημόσια** και **προστατευμένα** μέλη της βασικής κλάσης γίνονται **προστατευμένα** μέλη της παράγωγης κλάσης
- Επομένως, όπως και στην ιδιωτική πρόσβαση, είναι προσπελάσιμα από τις συναρτήσεις της παράγωγης κλάσης, αλλά όχι έξω αυτήν

Προστατευμένη Κληρονομικότητα

- Η κύρια διαφορά μεταξύ της ιδιωτικής και προστατευμένης κληρονομικότητας είναι όταν παράγεται μία νέα κλάση από την παράγωγη κλάση. Με την ιδιωτική κληρονομικότητα, η δεύτερη παράγωγη κλάση δεν έχει απευθείας πρόσβαση στα μέλη της βασικής κλάσης γιατί έχουν γίνει ιδιωτικά στην πρώτη παράγωγη. Για παράδειγμα, ας προσθέσουμε την κλάση C στο προηγούμενο πρόγραμμα:

```
class C : private B
{
public:
        void f();
};
void C::f() /* Όλες οι παρακάτω ενέργειες είναι μη επιτρεπτές. */
{
        add(1);
        sub(2);
        cout << v+r << '\n';
}
```

- Επειδή τα μέλη και οι συναρτήσεις της βασικής κλάσης **A** έχουν γίνει ιδιωτικά μέλη της κλάσης **B**, **δεν επιτρέπεται** η απευθείας πρόσβαση σε αυτά από την κλάση C. Αντίθετα, αν η κληρονομικότητα είναι προστατευμένη, η πρόσβαση **επιτρέπεται**

Κανόνες Κληρονομικότητας

- Τώρα που έχουμε μιλήσει για όλα τα είδη κληρονομικότητας, ας συνοψίσουμε τους κανόνες που ισχύουν για την πρόσβαση στα μέλη της βασικής κλάσης. Έστω ότι η κλάση **B** παράγεται από την κλάση **A**:

α) Αν η **A** είναι **ιδιωτική** βασική κλάση, τα δημόσια και προστατευμένα μέλη της **A** γίνονται ιδιωτικά μέλη της **B** και είναι προσβάσιμα μόνο από συναρτήσεις-μέλη και φιλικές συναρτήσεις της **B**

β) Αν η **A** είναι **προστατευμένη** βασική κλάση, τα δημόσια και προστατευμένα μέλη της **A** γίνονται προστατευμένα μέλη της **B** και είναι προσβάσιμα μόνο από συναρτήσεις-μέλη και φιλικές συναρτήσεις της **B**, καθώς και από συναρτήσεις-μέλη και φιλικές συναρτήσεις των κλάσεων που παράγονται από την **B**

γ) Αν η **A** είναι **δημόσια** βασική κλάση, τα δημόσια μέλη της είναι προσβάσιμα από οποιαδήποτε συνάρτηση της **B**. Τα προστατευμένα μέλη της **A** είναι προσβάσιμα από συναρτήσεις-μέλη και φιλικές συναρτήσεις της **B**, καθώς και από συναρτήσεις-μέλη και φιλικές συναρτήσεις των κλάσεων που παράγονται από την **B**

Πολλαπλή Κληρονομικότητα

- Στο Κ.20 μιλήσαμε για την **μονή** κληρονομικότητα, δηλαδή, μία κλάση να παράγεται από μία μόνο βασική κλάση
- Σε αυτήν την ενότητα, θα μιλήσουμε για **πολλαπλή** κληρονομικότητα, δηλαδή, την δυνατότητα μία κλάση να παράγεται από παραπάνω από μία βασικές κλάσεις (multiple inheritance)
- Η πολλαπλή κληρονομικότητα είναι χρήσιμη όταν θέλουμε να **συγχωνεύσουμε** στοιχεία διαφορετικών, συνήθως μη σχετικών κλάσεων, σε μία τρίτη κλάση
- Όμως, επειδή αυξάνει η πολυπλοκότητα του προγράμματος, θα πρέπει να χρησιμοποιείται με σύνεση
- Ας δούμε ένα παράδειγμα και προβλήματα που μπορεί να προκύψουν

Παράδειγμα

```
#include <iostream>

class A
{
public:
        void show() const {...}
        ...
};

class B
{
public:
        void show() const {...}
        ...
};

class C : public A, public B
{
        ...
}

int main()
{
        C c;

        c.show(); /* Ασάφεια, ποια show() θα κληθεί; */
        ...
}
```

Πολλαπλή Κληρονομικότητα

- Η κλάση C παράγεται με δημόσια κληρονομικότητα από τις κλάσεις A και B. Κληρονομεί όλα τα μέλη των A και B κλάσεων
- Όπως φαίνεται στη δήλωση της C, οι κλάσεις διαχωρίζονται με κόμμα και καθορίζουμε το προσδιοριστικό πρόσβασης για κάθε βασική κλάση
- Για την κατασκευή του c, πρώτα κατασκευάζονται τα υπο-αντικείμενα των βασικών του κλάσεων
- Η σειρά κατασκευής είναι, από αριστερά προς τα δεξιά, όπως οι βασικές κλάσεις εμφανίζονται στη δήλωση της κλάσης (π.χ. public A, public B)
- Συγκεκριμένα, όταν δημιουργείται το c καλείται ο εξ'ορισμού κατασκευαστής της C, ο οποίος καλεί τον εξ'ορισμού κατασκευαστή της A, μετά της B, και μετά ολοκληρώνεται η δημιουργία του c
- Δηλαδή, πρώτα κατασκευάζεται το A υπο-αντικείμενο και μετά το B
- Όταν καταστρέφεται το c, οι αποδομητές καλούνται με αντίστροφη σειρά, δηλαδή, πρώτα της C, μετά της B, και τελευταία της A

Πολλαπλή Κληρονομικότητα

- Η υλοποίηση της πολλαπλής κληρονομικότητας μπορεί να είναι πιο δύσκολη και επιρρεπής σε προβλήματα σε σύγκριση με την απλή
- Για παράδειγμα, με την πολλαπλή κληρονομικότητα μπορεί να κληρονομηθούν ίδια ονόματα, και όπως φαίνεται και στον παραπάνω κώδικα, να προκύψουν προβλήματα ασάφειας
- Ποια `show()` θα κληθεί; Της `A` ή της `B`;
- Επειδή ο μεταγλωττιστής δεν ξέρει ποια να καλέσει, θα εμφανίσει μήνυμα λάθους παρόμοιο με «call to show() is ambiguous»
- Για να ξεπεράσουμε το πρόβλημα, μία λύση είναι να χρησιμοποιήσουμε τον τελεστή εμβέλειας `::` και το όνομα της κλάσης που περιέχει τη `show()` που θέλουμε να καλέσουμε. Για παράδειγμα:

```
c.A::show(); // Καλείται η show() της A
```

Κληρονομικότητα και Στατικά Μέλη

- Αν σε μία βασική κλάση ορίζεται ένα `static` μέλος, αυτό δημιουργείται μόνο μία φορά ανεξάρτητα από τον αριθμό των κλάσεων που παράγονται από την βασική. Για κάθε στατικό μέλος ισχύουν οι ίδιοι κανόνες πρόσβασης που ισχύουν και για τα συνηθισμένα μέλη. Για παράδειγμα:

Παράδειγμα

```
#include <iostream>

class A
{
public:
        static inline int v = 10;
};

class B : public A
{
public:
        void f() {v = 20;}
};

class C : public A
{
public:
        void g() {v = 30;}
}

int main()
{
        B b;
        C c;
        b.f();
        c.g();
        std::cout << b.v << ' ' << c.v << '\n'; /* Εναλλακτικά, μπορούμε να
γράψουμε cout << B::v << ' ' << C::v; */
}
```

Κληρονομικότητα και Στατικά Μέλη

- Επειδή υπάρχει μόνο μία υπόσταση του `v` οι συναρτήσεις `f()` και `g()` αλλάζουν διαδοχικά την τιμή του. Έτσι, το πρόγραμμα εμφανίζει `30 30`. Αν δεν δηλώσουμε το `v` ως `static`, το πρόγραμμα θα εμφανίσει `20 30`, αφού η `b.v` είναι διαφορετική μεταβλητή από την `c.v`

C++: Από τη Θεωρία στην Εφαρμογή

Κεφάλαιο 22ο
Εξαιρέσεις

Έλεγχος Σφαλμάτων

- Όταν εκτελείται ένα πρόγραμμα είναι πιθανό να εμφανιστούν σφάλματα που, αν το πρόγραμμα δεν τα χειριστεί με κατάλληλο τρόπο, μπορεί να προκαλέσουν τον βίαιο τερματισμό του
- Η παραδοσιακή τεχνική διαχείρισης σφαλμάτων είναι να υπάρχει διάσπαρτος κώδικας με το κάθε τμήμα του προγράμματος να χειρίζεται τα δικά του σφάλματα
- Για παράδειγμα, όταν χρησιμοποιούμε μία συνάρτηση πρέπει να ελέγχουμε αν εκτελέστηκε με επιτυχία. Αν αποτύχει, ο προγραμματιστής πρέπει να έχει προσθέσει κώδικα που να συλλαμβάνει το λάθος και να το χειρίζεται
- Αν η συνάρτηση επιστρέφει τιμή, ένας συνηθισμένος τρόπος για τον έλεγχο λαθών είναι:

Παράδειγμα

```
k = f();
if(k == ERROR_VALUE_1)
{
    ...
}
else if(k == ERROR_VALUE_2)
{
    ...
}
else if(k == ERROR_VALUE_N)
{
    ...
}
else // Δεν συνέβη λάθος.
{
    ...
}
```

Σημειώστε ότι αν η συνάρτηση δεν επιστρέφει τιμή, μπορούμε να χρησιμοποιήσουμε ένα όρισμα δείκτη ή αναφορά για να επιστραφεί το αποτέλεσμα της εκτέλεσης της συνάρτησης

Εξαιρέσεις

- Η C++ παρέχει τον μηχανισμό των **εξαιρέσεων** (exceptions) ως μία εναλλακτική τεχνική αντιμετώπισης προβλημάτων
- Οι εξαιρέσεις αναφέρουν λάθη που συμβαίνουν κατά την **εκτέλεση** του προγράμματος
- Ειδικότερα, με τις εξαιρέσεις επιτρέπεται ο **διαχωρισμός** της **ανίχνευσης** των σφαλμάτων από τον **χειρισμό** τους
- Δηλαδή, το λάθος συμβαίνει σε ένα σημείο του προγράμματος και η **διαχείριση** σε κάποιο άλλο σημείο
- Η **φιλοσοφία** του μηχανισμού είναι να υπάρχουν συγκεκριμένα σημεία στο πρόγραμμα για την διαχείριση των σφαλμάτων και όχι διάσπαρτος κώδικας που να τα διαχειρίζεται
- Έτσι, ο κώδικας γίνεται πιο απλός και η αντιμετώπιση προβλημάτων διαβάζεται και ελέγχεται πιο εύκολα
- Ας δούμε την σύνταξη για ένα συνηθισμένο παράδειγμα για τη δημιουργία και σύλληψη εξαιρέσεων:

Παράδειγμα

```
f()
{
    try
    {
        g();
    }
    try(type exc_1)
    {
        ...
    }
    catch(type exc_2)
    {
        ...
    }
    catch(...) // Συλλαμβάνει κάθε τύπο εξαίρεσης.
    {
        ...
    }
}

g()
{
    ...
    if(error_occurs)
        throw exception; // Η g() δημιουργεί μία εξαίρεση.
    ...
}
```

Εξαιρέσεις

- Η ιδέα είναι ότι αν συμβεί κάποιο λάθος σε μία συνάρτηση (π.χ. `g()`) και η συνάρτηση δεν μπορεί να το διαχειριστεί, μπορεί να χρησιμοποιήσει την εντολή `throw` για να **δημιουργήσει** μία εξαίρεση που να ενημερώνει την συνάρτηση που την κάλεσε για το λάθος
- Για να μπορεί να **συλλάβει** (*catch*) την εξαίρεση, η κλήση της συνάρτησης που μπορεί να δημιουργήσει εξαιρέσεις πρέπει να τοποθετηθεί σε ένα τμήμα που αρχίζει με την εντολή `try`
- Ο κώδικας που χειρίζεται τις εξαιρέσεις τοποθετείται μετά την εντολή `try` σε ένα ή περισσότερα τμήματα σύλληψης που αρχίζουν με την εντολή `catch`
- Σημειώστε ότι δεν επιτρέπεται να εισάγεται κώδικας μεταξύ του `try` τμήματος και του πρώτου `catch` τμήματος ή μεταξύ διαδοχικών `catch` τμημάτων, το πρόγραμμα δεν θα μεταγλωττιστεί
- Κάθε `catch` τμήμα καθορίζει τον τύπο της εξαίρεσης που μπορεί να συλλάβει. Ο τύπος δηλώνεται μετά την `catch` λέξη μέσα στις παρενθέσεις και μπορεί να είναι ένας βασικός τύπος αλλά και ένας τύπος που έχει οριστεί από τον προγραμματιστή

Παρατηρήσεις

- Η **βασική** έννοια των εξαιρέσεων είναι όταν συμβαίνει ένα «εξαιρετικό» γεγονός σε ένα τμήμα του προγράμματος (π.χ. στη `g()`), αυτό το τμήμα να μπορεί να **πληροφορήσει** ένα άλλο (π.χ. την `f()`) για αυτό το «**εξαιρετικό**» γεγονός και να του μεταφέρει πληροφορία σχετικά με αυτό
- Ο προγραμματιστής πρέπει να **καθορίσει** τι είναι αυτό το «εξαιρετικό». Για παράδειγμα, μία απόπειρα διαίρεσης με το μηδέν είναι κάτι «εξαιρετικό»
- Σημειώστε ότι το «εξαιρετικό» δεν είναι απαραίτητο να είναι πάντα καταστροφικό, ο προγραμματιστής θα πρέπει να καθορίσει την σοβαρότητά του, μπορεί να είναι απλά η αδυναμία εκτέλεσης κάποιας εργασίας
- Ουσιαστικά, η δημιουργία εξαίρεσης αποτελεί έναν τρόπο ώστε το τμήμα του προγράμματος το οποίο δεν μπορεί να διαχειριστεί τοπικά το «εξαιρετικό» γεγονός (π.χ. `g()`) να το **προωθήσει** σε ένα ανώτερο επίπεδο (π.χ. `f()`), το οποίο ίσως να μπορεί να το διαχειριστεί

Σύλληψη Εξαιρέσεων

- Η εντολή `throw` δημιουργεί μία εξαίρεση και το όρισμά της καθορίζει τον τύπο της εξαίρεσης
- Η `throw` τερματίζει την εκτέλεση της τρέχουσας συνάρτησης (π.χ. `g()`) και η εκτέλεση του προγράμματος μεταβαίνει στην καλούσα συνάρτηση (π.χ. `f()`)
- Εκεί, το πρόγραμμα ελέγχει διαδοχικά τις `catch` εντολές για να δει αν υπάρχει κάποια που να ταιριάζει με τον τύπο της εξαίρεσης. Για παράδειγμα, η εντολή `throw 20;` δημιουργεί μία εξαίρεση τύπου `int` και μεταβιβάζει την τιμή `20`
- Αν υπάρχει μία `catch` εντολή για εξαιρέσεις ακέραιου τύπου, π.χ. `catch(int a)`, θα συλλάβει τη συγκεκριμένη εξαίρεση, η τιμή του `a` θα γίνει `20` και θα εκτελεστεί ο κώδικας του τμήματος
- Σημειώστε ότι αν γράψουμε `catch(double a)`, η εξαίρεση δεν θα συλληφθεί γιατί ο μεταγλωττιστής δεν πραγματοποιεί τις συνηθισμένες αριθμητικές μετατροπές. Οι μετατροπές που υποστηρίζονται είναι περιορισμένες

Σύλληψη Εξαιρέσεων

- Το `catch` τμήμα μπορεί να περιέχει κώδικα που να διορθώνει το πρόβλημα, αν αυτό μπορεί να διορθωθεί
- Αν όχι, ο προγραμματιστής μπορεί να αποφασίσει να προσθέσει κώδικα που να τερματίζει το πρόγραμμα. Τα υπόλοιπα τμήματα δεν ελέγχονται
- Σημειώστε ότι δεν είναι υποχρεωτικό να δηλώσουμε ονόματα παραμέτρων, π.χ. `catch(int)`
- Συνήθως, το `throw` όρισμα που μεταβιβάζεται στο `catch` τμήμα παρέχει πληροφορία που σχετίζεται με την εξαίρεση, οπότε η `catch` παράμετρος χρησιμεύει για την αποθήκευση αυτής της πληροφορίας

Σύλληψη Εξαιρέσεων

- Το τμήμα σύλληψης `catch(...)` είναι **προαιρετικό** και **συλλαμβάνει** όλους τους τύπους των εξαιρέσεων
- Αν υπάρχει και η εξαίρεση δεν συλληφθεί από κάποιο προηγούμενο `catch` τμήμα θα συλληφθεί από αυτό. Μοιάζει κάπως με την `default` περίπτωση στην εντολή `switch`
- Σημειώστε ότι θα πρέπει να τοποθετηθεί στο **τέλος** των `catch` τμημάτων. Αν τοποθετηθεί σε άλλη θέση, επειδή συλλαμβάνει όλες τις εξαιρέσεις, οι `catch` εντολές που μπορεί να υπάρχουν μετά από αυτό δεν θα ελεγχθούν
- Αν λείπει, και η εξαίρεση δεν συλληφθεί από **κανένα** τμήμα, τότε η εξαίρεση **προωθείται** προς την `main()`, μέχρι να βρεθεί κάποιο τμήμα σύλληψης που να ταιριάζει με τον τύπο της. Αν δεν βρεθεί κάποιο τέτοιο τμήμα, το πρόγραμμα **τερματίζει**
- Αν δεν δημιουργηθεί κάποια εξαίρεση στο `try` τμήμα, οι `catch` εντολές **παρακάμπτονται** και η εκτέλεση του προγράμματος συνεχίζει με την εντολή μετά το τελευταίο `catch` τμήμα
- Ας δούμε ένα παράδειγμα. Το παρακάτω πρόγραμμα διαβάζει ένα συνθηματικό και ελέγχει αν είναι έγκυρο

Παράδειγμα

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

int check_pswd(const string& str);

int main()
{
	string pswd;
	while(1)
	{
		try
		{
			cout << "Enter password: ";
			cin >> pswd;
			if(pswd == "end")
				break;
			check_pswd(pswd);
		}
		catch(const char *msg)
		{
			cout << msg;
			continue; /* Αν έλειπε, θα εκτελούνταν η εντολή μετά το τελευταίο catch τμήμα. */
		}
		catch(int code)
		{
			if(code == 10)
				cout << "Error: Password must contain three digits at least\n";
			else if(code == 20)
				cout << "Error: Password must contain one special character at least\n";
			continue;
		}
		catch(...)
		{
			cout << "Error: Another exception\n";
			continue;
		}
		cout << "Password " << pswd << " is accepted\n";
	}
	return 0;
}
```

```
int check_pswd(const string& str)
{
	bool found;
	int i, len, dig;

	len = str.size();
	if(len < 6)
		throw "Error: Short length\n";

	dig = 0;
	found = false;
	for(i = 0; i < len; i++)
	{
	if(str[i] >= '0' && str[i] <= '9')
		dig++;
	if(str[i] == '!' || str[i] == '$' || str[i] == '#')
		found = true;
	}
	if(dig < 3)
		throw 10;
	if(found == false)
		throw 20;
	return 0;
}
```

Παράδειγμα

- Αρχικά, η `check_pswd()` ελέγχει το μήκος του συνθηματικού. Αν είναι μικρότερο από έξι χαρακτήρες η εντολή `throw` δημιουργεί μία εξαίρεση και μεταβιβάζει ένα κυριολεκτικό αλφαριθμητικό στην καλούσα συνάρτηση, δηλαδή, στη `main()`
- Άρα, ο τύπος της εξαίρεσης είναι `const char*`. Η `throw` τερματίζει την `check_pswd()` και η εκτέλεση του προγράμματος μεταβαίνει στη `main()`
- Γενικά, όταν δημιουργείται μία εξαίρεση, η στοίβα με τις κλήσεις των συναρτήσεων ξετυλίγεται ώστε να μεταφερθεί ο έλεγχος του προγράμματος στη συνάρτηση που μπορεί να διαχειριστεί αυτόν τον τύπο της εξαίρεσης
- Αφού η κλήση της `check_pswd()` έγινε μέσα από `try` τμήμα, το πρόγραμμα μπορεί να συλλάβει τις εξαιρέσεις που δημιουργεί
- Αν είχε γίνει έξω από `try` τμήμα, οι εξαιρέσεις δεν θα μπορούσαν να συλληφθούν

Παράδειγμα

- Τώρα, το πρόγραμμα ελέγχει αν υπάρχει κάποια `catch` εντολή που να ταιριάζει με τον τύπο της εξαίρεσης. Έτσι, το τμήμα με τύπο `const char*` συλλαμβάνει την εξαίρεση και εκτελείται ο κώδικάς του
- Το αλφαριθμητικό "Error: Short length\n" μεταβιβάζεται στην παράμετρο `msg` και αυτό εμφανίζεται στην οθόνη
- Με την εντολή `continue` το πρόγραμμα συνεχίζει με την επόμενη επανάληψη του βρόχου και ο χρήστης εισάγει νέο συνθηματικό. Θα μπορούσαμε στη θέση της `continue` να έχουμε `break` ή `return` για τον τερματισμό του προγράμματος ή κάποια άλλη προσέγγιση, όπως ότι αν ο χρήστης εισάγει τρεις φορές ένα μη έγκυρο συνθηματικό το πρόγραμμα να τερματίζει
- Σε κάθε τμήμα σύλληψης, ο προγραμματιστής **αποφασίζει** για τον τρόπο που θα χειριστεί την εξαίρεση
- Σημειώστε ότι αν δεν υπήρχε το `const char*` τμήμα, η εξαίρεση θα συλλαμβανόταν από το `catch(...)` τμήμα και θα εκτελούνταν ο κώδικάς του. Αν δεν υπήρχε κανένα από τα δύο, η εκτέλεση του προγράμματος θα τερματιζόταν

Παράδειγμα

- Αν το μήκος του συνθηματικού είναι αποδεκτό, η `check_pswd()` ελέγχει αν περιέχει λιγότερα από τρία ψηφία
- Αν ναι, δημιουργεί μία εξαίρεση τύπου `int`. Τότε, η εκτέλεση της `check_pswd()` τερματίζεται και το `int` τμήμα στη `main()` συλλαμβάνει την εξαίρεση. Η τιμή του `code` γίνεται ίση με `10` και το πρόγραμμα εμφανίζει το αντίστοιχο μήνυμα
- Αν ο αριθμός των ψηφίων είναι έγκυρος, η `check_pswd()` ελέγχει αν το συνθηματικό περιέχει κάποιον από τους χαρακτήρες `!`, `$` ή `#`
- Αν όχι, δημιουργεί μία εξαίρεση τύπου `int`, όπου στο τμήμα σύλληψης το `code` γίνεται ίσο με `20` και το πρόγραμμα εμφανίζει το αντίστοιχο μήνυμα
- Όπως και πριν, η εντολή `continue` προκαλεί την επόμενη επανάληψη του βρόχου. Σημειώστε ότι, σε αντίθεση με την `return` η οποία μπορεί να επιστρέψει έναν τύπο δεδομένων, η `throw` μπορεί να επιστρέψει διαφορετικούς τύπους

Παράδειγμα

- Αν το συνθηματικό είναι αποδεκτό, η `check_pswd()` δεν δημιουργεί κάποια εξαίρεση και επιστρέφει την τιμή `0`
- Ο λόγος που επιλέχθηκε η συνάρτηση να επιστρέφει τιμή είναι για να δείτε ότι μία συνάρτηση μπορεί να δημιουργεί εξαιρέσεις αλλά και να επιστρέφει τιμή, η οποία και αυτή να δηλώνει αν εκτελέστηκε σωστά ή όχι
- Στη συνέχεια, το πρόγραμμα παρακάμπτει τα `catch` τμήματα και εμφανίζει το αποδεκτό συνθηματικό
- Το πρόγραμμα εκτελείται μέχρι ο χρήστης να εισάγει το `end`. Τότε, η εντολή `break` θα τερματίσει την εκτέλεση του βρόχου
- Τέλος, με το συγκεκριμένο παράδειγμα βλέπουμε ότι οι `break` και `continue` εντολές μέσα σε ένα `try-catch` τμήμα λειτουργούν με τον συνηθισμένο τρόπο

Εξαιρέσεις με Τύπο Κλάση

- Εκτός από τη δημιουργία εξαιρέσεων με βασικούς τύπους δεδομένων (π.χ. `int`), μπορεί να δημιουργηθούν εξαιρέσεις που ο τύπος τους να είναι **κλάση**
- Γενικά, αυτή είναι η συνηθέστερη επιλογή
- Ένα πλεονέκτημα αυτής της επιλογής είναι ότι μπορούμε να χρησιμοποιήσουμε διαφορετικά αντικείμενα για να ξεχωρίζουν τα διαφορετικά είδη προβλημάτων που δημιουργούν εξαιρέσεις
- Ένα άλλο πλεονέκτημα είναι ότι ένα αντικείμενο μπορεί να περιέχει όση πληροφορία χρειάζεται για να εντοπιστούν οι αιτίες που προκάλεσαν την εξαίρεση
- Η πρότυπη βιβλιοθήκη παρέχει αρκετές κλάσεις εξαιρέσεων τις οποίες μπορούμε να χρησιμοποιήσουμε στα προγράμματά μας Βρίσκονται στον χώρο ονομάτων `std` και όλες προέρχονται από την κλάση `exception`
- Για παράδειγμα, ας αλλάξουμε το προηγούμενο πρόγραμμα ώστε η εξαίρεση να μεταβιβάζει ένα αντικείμενο:

Παράδειγμα

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

class Err_Rpt
{
private:
      int code;
      string msg;
public:
      Err_Rpt(): code(0), msg("") {}
      Err_Rpt(int c, const char *m) : code(c), msg(m) {}
      void show() const {cout << "C:" << code << ' ' << msg;}
};
int check_pswd(const string& str);

int main()
{
      string pswd;
      while(1)
      {
                  try
                  {
                              cout << "Enter password: ";
                              cin >> pswd;
                              check_pswd(pswd);
                              break;
                  }
                  catch(const Err_Rpt& err)
                  {
                              err.show();
                  }
                  catch(int code)
                  {
                  cout << "C:" << code << ' ' << "Error:
      Password must contain one special character at least\n";
                  }
                  catch(...)
                  {
                        cout << "Error: Another exception\n";
                  }
      }
      cout << "Password " << pswd << " is accepted\n";
      return 0;
}
```

```
int check_pswd(const string& str)
{
            bool found;
            int i, len, dig;

            len = str.size();
            if(len < 6)
                  throw Err_Rpt(5, "Error: Short
length\n"); /* Δημιουργία εξαίρεσης με τύπο
αντικείμενο της κλάσης Err_Rpt. */

            dig = 0;
            found = false;
            for(i = 0; i < len; i++)
            {
            if(str[i] >= '0' && str[i] <= '9')
                        dig++;
            if(str[i] == '!' || str[i] == '$' ||
str[i] == '#')
                        found = true;
            }
            if(dig < 3)
                        throw Err_Rpt(10, "Error:
Password must contain three digits at
least\n");
            if(found == false)
                        throw 20;
            return 0;
}
```

Παράδειγμα

- Αν ο χρήστης εισάγει ένα έγκυρο συνθηματικό, η εντολή `break` τερματίζει τον βρόχο και το πρόγραμμα το εμφανίζει
- Η `check_pswd()` δημιουργεί μία εξαίρεση με τύπο `int` και άλλες δύο με τύπο **αντικείμενο** της κλάσης `Err_Rpt`
- Κάθε αντικείμενο αρχικοποιείται με τιμές που υποδεικνύουν τον τύπο του λάθους
- Αυτές οι εξαιρέσεις συλλαμβάνονται από το αντίστοιχο `catch` τμήμα, το οποίο καλεί την `show()` του αντικειμένου που μεταβιβάζεται για να εμφανίσει την πληροφορία που περιέχεται στα πεδία του
- Σημειώστε ότι στη θέση της `int` εξαίρεσης θα μπορούσαμε να δημιουργήσουμε και άλλη μία εξαίρεση με `Err_Rpt` τύπο. Για παράδειγμα, μπορούμε να γράψουμε `throw Err_Rpt();` όπου το αντικείμενο που μεταβιβάζεται δεν περιέχει πληροφορία
- Απλά, αυτό το πρόγραμμα αποτελεί ένα παράδειγμα που **συνδυάζει** τη δημιουργία και σύλληψη βασικών τύπων και τύπων που ορίζονται από τον χρήστη

Εξαιρέσεις και Κληρονομικότητα

- Για παράδειγμα, στο παρακάτω πρόγραμμα έχουμε τρεις κλάσεις εξαιρέσεων που σχετίζονται μεταξύ τους με μία σχέση κληρονομικότητας:

```
#include <iostream>

class Err1
{
public:
	virtual void show() const {std::cout << "Err1\n";}
};

class Err2 : public Err1
{
public:
	virtual void show() const {std::cout << "Err2\n";}
};

class Err3 : public Err2
{
public:
	virtual void show() const {std::cout << "Err3\n";}
};

void f()
{
	throw Err3();
}

int main()
{
	try
	{
		f();
	}
	catch(const Err1& err)
	{
		err.show();
	}
	return 0;
}
```

Εξαιρέσεις και Κληρονομικότητα

- Επειδή η παράμετρος στο `catch` τμήμα είναι αναφορά στη βασική κλάση, ταιριάζει με οποιαδήποτε εξαίρεση παράγωγης κλάσης
- Επειδή η `show()` είναι εικονική, η επιλογή της `show()` που θα εκτελεστεί γίνεται με βάση τον τύπο του αντικειμένου στον οποίο αναφέρεται η `err`. Έτσι, το πρόγραμμα εμφανίζει `Err3`
- Αν δεν χρησιμοποιούσαμε αναφορά θα καλούνταν πάντα η `show()` της `Err1` και το πρόγραμμα θα εμφάνιζε `Err1`, ανεξάρτητα από το αντικείμενο-όρισμα της `throw`
- Όταν δημιουργούνται εξαιρέσεις, η καλύτερη πρακτική είναι η μεταβίβαση αντικειμένου. Ο γενικός κανόνας είναι να μεταβιβάζεται το αντικείμενο κατ'αξία και να συλλαμβάνεται μέσω αναφοράς

Εξαιρέσεις και Κληρονομικότητα

- Αν θέλουμε να χειριστούμε με διαφορετικό τρόπο την κάθε εξαίρεση, προσθέτουμε ξεχωριστές **catch** εντολές για τον κάθε τύπο. **Προσοχή** όμως στη σειρά. Η σειρά των **catch** τμημάτων θα πρέπει να είναι η **αντίστροφη** της παραγωγής των κλάσεων, δηλαδή, από την τελευταία παράγωγη προς την βασική. Για παράδειγμα:

```
int main()
{
        int i;

        try
        {
                f();
        }
        catch(const Err3& err)
        {
                i = 3;
        }
        catch(const Err2& err)
        {
                i = 2;
        }
        catch(const Err1& err)
        {
                i = 1;
        }
        return 0;
}
```

- Αν το **Err1&** τμήμα σύλληψης ήταν πρώτο, θα συλλάμβανε όλες τις **Err1**, **Err2**, και **Err3** εξαιρέσεις και η τιμή του **i** θα ήταν πάντα **1**
- Σε μία ιεραρχική σχέση κληρονομικότητας με κλάσεις εξαιρέσεων, το πρώτο τμήμα σύλληψης **πρέπει** να αναφέρεται στην τελευταία παράγωγη κλάση και το τελευταίο τμήμα στη βασική κλάση

Προώθηση Εξαιρέσεων

- Το σύστημα διαχείρισης των εξαιρέσεων μπορεί να είναι πολυεπίπεδο
- Δηλαδή, το κάθε επίπεδο να διαχειρίζεται όσα προβλήματα μπορεί και τα υπόλοιπα να τα αφήνει για τα ανώτερα επίπεδα
- Έτσι, αν μία εξαίρεση δεν συλληφθεί από την τρέχουσα συνάρτηση, η εξαίρεση **προωθείται** στη συνάρτηση που την κάλεσε, και από εκεί σε αυτήν που την κάλεσε, σε όλη τη διαδρομή μέχρι την `main()`, μέχρι να βρεθεί κάποιο τμήμα σύλληψης που να συλλάβει την εξαίρεση
- Αν η `main()` δεν συλλάβει την εξαίρεση, το πρόγραμμα, εξ'ορισμού θα τερματιστεί
- Αυτή η διαδικασία ονομάζεται ξεδίπλωμα στοίβας (stack unwinding). Για παράδειγμα, ας δούμε τον παρακάτω κώδικα:

```
int main()
{
 ...
 try
 {
    f();
 }
 catch(int)
 {
    ...
 }
}
```

```
f()
{
 ...
 try
 {
        g();
 }
 catch(double)
 {
        ...
 }
}
```

```
g()
{
 ...
 try
 {
        h();
 }
 catch(const char*)
 {
        ...
 }
}
```

```
h()
{
   ...
throw 20;
}
```

Προώθηση Εξαιρέσεων

- Η εξαίρεση στην `h()` προωθείται σε υψηλότερα επίπεδα και τελικά συλλαμβάνεται από το τμήμα σύλληψης της `main()`, το οποίο μπορεί να χειριστεί εξαιρέσεις ακεραίου τύπου. Αν αντί για `throw 20;` γράψουμε `throw 1.2;` αυτή η εξαίρεση θα συλληφθεί από το τμήμα σύλληψης της `f()`, ενώ αν γράψουμε `throw "Test";` θα συλληφθεί από το τμήμα σύλληψης της `g()`. Αν έχει δηλωθεί κάποια κλάση `A` και `a` είναι αντικείμενό της και γράψουμε `throw a;` αυτή η εξαίρεση θα προωθηθεί μέχρι την `main()` και επειδή δεν υπάρχει `catch` τμήμα για τον τύπο `A` το πρόγραμμα θα τερματιστεί
- Αυτό το παράδειγμα δείχνει επίσης μία πολύ σημαντική ιδιότητα της `throw`. Ενώ η `return` μεταφέρει την εκτέλεση του προγράμματος στην πρώτη εντολή μετά την κλήση της συνάρτησης, η `throw` μεταφέρει την εκτέλεση σε όλη τη διαδρομή μέχρι να συναντήσει το πρώτο `catch` τμήμα που μπορεί να συλλάβει την εξαίρεση

Επανακατάθεση Εξαίρεσης

- Όταν ένα `catch` τμήμα συλλαμβάνει μία εξαίρεση είναι πιθανό να μη μπορεί να την διαχειριστεί ή να αποφασίσει ότι είναι καλύτερα να την διαχειριστεί ένα ανώτερο επίπεδο
- Σε αυτή την περίπτωση μπορούμε να **επανακαταθέσουμε** την εξαίρεση, ώστε να προωθηθεί σε **ανώτερο** επίπεδο, πιο σχετικό με τον τύπο της εξαίρεσης, και να αναλάβει αυτό την διαχείρισή της
- Για να κάνουμε κάτι τέτοιο γράφουμε **μόνο** το όνομα `throw`, χωρίς κάποιο όρισμα
- Η `throw` πρέπει να βρίσκεται μέσα σε ένα `catch` τμήμα ή μέσα σε μία συνάρτηση που να καλείται (άμεσα ή έμμεσα σε μία αλυσίδα κλήσεων) από ένα `catch` τμήμα. Αλλιώς, η κλήση της `throw` προκαλεί τον **τερματισμό** του προγράμματος
- Η εξαίρεση που επανακατατίθεται είναι η **αρχική** εξαίρεση που είχε συλληφθεί. Για παράδειγμα, στο επόμενο πρόγραμμα, το αρχικό όρισμα της `throw` μεταβιβάζεται σε όλη τη διαδρομή μέχρι το τμήμα σύλληψης που τελικά θα διαχειριστεί την εξαίρεση:

Παράδειγμα

```
#include <iostream>

void f();
void g();

int main()
{
	try
	{
		f();
	}
	catch(int a)
	{
		std::cout << "int exception is caught: " << a << '\n';
	}
	return 0;
}
void f()
{
	try
	{
		g();
	}
	catch(int) /* Αφού το τμήμα σύλληψης δεν χρησιμοποιεί το όρισμα δεν δηλώνω αντίστοιχη
	μεταβλητή. */
	{
		std::cout << "int exception is rethrown\n";
		throw; // Επαναχατάθεση της εξαίρεσης.
	}
	std::cout << "f() terminates\n"; /* Αφού η throw επαναχαταθέτει την εξαίρεση αυτή η εντολή
	δεν θα εκτελεστεί. */
}

void g()
{
	 throw 10;
}
```

Παράδειγμα

- Η int εξαίρεση που δημιουργεί η g() συλλαμβάνεται από το catch τμήμα της f(). Η f() την επανακαταθέτει και αυτή συλλαμβάνεται στη main(). Στη main() μεταβιβάζεται και το αρχικό όρισμα της throw. Άρα, το πρόγραμμα εμφανίζει:

```
int exception is rethrown
int exception is caught: 10
```

- Αυτό το πρόγραμμα αποτελεί ένα παράδειγμα ένθετων try τμημάτων. Όπως είδαμε και στο προηγούμενο παράδειγμα, ένα try τμήμα, μαζί με catch τμήματά του, μπορεί να τοποθετηθεί μέσα σε ένα άλλο try τμήμα
- Σε αυτό το παράδειγμα, το εξωτερικό τμήμα είναι αυτό στο main(), ενώ το εσωτερικό τμήμα είναι αυτό στην f()
- Κάθε try τμήμα συσχετίζεται με το δικό του σύνολο catch τμημάτων, το οποίο χειρίζεται τις εξαιρέσεις που ενδέχεται να δημιουργηθούν μέσα σε αυτό
- Αν κανένα δεν ταιριάζει με τον τύπο της εξαίρεσης, θα την χειριστούν τα catch τμήματα του εξωτερικού try τμήματος
- Για παράδειγμα, αν στην f() αλλάξουμε τον τύπο στο catch τμήμα από int σε double, τότε η εξαίρεση θα προωθηθεί στο εξωτερικό try τμήμα στη main(). Εκεί, το catch τμήμα για τον τύπο int θα συλλάβει την εξαίρεση

C++: Από τη Θεωρία στην Εφαρμογή

Κεφάλαιο 23ο
Πρότυπες Κλάσεις

Πρότυπες Κλάσεις

- Όπως οι πρότυπες συναρτήσεις, έτσι και οι **πρότυπες κλάσεις** παρέχουν τη δυνατότητα προγραμματισμού με **γενικό** τρόπο
- Με τα πρότυπα κλάσεων (class templates) μπορούμε να ορίσουμε κλάσεις γενικής χρήσης, δηλαδή, κλάσεις στις οποίες μπορούμε να μεταβιβάσουμε **διαφορετικούς τύπους δεδομένων**, χωρίς να χρειαστεί να ξαναγράψουμε τον κώδικά τους
- Για παράδειγμα, θεωρήστε ότι έχουμε υλοποιήσει μία στοίβα (π.χ. `Lifo`) όπου ο τύπος των στοιχείων που αποθηκεύονται σε αυτήν είναι `int`. Αν θέλουμε να αποθηκεύσουμε κάποιον άλλο τύπο (π.χ. `float`) πρέπει να αλλάξουμε όλες τις εμφανίσεις του τύπου `int` με τον νέο τύπο και να μεταγλωττίσουμε πάλι το πρόγραμμα
- Δηλαδή, δεν μπορούμε να έχουμε στο ίδιο πρόγραμμα δύο `Lifo` στοίβες, η μία με `int` στοιχεία και η άλλη με `float` στοιχεία. Θα πρέπει να δημιουργήσουμε μία νέα στοίβα (π.χ. `Lifo_Float`), να αντιγράψουμε τον κώδικα της υπάρχουσας και να αλλάξουμε τον τύπο από `int` σε `float`
- Αν θέλουμε να δημιουργήσουμε μία νέα στοίβα με κάποιον άλλο τύπο δεδομένων, θα πρέπει να επαναλάβουμε την παραπάνω διαδικασία

Πρότυπες Κλάσεις

- Για να αντιμετωπίσει η C++ αυτή την αδυναμία εισάγει τις πρότυπες κλάσεις, ώστε να μην χρειαστεί να αντιγράφουμε τον ίδιο κώδικα
- Ένα παράδειγμα πρότυπης κλάσης είναι η κλάση `vector`
- Μία πρότυπη κλάση ορίζεται με **γενικό** τρόπο ώστε να μην εξαρτάται η λειτουργία της από κάποιο συγκεκριμένο τύπο
- Ο τύπος των στοιχείων που θα χειρίζεται το κάθε αντικείμενο της κλάσης μεταβιβάζεται σαν **όρισμα** στις γενικές παραμέτρους της κλάσης

Παράδειγμα

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

template <typename T> class Lifo // Πρότυπη κλάση.
{
private:
      vector<T> vec_n;
public:
      void push(const T& t);
      void pop(T& t);
      void show_all() const;
      int get_nodes() const;
};

template <typename T> void Lifo<T>::push(const T& t)
{
      vec_n.push_back(t);
}

template <typename T> void Lifo<T>::pop(T& t)
{
      int size = vec_n.size();
      if(size > 0)
      {
                  t = vec_n[size-1];
                  vec_n.erase(vec_n.end()-1);
      }
      else
                  cout << "Stack is empty\n";
}

template <typename T> void Lifo<T>::show_all() const
{
      int i, size = vec_n.size();
      if(size > 0)
      {
                  for(i = 0; i < size; i++)
                              cout << vec_n[i] << '\n';
      }
      else
                  cout << "Stack is empty\n";
}
```

```
template <typename T> int
Lifo<T>::get_nodes()  const
{
            return vec_n.size();
}

int main()
{
            int i;
            string s;

            Lifo<string> lifo_str;
            Lifo<int> lifo_int;
            lifo_str.push("One");
            lifo_str.push("Two");
            lifo_str.push("Three");
            lifo_str.pop(s);
            cout << s << '\n';
            lifo_str.show_all();

            lifo_int.push(1);
            lifo_int.push(2);
            lifo_int.push(3);
            lifo_int.pop(i);
            cout << i << '\n';
            lifo_int.show_all();
            return 0;
}
```

Πρότυπες Κλάσεις

- Όπως και με τις απλές πρότυπες συναρτήσεις, για να δηλώσουμε μία πρότυπη κλάση χρησιμοποιούμε το πρόθεμα `template <typename T>`
- Το ίδιο ισχύει όταν ορίζουμε συναρτήσεις έξω από την κλάση
- Το όρισμα `T` το οποίο θα χρησιμοποιηθεί στη δήλωση της κλάσης αντιστοιχεί σε ένα όνομα τύπου (π.χ. `int`)
- Ο τύπος `T` μπορεί να χρησιμοποιηθεί όπως ένας **συνηθισμένος** τύπος δεδομένων
- Για παράδειγμα, μπορούμε να τον χρησιμοποιήσουμε για να δηλώσουμε τον τύπο των μελών της κλάσης ή να δηλώσουμε τον τύπο επιστροφής και των παραμέτρων των συναρτήσεων μελών, είτε από μόνον του είτε συνδυασμένο σε τύπους όπως `T&` ή `T*`
- Όσον αφορά το όνομα του ορίσματος μπορείτε να επιλέξετε όποιο επιθυμείτε σύμφωνα με τους κανόνες ονοματοδοσίας. Μικρά ονόματα που αρχίζουν με το γράμμα `T` αποτελούν συνηθισμένες επιλογές
- Το πρόγραμμα εμφανίζει: `Three One Two 3 1 2`

Συγκεκριμενοποίηση Πρότυπης Κλάσης

- Το πρότυπο της κλάσης **δεν** αποτελεί ορισμό κλάσης
- Είναι απλά μία **οδηγία** για τον μεταγλωττιστή για το πώς να ορίσει την κλάση
- Η κλάση **ορίζεται** όταν απαιτείται η **συγκεκριμενοποίηση** της κλάσης για τη **δημιουργία** κάποιου **αντικειμένου**. Για παράδειγμα με την εντολή:

```
Lifo<int> lifo_int; // Είναι λάθος να γράψουμε Lifo lifo_int;
```

- ο μεταγλωττιστής **αντικαθιστά** τον τύπο `T` με `int` και ορίζεται η «ακέραια» έκδοση της κλάσης. Έτσι, ο τύπος των στοιχείων που περιέχει το `vec_n` είναι `int`
- Η κλάση `Lifo<int>` αποτελεί έμμεση συγκεκριμενοποίηση (implicit instantiation) της πρότυπης κλάσης

Συγκεκριμενοποίηση Πρότυπης Κλάσης

- Κάθε έκδοση της πρότυπης κλάσης αποτελεί μία ανεξάρτητη κλάση
- Η κλάση που παράγεται λειτουργεί όπως μία συνηθισμένη κλάση
- Παρατηρήστε ότι γράφουμε `Lifo<int>` κλάση, όχι `Lifo`, γιατί αυτή είναι η κλάση που ορίζεται για τον συγκεκριμένο τύπο. Το `lifo_int` αποτελεί αντικείμενο αυτής της έκδοσης, δηλαδή, αντικείμενο της `Lifo<int>` κλάσης. Και για να είναι ξεκάθαρο, είναι λάθος να γράψετε: `Lifo lifo_int;`
- Κάθε αντικείμενο που δηλώνετε πρέπει να αποτελεί αντικείμενο μίας συγκεκριμένης έκδοσης της πρότυπης κλάσης

Συγκεκριμενοποίηση Πρότυπης Κλάσης

- Παρόμοια, με τη δήλωση του αντικειμένου `lifo_string` ο μεταγλωττιστής αντικαθιστά τον τύπο `T` με `string` και ορίζεται η «αλφαριθμητική» έκδοση της κλάσης, δηλαδή, η `Lifo<string>`
- Η `Lifo<string>` δεν έχει καμία σχέση με την `Lifo<int>`, είναι δύο ανεξάρτητες κλάσεις
- Βλέπουμε δηλαδή ότι όταν δηλώνεται ένα αντικείμενο, ο μεταγλωττιστής αντικαθιστά τον γενικό τύπο με τον τύπο του ορίσματος και δημιουργεί την ανάλογη έκδοση της κλάσης
- Έτσι, χάρη στην πρότυπη κλάση μπορούμε να έχουμε στο ίδιο πρόγραμμα διαφορετικούς ορισμούς κλάσεων και αντίστοιχα αντικείμενα χωρίς να χρειαστεί να αντιγράψουμε κώδικα
- Δεν είναι μόνο ότι αποφεύγουμε με τις πρότυπες κλάσεις να ξαναγράψουμε παρόμοιο κώδικα. Το πρόγραμμα γίνεται πιο ευέλικτο, συμπαγές, κατανοητό και πιο εύκολο να συντηρηθεί

Χρήσεις Πρότυπης Κλάσης

- Μία πρότυπη κλάση μπορεί να χρησιμοποιηθεί όπως μία συνηθισμένη κλάση. Για παράδειγμα, μπορεί να αποτελέσει μέλος μίας άλλης κλάσης, να χρησιμοποιηθεί σαν βασική κλάση σε μία σχέση κληρονομικότητας ή να αποτελέσει τον τύπο ορίσματος στην παραγωγή κάποιας άλλης πρότυπης κλάσης. Ας δούμε ένα παράδειγμα κληρονομικότητας:

```
#include <iostream>

template <typename T> class A
{
private:
    T m;
public:
    A() {m = 10;}
    void show() const {std::cout << m << '\n';}
};

template <typename T> class B : public A<T>
{
};

int main()
{
    B<int> b;
    b.show();
    return 0;
}
```

Χρήσεις Πρότυπης Κλάσης

- Η δημιουργία του b αντικειμένου προκαλεί τον ορισμό της κλάσης A<int>, την κλήση του εξ'ορισμού κατασκευαστή και τη δημιουργία του A υπο-αντικειμένου που περιέχεται στο b αντικείμενο. Στη συνέχεια, το πρόγραμμα καλεί την show() της βασικής κλάσης και εμφανίζει 10
- Στο επόμενο παράδειγμα η πρότυπη κλάση A αποτελεί μέλος της C:

```
template <typename T> class C
{
        ...
        A<T> a;
};
```

Χρήσεις Πρότυπης Κλάσης

- Στο επόμενο παράδειγμα, ένα στιγμιότυπο της προτύπης κλάσης χρησιμοποιείται ως μέλος:

```
template <typename T> class A
{
    ...
public:
    T m;
};

class C
{
    ...
public:
    A<int> a;
};

int main()
{
    C c;
    c.a.m = 10;
    ...
}
```

Χρήσεις Πρότυπης Κλάσης

- Ας δούμε και ένα παράδειγμα, όπου ο τύπος της πρότυπης κλάσης χρησιμοποιείται σαν όρισμα στη δημιουργία μίας άλλης:

```
template <typename T> class A
{
        ...
        T m;
};

A<Lifo<int>> a;
```

Το m είναι αντικείμενο της κλάσης Lifo<int>

- Επίσης, μπορούμε να δηλώσουμε ένα δείκτη σε κάποια συγκεκριμένη έκδοση της πρότυπης κλάσης και να δεσμεύσουμε μνήμη για αντίστοιχο αντικείμενο. Για παράδειγμα:

```
Lifo<int> *p;
p = new Lifo<int>; // Δημιουργία αντικειμένου
```

C++: Από τη Θεωρία στην Εφαρμογή

Κεφάλαιο 24ο
Αρχεία

Ρεύματα (Ι)

- Στη C++, ο όρος **ρεύμα** (stream) αναφέρεται στη ροή δεδομένων από μία πηγή προς το πρόγραμμά μας ή από το πρόγραμμά μας προς κάποιον προορισμό
- Αν η ροή είναι εισερχόμενη προς το πρόγραμμά μας, το ρεύμα ονομάζεται **ρεύμα εισόδου**, ενώ, αν η ροή είναι εξερχόμενη, το ρεύμα ονομάζεται **ρεύμα εξόδου**
- Π.χ., στα προγράμματα που έχουμε δει μέχρι τώρα το ρεύμα εισόδου σχετίζεται με το πληκτρολόγιο (πηγή) και το ρεύμα εξόδου με την οθόνη (προορισμός)
- Μεγαλύτερα προγράμματα μπορεί να χρειαστούν περισσότερα ρεύματα, όπως ρεύματα που συνδέονται με αρχεία, αλλά και με άλλα συστήματα (π.χ. εκτυπωτής, κάμερα, αισθητήρες, ...)

Ρεύματα (ΙΙ)

- Το `cout` σχετίζεται με το προκαθορισμένο ρεύμα εξόδου (π.χ. οθόνη)
- Όταν λέμε ότι το `cout` αντιπροσωπεύει το ρεύμα εξόδου βασικά εννοούμε ότι η πληροφορία ρέει, σαν ρεύμα, από το πρόγραμμα, και μέσω του `cout` αντικειμένου, προς την έξοδο (π.χ. οθόνη)
- Παρόμοια, όταν λέμε ότι το `cin` αντιπροσωπεύει το ρεύμα εισόδου εννοούμε ότι η πληροφορία ρέει, σαν ρεύμα, από την είσοδο (π.χ. πληκτρολόγιο), και μέσω του `cin` αντικειμένου, προς το πρόγραμμά μας
- Στη συνέχεια, θα μιλήσουμε για ρεύματα που συνδέονται με αρχεία, δηλαδή, δηλαδή, η έξοδος των δεδομένων κατευθύνεται σε αρχείο και, αντίστροφα, η είσοδος προέρχεται από αρχείο

Αρχεία

- Τα αρχεία χωρίζονται σε δύο κατηγορίες
  - αρχεία κειμένου (text files) και
  - δυαδικά αρχεία (binary files)

Αρχεία Κειμένου

- Τα αρχεία κειμένου αποτελούνται από μία η περισσότερες γραμμές, οι οποίες περιέχουν **χαρακτήρες** σε αναγνώσιμη μορφή, σύμφωνα με κάποια κωδικοποίηση, όπως ο **ASCII κώδικας**

- Κάθε γραμμή τελειώνει με τον **ειδικό χαρακτήρα** που χρησιμοποιεί το λειτουργικό σύστημα για να καθορίσει το **τέλος της γραμμής**

  - Στα Windows για παράδειγμα, το ζευγάρι των χαρακτήρων `'\r'` *(Carriage Return)* και `'\n'` *(Line Feed)*, CR/LF, με ASCII κωδικούς 13 και 10 αντίστοιχα, χαρακτηρίζει το τέλος της γραμμής

  - Άρα, ο χαρακτήρας νέας γραμμής `'\n'` **αντικαθίσταται** από αυτό το ζευγάρι, όταν γράφεται στο αρχείο ενώ η αντίστροφη αντικατάσταση συμβαίνει όταν διαβάζουμε αυτό το ζευγάρι χαρακτήρων από το αρχείο

  - Αντίθετα, σε Unix συστήματα, δεν συμβαίνει αυτή η μετατροπή, γιατί ο χαρακτήρας `'\n'` συμβολίζει το τέλος της γραμμής

Δυαδικά Αρχεία

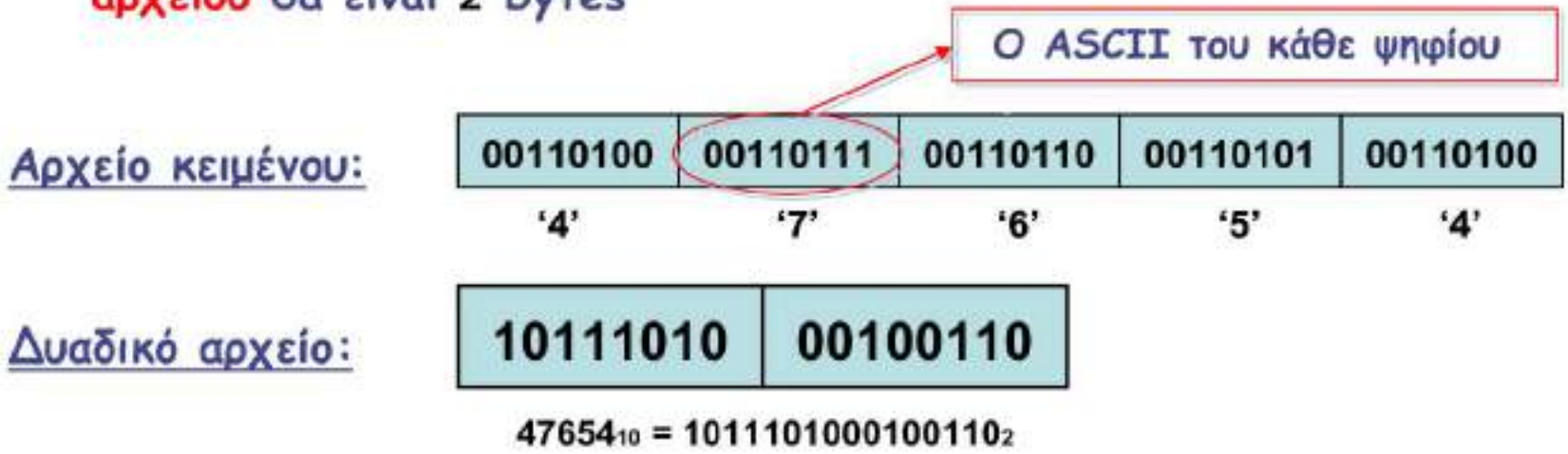
- Αντίθετα με τα αρχεία κειμένου, ένα δυαδικό αρχείο δεν περιέχει απαραίτητα αναγνώσιμους χαρακτήρες
- Για παράδειγμα, το εκτελέσιμο αρχείο ενός C++ προγράμματος, τα δεδομένα μίας εικόνας ή ενός αρχείου ήχου αποθηκεύονται σε δυαδικό αρχείο
- Αν το ανοίξετε με ένα πρόγραμμα κειμενογράφου, το πιθανότερο είναι να δείτε κάποιους «παράξενους» χαρακτήρες
- Επίσης, τα περιεχόμενα ενός δυαδικού αρχείου δεν διαχωρίζονται σε γραμμές και δεν συμβαίνει **καμία** μετατροπή χαρακτήρων
- Στα Windows για παράδειγμα, ο χαρακτήρας νέας γραμμής αποθηκεύεται απευθείας όπως είναι στο δυαδικό αρχείο και δεν μετατρέπεται σε `\r\n`, όπως γίνεται στα αρχεία κειμένου

Δυαδικά Αρχεία

- Μία ακόμα διαφορά μεταξύ των αρχείων κειμένων και των δυαδικών αρχείων είναι ότι στα αρχεία κειμένου το λειτουργικό σύστημα μπορεί να προσθέσει έναν ειδικό χαρακτήρα στο τέλος, ο οποίος δηλώνει το τέλος του αρχείου κειμένου
  - Π.χ. στα Windows αυτός ο χαρακτήρας είναι ο Control-Z (`CTRL-Z`)
- Αντίθετα, στα δυαδικά αρχεία δεν υπάρχει κανένας χαρακτήρας με ειδική σημασία, αλλά αντιμετωπίζονται όλοι με τον ίδιο ακριβώς τρόπο

Δυαδικά Αρχεία

- Η αποθήκευση δεδομένων σε ένα δυαδικό αρχείο μπορεί να εξοικονομήσει χώρο συγκριτικά με ένα αρχείο κειμένου, γιατί στα δυαδικά αρχεία κάθε τύπος δεδομένων αποθηκεύεται με τον αντίστοιχο αριθμό bytes που απαιτούνται
  - Π.χ. το μέγεθος ενός **αρχείου κειμένου** που περιέχει τον αριθμό 47654 (47654_{10} = 1011101000100110_2) θα είναι 5 bytes (αφού περιέχει 5 ψηφία), ενώ το μέγεθος ενός αντίστοιχου **δυαδικού αρχείου** θα είναι 2 bytes



Δυαδικά Αρχεία

- Σημειώστε όμως ότι, όταν ένα δυαδικό αρχείο μεταφέρεται από ένα σύστημα σε ένα άλλο, τα δεδομένα μπορεί να **μην** αναπαρίστανται το ίδιο, γιατί διαφορετικά συστήματα μπορεί να τα αποθηκεύουν με διαφορετικούς τρόπους
  - Π.χ., ένα σύστημα μπορεί να αποθηκεύει πρώτα την υψηλότερη οκτάδα του αριθμού, ενώ κάποιο άλλο τη χαμηλότερη

- Επίσης, επειδή τα μεγέθη των τύπων μπορεί να **διαφέρουν** από σύστημα σε σύστημα, μπορεί να εγγραφεί **διαφορετικός** αριθμός οκτάδων
  - Π.χ., μία συνάρτηση που χρησιμοποιεί το `sizeof(int)` για να αποθηκεύσει έναν ακέραιο σε ένα αρχείο μπορεί να γράψει τέσσερις οκτάδες σε ένα σύστημα, ενώ σε κάποιο άλλο σύστημα οκτώ

Διαχείριση Αρχείων

- Για τη διαχείριση αρχείων, η C++ παρέχει πολλές χρήσιμες κλάσεις
- Ειδικότερα, η κλάση `ofstream` που παράγεται από την `ostream` χρησιμοποιείται για την εγγραφή δεδομένων σε αρχείο
- Η κλάση `ifstream` που παράγεται από την `istream` χρησιμοποιείται για το διάβασμα αρχείου
- Η κλάση `fstream` που παράγεται από τις `ostream` και `istream` χρησιμοποιείται για εγγραφή και διάβασμα από αρχείο
- Αυτές οι κλάσεις κληρονομούν σταθερές και συναρτήσεις για τον έλεγχο της κατάστασης του ρεύματος
- Για να τις χρησιμοποιήσουμε, συμπεριλαμβάνουμε το αρχείο επικεφαλίδας `fstream`

Εγγραφή σε Αρχείο

- Ας δούμε ένα απλό παράδειγμα εγγραφής δεδομένων σε ένα αρχείο. Συνοπτικά, τα βήματα που ακολουθούμε είναι πρώτα να δημιουργήσουμε ένα `ofstream` αντικείμενο, μετά να το συσχετίσουμε με το αρχείο και να χρησιμοποιήσουμε το αντικείμενο, όπως χρησιμοποιούμε το `cout`, για την εγγραφή των δεδομένων

Παράδειγμα

```
#include <iostream>
#include <fstream>
#include <cstdlib> // Για την exit()

int main()
{
    int i;
    std::ofstream fout; /* Το fout είναι ένα ρεύμα εξόδου που δεν έχει
    συσχετιστεί με κάποιο αρχείο. */

    fout.open("test.txt"); /* Άνοιγμα αρχείου για εγγραφή. Θα μπορούσαμε
    να δημιουργήσουμε το fout και να το συσχετίσουμε με το αρχείο σε μία
    εντολή, όπως ofstream fout("test.txt");  */
    if(fout.is_open() == false) /* Εναλλακτικά, μπορούμε να γράψουμε
    if(!fout). */
    {
            std::cout << "Error: File can not be opened\n";
            exit(EXIT_FAILURE); /* Το αρχείο δεν άνοιξε, οπότε το
    πρόγραμμα τερματίζεται. */
    }
    for(i = 0; i < 3; i++)
            fout << "Hello_" << i+1 << '\n';
    fout.close(); // Κλείσιμο αρχείου.
    return 0;
}
```

Παράδειγμα

- Το όνομα του αντικειμένου μπορεί να είναι οποιοδήποτε έγκυρο C++ όνομα (π.χ. `fout`)
- Όταν ανοίγουμε ένα αρχείο για εγγραφή με αυτό τον τρόπο, το πρόγραμμα δημιουργεί το αρχείο, εφόσον αυτό δεν υπάρχει. Αν υπάρχει, το περιεχόμενό του διαγράφεται. Στη συνέχεια, θα δούμε με ποιο τρόπο μπορούμε να γράψουμε σε ένα υπάρχον αρχείο, χωρίς να διαγραφεί το περιεχόμενό του
- Αν το αρχείο βρίσκεται στον ίδιο φάκελο με το εκτελέσιμο πρόγραμμα, απλά γράφουμε το όνομα του αρχείου σε διπλά εισαγωγικά. Αν το αρχείο δεν υπάρχει, το αρχείο δημιουργείται στον ίδιο φάκελο με το εκτελέσιμο πρόγραμμα. Αν επιθυμούμε να ανοίξουμε ένα αρχείο που βρίσκεται σε διαφορετικό φάκελο από το εκτελέσιμο πρόγραμμα, τότε πρέπει να καθοριστεί η πλήρης διαδρομή

Παράδειγμα

- Αν το λειτουργικό σύστημα χρησιμοποιεί τον χαρακτήρα \ για τον καθορισμό της διαδρομής, γράφουμε \\, γιατί, όπως είδαμε στο Κ.3, η C++ χειρίζεται τον χαρακτήρα \ σαν την αρχή μίας ακολουθίας διαφυγής

- Π.χ. αν το πρόγραμμα εκτελείται σε Windows και θέλουμε να ανοίξουμε για διάβασμα το αρχείο `test.txt` που υπάρχει στη διαδρομή `d:\dir1\dir2`, πρέπει να γράψουμε:

```
fout.open("d:\\dir1\\dir2\\test.txt");
```

Ωστόσο, αν το πρόγραμμα δέχεται το όνομα του αρχείου από τη γραμμή εντολών, δεν χρειάζεται να προσθέσουμε τη δεύτερη \, δηλαδή θα πρέπει να πληκτρολογήσουμε:

`d:\dir1\dir2\test.txt` (με μία \)

Παράδειγμα

- Με την `is_open()` ελέγχουμε αν η προσπάθεια ανοίγματος του αρχείου ήταν επιτυχής. Αν όχι, το πρόγραμμα τερματίζει
- Για παράδειγμα, η `is_open()` αποτυγχάνει όταν η διαδρομή του αρχείου δεν είναι έγκυρη ή δεν έχουμε δικαίωμα εγγραφής στο αρχείο
- Επειδή η κλάση `ofstream` παράγεται από την `ostream`, τα αντικείμενά της μπορούν να χρησιμοποιήσουν τις συναρτήσεις της, όπως, για παράδειγμα, τις υπερφορτωμένες `operator<<` συναρτήσεις
- Η `close()` κλείνει το αρχείο που είναι συνδεδεμένο με το αντικείμενο που την καλεί. Όσα δεδομένα υπάρχουν στη μνήμη γράφονται στο αρχείο
- Επίσης, ένα ανοικτό αρχείο κλείνει όταν το αντίστοιχο αντικείμενο παύει να υπάρχει, για παράδειγμα, όταν τερματίσει το πρόγραμμα

Διάβασμα από Αρχείο

- Παρόμοια, για να διαβάσουμε το περιεχόμενο ενός αρχείου, θα πρέπει να δημιουργήσουμε ένα `ifstream` αντικείμενο, να το συσχετίσουμε με το αρχείο και να χρησιμοποιήσουμε το αντικείμενο, όπως χρησιμοποιούμε το `cin`, για το διάβασμα του περιεχομένου του
- Επειδή η κλάση `ifstream` παράγεται από την `istream`, τα αντικείμενά της μπορούν να χρησιμοποιήσουν τις συναρτήσεις της, όπως, για παράδειγμα, τις υπερφορτωμένες `operator>>` συναρτήσεις
- Στο παρακάτω παράδειγμα, ας υποθέσουμε ότι κάθε γραμμή του αρχείου που θα εισάγει ο χρήστης περιέχει τις θερμοκρασίες μίας περιοχής. Το πρόγραμμα τις διαβάζει και εμφανίζει αυτές με τιμή στο [-5, 5]

Παράδειγμα

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

int main()
{
    double tmp;
    string fname;

    cout << "Enter file name: ";
    cin >> fname;

    ifstream fin(fname.c_str()); /* Επειδή η παράμετρος του κατασκευαστή έχει τύπο
    const char* μεταβιβάζουμε σαν όρισμα το C-μορφής αλφαριθμητικό. */
    if(fin.is_open() == false)
    {
            cout << "Error: File can not be opened\n";
            exit(EXIT_FAILURE);
    }
    while(1)
    {
            fin >> tmp;
            if(!fin) /* Αν συμβεί κάποιο λάθος στο διάβασμα των δεδομένων ή
    φτάσουμε στο τέλος του αρχείου η συνάρτηση operator!() επιστρέφει true. */
                    break;
            if(tmp >= -5 && tmp <= 5)
                    cout << tmp << '\n';
    }
    fin.close();
    return 0;
}
```

Εγγραφή/Διάβασμα από Αρχείο

- Αν θέλουμε να διαβάσουμε και να γράψουμε σε ένα αρχείο θα πρέπει να συσχετίσουμε το αρχείο με ένα `fstream` αντικείμενο. Η κλάση `fstream` παράγεται από τις `istream` και `ostream` κλάσεις, άρα κληρονομεί τις συναρτήσεις τους. Ας δούμε ένα παράδειγμα:

Παράδειγμα

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

struct Student
{
    string name;
    int code;
    float grd;
};

int main()
{
    Student s1 = {"J.Lee", 100, 5.8}, s2;

    fstream fstr("test.txt", ios_base::in | ios_base::out | ios_base::trunc);
    if(fstr.is_open() == false)
    {
            cout << "Error: File can not be opened\n";
            exit(EXIT_FAILURE);
    }
    fstr << s1.name << ' ' << s1.code << ' ' << s1.grd << '\n';
    fstr.seekg(0);
    fstr >> s2.name >> s2.code >> s2.grd;
    cout << s2.name << ' ' << s2.code << ' ' << s2.grd << '\n';
    fstr.close();

    return 0;
}
```

Παράδειγμα

- Το αρχείο ανοίγει για διάβασμα και εγγραφή. Αν υπάρχει, το περιεχόμενό του διαγράφεται
- Ο λόγος που χρησιμοποιείται δομή, αντί για απλές μεταβλητές, είναι για να δείτε ένα παράδειγμα χρήσης των πεδίων της για ανάγνωση/εγγραφή από/στο αρχείο
- Γράφουμε τα στοιχεία της δομής στο αρχείο και με την `seekg()`, για την οποία θα μιλήσουμε στη συνέχεια, επαναφέρουμε τον δείκτη θέσης στην αρχή του αρχείου, ώστε να διαβάσουμε το περιεχόμενό του
- Για την αποθήκευση της πληροφορίας, χρησιμοποιούμε τη δεύτερη δομή `s2`. Θα μπορούσαμε να χρησιμοποιήσουμε πάλι την `s1`, ο λόγος που δεν τη χρησιμοποιούμε είναι για να είναι πιο ξεκάθαρο το παράδειγμα
- Το πρόγραμμα εμφανίζει τα στοιχεία της `s2`, που είναι τα ίδια με της `s1`, κλείνει το αρχείο και τερματίζει
- Σημειώστε ότι αν αντί για δομή χρησιμοποιήσουμε κλάση με τα αντίστοιχα πεδία δηλωμένα δημόσια, ο κώδικας παραμένει ίδιος

Τρόποι Ανοίγματος Αρχείου

- Όταν συσχετίζουμε ένα αρχείο με ένα αντικείμενο, είτε αρχικοποιώντας το αντικείμενο με το όνομα του αρχείου είτε πρώτα να δημιουργήσουμε το αντικείμενο και μετά να χρησιμοποιήσουμε τη συνάρτηση `open()`, μπορούμε να χρησιμοποιήσουμε ένα δεύτερο όρισμα, το οποίο καθορίζει τον τρόπο ανοίγματος του αρχείου
- Η κλάση `ios_base` ορίζει σταθερές που αντιπροσωπεύουν τον τρόπο ανοίγματος (π.χ. `ios_base::in`). Ο bit τελεστής | μπορεί να χρησιμοποιηθεί για την ενεργοποίηση πολλαπλών επιλογών, όπως φαίνεται στον παρακάτω πίνακα

Τρόποι Ανοίγματος Αρχείου

Επιλογή	Ενέργεια
`ios_base::in`	Ανοίγει το αρχείο για ανάγνωση.
`ios_base::out` ή `ios_base::out | ios_base::trunc`	Ανοίγει το αρχείο για εγγραφή. Αν το αρχείο δεν υπάρχει, δημιουργείται. Αν υπάρχει, το περιεχόμενό του διαγράφεται.
`ios_base::app`	Ανοίγει το αρχείο για προσάρτηση. Αν το αρχείο δεν υπάρχει, δημιουργείται. Αν υπάρχει, τα υπάρχοντα δεδομένα διατηρούνται και τα νέα δεδομένα προστίθενται στο τέλος.
`ios_base::in | ios_base::out`	Ανοίγει το αρχείο για ανάγνωση και εγγραφή.
`ios_base::in | ios_base::out | ios_base::trunc`	Ανοίγει το αρχείο για ανάγνωση και εγγραφή. Αν το αρχείο δεν υπάρχει, δημιουργείται. Αν υπάρχει, το περιεχόμενό του διαγράφεται.
`ios_base::in | ios_base::out | ios_base::app`	Ανοίγει το αρχείο για ανάγνωση και προσάρτηση. Αν το αρχείο δεν υπάρχει, δημιουργείται. Αν υπάρχει, τα υπάρχοντα δεδομένα διατηρούνται και τα νέα δεδομένα προστίθενται στο τέλος.

Τρόποι Ανοίγματος Αρχείου

- Επίσης, υπάρχει και η επιλογή `ios_base::ate`, η οποία τοποθετεί τον δείκτη θέσης του αρχείου στο τέλος του. Η διαφορά μεταξύ των `ios_base::app` και `ios_base::ate` είναι ότι η πρώτη επιλογή επιτρέπει την προσθήκη δεδομένων μόνο στο τέλος του αρχείου, ενώ η δεύτερη απλά τοποθετεί τον δείκτη θέσης στο τέλος του αρχείου
- Για το άνοιγμα ενός αρχείου κειμένου επιλέγουμε κάποιον από τους παραπάνω τρόπους. Για το άνοιγμα ενός δυαδικού αρχείου, προσθέτουμε την σταθερά `ios_base::binary`. Για παράδειγμα, για να ανοίξουμε ένα δυαδικό αρχείο για διάβασμα γράφουμε `ios_base::in | ios_base::binary`, ενώ για να το ανοίξουμε για διάβασμα και εγγραφή γράφουμε `ios_base::in | ios_base::out | ios_base::trunc | ios_base::binary ios_base::in |`

Τρόποι Ανοίγματος Αρχείου

- Όπως είπαμε, ο τρόπος ανοίγματος είναι προαιρετικό όρισμα. Αν δεν καθοριστεί, όπως στα προηγούμενα παραδείγματα, θα ισχύσει ο προκαθορισμένος τρόπος. Ειδικότερα, η προκαθορισμένη τιμή για την `ifstream open()` και τον κατασκευαστή της είναι η `ios_base::in`, ενώ για την `ofstream open()` και τον κατασκευαστή της είναι η `ios_base::out`. Η προκαθορισμένη τιμή για την `fstream open()` και τον κατασκευαστή της είναι η `ios_base::in | ios_base::out`. Ας δούμε μερικά παραδείγματα:

```
ifstream fin("test.txt"); /* Το αρχείο ανοίγει για διάβασμα με τον προκαθορισμένο τρόπο ios_base::in */
ofstream fout("test.txt", ios_base::app); /* Το αρχείο ανοίγει για εγγραφή και προσάρτηση νέων δεδομένων στο τέλος του. Το αρχικό περιεχόμενο του αρχείου διατηρείται. */
fstream f("test.txt", ios_base::in | ios_base::out | ios_base::trunc); /* Το αρχείο ανοίγει για διάβασμα και εγγραφή. Το περιεχόμενό του διαγράφεται. */
```

Τυχαία Προσπέλαση Αρχείου

- Για την τυχαία προσπέλαση ενός αρχείου που έχει συσχετιστεί με ένα `ifstream` αντικείμενο χρησιμοποιούμε τη συνάρτηση `seekg()`. Υπάρχουν δύο εκδόσεις της με τα ακόλουθα πρωτότυπα:

```
istream& seekg(streamoff offset, ios_base::seekdir origin);
istream& seekg(streampos offset);
```

- Ένα `ifstream` αντικείμενο περιέχει ένα δείκτη για την ανάγνωση του αρχείου. Η πρώτη έκδοση της `seekg()` μετακινεί αυτόν τον δείκτη σε μία νέα θέση που απέχει `offset` οκτάδες από τη θέση που δηλώνει το origin. Ο τύπος `streamoff` είναι συνώνυμο κάποιου ακέραιου τύπου

Τυχαία Προσπέλαση Αρχείου

- Η τιμή του origin πρέπει να είναι κάποια από τις ακόλουθες ακέραιες σταθερές της ios_base κλάσης:

ios_base::beg. Ο δείκτης θα μετατοπιστεί offset οκτάδες από την αρχή του αρχείου

ios_base::cur. Ο δείκτης θα μετατοπιστεί offset οκτάδες από την τρέχουσα θέση του

ios_base::end. Ο δείκτης θα μετατοπιστεί offset οκτάδες από το τέλος του αρχείου

- Στη δεύτερη έκδοση, η offset τιμή υποδεικνύει σε οκτάδες από την αρχή του αρχείου τη νέα θέση του αρχείου στην οποία θα μετακινηθεί ο δείκτης. Ας δούμε μερικά παραδείγματα:

```
seekg(0, ios_base::end); /* Μετακίνηση στο τέλος του αρχείου. */
seekg(20, ios_base::beg); /* Μετακίνηση 20 οκτάδες από την αρχή του αρχείου. */
seekg(-5, ios_base::cur); /* Μετακίνηση 5 οκτάδες πίσω από την τρέχουσα θέση. */
```

Τυχαία Προσπέλαση Αρχείου

- Αν θέλουμε να βρούμε σε ποιο σημείο του αρχείου βρίσκεται ο δείκτης μπορούμε να χρησιμοποιήσουμε τη συνάρτηση `tellg()`. Η `tellg()` επιστρέφει μία `streampos` τιμή, η οποία αντιπροσωπεύει την τρέχουσα θέση του δείκτη θέσης σε οκτάδες από την αρχή του αρχείου του αρχείου ή `-1` αν συμβεί κάποιο λάθος. Όπως φαίνεται στον παρακάτω κώδικα, η `tellg()` μπορεί να συνδυαστεί με την `seekg()`, για να επιστρέψουμε σε μία προηγούμενη θέση. Για παράδειγμα:

```
// Αποθηκεύουμε τη θέση στην οποία θέλουμε να επιστρέψουμε.
streampos old_pos = fin.tellg();
// ... πηγαίνουμε σε άλλη θέση για διάβασμα.
fin.seekg(old_pos); // Επιστροφή στην παλιά θέση.
```

- Παρόμοια, ένα `ofstream` αντικείμενο περιέχει ένα δείκτη για την εγγραφή δεδομένων στο αρχείο
- Για την τυχαία προσπέλαση ενός αρχείου που έχει συσχετιστεί με ένα `ofstream` αντικείμενο χρησιμοποιούμε τις συναρτήσεις `seekp()` και `tellp()`

Τυχαία Προσπέλαση Αρχείου

- Σημειώστε ότι αν οι `seekg()` και `seekp()` χρησιμοποιούνται σε αρχεία κειμένου, χρειάζεται **προσοχή** με τους χαρακτήρες νέας γραμμής
- Για παράδειγμα, ας υποθέσουμε ότι ο παρακάτω κώδικας γράφει κάποιο κείμενο στις δύο πρώτες γραμμές ενός αρχείου κειμένου. Αν το λειτουργικό σύστημα μετατρέπει το '\n' σε '\r' και '\n', η τιμή του `offset` πρέπει να είναι 6 (και όχι 5), για να μετακινηθούμε στην αρχή της δεύτερης γραμμής.

```
fout << "text\n";
fout << "another text\n";
/* Μετακίνηση στην αρχή της δεύτερης γραμμής. */
fout.seekp(6, ios_base::beg);
```

- Είναι **ασφαλέστερο** να χρησιμοποιείτε τις `seekg()` και `seekp()` σε δυαδικά αρχεία και όχι σε αρχεία κειμένου, αλλιώς πρέπει να λαμβάνετε υπόψη σας ενδεχόμενες μετατροπές του χαρακτήρα νέας γραμμής

Παράδειγμα

- Να γραφεί ένα πρόγραμμα το οποίο να διαβάζει συνεχώς τους κωδικούς προϊόντων και τις τιμές τους και να τα αποθηκεύει στο αρχείο κειμένου `test.txt`, ως εξής:

```
C101  17.5
C102  32.8
...
```

- Αν ο χρήστης εισάγει την τιμή `-1` ως τιμή προϊόντος, η εισαγωγή των προϊόντων να τερματίζει. Στη συνέχεια, το πρόγραμμα να διαβάζει τον κωδικό ενός προϊόντος και να ψάχνει το αρχείο για να βρει και να εμφανίσει την τιμή του, αν υπάρχει

Παράδειγμα

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

int main()
{
    bool found;
    string code, tmp;
    double prc;

    fstream fstr("test.txt", ios_base::in | ios_base::out | ios_base::trunc);
    if(fstr.is_open() == false)
    {
            cout << "Error: is_open() failed\n";
            exit(EXIT_FAILURE);
    }
    while(1)
    {
            cout << "Enter price: ";
            cin >> prc;
            if(prc == -1)
                    break;
            cin.get();
            cout << "Enter product code: ";
            getline(cin, code);
            fstr << code << ' ' << prc << '\n';
    }
```

Παράδειγμα

```
    cin.get();
    cout << "Enter product code to search for: ";
    getline(cin, tmp);

    found = 0;
    fstr.seekg(0);
    while(1)
    {
            fstr >> code >> prc;
            if(!fstr)
                    break;
            if(code == tmp)
            {
                    found = 1;
                    break; /* Αφού το προϊόν βρέθηκε μπορούμε να τερματίσουμε τον
βρόχο. */
            }
    }
    if(found == 0)
            cout << tmp << " code is not listed\n";
    else
            cout << "The price for product " << code << " is " << prc << '\n';
    fstr.close();
    return 0;
}
```

Εγγραφή και Διάβασμα από Δυαδικό Αρχείο

- Οι συναρτήσεις που χρησιμοποιούνται για εγγραφή και διάβασμα δεδομένων σε/από ένα δυαδικό αρχείο είναι οι `write()` και `read()`, αντίστοιχα
- Αν και συνήθως χρησιμοποιούνται σε δυαδικά αρχεία, μπορούν να χρησιμοποιηθούν με προσοχή και σε αρχεία κειμένου

Η Συνάρτηση write()

- Η συνάρτηση write() είναι πολύ χρήσιμη για την εγγραφή μεγάλου όγκου δεδομένων σε ένα βήμα. Το πρωτότυπό της είναι:

```
ostream& write(const char *buf, streamsize count);
```

- Η παράμετρος buf είναι ένας δείκτης στη διεύθυνση μνήμης που βρίσκονται τα δεδομένα που θα εγγραφούν στο αρχείο. Το streamsize αποτελεί συνώνυμο για κάποιον ακέραιο τύπο. Η παράμετρος count καθορίζει τον αριθμό των οκτάδων που θα αποθηκευτούν στο αρχείο. Για παράδειγμα, για την εγγραφή ενός πίνακα 1000 ακεραίων γράφουμε:

```
int arr[1000];
ofstream fout("test.dat", ios_base::out | ios_base::trunc |
ios_base::binary);
fout.write((char*)arr, sizeof(arr));
```

- Όπως φαίνεται στο παράδειγμα, η διεύθυνση πρέπει να προσαρμοστεί σε char*
- Επίσης, η καλύτερη πρακτική για τον υπολογισμό του μεγέθους είναι να χρησιμοποιείτε τον τελεστή sizeof, ώστε το πρόγραμμα να μην εξαρτάται από το σύστημα στο οποίο εκτελείται
- Και αν θέλουμε να ελέγξουμε ότι δεν έγινε κάποιο λάθος, ελέγχουμε την κατάσταση του ρεύματος. Για παράδειγμα:

```
if(!fout)
        cout << "write error\n";
```

Η Συνάρτηση read()

- Παρόμοια με την write(), η read() είναι πολύ χρήσιμη για το διάβασμα μεγάλου όγκου δεδομένων σε ένα βήμα. Το πρωτότυπό της είναι:

```
istream& read(char *buf, streamsize count);
```

- Η παράμετρος buf είναι ένας δείκτης στη διεύθυνση μνήμης, στην οποία θα αποθηκευτούν τα δεδομένα που θα διαβαστούν από το αρχείο. Η παράμετρος count καθορίζει τον αριθμό των οκτάδων που θα διαβαστούν. Στο επόμενο παράδειγμα διαβάζονται 1000 ακέραιοι και αποθηκεύονται στον πίνακα arr

```
int arr[1000];
ifstream fin("test.dat", ios_base::in |
ios_base::binary);
fin.read((char*)arr, sizeof(arr));
```

Παράδειγμα

- Υποθέστε ότι το δυαδικό αρχείο `test.bin` περιέχει τους βαθμούς ενός φοιτητή. Το πλήθος των βαθμών είναι αποθηκευμένο στην αρχή του αρχείου. Να γραφεί ένα πρόγραμμα το οποίο να διαβάζει τους βαθμούς από το αρχείο (χρησιμοποιήστε τον τύπο `float`) και να τους αποθηκεύει σε μία δυναμικά δεσμευμένη μήμη. Στη συνέχεια, να διαβάζει έναν αριθμό και να εμφανίζει τους βαθμούς με μεγαλύτερη τιμή από αυτόν

Παράδειγμα

```
#include <iostream>
#include <fstream>
#include <cstdlib>
using namespace std;

int main()
{
	int i, grd_num;
	float *grd_arr, grd;

	ifstream fin("test.bin", ios_base::in | ios_base::binary);
	if(fin.is_open() == false)
	{
		cout << "Error: is_open() failed\n";
		exit(EXIT_FAILURE);
	}
	fin.read((char*)&grd_num, sizeof(int));
	if(!fin)
	{
		fin.close();
		cout << "Error: read() failed\n";
		exit(EXIT_FAILURE);
	}
	grd_arr = new float[grd_num]; /* Δέσμευση μνήμης για την αποθήκευση των βαθμών. */
	fin.read((char*)grd_arr, grd_num * sizeof(float)); /* Διάβασμα των βαθμών και έλεγχος ότι δεν συνέβη
	κάποιο λάθος. */
	if(fin)
	{
		cout << "Enter grade: ";
		cin >> grd;
		for(i = 0; i < grd_num; i++)
			if(grd_arr[i] > grd)
				cout << grd_arr[i] << '\n';
	}
	else
		cout << "Error: read() failed to read grades\n";

	delete[] grd_arr;
	fin.close();
	return 0;
}
```

C++: Από τη Θεωρία στην Εφαρμογή

Κεφάλαιο 25ο

Χώροι Ονομάτων, Προσαρμογές Τύπων και Έξυπνοι Δείκτες

Χώροι Ονομάτων

- Ένας **χώρος ονομάτων** (namespace) είναι μία **ονοματισμένη** περιοχή με τη δική της εμβέλεια, στην οποία μπορούμε να τοποθετήσουμε τα στοιχεία του προγράμματός μας (π.χ. κλάσεις, συναρτήσεις, ...)
- Ένας χώρος ονομάτων μας επιτρέπει να **οργανώνουμε** τα στοιχεία του προγράμματός μας σε λογικές μονάδες και να αποφεύγουμε συγκρούσεις με ονόματα που δηλώνονται έξω από αυτόν τον χώρο
- Για παράδειγμα, μπορούμε να δημιουργήσουμε τον χώρο ονομάτων Network, στον οποίο να τοποθετήσουμε κλάσεις που σχετίζονται με πρωτόκολλα επικοινωνιών, δικτυακές συσκευές και υπηρεσίες
- Γενικά, σε μεγάλες εφαρμογές, χρησιμοποιούνται πολλοί χώροι ονομάτων, γιατί ο διαχωρισμός του προγράμματος σε λογικές μονάδες, ανάλογα με τη λειτουργικότητά τους (π.χ. βιβλιοθήκη γραφικών, βιβλιοθήκη δικτυακών υπηρεσιών, ...), αποτρέπει τη σύγκρουση ονομάτων και διευκολύνει την οργάνωση, κατανόηση, διαχείριση, και συντήρησή του
- Στοιχεία που δηλώνονται με καθολική εμβέλεια, όπως καθολικές μεταβλητές, λέμε ότι ανήκουν στον **καθολικό** χώρο ονομάτων *(global namespace)*

Δημιουργία Χώρου Ονομάτων

- Για να δημιουργήσουμε ένα χώρο ονομάτων χρησιμοποιούμε τη δεσμευμένη λέξη `namespace`. Για παράδειγμα:

```
namespace A
{
        int var;
        double d = 1.2;
        int f();
        struct S {...};
        class C {...};
} // Δεν απαιτείται ερωτηματικό.
```

- Μέσα σε ένα χώρο μπορούμε να δηλώσουμε τα στοιχεία που επιθυμούμε, όπως μεταβλητές, δηλώσεις και ορισμούς συναρτήσεων, πρότυπα, κλάσεις και αντικείμενα
- Ο χώρος ονομάτων επιτρέπεται να δηλωθεί στον καθολικό χώρο ή μέσα σε κάποιον άλλο χώρο, όχι όμως μέσα σε κάποιο τμήμα (π.χ. συνάρτηση)

Εμβέλεια Χώρου Ονομάτων

- Ένας χώρος ονομάτων ορίζει τη **δική** του εμβέλεια. Η εμβέλεια κάθε μέλους του εκτείνεται από το σημείο δήλωσής του μέχρι το τέλος του χώρου
- Έτσι, τα ονόματα που χρησιμοποιούμε σε ένα χώρο ονομάτων **δεν συγκρούονται** με ίδια ονόματα σε άλλους χώρους ονομάτων ή στον καθολικό χώρο
- Για παράδειγμα, αν δημιουργήσουμε τον χώρο ονομάτων `B` που να περιέχει μία μεταβλητή `var` και δηλώσουμε και μία καθολική μεταβλητή var δεν θα υπάρχει σύγκρουση μεταξύ τους, δηλαδή, μεταξύ των `A::var`, `B::var` και της καθολικής `var`
- Αυτό είναι και το **κύριο** πλεονέκτημα των χώρων ονομάτων, η αποφυγή συγκρούσεων. Σε μεγάλα έργα που συμμετέχουν πολλοί προγραμματιστές, η συνήθης πρακτική είναι να δημιουργούνται ξεχωριστοί χώροι ονομάτων, ώστε ακόμα και αν έχουν επιλεγεί κοινά ονόματα να μην προκαλούνται συγκρούσεις

Πρόσθεση Στοιχείων

- Μέσα στο χώρο μπορούμε να προσθέσουμε νέα στοιχεία χρησιμοποιώντας το όνομα του χώρου. Για παράδειγμα, η επόμενη εντολή επεκτείνει την εμβέλεια του χώρου **A** και προσθέτει τη δήλωση της συνάρτησης **f()** στα στοιχεία του **A**:

```
namespace A
{
        int f()
        {
                ...
        }
}
```

- Εναλλακτικά, μπορούμε να την ορίσουμε, όπως ορίζουμε μία συνάρτηση έξω από την κλάση της:

```
int A::f()
{
        ...
}
```

Πρόσπελαση Στοιχείων

- Αφού ένας χώρος ονομάτων ορίζει τη δική του εμβέλεια, τα στοιχεία του δεν είναι ορατά έξω από αυτόν. Ένας τρόπος για να τα προσπελάσουμε είναι με χρήση του τελεστή επίλυσης εμβέλειας `::`. Για παράδειγμα:

```
A::var = 10;
int i = A::f();
```

- Εναλλακτικά, μπορούμε να χρησιμοποιήσουμε `using` δηλώσεις και οδηγίες, όπως θα δούμε στη συνέχεια
- Μέσα στο χώρο, τα ονόματα του χώρου μπορούν να προσπελαστούν χωρίς τη χρήση του τελεστή εμβέλειας. Για παράδειγμα:

```
namespace A
{
        int f()
        {
                d = 5;
        }
}
```

using Δηλώσεις

- Για να μη γράφουμε κάθε φορά το όνομα του χώρου, που και ενοχλητικό ή κουραστικό μπορεί να είναι αλλά και ο κώδικας να γίνεται πιο δυσανάγνωστος, ιδιαίτερα αν το όνομα είναι μεγάλο, ένας πιο βολικός τρόπος είναι να χρησιμοποιήσουμε **using** δηλώσεις (using declarations)
- Μία **using** δήλωση μπορεί να τοποθετηθεί στον καθολικό χώρο όπως στα παραδείγματα του βιβλίου, μέσα σε ένα χώρο ονομάτων ή σε μία συνάρτηση, ακόμα και μέσα σε ένα τμήμα εντολών
- Μία **using** δήλωση κάνει διαθέσιμο **ένα μόνο** όνομα. Ας δούμε ένα παράδειγμα με τη χρήση του ίδιου ονόματος:

```
int var; // Καθολική μεταβλητή.
using A::var;
int main()
{
        ...
        var = 10; // Αναφερόμαστε στην A::var.
        ::var = 20; // Αναφερόμαστε στην καθολική var.
}
```

using Οδηγίες

- Σε αντίθεση με την using δήλωση, μία using οδηγία (using directive) κάνει διαθέσιμα όλα τα στοιχεία του χώρου ονομάτων. Έτσι, δεν χρειάζεται να χρησιμοποιούμε τον τελεστή ::
- Για παράδειγμα, έχουμε κάνει αρκετές φορές διαθέσιμο όλο τον χώρο ονομάτων std

```
...
using namespace std;
using namespace A;
...
cin >> var; // Αντί για std::cin >> A::var;
string f(vector<string>& v); /* Αντί για
std::string f(std::vector<std::string>& v); */
```

Ο Τελεστής dynamic_cast

- Ο τελεστής dynamic_cast χρησιμοποιείται συνήθως σε μία ιεραρχία κλάσεων και μας δίνει τη δυνατότητα να ελέγξουμε αν η προσαρμογή ενός τύπου σε έναν άλλο είναι έγκυρη
- Ο τελεστής dynamic_cast εφαρμόζεται μόνο σε δείκτες και αναφορές σε πολυμορφικές κλάσεις, δηλαδή, πρέπει να υπάρχει τουλάχιστον μία εικονική συνάρτηση στην κλάση (δηλωμένη ή κληρονομημένη) στην οποία δείχνει ο δείκτης ή αναφέρεται η αναφορά
- Δέχεται δύο τελεστέους, ένα δείκτη (ή αναφορά) σε έναν τύπο κλάσης μέσα σε <> και τον δείκτη (ή την αναφορά) που θα προσαρμοστεί μέσα σε ()
- Οι τύποι που προσαρμόζουμε πρέπει να είναι δείκτες ή αναφορές σε κλάσεις στην ίδια ιεραρχία κλάσεων. Ας δούμε ένα παράδειγμα σύνταξής του με δείκτη:

```
dynamic_cast<Type*>(p);
```

- Ο p είναι ο δείκτης που θα προσαρμοστεί. Ο Type είναι ο τύπος στον οποίο ο p θα πρέπει να προσαρμοστεί

Ο Τελεστής dynamic_cast

- Ο σκοπός του `dynamic_cast` είναι να εξασφαλίσει ότι το αποτέλεσμα της προσαρμογής θα δείχνει σε ένα έγκυρο αντικείμενο του επιλεγμένου τύπου (π.χ. `Type`)
- Αν η μετατροπή τύπου είναι επιτυχημένη, ο τελεστής επιστρέφει ένα δείκτη τύπου `Type*`. Αλλιώς, επιστρέφει μηδενικό δείκτη
- Αν ο `p` είναι ο μηδενικός δείκτης, ο τελεστής επιστρέφει μηδενικό δείκτη
- Η προσαρμογή τύπου γίνεται κατά την εκτέλεση του προγράμματος, όχι στη μεταγλώττιση. Άλλωστε, η λέξη `dynamic` (δυναμική) υποδηλώνει αυτή τη συμπεριφορά

Παράδειγμα

```
#include <iostream>
class A
{
public:
     virtual void show() const {std::cout << "A\n";}
};
class B : public A
{
public:
     virtual void show() const {std::cout << "B\n";}
};
class C : public B
{
public:
     virtual void show() const {std::cout << "C\n";}
};
int main()
{
     B *b = new B;

     A *a = dynamic_cast<A*>(b); /* Ασφαλής προσαρμογή. Ένα B αντικείμενο περιέχει ένα A υπο-
     αντικείμενο. */
     if(a)
     {
               a->show();
               std::cout << a << ' ' <<  b << '\n';
     }
     C *c = dynamic_cast<C*>(b); /* Μη ασφαλής προσαρμογή. Ένα B αντικείμενο δεν περιέχει ένα C
     υπο-αντικείμενο. */
     if(c)
               c->show();
     else
               std::cout << "Error\n";
     delete b;
     return 0;
}
```

Παράδειγμα

- Η πρώτη προσαρμογή είναι ασφαλής και ο τελεστής `dynamic_cast` επιστρέφει μη μηδενικό δείκτη. Επειδή η `show()` είναι εικονική και ο δείκτης `a` δείχνει σε αντικείμενο της κλάσης `B`, το πρόγραμμα εμφανίζει `B`. Στη συνέχεια, αφού οι δείκτες `a` και `b` δείχνουν στην ίδια διεύθυνση το πρόγραμμα εμφανίζει δύο φορές την ίδια τιμή. Επειδή η δεύτερη προσαρμογή δεν είναι ασφαλής, ο τελεστής επιστρέφει μηδενικό δείκτη και το πρόγραμμα εμφανίζει `Error`

Ο Τελεστής const_cast

- Ο τελεστής const_cast χρησιμοποιείται για να προσθέσει ή να απομακρύνει την const ιδιότητα από μία μεταβλητή
- Ο const_cast χρησιμοποιείται με δείκτες και αναφορές. Έχει παρόμοια σύνταξη με τον τελεστή dynamic_cast:

  const_cast<όνομα_τύπου>(έκφραση);
- Συνήθως, χρησιμοποιείται για να απομακρύνει την const ιδιότητα του ορίσματος που μεταβιβάζεται. Ας δούμε ένα παράδειγμα:

Παράδειγμα

```
#include <iostream>
int main()
{
    int *p1, i = 10;

    const int *p2 = &i;
    // Δεν επιτρέπεται να γράψουμε *p2 = 20.
    p1 = const_cast<int*>(p2);
    *p1 = 20;
    std::cout << i << '\n';
    return 0;
}
```

Παράδειγμα

- Ας υποθέσουμε ότι θέλουμε να αλλάξουμε την τιμή του `i` μέσω δείκτη. Αφού δεν μπορούμε να χρησιμοποιήσουμε τον `p2` για να το κάνουμε αυτό, χρησιμοποιούμε μία άλλη μεταβλητή-δείκτη
- Η εφαρμογή του `const_cast` απομακρύνει την `const` ιδιότητα και έτσι ο δείκτης `p1` μπορεί να αλλάξει την τιμή του `i`. Σημειώστε ότι αυτό εξακολουθεί να είναι αδύνατο για τον `p2`
- Άρα, το πρόγραμμα εμφανίζει `20`
- Σημειώστε ότι θα μπορούσαμε να μην χρησιμοποιήσουμε τον τελεστή `const_cast` και να κάνουμε μία προσαρμογή με τον παραδοσιακό τρόπο: `p1 = (int*)p2;`

Ο Τελεστής static_cast

- Συνήθως, ο τελεστής `static_cast` χρησιμοποιείται για την προσαρμογή ενός τύπου σε κάποιον άλλο σχετικό τύπο
- Για παράδειγμα, έναν τύπο δείκτη σε έναν άλλο σε μία ιεραρχία κλάσεων ή έναν αριθμητικό τύπο (π.χ. ακέραιο) σε έναν άλλο (π.χ. πραγματικό)
- Έχει την ίδια σύνταξη με τους υπόλοιπους τελεστές:

  `static_cast<όνομα_τύπου>(έκφραση);`
- Συνήθως, χρησιμοποιείται για μετατροπές αριθμητικών τύπων. Για παράδειγμα:

Παράδειγμα

```
#include <iostream>
int main()
{
    char c;
    int i = 97, j;
    double d = 3.9;

    c = static_cast<char>(i); // int σε char.
    j = static_cast<int>(d); // double σε int.
    std::cout << c << ' ' << j << '\n';
    return 0;
}
```

Το πρόγραμμα εμφανίζει a και 3

Ο Τελεστής reinterpret_cast

- Συνήθως, ο τελεστής `reinterpret_cast` χρησιμοποιείται για την προσαρμογή τύπων που δεν σχετίζονται μεταξύ τους
- Έχει την ίδια σύνταξη με τους υπόλοιπους τελεστές:

```
reinterpret_cast<όνομα_τύπου>(έκφραση);
```

- Ο τελεστής μπορεί να χρησιμοποιηθεί για την μετατροπή του τύπου ενός δείκτη σε οποιονδήποτε άλλο τύπο δείκτη. Επίσης, επιτρέπει την μετατροπή ενός ακέραιου τύπου σε τύπο δείκτη και το αντίστροφο. Για παράδειγμα:

```
int *p = reinterpret_cast<int*>(0xffcc);
```

- Με τον παραδοσιακό τρόπο, αυτή η μετατροπή γίνεται ως εξής:

```
int *p = (int*)(0xffcc);
```

- Γενικά, οι μετατροπές που γίνονται με αυτόν τον τελεστή είναι ενδεχομένως **μη ασφαλείς** και θα πρέπει να αποφεύγονται. Δηλαδή, μπορείτε να τον χρησιμοποιήσετε για να μετατρέψετε `float` σε `string*`, αλλά αυτό μπορεί να οδηγήσει σε επικίνδυνες καταστάσεις. Αποτελεί ευθύνη του προγραμματιστή να εξασφαλίσει ότι η μετατροπή δεν θα προκαλέσει προβλήματα. Για παράδειγμα, η αποαναφοροποίηση του δείκτη μπορεί να προκαλέσει την κατάρρευση του προγράμματος

Παρατηρήσεις

- Η C++ παρέχει τους τελεστές που είδαμε για να προσαρμόζουμε τύπους με μεγαλύτερη ασφάλεια έναντι του παραδοσιακού τρόπου
- Εκτός από μεγαλύτερη ασφάλεια, ένας δεύτερος λόγος χρήσης τους είναι ο εύκολος εντοπισμός των προσαρμογών μέσα στο πρόγραμμα
- Απλά, κάνουμε μία αναζήτηση με το όνομά τους και τους βρίσκουμε άμεσα. Αντίθετα, με τον παραδοσιακό τρόπο δεν μπορεί να γίνει μία παρόμοια αναζήτηση. Πρέπει να ψάξουμε γραμμή-γραμμή του κώδικα

Έξυπνοι Δείκτες

- Ένας έξυπνος δείκτης είναι ένα αντικείμενο το οποίο διευκολύνει τη διαχείριση δυναμικά δεσμευμένης μνήμης
- Συμπεριφέρεται όπως ένας συνηθισμένος δείκτης με τη διαφορά ότι μπορεί από μόνος του να απευθερώσει τη μνήμη στην οποία δείχνει
- Έτσι, δεν χρειάζεται να ανησυχούμε να μην ξεχάσουμε να απελευθερώσουμε τη δεσμευμένη μνήμη και να δημιουργήσουμε διαρροές μνήμης. Θα αποδεσμευτεί αυτόματα, όταν δεν χρειάζεται πλέον
- Για παράδειγμα, με ένα συνηθισμένο δείκτη, ο παρακάτω κώδικας μπορεί να δημιουργήσει διαρροή μνήμης:

```
void f()
{
        int *p = new int[100000];
        ...
        if(error_occurs)
                throw exception();
        ...
        delete[] p;
}
```

- Αν συμβεί κάποιο λάθος, θα δημιουργηθεί η εξαίρεση και η μνήμη δεν θα απελευθερωθεί. Άρα, θα έχουμε διαρροή μνήμης

Έξυπνοι Δείκτες

- Ωραία, υπάρχει άλλος τρόπος εκτός από το να "θυμηθούμε" να απελευθερώσουμε τη μνήμη, δηλαδή, υπάρχει τρόπος να εξασφαλίσουμε ότι η μνήμη θα απελευθερωθεί ακόμα κι αν ξεχάσουμε να το κάνουμε;
- Ναι, υπάρχει, με τους **έξυπνους** δείκτες. Αν ο δείκτης `p` ήταν αντικείμενο και είχε έναν αποδομητή, ο αποδομητής θα μπορούσε να απελευθερώσει τη μνήμη όταν το αντικείμενο θα καταστρεφόταν
- Αυτή είναι η βασική ιδέα των έξυπνων δεικτών, να χρησιμοποιούμε ένα αντικείμενο για να διαχειριστούμε τη μνήμη, ώστε να αποτρέψουμε διαρροές μνήμης
- Ουσιαστικά, ένας έξυπνος δείκτης είναι **αντικείμενο** μίας πρότυπης κλάσης, το οποίο απελευθερώνει τη μνήμη όταν καταστρέφεται
- Επομένως, δεν χρειάζεται να θυμόμαστε να απελευθερώσουμε τη μνήμη, θα το κάνει ο έξυπνος δείκτης για εμάς
- Η C++11 εισάγει τους τύπους `unique_ptr`, `shared_ptr` και `weak_ptr`

Παράδειγμα

```
#include <iostream>
#include <exception>
#include <memory>
using namespace std;

class C
{
public:
    double v;
    ~C() {cout << "End\n";}
};

double f(double a, int b);

int main()
{
    double i;
    int j;

    cout << "Enter two numbers: ";
    cin >> i >> j;
    try
    {
        cout << f(i, j) << '\n';
    }
    catch(exception& e)
    {
        cout << e.what() << '\n';
    }
    return 0;
}
```

```
double f(double a, int b)
{
    unique_ptr<C> p(new C);
    if(b == 0)
        throw exception();
    p->v = a/b;
    return (*p).v;
}
```

Παράδειγμα

- Για να δημιουργήσουμε ένα `unique_ptr` αντικείμενο χρησιμοποιούμε την πρότυπη σύνταξη. Όπως είπαμε, οι έξυπνοι δείκτες είναι πρότυπες κλάσεις
- Μέσα στις <> εισάγουμε τον τύπο στον οποίο θα δείξει ο έξυπνος δείκτης
- Όταν κατασκευάζουμε έναν έξυπνο δείκτη μπορούμε να τον αρχικοποιήσουμε με τον δείκτη που επιστρέφει ο τελεστής `new`
- Αν το `b` είναι `0`, η `f()` δημιουργεί εξαίρεση. Η `unique_ptr` κλάση υπερφορτώνει τους τελεστές `*` και `->`, ώστε να μπορούμε να χρησιμοποιήσουμε το αντικείμενο `p`, όπως και ένα συνηθισμένο δείκτη
- Παρατηρήστε ότι δεν υπάρχει `delete` για την αποδέσμευση της μνήμης. Η δεσμευμένη μνήμη θα αποδεσμευτεί όταν το αντικείμενο `p` καταστραφεί
- Αυτό θα συμβεί όταν το `p` βγει εκτός εμβέλειας, δηλαδή, όταν τερματιστεί η συνάρτηση, είτε λόγω της εξαίρεσης είτε όχι
- Δηλαδή, η `unique_ptr` εξασφαλίζει ότι η μνήμη θα αποδεσμευτεί ακόμα και αν γίνει μία βίαιη έξοδος από το τμήμα. Τότε, το `p` θα καταστραφεί και ο αποδομητής του θα αποδεσμεύσει την μνήμη
- Έτσι, καλείται ο αποδομητής της `C` και το πρόγραμμα εμφανίζει `End` και στις δύο περιπτώσεις, είτε προκληθεί η εξαίρεση είτε όχι

C++: Από τη Θεωρία στην Εφαρμογή

Κεφάλαιο 26ο

Εισαγωγή στην Καθιερωμένη Βιβλιοθήκη Προτύπων

Καθιερωμένη Βιβλιοθήκη Προτύπων

- Σε αυτήν την ενότητα θα κάνουμε μία σύντομη εισαγωγή στην καθιερωμένη βιβλιοθήκη προτύπων (STL - Standard Template Library) και στις δυνατότητες που παρέχει στον προγραμματιστή
- Η STL αποτελεί τμήμα της C++ βιβλιοθήκης. Σχεδιάστηκε από τον Alex Stepanov και περιέχει μία μεγάλη συλλογή από πρότυπους αλγορίθμους, επαναλήπτες, αποδέκτες και αντικείμενα συναρτήσεων
- Όλα αυτά έχουν σχεδιαστεί με έναν ενιαίο τρόπο που βασίζεται στις αρχές του **γενικού** προγραμματισμού
- Οι έτοιμες κλάσεις και αλγόριθμοι που παρέχει μπορούν να χρησιμοποιηθούν σε ένα ευρύ φάσμα εφαρμογών και να απλουστεύσουν την εργασία του προγραμματιστή

Αλγόριθμοι

- Η STL παρέχει ένα πλήθος αλγορίθμων που μπορούν να εφαρμοστούν σε αποδέκτες
- Η σχεδιαστική φιλοσοφία είναι να παρέχουν ένα **γενικό** τρόπο προσπέλασης των στοιχείων των αποδεκτών, ανεξάρτητα από τον τύπο του αποδέκτη ή τον τρόπο που έχουν αποθηκευτεί σε αυτόν
- Για αυτό το λόγο, οι STL αλγόριθμοι έχουν υλοποιηθεί ως **πρότυπες** συναρτήσεις ώστε να λειτουργούν με έναν απλό, κομψό, και γενικό τρόπο, που να μην εξαρτάται από τον αποδέκτη στον οποίο εφαρμόζονται
- Για παράδειγμα, η ίδια `find()` μπορεί να χρησιμοποιηθεί για να βρούμε κάποια τιμή σε ένα `vector` ή `set` αποδέκτη ή για να βρούμε κάποια τιμή σε ένα `vector` που περιέχει ακεραίους και σε κάποιο άλλο που περιέχει `string` στοιχεία

Αλγόριθμοι

- Η προσέγγιση της STL είναι αυτές οι συναρτήσεις να μην ορίζονται σαν μέλη των αποδεκτών, αλλά **εξωτερικά**
- Δηλαδή, αντί κάθε αποδέκτης να περιέχει τη δική του `find()`, ορίζεται **μία** μόνο εξωτερική `find()` η οποία μπορεί να χρησιμοποιηθεί από όλους τους αποδέκτες
- Όπως θα δούμε αργότερα, για να γίνει αυτό, οι STL αλγόριθμοι χρησιμοποιούν **επαναλήπτες** για να διασχίσουν τον αποδέκτη
- Με αυτή την πρακτική δεν αυξάνεται η πολυπλοκότητα του κάθε αποδέκτη, αλλά και αποφεύγεται η επανάληψη του ίδιου κώδικα
- Ωστόσο, υπάρχουν περιπτώσεις όπου η STL ορίζει μία συνάρτηση **μέσα** σε έναν αποδέκτη που να επιτελεί την **ίδια** εργασία με μία μη-μέλος συνάρτηση
- Ο λόγος είναι ότι μία συνάρτηση μέλος για ένα συγκεκριμένο αποδέκτη μπορεί να είναι πιο αποδοτική από τη γενική συνάρτηση
- Γενικά, όταν ένας αποδέκτης παρέχει μία συνάρτηση μέλος που να είναι λειτουργικά ισοδύναμη με μία γενική συνάρτηση, θα είναι καλύτερα να χρησιμοποιήσετε την πρώτη
- Ας δούμε κάποια παραδείγματα συναρτήσεων

Η Συνάρτηση `for_each()`

- Η συνάρτηση `for_each()` λειτουργεί με κάθε αποδέκτη
- Δέχεται τρεις παραμέτρους. Η πρώτη είναι επαναλήπτης εισόδου που δηλώνει την αρχή του διαστήματος (π.χ. `first`), η δεύτερη είναι επαναλήπτης εισόδου που δηλώνει το τέλος χωρίς αυτό να περιλαμβάνεται (π.χ. `last`) και η τρίτη ένας δείκτης σε μία συνάρτηση ή, όπως θα δούμε στη συνέχεια, ένα αντικείμενο συνάρτησης
- Η συνάρτηση θα πρέπει να δέχεται μία παράμετρο με τύπο ίδιο με τον τύπο του αποδέκτη
- Η `for_each()` εφαρμόζει την συνάρτηση στην οποία δείχνει ο δείκτης σε όλα τα στοιχεία του αποδέκτη μέσα στο εύρος, δηλαδή, στο [`first`, `last`). Ας δούμε ένα παράδειγμα χρήσης της:

Παράδειγμα

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

void show(int k);

int main()
{
    std::vector<int> v(3, 30); /* Δημιουργία διανύσματος τριών ακεραίων στοιχείων
    με αρχική τιμή 30. */
    v[0] = 20;
    v[1] = 10;
    for_each(v.begin(), v.end(), show);
    return 0;
}

void show(int k)
{
    std::cout << k << '\n';
}
```

- Κάθε αποδέκτης παρέχει την `begin()` συνάρτηση, η οποία επιστρέφει έναν επαναλήπτη που δείχνει στο πρώτο στοιχείο του και την `end()` συνάρτηση, η οποία επιστρέφει έναν επαναλήπτη που δείχνει στην επόμενη θέση μετά το τελευταίο στοιχείο του. Όπως θα δούμε στη συνέχεια, οι επαναλήπτες αποτελούν γενικεύσεις δεικτών
- Με την `for_each()` η `show()` καλείται τρείς φορές, μία για το κάθε στοιχείο του αποδέκτη `v`. Επομένως, το πρόγραμμα θα εμφανίσει `20`, `10` και `30`

Η Συνάρτηση `find()`

- Η συνάρτηση `find()` λειτουργεί με κάθε αποδέκτη και χρησιμοποιείται για την αναζήτηση μίας τιμής σε έναν αποδέκτη
- Δέχεται τρεις παραμέτρους. Οι δύο πρώτες παράμετροι είναι επαναλήπτες εισόδου (π.χ. `first`, `last`) που δηλώνουν το εύρος της αναζήτησης, το οποίο είναι το [`first`, `last`). Η τρίτη είναι η τιμή αναζήτησης
- Αν η τιμή βρεθεί, η `find()` επιστρέφει έναν επαναλήπτη που δείχνει στη θέση της πρώτης εμφάνισής της. Αν όχι, επιστρέφει την τιμή του δεύτερου ορίσματος (π.χ. `last`)
- Επειδή, όπως θα δούμε στη συνέχεια, ένας δείκτης λειτουργεί και σαν επαναλήπτης, ας δούμε ένα παράδειγμα χρήσης της με πίνακα και αποδέκτη:

Παράδειγμα

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
    int *p, arr[] = {10, 20, 30, 40, 50};

    p = find(arr, arr+5, 30);
    if(p != arr+5)
            cout << "Element " << *p << " found in position " << p-arr+1 << '\n';
    else
            cout << "Element not found\n";

    vector<int> v(arr, arr+5); /* Δημιουργία διανύσματος με τα στοιχεία του
    πίνακα. */
    vector<int>::iterator it;
    it = find(v.begin(), v.end(), 30);
    if(it != v.end())
            cout << "Element " << *it << " found in position " << it-v.begin()+1
    << '\n';
    else
            cout << "Element not found\n";
    return 0;
}
```

Επαναλήπτες

- Ένας επαναλήπτης είναι ένα αντικείμενο ή ένας απλός δείκτης που δείχνει στα στοιχεία του αποδέκτη και μπορούμε να τον χρησιμοποιήσουμε για να προσπελάσουμε τα στοιχεία του
- Με τους επαναλήπτες μπορούμε να προσπελάσουμε στοιχεία διαφορετικών αποδεκτών με τον ίδιο τρόπο, ανεξάρτητα από τους αποθηκευμένους τύπους δεδομένων
- Έτσι, οι ίδιοι STL αλγόριθμοι μπορούν να εφαρμοστούν σε διαφορετικούς αποδέκτες με ενιαίο τρόπο. Για παράδειγμα, έστω ότι ψάχνουμε μία τιμή σε ένα `vector<int>` αντικείμενο:

```
vector<int> vec;
...
for(i = 0; i < vec.size(); i++)
        if(vec[i] == val) // Ψάχνουμε την τιμή val.
...
```

- Έστω τώρα ότι ψάχνουμε μία τιμή σε ένα `list<int>` αντικείμενο, το οποίο είναι μία συνδεδεμένη λίστα με ακέραιους αριθμούς:

```
list<int> lst;
...
for(i = 0; i < lst.size(); i++)
        if(lst[i] == val) // Δεν επιτρέπεται να γράψουμε lst[i].
...
```

- Όπως φαίνεται από το σχόλιο, αν και είναι λογικό να χρησιμοποιήσουμε σημειογραφία πίνακα, δεν επιτρέπεται γιατί ο αποδέκτης `list` δεν παρέχει συνάρτηση υπερφόρτωσης του `[]`

Επαναλήπτες

- Ωστόσο, με τους επαναλήπτες μπορούμε να γράψουμε τον κώδικα αυτής της αναζήτησης με τον ίδιο τρόπο και για τους δύο αποδέκτες
- Η χρήση των επαναληπτών είναι παρόμοια με τους δείκτες. Για παράδειγμα, έστω ότι το όνομα του αντικειμένου (vector ή list) είναι c. Στην περίπτωση του vector<int> γράφουμε:

```
vector<int>::iterator it;
for(it = c.begin(); it != c.end(); ++it)
        if(*it == val)
```

- Ενώ, στην περίπτωση του list<int> γράφουμε:

```
list<int>::iterator it;
for(it = c.begin(); it != c.end(); ++it)
        if(*it == val)
```

- Το μόνο που αλλάζει στον κώδικα είναι η δήλωση του επαναλήπτη, ανάλογα με τον τύπο του αποδέκτη. Χρησιμοποιούμε τη λέξη auto και ο κώδικας αναζήτησης γίνεται κοινός:

```
for(auto it = c.begin(); it != c.end(); ++it)
        if(*it == val)
```

- Δηλαδή, με τη χρήση επαναληπτών μπορούμε να μετακινηθούμε μέσα σε διαφορετικούς αποδέκτες με τον ίδιο τρόπο (π.χ. ++it) και να προσπελάσουμε τα στοιχεία τους με τον ίδιο τρόπο (π.χ. *it), ανεξάρτητα από τον τύπο τους και από το αν αυτά τα στοιχεία βρίσκονται σε διαδοχικές θέσεις μνήμης (π.χ. στο vector) ή σε τυχαίες θέσεις (π.χ. στο list)

Επαναλήπτες

- Αυτή είναι και η φιλοσοφία της **STL**, δηλαδή, εισάγει την έννοια των επαναληπτών και τους χρησιμοποιεί, ώστε η υλοποίηση της κάθε συνάρτησης να γίνεται με **γενικό** τρόπο, η οποία να μην εξαρτάται από τον τύπο του αποδέκτη
- Ουσιαστικά, οι επαναλήπτες αποτελούν το σύνδεσμο μεταξύ των αποδεκτών και των αλγορίθμων. Έτσι, μπορούμε να έχουμε μόνο μία `find()` για **όλους** τους αποδέκτες και όχι μία ξεχωριστή έκδοση για κάθε αποδέκτη
- Σημειώστε ότι αν και ένας δείκτης αποτελεί μία μορφή επαναλήπτη, **δεν ισχύει** και το αντίστροφο, δηλαδή, ένας επαναλήπτης δεν λειτουργεί απαραίτητα όπως ένας δείκτης
- Όμως, ένας δείκτης καλύπτει τις απαιτήσεις **οποιουδήποτε** επαναλήπτη

Αποδέκτες

- Ένας αποδέκτης (container) είναι μία κλάση, η οποία περιέχει ένα πλήθος αντικειμένων τα οποία είναι τα στοιχεία της
- Κάθε αποδέκτης διαχειρίζεται τον αποθηκευτικό χώρο για τα στοιχεία του και παρέχει συναρτήσεις για την πρόσβαση σε αυτά
- Υλοποιούνται ως πρότυπες κλάσεις, γεγονός που παρέχει μεγάλη ευελιξία στον τύπο των στοιχείων που θα αποθηκευτούν
- Τα στοιχεία μπορεί να είναι αντικείμενα κλάσεων ή βασικοί τύποι δεδομένων. Για παράδειγμα, `vector<Student>` και `vector<int>`
- Γενικά, η απόφαση για το ποιον τύπο αποδέκτη να χρησιμοποιήσουμε σε μία εφαρμογή δεν εξαρτάται μόνο από τις απαιτήσεις της εφαρμογής, αλλά και από την απόδοση των λειτουργιών που παρέχει ο υποψήφιος αποδέκτης
- Δεν υπάρχει η έννοια του «καλύτερου» αποδέκτη, εξαρτάται από τις απαιτήσεις της εφαρμογής. Για παράδειγμα, ο αποδέκτης `vector` παρέχει ταχύτητα στην προσπέλαση των στοιχείων του, όχι όμως στην εισαγωγή στοιχείων, καθώς τα στοιχεία μετά τη θέση εισαγωγής θα πρέπει να μετακινηθούν για να δημιουργηθεί χώρος για τα νέα στοιχεία
- Οι αποδέκτες χωρίζονται σε τρεις βασικές κατηγορίες, στους ακολουθιακούς αποδέκτες, στους προσαρμογείς αποδεκτών και στους συσχετιστικούς αποδέκτες

Ακολουθιακοί Αποδέκτες

- Οι ακολουθιακοί αποδέκτες (sequence containers) περιέχουν αντικείμενα του ίδιου τύπου, τα οποία είναι αποθηκευμένα διαδοχικά
- Οι πρότυπες κλάσεις `vector`, `deque`, `list` αποτελούν παραδείγματα ακολουθιακών αποδεκτών. Η C++11 προσθέτει και τις κλάσεις `array` και `forward_list`
- Για την κλάση `vector` έχουμε ήδη μιλήσει. Η πρότυπη κλάση `deque` (double-ended queue) υποστηρίζει την γρήγορη προσθήκη/διαγραφή στοιχείων στην αρχή και στο τέλος της
- Η `array` αποτελεί μία εύκολη στη χρήση και ασφαλέστερη επιλογή έναντι του συνηθισμένου πίνακα. Όπως και ο πίνακας, η `array` είναι σταθερού μήκους που καθορίζεται κατά τη μεταγλώττιση, δηλαδή, δεν επιτρέπεται η δυναμική αλλαγή του μεγέθους της
- Η `forward_list` υλοποιεί μία απλή συνδεδεμένη λίστα. Όλοι αυτοί οι αποδέκτες, εκτός από την `array`, υποστηρίζουν την προσθήκη/διαγραφή στοιχείων κατά την εκτέλεση του προγράμματος, δηλαδή, το μέγεθος της δεσμευμένης μνήμης αλλάζει δυναμικά

Ακολουθιακοί Αποδέκτες

- Όπως είπαμε, η απόφαση για το ποια κλάση να χρησιμοποιήσουμε εξαρτάται από τις απαιτήσεις της εφαρμογής
- Για παράδειγμα, αν η εφαρμογή απαιτεί την εισαγωγή/διαγραφή στοιχείων μόνο στην αρχή ή στο τέλος, η καλύτερη επιλογή είναι η `deque`. Αν η εφαρμογή απαιτεί πολύ συχνότερα τυχαία προσπέλαση στοιχείων από ότι εισαγωγή/διαγραφή στοιχείων, η καλύτερη επιλογή είναι η `vector`
- Γενικά, η κλάση `vector` είναι συνήθως η καλύτερη επιλογή. Ας δούμε συνοπτικά την `list`

Η Πρότυπη Κλάση `list`

- Η πρότυπη κλάση `list` υλοποιεί μία διπλά συνδεδεμένη λίστα. Κάθε στοιχείο της λίστας, εκτός από το πρώτο και το τελευταίο, συνδέεται με το προηγούμενο και το επόμενο στοιχείο, έτσι ώστε η λίστα να μπορεί να διασχιστεί και από τις δύο κατευθύνσεις
- Αντίθετα με τη `vector`, η `list` δεν επιτρέπει τυχαία προσπέλαση των στοιχείων, αλλά μόνο σειριακή
- Δηλαδή, δεν μπορούμε να χρησιμοποιήσουμε σημειογραφία πίνακα και να γράψουμε κάτι παρόμοιο με `lst[i]`, όπου το `lst` είναι αντικείμενο
- Συγκριτικά, το πλεονέκτημα της `vector` είναι ότι παρέχει άμεση πρόσβαση σε οποιοδήποτε στοιχείο, ενώ η `list` ταχύτητα στην προσθήκη/διαγραφή στοιχείων
- Για παράδειγμα, για να προσπελάσουμε το δέκατο στοιχείο μίας `list` θα πρέπει να περάσουμε από όλα τα προηγούμενα στοιχεία μέσω των μελών-δεικτών για να φτάσουμε σε αυτό
- Αντίθετα, στο `vector`, επειδή τα στοιχεία βρίσκονται σε συνεχόμενες θέσεις μνήμης, η προσπέλαση είναι γρήγορη
- Από την άλλη, το πλεονέκτημα της `list` είναι ότι το κόστος προσθήκης/διαγραφής ενός στοιχείου σε οποιαδήποτε θέση είναι σταθερό, απλά αλλάζουν οι τιμές των αντίστοιχων δεικτών
- Αντίθετα, στο `vector`, αυτές οι λειτουργίες απαιτούν περισσότερο χρόνο, αφού τα στοιχεία πρέπει να αναδιαταχθούν. Για παράδειγμα, η εισαγωγή ενός νέου στοιχείου απαιτεί όλα τα στοιχεία μετά το σημείο εισαγωγής να μετακινηθούν μία θέση δεξιά για να δημιουργηθεί χώρος για το νέο στοιχείο

Παράδειγμα

```
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

void show(int k);

int main()
{
	int arr[5] = {2, 3, 2, 1, 3};
	list<int> lst(3, 1); // Λίστα τριών ακεραίων με τιμή 1.
	/* Αν θέλαμε να δηλώσουμε μία iterator μεταβλητή θα γράφαμε list<int>::iterator it; Για απλότητα,
	χρησιμοποιούμε τη λέξη auto. */
	lst.insert(lst.begin(), arr, arr+5);
	cout << "List after insert: ";
	for(auto it = lst.begin(); it != lst.end(); ++it)
		cout << *it << ' ';
	lst.remove(2);
	cout << "\nList after remove: ";
	for_each(lst.begin(), lst.end(), show);

	lst.unique();
	cout << "\nList after unique: ";
	for_each(lst.begin(), lst.end(), show);

	lst.pop_front();
	// Εισαγωγή κάποιων στοιχείων.
	lst.push_back(6);
	lst.push_back(5);
	lst.push_front(7);
	lst.sort();
	cout << "\nList after sorting: ";
	for_each(lst.begin(), lst.end(), show);
	return 0;
}

void show(int k)
{
	std::cout << k << '\n';
}
```

Παράδειγμα

- Η `insert()` εισάγει στοιχεία στη θέση της λίστας που υποδεικνύει ο επαναλήπτης
- Η `remove()` απομακρύνει όλα τα στοιχεία που έχουν ίδια τιμή με την τιμή του ορίσματος
- Η `unique()` απομακρύνει όλα τα διαδοχικά στοιχεία με την ίδια τιμή και διατηρεί μόνο το πρώτο από κάθε ομάδα ίδιων στοιχείων
- Η `pop_front()` απομακρύνει το πρώτο στοιχείο της λίστας
- Η `push_front()` προσθέτει το στοιχείο στην αρχή της λίστας, ενώ η `push_back()` στο τέλος της
- Η `sort()` ταξινομεί τη λίστα. Αν χρησιμοποιήσουμε πρώτα την `sort()` και μετά την `unique()` κάθε στοιχείο θα εμφανίζεται μόνο μία φορά
- Το πρόγραμμα χρησιμοποιεί δύο τρόπους για την εμφάνιση των στοιχείων της λίστας, με επαναλήπτη και με την `for_each()`
- Τελικά, το πρόγραμμα εμφανίζει:

```
List after insert: 2  3  2  1  3  1  1  1
List after remove: 3  1  3  1  1  1
List after unique: 3  1  3  1
List after sorting: 1  1  3  5  6  7
```

Προσαρμογείς Αποδεκτών

- Οι προσαρμογείς αποδεκτών (container adaptors) υλοποιούνται με βάση άλλους αποδέκτες και προσαρμόζουν τη λειτούργικότητα τους στις δικές τους απαιτήσεις
- Για παράδειγμα, ο αποδέκτης `stack`, αν και εξ'ορισμού βασίζεται στην `deque`, επιτρέπει την εισαγωγή και διαγραφή στοιχείων μόνο στην κορυφή της στοίβας
- Δεν διαθέτουν επαναλήπτες και δεν υποστηρίζουν προσπέλαση με αριθμοδείκτες
- Οι προσαρμογείς αποδεκτών είναι οι πρότυπες κλάσεις `queue`, `priority_queue` και `stack`. Ας δούμε περιληπτικά την κλάση `stack`

Η Πρότυπη Κλάση stack

- Η πρότυπη κλάση **stack** υλοποιεί μία LIFO (Last In First Out) στοίβα, όπου τα στοιχεία εισάγονται και εξάγονται μόνο από το τέλος του αποδέκτη
- Η υλοποίηση υποστηρίζει τη βασική λειτουργικότητα μίας στοίβας. Έτσι, μπορούμε να προσθέσουμε ένα στοιχείο στην κορυφή της στοίβας, να αφαιρέσουμε το στοιχείο από την κορυφή, να βρούμε την τιμή του στοιχείου της κορυφής, να ελέγξουμε αν η στοίβα είναι άδεια, και να βρούμε τον αριθμό των στοιχείων που περιέχει
- Δεν υποστηρίζεται τυχαία προσπέλαση των στοιχείων της, ούτε και η διάσχισή της. Ας δούμε ένα παράδειγμα:

```
#include <iostream>
#include <stack>
int main()
{
        std::stack<int> s;

        s.push(10);
        s.push(20);
        s.push(30);
        std::cout << "Top: " << s.top();
        s.pop();
        std::cout << " Size: " << s.size();
        return 0;
}
```

- Η **push()** προσθέτει το στοιχείο στην κορυφή της στοίβας. Η **top()** επιστρέφει την τιμή του στοιχείου στην κορυφή. Η **pop()** απομακρύνει την κορυφή και η **size()** επιστρέφει τον αριθμό των στοιχείων στη στοίβα. Τελικά, το πρόγραμμα εμφανίζει:
Top: 30 Size: 2

Συσχετιστικοί Αποδέκτες

- Οι συσχετιστικοί αποδέκτες (associative containers) συσχετίζουν κάθε στοιχείο με ένα **κλειδί** και χρησιμοποιούν το κλειδί για να **εντοπίσουν** το στοιχείο στον αποδέκτη
- Δηλαδή, οι συσχετιστικοί αποδέκτες έχουν σχεδιαστεί έτσι ώστε να εντοπίζονται και να προσπελαύνονται τα στοιχεία τους με βάση το **κλειδί** τους, σε αντίθεση με τους ακολουθιακούς αποδέκτες που έχουν σχεδιαστεί για να εντοπίζονται και να προσπελαύνονται τα στοιχεία τους με βάση τη **θέση** τους
- Για παράδειγμα, ας υποθέσουμε ότι κάθε στοιχείο είναι μία δομή που αντιπροσωπεύει ένα φοιτητή με πεδία όπως το όνομα, ο κωδικός, ο βαθμός, η διεύθυνση, στοιχεία επικοινωνίας, κ.α.. Το κλειδί μπορεί να είναι ο κωδικός του φοιτητή, ο οποίος είναι μοναδικός. Ο αποδέκτης χρησιμοποιεί αυτό το κλειδί για να αποθηκεύσει και να εντοπίζει το κάθε στοιχείο
- Όταν προσθέτουμε στοιχεία δεν μπορούμε να καθορίσουμε τη θέση στην οποία θα προστεθούν. Ο αποδέκτης υποστηρίζει αλγόριθμο, ο οποίος θα προσδιορίσει τη θέση εισαγωγής
- Ο σύνηθης τρόπος υλοποίησης ενός συσχετιστικού αποδέκτη είναι με τη μορφή δυαδικού δέντρου αναζήτησης. Ο αποδέκτης παρέχει ταχύτατη πρόσβαση στα στοιχεία του

Συσχετιστικοί Αποδέκτες

- Η STL παρέχει τους αποδέκτες set, multiset, map και multimap, οι οποίοι αποθηκεύουν τα στοιχεία σε διάταξη
- Στον αποδέκτη set η τιμή του κάθε στοιχείου ταυτίζεται με το κλειδί του και το κάθε κλειδί είναι μοναδικό, ενώ ο αποδέκτης multiset μπορεί να περιέχει στοιχεία με το ίδιο κλειδί
- Στον αποδέκτη map το κλειδί του κάθε στοιχείου συνδέεται με αντίστοιχα δεδομένα. Το κάθε κλειδί είναι μοναδικό, ενώ ο αποδέκτης multimap μπορεί να περιέχει στοιχεία με το ίδιο κλειδί
- Επίσης, η STL παρέχει τους αποδέκτες unordered_set, unordered_multiset, unordered_map και unordered_multimap, οι οποίοι δεν αποθηκεύουν τα στοιχεία σε διάταξη. Αυτοί οι αποδέκτες είναι χρήσιμοι σε εφαρμογές που δεν χρειάζεται τα στοιχεία να αποθηκεύονται με κάποια σειρά διάταξης
- Ας δούμε δύο παραδείγματα με τους αποδέκτες set και map

Η Πρότυπη Κλάση `set`

- Ο αποδέκτης `set` περιέχει στοιχεία, όπου η τιμή του κάθε στοιχείου είναι και το κλειδί του
- Είναι χρήσιμος σε εφαρμογές όπου απλά θέλουμε να ελέγξουμε γρήγορα αν μία τιμή περιέχεται στον αποδέκτη. Οι τιμές (κλειδιά) των στοιχείων είναι μοναδικές στο `set`, δηλαδή, δεν υπάρχουν επαναλήψεις του ίδιου κλειδιού
- Τα στοιχεία ταξινομούνται με βάση το κλειδί τους. Εξ'ορισμού, ο αποδέκτης χρησιμοποιεί τον τελεστή < για την ταξινόμηση των στοιχείων σε αύξουσα σειρά, ωστόσο, μπορούμε να καθορίσουμε τον δικό μας τρόπο ταξινόμησης
- Η αναζήτηση ή εισαγωγή ενός νέου στοιχείου εκτελείται πολύ γρήγορα
- Η τιμή ενός στοιχείου που έχει εισαχθεί στον αποδέκτη δεν μπορεί να τροποποιηθεί, μπορεί μόνο να διαγραφεί. Ας δούμε ένα παράδειγμα:

Παράδειγμα

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

void show(const string& s);

int main()
{
     set<string> s = {"car", "the", "get", "sea", "sky", "the"};
     set<string>::iterator it;

     cout << "Set after init: ";
     for(it = s.begin(); it != s.end(); ++it)
               cout << *it << ' ';

     // Εκτέλεση κάποιων λειτουργιών.
     s.insert("new");
     cout << "\nSet after insert: ";
     for_each(s.begin(), s.end(), show);

     it = s.find("sky");
     s.erase(it);
     cout << "\nSet after erase: ";
     for_each(s.begin(), s.end(), show);
     return 0;
}

void show(const string& s)
{
     cout << s << ' ';
}
```

Παράδειγμα

- Όταν δημιουργούμε ένα `set` αντικείμενο καθορίζουμε τον τύπο του κλειδιού που περιέχει (π.χ. `string`). Η `insert()` εισάγει το όρισμα στον αποδέκτη. Η `find()` επιστρέφει έναν επαναλήπτη, ο οποίος δείχνει στη θέση που είναι αποθηκευμένο το όρισμα, αν υπάρχει. Η `erase()` διαγράφει το στοιχείο που υποδεικνύει ο επαναλήπτης. Τελικά, το πρόγραμμα εμφανίζει:

```
Set after init: car  get  sea  sky  the
Set after insert: car  get  new sea  sky  the
Set after erase: car  get  new sea  the
```

- Όπως φαίνεται, αν και το `the` περιέχεται δύο φορές στη λίστα, αποθηκεύεται μόνο μία φορά στον αποδέκτη, αφού κάθε κλειδί πρέπει να είναι μοναδικό
- Επίσης, η έξοδος του προγράμματος επιβεβαιώνει ότι τα στοιχεία αποθηκεύονται με αύξουσα διάταξη.

Η Πρότυπη Κλάση map

- Ο αποδέκτης map περιέχει στοιχεία τα οποία σχηματίζονται με τον συνδυασμό ενός κλειδιού και αντίστοιχων δεδομένων
- Είναι χρήσιμος σε εφαρμογές όπου μία τιμή-κλειδί (π.χ. κωδικός φοιτητή) σχετίζεται με ένα σύνολο πληροφοριών (π.χ. στοιχεία φοιτητή)
- Με δεδομένη μία τιμή κλειδιού, μπορούμε γρήγορα να αποθηκεύσουμε ή να ανακτήσουμε τη σχετική πληροφορία
- Όπως και στον set αποδέκτη, κάθε στοιχείο σε ένα map αναγνωρίζεται μοναδικά από την τιμή του κλειδιού, τα κλειδιά είναι μοναδικά και τα στοιχεία ταξινομούνται με βάση το κλειδί τους
- Όταν δημιουργούμε ένα map αντικείμενο καθορίζουμε τον τύπο του κλειδιού και τον τύπο των δεδομένων που περιέχει
- Για παράδειγμα, η δήλωση: `map<int, string> m;` δημιουργεί ένα map αντικείμενο όπου ο τύπος του κλειδιού είναι int και ο τύπος των δεδομένων string
- Ένα τρίτο όρισμα μπορεί να προστεθεί, το οποίο να δηλώνει τη συνάρτηση ταξινόμησης που θα χρησιμοποιηθεί. Η εξ'ορισμού διάταξη είναι αύξουσα

Παράδειγμα

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

int main()
{
    map<int, string> m;
    map<int, string>::iterator it;
    m[50] = "one";
    m[40] = "two";
    m[30] = "three";
    m[20] = "four";

    it = m.find(40); /* Θα μπορούσαμε να δηλώσουμε εδώ τον it επαναλήπτη και να
    γραψουμε auto it = m.find(40); */
    if(it != m.end())
            cout << "Find: " << it->first << ' ' << it->second << '\n';
    else
            cout << "Not found\n";
    m.erase(30);
    cout << "Map after erase: ";
    for(it = m.begin(); it != m.end(); ++it)
            cout << it->first << ' ' << it->second << ' ';
    return 0;
}
```

Η **find()** επιστρέφει έναν επαναλήπτη που δείχνει στη θέση που είναι αποθηκευμένο το όρισμα-κλειδί, αν υπάρχει. Η **erase()** διαγράφει το στοιχείο που έχει την ίδια τιμή κλειδιού με το όρισμα, αν υπάρχει. Τελικά, το πρόγραμμα εμφανίζει:

```
Find: 40  two
Map after erase: 20  four  40 two  50  one
```

Αντικείμενα Συναρτήσεων

- Ένα αντικείμενο συνάρτησης (function object ή functor) είναι ένα αντικείμενο στο οποίο μπορεί να εφαρμοστεί η σύνταξη `()`, παρόμοια με μία συνάρτηση
- Για να επιτραπεί αυτή η σύνταξη, πρέπει η κλάση να υπερφορτώνει τον τελεστή `()` ορίζοντας τη συνάρτηση `operator()`. Ας δούμε ένα παράδειγμα:

```
class T
{
private:
        int a;
public:
        int operator()(int v) {a = v; return a;}
};
int main()
{
        T t;
        int i = t(20); // Το i γίνεται 20.
        ...
}
```

- Αν και το `t` είναι αντικείμενο, μπορούμε να «καλέσουμε» το αντικείμενο, όπως καλούμε και μία συνάρτηση
- Επειδή ένα τέτοιο αντικειμένο συμπεριφέρεται όπως μία συνάρτηση ονομάζεται αντικείμενο συνάρτησης
- Η έκφραση `t(20)` προκαλεί την κλήση της `operator()` συνάρτησης, η οποία επιστρέφει μία ακέραια τιμή

Λάμδα Συναρτήσεις

- Ένα από τα πιο σημαντικά χαρακτηριστικά που εισήγαγε η C++11 είναι η δυνατότητα δημιουργίας **λάμδα** (lambda) συναρτήσεων
- Η συνήθης χρήση των λάμδα συναρτήσεων είναι ως όρισμα σε STL συναρτήσεις που δέχονται ως όρισμα ένα δείκτη σε συνάρτηση ή αντικείμενο συνάρτησης
- Μπορούμε να θεωρήσουμε ότι μία λάμδα συνάρτηση είναι μία **ανώνυμη** inline συνάρτηση, η οποία ορίζεται στο σημείο όπου θέλουμε να τη χρησιμοποιήσουμε
- Έτσι, δεν χρειάζεται να ορίζουμε ξεχωριστές συναρτήσεις ή αντικείμενα συναρτήσεων. Για παράδειγμα:

Παράδειγμα

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

int main()
{
    int i = 6, j = 3;
    string tmp("test");
    vector<int> v = {5, 3, 8, 12, 1};

    cout << "Show less than 6: ";
    for_each(v.begin(), v.end(),
             [i](int a) {if(a < i) cout << a << ' ';});

    cout << "\nShow less than 3: ";
    for_each(v.begin(), v.end(),
             [&j](int a) {if(a < j) cout << a << ' ';});

    vector<string> s{"one", "two", "three"};
    cout << "\nShow less strings: ";
    for_each(s.begin(), s.end(),
             [&tmp](const string& a) {if(a < tmp) cout << a << ' ';});
    return 0;
}
```

Όπως φαίνεται, μεταβιβάζουμε τη λάμδα συνάρτηση ως τρίτο όρισμα. Η έξοδος του προγράμματος είναι:

```
Show less than 6: 5  3  1
Show less than 3: 1
Show less strings: one
```

Σύνταξη Λάμδα Συνάρτησης

- Ας εξηγήσουμε περιληπτικά τη σύνταξη της λάμδα συνάρτησης. Στην πιο απλή μορφή της ορίζεται ως εξής:

```
[τμήμα_σύλληψης](λίστα_παραμέτρων) -> τύπος_επιστροφής
{
        σώμα_λάμδα;
}
```

- Η λίστα παραμέτρων και ο επιθεματικός τύπος επιστροφής με την `->` σύνταξη είναι προαιρετικά μέρη
- Μία λάμδα συνάρτηση αρχίζει με το τμήμα σύλληψης `[]`. Αυτό το τμήμα καθορίζει τις μεταβλητές που συλλαμβάνονται (π.χ. `i`) και για το αν συλλαμβάνονται μέσω αναφοράς ή μέσω τιμής
- Οι μεταβλητές πρέπει να βρίσκονται στην περικλείουσα εμβέλεια, όπου εμφανίζεται ο ορισμός της λάμδα

Σύνταξη Λάμδα Συνάρτησης

- Μία άδεια λίστα σύλληψης (π.χ. `[]`) υποδεικνύει ότι δεν συλλαμβάνεται τίποτα. Σε αυτή την περίπτωση, η λάμδα μπορεί να λειτουργήσει μόνο με τα ορίσματά της και τις μεταβλητές που δηλώνονται τοπικά μέσα σε αυτήν
- Για παράδειγμα, αν δεν προσθέταμε το `i` στην πρώτη κλήση της `for_each()` ο μεταγλωττιστής θα εμφάνιζε μήνυμα λάθους. Επίσης, αν προσπαθήσουμε να προσπελάσουμε το `j` μέσα στη λάμδα, η μεταγλώττιση θα αποτύχει, επειδή η `j` δεν περιέχεται στη λίστα σύλληψης
- Ο λόγος που χρησιμοποιήθηκαν οι `i` και `j` διαφορετικά στο `[]` είναι για να δείτε τους τρόπους σύλληψης
- Αν μπροστά από τη μεταβλητή υπάρχει το `&` σημαίνει ότι η μεταβλητή συλλαμβάνεται μέσω αναφοράς (π.χ. `[&j]`), δηλαδή, η λάμδα μπορεί να τροποποιήσει την τιμή της
- Αν χρησιμοποιείται μόνο το όνομα, σημαίνει ότι συλλαμβάνεται μέσω τιμής (π.χ. `[i]`), δηλαδή, στη συνάρτηση μεταβιβάζεται ένα αντίγραφο της τοπικής μεταβλητής
- Σημειώστε ότι επιτρέπεται να συνδυάσουμε τους τρόπους, δηλαδή, να συλλάβουμε μεταβλητές μέσω αναφοράς και μέσω τιμής (π.χ. `[i, &j]`)

Σύνταξη Λάμδα Συνάρτησης

- Αν γράψουμε `[&]` σημαίνει ότι όλες οι μεταβλητές στην περικλείουσα εμβέλεια συλλαμβάνονται μέσω αναφοράς, ενώ το `[=]` σημαίνει ότι όλες συλλαμβάνονται μέσω τιμής
- Όπως και σε μία συνηθισμένη συνάρτηση, η προαιρετική λίστα ορισμάτων (π.χ. `a`) εισάγεται μέσα σε `()`, ενώ το σώμα της λάμδα μέσα σε `{}`
- Μία λάμδα μπορεί να χρησιμοποιήσει ονόματα που ορίζονται έξω από την συνάρτηση στην οποία περιέχεται. Για παράδειγμα, μπορεί να χρησιμοποιήσει το `cout`
- Αν η λάμδα δεν έχει `return` εντολή, ο τύπος επιστροφής είναι `void`. Αλλιώς, ο τύπος επιστροφής εξάγεται από τον τύπο της `return` έκφρασης. Αν υπάρχουν πολλές `return` εκφράσεις, τότε όλες πρέπει να επιστρέφουν μία τιμή με τον ίδιο τύπο. Εναλλακτικά, μπορεί να χρησιμοποιηθεί ένας επιθεματικός τύπος επιστροφής (π.χ. `[i]() -> int {...}`)

Παρατηρήσεις

- Η σύνταξη της λάμδα συνάρτησης υποστηρίζει πολλές περισσότερες επιλογές από ότι αυτές, σε αυτό το απλό παράδειγμα. Μάλιστα, μπορεί να είναι αρκετά πολύπλοκη
- Ωστόσο, όταν εξοικειωθείτε με τις λεπτομέρειες της σύνταξης, μπορεί να προτιμάτε να χρησιμοποιείτε λάμδα συνάρτηση ως όρισμα, αντί να ορίζετε ξεχωριστές συναρτήσεις ή αντικείμενα συναρτήσεων
- Γενικά, μία λάμδα συνάρτηση θα πρέπει να είναι όσο πιο απλή γίνεται, ώστε να είναι κατανοητή η λειτουργία της
- Μάλιστα, για να διαβάζεται πιο εύκολα ο κώδικας, μπορούμε να ονοματίσουμε τη λάμδα συνάρτηση ώστε να δηλώσουμε το σκοπό της, καθώς και να την κάνουμε διαθέσιμη για γενικότερη χρήση στο πρόγραμμά μας. Για παράδειγμα:

```
auto show_less = [i](int a) {if(a < i) cout << a << ' ';};
for_each(v.begin(), v.end(), show_less);
```

Η λέξη `auto` κάνει τον μεταγλωττιστή να βρει τον τύπο της `show_less` από τη δεξιά πλευρά του `=`, όπου βρίσκεται η λάμδα συνάρτηση