ΟΓΔΟΗ ΔΙΑΛΕΞΗ

Στατικές ποσότητες

- Οι μεταβλητές που ορίζονται στο σώμα μιας συνάρτησης κανονικά δημιουργούνται όταν η ροή του προγράμματος φτάσει στο σημείο δήλωσής τους και καταστρέφονται όταν η ροή φύγει από την εμβέλειά τους.
- Μπορούμε να ορίσουμε ως **static** μια ποσότητα ώστε να δημιουργηθεί και να πάρει αρχική τιμή κατά τη μεταγλώττιση (αν είναι εφικτό) ή μόνο την πρώτη φορά που η ροή θα συναντήσει τη δήλωσή της και, επιπλέον, να μην καταστραφεί κατά την έξοδο από τη συνάρτηση.
- Πρόσβαση σε ποσότητες δηλωμένες ως **static** έχουμε μόνο στην εμβέλειά τους.

Παράδειγμα

```
void func(double a)
{
  static int howmany{0};
  // .....
  ++howmany;
}
```

Η μεταβλητή howmany στο παράδειγμα ουσιαστικά μετρά πόσες φορές κλήθηκε η συνάρτηση.

Συνάρτηση `main()`

Μορφές ορισμού

- **`int`** `main() {...}` ή, ισοδύναμα, **`int`** `main(`**`void`**`) {...}`.
- **`int`** `main(`**`int`** `argc,` **`char`** `*argv[]) {...}`.

Ο δεύτερος τρόπος σύνταξης επιτρέπει να «περάσουν» ορίσματα στη `main()` από το λειτουργικό σύστημα (το οποίο καλεί τη συνάρτηση) *κατά την έναρξη εκτέλεσης του προγράμματος*.

- Το πρώτο όρισμα (argc), παίρνει τιμή κατά 1 μεγαλύτερη από το πλήθος των ορισμάτων που δίνονται στη `main()`.
- Το δεύτερο όρισμα (argv), είναι διάνυσμα με διάσταση argc+1 και περιέχει τα ορίσματα (ως σειρά χαρακτήρων).
  Η τιμή `argv[0]` είναι πάντα το όνομα του εκτελέσιμου, τα `argv[1]`, `argv[2]`, ... το πρώτο, δεύτερο, ... όρισμα, ενώ η τελευταία τιμή, `argv[argc]`, είναι 0 (`NULL`).

Προσοχή: Τα ορίσματα με αριθμητικές τιμές περνούν, όπως όλα, ως σειρές χαρακτήρων. Χρειάζονται μετατροπή με τις συναρτήσεις του `<cstdlib>`, `std::atoi()` (για **`int`**) και `std::atof()` (για **`double`**).

Μαθηματικές συναρτήσεις

- Παρέχονται, από το header `<cmath>` κυρίως, δεκάδες μαθηματικές συναρτήσεις για διάφορους τύπους ορισμάτων, πραγματικούς και ακέραιους.
- Ορισμένες συναρτήσεις παρέχονται από το `<cstdlib>`.
- Όσες μαθηματικές συναρτήσεις έχουν νόημα για μιγαδικούς αριθμούς παρέχονται από το `<complex>`.

Δείτε τις στις σημειώσεις (Κεφάλαιο 7, παρ. 15).

Μαθηματικές συναρτήσεις

Συνάρτηση	Επιστρεφόμενη τιμή
`double cos(double x)`	Συνημίτονο του x.
`double sin(double x)`	Ημίτονο του x.
`double tan(double x)`	Εφαπτομένη του x.
`double acos(double x)`	Τόξο συνημιτόνου του x.
`double asin(double x)`	Τόξο ημιτόνου του x.
`double atan(double x)`	Τόξο εφαπτομένης του x.
`double atan2(double x, double y)`	Τόξο εφαπτομένης $\tan^{-1}(x/y)$.
`double pow(double x, double a)`	Ύψωση σε δύναμη, x^a.
`double sqrt(double x)`	Η τετραγωνική ρίζα του x.
`double cbrt(double x)`	Η κυβική ρίζα του x.
`double hypot(double x, double y)`	$\sqrt{x^2+y^2}$.
`double exp(double x)`	Εκθετικό του x (e^x).
`double log(double x)`	Φυσικός λογάριθμος του x ($\ln x$).
`double log2(double x)`	Δυαδικός λογάριθμος του x ($\log_2 x$).
`double log10(double x)`	Δεκαδικός λογάριθμος του x ($\log_{10} x$).
`double abs(double x)`	Απόλυτη τιμή του x.

Όλες χρειάζονται πρόθεμα `std::`. Προϋποθέτουν το `#include <cmath>`.
Οι τριγωνομετρικές συναρτήσεις δέχονται/επιστρέφουν τις γωνίες σε rad.

Παράδειγμα

Το $e^{-x}\cos(y)$ γράφεται `std::exp(-x) * std::cos(y)`.

Υπόδειγμα (template) συνάρτησης

Εισαγωγή

Συναρτήσεις που επιστρέφουν το μικρότερο από δύο ορίσματα:

```
int min(int const & a, int const & b)
{
   return (a < b ? a : b);
}

double min(double const & a, double const & b)
{
   return (a < b ? a : b);
}
```

Ο κώδικας είναι ίδιος, μόνο οι τύποι αλλάζουν.

Χρήσιμο είναι το overloading ώστε να έχουν όλες το ίδιο όνομα, αλλά μήπως μπορεί να τις γράφει ο compiler για όλους τους ενσωματωμένους αριθμητικούς τύπους και όχι εμείς;

Υπόδειγμα (template) συνάρτησης

Ορισμός και δήλωση

Δίνουμε στον compiler ένα υπόδειγμα (template) για το πώς να γράφει τις ζητούμενες συναρτήσεις.

Ο ορισμός του είναι

```
template<typename T>
T min(T const & a, T const & b)
{
   return (a < b ? a : b);
}
```

Η δήλωσή του είναι

```
template<typename T>
T min(T const & a, T const & b);
```

Υπόδειγμα (template) συνάρτησης

Παρατηρήσεις

- Το όνομα της *παραμέτρου του template* είναι της επιλογής μας, στο παράδειγμα T. Εδώ συμβολίζει τύπο. Θα μπορούσε να συμβολίζει ακέραια ποσότητα.
- Μπορούμε να έχουμε πολλές παραμέτρους σε ένα template, και όχι μόνο για τύπους ορισμάτων:

```
template<typename T1, typename T2, int N>
T2 f(T1 const & x, T1 const & y)
{
   return static_cast<T2>(x+N)/static_cast<T2>(y+N);
}
```

- Μπορούμε να έχουμε προκαθορισμένες "τιμές" για τις τελευταίες παραμέτρους ενός template:

```
template <typename X,typename Y=double,typename Z=int>
...
```

Υπόδειγμα (template) συνάρτησης

Χρήση

- Αν οι παράμετροι είναι τύποι ορισμάτων, μπορούν να υπολογιστούν από τα ορίσματα. Για π.χ. το template `min` γράφουμε:

```
auto x = min(10, 12);
auto y = min(1.4, 2.8);
```

- Αλλιώς, προσδιορίζουμε τις "τιμές" των παραμέτρων εντός `<>` μετά το όνομα της συνάρτησης. Για π.χ. το template `f` γράφουμε

```
template<typename T1, typename T2, int N>
T2 f(T1 const & x, T1 const & y);

auto x = f<long int,double,10>(2L,6L); // x: double
```

Παρατήρηση

Ο compiler πρέπει να γνωρίζει τον *ορισμό* του template, δεν αρκεί η δήλωση. Επομένως, ο *ορισμός* πρέπει να βρίσκεται (ή να συμπεριλαμβάνεται) στο αρχείο που χρησιμοποιείται το template.

Υπόδειγμα (template) συνάρτησης

Εξειδίκευση

Αν επιθυμούμε να έχουμε ένα γενικό template για κάθε "τιμή" των παραμέτρων του, αλλά κάποια άλλη συνάρτηση για κάποιο συγκεκριμένο σύνολο παραμέτρων γράφουμε:

```
template<typename T> T min(T const & a, T const & b)
{
   return (a < b ? a : b);
}
template<> int min(int const & a, int const & b)
{
   return b ^ ((a ^ b) & -(a < b));
}
```

Η συνάρτηση με το **template**<> αποτελεί εξειδίκευση (specialization) του γενικού template. Τυχόν άλλες παράμετροι μπορούν να παραμείνουν ώστε να έχουμε νέο template (*μερική εξειδίκευση*).

Αλγόριθμοι αναζήτησης

Αναζητούμε συγκεκριμένη τιμή σε ένα διάνυσμα.

- Αν τα στοιχεία δεν έχουν κάποια οργάνωση (ταξινόμηση, κατηγοριοποίηση, κλπ.), πρέπει να τη συγκρίνουμε με κάθε στοιχείο του διανύσματος. Αυτός είναι ο αλγόριθμος γραμμικής αναζήτησης. Κατά μέσο όρο χρειάζονται $(1 + N)/2$ συγκρίσεις, όπου N το πλήθος των στοιχείων του διανύσματος.
- Αν τα στοιχεία είναι ταξινομημένα, μπορούμε με διαδοχικές συγκρίσεις να αποκλείουμε το μισό τμήμα του διανύσματος κάθε φορά. Αυτός είναι ο αλγόριθμος δυαδικής αναζήτησης.
  Κατά μέσο όρο χρειάζονται $1 + \log_2 N$ συγκρίσεις, όπου N το πλήθος των στοιχείων του διανύσματος.
- Αν τα στοιχεία είναι ομαδοποιημένα, π.χ. ακέραιοι οργανωμένοι με βάση το τελευταίο ψηφίο, η αναζήτηση είναι πιο γρήγορη. Αυτός είναι ο αλγόριθμος αναζήτησης με hash.

Αλγόριθμος δυαδικής αναζήτησης

Μη αναδρομικός

Σε ταξινομημένο, με αύξουσα σειρά, διάνυσμα:

1. Συγκρίνουμε το μεσαίο στοιχείο του διανύσματος (ή ένα από τα δύο πλησιέστερα στη μέση αν το διάνυσμα έχει άρτιο πλήθος στοιχείων) με τη ζητούμενη τιμή:
   - Αν είναι ίσα, έχουμε βρει το ζητούμενο και επιστρέφουμε τη θέση στην οποία το βρήκαμε.
   - Αν η ζητούμενη τιμή είναι μικρότερη, σημαίνει ότι, αν υπάρχει, βρίσκεται στο πρώτο μισό του διανύσματος.
   - Αν η ζητούμενη τιμή είναι μεγαλύτερη, σημαίνει ότι, αν υπάρχει, βρίσκεται στο δεύτερο μισό του διανύσματος.

   Επομένως, με την πρώτη σύγκριση, αν δεν βρήκαμε την ζητούμενη τιμή, περιορίζουμε στο μισό τον χώρο αναζήτησης.
2. Επαναλαμβάνουμε τη διαδικασία για το τμήμα του διανύσματος που επιλέξαμε στο προηγούμενο βήμα έως ότου, με διαδοχικές διχοτομήσεις, περιοριστούμε σε ένα στοιχείο. Τότε, αν είναι ίσο με τη ζητούμενη τιμή επιστρέφουμε τη θέση του, αλλιώς η ζητούμενη τιμή δεν περιέχεται στο διάνυσμα.

Αλγόριθμος δυαδικής αναζήτησης

Αναδρομικός

Σε ταξινομημένο, με αύξουσα σειρά, διάνυσμα:

1. αν το πλήθος των στοιχείων είναι 0, το ζητούμενο στοιχείο δεν υπάρχει.
2. αν το πλήθος των στοιχείων είναι 1, συγκρίνουμε το μοναδικό στοιχείο με το ζητούμενο. Αν είναι ίσα επιστρέφουμε την *απόλυτη* θέση του (δηλαδή, από την αρχή του διανύσματος), αλλιώς το ζητούμενο στοιχείο δεν υπάρχει.
3. αν το πλήθος των στοιχείων είναι μεγαλύτερο από 1, συγκρίνουμε το μεσαίο στοιχείο (ή ένα από τα δύο στοιχεία που είναι πλησιέστερα στο μέσο του διανύσματος) με το ζητούμενο. Αν το ζητούμενο στοιχείο είναι μικρότερο, το αναζητούμε στο ίδιο διάνυσμα, στο «πρώτο μισό». Αλλιώς, το αναζητούμε στο «δεύτερο μισό».

Αλγόριθμος ταξινόμησης quick sort

Ο αλγόριθμος quick sort χρειάζεται συνήθως $O(N \log N)$ συγκρίσεις για να ταξινομήσει διάνυσμα N στοιχείων και είναι ο ακόλουθος:

1. Αν η λίστα στοιχείων έχει ένα ή κανένα στοιχείο επιστρέφουμε, καθώς δεν χρειάζεται ταξινόμηση.
2. Επιλέγουμε ένα οποιοδήποτε στοιχείο της αρχικής λίστας.
3. Επιδιώκουμε να μεταφέρουμε στην αρχή της λίστας τα στοιχεία που είναι μικρότερα ή ίσα με το επιλεγμένο και στο τέλος της λίστας όσα είναι μεγαλύτερα από το επιλεγμένο. Στη μοναδική θέση που απομένει, μεταφέρουμε το επιλεγμένο στοιχείο.
4. Εφαρμόζουμε την ίδια διαδικασία στις υπο-λίστες πριν και μετά το επιλεγμένο στοιχείο (στη νέα του θέση), χωρίς να το περιλαμβάνουμε.

Ο διαχωρισμός των στοιχείων μπορεί να γίνει ως εξής:

- διατρέχουμε τη λίστα με δύο δείκτες· ο ένας ξεκινά από την αρχή και ο άλλος από το τέλος. Ο πρώτος θα αυξάνει όσο δείχνει σε στοιχεία μικρότερα ή ίσα με το επιλεγμένο ενώ ο δεύτερος θα μειώνεται όσο δείχνει σε στοιχεία μεγαλύτερα από το επιλεγμένο. Οι δείκτες θα παραλείπουν το επιλεγμένο στοιχείο.
- Όσο ο πρώτος δείκτης δεν έχει ξεπεράσει το δεύτερο, εναλλάσσουμε τα στοιχεία στα οποία δείχνουν οι δύο δείκτες.
- Όταν ο πρώτος δείκτης ξεπεράσει το δεύτερο, εναλλάσσουμε το επιλεγμένο στοιχείο με αυτό που δείχνει ο δεύτερος δείκτης.

Αλγόριθμος ταξινόμησης merge sort

Ο αλγόριθμος merge sort ταξινομεί λίστα N στοιχείων με πλήθος συγκρίσεων ανάλογο του $N \log N$. Σύμφωνα με αυτόν:

1. Αν η λίστα στοιχείων δεν έχει κανένα ή έχει μόνο ένα στοιχείο, επιστρέφουμε καθώς δεν χρειάζεται ταξινόμηση.
2. Χωρίζουμε τη λίστα σε δύο περίπου ίσα μέρη.
3. Ταξινομούμε κάθε νέο τμήμα με ξεχωριστή εφαρμογή της τρέχουσας διαδικασίας (επομένως καλούμε τη συνάρτηση που γράφουμε και που υλοποιεί τη merge sort).
4. Συγχωνεύουμε τις δύο ταξινομημένες λίστες με τέτοιο τρόπο ώστε η τελική να είναι επίσης ταξινομημένη.