ΠΡΟΓΡΑΜΜΑΤΙΣΤΙΚΕΣ ΤΕΧΝΙΚΕΣ

https://courses.pclab.ece.ntua.gr/course/view.php?id=64

Διδάσκοντες:	Νίκος Παπασπύρου	(nickie@softlab.ntua.gr)
	Άρης Παγουρτζής	(pagour@cs.ntua.gr)
	Γιώργος Στάμου	(gstam@cs.ntua.gr)
	Γιώργος Γκούμας	(goumas@cslab.ece.ntua.gr)
	Γιώργος Αλεξανδρίδης	(gealexandri@islab.ntua.gr)
	Γιώργος Σιόλας	(gsiolas@image.ntua.gr)
	Παρασκευή Τζούβελη	(tpar@image.ntua.gr)

Διαφάνειες παρουσιάσεων

12/3/21

- ✓Η γλώσσα προγραμματισμού C++
- ✓Αρχές αντικειμενοστρεφούς προγραμματισμού
- ✓Δομές δεδομένων

Εισαγωγή στις δομές δεδομένων (i)

- Αλγόριθμος

  Πεπερασμένο σύνολο εντολών που,
  όταν εκτελεστούν,
  επιτυγχάνουν κάποιο επιθυμητό αποτέλεσμα

  - Δεδομένα εισόδου και εξόδου
  - Κάθε εντολή πρέπει να είναι:
    - καλά ορισμένη
    - απλή
  - Η εκτέλεση πρέπει να σταματά

Εισαγωγή στις δομές δεδομένων (ii)

- Πολυπλοκότητα

  Μέτρο εκτίμησης της απόδοσης αλγορίθμων ως συνάρτηση του μεγέθους του προβλήματος που επιλύουν

  - Χρόνος εκτέλεσης
  - Χωρητικότητα μνήμης που απαιτείται
- Ακριβής μέτρηση πολυπλοκότητας

  $t = f(n)$
- Τάξη μεγέθους, συμβολισμοί Ο, Ω, Θ

  $t = O(f(n))$

Εισαγωγή στις δομές δεδομένων (iii)

- Ορισμός των κλάσεων O, Ω, Θ
  - Άνω όριο

    $g = O(f) \Leftrightarrow \exists c.\ \exists n_0.\ \forall n > n_0.\ g(n) < c\ f(n)$
  - Κάτω όριο

    $g = \Omega(f) \Leftrightarrow \exists c.\ \exists n_0.\ \forall n > n_0.\ g(n) > c\ f(n)$
  - Τάξη μεγέθους

    $g = \Theta(f) \Leftrightarrow \exists c_1, c_2.\ \exists n_0.\ \forall n > n_0.$
    $c_1\ f(n) < g(n) < c_2\ f(n)$
- Διάταξη μερικών κλάσεων πολυπλοκότητας

  $O(1) < O(\log n) < O(\sqrt{n}) < O(n) < O(n \log n)$
  $< O(n^2) < O(n^3) < O(2^n) < O(n!) < O(n^n)$

Αφαίρεση (abstraction)

- Νοητική διαδικασία αντίληψης των θεμελιωδών ιδιοτήτων μιας οντότητας, πέρα από τα φυσικά χαρακτηριστικά ή/και λεπτομέρειες που ξεχωρίζουν τα συγκεκριμένα αντικείμενα του φυσικού κόσμου
  - π.χ. η *έννοια* του μεγέθους στη φύση
- Χρησιμοποιείται:
  - για να συγκαλύψει άσχετες λεπτομέρειες
  - για να δώσει έμφαση σε σχετικές ιδιότητες με το πρόβλημα και την επίλυσή του.

Αφαίρεση και δομές δεδομένων (i)

- Αφαίρεση δεδομένων (data abstraction)
  - Διαχωρισμός ιδιοτήτων και υλοποίησης
  - Ιδιότητες ενός τύπου δεδομένων:
    ο τρόπος με τον οποίο δημιουργεί κανείς και χρησιμοποιεί δεδομένα αυτού του τύπου
  - Υλοποίηση ενός τύπου δεδομένων:
    ο τρόπος με τον οποίο αναπαρίστανται τα δεδομένα στη μνήμη του υπολογιστή και προγραμματίζονται οι διαθέσιμες πράξεις

Αφαίρεση και δομές δεδομένων (ii)

- Οι λεπτομέρειες υλοποίησης επηρεάζουν την πολυπλοκότητα των αλγορίθμων
- Είναι συχνή επομένως η αλλαγή τρόπου υλοποίησης κάποιου τύπου δεδομένων
- Η αφαίρεση δεδομένων ελαχιστοποιεί τις αλλαγές που απαιτούνται στο πρόγραμμα που χρησιμοποιεί έναν τύπο δεδομένων, όταν αλλάξει ο τρόπος υλοποίησης

Αφηρημένοι τύποι δεδομένων

- Αφηρημένος τύπος δεδομένων – ΑΤΔ (abstract data type)
  - καθορίζει τις ιδιότητες του τύπου δεδομένων
  - δεν καθορίζει την υλοποίησή του
  - ⇒ το public μέρος μιας κλάσης στη C++
- Αλγεβρικός ορισμός ΑΤΔ
  - σύνταξη: όνομα του τύπου και επικεφαλίδες των πράξεων
  - σημασιολογία: κανόνες που περιγράφουν τη λειτουργία των πράξεων

Συγκεκριμένοι τύποι δεδομένων

- Συγκεκριμένος τύπος δεδομένων – ΣΤΔ (concrete data type)
  - καθορίζει τις ιδιότητες του τύπου δεδομένων
  - καθορίζει επακριβώς την υλοποίησή του
- Κάθε γλώσσα προγραμματισμού υποστηρίζει ορισμένους ΣΤΔ, π.χ.
  - απλοί τύποι: ακέραιοι αριθμοί, πραγματικοί αριθμοί, χαρακτήρες, λογικές τιμές
  - σύνθετοι τύποι: πίνακες (arrays), εγγραφές (records), δείκτες (pointers), σύνολα (sets)

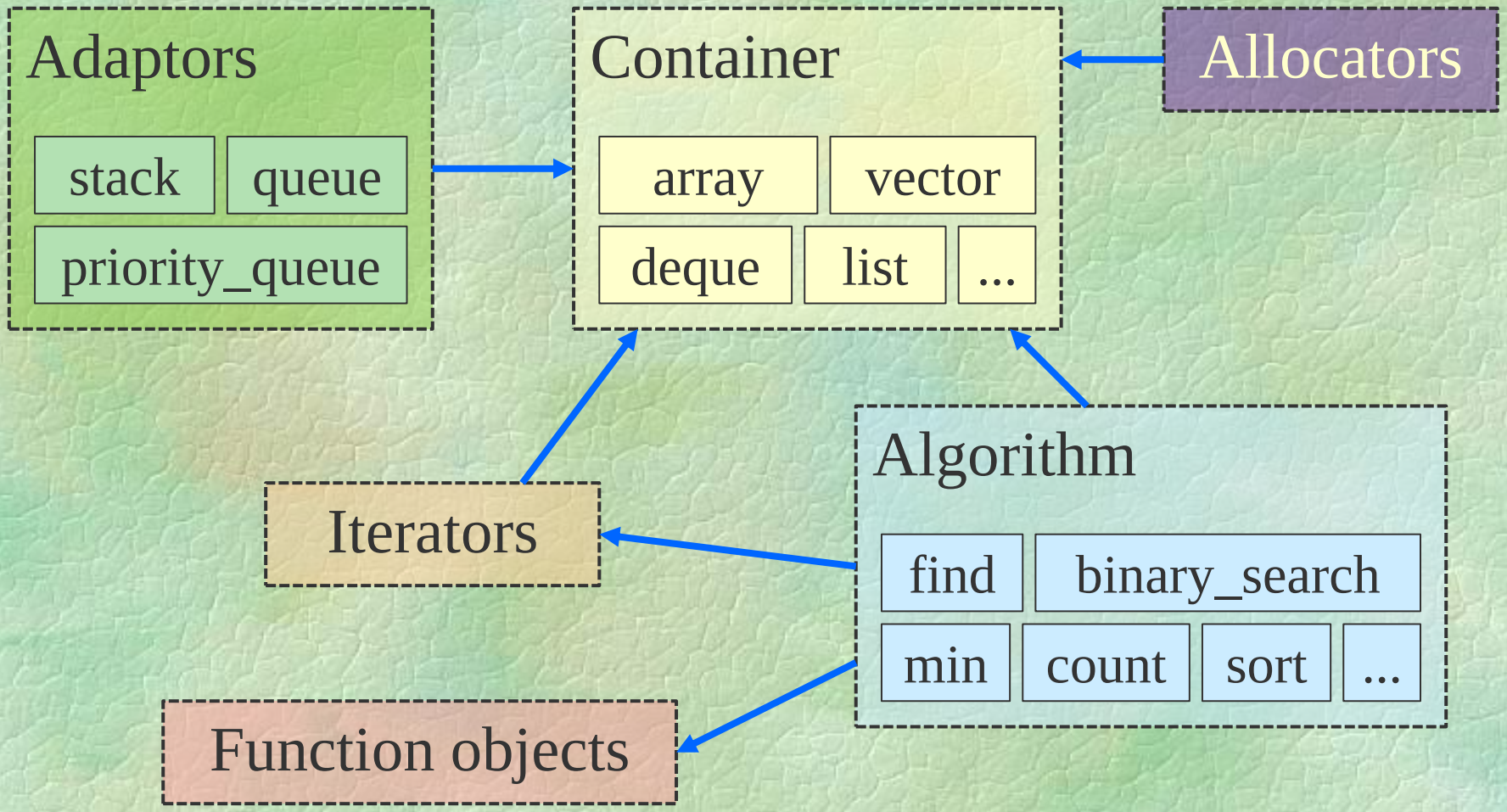
Σχεδιαστικό μόρφημα (design pattern)

- Επαναχρησιμοποιήσιμη μορφή λύσης σε κάποιο συχνά εμφανιζόμενο σχεδιαστικό πρόβλημα
- Τα σχεδιαστικά μορφήματα είναι γενικές, καταγεγραμμένες καλές πρακτικές που οι προγραμματιστές μπορούν να χρησιμοποιούν
- Συνήθως αφορούν γενικές μορφές τύπων δεδομένων και αλγορίθμων, αλλά συνηθέστερα τρόπους αρχιτεκτονικής δόμησης συστημάτων λογισμικού
- Στη C++, τα σχεδιαστικά μορφήματα σχεδιάζονται και παρουσιάζονται σαν μία ιεραρχία κλάσεων

Εισαγωγή στην STL (i)

- Standard Template Library
- Μέρος της βιβλιοθήκης της C++
- Αρχική σχεδίαση: Alexander Stepanov
- Περιέχει χρήσιμες υλοποιήσεις:
  - δομών δεδομένων
  - αλγορίθμων
- Για να καταλάβουμε σε βάθος την STL:
  - templates
  - ιεραρχίες κλάσεων

Εισαγωγή στην STL (ii)



Εισαγωγή στην STL (iii)

www.cplusplus.com/reference/stl/

Container class templates

Sequence containers:

array C++11	Array class (class template)
vector	Vector (class template)
deque	Double ended queue (class template)
forward_list C++11	Forward list (class template)
list	List (class template)

Container adaptors:

stack	LIFO stack (class template)
queue	FIFO queue (class template)
priority_queue	Priority queue (class template)

Associative containers:

set	Set (class template)
multiset	Multiple-key set (class template)
map	Map (class template)
multimap	Multiple-key map (class template)

Unordered associative containers:

unordered_set C++11	Unordered Set (class template)
unordered_multiset C++11	Unordered Multiset (class template)
unordered_map C++11	Unordered Map (class template)
unordered_multimap C++11	Unordered Multimap (class template)

Συμβολοσειρά

- Τύπος **`string`**
  - περιτύλιγμα για array από **`char`**
- Χρήσιμες μέθοδοι
  - **`size`**, **`length`**
  - **`c_str`**
  - **`empty`**
  - **`operator[]`**, **`substr`**
  - **`append`**, **`operator+=`**
  - **`compare`**, **`operator==`**, **`operator<`**, κ.λπ.
  - **`insert`**, **`replace`**
  - **`find`**, **`rfind`**

Πίνακας

- Τύπος **`array<T, n>`**
  - παρόμοια συμπεριφορά με τους απλούς πίνακες
  - σταθερού μεγέθους, γνωστού at compile time

```
#include <array>
...
array<int, 10> a;    // παρόμοιο με int a[10];
array<string, 20> s;
a[1] = 42;
s[2] = "Hello";
```

Διάνυσμα (i)

- Τύπος **`vector<T>`**
  - παρόμοια συμπεριφορά με array
  - μεταβλητού μεγέθους
    (εισαγωγή και αφαίρεση στο τέλος)
  - **`push_back`**, **`pop_back`**
  - ειδική αποδοτική υλοποίηση: **`vector<bool>`**

Διάνυσμα (ii)

```
#include <vector>
...

vector<int> x(100), y(200, 42);
cout << y[199] << endl;              // 42
cout << x.size() << endl;            // 100

x.push_back(17);
cout << x[100] << endl;              // 17
cout << x.size() << endl;            // 101

x.pop_back();
cout << x.size() << endl;            // 100
```

Λίστα (i)

- Τύπος `list<T>`
  - διπλά συνδεδεμένη λίστα, γραμμική διάσχιση
  - εισαγωγή και αφαίρεση σε αρχή, τέλος
  - **`front`**, **`back`**, **`empty`**
  - **`push_front`**, **`pop_front`**
  - **`push_back`**, **`pop_back`**
  - εισαγωγή και διαγραφή ενδιάμεσα
  - **`insert`**, **`erase`**

Λίστα (ii)

```
#include <list>
...
list<int> l;
for (int i = 0; i < 10; ++i)
  if (i % 2 == 0) l.push_front(i);
  else            l.push_back(i);
for (int i = 0; !l.empty(); ++i)
  if (i % 2 == 0) {
    cout << l.back() << endl;
    l.pop_back();
  } else {
    cout << l.front() << endl;
    l.pop_front();
  }
```

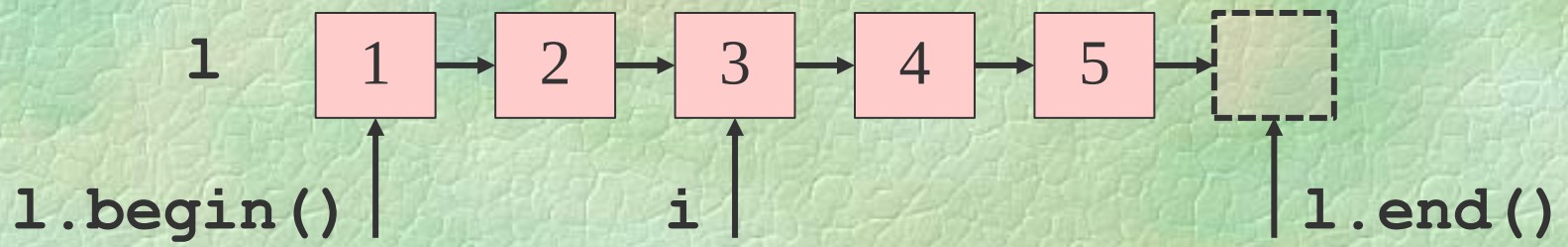
```
9
8
7
6
5
4
3
2
1
0
```

Λίστα (iii)

- Iterators `list<T>::iterator`
  - επιτρέπουν τη διάσχιση ενός container
  - συμπεριφέρονται σαν να ήταν δείκτες

```
list<int> l;
for (int i = 1; i <= 5; ++i) l.push_back(i);

list<int>::iterator i;
for (i = l.begin(); i != l.end(); ++i)
  cout << *i << endl;
```



Λίστα (iv)

- Εναλλακτική, απλούστερη διάσχιση (C++11)

```
list<int> l;
...
for (int x : l)
  cout << x << endl;
for (int &x : l)
  x = 42;
for (const int &x : l)
  cout << x << endl;
```

Στοίβα και ουρά

- Τύπος **`stack<T>`**
  - εισαγωγή και αφαίρεση στην κορυφή
  - **`push`**, **`pop`**, **`top`**, **`empty`**
- Τύπος **`queue<T>`**
  - εισαγωγή στο τέλος, αφαίρεση από την αρχή
  - **`push`**, **`pop`**, **`front`**, **`empty`**
- Και τα δύο μπορούν να υλοποιηθούν με βάσει διαφορετικών containers

```
stack<int, vector<int>> s;
```

Ζεύγη

- Τύπος `pair<T1, T2>`
  - `make_pair`, `first`, `second`

```
#include <utility>

...

int sum(const pair<int, int> &p) {
  return p.first + p.second;
}

pair<int, int> p(1, 2);

cout << sum(p) << endl;
cout << sum(make_pair(3, 4)) << endl;
```

Πλειάδες (tuples) στη C++11

- Τύπος `tuple<T1, ... Tn>`
  - `make_tuple`, `get`

```
#include <tuple>
...
void f(const tuple<string, int, bool> &t) {
  if (get<2>(t)) cout << get<0>(t) << endl;
  else           cout << get<1>(t) << endl;
}
tuple<string, int, bool> t("Hi", 1, true);
f(t); f(make_tuple("No", 42, false));
```

Σύνολο (i)

- Τύπος **`set<T>`**
  - μοναδικά στοιχεία (αλλιώς **`multiset<T>`**)
  - υλοποίηση με ισοζυγισμένο δένδρο δυαδικής αναζήτησης
  - εισαγωγή, εύρεση και αφαίρεση σε O(log*n*)
  - **`insert`**, **`erase`**, **`find`**, **`size`**
  - πράξεις συνόλων (υλοποιούνται και για άλλους containers, λιγότερο αποδοτικά)
  - **`includes`**, **`set_union`**, **`set_intersection`**, **`set_difference`**

Σύνολο (ii)

```
#include <set>
...
set<int> s;
while (true) {
  int x;
  cin >> x;
  if (s.find(x) == s.end())
    s.insert(x);
  else
    cout << "You lose, I've seen" << x
         << "before!" << endl;
}
```

Χάρτης (i)

- Τύπος **`map<K, T>`**
  - απεικόνιση κλειδιών τύπου **`K`** σε τιμές τύπου **`T`**
  - μοναδικές τιμές κλειδιών (αλλιώς **`multimap<K, T>`**)
  - υλοποίηση με ισοζυγισμένο δένδρο δυαδικής αναζήτησης
  - εισαγωγή, εύρεση και αφαίρεση σε O(log*n*)
  - **`insert`**, **`erase`**, **`find`**, **`size`**
  - **`operator[]`**

Χάρτης (ii)

```
#include <map>
...

map<int, string> m;

m.insert(make_pair(1, "one"));
m.insert(make_pair(2, "two"));
m.insert(make_pair(3, "three"));

int x;
cin >> x;

map<int, string>::iterator p;
p = m.find(x);

if (p != m.end())
  cout << p->second << endl;
```

Συνήθως όμως, τα maps δε χρησιμοποιούνται έτσι!

Χάρτης (iii)

Αλλά έτσι!

```
#include <map>
...
map<int, string> m;

m[1] = "one”;
m[2] = "two";
m[3] = "three";

int x;
cin >> x;

if (m[x]) cout << m[x] << endl;

/* Προσοχή! Αν δεν υπάρχει το m[x] τότε η αναφορά σε
   αυτό θα το δημιουργήσει με τιμή ένα κενό string  */
```

Χάρτης (iv)

$O(2^n)$

```
int fib(int n) {
  return n < 2 ? n : fib(n-1) + fib(n-2);
}

int memo_fib(int n) {
  static map<int, int> cache;
  // Αν είναι εύκολο, τελείωνε!
  if (n < 2) return n;
  // Αν υπάρχει ήδη, μην το υπολογίσεις.
  map<int, int>::iterator p = cache.find(n);
  if (p != cache.end()) return p->second;
  // Αλλιώς υπολόγισε και αποθήκευσέ το.
  int v = memo_fib(n-1) + memo_fib(n-2);
  cache[n] = v;  return v;
}
```

Memoization

$O(n \log n)$

Χρήσιμοι αλγόριθμοι

- Αναζήτηση
- Ταξινόμηση
- Λεξικογραφική διάταξη
- Γεννήτρια μεταθέσεων
- κ.λπ.

```
#include <algorithm>
```

Αναζήτηση (i)

- `find(first, last, value)`
  - σε χρόνο O(n)

- `binary_search(first, last, value)`
- `binary_search(first, last, value, compare)`
  - σε χρόνο O(log$n$)

Αναζήτηση (ii)

```
vector<int> v;
...
vector<int>::iterator i =
  find(v.begin(), v.end(), 42);
if (i == v.end())
  cout << "not found" << endl;

// Αν οι αριθμοί είναι ταξινομημένοι:
vector<int>::iterator i =
  binary_search(v.begin(), v.end(), 42);
if (i != v.end())
   *i = 17;              // Άλλαξε το 42 σε 17
```

Ταξινόμηση (i)

- `sort(first, last)`
- `sort(first, last, compare)`
  - σε χρόνο O(*n*log*n*) στη μέση περίπτωση
- `stable_sort(first, last)`
- `stable_sort(first, last, compare)`
  - σε χρόνο O(*n*log*n*) στη χειρότερη περίπτωση
  - διατηρεί τη σειρά ίσων στοιχείων

Ταξινόμηση (ii)

```
vector<int> v;
...
// Σε αύξουσα σειρά:
sort(v.begin(), v.end());

// Σε φθίνουσα σειρά:
sort(v.begin(), v.end(), greater<int>());
```

- Το **greater<int>()** κατασκευάζει ένα **function object**. Χρειάζεται:

```
#include <functional>
```

Ταξινόμηση (iii)

```
bool my_less_than(int x, int y) {
  return abs(x) < abs(y);
}

struct my_comparator {
  bool operator<(int x, int y) {
    return abs(x) < abs(y);
  }
};

vector<int> v;
...
// Σε αύξουσα σειρά κατ' απόλυτο τιμή, εναλλακτικά:
sort(v.begin(), v.end(), my_less_than);
sort(v.begin(), v.end(), my_comparator);
```

Ταξινόμηση (iv)

- **`partial_sort(first, middle, last)`**
- **`partial_sort(first, middle, last, compare)`**
  - σε χρόνο O(*n*log*m*) στη χειρότερη περίπτωση (*n*=last–first, *m*=middle–first)
  - ταξινομεί μόνο τα *m* πρώτα στοιχεία

```
vector<int> v;
...
partial_sort(v.begin(), v.begin() + 10,
             v.end());
```

Ταξινόμηση (v)

- `nth_element(first, nth, last)`
- `nth_element(first, nth, last, compare)`
  - σε χρόνο O(*n*) στη μέση περίπτωση
  - το ζητούμενο στοιχείο θα είναι στη θέση του
  - τα προηγούμενά του θα είναι μικρότερα, τα επόμενά του θα είναι μεγαλύτερα

```
vector<int> v;
...
nth_element(v.begin(), v.begin() + 10,
            v.end());
```

Ταξινόμηση (vi)

- `min_element(first, last)`
- `min_element(first, last, compare)`
- `max_element(first, last)`
- `max_element(first, last, compare)`
  - σε χρόνο $O(n)$