Προγραμματιστικές Τεχνικές

Παραδόσεις 19/3/2021, 26/3/2021 και 2/4/2021, Νίκος Παπασπύρου

Ιεραρχία containers

1. Οι containers της STL (επίτηδες) δεν είναι οργανωμένοι σε ιεραρχία κληρονομικότητας, για λόγους απόδοσης των υλοποιήσεών τους. Αντίθετα, σε άλλες αντικειμενοστρεφείς γλώσσες (π.χ. στην Java), οι containers είναι οργανωμένοι σε τέτοια ιεραρχία ούτως ώστε να μπορεί κανείς να τους χειριστεί αφαιρετικά, μέσω του πολυμορφισμού.

   Στη συνέχεια θα δούμε πώς θα μπορούσαμε να κατασκευάσουμε στη C++ μία τέτοια ιεραρχία από containers, βοηθητικά αντικείμενα και αντικείμενα που υλοποιούν χρήσιμα σχεδιαστικά πρότυπα (design patterns).

2. Η λογική της C++ και της STL είναι ότι αν θέλει κάποιος να ορίσει μία αφαιρετική συνάρτηση που να δέχεται διαφορετικά είδη container μπορεί να κάνει κάτι σαν το παρακάτω:

   ```
   #include <vector>
   #include <list>
   #include <iostream>

   using namespace std;

   template <typename C>
   void print(C &c) {
     for (auto &x : c) cout << x << endl;
   }

   int main() {
     vector<int> v;
     for (int i = 1; i <= 5; ++i) v.push_back(i);
     print(v);      // print the vector -- C = vector<int>
     list<int> l;
     for (int i = 1; i <= 5; ++i) l.push_back(i);
     print(l);      // print the list -- C = list<int>
   }
   ```

   Προσέξτε ότι ο τύπος `C` στο function template `print` πρέπει να είναι κάποιος τύπος container. Συγκεκριμένα, πρέπει υποστηρίζει τη χρήση iterator, δηλαδή να διαθέτει μεθόδους `begin` και `end` που να επιστρέφουν κατάλληλα iterator objects. Το `auto` στην ίδια συνάρτηση είναι σχεδόν υποχρεωτικό. Δεν είναι καθόλου εύκολο να συσχετίσουμε τον άγνωστο τύπο `C` του container (που στην κλήση είναι π.χ. `vector<int>`) με τον τύπο του στοιχείου που περιέχει (που στην κλήση είναι `int`).

   Με χρήση μίας ιεραρχίας από containers, το ίδιο function template θα γραφόταν κάπως έτσι:

   ```
   template <typename T>
   void print(Container<T> &c) {
     for (T &x : c) cout << x << endl;
   }
   ```

   όπου αφενός ο ορισμός του class template `Container<T>` θα καθιστούσε υποχρεωτική την ύπαρξη των μεθόδων `begin`, `end` και του αντίστοιχου iterator, αφετέρου ο τύπος `T` των στοιχείων του container θα ήταν εμφανής.

3. Για να κατασκευάσουμε μία τέτοια ιεραρχία, ξεκινάμε ορίζοντας ένα αφηρημένο class template `Container<T>` που αντιπροσωπεύει τη γενική μορφή container με περιεχόμενα τύπου `T`. Οι πραγματικοί containers θα κληρονομούν αυτό το class template, θα υλοποιούν τις μεθόδους που ορίζει όπως επίσης και νέες, δικές τους.

   ```
   template <typename T>
   class Container {
   public:
     virtual ~Container() {}
     virtual int size() const = 0;
     virtual bool empty() const { return size() == 0; }
     virtual void clear() = 0;
   };
   ```

Ο εικονικός destructor χρειάζεται ώστε, αν καταστραφεί ένας container που δεν γνωρίζουμε τι τύπου είναι, να κληθεί ο σωστός destructor. Η μέθοδος `size` επιστρέφει το πλήθος των αποθηκευμένων στοιχείων και η μέθοδος `clear` αδειάζει τον container, καταστρέφοντας τα στοιχεία που περιέχει. Τέλος, η μέθοδος `empty` ελέγχει αν ένας container είναι ή όχι άδειος. Η default υλοποίηση χρησιμοποιεί την `size`. Όμως, οι παραγόμενες κλάσεις μπορούν να επανορίσουν αυτή τη μέθοδο αν π.χ. η υλοποίηση της `size` δεν είναι αποδοτική (π.χ. σε μια λίστα που δεν κρατάει προϋπολογισμένο το πλήθος των στοιχείων) αλλά υπάρχει αποδοτική υλοποίηση για την `empty`.

Επίτηδες δεν έχουμε βάλει (ακόμη) υποστήριξη για iterators. Θα επανέλθουμε σε αυτό αργότερα.

Πρώτο παράδειγμα: διπλά συνδεδεμένη λίστα

1. Θα κατασκευάσουμε τώρα ένα πρώτο παράδειγμα τέτοιου container, που θα υλοποιεί τη διπλά συνδεδεμένη λίστα.
2. Ορίζουμε το class template `dlist<T>` που κληρονομεί το `Container<T>`. Η δημόσια διαπροσωπεία (public interface) του θα είναι η εξής (υποθέτουμε ότι βάλαμε τον ορισμό του `Container<T>` στο αρχείο επικεφαλίδας "container.hpp"):

```
#include "container.hpp"

template <typename T>
class dlist: public Container<T> {
public:
  // Constructors: default and copy.
  dlist();
  dlist(const dlist &l);
  // Destructor.
  ~dlist() override;
  // Assignment operator.
  dlist & operator=(const dlist &l);
  // Methods overriden from Container<T>.
  int size() const override;
  void clear() override;
  // Insert elements.
  void push_back(const T &x);
  void push_front(const T &x);
  // Remove elements.
  void pop_front();
  void pop_back();
  // Inspect elements.
  T & front();
  T & back();
};
```

3. Η αναπαράσταση της διπλά συνδεδεμένης λίστας θα γίνει με ένα `struct node` που θα οριστεί στο εσωτερικό της κλάσης `dlist<T>`, στο ιδιωτικό μέρος της ώστε να μην είναι ορατό από τον έξω κόσμο. Ορίζουμε έναν κατασκευαστή για τον κόμβο, έτσι ώστε να διευκολύνεται η κατασκευή των κόμβων και να μη χρειάζεται ρητά να αναθέτουμε τιμές στα πεδία τους.

```
template <typename T>
class dlist: public Container<T> {
  // ...
private:
  struct node {
    T data;
    node *next, *prev;
    node(const T &x, node *n, node *p): data(x), next(n), prev(p) {}
  };

  node *the_front, *the_back;
  int the_size;
};
```

Οι δείκτες `the_front` και `the_back` δείχνουν αντίστοιχα στον πρώτο και τον τελευταίο κόμβο της λίστας. Το πεδίο `the_size` περιέχει το πλήθος των κόμβων της λίστας (για να μη χρειαστεί να τη διασχίζουμε στη μέθοδο `size`).

4. Για να υλοποιήσουμε τις δημόσιες μεθόδους της κλάσης `dlist<T>` θα χρειαστούμε δύο βοηθητικές μεθόδους:

   - Μία μέθοδο `copy` που θα αντιγράφει τους κόμβους μίας δεύτερης λίστας, προσθέτοντάς τους στο τέλος της τρέχουσας λίστας. Αυτή θα χρησιμοποιηθεί δύο φορές: στον copy constructor και στον τελεστή της ανάθεσης. Για ευκολία στην υλοποίηση, μπορούμε να καλούμε τη μέθοδο `push_back` για την προσθήκη των κόμβων.
   - Μία μέθοδο `purge` που θα διατρέχει τη λίστα των κόμβων και θα τους καταστρέφει έναν-έναν. Αυτή θα χρησιμοποιηθεί δύο φορές: στον destructor και στη μέθοδο `clear`.

```
template <typename T>
class dlist: public Container<T> {
  // ...
private:
  void copy(const dlist &l) {
    for (node *p = l.the_front; p != nullptr; p = p->next)
      push_back(p->data);
  }
  void purge() {
    node *p = the_front;
    while (p != nullptr) {
      node *q = p;
      p = p->next;
      delete q;
    }
  }
};
```

5. Στη συνέχεια υλοποιούμε τις δημόσιες μεθόδους της κλάσης `dlist<T>`.

```
template <typename T>
class dlist: public Container<T> {
  // ...
public:
  // Default constructor.
  dlist(): the_front(nullptr), the_back(nullptr), the_size(0) {}
  // Copy constructor.
  dlist(const dlist &l): the_front(nullptr), the_back(nullptr), the_size(0) { copy(l); }
  // Destructor.
  ~dlist() override { purge(); }
  // Assignment operator.
  dlist & operator= (const dlist &l) {
    clear(); copy(l); return *this;
  }
  // Methods overriden from Container<T>.
  int size() const override { return the_size; }
  void clear() override {
    purge();
    the_front = the_back = nullptr;
    the_size = 0;
  }
  // Insert elements, back and front.  (Notice the symmetry between the two!)
  void push_back(const T &x) {
    node *p = new node(x, nullptr, the_back);
    if (the_back != nullptr) the_back->next = p; else the_front = p;
    the_back = p;
    ++the_size;
  }
  void push_front(const T &x) {
    node *p = new node(x, the_front, nullptr);
    if (the_front != nullptr) the_front->prev = p; else the_back = p;
    the_front = p;
    ++the_size;
  }
```

```
  // Remove elements, back and front.  (Notice the symmetry between the two!)
  void pop_back() {
    node *p = the_back;
    the_back = p->prev;
    if (the_back != nullptr) the_back->next = nullptr; else the_front = nullptr;
    delete p;
    --the_size;
  }
  void pop_front() {
    node *p = the_front;
    the_front = p->next;
    if (the_front != nullptr) the_front->prev = nullptr; else the_back = nullptr;
    delete p;
    --the_size;
  }
  // Inspect elements, back and front.  (Notice the symmetry between the two!)
  T & front() { return the_front->data; }
  T & back() { return the_back->data; }
};
```

Προσπάθειες εκτύπωσης της λίστας που κατασκευάσαμε

1. Θα χρησιμοποιήσουμε τώρα τη διπλά συνδεδεμένη λίστα που κατασκευάσαμε προηγουμένως για να αποθηκεύσουμε μερικούς αριθμούς και στη συνέχεια να τους εκτυπώσουμε. Προσπαθώντας όμως να εκτυπώσουμε τα περιεχόμενα της λίστας, διαπιστώνουμε ότι δεν υπάρχει τρόπος να το πετύχουμε χωρίς να αδειάσουμε τη λίστα:

   ```
   #include <iostream>
   #include "dlist.hpp"

   using namespace std;

   template <typename T>
   void print(dlist<T> &l) {
     bool first = true;
     while (!l.empty()) {
       if (first) first = false; else cout << " ";
       cout << l.front();
       l.pop_front();
     }
   }

   int main() {
     dlist<int> l;
     for (int i = 0; i < 10; ++i) l.push_back(i);
     cout << "l of size " << l.size() << ": "; print(l); cout << endl;
     cout << "Print the list again..." << endl;
     cout << "l of size " << l.size() << ": "; print(l); cout << endl;
   }
   ```

   Το παραπάνω πρόγραμμα δυστυχώς εκτυπώνει:

   ```
   l of size 10: 0 1 2 3 4 5 6 7 8 9
   Print the list again...
   l of size 0:
   ```

   γιατί μετά την πρώτη της εκτύπωση, η λίστα έχει αδειάσει.

2. Μπορούμε να εκτυπώσουμε μία λίστα χωρίς να την αδειάσουμε αν αλλάξουμε την παράμετρο του function template `print` έτσι ώστε να μην περνά κατ' αναφορά. Το πέρασμα κατ' αναφορά το θέλαμε για να μην αντιγράφεται ασκόπως η λίστα που θέλουμε να εκτυπώσουμε. Όμως, τώρα μπορούμε να χρησιμοποιήσουμε ακριβώς την αντιγραφή της λίστας ως λύση στο πρόβλημά μας — να βάλουμε την παράμετρο της `print` να μην περνά κατ' αναφορά:

```
template <typename T>
void print(dlist<T> l) {
  bool first = true;
  while (!l.empty()) {
    if (first) first = false; else cout << " ";
    cout << l.front();
    l.pop_front();
  }
}
```

Μετά από αυτή την αλλαγή, το παραπάνω πρόγραμμα τώρα εκτυπώνει:

```
l of size 10: 0 1 2 3 4 5 6 7 8 9
Print the list again...
l of size 10: 0 1 2 3 4 5 6 7 8 9
```

Είναι βέβαια εντελώς αναποδοτικό να πρέπει να αντιγράψουμε κάθε λίστα που θέλουμε να εκτυπώσουμε, επειδή η συνάρτηση εκτύπωσης θα καταστρέψει το αντίγραφο.

3. Ιδανικά θα θέλαμε να χρησιμοποιήσουμε το μηχανισμό των iterators που διαθέτει η βιβλιοθήκη της C++ για να ορίσουμε την `print` ως εξής:

```
template <typename T>
void print(dlist<T> &l) {
  bool first = true;
  for (T &x : l) {
    if (first) first = false; else cout << " ";
    cout << x;
  }
}
```

Θυμίζουμε ότι αυτό είναι συντομογραφία του (περισσότερο περιφραστικού):

```
template <typename T>
void print(dlist<T> &l) {
  bool first = true;
  for (auto i = l.begin(); i != l.end(); ++i) {
    if (first) first = false; else cout << " ";
    cout << *i;
  }
}
```

Για να γίνει αυτό, όμως, πρέπει να συμβούν τα εξής:

- Η κλάση `dlist<T>` να διαθέτει μεθόδους `begin` και `end` που να επιστρέφουν κάποιο κατάλληλο τύπο iterator.
- Ο τύπος του iterator να διαθέτει:
  - copy constructor, για το `auto i = l.begin()`
  - `operator!=`, για το `i != l.end()`
  - `operator++`, για το `++i`
  - `operator*`, για το `*i`

Ορισμός iterator για τις διπλά συνδεδεμένες λίστες

1. Για να διασχίσουμε και να εκτυπώσουμε μία διπλά συνδεδεμένη λίστα, χρειαζόμαστε ένα βρόχο της μορφής:

```
for (node *p = the_front; p != nullptr; p = p->next)
  cout << p->data;
```

Αντιπαραβάλλοντας αυτόν με τον επιθυμητό βρόχο:

```
for (auto i = l.begin(); i != l.end(); ++i)
  cout << *i;
```

οδηγούμαστε στα εξής συμπεράσματα:

- Ο τύπος iterator για τη διπλά συνδεδεμένη λίστα πρέπει να υλοποιείται μέσω ενός `node *`.
- Η μέθοδος `begin` πρέπει να επιστρέφει έναν iterator που να περιέχει το `the_front`.
- Η μέθοδος `end` πρέπει να επιστρέφει έναν iterator που να περιέχει το `nullptr`.

- Ο `operator!=` για iterators πρέπει να συγκρίνει δείκτες.
- Ο `operator*` πρέπει να επιστρέφει το `p->data`.
- Ο `operator++` πρέπει να αντιστοιχεί στο `p = p->next`.

2. Θα ορίσουμε λοιπόν έναν τύπο iterator για τις διπλά συνδεδεμένες λίστες που να είναι ένα σχετικά απλό περιτύλιγμα για τον τύπο `node *`.

```
template <typename T>
class dlist: public Container<T> {
  ...
public:
  class iterator {
  public:
    // Copy constructor.
    iterator(const iterator &i): ptr(i.ptr) {}
    // Dereference operator.
    T & operator*() { return ptr->data; }
    // Prefix increment: ++i
    iterator & operator++() {
      ptr = ptr->next;
      return *this;
    }
    // Postfix increment: i++
    iterator operator++(int) {
      iterator result(*this);
      ptr = ptr->next;
      return result;
    }
    // Comparison operators.
    bool operator==(const iterator &i) const { return ptr == i.ptr; }
    bool operator!=(const iterator &i) const { return ptr != i.ptr; }

  private:
    node *ptr;
    // Private constructor from node *, used only by begin() and end().
    iterator(node *p): ptr(p) {}
    friend class dlist;
  };

  iterator begin() { return iterator(the_front); }
  iterator end() { return iterator(nullptr); }
};
```

Προσοχή στα εξής σημεία:

- Ο τελεστής αποδεικτοδότησης (`operator*`) επιστρέφει αναφορά στο περιεχόμενο ενός κόμβου, προκειμένου να μπορεί κανείς να μεταβάλλει τα περιεχόμενα μίας λίστας διασχίζοντάς την (βλ. το παράδειγμα στο τέλος των σημερινών σημειώσεων).
- Ο τελεστής αύξησης (`++`) έχει ως γνωστόν δύο μορφές, την prefix (`++i`) και την postfix (`i++`), που διαφέρουν ως προς την τιμή που επιστρέφουν. Η πρώτη μορφή επιστρέφει την τιμή **μετά** την αύξηση, ενώ η δεύτερη επιστρέφει την τιμή **πριν** την αύξηση.
- Για να οριστούν και οι δύο αυτές μορφές του τελεστή αύξησης σε μία κλάση χρησιμοποιούμε υπερφόρτωση: η prefix μορφή δεν παίρνει παραμέτρους ενώ η postfix μορφή παίρνει μία (άχρηστη) ακέραια παράμετρο.
- Και οι δύο μορφές του τελεστή αύξησης μετακινούν τον εσωτερικό δείκτη (`ptr = ptr->next`). Όμως, η prefix μορφή επιστρέφει την τιμή μετά την αύξηση (δηλαδή τον ίδιο τον iterator — `*this`) ενώ η postfix μορφή είναι αναγκασμένη να κατασκευάσει ένα αντίγραφο του iterator πριν την αύξηση, το οποίο επιστρέφει.
- Η κλάση `iterator` έχει έναν ιδιωτικό κατασκευαστή που δέχεται ως παράμετρο έναν δείκτη σε κόμβο της λίστας. Ο κατασκευαστής αυτός καλείται από τις μεθόδους `begin` και `end` της κλάσης `dlist<T>`, η οποία δηλώνεται ως φίλη για το σκοπό αυτό.

3. Με την προσθήκη της υποστήριξης για iterators, το παρακάτω παράδειγμα λειτουργεί πια σωστά:

```
#include <iostream>
#include "dlist.hpp"
```

```
using namespace std;

template <typename T>
void print(dlist<T> &l) {
  bool first = true;
  for (T &x : l) {
    if (first) first = false; else cout << " ";
    cout << x;
  }
}

int main() {
  dlist<int> l;
  for (int i = 0; i < 10; ++i) l.push_back(i);
  cout << "l of size " << l.size() << ": "; print(l); cout << endl;
  cout << "Print the list again..." << endl;
  cout << "l of size " << l.size() << ": "; print(l); cout << endl;
  cout << "Double every element..." << endl;
  for (int &x : l) x *= 2;
  cout << "l of size " << l.size() << ": "; print(l); cout << endl;
}
```

Όταν εκτελείται, εκτυπώνει:

```
l of size 10: 0 1 2 3 4 5 6 7 8 9
Print the list again...
l of size 10: 0 1 2 3 4 5 6 7 8 9
Double every element...
l of size 10: 0 2 4 6 8 10 12 14 16 18
```

Δεύτερο παράδειγμα: δέντρο δυαδικής αναζήτησης

1. Ως δεύτερο παράδειγμα container, θα κατασκευάσουμε τώρα ένα class template που θα υλοποιεί δέντρα δυαδικής αναζήτησης. Δε θα υλοποιήσουμε κάποιο μηχανισμό ισοζυγίσματος (balancing), που θα ήταν απαραίτητος για να γίνουν αποδοτικές οι πράξεις προσπέλασης στα στοιχεία του δέντρου. Με μηχανισμούς ισοζυγίσματος (AVL trees, red-black trees, κ.λπ.) θα ασχοληθούμε σε επόμενες διαλέξεις του μαθήματος. Επίσης, δε θα υλοποιήσουμε μεθόδους για την αναζήτηση και τη διαγραφή στοιχείων αλλά θα περιοριστούμε στην εισαγωγή.

2. Ορίζουμε το class template `bstree<T>` που κληρονομεί το `Container<T>`. Η δημόσια διαπροσωπεία του θα είναι:

```
template <typename T>
class bstree: public Container<T> {
public:
  // Constructors: default and copy.
  bstree();
  bstree(const bstree &t);
  // Destructor.
  ~bstree() override;
  // Assignment operator.
  bstree & operator=(const bstree &t);
  // Methods overriden from Container<T>.
  int size() const override;
  void clear() override;
  // Insert elements.
  void insert(const T &x);
};
```

3. Η αναπαράσταση του δέντρου δυαδικής αναζήτησης θα γίνει και πάλι με ένα `struct node` που θα οριστεί στο εσωτερικό της κλάσης `bstree<T>`, στο ιδιωτικό μέρος της, όπως κάναμε και για την περίπτωση της διπλά συνδεδεμένης λίστας. Στον κόμβο του δέντρου, εκτός από τους δείκτες προς τα κάτω, δηλαδή προς το αριστερό και στο δεξί παιδί, επιλέγουμε να προσθέσουμε και ένα δείκτη προς τα πάνω, δηλαδή προς τον "πατέρα" κόμβο. Ο δείκτης αυτός θα μας είναι ιδιαίτερα χρήσιμος όταν αργότερα θα υλοποιήσουμε τους iterators.

```
template <typename T>
class bstree: public Container<T> {
  // ...
private:
  struct node {
    T data;
    node *left, *right, *parent;
    node(const T &x, node *l, node *r, node *p) : data(x), left(l), right(r), parent(p) {}
  };

  node *root;
  int the_size;
};
```

Ο δείκτης `root` δείχνει προς τη ρίζα του δέντρου και το πεδίο `the_size` περιέχει το συνολικό πλήθος των κόμβων.

4. Για να υλοποιήσουμε τους κατασκευαστές, τον καταστροφέα, τον τελεστή ανάθεσης και τη μέθοδο `clear` θα χρειαστούμε δύο βοηθητικές μεθόδους, όπως κάναμε και στην υλοποίηση της διπλά συνδεδεμένης λίστας:

   - Μία μέθοδο `copy` που θα αντιγράφει τους κόμβους ενός δεύτερου δέντρου, προσθέτοντάς τους όπου πρέπει στο τρέχον δέντρο. Αυτή χρησιμοποιείται δύο φορές: στον copy constructor και στον τελεστή της ανάθεσης. Για ευκολία στην υλοποίηση, καλούμε τη μέθοδο `insert` για την προσθήκη των κόμβων.
   - Μία μέθοδο `purge` που θα διατρέχει τους κόμβους και θα τους καταστρέφει έναν-έναν. Αυτή θα χρησιμοποιηθεί δύο φορές: στον destructor και στη μέθοδο `clear`.

```
template <typename T>
class bstree: public Container<T> {
  // ...
private:
  void copy(node *t) {
    if (t != nullptr) {
      insert(t->data);
      copy(t->left);
      copy(t->right);
    }
  }

  static void purge(node *t) {
    if (t != nullptr) {
      purge(t->left);
      purge(t->right);
      delete t;
    }
  }
};
```

Σε αντίθεση με τις αντίστοιχες μεθόδους στην υλοποίηση της διπλά συνδεδεμένης λίστας, και οι δύο παραπάνω μέθοδοι είναι αναδρομικές. Η `copy` υλοποιεί μια προδιατεταγμένη (pre-order) διάσχιση στο δέντρο, ώστε οι κόμβοι να προστίθενται με τη σωστή σειρά, ενώ η `purge` υλοποιεί μια μεταδιατεταγμένη (post-order) διάσχιση, ώστε πρώτα να καταστρέφονται τα παιδιά ενός κόμβου και στη συνέχεια ο ίδιος. Και οι δύο παίρνουν ως παράμετρο τον κόμβο όπου έχουμε φτάσει — η αρχική κλήση πρέπει να γίνει για τη ρίζα του δέντρου. Προσέξτε επίσης ότι η μέθοδος `purge` είναι `static`, καθώς δε χρειάζεται να έχει πρόσβαση στα πεδία του τρέχοντος δέντρου (αυτού που θέλουμε να καταστρέψουμε) — αρκεί ο δείκτης `t` που περνά ως παράμετρος. Αντίθετα, η μέθοδος `copy` δεν είναι `static` γιατί πρέπει να καλέσει τη μέθοδο `insert`, η οποία χρειάζεται πρόσβαση στα πεδία του τρέχοντος δέντρου (αυτού όπου θέλουμε να γίνουν οι προσθήκες) — ο δείκτης `t` δείχνει σε κόμβους του δέντρου το οποίο θέλουμε να αντιγράψουμε.

5. Στη συνέχεια υλοποιούμε τις δημόσιες μεθόδους της κλάσης `bstree<T>`.

```
template <typename T>
class bstree: public Container<T> {
  // ...
public:
  // Default constructor.
  bstree() : root(nullptr), the_size(0) {}
```

```
  // Copy constructor.
  bstree(const bstree &t) : root(nullptr), the_size(0) { copy(t.root); }
  // Destructor.
  ~bstree() override { purge(root); }
  // Assignment operator.
  bstree & operator=(const bstree &t) {
    clear(); copy(t.root); return *this;
  }
  // Methods overriden from Container<T>.
  int size() const override { return the_size; }
  void clear() override {
    purge(root); root = nullptr; the_size = 0;
  }
};
```

6. Για την υλοποίηση της μεθόδου `insert` θα χρειαστούμε πάλι μια αναδρομική συνάρτηση που θα κατεβαίνει το δέντρο και θα προσθέτει το νέο κόμβο στο σωστό σημείο. Θα έχουμε λοιπόν δύο υπερφορτωμένες μεθόδους `insert`: μία με την επικεφαλίδα που βάλαμε στη δημόσια διαπροσωπεία της κλάσης, και μία δεύτερη στατική μέθοδο (και πάλι, δε χρειάζεται πρόσβαση στα πεδία του τρέχοντος δέντρου), που θα υλοποιεί την αναδρομή.

   Αποφασίζουμε να μην προσθέτουμε διπλότυπα στοιχεία στο δέντρο δυαδικής αναζήτησης, δηλαδή αν το στοιχείο που θέλουμε να προσθέσουμε υπάρχει ήδη, η μέθοδος `insert` δε θα κάνει τίποτα. Η αναδρομική έκδοση της μεθόδου `insert` επιστρέφει αποτέλεσμα `bool` το οποίο θα είναι `true`, αν ο νέος κόμβος όντως προστέθηκε, ή `false`, αν δεν προστέθηκε επειδή υπήρχε ήδη στο δέντρο. Αυτό χρησιμοποιείται για να αυξηθεί ανάλογα (ή να μην αυξηθεί) η τιμή του πεδίου `the_size`.

```
template <typename T>
class bstree: public Container<T> {
  // ...
public:
  // Insert elements.
  void insert(const T &x) {
    if (root == nullptr) {
      root = new node(x, nullptr, nullptr, nullptr);
      ++the_size;
    }
    else if (insert(root, x))
      ++the_size;
  }

private:
  static bool insert(node *t, const T &x) {
    if (x < t->data) {
      if (t->left == nullptr) {
        t->left = new node(x, nullptr, nullptr, t);
        return true;
      }
      else
        return insert(t->left, x);
    } else if (x > t->data) {
      if (t->right == nullptr) {
        t->right = new node(x, nullptr, nullptr, t);
        return true;
      }
      else
        return insert(t->right, x);
    }
    return false;
  }
};
```

7. Στο σημείο αυτό, μπορούμε να χρησιμοποιήσουμε την υλοποίηση του `bstree<T>` σε ένα παράδειγμα. Προκειμένου να μπορούμε να εκτυπώσουμε τους κόμβους του δέντρου, υλοποιούμε (προσωρινά) μία μέθοδο `print`, πάλι σε δύο

εκδόσεις: τη δημόσια μέθοδο και μία στατική ιδιωτική μέθοδο που υλοποιεί μια αναδρομική ενδοδιατεταγμένη (in-order) διάσχιση. Η εκτύπωση θα εμφανίζει τα στοιχεία του δέντρου σε αύξουσα σειρά.

```
template <typename T>
class bstree: public Container<T> {
  // ...
public:
  void print() { print(root); }

private:
  static void print(node *t) {
    if (t != nullptr) {
      print(t->left);
      cout << " " << t->data;
      print(t->right);
    }
  }
};
```

Η χρήση όλων αυτών φαίνεται στο ακόλουθο παράδειγμα, στο οποίο προστίθενται σε ένα δέντρο οι αριθμοί 5, 2, 8, 4, 1, 7, 6, 0, 9, και 3, με αυτή τη σειρά.

```
int main() {
  bstree<int> t;
  int x[] = { 5, 2, 8, 4, 1, 7, 6, 0, 9, 3 };
  for (int i = 0; i < 10; ++i) t.insert(x[i]);
  cout << "t of size " << t.size() << ":"; t.print(); cout << endl;
}
```

Όταν εκτελείται, εκτυπώνει:

```
t of size 10: 0 1 2 3 4 5 6 7 8 9
```

Ορισμός iterator για δυαδικά δέντρα

1. Το επόμενο πράγμα που θα θέλαμε να κάνουμε είναι να υλοποιήσουμε το μηχανισμό των iterators για τα δυαδικά δέντρα. Θα θέλαμε αντί να διασχίζουμε το δέντρο με την αναδρομική μέθοδο `print` που γράψαμε παραπάνω, να μπορούμε να το διασχίζουμε με το γενικότερο τρόπο που χρησιμοποιήσαμε και για λίστες:

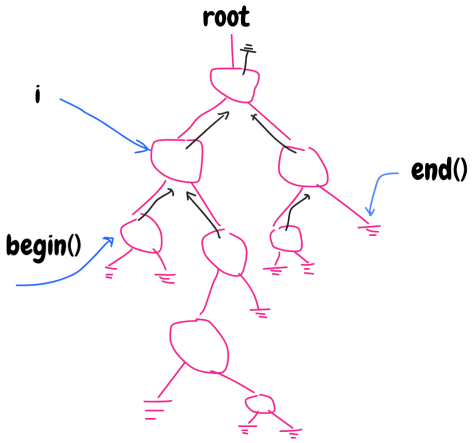
   ```
   template <typename T>
   void print(bstree<T> &t) {
     for (auto &x : t) cout << " " << x;
   }
   ```

   Για να το πετύχουμε, θα ορίσουμε και πάλι έναν τύπο iterator για τα δυαδικά δέντρα, που και πάλι θα είναι ένα σχετικά απλό περιτύλιγμα για τον τύπο `node *` (βλ. Σχήμα 1).

2. Το μεγαλύτερο μέρος της υλοποίησης του `bstree<T>::iterator` θα είναι ίδιο με την υλοποίηση του `dlist<T>::iterator`. Το γράφουμε εδώ με τις απαραίτητες ελάχιστες τροποποιήσεις. Σημειώστε ότι, στην υλοποίηση του postfix `operator++` αυτή τη φορά φροντίζουμε να χρησιμοποιήσουμε την υλοποίηση του prefix `operator++`, έτσι ώστε να μη χρειαστεί να γράφουμε τα ίδια πράγματα δύο φορές.

   ```
   template <typename T>
   class bstree: public Container<T> {
     // ...
     class iterator {
     public:
       iterator(const iterator &i): ptr(i.ptr) {}
       T & operator*() { return ptr->data; }

       // prefix ++, i.e. ++i
       iterator & operator++();                    // TODO: implementation missing!
       // postfix ++, i.e. i++
       iterator operator++(int) {
   ```



Σχήμα 1: Δυαδικά δέντρα και iterators

```
    iterator result(*this);
    ++(*this);
    return result;
  }

  bool operator==(const iterator &i) const { return ptr == i.ptr; }
  bool operator!=(const iterator &i) const { return ptr != i.ptr; }

private:
  node *ptr;
  iterator(node *p): ptr(p) {}
  friend class bstree;
};

iterator begin();                              // TODO: implementation missing!
iterator end() { return iterator(nullptr); }
};
```

Μία βασική όμως διαφορά είναι ότι η σειρά της διάσχισης στα δυαδικά δέντρα είναι μακράν πιο περίπλοκη από ό,τι στις λίστες. Σε μια λίστα, το μόνο που μπορεί κάποιος να κάνει είναι να ξεκινήσει από την αρχή και να πηγαίνει κάθε φορά στον επόμενο κόμβο. Στα δέντρα αντίθετα, πρέπει να αποφασίσουμε πρώτα το είδος της διάσχισης που θέλουμε. Ας υποθέσουμε ότι επιλέγουμε την **ενδοδιατεταγμένη** (in-order) διάσχιση, δηλαδή πρώτα ό,τι υπάρχει στο αριστερό παιδί, μετά τη ρίζα και μετά ό,τι υπάρχει στο δεξιό παιδί. Σε ένα δέντρο δυαδικής αναζήτησης, η διάσχιση αυτή θα μας δίνει τους κόμβους κατά αύξουσα σειρά του περιεχομένου τους.

3. Δύο σημεία θέλουν ιδιαίτερη προσοχή στην υλοποίηση των iterators για δυαδικά δέντρα, και αυτά παραλείψαμε να ορίσουμε παραπάνω:

   - Από πού ξεκινά η διάσχιση; Δηλαδή, πώς θα υλοποιήσουμε τη μέθοδο `begin`;
   - Ποιος είναι ο επόμενος κόμβος στη διάσχιση; Δηλαδή, πώς θα υλοποιήσουμε τον `operator++`;

   Η απάντηση και στα δύο εξαρτάται από το είδος της διάσχισης — εδώ επιλέξαμε την ενδοδιατεταγμένη (in-order).

4. Για να βρούμε τον πρώτο κόμβο του δέντρου κατά την ενδοδιατεταγμένη διάσχιση πρέπει να ξεκινήσουμε από τη ρίζα και να πηγαίνουμε προς τα "κάτω και αριστερά", μέχρι να φτάσουμε σε έναν κόμβο που δεν έχει αριστερό παιδί. Αυτός πρέπει να είναι ο πρώτος κόμβος (βλ. Σχήμα 1). Η μέθοδος `leftdown` υλοποιεί ακριβώς αυτό, θεωρώντας ότι αρχίζουμε από τον κόμβο `t`:

```
template <typename T>
class bstree: public Container<T> {
  // ...
private:
  static node * leftdown(node *t) {
    if (t == nullptr || t->left == nullptr) return t;
    else                                    return leftdown(t->left);
  }
```

```
public:
  iterator begin() { return iterator(leftdown(root)); }
};
```

5. Κάπως πιο μπερδεμένο είναι το να προχωρήσουμε στον επόμενο κόμβο κατά τη διάρκεια της διάσχισης. Έστω ότι ο τελευταίος κόμβος που έχουμε επισκεφτεί κατά τη διάσχιση είναι αυτός όπου δείχνει ο `ptr`. Τότε:

   - Αν το `ptr` είναι κενό, αυτό σημαίνει ότι έχουμε ήδη φτάσει στο τέλος της διάσχισης και δεν υπάρχει κάτι άλλο να κάνουμε.
   - Αν το `ptr` δείχνει σε ένα κόμβο που έχει δεξιό παιδί (όπως αυτός που δείχνει ο iterator `i` στο Σχήμα 1), τότε η διάσχιση πρέπει να συνεχιστεί στο δεξιό υποδέντρο, αρχίζοντας από τον αριστερότερο κόμβο του. Εδώ μπορούμε και πάλι να χρησιμοποιήσουμε τη συνάρτηση `leftdown` που ορίσαμε πριν.
   - Αν το `ptr` δείχνει σε έναν κόμβο που δεν έχει δεξιό παιδί (όπως το δεξί παιδί του κόμβου όπου δείχνει ο iterator `i` στο Σχήμα 1) τότε η διάσχιση του τρέχοντος υποδέντρου έχει ολοκληρωθεί. Στην περίπτωση αυτή πρέπει να επιστρέψουμε "προς τα πάνω", στον πατέρα αυτού του κόμβου, και για το λόγο αυτό χρειαζόμαστε το δείκτη `parent`. Τα πράγματα όμως είναι λίγο πιο μπερδεμένα:
     - Αν κινηθούμε "προς τα πάνω και αριστερά", δηλαδή φτάσουμε στον πατέρα ενός κόμβου από το δεξί παιδί του, τότε πρέπει να συνεχίσουμε να ανεβαίνουμε προς τα πάνω γιατί και πάλι η διάσχιση του τρέχοντος υποδέντρου έχει ολοκληρωθεί.
     - Αν κινηθούμε "προς τα πάνω και δεξιά", δηλαδή φτάσουμε στον πατέρα ενός κόμβου απο το αριστερό παιδί του, τότε αυτός είναι ο επόμενος κόμβος στη διάσχιση, καθώς έχουμε μόλις ολοκληρώσει τη διάσχιση του αριστερού του παιδιού! Χρειαζόμαστε λοιπόν μία ακόμα βοηθητική συνάρτηση που θα ξεκινά από εκεί που βρισκόμαστε και θα πηγαίνει προς τα "πάνω και αριστερά", μέχρι να φτάσουμε σε έναν πατέρα κόμβο από το αριστερό παιδί του. Θα την ονομάσουμε `leftup` και θα τη χρησιμοποιήσουμε για να ολοκληρώσουμε τον ορισμό του `operator++`.

```
template <typename T>
class bstree: public Container<T> {
  // ...
private:
  static node * leftup(node *t) {
    if (t->parent == nullptr || t->parent->left == t) return t->parent;
    else                                              return leftup(t->parent);
  }

public:
  class iterator {
    // ...
  public:
    // prefix ++, i.e. ++i
    iterator & operator++() {
      if (ptr != nullptr) {
        if (ptr->right != nullptr) ptr = leftdown(ptr->right);
        else                       ptr = leftup(ptr);
      }
      return *this;
    }
  };
};
```

6. Και τώρα μπορούμε να χρησιμοποιήσουμε τους iterators του `bstree<T>` στο παράδειγμά μας, για να διασχίσουμε το δέντρο και να εκτυπώσουμε τις τιμές των κόμβων. Η συνάρτηση `print` που χρησιμοποιούμε έχει ακριβώς τον ίδιο κώδικα με εκείνη που χρησιμοποιήσαμε για να εκτυπώσουμε λίστες — διαφέρει μόνο στον τύπο της παραμέτρου.

```
template <typename T>
void print(bstree<T> &t) {
  bool first = true;
  for (T &x : t) {
    if (first) first = false; else cout << " ";
    cout << x;
  }
}
```

```
int main() {
  bstree<int> t;
  int x[] = { 5, 2, 8, 4, 1, 7, 6, 0, 9, 3 };
  for (int i = 0; i < 10; ++i) t.insert(x[i]);
  cout << "t of size " << t.size() << ": "; print(t); cout << endl;
  cout << "Print the tree again..." << endl;
  cout << "t of size " << t.size() << ": "; print(t); cout << endl;
  for (int &x : t) x *= 2;
  cout << "Double every element..." << endl;
  cout << "t of size " << t.size() << ": "; print(t); cout << endl;
}
```

Όταν εκτελείται, εκτυπώνει:

```
t of size 10: 0 1 2 3 4 5 6 7 8 9
Print the tree again...
t of size 10: 0 1 2 3 4 5 6 7 8 9
Double every element...
t of size 10: 0 2 4 6 8 10 12 14 16 18
```

Ορισμός αφηρημένων containers που υποστηρίζουν iterators

1. Σχεδιάσαμε και υλοποιήσαμε δύο είδη containers που υποστηρίζουν iterator για τη διάσχιση των στοιχείων τους: μια διπλά συνδεδεμένη λίστα `dlist<T>` και ένα δέντρο δυαδικής αναζήτησης `bstree<T>` . Για την εκτύπωση των στοιχείων τους, ορίσαμε το function template `print`, το οποίο όμως έπρεπε κάθε φορά να διαφέρει ως προς τον τύπο της παραμέτρου. Ιδανικά, θα θέλαμε να έχουμε μία υλοποίηση για αυτό το function template που να μοιάζει κάπως έτσι:

   ```
   template <typename T>
   void print(Container<T> &c) {
     bool first = true;
     for (T &x : c) {
       if (first) first = false; else cout << " ";
       cout << x;
     }
   }
   ```

   Δηλαδή, η συνάρτηση `print` να δέχεται ως παράμετρο οποιοδήποτε είδος container. Χάρη στην ιεραρχία των containers και στον πολυμορφισμό, θα μπορούμε να καλούμε τη συνάρτηση αυτή δίνοντάς της είτε `dlist<T>` είτε `bstree<T>`, ή και οποιοδήποτε άλλο είδος container που να υποστηρίζει iterators για τη διάσχιση των στοιχείων του.

2. Ως μία πρώτη προσέγγιση, σκεφτόμαστε να μετακινήσουμε τις μεθόδους `begin` και `end`, που επιστρέφουν iterators στην αρχή και στο τέλος ενός container, από τις κλάσεις `dlist<T>` και `bstree<T>` στη γενικότερη κλάση `Container<T>`.

   ```
   template <typename T>
   class Container {
   public:
     virtual ~Container() {}
     virtual int size() const = 0;
     virtual bool empty() const { return size() == 0; }
     virtual void clear() = 0;

     virtual iterator begin() = 0;
     virtual iterator end() = 0;
   };
   ```

   Κάτι τέτοιο όμως δε θα λειτουργήσει γιατί ο τύπος `iterator` δεν έχει ορισθεί και δεν είναι δυνατόν να ορισθεί γενικά για κάθε τύπο container. Διαφορετικό τύπο iterator απαιτεί ένας πίνακας (αρκεί το index του τρέχοντος στοιχείου), διαφορετικό μία λίστα ή ένα δέντρο (εκεί χρειαζόμαστε ένα δείκτη στον τρέχοντα κόμβο, όπως είδαμε παραπάνω) και διαφορετικό για κάποιο άλλο είδος container.

3. Ορίζουμε λοιπόν μία αφηρημένη μορφή iterator που αναλαμβάνει να διασχίζει αφηρημένους containers. Μία πρώτη προσπάθεια ορισμού μίας τέτοιας κλάσης με τις ελάχιστες απαραίτητες μεθόδους που απαιτούνται για τους iterators (βλ. παραπάνω) μας δίνει:

```
template <typename T>
class Iterator {
public:
  Iterator(const Iterator &i);
  T & operator*();
  Iterator & operator++();    // prefix form:  ++i
  Iterator operator++(int) {  // postfix form: i++
    Iterator result(*this);
    ++(*this);
    return result;
  }
  bool operator==(const Iterator &i) const;
  bool operator!=(const Iterator &i) const { return !(*this == i); }
};
```

Εξυπακούεται ότι έχουμε την πρόθεση να ορίσουμε παραγόμενες κλάσεις που να κληρονομούν την `Iterator<T>` και να υλοποιούν τις ειδικές μορφές iterator, π.χ. για διπλά συνδεδεμένες λίστες και για δυαδικά δέντρα.

Στη συνέχεια, μπορούμε να βάλουμε τις μεθόδους `begin` και `end` στην αφηρημένη κλάση `Container<T>` ή, ακόμα καλύτερα, να ορίσουμε μία νέα αφηρημένη κλάση `Iterable<T>` που να επιδέχεται διάσχιση μέσω iterator. (Αυτό θα μας επιτρέψει να ορίζουμε και iterables που δεν είναι containers, π.χ. μία αριθμητική πρόοδο ή την ακολουθία των αριθμών Fibonacci.)

```
template <typename T>
class Iterable {
public:
  virtual Iterator<T> begin() = 0;
  virtual Iterator<T> end() = 0;
};
```

4. Δυστυχώς όμως, και πάλι αυτή η προσέγγιση δε θα λειτουργήσει τόσο απλά. Η κλάση `Iterator<T>` δεν μπορεί να είναι αφηρημένη (δηλαδή να έχει εικονικές μεθόδους που δεν υλοποιούνται). Αν ήταν αφηρημένη, τότε οι μέθοδοι `begin` και `end` της `Iterable<T>` δεν θα μπορούσαν να επιστρέφουν αντικείμενα της κλάσης `Iterator<T>` (θα μπορούσαν να επιστρέφουν δείκτες ή references σε τέτοια αντικείμενα που θα προέρχονταν από παραγόμενες κλάσεις αλλά αυτό θα άλλαζε αναγκαστικά τη λογική των δύο μεθόδων.

5. Μπορούμε όμως να χρησιμοποιήσουμε ένα επίπεδο έμμεσης αναφοράς (indirection level): αντί να ορίσουμε μία ιεραρχία από iterators (κάτι που όπως εξηγήσαμε δεν μπορούμε να κάνουμε), θα βάλουμε μέσα σε κάθε iterator έναν δείκτη σε μία **υλοποίηση iterator** και θα ορίσουμε μία ιεραρχία από τέτοιες υλοποιήσεις.

```
template <typename T>
class Iterator {
public:
  // Copy constructor.
  Iterator(const Iterator &i): impl(i.impl->clone()) {}
  // Destructor.
  ~Iterator() { delete impl; }
  // Dereference operator.
  T & operator*() { return impl->access(); }
  // Increment, prefix form: ++i
  Iterator & operator++() {
    impl->advance();
    return *this;
  }
  // Increment, postfix form: i++
  Iterator operator++(int) {
    Iterator result(*this);
    impl->advance();
    return result;
  }
```

```
  // Comparisons.
  bool operator==(const Iterator &i) const { return impl->equal(*(i.impl)); }
  bool operator!=(const Iterator &i) const { return !(*this == i); }

  class Impl {
  public:
    virtual ~Impl() {}
    virtual void advance() = 0;
    virtual T & access() const = 0;
    virtual bool equal(const Impl &i) const = 0;
    virtual Impl * clone() const = 0;
  };

  // Constructor from a given implementation.
  Iterator(Impl *i): impl(i) {}

private:
  Impl *impl;
};
```

Προσέξτε τα εξής:

- Η κλάση `Iterator<T>::Impl` είναι η αφηρημένη βάση της ιεραρχίας των υλοποιήσεων των iterators.
- Το πεδίο `impl` περιέχει έναν δείκτη σε μία τέτοια υλοποίηση και οι μέθοδοι της κλάσης `Iterator<T>` καλούν απλά τις αντίστοιχες μεθόδους των υλοποιήσεων, που είναι εικονικές, μέσω αυτού του δείκτη.
- Το αντικείμενο της υλοποίησης, στο οποίο δείχνει το πεδίο `impl`, θεωρείται ιδιοκτησία του iterator. Ο καταστροφέας της κλάσης το καταστρέφει, καλώντας την `delete`, και ο copy constructor φροντίζει να καλέσει μία κατάλληλη μέθοδο `clone` της υλοποίησης, η οποία θα κατασκευάσει ένα αντίγραφο, έτσι ώστε να μην υπάρχουν ποτέ δύο iterators που να μοιράζονται την ίδια υλοποίηση.

6. Το επόμενο βήμα είναι να κάνουμε την κλάση `dlist<T>` να κληρονομεί την `Iterable<T>` και, αντί να ορίζει τους δικούς της iterators, να ορίζει μία δική της υλοποίηση iterator που να κληρονομεί την `Iterator<T>::Impl`.

```
template <typename T>
class dlist: public Container<T>, public Iterable<T> {
  ...
public:
  class ListIteratorImpl: public Iterator<T>::Impl {
  private:
    typedef typename Iterator<T>::Impl Impl;

  public:
    ~ListIteratorImpl() override {}
    void advance() override { ptr = ptr->next; }
    T & access() const override { return ptr->data; }
    bool equal(const Impl &i) const override { return ptr == ((ListIteratorImpl *) &i)->ptr; }
    Impl * clone() const override { return new ListIteratorImpl(ptr); }

  private:
    node *ptr;
    ListIteratorImpl(node *p): ptr(p) {}
    friend class dlist;
  };

  Iterator<T> begin() override { return Iterator<T>(new ListIteratorImpl(the_front)); }
  Iterator<T> end() override { return Iterator<T>(new ListIteratorImpl(nullptr)); }
};
```

Προσέξτε τα εξής:

- Η κλάση `ListIteratorImpl` κληρονομεί την αφηρημένη κλάση `Iterator<T>::Impl` και συγκεκριμενοποιεί την υλοποίηση των iterators για διπλά συνδεδεμένες λίστες. Είναι ουσιαστικά πολύ παρόμοια με την κλάση `dlist<T>::iterator` που είχαμε ορίσει.

- Το `typedef` μας βοηθάει να χρησιμοποιούμε τη συντομογραφία `Impl` στο εσωτερικό της κλάσης `ListIteratorImpl`, όταν θέλουμε να αναφερθούμε στην αφηρημένη κλάση `Iterator<T>::Impl`. Η λέξη κλειδί `typename` είναι μία τεχνική λεπτομέρεια της C++.
- Για τον έλεγχο της ισότητας δύο υλοποιήσεων iterator, είμαστε αναγκασμένοι να δεχθούμε τη δεύτερη υλοποίηση (με την οποία συγκρίνουμε το τρέχον αντικείμενο) μέσω μίας αναφοράς στην αφηρημένη κλάση `Impl` και όχι στη συγκεκριμένη κλάση `ListIteratorImpl`. Αυτό το επιβάλλει η επικεφαλίδα της μεθόδου `equal` στην κλάση `Iterator<T>::Impl`. Η σύγκριση, όμως, έχει νόημα μόνο αν και η δεύτερη υλοποίηση αναφέρεται στην πραγματικότητα σε ένα αντικείμενο της κλάσης `ListIteratorImpl`. Για να προσπελάσουμε αυτό το αντικείμενο είμαστε υποχρεωμένοι να κάνουμε μετατροπή τύπου (type casting). Αυτό είναι σωστό μόνο αν το εν λόγω αντικείμενο ανήκει πραγματικά στην κλάση `ListIteratorImpl`.

  Αν θέλαμε αυτή η μετατροπή να είναι πάντα ασφαλής, θα μπορούσαμε να χρησιμοποιήσουμε τον τελεστή `typeid` που επιστρέφει τον πραγματικό τύπο ενός αντικειμένου. Για να αποφύγουμε να κάνουμε τον ίδιο έλεγχο σε κάθε παραγόμενη κλάση της `Iterator<T>::Impl`, θα μπορούσαμε να κάνουμε τον έλεγχο μία φορά στον τελεστή ισότητας της κλάσης `Iterator<T>::Impl` και στη συνέχεια να υποθέτουμε ότι η μέθοδος `equal` της υλοποίησης καλείται μόνο όταν αυτός ο έλεγχος επιτύχει.

  ```
  template <typename T>
  class Iterator {
  public:
    ...
    class Impl {
    public:
      bool operator==(const Impl &i) const { return typeid(*this) == typeid(i) && equal(i); }
      ...
    };
    ...
  };
  ```

7. Ακριβώς τα ίδια μπορούμε να κάνουμε και για την περίπτωση των iterators των δέντρων δυαδικής αναζήτησης.

8. Κατόπιν τούτων, είμαστε σε θέση να ορίσουμε μία αφαιρετική μέθοδο `print` που να διατρέχει και να εκτυπώνει οποιοδήποτε `Iterable<T>` και στη συνέχεια να τη χρησιμοποιήσουμε για να εκτυπώσουμε τα στοιχεία μίας `dlist<T>` και ενός `bstree<T>`.

  ```
  #include <iostream>
  #include "dlist.hpp"
  #include "bstree.hpp"

  using namespace std;

  template <typename T>
  void print(Iterable<T> &c) {
    bool first = true;
    for (T &x : c) {
      if (first) first = false; else cout << " ";
      cout << x;
    }
  }

  int main() {
    dlist<int> l;
    for (int i = 0; i < 10; ++i) l.push_back(i);
    cout << "l of size " << l.size() << ": "; print(l); cout << endl;
    cout << "Print the list again..." << endl;
    cout << "l of size " << l.size() << ": "; print(l); cout << endl;
    cout << "Double every element..." << endl;
    for (int &x : l) x *= 2;
    cout << "l of size " << l.size() << ": "; print(l); cout << endl;
    cout << "--------------------------------------------------" << endl;
    bstree<int> t;
    int x[] = { 5, 2, 8, 4, 1, 7, 6, 0, 9, 3 };
    for (int i = 0; i < 10; ++i) t.insert(x[i]);
  ```

```
  cout << "t of size " << t.size() << ": "; print(t); cout << endl;
  cout << "Print the tree again..." << endl;
  cout << "t of size " << t.size() << ": "; print(t); cout << endl;
  for (int &x : t) x *= 2;
  cout << "Double every element..." << endl;
  cout << "t of size " << t.size() << ": "; print(t); cout << endl;
}
```

Το παραπάνω πρόγραμμα, όταν εκτελείται, σωστά εκτυπώνει:

```
l of size 10: 0 1 2 3 4 5 6 7 8 9
Print the list again...
l of size 10: 0 1 2 3 4 5 6 7 8 9
Double every element...
l of size 10: 0 2 4 6 8 10 12 14 16 18
------------------------------------------------
t of size 10: 0 1 2 3 4 5 6 7 8 9
Print the tree again...
t of size 10: 0 1 2 3 4 5 6 7 8 9
Double every element...
t of size 10: 0 2 4 6 8 10 12 14 16 18
```

Επισκέπτες

1. Οι iterators μάς παρέχουν έναν τρόπο να διατρέξουμε αφαιρετικά τα στοιχεία ενός container και να κάνουμε κάποια πράξη με καθένα από αυτά. Αυτό είναι το **iterator design pattern**. Ας προσπαθήσουμε τώρα να καταλάβουμε ακριβώς ποιο είναι το είδος της αφαίρεσης που μας παρέχει.

   Έστω ότι θέλουμε να αθροίσουμε τα στοιχεία μίας διπλά συνδεδεμένης λίστας που περιέχει ακέραιους αριθμούς. Αν υποθέσουμε ότι η παρακάτω συνάρτηση `sum` έχει πρόσβαση στα ιδιωτικά μέρη της λίστας (π.χ. είναι ορισμένη ως `friend` της κλάσης `dlist<T>`) τότε η υλοποίησή της θα μπορούσε να είναι η ακόλουθη, χρησιμοποιώντας απευθείας ένα βρόχο που διατρέχει τη λίστα μετακινώντας έναν δείκτη από κόμβο σε κόμβο:

   ```
   int sum(const dlist<int> &l) {
     int s = 0;
     for (dlist<int>::node *p = l.the_front; p != nullptr; p = p->next)
       s += p->data;
     return s;
   }
   ```

   Με τη χρήση iterator, η συνάρτηση αυτή γίνεται:

   ```
   int sum(Iterable<int> &c) {
     int s = 0;
     for (Iterator<int> i = c.begin(); i != c.end(); ++i)
       s += *i;
     return s;
   }
   ```

   δηλαδή η αφαίρεση που μας παρέχει ο iterator κρύβει τις λεπτομέρειες της διάσχισης της λίστας, αφήνοντας όμως εκτεθειμένες τις λεπτομέρειες της άθροισης. Η πράξη που θέλουμε να εφαρμόσουμε (εν προκειμένω η συνάρτηση `sum`) δέχεται ως παράμετρο τον container και χρησιμοποιεί τον αντίστοιχο iterator για να ολοκληρώσει το έργο της.

   Ας φανταστούμε ότι ο container είναι ένα ξενοδοχείο, στα δωμάτια του οποίου διαμένουν κάποιοι ένοικοι, και ότι θέλουμε να κάνουμε μία πράξη με κάθε έναν από αυτούς. Τότε ο iterator κάνει τη δουλειά ενός υπαλλήλου στη ρεσεψιόν του ξενοδοχείου που αναλαμβάνει να φέρνει στην είσοδο έναν-έναν τους ενοίκους (διατρέχοντας τα δωμάτια με κάποια κατάλληλη σειρά) ούτως ώστε να μπορέσουμε να τους εξετάσουμε.

2. Στη συνέχεια αυτής της ενότητας θα εστιάσουμε στο αντίστροφο. Θα κατασκευάσουμε μία αφαίρεση που θα κρύβει τις λεπτομέρειες της πράξης που θέλουμε να εφαρμόσουμε πάνω στα στοιχεία ενός container (της άθροισης, εν προκειμένω) και θα αφήνει εκτεθειμένες τις λεπτομέρειες της διάσχισης των στοιχείων του container.

   Η αντίστοιχη αφαίρεση, αν τη μεταφέρουμε στην αναλογία του ξενοδοχείου, είναι ένας **επισκέπτης** ο οποίος εμφανίζεται στη ρεσεψιόν και ζητάει να επισκεφθεί όλους τους ενοίκους. Ο υπάλληλος της ρεσεψιόν αναλαμβάνει να τον πάει από δωμάτιο σε δωμάτιο (διατρέχοντας πάλι τα δωμάτια με κάποια κατάλληλη σειρά) ούτως ώστε ο επισκέπτης να μπορέσει να εξετάσει τους ενοίκους.

Οι ρόλοι εδώ αντιστρέφονται: ο container έχει μία μέθοδο `accept` που δέχεται ως παράμετρο έναν τέτοιον επισκέπτη και αναλαμβάνει να τον "ξεναγήσει" στα στοιχεία που περιέχονται στον container. Ο ίδιος ο επισκέπτης διαθέτει μία μέθοδο `visit` που είναι η αφαίρεση της πράξης που πρέπει να εφαρμοστεί πάνω στα στοιχεία του container.

3. Κατ' αντιστοιχία με τις κλάσεις `Iterable<T>` και `Iterator<T>` που ορίσαμε πριν για να υλοποιήσουμε τους iterators, για την υλοποίηση του **visitor design pattern** θα χρησιμοποιήσουμε δύο κλάσεις: `Visitable<T>` και `Visitor<T>`. Αυτή τη φορά, και οι δύο είναι αφηρημένες κλάσεις.

```
template <typename T>
class Visitor {
public:
  virtual ~Visitor() {}
  virtual void visit(T &x) = 0;
  virtual bool finished() const { return false; }
};

template <typename T>
class Visitable {
public:
  virtual void accept(Visitor<T> &visitor) = 0;
};
```

Όπως είπαμε ήδη, η κλάση `Visitable<T>` διαθέτει μία μέθοδο `accept` που παίρνει ως παράμετρο έναν `Visitor<T>`. Η κλάση `Visitor<T>` με τη σειρά της διαθέτει μία μέθοδο `visit` που καλείται κάθε φορά που ο επισκέπτης πρέπει να επισκεφθεί κάποιο συγκεκριμένο στοιχείο του container.

Προκειμένου ένας επισκέπτης να μπορεί να τελειώσει πρόωρα την επίσκεψή του χωρίς να επισκεφθεί όλα τα περιεχόμενα ενός container (π.χ. στην περίπτωση που υλοποιεί κάποια αναζήτηση, όπως θα δούμε σε ένα παράδειγμα στη συνέχεια), ορίζουμε μία μέθοδο `finished` που επιστρέφει `true` αν ο επισκέπτης έχει ολοκληρώσει το έργο του. Η default υλοποίησή της επιστρέφει πάντα `false`.

4. Ένας επισκέπτης που αθροίζει ακεραίους θα μπορούσε να υλοποιηθεί ως εξής. Διατηρεί ένα ιδιωτικό πεδίο `sum`, στο οποίο αθροίζει τις τιμές των στοιχείων τα οποία επισκέπτεται.

```
class AddingVisitor: public Visitor<int> {
public:
  AddingVisitor() : sum(0) {}
  void visit(int &x) override { sum += x; }
  int getSum() const { return sum; }
private:
  int sum;
};
```

και η χρήση του σε ένα πρόγραμμα θα ήταν ως εξής (έχουμε υποθέσει ότι η κλάση `dlist<T>` κληρονομεί την `Visitable<T>`):

```
int main() {
  dlist<int> l;
  for (int i = 1; i <= 5; ++i) l.push_back(i);
  AddingVisitor v;
  l.accept(v);
  cout << "The sum is " << l.getSum() << endl;
}
```

5. Ως ένα δεύτερο παράδειγμα επισκέπτη, μπορούμε να υλοποιήσουμε μία γενική συνάρτηση `visit_and_print` που να εκτυπώνει τα στοιχεία ενός container χρησιμοποιώντας το visitor design pattern, αντί των iterators. Ορίζουμε πρώτα μία κλάση `PrintingVisitor<T>` που εκτυπώνει στο `cout` όλα τα στοιχεία που επισκέπτεται:

```
template <typename T>
class PrintingVisitor: public Visitor<T> {
public:
  void visit(T &x) override { cout << x << endl; }
};
```

Τέλος, ορίζουμε τη συνάρτηση `visit_and_print` που επιτελεί την ίδια λειτουργία με την `print` για κάθε `Visitable<T>`, χρησιμοποιώντας όμως ένα `PrintingVisitor<T>`:

```
template <typename T>
void visit_and_print(Visitable<T> &c) {
  PrintingVisitor<T> v;
  c.accept(v);
}
```

Το παρακάτω πρόγραμμα καλεί αυτή τη συνάρτηση για να εκτυπώσει τα στοιχεία μίας διπλά συνδεδεμένης λίστας:

```
int main() {
  dlist<int> l;
  for (int i = 1; i <= 5; ++i) l.push_back(i);
  cout << "l of size " << l.size() << ":" << endl;
  visit_and_print(l);
}
```

6. Ένα τρίτο και τελευταίο παράδειγμα επισκέπτη είναι ο ακόλουθος, ο οποίος αναζητά σε έναν container μέχρι να βρει ένα στοιχείο που να ικανοποιεί κάποια ιδιότητα (στην προκειμένη περίπτωση, αν υπάρχει κάποιος ακέραιος με τιμή μεταξύ 17 και 42). Ο επισκέπτης θυμάται αν έχει βρει ήδη ένα τέτοιο στοιχείο (πεδίο `found`) και ποια ήταν η τιμή του (πεδίο `value`):

```
class PZSearchingVisitor: public Visitor<int> {
public:
  PZSearchingVisitor() : found(false) {}
  void visit(int &x) override {
    if (17 <= x && x <= 42) { found = true; value = x; }
  }
  bool finished() const override { return found; }
  bool getFound() const { return found; }
  int getValue() const { return value; }
private:
  bool found;
  int value;
};
```

Μπορεί να χρησιμοποιηθεί σε ένα πρόγραμμα ως εξής:

```
int main() {
  bstree<int> t;
  int x[] = { 15, 72, 48, 14, 1, 37, 66, 50, 9, 83 };
  for (int i = 0; i < 10; ++i) t.insert(x[i]);
  PZSearchingVisitor pz;
  t.accept(pz);
  if (pz.getFound())
    cout << "I found the number " << pz.getValue() << endl;
}
```

7. Στα τρία παραπάνω παραδείγματα φαίνεται πολύ καθαρά ότι, σύμφωνα με το visitor design pattern, οι λεπτομέρειες της διάσχισης του container κρύβονται μέσα στη μέθοδο `accept` (για την ακρίβεια, δεν τις έχουμε ακόμα δει), ενώ οι λεπτομέρειες της πράξης που πρέπει να εφαρμοστεί στα στοιχεία του container κρύβονται μέσα στο αντικείμενο του επισκέπτη και στη μέθοδο `visit`.

8. Για να υποστηρίζουν οι containers μας το visitor design pattern, πρέπει να τους υλοποιήσουμε ώστε να κληρονομούν την `Visitable<T>`. Στην περίπτωση της κλάσης `dlist<T>`, αυτό μπορεί να γίνει ως εξής:

```
template <typename T>
class dlist: public Container<T>, public Iterable<T>, public Visitable<T> {
  ...
public:
  void accept(Visitor<T> &visitor) override {
    for (node *p = the_front; p != nullptr; p = p->next) {
      if (visitor.finished()) return;
      visitor.visit(p->data);
    }
  }
};
```

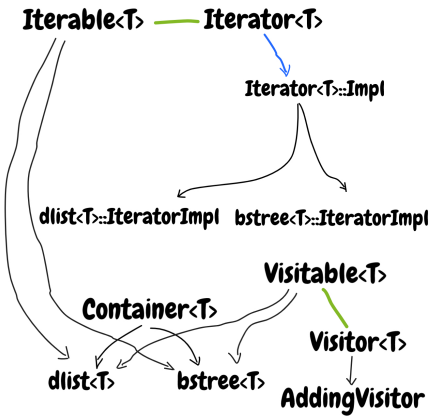
9. Στην περίπτωση της κλάσης `bstree<T>` μπορούμε να χρησιμοποιήσουμε μία κλασική αναδρομική διάσχιση του δέντρου, με μια στατική βοηθητική μέθοδο.

```
template <typename T>
class bstree: public Container<T>, public Iterable<T>, public Visitable<T> {
  ...
public:
  void accept(Visitor<T> &visitor) override { accept(visitor, root); }

private:
  static void accept(Visitor<T> &visitor, node *t) {
    if (t != nullptr) {
      accept(visitor, t->left);
      if (visitor.finished()) return;
      visitor.visit(t->data);
      accept(visitor, t->right);
    }
  }
};
```

   Η παραπάνω υλοποίηση ακολουθεί ενδοδιατεταγμένη διάσχιση (in-order) αλλά, εν αντιθέσει με την υλοποίηση των iterators που είναι σημαντικά πιο περίπλοκη, εδώ είναι εξαιρετικά απλή η μετατροπή της ώστε να ακολουθείται προδιατεταγμένη (pre-order) ή μεταδιατεταγμένη (post-order) διάσχιση.

10. Στο Σχήμα 2 φαίνεται η ιεραρχία των περισσότερων από τις κλάσεις που ορίσαμε σε αυτά τα μαθήματα. Τα μαύρα βέλη υποδηλώνουν κληρονομικότητα, το μπλε βέλος υποδηλώνει μία σχέση ιδιοκτησίας (ότι δηλαδή το `Iterator<T>` περιέχει ένα δείκτη σε ένα "δικό του" αντικείμενο `Iterator<T>::Impl`) και η πράσινη γραμμή μια σχέση συνεργασίας μεταξύ κλάσεων.



Σχήμα 2: Η ιεραρχία των κλάσεων που ορίσαμε