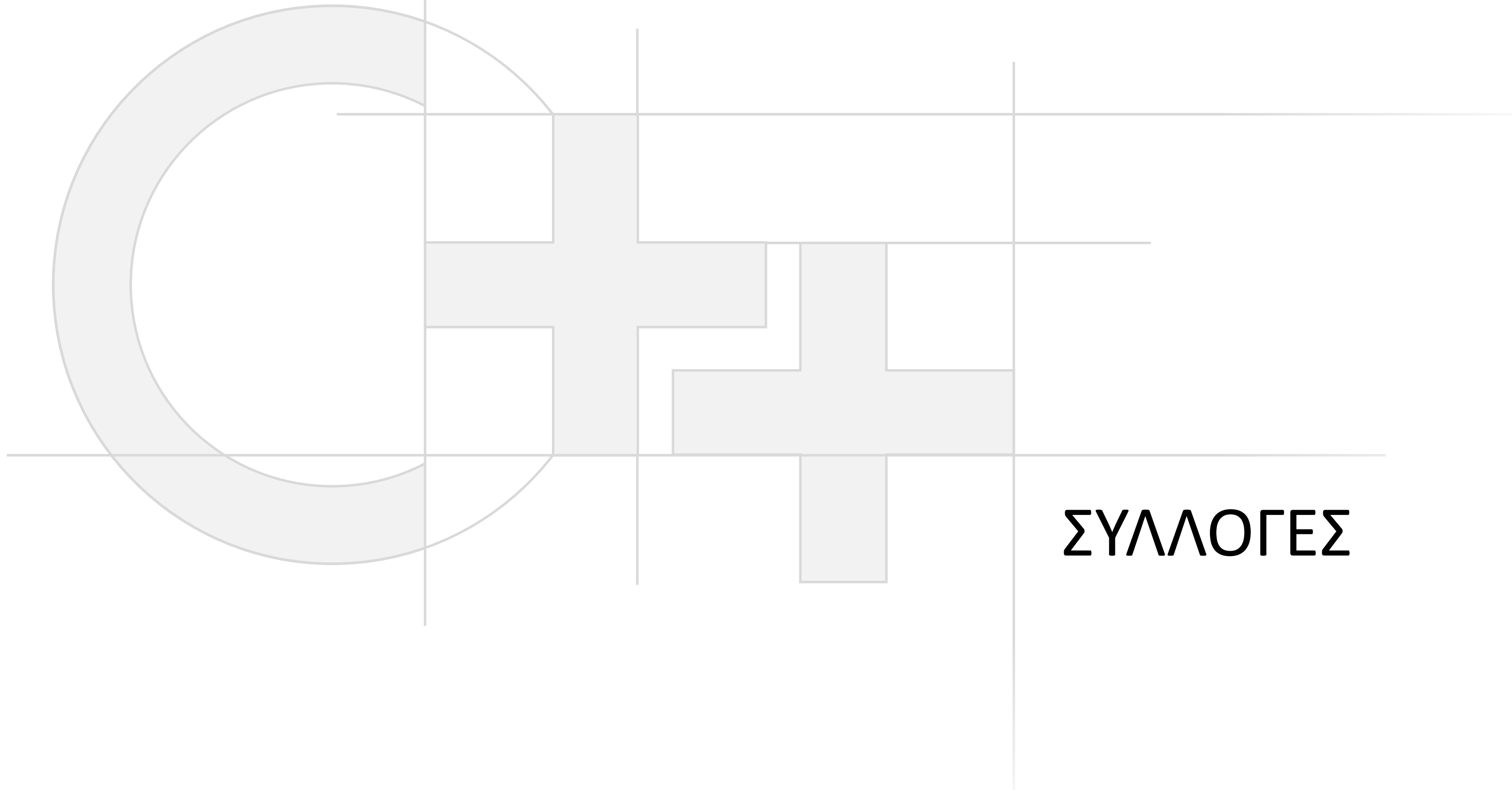
Προγραμματισμός Υπολογιστών με C++



ΣΥΛΛΟΓΕΣ

Γεώργιος Παπαϊωάννου (2013-16)
gepap@aueb.gr

Περιεχόμενο Παρουσίασης

- Περιγραφή:
  - Βασικές Συλογές της STL
  - Η έννοια του iterator
  - Τροποποίηση και παραμετροποίηση συλλογών της STL
    - Έλεγχος ισοτιμίας και διάταξης
    - Hashing
  - Υλοποίηση δικών μας δομών (Stack<T>)
  - Iterators σε δικές μας δομές και συλλογές (Stack<T>::iterator)
- Τελευταία ενημέρωση: Νοέμβριος 2016

Ρεπερτόριο Δομών της STL

- vector<T>
- list<T> (και forward_list<T> (C++11))
- deque<T> (**d**ouble-**e**nded **que**ue, προφ. «ντεκ»)
- stack<T>
- queue<T>
- set<T> (και multiset<T>)
- map<Key,T> (και multimap<Key,T>)
- priority_queue<T>
- unordered_set<T> (και unordered_multiset<T>) (C++11)
- unordered_map<Key,T> (και unordered_multimap<Key,T>)(C++11)

Ποια Συλλογή να Χρησιμοποιήσω;

- Οι συλλογές διαφοροποιούνται με βάση:
  - Τη λειτουργικότητά τους
    - Θέλω FIFO, LIFO, τυχαία προσπέλαση;
    - Θέλω ταξινομημένα στοιχεία;
    - Θέλω μοναδικά στοιχεία;
  - Την ταχύτητα που θέλω σε κάποια ενέργεια πάνω σε αυτές
    - Θέλω γρήγορη σειριακή προσπέλαση;
    - Θέλω γρήγορη εισαγωγή στην αρχή / τέλος;
    - Θέλω γρήγορες διαγραφές/εισαγωγές σε τυχαία θέση;
    - …
- Δεν υπάρχει «η» βέλτιστη δομή συλλογής! Υπάρχει η βέλτιστη «για μια συγκεκριμένη δουλειά»

Υλοποιήσεις Συλλογών

- Κάθε συλλογή βασίζεται σε μια συγκεκριμένη δομή δεδομένων που της προσδίδει ειδικά πλεονεκτήματα / μειονεκτήματα απόδοσης, π.χ.
  - Πίνακες
  - Συνδεδεμένες λίστες
  - Hash maps
  - Red-Black trees

Η Συλλογή std::vector (#include <vector>)

- Ίδια λογική με τη Java: δυναμική γραμμική δομή τυχαίας προσπέλασης (δυναμικός πίνακας)
  – Εγγυάται τη διατήρηση της σειράς εισαγωγής των δεδομένων
- Επιδόσεις:
  – Γρήγορη σειριακή διάσχιση
  – Γρήγορη εισαγωγή/διαγραφή στο τέλος
  – Αργή εισαγωγή/διαγραφή στην αρχή ή ενδιάμεσα
  – Συμπαγής αποθήκευση (συνεχόμενη μνήμη) → cache coherence, απλό indexing
  – Δυναμική επέκταση μεγέθους, όταν υπερβούμε τον μέγιστο προ-εκχωρημένο χώρο

Η Συλλογή std::list (#include <list>)

- Είναι μια διπλά συνδεδεμένη λίστα από κόμβους που δεν είναι απαραίτητα συνεκτικοί στη μνήμη
- Υλοποιεί πράξεις εισαγωγής και διαγραφής και από τα δύο άκρα, καθώς και σε συγκεκριμένους κόμβους
- Επιδόσεις:
  - Μέτρια σειριακή διάσχιση (συγκριτικά με το vector) λόγω μετακίνησης μέσω δείκτη και κατακερματισμού της μνήμης
  - Αργή τυχαία προσπέλαση (γίνεται ακολουθιακά)
  - Γρήγορη διαγραφή και εισαγωγή σε οποιοδήποτε σημείο

Η Συλλογή std::deque (#include <deque>)

- Σειριακές συλλογές που επιτρέπουν αποδοτική εισαγωγή και διαγραφή στα δύο άκρα (αντίθετα με το vector στον οποίο είναι γρήγορη μόνο στο τέλος)
- Υλοποιούνται με δυναμικούς πίνακες
- Δεν εγγυώνται την αποθήκευση όλων των δεδομένων σε ένα συνεχόμενο πίνακα στη μνήμη
- Βασική δομή wrapper συλλογών όπως η stack και η queue
- Επιδόσεις:
  - Σειριακή και τυχαία προσπέλαση συγκρίσιμη με του vector
  - Γρήγορη εισαγωγή / διαγραφή στην αρχή και στο τέλος
  - Αργή εισαγωγή / διαγραφή σε τυχαίο σημείο

Η Συλλογή std::set (#include <set>)

- Υλοποιεί ένα ταξινομημένο σύνολο με μοναδικά στοιχεία
- Τα στοιχεία διαβάζονται ταξινομημένα, ανεξάρτητα από τη σειρά εισαγωγής τους
- Επιτρέπει ελέγχους ύπαρξης στοιχείου (find())
- Προσαρμόσιμος έλεγχος ισοτιμίας
- Επιδόσεις:
  - Είναι δυαδικά δένδρα, «σχεδόν» ισοζυγισμένα (π.χ. RB-trees)
  - Γρήγορη αναζήτηση (όχι όμως O(1))
  - Μέτρια γρήγορη (amortized) εισαγωγή και διαγραφή παντού

Η Συλλογή std::map (#include <map>)

- Υλοποιεί ένα ταξινομημένο σύνολο ζευγών κλειδιών-τιμών
- Τα στοιχεία διαβάζονται ταξινομημένα, ανεξάρτητα από τη σειρά εισαγωγής τους
- Επιτρέπει ελέγχους ύπαρξης στοιχείου (find())
- Προσαρμόσιμος έλεγχος ισοτιμίας
- Επιδόσεις:
  - Υλοποιημένα ως δυαδικά δέντρα αναζήτησης
  - Μέτρια γρήγορη εισαγωγή/αναζήτηση/διαγραφή με κλειδί

Η Συλλογή std::map - Παράδειγμα

```
#include <map>
int main() {
  map<string,int> data;
  data.insert(pair<string,int>("Papaioannou",8));
  data["Tsatiris"]=9;

  map<string,int>::iterator result = data.find("Tsatiris");

  if (result!=data.end())
     cout << result->first << " grade is " << result->second << endl;

  return 0;
}
// Τυπώνει: Tsatiris grade is 9
```

Ισοδύναμο της insert

Επιστρέφει iterator: `map<string,int>::iterator`

Έλεγχος ύπαρξης στοιχείου

`<string,int>` first:Το κλειδί

`<string,int>` second: Η τιμή

Η Συλλογή unordered_map (#include <unordered_map)

- Υλοποιεί ένα hash table
- Τα στοιχεία δε διαβάζονται ταξινομημένα
- Επιτρέπει ελέγχους ύπαρξης στοιχείου (find())
- Προσαρμόσιμος έλεγχος ισοτιμίας
- Προσαρμόσιμο hash code
- Επιδόσεις:
  - Υλοποιημένα ως τυπικό hash table
  - Γρήγορη εισαγωγή, διαγραφή, αναζήτηση (εξαρτάται από την ποιότητα του hash code)
  - Μέτρια διάσχιση στοιχείων

Container Adapters

- Κάποιες συλλογές είναι adapters άλλων, δηλαδή διαμορφώνουν τη λειτουργικότητα βασικότερων συλλογών για να προσδώσουν ειδικά χαρακτηριστικά
  - π.χ.: η queue είναι εσωτερικά ένα deque ()
- Τέτοιες είναι οι stack, queue, priority_queue
- Μπορούμε να αλλάξουμε την εσωτερική δομή με άλλη που μας κάνει καλύτερα για ένα πρόβλημα (ή δική μας!):
  - Αρκεί να παρέχει τις μεθόδους που καλεί η εξωτερική κλάση
  - Π.χ.: template <class T, class Container = deque<T> > class queue →

    template <typename T> class MyQueue : public queue<T,list<T>> {};

Iterators

- Όλες οι συλλογές υποστηρίζουν κάποια μορφή iterator
- Ο iterator είναι μια κλάση που μας επιτρέπει να διασχίζουμε μια συλλογή διαβάζοντας το επόμενο (το προηγούμενο ή και τυχαίο) στοιχείο από αυτή
  - Παρέχει ένα ενιαίο Interface για τη διάσχιση συλλογών, ανεξάρτητα από την εσωτερική τους αναπαράσταση
  - Η ακριβής λειτουργία που επιτρέπεται να κάνει ο Iterator εξαρτάται από το είδος του (forward iterator, bidirectional iterator, random iterator)
- Κάθε συλλογή παράγει μια εξειδίκευση αυτής της κλάσης και μάλιστα μέσω templates

Δήλωση ενός Iterator

```
map<string,int> grades;
grades["Tsatiris"]=8;
grades["Tampourlos"]=9;
grades["Papapanagiotou"]=6;
grades["Makris"]=5;

map<string,int>::iterator result;

typedef map<string,int>::iterator grade_iterator;

for (grade_iterator iter = grades.begin(); iter!= grades.end(); ++iter)
   cout << iter->first << " grade is " << iter->second << endl;
```

C++11: Αυτόματος τύπος (εξάγεται από τον compiler με βάση τον τύπο του δεξιού μέλους της ανάθεσης)

C++11: “for each … in” operator

```
for (auto iter : grades)
   cout << iter->first << " grade is " << iter->second << endl;
```

begin(), end()

- Όλες οι συλλογές διαθέτουν τουλάχιστον δύο μεθόδους επιστροφής τύπου iterator στα "άκρα" τους:
  - begin(): iterator που «δείχνει» στο πρώτο διαθέσιμο στοιχείο. Το «πρώτο» δεν έχει να κάνει με τη φυσική διάταξη των στοιχείων στη μνήμη, αλλά καθορίζεται από τη λειτουργία της δομής
  - end(): iterator σε μια άκυρη θέση («τέρμα») στη συλλογή
- Παρατήρηση: Οι iterators ΔΕΝ είναι pointers! Είναι στιγμιότυπα τύπου iterator
  - Όμως συνήθως κάνουν override τον τελεστή * για ομοιομορφία (περισσότερα στους Custom Iterators)

Παραμετροποίηση Συλλογών

- Εκτός από τους τύπους που αποθηκεύονται μέσα σε μια συλλογή, επιτρέπεται να τροποποιήσουμε (μεταξύ άλλων) και:
  - Την κλάση που προσδιορίζει τον τελεστή σύγκρισης για την ταξινόμηση ή ισοτιμία
  - Τη γεννήτρια hash codes (όπου υπάρχει)
- Προφανώς, μπορούμε
  - Να επεκτείνουμε μια από τις υπάρχουσες συλλογές ή
  - Να ορίσουμε δικές μας συλλογές και δικούς μας iterators

Συλλογές: Αλλαγή Σύγκρισης

- Στις συλλογές που δεν ορίζεται διάταξη (π.χ. unordered_map), μπορούμε να αλλάξουμε την πράξη της ισοτιμίας equal_to (==)
- Στις συλλογές που ορίζεται διάταξη των στοιχείων (ταξινομημένες, π.χ. set) μπορούμε να αλλάξουμε τη συμπεριφορά της πράξης less (<)
- Έμμεσα, στη δεύτερη περίπτωση προσδιορίζεται και η ισοτιμία: equal_to(A,B):= !A<B && !B<A

Συλλογές: Αλλαγή Σύγκρισης – Κλάση Σύγκρισης

```
class Bottle {
private:
  float volume;
public:
  Bottle(float vol): volume(vol) {};
  inline const float getVolume() const {return volume;}
};

class BottleCompare {
  public: bool operator() (const Bottle& x, const Bottle& y) const {
    return x.getVolume()<y.getVolume();
  }
};

int main() {
  set<Bottle,BottleCompare> bottle_types;

  bottle_types.insert(Bottle(500.0f));  bottle_types.insert(Bottle(330.0f));
  bottle_types.insert(Bottle(330.0f));  bottle_types.insert(Bottle(200.0f));

  for (auto iter=bottle_types.begin(); iter!=bottle_types.end(); iter++) {
    Bottle type = *iter;
    cout << type.getVolume() << " " << endl;
  }

  return 0;
}
```

Συλλογές: Αλλαγή Σύγκρισης – Custom Τελεστής <

```
class Bottle {
private:
  float volume;
public:
  Bottle(float vol): volume(vol) {};
  const float getVolume() {return volume;}
  const bool operator < (const Bottle & right) const {
    return volume < right.volume;
  }
};

int main() {
  set<Bottle> bottle_types;

  bottle_types.insert(Bottle(500.0f));  bottle_types.insert(Bottle(330.0f));
  bottle_types.insert(Bottle(330.0f));  bottle_types.insert(Bottle(200.0f));

  for (auto iter=bottle_types.begin(); iter!=bottle_types.end(); iter++) {
    Bottle type = *iter;
    cout << type.getVolume() << " " << endl;
  }
  return 0;
}
```

Αλλαγή της Ισοτιμίας – Παράδειγμα (1)

- Έστω ότι θέλουμε να κρατάμε σε ένα unordered_map ζεύγη με την ιδιότητα της κυκλικής ολίσθησης (π.χ. τύπου string), δηλαδή τα κλειδιά "abcd", "dabc", "cdab" και "bcda" να θεωρούνται «ίσα».
- Ένας τρόπος να το κάνουμε αποδοτικά αυτό είναι να κατασκευάσουμε μια κλάση σύγκρισης που να εξασφαλίζει ακριβώς αυτό και να την περάσουμε ως όρισμα στο unordered_map

Αλλαγή της Ισοτιμίας – Παράδειγμα (2)

```
class equals
{
public:
  bool operator() (const string& t1, const string& t2) const
  {
    size_t sz1 = t1.size();
    size_t sz2 = t2.size();
    if (sz1!=sz2)
      return false;

    string temp = t1;
    for (unsigned int i=0; i<sz1; i++)
    {
      std::rotate(temp.begin(),temp.begin()+1,temp.end());
      if ( !temp.compare(t2) )
        return true;
    }
    return false;
  }
};
```

Ναι, είμαστε τυχεροί, ένα string είναι κι αυτό συλλογή! Μπορούμε να εφαρμόσουμε πάνω του έναν από τους πολλούς έτοιμους αλγόριθμους που ισχύουν για επεξεργασία συλλογών (εδώ rotate: κυκλική ολίσθηση – C++11)

Αλλαγή της Ισοτιμίας – Παράδειγμα (3)

```
int main()
{
   unordered_map<string,int,hash<string>,equals> data;

   data.rehash(32);

   data["abcd"] = 100;
   data["bcda"] = 200;

   for (auto iter=data.begin(); iter!=data.end(); iter++)
   {
      cout << iter->second << " " << endl;
   }

}

// τυπώνει: 100
//          200
```

Ορίζουμε ένα unordered_map με κλειδιά string, τιμές int, τη default hashing κλάση (std::hash) και δική μας κλάση ισοτιμίας (καλείται ο τελεστής της operator())

Ορίζουμε τον αριθμό των buckets…

Βάζουμε 2 στοιχεία με ισότιμο κλειδί…

Και περιμένουμε να μη βάλει 2 διαφορετικά δεδομένα, αλλά να αντικαταστήσει την τιμή στο υπάρχον κλειδί

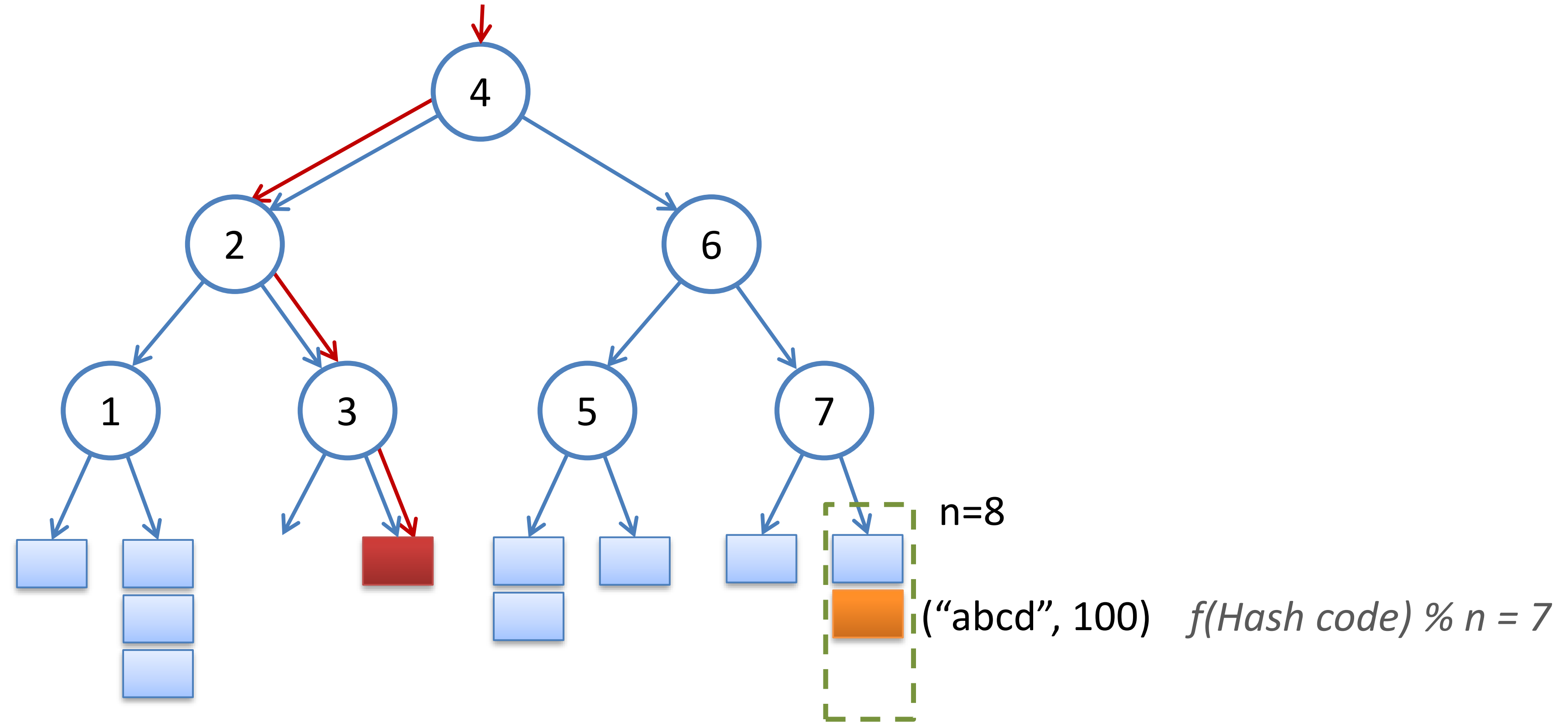
… αλλά δε συμβαίνει αυτό! Τι πήγε στραβά;

Αλλαγή της Ισοτιμίας – Παράδειγμα (4)

Τι πήγε στραβά;

- Για να φτάσει το unordered_map να εξετάσει την ισοτιμία, πρέπει να κάνει τα ακόλουθα βήματα:
    1. Υπολογισμός του hash code του νέου κλειδιού
    2. Εύρεση του κατάλληλου κουβά που αντιστοιχεί στο hash code (f(hash code) mod $N_{buckets}$)
    3. Αναζήτηση (γραμμική) μέσα στον κουβά, εξετάζοντας την ισοτιμία με όλα τα στοιχεία του κουβά
- Στο παράδειγμά μας, δεν έφτασε ποτέ στο βήμα 3!
- Το βήμα 2 που χρησιμοποιεί τον προεπιλεγμένο έλεγχο ισοτιμίας για strings (==) έβγαλε άλλο hash code και άρα η αναζήτηση έγινε σε λάθος κουβά

Αλλαγή της Ισοτιμίας – Παράδειγμα (5)



Σωστός κουβάς για να γίνει ο έλεγχος ισοτιμίας και να δώσει true

Αλλαγή της Ισοτιμίας – Παράδειγμα (6)

```
class hashing
{
public:
  std::size_t operator()(const string& key) const
  {
    size_t _hash = 0;
    size_t sz = key.size();
    char max = 0;

    for (unsigned int i=0; i<sz; i++)
    {
      _hash += 71*(255+key[(i+1)%sz] - key[i]);
      max = max<key[i] ? key[i] : max;
    }
    _hash+=max;
    return _hash;
  }
};
```

Κατασκευάζουμε ένα hash code που να είναι αναλλοίωτο στις ολισθήσεις: βασισμένο στο άθροισμα των διαφορών των διαδοχικών στοιχείων. Έτσι, ο κωδικός για το “abcd” και το “cdab” είναι ο ίδιος

Προσθέτουμε το μέγιστο στοιχείο του string (αριθμητικά) για να διαφέρει ο κωδικός π.χ. του “abcd” από τον “bcde”

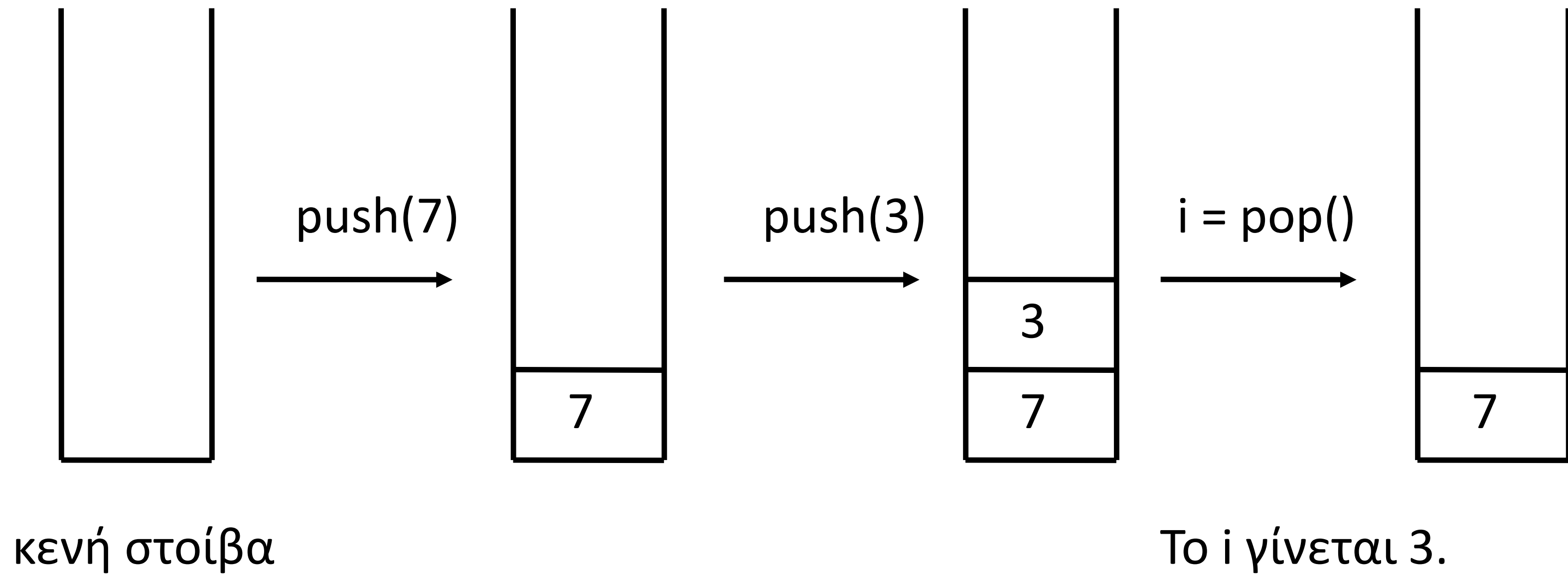
Αλλαγή της Ισοτιμίας – Παράδειγμα (7)

```
int main()
{
   unordered_map<string, int, hashing, equals> data;

   data.rehash(32);

   data["abcd"] = 100;
   data["bcda"] = 200;


   for (auto iter=data.begin(); iter!=data.end(); iter++)
   {
      cout << iter->second << " " << endl;
   }

}


// τυπώνει: 200
```

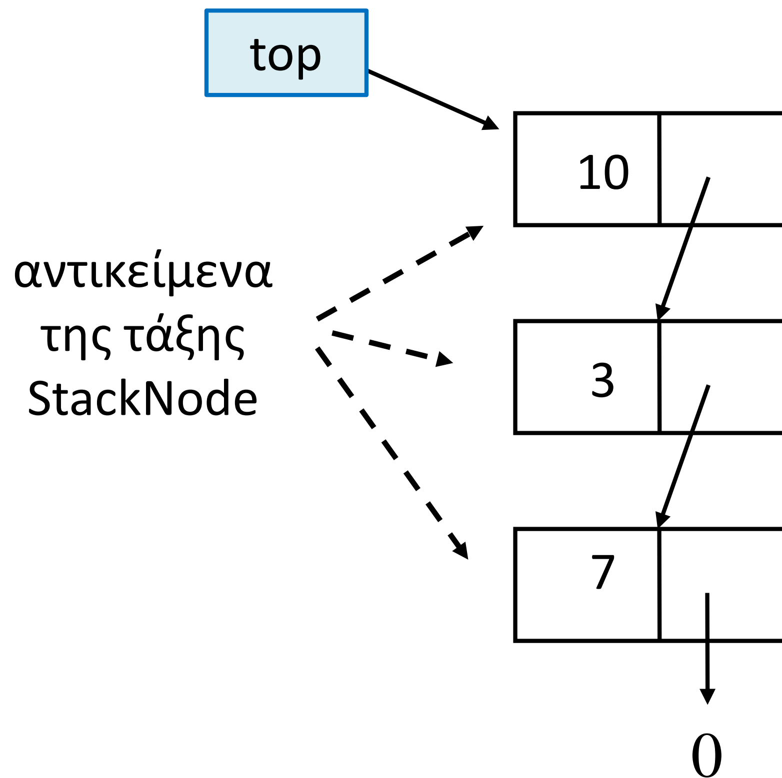
Παρατήρηση: Αρκεί να περάσουμε οποιαδήποτε κλάση που υλοποιεί τον τελεστή
`std::size_t operator()(const T1& key) const`

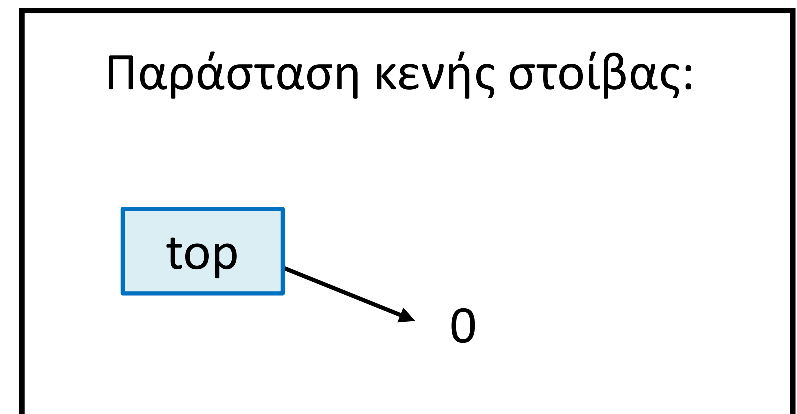
Όπου `T1` ο τύπος του `unordered_map <T1, T2, Hash, Compare>`

Δικές μας Συλλογές – Παράδειγμα: Stack



Μια Πρώτη Υλοποίηση της Στοίβας



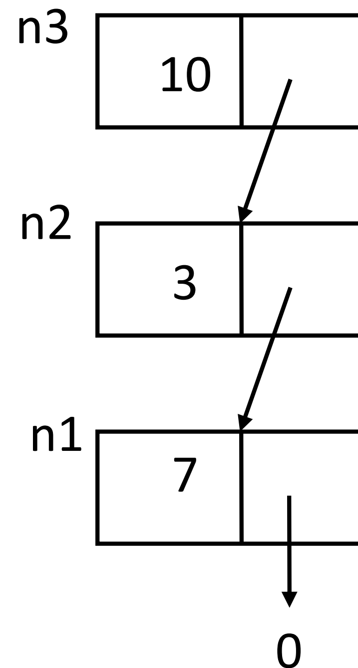


Η Κλάση StackNode

```
template <typename T>
class StackNode {
   const T i;
   const StackNode* const next;
public:
   T getValue() const { return i; }
   const StackNode* const getNext() const { return next; }
   StackNode(const T iIn, const StackNode* const nextIn = 0) :
      i(iIn), next(nextIn) {}
};
```

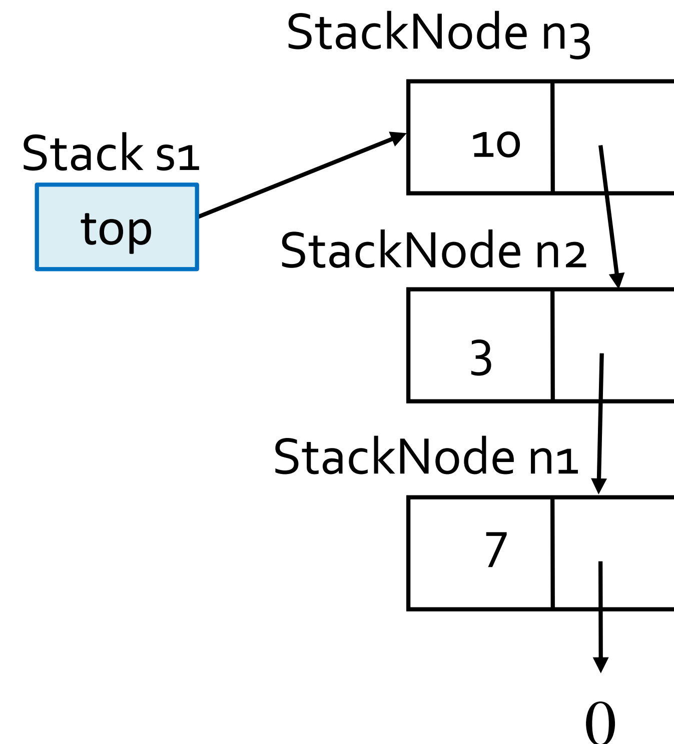
Παράδειγμα Χρήσης της StackNode

```
int main() {
    StackNode<int> n1(7), n2(3, &n1), n3(10, &n2);
}
```



Υλοποίηση μιας Stack

- Θέλουμε να μπορούμε να αποθηκεύσουμε οτιδήποτε μέσα στη στοίβα μας → templated
- Θέλουμε να υποστηρίζεται η έννοια της άδειας στοίβας και να μπορούμε να καλέσουμε μεθόδους σε αυτή → χρειαζόμαστε μια δομή γενικότερη της StackNode που να περιλαμβάνει StackNode(s)



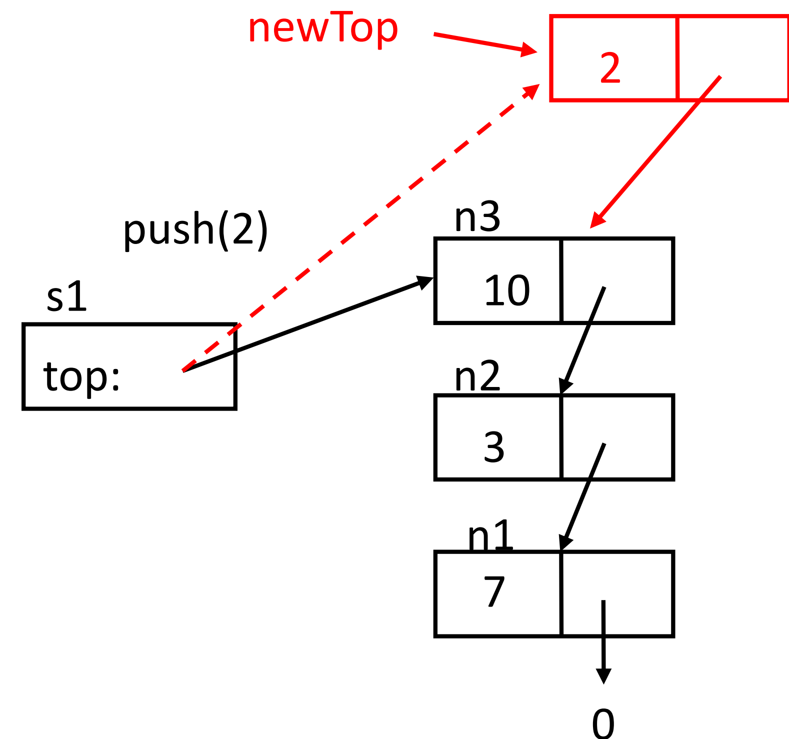
Η Κλάση Stack

```
template <typename T = int>
class Stack
{
   const StackNode<T>* top;
   void deleteStackNodes();
   static const StackNode<T>* const copyStackNodes(
      const StackNode<T>* const originalPtr);
public:
   bool isEmpty() const { return (top == nullptr); }
   void push(T i);
   T pop();
   Stack() : top(nullptr) {}
   ~Stack() { deleteStackNodes(); }
   Stack(const Stack& original);
   Stack& operator=(const Stack<T>& right);
};
```

default τύπος, αν δεν προσδιοριστεί κάποιος άλλος. Χρήση: Stack<> mystack

Stack: H Μέθοδος push

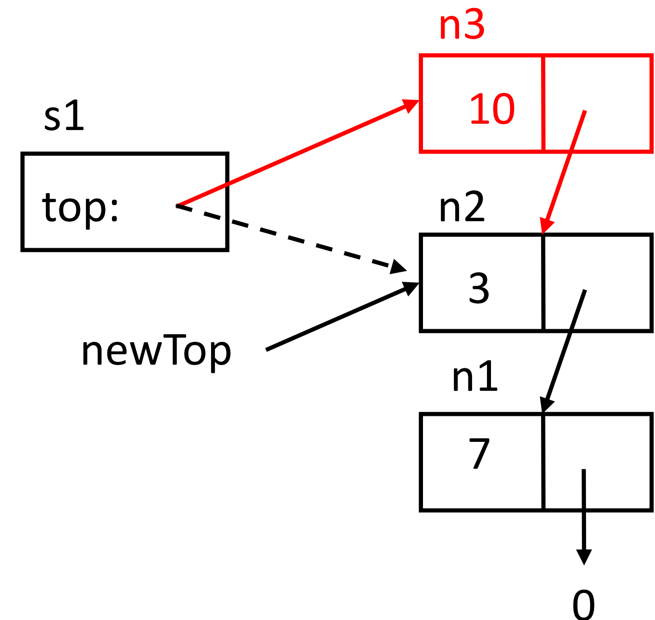
```
template <typename T>
void Stack<T>::push(const T i)
{
   const StackNode<T>* const newTop = new StackNode<T>(i, top);
   top = newTop;
}
```



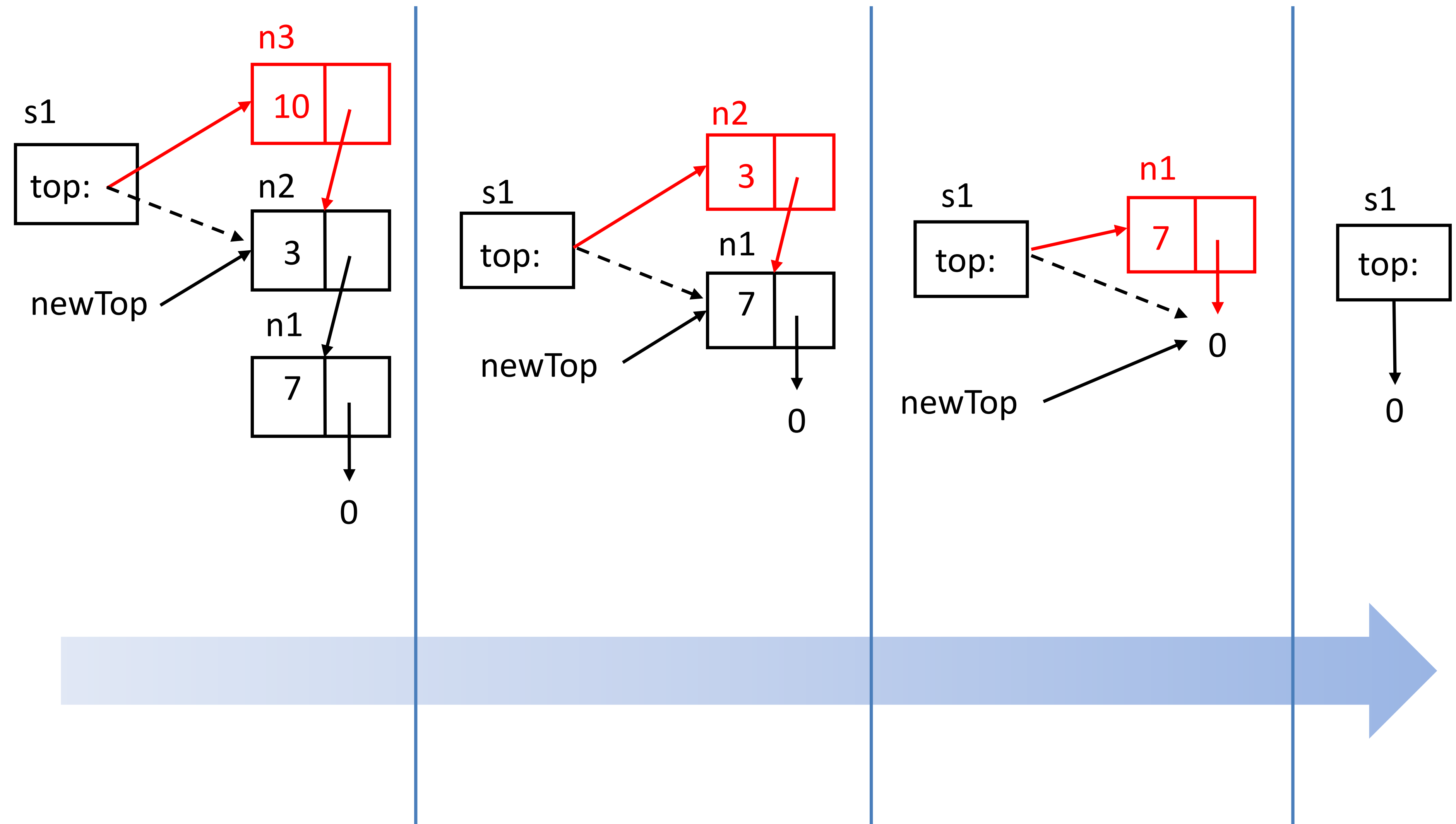
Stack: Η Μέθοδος pop

```
template <typename T>
T Stack<T>::pop()
{
    if(isEmpty()) { return T(); }
    const T value = top->getValue();
    const StackNode<T>* const newTop = top->getNext();
    delete top;
    top = newTop;
    return value;
}
```

default “initializer” (constructor για κλάσεις, default τιμές για βασικούς τύπους)



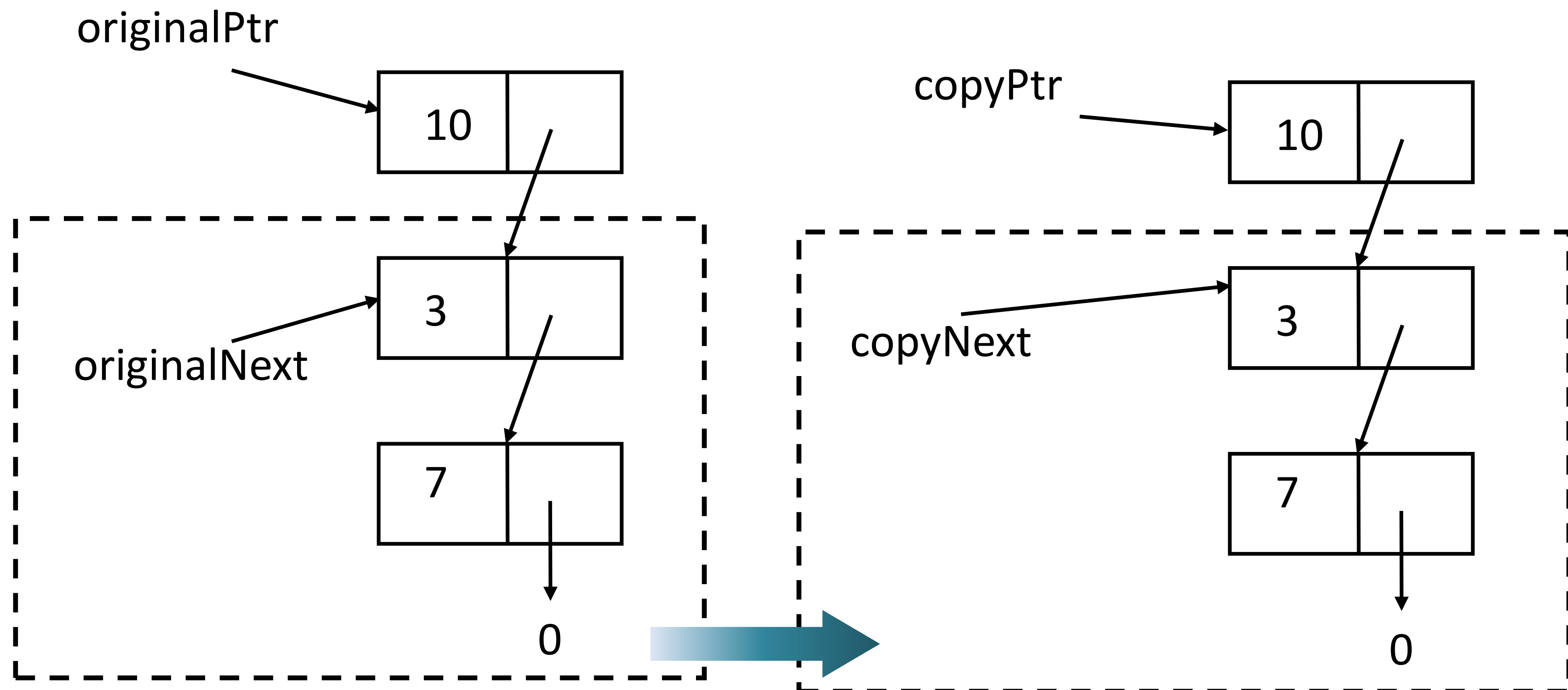
Stack: η Μέθοδος deleteStackNodes (1)



Stack: η Μέθοδος deleteStackNodes (2)

```
template <typename T>
void Stack<T>::deleteStackNodes()
{
   const StackNode<T>* newTop;
   while(top != nullptr)
   {
      newTop = top->getNext();
      delete top;
      top = newTop;
   }
}
```

Stack: η Μέθοδος copyStackNodes (1)



Αναδρομική αντιγραφή μικρότερου μέρους της αλυσίδας.

Stack: η Μέθοδος copyStackNodes (2)

```
template <typename T>
const StackNode<T>* const Stack<T>::copyStackNodes(const StackNode<T>* const
originalPtr)
{
   if(originalPtr == 0)
      return nullptr;

   const StackNode<T> const * originalNext = originalPtr->getNext();
   const StackNode<T> const * copyNext = copyStackNodes(originalNext);
   const StackNode<T> const * copyPtr =
      new StackNode(originalPtr->getValue(), copyNext);
   return copyPtr;
}
```

Stack: η Μέθοδος copyStackNodes (3)

- Συντομότερη εκδοχή:

```
template <typename T>
const StackNode<T>* const Stack<T>::copyStackNodes(const StackNode<T>* const
originalPtr)
{
   if(originalPtr == 0)
      return nullptr;
   return new StackNode( originalPtr->getValue(),
        copyStackNodes(originalPtr->getNext()) );
}
```

Stack: Copy Constructor & operator=()

```
template <typename T>
Stack<T>::Stack(const Stack<T>& original) : top(copyStackNodes(original.top)) {}


template <typename T>
Stack<T>& Stack<T>::operator=(const Stack<T>& right)
{
   if ( this == &right)
      return *this;

   deleteStackNodes();
   top = copyStackNodes(right.top);
   return *this;
}
```

Stack: Χρήση

```
int main() {
   Stack<string>  s3;
   s3.push("seven"); s3.push("three"); s3.push("ten");
   Stack<string> s4(s3);
   s4.push("twenty");
   cout << s3.pop() << endl;
   s3 = s4;
   cout << s3.pop() << endl;
   cout << s4.pop() << endl;
}
```

Iterators σε Δικές μας Συλλογές

- Τόσο για λόγους συμβατότητας, όσο και για λόγους ευχρηστίας, είναι καλό να ορίζουμε iterators στις συλλογές που κατασκευάζουμε
- Να τονίσουμε ότι αυτό είναι ιδιαίτερα χρήσιμο σε μη γραμμικές δομές (π.χ. γράφους, δένδρα) όπου δεν είναι προφανής στο χρήστη ο μηχανισμός διάσχισης της συλλογής μας

Iterators – Το Παράδειγμα της Stack

```
template <typename T> class StackIterator;   // forward class declaration
                                             // Τη χρειάζεται η Stack
template <typename T = int>
class Stack
{
   ...
public:
   typedef StackIterator<T> iterator;        // Ορίζουμε κι ένα ψευδώνυμο για τον StackIterator
                                             // ως Stack::iterator, για συμβατότητα με την STL
   StackIterator<T> begin() const;           // Μας επιστρέφει την αρχή και το τέλος της
   StackIterator<T> end() const;             // συλλογής μας.
   ...
};                                           // Προφανώς, αφού ένας iterator επιστρέφει την
                                             // τρέχουσα τιμή της διάσχισης της συλλογής τύπου
                                             // T, είναι κι αυτός template
```

Η Υπογραφή ενός Iterator – Ο StackIterator

```
template <typename T>
class StackIterator : public std::iterator<std::forward_iterator_tag, T>
{
    const StackNode<T> * node_ptr;                         // δείκτης στην τρέχουσα θέση
    const T last_value;                                    // στη Stack

public:
    StackIterator()                                        // default constructor
        : node_ptr(nullptr), last_value() {}
    StackIterator(const StackIterator<T> & src )           // copy constructor
        : node_ptr(src.node_ptr), last_value() {}

    StackIterator<T> & operator++();                       // ++iter
    StackIterator<T> operator++(int);                      // iter++
    bool operator==(const StackIterator<T> & rhs);
    bool operator!=(const StackIterator<T> & rhs);
    const T operator*() const;                             // επιστροφή περιεχομένου

protected:
    StackIterator(const StackNode<T> * const p_node)  // χρειαζόμαστε πρόσβαση στον
        : node_ptr(p_node), last_value() {}            // ειδικό αυτό κατασκευαστή
                                                       // μόνο από τη Stack (begin,
    friend class Stack<T>;                             // end) → protected και
};                                                     // κάνουμε friend τη Stack
```

Υλοποίηση του StackIterator: ++

```
template <typename T>
inline StackIterator<T> & StackIterator<T>::operator++()
{
    if (node_ptr==nullptr)             // Αν ξεπέρασες το τελευταίο στοιχείο, επέστρεψε
     return *this;
    node_ptr = node_ptr->getNext();  // Αλλιώς, προχώρα στο επόμενο
    return *this;
}

template <typename T>
StackIterator<T> StackIterator<T>::operator++(int)
{
    StackIterator tmp(*this); // Ο μεταθεματικός τελεστής πρέπει να επιστρέψει ένα αντίγραφο
    operator++();             // του αντικειμένου πριν συμβεί η μετάβαση στο επόμενο στοιχείο
    return tmp;               // και για αυτό έχει πάντα το επιπρόσθετο κόστος της
}                             // δημιουργίας αντιγράφου (με copy constructor)

                              // Να το θυμόμαστε αυτό και να προτιμάμε τον προθεματικό
                              // τελεστή μετακίνησης του iterator.
```

Υλοποίηση του StackIterator: == , !=, *

```
template <typename T>
bool StackIterator<T>::operator==(const StackIterator<T>& rhs)
{   return node_ptr==rhs.node_ptr;
}

template <typename T>
bool StackIterator<T>::operator!=(const StackIterator& rhs)
{   return !this->operator==(rhs);
}

template <typename T>
const T StackIterator<T>::operator*() const
{
    if (node_ptr!=nullptr)
     return node_ptr->getValue();     // Για να μπορούμε να επιστρέφουμε κάτι πάντα, ακόμα
    else                             // και σε κενή Stack ή όταν βρίσκεται στο τέρμα ο
     return last_value;              // έχουμε ορίσει μια const μεταβλητή του ίδιου τύπου
}                                    // με το template
```

Υλοποίηση των begin και end της Stack

```
template <typename T>
inline StackIterator<T> Stack<T>::begin() const
{
    return StackIterator<T>(top);
}

template <typename T>
inline StackIterator<T> Stack<T>::end() const
{
    return StackIterator<T>(nullptr);
}
```

Παράδειγμα Χρήσης του Stack::iterator

```
int main()
{
   Stack<string> mystack;

    mystack.push("10");
    mystack.push("20");
    mystack.push("30");

    cout << endl;
    for (Stack<string>::iterator iter=mystack.begin(); iter!=mystack.end(); ++iter)
       cout << *iter << " ";

    cout << endl;
    return 0;
}                         // τυπώνει 30 20 10
```