Προγραμματισμός Υπολογιστών με C++

ΣΧΕΔΙΟΤΥΠΑ (TEMPLATES)

Γεώργιος Παπαϊωάννου (2013-20)
gepap@aueb.gr

Περιεχόμενο Παρουσίασης

- Περιγραφή:
  - Τι είναι τα σχεδιότυπα και που μας εξυπηρετούν
  - Template συναρτήσεις
  - Template κλάσεις
  - Εξειδίκευση σχεδιοτύπων
  - Σχεδιότυπα και αρχεία .h kai .cpp/c++
  - Παραδείγματα χρήσης σχεδιοτύπων

- Τελευταία ενημέρωση: Νοέμβριος 2020

Σχεδιότυπα (Templates) Συναρτήσεων

```
void myswap(int& a, int& b)
{
   const int temp = a;
   a = b;
   b = temp;
}

void myswap(float& a, float& b)
{
   const float temp = a;
   a = b;
   b = temp;
}

void myswap(string& a, string& b)
{
   const string temp = a;
   a = b;
   b = temp;
}
...
```

- Εναλλακτικά: Αντί να ορίσουμε ξεχωριστές μορφές της συνάρτησης, καθορίζουμε πώς μπορεί να παραχθεί μια μορφή της για οποιονδήποτε τύπο T:

```
template <typename T>
void myswap(T& a, T& b)
{
   const T temp = a;
   a = b;
   b = temp;
}
```

Χρήση Σχεδιοτύπου – Παράδειγμα 1

```
template <typename T>
void myswap(T& a, T& b) {  // Το σχεδιότυπο απαιτεί και τα δύο
   const T temp = a;       // ορίσματα να είναι του ιδίου τύπου T.
   a = b;
   b = temp; }

int main() {
   int i = 10, j = 20;
   myswap(i, j);           // Παράγεται αυτόματα η κατάλληλη συνάρτηση.
   cout << i << “” << j;   // Τυπώνει: «20, 10».
   string s1= "hi", s2 = “ there";
   myswap(s1, s2);         // Παράγεται άλλη μορφή της συνάρτησης.
   cout << s1 << s2 <<;    // Τυπώνει: «there», «hi».
   myswap(i, s1);          // Λάθος: ορίσματα διαφορετικών τύπων.
}                          // Δεν έχουμε σχεδιότυπο για αυτή την
                           // περίπτωση (διαφορετικοί τύποι).
```

Χρήση Σχεδιοτύπου – Παράδειγμα 2

```
template <typename T1, typename T2>
void printFirstPairs( const unsigned int n, const vector<T1>& v1,
                      const vector<T2>& v2)
{
   for( unsigned i = 0; i < v1.size() && i < v2.size() && i < n; i++)
   {
         cout << i << ": " << v1[i] << ", " << v2[i] << endl;
   }
}

int main() {
   vector<string> vec1;
   vector<unsigned int> vec2;
   vec1.push_back("one"); vec1.push_back("two"); vec1.push_back("three");
   vec2.push_back(10); vec2.push_back(20); vec2.push_back(30);
   printFirstPairs(2, vec1, vec2);       // Τυπώνει: 0: one, 10 | 1: two, 20
}
```

Παρατήρηση: Ο τύπος vector<string> είναι διαφορετικός από τον τύπο vector<unsigned int>. Δεν έχει σημασία ότι είναι και οι δύο vector<...>

Σχεδιότυπα Κλάσεων (1)

Δήλωση:

```
template <typename T1, typename T2>
class Pair {
   T1 first; T2 second;
public:
   Pair(const T1& firstIn, const T2& secondIn) :
      first(firstIn), second(secondIn) {}
   T1& getFirst() { return first; }
   T2& getSecond() { return second; }
};
```

Σχεδιότυπα Κλάσεων (2)

Χρήση:

```
int main()
{
   Pair<string, unsigned> p1("word1", 100), p2("word2", 200);
   vector< Pair<string, unsigned> > vec;
   vec.push_back(p1); vec.push_back(p2);
   vec[1].getFirst() = "word3";
   cout << vec[0].getFirst() << ", " << vec[0].getSecond() << endl;
   cout << vec[1].getFirst() << ", " << vec[1].getSecond() << endl;
}
```

Δήλωση Σχεδιοτύπων στον Κώδικα

- Όλα τα σχεδιότυπα δηλώνονται σε header files και όχι σε αρχεία υλοποίησης (.cpp, .c++), ακόμα και οι υλοποιήσεις των συναρτήσεων και κλάσεων
- Διαφορετικά θα έχει πρόβλημα ο compiler
- Δηλαδή στην περίπτωση των σχεδιότυπων κλάσεων, το αρχείο κεφαλίδας περιέχει και τους ορισμούς των μεθόδων (τον κώδικά τους)
-

Εξειδίκευση Σχεδιοτύπων

- Πολλές φορές, παρότι δηλώνουμε γενικά σχεδιότυπα για κλάσεις ή συναρτήσεις, μας ενδιαφέρει να ορίσουμε διαφορετική συμπεριφορά του κώδικα για ορισμένους template τύπους
- Σε αυτή την περίπτωση, μπορούμε να επαναλάβουμε την υλοποίηση των συγκεκριμένων μεθόδων και συναρτήσεων, δίνοντας συγκεκριμένα ονόματα τύπων όπου χρειάζεται
- Για να μη δημιουργηθεί σύγκρουση, πάντα προσδιορίζουμε το πρόθεμα “inline” στις υλοποιήσεις αυτές (γιατί;)

Εξειδίκευση Σχεδιοτύπων - Παράδειγμα (1)

```
template <typename T> class vec2_t
{
public:
    T x, y;
    double length () const;

    T dot ( vec2_t<T> v ) const;
    static T dot (vec2_t<T> v1, vec2_t<T> v2);
    const vec2_t<T> operator+(const vec2_t<T> & right) const;
    const vec2_t<T> operator-(const vec2_t<T> & right) const;
    ...
}
// component-wise "equal" operator (global, not member):
template<typename T> vec2_t<bool> operator==( const vec2_t<T> & left,
                                              const vec2_t<T> & right);
```

Εξειδίκευση Σχεδιοτύπων - Παράδειγμα (2)

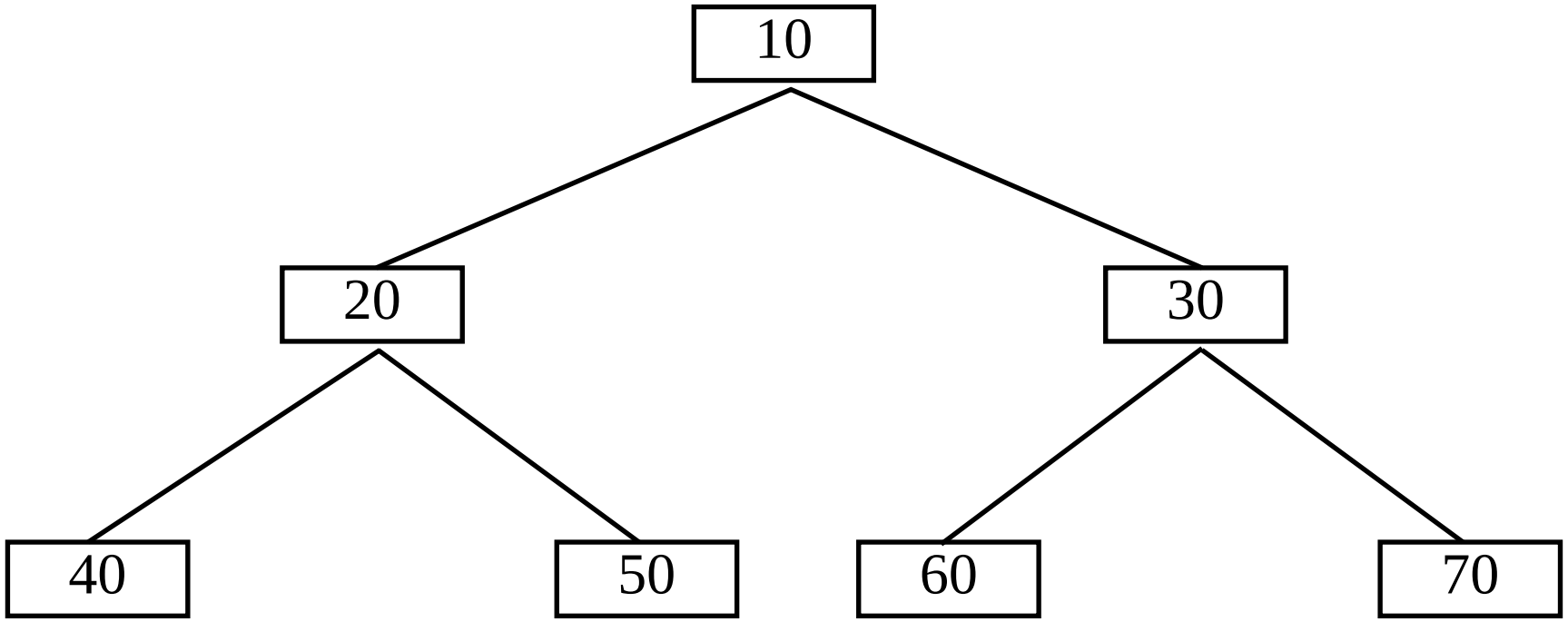
```
// Γενικευμένη υλοποίηση του τελεστή:
template<typename T> vec2_t<bool> operator==( const vec2_t<T> & left,
                                              const vec2_t<T> & right)
{
    vec2_t<bool> res;
    res.x = left.x==right.x;
    res.y = left.y==right.y;
    return res;
}
```

- Όμως ο παραπάνω κώδικας θα δουλεύει καλά μόνο για ακεραίους, αφού στους αριθμούς κινητής υποδιαστολής χάνουμε ακρίβεια αναπαράστασης →
- Παράγουμε εξειδικευμένες υλοποιήσεις γι αυτούς

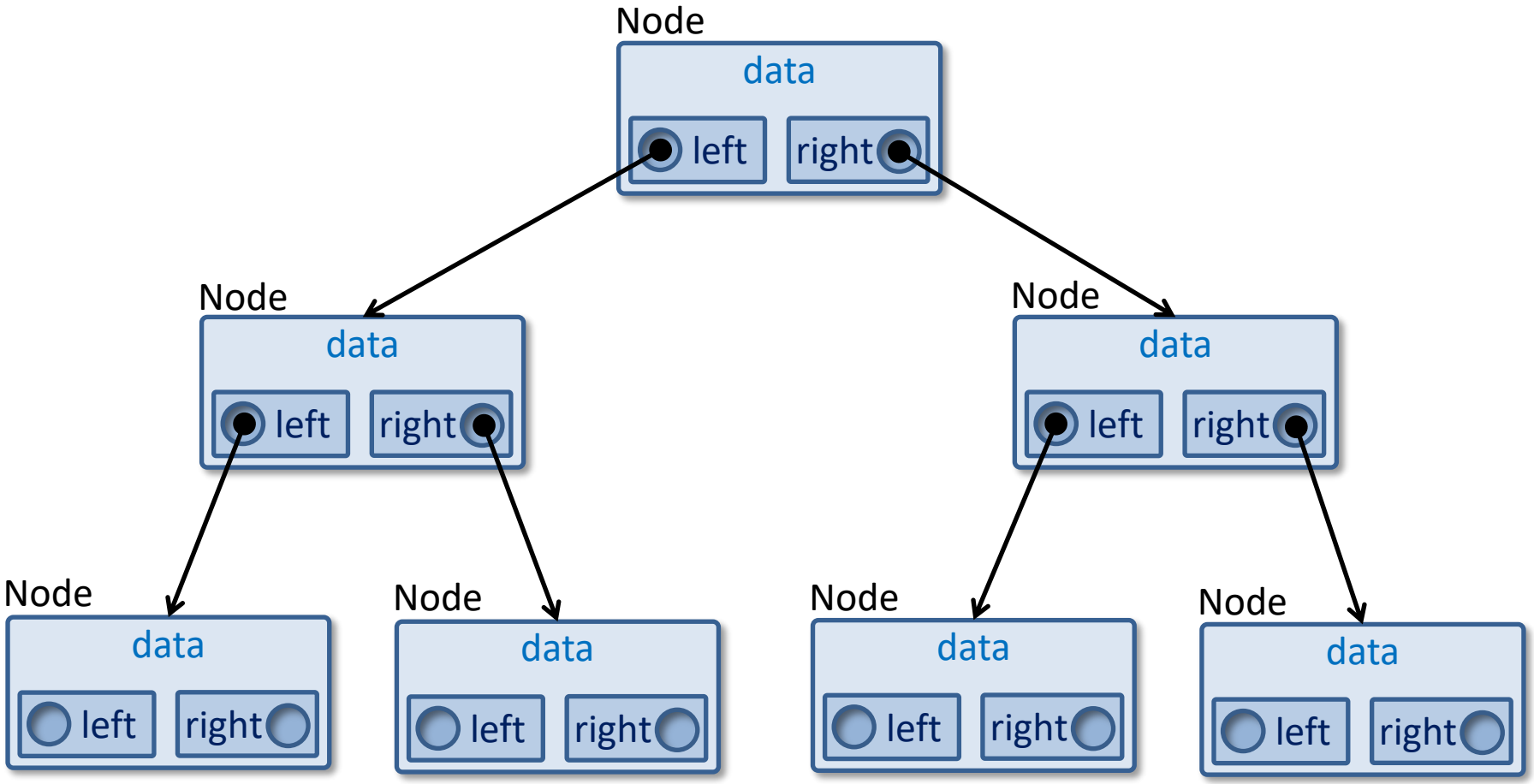
Εξειδίκευση Σχεδιοτύπων - Παράδειγμα (3)

```
#define COORD_EPSILON 10e-6

// specialized template implementation:
inline vec2_t<bool> operator==( const vec2_t<float> & left,
                                const vec2_t<float> & right) {
    vec2_t<bool> res;
    res.x = fabs(left.x-right.x) < COORD_EPSILON;
    res.y = fabs(left.y-right.y) < COORD_EPSILON;
    return res;
}

inline vec2_t<bool> operator==( const vec2_t<double> & left,
                                const vec2_t<double> & right) {
    vec2_t<bool> res;
    res.x = abs(left.x-right.x) < COORD_EPSILON;
    res.y = abs(left.y-right.y) < COORD_EPSILON;
    return res;
}
```

Παράδειγμα Υλοποίησης Δυαδικού Δένδρου



Παράσταση Δυαδικού Δένδρου



Η Κλάση Node

```
class Node {
   const int data;
   Node const * const left;        // Διαβάζεται ανάποδα:
   Node const * const right;       // "a const pointer to a const Node"
                                   // Θα μπορούσαν και να μην είναι
                                   // const εδώ
public:
   Node( const int data, const Node* const left = 0,
         const Node* const right = 0) :
         data(data), left(left), right(right) { }
   void printTree() const;
};
```

0 == NULL (C) == nullptr (C++11)

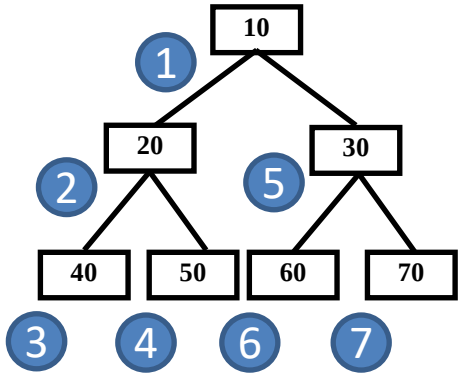
Υπενθύμιση: μπορούμε να έχουμε ίδια ονόματα ορισμάτων και πεδίων. Εδώ υπονοείται: this->data(data) κλπ

Παράδειγμα Χρήσης της Node

```
int main() {
   Node n4(40), n5(50), n6(60), n7(70),   // φύλλα
            n2(20, &n4, &n5),             // ενδιάμεσοι κόμβοι
            n3(30, &n6, &n7),             //
            n1(10, &n2, &n3);             // ρίζα
   n1.printTree();
}
```

Αναδρομική Εκτύπωση

```
void Node::printTree() const
{
    cout << data << " ";
    if(left != nullptr)
    {
        left->printTree();
    }
    if(right != nullptr)
    {
        right->printTree();
    }
    // Τυπώνει με αναζήτηση πρώτα σε βάθος (DFS).
    // Στο δέντρο του παραδείγματος τυπώνει:
    // 10 20 40 50 30 60 70
}
```

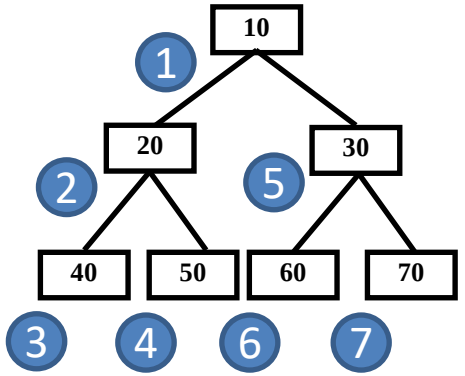


Μετατροπή της printTree σε Μη Αναδρομική

- Με τη χρήση αναδρομής, συνεχώς μεγαλώνουμε τη stack της εφαρμογής όσο κάνουμε αναδρομικές κλήσεις με αποτέλεσμα:
  - Να βάζουμε στον κώδικά μας το overhead της κλήσης συνάρτησης και επιστροφής από αυτή σε κάθε βήμα
  - Να κινδυνεύουμε να προκαλέσουμε stack overflow για δένδρα μεγάλου βάθους
- Μπορούμε εύκολα να μετατρέψουμε τον αναδρομικό μας κώδικα σε επαναληπτικό με τη χρήση μιας **στοίβας** (LIFO γραμμική δομή: Last In First Out)

Μη Αναδρομική printTree με Στοίβα (DFS)

```
#include <stack>

void Node::printTree() const
{
    stack<const Node*> remaining;
    remaining.push(this);

    while (!remaining.empty())
    {
        const Node * current = remaining.top();
        remaining.pop();
        cout << current->data << " ";
        if (current->right)
            remaining.push(current->right);
        if (current->left)
            remaining.push(current->left);
    }
}
```

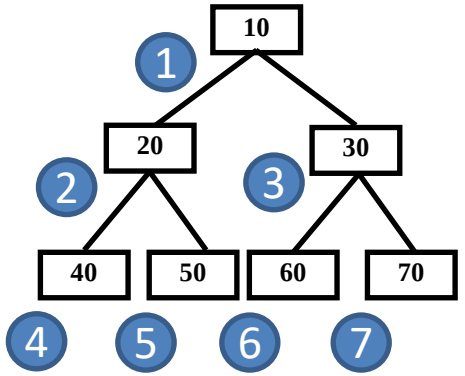


printTree με Διάσχιση κατά Πλάτος (BFS)

- Τώρα, όταν επισκεπτόμαστε έναν κόμβο, στη συνέχεια επιχειρούμε να τυπώσουμε πρώτο το αριστερό παιδί του, μετά το δεξί και στο τέλος τον επόμενο κόμβο ίδιου επιπέδου (Depth-First Search)
- Αν θέλουμε να δίνουμε προτεραιότητα στους κόμβους ίδιου επιπέδου (siblings), τότε πρέπει να κάνουμε διάσχιση κατά πλάτος των επιπέδων του δένδρου (Breadth-First Search)
- Για τη BFS θα χρειαστούμε μια FIFO γραμμική δομή αποθήκευσης (Firtst In First Out), δηλαδή μια **ουρά**

Μη Αναδρομική printTree με Ουρά (BFS)

```
#include <queue>

void Node::printTree() const
{
    queue<const Node*> remaining;
    remaining.push(this);

    while (!remaining.empty())
    {
        const Node * current = remaining.front();
        remaining.pop();
        cout << current->data << " ";
        if (current->left)
            remaining.push(current->left);
        if (current->right)
            remaining.push(current->right);
    }
}
```



Μετατροπή της Κλάσης Node σε Σχεδιότυπο (1)

```
template <typename T>
class Node {
   const T data;
   Node const * const left;
   Node const * const right;

public:
   Node( const T data, const Node* const left = 0,
         const Node* const right = 0) :
         data(data), left(left), right(right) { }
   void printTree() const;
};
```

Μετατροπή της Κλάσης Node σε Σχεδιότυπο (2)

```
template <typename T>
void Node<T>::printTree() const {
    queue<const Node*> remaining;
    remaining.push(this);
    while (!remaining.empty())
    {
        const Node * current = remaining.front();
        remaining.pop();
        cout << current->data << ", " << endl;
        if (current->left)
            remaining.push(current->left);
        if (current->right)
            remaining.push(current->right);
    }
}
```

Διαφορές με τη Java

Σε γενικές γραμμές, τα templates της C++ αντιστοιχούν (αλλά προϋπάρχουν) των generics της Java	Τα generics υλοποιούν περίπου τη λογική των templates της C++
Η C++ επιτρέπει την εξειδίκευση μεθόδων και συναρτήσεων, χωρίς να απαιτείται η εξειδίκευση ή η παραγωγή νέας κλάσης	Δεν επιτρέπεται η εξειδίκευση μεθόδων
Δεν υπάρχει στενός προσδιορισμός τύπων σε template	Υπάρχει η έννοια του στενότερου προσδιορισμού του generic με wildcards (<? extends Type>)