

Τεχνολογίες Υλοποίησης Αλγορίθμων

Υλοποίηση Παραμετρικών Τύπων

Χρήστος Δ. Ζαρολιάγκης

Καθηγητής

email: `zaro@ceid.upatras.gr`

Τμήμα Μηχ/κών Υπολογιστών & Πληροφορικής

Πανεπιστήμιο Πατρών

Υλοποίηση Παραμετρικών Τύπων

- Η μέθοδος των αρχετύπων (template approach).
- Η προσέγγιση της LEDA.
- *Παράδειγμα Μελέτης:*
η υλοποίηση ενός τύπου δεδομένων `list`.
 - ▷ Δηλώσεις (προδιαγραφές) της κλάσης → `.h` αρχείο
 - ▷ Υλοποίηση των μεθόδων της κλάσης → `.C` αρχείο

Η μέθοδος των αρχετύπων - οι προδιαγραφές της κλάσης

```
// file t_list.h

template <class E>
class t_list
{
public:
    void push(const E&);
        // inserts in the front of the list

    void pop(E&);
        // removes first element and returns it

    const E& head() const;
        // returns first element

    int size() const;
        // returns the number of elements in the list

    t_list();
    t_list(const t_list<E>&);
    ~t_list();
```

```

// file t_list.h - 2

private:
    struct list_elem
    { // local structure for representing
      // the elements of the list

      E entry;
      list_elem * succ;
      list_elem(const E& x, list_elem* s) :
          entry(x), succ(s) {}
      friend class t_list;
    };

    list_elem * hd; // head of list
    int        sz; // size of list
};

```

Η μέθοδος των αρχετύπων - υλοποίηση των μεθόδων

```
// file t_list.C - 1

#include "t_list.h"

template <class E> t_list<E>::t_list()
{ hd = 0; sz = 0; }

int t_list<E>::size() const
{ return sz; }

template <class E> const E& t_list<E>::head() const
{ return hd->entry; }

template <class E> void t_list<E>::push(const E& x)
{
    hd = new list_elem(x, hd);
    sz++;
}

template <class E> void t_list<E>::pop(E& y)
{
    y = hd->entry;
    list_elem * p = hd;
    hd = p->succ;
    delete p;
    sz--;
}
```

```


// file t_list.C - 2

template <class E> t_list<E>::t_list(const t_list<E>& L)
{
    // construct a copy of L
    hd = NULL;
    sz = L.sz;
    if (sz > 0)
    {
        hd = new list_elem(L.hd->entry,0); // first element
        list_elem * q = hd;
        // subsequent elements
        for (list_elem * p = L.hd->succ;
            p != NULL;
            p = p->succ)
        {
            q->succ = new list_elem(p->entry, NULL);
            q = q->succ;
        }
    }
}

template <class E> t_list<E>::~~t_list()
{ // destroy the list
    while (hd)
    {
        list_elem * p = hd->succ;
        delete hd;
        hd = p;
    }
}

```

Χαρακτηριστικά αυτής της μεθόδου

- Δυναμική και κομψή για υλοποιητές και χρήστες.
- Πολλαπλασιάζει κώδικα κατά τη μεταγλώπηση (ανάλογα με το πόσες φορές χρησιμοποιείται) \Rightarrow αύξηση του χρόνου μεταγλώπησης
(η απεικόνιση της δομής δεδομένων στη μνήμη εξαρτάται από το μέγεθος των αντικειμένων του παραμετρικού τύπου).
- Ξεχωριστή μεταγλώπηση δεν είναι δυνατή: και τα δύο αρχεία `t_list.h` και `t_list.C` πρέπει να συμπεριληφθούν σε μια εφαρμογή και να μεταγλωπριστούν μαζί με την εφαρμογή.

 - ▷ Μεγάλος κώδικας και μεγάλοι χρόνοι μεταγλώπησης σε εφαρμογές που χρησιμοποιούν πολλούς παραμετρικούς τύπους από μια βιβλιοθήκη.
 - ▷ Εξαναγκάζει τον υλοποιητή της βιβλιοθήκης να παρέχει δημόσια τις λεπτομέρειες της υλοποίησης (.C αρχεία).
- Η προ-μεταγλώπηση κώδικα που χρησιμοποιεί αρχέτυπα είναι προβληματική σε όλους τους μεταγλωπριστές της C++.

Η προσέγγιση της LEDA

- Κάθε παραμετρικός τύπος υλοποιείται από δύο κλάσεις:
 - *Κλάση Τύπου Δεδομένων*: κλάση για τον ΑΤΔ (π.χ. κλάση λεξικού).
 - *Κλάση Δομής Δεδομένων*: κλάση για τη δομή δεδομένων (π.χ. κλάση δυαδικού δένδρου) που χρησιμοποιείται για την υλοποίηση της κλάσης τύπου δεδομένων.
- Μόνο οι κλάσεις τύπων δεδομένων χρησιμοποιούν τον μηχανισμό αρχετύπων. Τα αρχεία κεφαλίδων τους βρίσκονται στον κατάλογο `LEDAROOT/inc1/LEDA`.
- Οι κλάσεις δομών δεδομένων είναι προμεταγλωπτισμένες στις βιβλιοθήκες με κώδικα αντικειμένου (object code) και συνδέονται σε προγράμματα εφαρμογών μέσω του διασυνδετή (linker) της C++.

- **Προμεταγλώττιση** μιας δομής δεδομένων είναι δυνατή μόνο αν η υλοποίησή της δεν εξαρτάται από τους πραγματικούς τύπους των παραμέτρων που ανήκουν στον παραμετρικό τύπο που υλοποιεί.

- Η **απεικόνιση** της δομής δεδομένων στη μνήμη **δεν** πρέπει να εξαρτάται από το μέγεθος των αντικειμένων που αποθηκεύονται σε αυτήν.

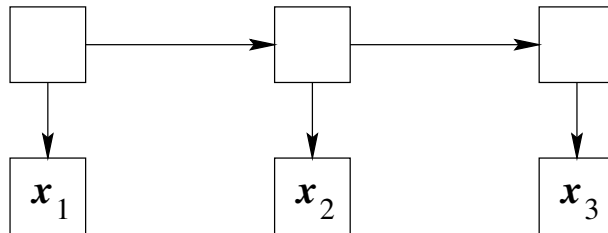
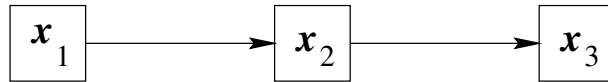


Στην LEDA αυτό επιτυγχάνεται με την αποθήκευση δεικτών σε αντικείμενα (εκτός από τύπους αντικειμένων των οποίων το μέγεθος είναι μικρό, δηλ. δεν ξεπερνάει το μέγεθος ενός δείκτη).

- Όλες οι συναρτήσεις που χρησιμοποιούνται στην υλοποίηση και των οποίων η σημασιολογία εξαρτάται από τους πραγματικούς τύπους των παραμέτρων χρησιμοποιούν τον μηχανισμό δυναμικής δέσμευσης (dynamic binding) της C++, δηλ. υλοποιούνται σαν εικονικές συναρτήσεις.

Βασική Ιδέα

- Τα πεδία δεδομένων στους περιέχοντες τύπους όλων των δομών δεδομένων είναι τύπου `void *`, δηλ. του γενικευμένου τύπου δείκτη της C++. Περιέχουν δείκτες σε αντικείμενα πραγματικών τύπων παραμέτρων.
- Τα αντικείμενα ενός τύπου `T` δεν αποθηκεύονται άμεσα στη δομή δεδομένων, αλλά στη δυναμική μνήμη (σωρός).



Βασική Ιδέα (συνέχεια)

- Μια κλάση τύπου δεδομένων χρησιμοποιεί τον μηχανισμό αρχετύπων και παράγεται από την αντίστοιχη κλάση δομής δεδομένων (κλάση υλοποίησης).
- Χρησιμοποιείται μετατροπή τύπων (type casting) για την «γεφύρωση» μεταξύ του «άτυπου κόσμου» της κλάσης δομής δεδομένων (όλα τα δεδομένα είναι τύπου `void *`) και εκείνου της κλάσης του τύπου δεδομένων.

Παράδειγμα

Υλοποίηση του παραμετρικού τύπου `list<T>` από μια συγκεκριμένη δομή δεδομένων `list_impl` η οποία αποθηκεύει δείκτες τύπου `void *`.

```
// file list_impl.h -- essentially t_list<void *>
class list_impl
{
protected:
    list_impl();
    ~list_impl();

    void * head() const;
    void * pop();
    void  push(void* x);
    void  clear();
    int   size() const;

private:
    struct list_impl_elem
    {
        void * entry;
        list_impl_elem * succ;
        list_impl_elem(void * x, list_impl_elem * s):
            entry(x), succ(s) {}

        friend class list_impl;
    };

    list_impl_elem * hd;
    int             sz;
};
```

Υλοποίηση μεθόδων κλάσης `list_impl`

```
// file list_impl.C

#include "list_impl.h"

list_impl::list_impl() : hd(0), sz(0) {}
list_impl::~list_impl() { clear(); }
void * list_impl::head() const { return hd->entry; }

void list_impl::push(void * x)
{
    hd = new list_impl_elem(x,hd);
    sz++;
}

void * list_impl::pop()
{
    void * x = hd->entry;
    list_impl_elem * p = hd;
    hd = p->succ;
    delete p;
    sz--;
    return x;
}

void list_impl::clear() { while (hd) pop(); }
int list_impl::size() const { return sz; }
```

Πώς παράγεται τώρα ο τύπος `list<T>`;

- Η `list_impl` γίνεται μια *ιδιωτική* κλάση-βάσης της κλάσης `list<T>` και οι συναρτήσεις-μέλη της `list<T>` υλοποιούνται με χρήση των συναρτήσεων-μελών της κλάσης βάσης.
- Μια συνάρτηση-μέλος της `list<T>` με ένα όρισμα τύπου `T`:
 1. Αντιγράφει το όρισμα στη δυναμική μνήμη.
 2. Μετατρέπει έναν δείκτη στο αντίγραφο σε τύπο `void *`.
 3. Μεταβιβάζει τον δείκτη στην αντίστοιχη συνάρτηση της κλάσης βάσης.
- Όλες οι συναρτήσεις-μέλη της `list<T>` που επιστρέφουν ένα αποτέλεσμα τύπου `T`:
 1. Καλούν την αντίστοιχη συνάρτηση της κλάσης βάσης (η οποία επιστρέφει ένα αποτέλεσμα τύπου `void *`).
 2. Μετατρέπουν τον δείκτη σε τύπο `T *`.
 3. Επιστρέφουν τον αποαναφοροποιημένο δείκτη.

- ```
template<class T>
class list : private list_impl
{
 public:
 list() : list_impl() {}
```

Η συνάρτηση κατασκευής της `list<T>` κατασκευάζει μια κενή `list_impl`.

- ```
void push(const T& x)
{ list_impl::push((void *) new T(x)); }
```

Η εντολή `L.push(x)` δημιουργεί ένα αντίγραφο του `x` στη δυναμική μνήμη (καλώντας τη συνάρτηση κατασκευής αντιγράφου του `T` μέσω του τελεστή `new`) και μεταβιβάζει έναν δείκτη σε αυτό το αντίγραφο (αφού τον μετατρέψει σε τύπο `void *`) στη συνάρτηση `list_impl::push()`.

Επειδή η μετατροπή από `T *` σε `void *` είναι ενσωματωμένη στην `C++`, το «`(void *)`» μπορεί να παραληφθεί:

```
void push(const T& x)
{ list_impl::push(new T(x)); }
```

- ```
const T& head() const
{ return *(T*)list_impl::head(); }
```

Η εντολή `L.head()` μετατρέπει το αποτέλεσμα τύπου `void *` της `list_impl::head()` σε έναν δείκτη τύπου `T *`, αποαναφοροποιεί το αποτέλεσμα και επιστρέφει το αντικείμενο σαν μια `const`-αναφορά.

- ```
void pop(T& x)
{
    T * p = (T*)list_impl::pop();
    x = *p;
    delete p;
}
```

Η εντολή `L.pop(x)` μετατρέπει το, τύπου `void *`, αποτέλεσμα της `list_impl::pop()` σε έναν δείκτη τύπου `T *`, το αποαναφοροποιεί αντιγράφοντάς το στη μεταβλητή `x` και διαγράφει το αντίγραφο του που υπάρχει στη λίστα.

- `int size() const { return list_impl::size(); }`

```
void clear()
{
    while (size() > 0)
        delete (T*)list_impl::pop();
        // cast absolutely necessary !

    list_impl::clear(); // not necessary
}
```

Η συνάρτηση `clear()` πρώτα διαγράφει όλα τα στοιχεία της λίστας (συμπεριλαμβανομένων των συνολοστοιχείων που υπάρχουν στη δυναμική μνήμη) και μετά καλεί τη συνάρτηση `list_impl::clear()`.

- `~list()`
`{`
 `clear();`
`}`

Η συνάρτηση κατάργησης καλεί την `clear()` για να διαγράψει τα αντικείμενα που υπάρχουν στη δυναμική μνήμη (τα συνολοστοιχεία της λίστας). Η `~list_impl()` καλείται αυτόματα για να διαγράψει τη λίστα.

Αξιολόγηση της υλοποίησης

- Σαφώς πιο πολύπλοκη από την μέθοδο αρχετύπων.
- Ο τύπος δεδομένων `list<T>` προσομοιώνει τον τύπο δεδομένων `t_list<T>`.
- Η υλοποίηση των μεθόδων του `list<T>` είναι σημαντικά απλούστερη.
- Μπορεί να γίνει προμεταγλώπιση της κλάσης υλοποίησης.
- Υπάρχει ασφάλεια τύπων (type safety).
- Είναι μία ολοκληρωμένη λύση.
- Γενίκευση:
 - (α) ορισμός της `list_impl ≡ t_list<void *>`.
 - (β) ορισμός της `list<T>` ίσως όχι τόσο κομψός λόγω των διαφόρων μετατροπών τύπων. Αλλά, οι μετατροπές τύπων ακολουθούν έναν απλό κανόνα: εισερχόμενες τιμές μετατρέπονται σε `void *` και εξερχόμενες πάλι πίσω σε `T`.

Τι κάνουμε όταν η κλάση υλοποίησης χρειάζεται να έχει γνώση του πραγματικού τύπου των παραμέτρων της κλάσης τύπου δεδομένων;

- Π.χ.-1:

θέλουμε να συμπεριλάβουμε μια μέθοδο `print()` στον τύπο λίστας που δημιουργήσαμε και θέλουμε να την υλοποιήσουμε σαν συνάρτηση-μέλος της κλάσης `list_impl`.

- Π.χ.-2:

Κάθε υλοποίηση ενός παραμετρικού τύπου `dictionary<K, I>` η οποία βασίζεται σε συγκρίσεις κλειδιών (π.χ. δυαδικά δένδρα διερεύνησης) πρέπει να γνωρίζει πως να κάνει συγκρίσεις.

Στην LEDA μια γραμμική διάταξη σε έναν τύπο κλειδιού `K` ορίζεται από μια καθολική συνάρτηση

```
int compare(const K&, const K&)
```

και επομένως η κλάση υλοποίησης πρέπει να μπορεί να καλεί αυτή τη συνάρτηση.

Λύση : Εικονικές Συναρτήσεις και Δυναμική Δέσμευση

- Π.χ.-1: Η κλάση `list_impl` χρησιμοποιεί μια γνήσια εικονική συνάρτηση για να εκτυπώσει τα συνολοστοιχεία της λίστας.

```
class list_impl
{
    ...

    // declared as a pure virtual function
    virtual void print_elem(void*) const = 0;
    ...

    // can be used by other member functions
    void print() const
    {
        for(list_impl_elem* p = hd; p; p = p->succ)
            print_elem(p->entry);
    }
    ...
};
```

- Η υλοποίηση της `print_elem()` παρέχεται από την παραγόμενη κλάση `list<T>`.

```
template<class T>
class list : private list_impl
{
    ...

    // operator << must be defined for T
    void print_elem(void* p) const
    { cout << *(T*)p << endl; }

    void print() const { list_impl::print(); }
    ...
};
```

- Τι συμβαίνει όταν δηλώνουμε μια λίστα,

π.χ. με `list<char * > L;`

Ο ορισμός της `print_elem()` σε σχέση με τον τελεστή `<<` συσχετίζεται με την `L`.

Η κλήση `L.print()` οδηγεί σε μια κλήση της `list_impl::print_elem`, μέσω της `list_impl::print()`, στην οποία χρησιμοποιείται η υλοποίηση της `print_elem()` που είναι δεσμευμένη με την `L`.

- Π.χ.-2:
Έστω `bin_tree` η κλάση υλοποίησης (δυναμικό δένδρο διερεύνησης) ενός λεξικού `dictionary<K, I>`. Η `bin_tree` (όπως και κάθε κλάση υλοποίησης ενός `dictionary<K, I>`) χρησιμοποιεί για τη σύγκριση των κλειδιών μια γνήσια εικονική συνάρτηση `int cmp_key(void*, void*)`.

```
class bin_tree
{
    ...
    virtual int cmp_key(void*, void*) const = 0;
    ...
};
```

- Στην παραγόμενη κλάση `dictionary<K, I>`, η `cmp_key()` ορίζεται μέσω της καθολικής συνάρτησης σύγκρισης του τύπου `K`.

```
template<class K, class I>
class dictionary: private bin_tree
{
    ...
    int cmp_key(void* x, void* y) const
    { return compare(*(K*)x, *(K*)y); }
    ...
};
```

Συνάρτηση Κατασκευής Αντιγράφου, Συνάρτηση Κατάργησης, και Τελεστής Ανάθεσης

- Πώς υλοποιείται η εντολή $D1 = D2$, όπου τα $D1$ και $D2$ είναι τύπου $dictionary<K, I>$;

1. Υλοποίηση του τελεστή μέσα στην κλάση τύπου δεδομένων, δηλ. στην $dictionary<K, I>$. Π.χ.,

```
D1.clear();  
forall_items(it, D2)  
{ D1.insert(D2.key(it), D2.inf(it)); }
```

⇒ η ανάθεση χρειάζεται χρόνο $O(n \log n)$.

2. Υλοποίηση του τελεστή μέσα στην κλάση δομής δεδομένων, δηλ. στην bin_tree . Σε αυτή την περίπτωση η κλάση bin_tree πρέπει να γνωρίζει πώς αντιγράφονται κλειδιά (αντικείμενα του τύπου K) και πληροφορίες (αντικείμενα του τύπου I) καθώς και πώς αυτά καταργούνται.



Εικονικές συναρτήσεις και δυναμική δέσμευση.

Παράμετροι Υλοποίησης

Τι ιδιότητες πρέπει να έχει ένας μηχανισμός για επιλογή μεταξύ διαφορετικών υλοποιήσεων ενός τύπου δεδομένων

(1) Απλότητα στη σύνταξη. Π.χ.

```
_dictionary<K,I,rb_tree> D;  
  // chooses the red-black tree  
  // implementation of dictionaries
```

```
dictionary<K,I> D;  
  // selects the default  
  // implementation (skiplists)
```

Παρατήρηση: Χρησιμοποιούνται διαφορετικά ονόματα, γιατί σε παλαιότερους μεταγλωττιστές δεν επιτρεπόταν η υπερφόρτωση αρχετύπων. Στις νεότερες εκδόσεις της LEDA ο χαρακτήρας `_` στην αρχή του ονόματος δεν είναι πλέον απαραίτητος.

(2) Πρέπει να μπορούν να γραφούν εφαρμογές που να δουλεύουν με κάθε υλοποίηση.

Π.χ.

```
void WORD_COUNT(const list<string>& L,
                dictionary<string,int>& D)
{
    string s;
    forall(s,L)
    {
        dic_item it = D.lookup(s);
        if (it == nil)
            D.insert(s,1);
        else
            D.change_inf(it,D.inf(it)+1);
    }
    dic_item it;
    forall_items(it,D)
        cout << D.key(it) << " appeared "
             << D.inf(it) << " times.";
}
```

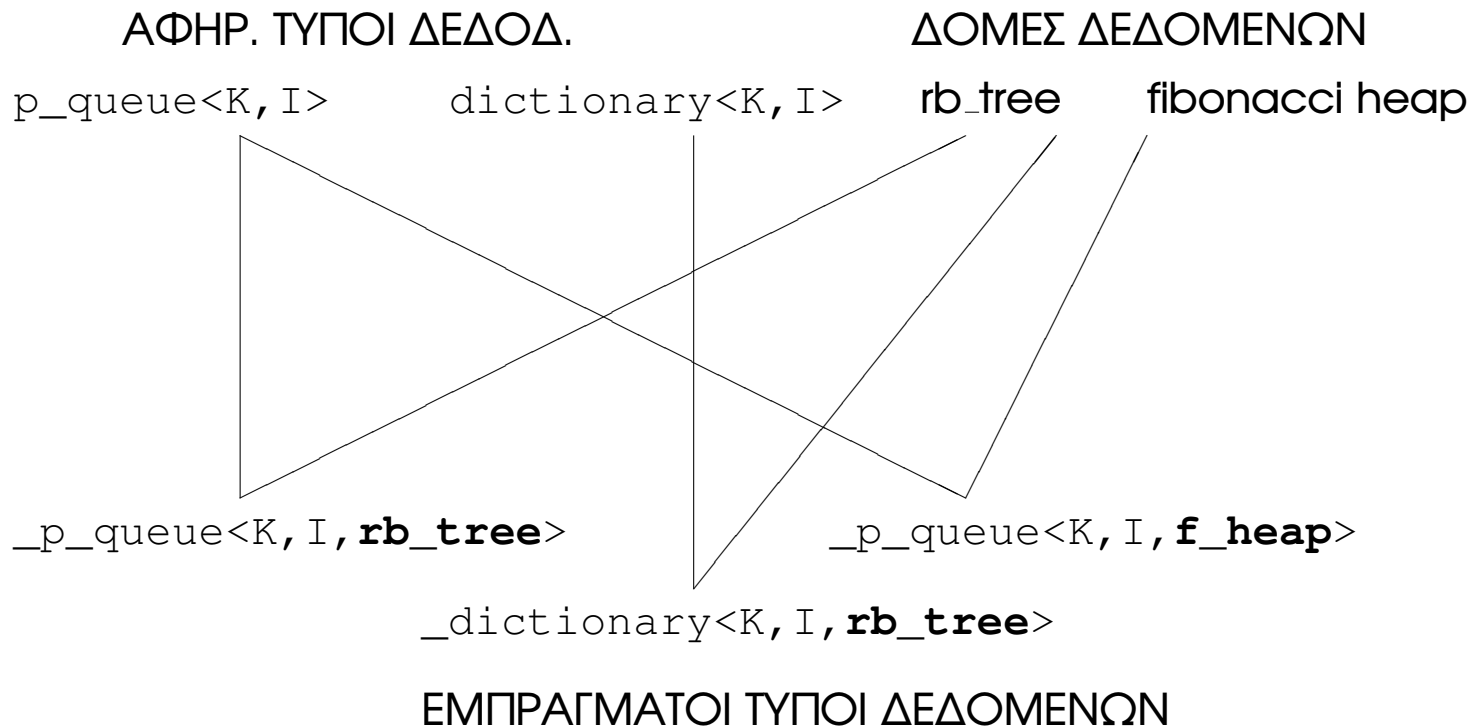
Με τις δηλώσεις

```
dictionary<string, int>          SL_D; // skiplists  
_dictionary<string, int, rb_tree> RB_D; // red-black trees  
_dictionary<string, int, my_impl> MY_D;  
                                // user implementation
```

μπορούμε να κάνουμε τις παρακάτω κλήσεις

```
WORD_COUNT(L, SL_D);  
WORD_COUNT(L, RB_D);  
WORD_COUNT(L, MY_D);
```

- Η υλοποίηση του μηχανισμού επιλογής μεταξύ διαφορετικών υλοποιήσεων ενός τύπου δεδομένων χρησιμοποιεί πολλαπλή κληρονομικότητα.



Ορισμός της κλάσης dictionary

```
typedef void * GenPtr;

template <class T>
inline T& leda_access(const T*, const GenPtr& p)
    { return *(T*)p; }
// returns a reference to the object of type T
// pointed to by p
#define LEDA_ACCESS(T,p) leda_access((T*)0,p)

template<class K,class I>
class dictionary : private default_impl
{
    int cmp_key(GenPtr x, GenPtr y)
    { return compare(LEDAS_ACCESS(K,x), LEDAS_ACCESS(K,y)); }

    void clear_key(GenPtr x)
    { leda_clear(LEDAS_ACCESS(K,x)); }

public:
    virtual K      key(dic_item it) = 0;
    virtual dic_item lookup(K y)    = 0;
    virtual dic_item insert(K x, I y) = 0;
    virtual void   del(K y)         = 0;
    ...
};
```

Ορισμός της κλάσης `_dictionary`

```
template <class T>
inline GenPtr leda_cast(const T& x) { return (GenPtr)&x; }

template<class K, class I, class IMPL>
class _dictionary : private IMPL, public dictionary<K, I>
{
public:
    K      key(dic_item it)
    { return LEAD_ACCESS(K, IMPL::key(it)); }

    dic_item lookup(K y)
    { return IMPL::lookup(leda_cast(y)); }

    dic_item insert(K x, I y)
    { return IMPL::insert(leda_cast(x), leda_cast(y)); }

    void    del(K y)
    { IMPL::del(leda_cast(y)); }

    ...
};
```