HY352 : ΤΕΧΝΟΛΟΓΙΑ ΛΟΓΙΣΜΙΚΟΥ

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ,
ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ,
ΤΜΗΜΑ ΕΠΙΣΤΗΜΗΣ ΥΠΟΛΟΓΙΣΤΩΝ

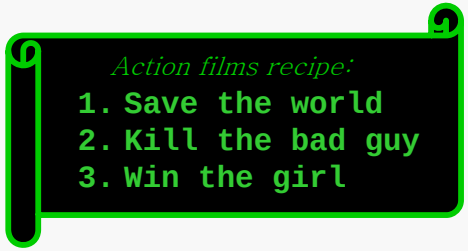
ΔΙΔΑΣΚΩΝ
Αντώνιος Σαββίδης

CSD

ΕΝΟΤΗΤΑ 5

GENERIC PROGRAMMING

Αριθμός διαλέξεων 1 – Διάλεξη 1η



CSD

Περιεχόμενα

- *Ορισμοί*
- Καλούπια
- Templates (C++)
- Generic (mixin) inheritance

CSD

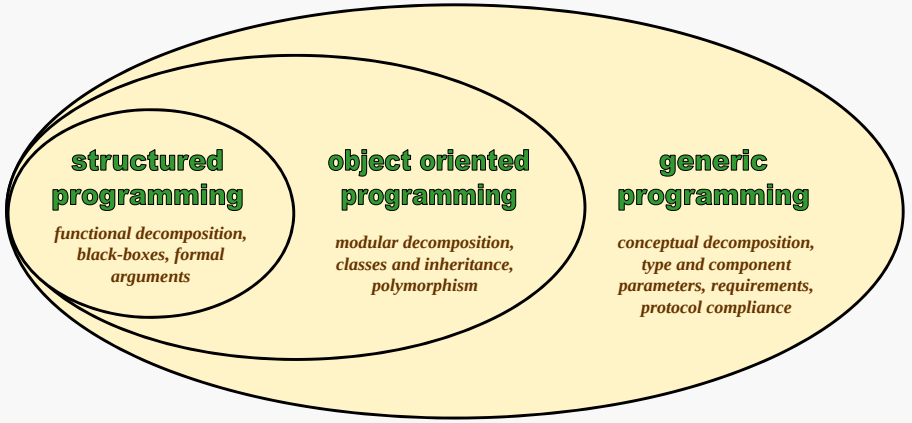
Ορισμοί (1/3)

- Τακτική προγραμματισμού η οποία ωστόσο δε συνιστά κυρίαρχο στοιχείο στις σημερινές γλώσσες, όπως συνέβηκε και με τον οντοκεντρικό προγραμματισμό
- Ο κώδικας που κατασκευάζουμε (αλγόριθμος, βιβλιοθήκη, τμήματα) μπορεί να εφαρμοστεί σε ένα ανοικτό σύνολο από τύπους (types)
  - οι οποίοι μπορεί ήδη να υπάρχουν
  - έχοντας οριστεί ανεξάρτητα από τον δικό μας κώδικα
- Οι απαιτήσεις ως προς τους τύπους δεν βασίζονται μόνο σε συμβατότητα ως προς κάποια interfaces ή superclasses
  - genericity (γενικότητα) ≠ polymorphism (πολυμορφισμό)
  - συνήθως διατυπώνεται η άποψη ότι ο πολυμορφισμός είναι ειδική περίπτωση της γενικότητας

Ορισμοί (2/3)

- Σχεδόν πάντοτε απαιτείται κάποια ειδική δυνατότητα από τη γλώσσα για να υποστηρίζεται χρήση τύπων ως παραμέτρους
- Σε αντιστοιχία με τους απλούς αλγορίθμους όπου οι παράμετροι είναι μεταβλητές
  - Έχουμε generic αλγορίθμους όταν επιπλέον οι παράμετροι τους μπορεί να είναι και τύποι δεδομένων
- Η παραμετροποίηση αλγορίθμων ως προς τους τύπους που υποστηρίζουν μπορεί να είναι:
  - *typed*: η γλώσσα ειδικό υπόστρωμα για τον ορισμό τους και ο έλεγχος γίνεται από τον compiler
  - *untyped*: δίνεται ελευθερία αλλά και ευθύνη στον προγραμματιστή για τον έλεγχο του τύπου των παραμέτρων

Ορισμοί (3/3)



Περιεχόμενα

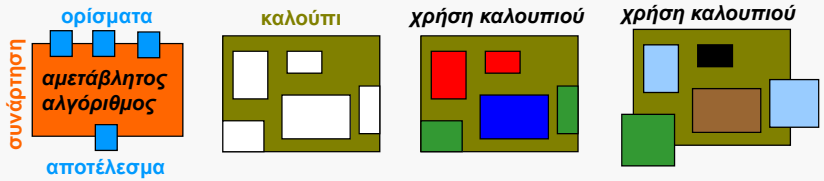
- Ορισμοί
- *Καλούπια*
- Templates (C++)
- Generic (mixin) inheritance

Καλούπια κώδικα (1/9)

- *Ένα καλούπι κώδικα (code template) ορίζεται ως ένα επαναχρησιμοποιήσιμο κομμάτι το οποίο προσφέρει παραμετροποίηση των τύπων που εμπλέκονται στην αλγοριθμική του λογική*
- Οι παράμετροι - τύποι πρέπει να τηρούν τις προϋποθέσεις που θέτει ο σχεδιαστής του καλουπιού, και σχετίζονται πάντα με το αλγοριθμικό πρόβλημα που λύνει το καλούπι
- Το ίδιο το καλούπι ενδέχεται να μη συνιστά κώδικα που μπορεί να μεταγλωττιστεί παρά μόνο εάν χρησιμοποιηθεί στο πρόγραμμα με πραγματικές παραμέτρους κλάσεις

Καλούπια κώδικα (2/9)

- Τα καλούπια κώδικα δεν εφαρμόζουν την κλασική παραμετροποίηση ορισμάτων συναρτήσεων, που ουσιαστικά πρόκειται για παραμέτρους τιμές σε αμετάβλητες αλγοριθμικά συναρτήσεις κατά την εκτέλεση,
- ...*αλλά* παραμετροποίηση τύπων, σε ημιτελή τμήματα κώδικα, με απόδοση πραγματικών ορισμάτων κατά τη μεταγλώττιση



Καλούπια κώδικα (3/9)

- Το καλούπι βασίζεται στην παρακάτω θεμελιώδη σχεδιαστική αρχή:
  - ***ενθυλάκωσε και παραμετροποίησε το μεταβλητό χαρακτηριστικό – encapsulate the feature that varies***
- Βάσει της αρχής αυτής, όταν σχεδιάζουμε παρόμοια αλγοριθμικά τμήματα κώδικα, με μοναδική διαφορά ότι απευθύνονται σε διαφορετικούς τύπους
  - ➔ *θα πρέπει να δημιουργήσουμε καλούπια στα οποία αυτοί οι τύποι είναι παράμετροι*

Καλούπια κώδικα (4/9)

- Θα μελετήσουμε την κατασκευή καλουπιών με τη βοήθεια του C preprocessor (1/4)

```
int min (int x, int y) { return x < y ? x : y; }

int a, b;
int mi = min(a,b);

float x, y;
float mf = min(x, y);            ← Χάνουμε όλο το δεκαδικό κομμάτι, δηλ. ακρίβεια

class A {
   public:
   bool operator<(const A&) const;
};
A a1, a2;
int ma = min(a1, a2);            ← Compile error, αν και ισχύει η ίδια λογική
```


Καλούπια κώδικα (5/9)

Καλούπια με το C preprocessor (2/4)

1η λύση με υπερφόρτωση συναρτήσεων – άκομψα επαναλαμβανόμενος κώδικας

```
int min (int x, int y)          { return x < y ? x : y; }
float min (float x, float y)    { return x < y ? x : y; }
A& min (A& a1, A& a2)           { return x < y ? x : y; }
```

2η λύση με πολύ απλό καλούπι κώδικα – ο κώδικας επαναλαμβάνεται αυτόματα στα σημεία χρήσης του καλουπιού

```
#define MIN(x,y) ((x) < (y) ? (x) : (y))
```

Πιο προχωρημένο παράδειγμα είναι η κατασκευή αλγορίθμου για την εύρεση του μικρότερου στοιχείου σε συλλογή από συνεχόμενα στοιχεία:

```
int min (int* arr, unsigned n) {
  return n>1 ? MIN(*arr, min(arr+1, n-1)) : *arr;  ← Προσοχή στο διπλό υπολογισμό !
}

#define MINMANY(T) \
T min (T* arr, unsigned n) \
  { return n>1 ? MIN(*arr, min(arr+1, n-1)) : *arr;  }
```

Καλούπια κώδικα (6/9)

Καλούπια με το C preprocessor (3/4)

Για να χρησιμοποιήσετε την αντίστοιχη σε τύπο συνάρτηση πρέπει να κάνετε *instantiation* το καλούπι, *δηλ. δήλωση του macro*, αλλά μόνο μία φορά στο πρόγραμμά σας, όπως παρακάτω:

```
MINMANY(int)
MINMANY(float)
MINMANY(unsigned int)
MINMANY(A)
```

Αλλά υπάρχουν και άλλες ευρεστικές λύσεις που μπορεί να σκεφτείτε:

```
#define MINMANYIMPL \
  return n>1 ? MIN(*arr, min(arr+1, n-1)) : *arr;

int   min (int* arr, unsigned n)   { MINMANYIMPL }
float min (float* arr, unsigned n) { MINMANYIMPL }
A     min (A* arr, unsigned n)     { MINMANYIMPL }
```

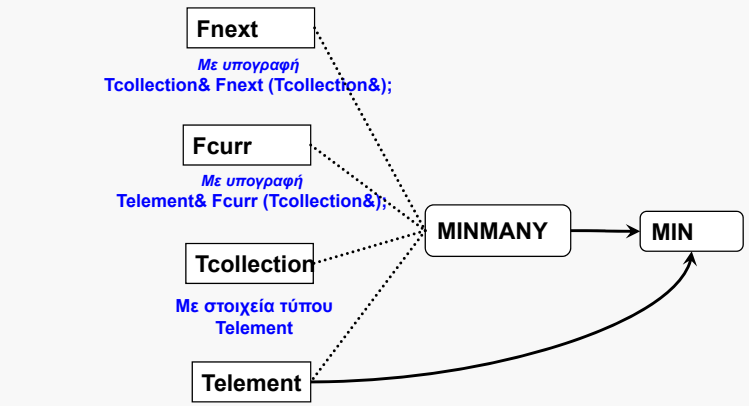
Όμως το καλούπι αυτό δεν είναι αρκετά γενικό, αφού δουλεύει μόνο σε υλοποίηση πινάκων μέσω δεικτών. Ας μελετήσουμε λοιπόν ποια είναι εκείνα τα χαρακτηριστικά που διαφοροποιούνται για καλυτέρευση του καλουπιού: (α) τύπος στοιχείου, (β) τύπος συλλογής, (γ) τρόπος μετάβασης στο επόμενο στοιχείο, και (δ) τρόπος εξαγωγής του παρόντος στοιχείου.

Καλούπια κώδικα (7/9)

Καλούπια με το C preprocessor (4/4)

```
#define MINMANY(Telement, Tcollection, Fcurr, Fnext) \
Telement min(Tcollection arr, unsigned n) \
  { return n>1 ? MIN(Fcurr(arr), min(Fnext(arr), n-1)) : Fcurr(arr); }

MINMANY(int, int*, *, 1+)        ← Σωστό, αλλά κακή χρήση γνώσης κατασκευής του macro

int  ptrarrcurr (int* arr) { return *arr; }
int* ptrarrnext (int* arr) { return arr+1; }

#define PTRARRCURR(arr) *(arr)   ← Η καλύτερα με καλούπια για όλους τους τύπους
#define PTRARRNEXT(arr) (arr)+1

MINMANY(int, int*, PTRARRCURR, PTRARRNEXT)
MINMANY(float, float*, PTRARRCURR, PTRARRNEXT)

struct ShapeList { Shape* shape; ShapeList* next; };
inline ShapeList*     next (ShapeList* p) { return p->next; }
inline Shape&         curr (ShapeList* p) { return *(p->shape); }
MINMANY(Shape&, ShapeList*, curr, next)
```


Καλούπια κώδικα (8/9)



Τεκμηρίωση κανόνων χρήσης

Καλούπια κώδικα (9/9)

- Οι κανόνες χρήσης τυποποιούν κάποιους περιορισμούς για τους πραγματικούς τύπους που μπορεί να δεχθεί ένα καλούπι
  - Κάποιες φορές η γλώσσα υποστηρίζει τον ορισμό τέτοιων περιορισμών πλήρως ή έστω σε κάποιο ικανοποιητικό βαθμό
  - αλλιώς αποτελούν τεκμηρίωση που οφείλουν να γνωρίζουν και σέβονται οι χρήστες τους
- Πολλές φορές τη χρήση ενός καλουπιού η γενικού αλγορίθμου με συγκεκριμένες παραμέτρους-τύπους λέγεται *instantiation*

Περιεχόμενα

- Ορισμοί
- Καλούπια
- *Templates (C++)*
- Generic (mixin) inheritance

Templates (1/13)

- Η χρήση του preprocessor είναι ατελής λύση γιατί έχει ορισμένες βασικές αρνητικές συνέπειες:
  - επειδή οι συναρτήσεις ορίζονται με τις δηλώσεις των macros, ενδέχεται σε ανεξάρτητα τμήματα κώδικα να υπάρχουν ***πολλαπλοί ορισμοί***, αλλιώς πρέπει οι προγραμματιστές να συνεννοούνται μεταξύ τους
    - το ίδιο ισχύει και στην περίπτωση που τα καλούπια είναι κατασκευή κλάσεων και όχι απλώς συναρτήσεων
  - ο κώδικας είναι δύσκολα αναγνώσιμος, και περιπλέκεται με την ανάγκη ονομασίας των κλάσεων όταν το καλούπι χρησιμοποιείται
  - μερικοί debuggers δεν μπορούν να κάνουν step-in και trace σε κώδικα που βρίσκεται σε macros
  - υπάρχουν αρκετές προχωρημένες σχεδιαστικές τεχνικές για καλούπια που απλά δεν γίνονται με macros

Templates (2/13)

- Τα templates είναι ένας προηγμένος μηχανισμός της C++ για την κατασκευή καλουπιών κώδικα, τόσο συναρτήσεων όσο και κλάσεων
- Βασίζονται στην εγγενή υποστήριξη παραμέτρων οι οποίες είναι κλάσεις (δηλ. τύποι)
- Στη χρήση τους ο προγραμματιστής πρέπει να δώσει ως αντίστοιχα ορίσματα κατάλληλες κλάσεις (που είναι ήδη δηλωμένες στο πρόγραμμα πριν το σημείο της χρήσης του template)

Templates (3/13)

παραδείγματα

Στην θέση του T μπορείτε να βάλετε οποιοδήποτε νόμιμο αναγνωριστικό όνομα για τον τύπο επιθυμείτε (είθισται, όμως να βάζουμε T):

```
template <class T>
T* NeverNull (T* ptr) { assert(ptr); return ptr; }

int*   pi = (int*) 0;
float* pf = (float*) 0;
*NeverNull(pi) = 10;     ← θα αποτύχει το assertion
*NeverNull(pf) = 20.0;   ← θα αποτύχει το assertion
```

Ο compiler κάνει αυτόματα matching του template parameter βάσει των παραμέτρων κλήσης:

```
template <class T> T* Find (T* arr, unsigned n, const T& x) {
   return n ? (*arr == x ? arr : Find(arr+1, n-1, x)) : (T*) 0;
}
int    iarr[] = {1, 2, 4, 5};
int*   ipos   = Find(iarr, sizeof(iarr) / sizeof(int), 3);
float  farr[] = {0.3465754, 46674.6788768, .9878, 1.00001};
float* fpos   = Find(farr, sizeof(farr) / sizeof(float), (float) 3.141628);
```

Templates (4/13)

```
template <typename T> class List {
    private:
    struct Node {
        T val;
        Node* next;
        Node* prev;
        Node (Node* l, Node* r, const T& x) : val(x)  {
            if (l) l->next = this;
            prev = l;
            if (r) r->prev = this;
            next = r;
        }

    };
    Node* head;
    Node* tail;

    public:
    void PushBack (const T& x);
    void PushFront (const T& x) {
        Node* v = new Node((Node*) 0, head, x);
        head = v;
        if (!tail) tail = v;
    }
};
```

"Parameterize the feature that varies"

Use List<T> with T as any non-void type

- Τα ***class templates*** είναι πολύ συνηθισμένα και μπορεί να είναι αρκετά απλά, έως πάρα πολύ πολύπλοκα
- Συνήθως θέτουν προϋποθέσεις για τους τύπους-παραμέτρους οι οποίοι θα πρέπει να τεκμηριώνονται από τον κατασκευαστή και να ικανοποιούνται από τον χρήστη
- Π.χ., στο διπλανό template για τον τύπο T θα πρέπει να υπάρχει ορισμένος ο copy constructor

Templates (5/13)

```
// Requirements:
// Fhash    -> functor size_t operator()(const Tkey&);
// Fequal   -> functor bool opeartor()(const Tval&, const Tval&);
// Tkey     -> non-void, copy constructible, equality comparable
// Tval     -> non-void, copy constructible, equality comparable
//
template <
    typename Tkey,
    typename Tval,
    typename Fhash,
    typename Fequal
> class Dictionary {
    private:
    struct Bucket {
        Tval val;
        Tkey key;
        Bucket* next;
    };
    Bucket* table[211];
    public:
    const Tval& operator[](const Tkey& key) const {
        for (Bucket* b = table[Fhash()(key)]; b && b->key != key; b=b->next)
            ;
        return b? b->val : *((Tval*)0);
    }
};
```

"Parameterize the features that vary"

- Οι κανόνες που πρέπει να ισχύουν για τα template parameters ορίζουν συγκεκριμένους περιορισμούς και απαιτήσεις - constraints and requirements.
- Πρέπει να τεκμηριώνονται με σαφήνεια και λεπτομέρεια αφού απαιτούνται από τον κώδικα του template

Templates (6/13)

Requirements and constraints (1/2)

```
template <class Comparable>
const Comparable& pick(const Comparable& x, const Comparable& y) {
    if (better(x, y))
        return x;
    else
        return y;
}

struct Apple {
    int rating;
    Apple(int r) : rating(r) {}
};

bool better(const Apple& a, const Apple& b)
    { return b.rating < a.rating; }

struct Orange {
    std::string name;
    Orange(const std::string& s) : name(s) { }
};

bool better(const Orange& a, const Orange& b)
    { return strcmp(a.name.c_str(), b.name.c_str()) > 0; }
```

Generic algorithm to pick the best of two comparable items

In C++, via templates

Comparable

bool better(const T&, const T&)

•Υλοποιούμε έναν generic algorithm με το όνομα pick.

•Η απαίτηση που τίθεται έμμεσα είναι να υφίσταται για το Comparable (είναι απλώς το όνομα του type parameter) μία συνάρτηση όπως φαίνεται παρακάτω.

•Υλοποιούμε δύο ανεξάρτητες μεταξύ τους classes. Για να μπορεί να χρησιμοποιηθεί η συνάρτηση pick θα πρέπει να οριστούν και οι αντίστοιχες better συναρτήσεις (μέσω overloading).

Templates (7/13)

Requirements and constraints (2/2)

```
interface Comparable<T> {
    boolean better(T x);
}

class pick {
    static <T extends Comparable<T>>
    T pick(T a, T b) {
        if (a.better(b))
            return a;
        else
            return b;
    }
}

class Apple implements Comparable<Apple> {
    int rating;
    Apple(int r) { rating = r; }
    public boolean better(Apple x)
        { return x.rating < rating;}
}
```

Constraining with sub-typing

In Java, using generics

```
interface Comparable<T>
    { bool better(T x); }

class pick {
    static T go<T>(T a, T b) where T : Comparable<T> {
        if (a.better(b))
            return a;
        else
            return b;
    }
}

class Apple : Comparable<Apple> {
    private int rating;
    public Apple(int r) { rating = r;}
    public bool better(Apple x)
        { return x.rating < rating; }
}
```

Constraining with sub-typing

In C#, using generics

Templates (8/13)

- Η σχέση των templates με τον γενικό προγραμματισμό:
  - *Γενικοί αλγόριθμοι = function templates*
  - *Γενικά components = class templates*
    - Επιτρέπονται template methods σε ένα class template
  - Υπάρχουν και περιπτώσεις templates με πολύ ειδικό ρόλο που δεν είναι ούτε αλγόριθμοι ούτε components, αλλά χρησιμεύουν στο να εξάγουν type information από υπάρχοντες τύπους
    - Βασίζονται σε μία ειδική μέθοδο στη C++ που λέγεται partial template specialization

Templates (9/13)

- Γενικά χρησιμοποιούμε templates classes όταν χρειαζόμαστε:
  - containers που πρέπει να γενικεύονται ως προς το τι *αποθηκεύουν* (π.χ. λίστες, managers, πίνακες)
  - classes που γενικεύονται ως προς τις διάφορες άλλες classes που χρειάζονται, όταν απαιτείται πρόσβαση σε data types των κλάσεων αυτών
    - Τα data types που είναι ορισμένα σε ένα class λέγονται associated types. Δεν υποστηρίζουν όλες οι γλώσσες associated data types.
- Πολλές φορές αυτό που ποικίλει μπορεί να μοντελοποιηθεί ευκολότερα με inheritance παρά με templates
  - Όταν ένα class χρησιμοποιεί άλλα classes, με ανάγκες μόνο τη χρήση κάποιων λειτουργιών με παγιωμένα signatures, τότε αυτό που θέλουμε είναι API compliance (interface / abstract class)

Templates (10/13)

```
// USE THIS STYLE
class InputStream {
    virtual char get (void) = 0;
    virtual bool unget (void) = 0;
    virtual bool eof (void) const = 0;
};

class LexicalAnalyzer {
private:
    InputStream& input;
public:
    LexicalAnalyzer (InputStream& _input) : input(_input){}
};

// AND NOT THIS ONE
template <typename Tinput> class LexicalAnalyzer2 {
private:
    Tinput input;
public:
    LexicalAnalyzer2 (void){}
};
```

Parameterization with respect to an API is interface compliance. So we need an interface parameter

LexicalAnalyzer2<A> ≠ LexicalAnalyzer2<B> για οποιουσδήποτε τύπους A≠B

- Εν γένει δύο διαφορετικά instantiations ενός template class είναι classes διαφορετικού τύπου γεγονός που καθιστά τα αντίστοιχα object instances των classes ασύμβατα.
- Όταν το API που απαιτείται είναι fixed και δεν εμπεριέχει templates θέλουμε inheritance (δηλαδή ως παράμετρο ένα reference σε object που είναι compliant ως προς το αναγκαίο interface) και όχι templates.

Templates (11/13)

```
// STYLE-1: Non-template method in a template class.
template <typename T> struct Min_1 {
    const T operator()(const T& x, const T& y) const
        { return x < y ? x : y; }
};

// STYLE-2: Template function
template <typename T>
const T Min_2 (const T& x, const T& y)
        { return x < y ? x : y; }

// STYLE-3: Template method in non-template class
struct Min_3 {
template <typename T>
const T operator()(const T& x, const T& y) const
        { return x < y ? x : y; }
};

static void Test (void) {
    int a = Min_1<int>()(10, 20);
    int b = Min_2(30, 40);
    int c = Min_3()(50, 60);
}
```

Μπορούμε να έχουμε διαφορετικά στυλ υλοποίησης των generic algorithms, με σχετικά πλεονεκτήματα και μειονεκτήματα.

- Το ενδιαφέρον των function templates είναι ότι δίνουν τη δυνατότητα να κάνουμε ***extract τους τύπους των arguments*** χωρίς να χρειάζεται να τους απαριθμήσουμε όπως απαιτείται στα class templates.
- Η παραπάνω δυνατότητα συνήθως χρησιμοποιείται ως εξής: φτιάχνουμε template functions που δημιουργούν και επιστρέφουν instances ενός class template instantiation.

Templates (12/13)

```
template <typename Tf, typename Tg> struct composer {
    typedef typename Tg::result_t result_t;
    typedef typename Tf::arg1_t arg1_t;
    typedef typename Tf::arg2_t arg2_t;
    Tf f; Tg g;
    const result_t operator()(arg1_t _1, arg2_t _2) const
        { return g(f(_1, _2)); }
    composer (const composer& c) : f(c.f), g(c.g){}
    composer (const Tf& _f, const Tg& _g) : f(_f), g(_g) {}
};

template <typename Tf, typename Tg>
composer<Tf, Tg> compose (const Tf& f, const Tg& g)
    { return composer<Tf,Tg>(f,g); }
```

associated types

compose(f,g) = f ° g

```
template <typename R, typename A1, typename A2> struct functorizer2 {
    typedef R result_t;
    typedef A1 arg1_t;
    typedef A2 arg2_t;
    R (*f)(A1,A2);
    const result_t operator()( arg1_t _1, arg2_t _2) const
        { return(*f)(_1, _2); }
    functorizer2(const functorizer2& p) : f(p.f){}
    functorizer2 (R (*_f)(A1, A2)):f(_f){}
};

template <typename R, typename A1, typename A2>
const functorizer2<R,A1,A2> functorize (R (*f)(A1,A2))
    { return functorizer2<R,A1,A2>(f); }
```

- Πρώτα υλοποιούμε το composition πάνω σε functor classes χρησιμοποιώντας associated types.
- Η κλάση composer δέχεται ως constructor arguments δύο functors για την *f* και *g* και δημιουργεί ένα instance το οποίο υλοποιεί τον function call operator ώστε να συμπεριφέρεται ακριβώς όπως η σύνθεση των *f* και *g*.
- Για την μετατροπή μίας συνάρτησης σε functor instance χρησιμοποιούμε μία generic συνάρτηση που λαμβάνει function argument και επιστρέφει functor instance.
- Όμως χρειαζόμαστε ένα τύπο functorizer ανά πλήθος τυπικών ορισμάτων.

Templates (13/13)

```
template <typename R, typename A> struct functorizer1 {
    typedef R result_t;
    typedef A arg1_t;
    R (*f)(A);
    const result_t operator()(arg1_t _1) const
        { return(*f)(_1); }
    functorizer1(const functorizer1& p) : f(p.f){}
    functorizer1 (R (*_f)(A)):f(_f){}
};

template <typename R, typename A>
const functorizer1<R,A> functorize (R (*f)(A))
    { return functorizer1<R,A>(f); }
```

```
int _f(const std::string&, const std::string&);
std::string _g (int);

}static void dummy(void) {
    std::string s = compose(
                        functorize(&_f),
                        functorize(&_g)
                    )("hello", "world");
}
```

- Ωστόσο βασισμένοι στο overloading υλοποιούμε διαφορετικές εκδοχές της *functorize* ανά τύπο συνάρτησης (functor)
- Ο προγραμματιστής βλέπει απλώς μία generic overloaded functorize συνάρτηση και μία αντίστοιχη compose.
- Θα μπορούσαμε να κάνουμε την *compose* να υποστηρίζει *f* με οποιοδήποτε αριθμό από arguments (η παρούσα υλοποίηση υποστηρίζει μόνο δύο arguments)

Περιεχόμενα

- Ορισμοί
- Καλούπια
- Templates (C++)
- *Generic (mixin) inheritance*

Generic (mixin) inheritance (1/4)

- Στο κλασικό μοντέλο inheritance ορίζουμε ένα subclass ***D*** βασισμένοι σε ένα συγκεκριμένο base class ***B***.
  - `class ShadowedWindow : public Window {}`
- Μπορεί ωστόσο η λειτουργικότητα που προσφέρει το ίδιο το subclass πάνω στο base class να έχει νόημα όχι απλώς ως προς το base αλλά οτιδήποτε είναι base
  - `class PushButton : public Window {}`
  - `class ShadowedPushButton : public PushButton {}`
- Ο μόνος τρόπος να χρησιμοποιηθεί εκ νέου η λειτουργικότητα τύπου “shadow” είναι μέσω derivation, κάτι που είναι ιδιαίτερα επίπονο.

CSD

Generic (mixin) inheritance (2/4)

- Θα θέλαμε να μπορούμε να έχουμε ένα generic subclass *Shadowed* που μπορεί να εφαρμόζεται σε οποιοδήποτε base class, υπό κάποιες προϋποθέσεις, χωρίς να ορίζουμε κάθε φορά ένα εξειδικευμένο subclass.

`mixin Shadowed[T:Window] {` `}`	`mixin Shadowed[T] {` `}`

Χρήση μέσα στο πρόγραμμα με τρόπο που δίνει τη δυνατότητα για compositional inheritance (λέγεται και inheritance on demand):
`Shadowed[Window] sw = new Shadowed[Window] (...);`
`Shadowed[PushButton] sb = new Shadowed[PushButton] (...);`	

CSD

Generic (mixin) inheritance (3/4)

- Η μέθοδος αυτή ορίστηκε ως mixin-based inheritance το 1990 (Bracha & Cook) με απόδειξη ότι είναι ισοδύναμη με το classical inheritance
- Ωστόσο υπάρχουν κάποια προβλήματα σχετικά με τη σειρά εφαρμογής των mixins επειδή μπορεί να προκύψουν εναλλακτικά hierarchies. Π.χ:
  - **Shadowed[Framed[Window]] ≠ Framed [Shadowed [Window]]**
- Για το λόγο αυτό πρέπει χρησιμοποιείται με ιδιαίτερη προσοχή καθώς μπορεί να χάσουμε τον πολυμορφισμό

CSD

Generic (mixin) inheritance (4/4)

```
template <class Twin>
class Shadowed : public Twin {};

class Window{};
class Button : public Window{};
class Scrollbar : public Window{};

typedef Shadowed<Button>    ShadowedButton;
typedef Shadowed<Scrollbar> ShadowedScrollbar;

void f (Shadowed& s) {// Error, no Shadowed superclass (or class).
}

template <class T>
void f (Shadowed<T>& s) {
    // Now we can use all shadowed features.
    // Polymorphism with mixins requires generic
    // functions.
}

void g (void) {
    ShadowedButton* sb;
    ShadowedScrollbar* ss;
    f(*sb);
    f(*ss);
}
```

- Στη C++, τα mixin classes γίνονται template classes με παράμετρο την base class
- Μπορούμε εκτός της άμεσης χρήσης για ορισμό instances να έχουμε και typedefs για ευκολία.
- Ωστόσο τα mixins δεν υφίστανται ως base classes που σημαίνει ότι ο μόνος τρόπος να έχω πολυμορφισμό είναι με generic functions που παραμετροποιούν το type parameter του mixin.