ΗΥ352 : ΕΡΓΑΣΤΗΡΙΟ ΤΕΧΝΟΛΟΓΙΑΣ ΛΟΓΙΣΜΙΚΟΥ

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ,
ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ,
ΤΜΗΜΑ ΕΠΙΣΤΗΜΗΣ ΥΠΟΛΟΓΙΣΤΩΝ

6ο ΦΡΟΝΤΙΣΤΗΡΙΟ

ΔΙΔΑΣΚΟΝΤΑΣ
Αντώνιος Σαββίδης

Περιεχόμενα

- Templates

Το πρόβλημα

- Υπάρχει πολλές φορές η ανάγκη να έχουμε παραμετροποιημένους τύπους σε κάποια κλάση ή συνάρτηση
  - Π.χ. μπορεί να θέλουμε ένα container (όπως μια λίστα ή μια στοίβα) που να μπορεί άλλοτε να περιέχει ακεραίους, άλλοτε χαρακτήρες, άλλοτε αντικείμενα κάποιας user defined κλάσης, κ.λ.π.
  - Προφανώς θέλουμε κάτι παραπάνω από το να ξαναγράφουμε τον κώδικα της κλάσης για κάθε διαφορετικό τύπο

Πιθανές λύσεις

- C style, φτιάχνοντας macros με παράμετρο τον τύπο και χρησιμοποιώντας τα κατάλληλα
  - #define IMPL_MAX(T) T max_##T(T a, T b) { return a>b ? a : b; }
  - IMPL_MAX(int) IMPL_MAX(double)
  - max_int(1, 2); max_double(4.5, 3.14);
- Java style, βάζοντας κάθε κλάση να κληρονομεί από το generic base Object και δημιουργώντας μια object-based ιεραρχία που χρησιμοποιούμε για επαναχρησιμοποίηση κώδικα
  - LinkedList l = new LinkedList(); l.add(new Object());
  - Από την Java 1.5 υπάρχουν πλέον και τα *Generics*
- C++ style, **Templates!**

Εισαγωγή στα templates (1/2)

- Η έννοια ενός παραμετροποιημένου τύπου υλοποιείται στη C++ με τα *templates*
- Δηλώνονται με το keyword ***template***, ενημερώνοντας τον compiler ότι η δήλωση τύπου που ακολουθεί θα περιέχει και ορισμένους μη προσδιορισμένους τύπους δεδομένων
- Οι τύποι αυτοί συγκεκριμενοποιούνται κατά την χρήση (instantiation) του template οπότε και ο compiler παράγει τον πραγματικό κώδικα για την εκάστοτε δήλωση

Εισαγωγή στα templates (2/2)

- Στη C++ έχουμε δύο είδη templates
  - ***Template functions***
    - Π.χ. γενικές συναρτήσεις αναζήτησης και ταξινόμησης για οποιονδήποτε τύπο δεδομένων
  - ***Template classes***
    - Π.χ. containers που μπορούν να περιέχουν οποιονδήποτε τύπο δεδομένων
- Χαρακτηριστικό παράδειγμα templates αποτελεί η STL (Standard Template Library) που παρέχει γενικούς αλγορίθμους ως template functions και γενικά containers ως template classes

Template functions

- Είναι σαν κανονικές συναρτήσεις, απλά πριν τη δήλωσή τους προηγείται το keyword template με τις παραμέτρους μέσα σε **<>**
- Τα template parameters δηλώνονται ως *class T* ή *typename T* (όπου *T* κάποιο identifier) και μπορούν να χρησιμοποιηθούν ως τύποι στο σώμα της συνάρτησης
  - Οι τιμές τους μπορούν να δοθούν explicitly κατά το instantiation ή implicitly κάνοντας match τους τύπους των arguments

```
template<class T> T max(T a, T b) { return a > b ? a : b; }
int main() {
  int i = max(10, 15);                  //implicit int parameter
  i = max<int>(10, 20);                 //explicit int parameter
  char c = max('k', 's');               //implicit char parameter
  double d = max<double>(10.1, 15);     //ok, explicit double parameter
  i = max('k', 10); //compile error: ambiguous template parameter
  return 0;          //'k' is char, 10 is int so T can't match both
}
```

Template classes

- Δηλώνουμε το template class με παράμετρο ***class T***
  - Ή και με ***typename T*** (είναι το ίδιο)
- Το T μπορεί πλέον να χρησιμοποιηθεί μέσα στο σώμα της κλάσης ως τύπος
- Τα Array<int>, Array<float> είναι instantiations που ορίζουν δυο νέες κλάσεις
  - Σαν να είχαμε ορίσει τις κλάσεις Array_int, Array_float
  - Παρόλο που ορίζουμε παράμετρο *class* T, μπορούμε να κάνουμε instantiate με οποιοδήποτε τύπο (int, float, pointer, reference, κ.λ.π.)

```
#include <iostream>
template<class T> class Array {
    T a[100];
public:
    T& operator[](int index)
        { return a[index]; }
};
int main(){
    Array<int> ia;
    Array<float> fa;
    ia[0] = 1;
    std::cout << fa[0];
}
```

Non-inline template class functions (1/2)

- Μπορούμε να έχουμε και non-inline functions μέσα σε ένα template class
  - Προσοχή στα πολλά *T*
  - Το πρώτο είναι το *template parameter*
  - Το δεύτερο είναι ο *τύπος της επιστρεφόμενης τιμής* της συνάρτησης
  - Το τρίτο ορίζει ότι το template parameter της συνάρτησης είναι το ίδιο με αυτό της κλάσης
    - Μπορεί να έχουμε και template function σε non-template class

```
#include <iostream>
template<class T> class Array {
    T a[100];
public:
    T& operator[](int index);
};
                    1
template<class T>
 2          3
T& Array<T>::operator[](int index)
    { return a[index]; }

int main(){
    Array<int> ia;
    std::cout << ia[0];
}
```

Non-inline template class functions (2/2)

- Αν υλοποιήσουμε ένα member function εκτός class και *συγκεκριμένα στο cpp* δε θα είναι ορατά στα σημεία χρήσης του header file
- Το αποτέλεσμα θα είναι ένα link error στα σημεία χρήσης καθώς θα γίνουν instantiated μόνο οι δηλώσεις των συναρτήσεων
- Συνεπώς υλοποιούμε τις non-inline συναρτήσεις πάντα *μέσα στο* ***header***, έξω από το σώμα της κλάσης

```
//X.h
#ifndef X_H
#define X_H
template<class t> class X {
public:
   X();
   ~X();
};
#endif
```

```
// X.cpp
#include “X.h"
template<class t> X<t>::X() {}
template<class t> X<t>::~X() {}
```

```
//main.cpp
#include “X.h"
void main() {
    X<int>  xi ; // link error
    X<float> xf; // link error
}
```

Template parameters (1/2)

- Μπορούμε να έχουμε και άλλους τύπους ως template parameters, π.χ. built-in τύπους
  - Δε μπορούν να είναι floating point τύπου
  - Πρέπει να είναι *compile time constants*
  - Δε μπορούν να αλλάξουν μέσα στον ορισμό του template
- Χρησιμοποιούνται μέσα στην κλάση σαν να ήταν data members
- Μπορούμε επίσης να έχουμε και default τιμές στα template parameters
  - Ισχύουν τα γνωστά σχετικά με τις default τιμές

```
template<class T=float, int size=50>
class Array {
  T a[size];
public:
  T& operator[](int index)
        { return a[index]; }
  int length() const { return size; }
  void expand() { size++; } //error
};
int main(){
  Array<int, 20> a;//T=int,  size=20
  Array<int> a2;   //T=int,  size=50
  Array defaultA;  //T=float,size=50
}
```

Template parameters (2/2)

- Η παράμετρος ενός template θεωρείται δεσμευμένη λέξη για όλο το scope του template (και για τα inner scopes)
  - Είναι error αν ξαναδηλωθεί
  - ΠΡΟΣΟΧΗ: το visual studio το αφήνει να περάσει κανονικά χωρίς λάθη, δηλώνοντας το σαν κανονική μεταβλητή (κάνοντας shadow το template parameter)

```
int T, size;
template <class T, T size> class Array {
    int T;                  //error redefinition of template parameter
    unsigned size;          //error redefinition of template parameter
    void f() { char T; } //error redefinition of template parameter
    void func() {
       T tmp = size;        //template parameters T and size
       ::T  = ::size;       //access globals T and size using ::
};
```

Template code generation

- Παράγωγή κώδικα γίνεται μόνο για τα template class που γίνονται instantiated
- Παράγωγή κώδικα μόνο για συναρτήσεις που χρησιμοποιούνται
- Το αν θα παραχθεί κώδικας για τις **virtual** συναρτήσεις που δε χρησιμοποιούνται εξαρτάται από τον εκάστοτε compiler

```
template <class T>
class Z {
public:
    Z() {}
    ~Z() {}
    void f(){}
    virtual void g(){}
} ;

int main() {
  Z<int> zi; //instantiation of  Z<int>
  zi.f();    //generation of Z<int>::f()
  Z<float> f;//instantiation of Z<float>
  template class Z<char>; //explicit
          //instantiation of Z<char>
   Z<double>* p_zi; //no instantiation
   return 0;        //of Z<double>
}
```

Template specialization (1/3)

- Πολλές φορές ο γενικός κώδικας ενός template δεν είναι ο κατάλληλος για κάθε τύπο δεδομένων
- Χρειάζεται λοιπόν να μπορούμε να ορίσουμε *διαφορετικό κώδικα* που θα χρησιμοποιηθεί για κάποιο *συγκεκριμένο τύπο δεδομένων*
- Αυτό γίνεται με ***template specialization***

```
#include<iostream>
using std::cout; using std::endl;
template<class T> T max(T a, T b) { return a > b ? a : b; }
int main() {
    cout << max(10, 15) << endl;                  //prints 15
    cout << max('k', 's') << endl;                //prints s
    cout << max(10.1, 15.2) << endl;              //prints 15.2
    cout << max("Aladdin", "Jasmine") << endl; //prints "Aladdin"... Why?
    return 0;                          //compiler generates max(char*, char*)
}                                      //and compares string addresses!
```

Template specialization (2/3)

- Παρακάμπτουμε το γενικό ορισμό του template
  - Αφαιρούμε τους παραμετροποιημένους τύπους από το *template<>*
  - Χρησιμοποιούμε τους συγκεκριμένους τύπους μετά δήλωση του ονόματος καθώς στα σημεία χρήσης τους

```
#include<iostream>
using std::cout; using std::endl;
template<class T> T max(T a, T b) { return a > b ? a : b; }
template<> const char* max<const char*>(const char* a, const char* b)
  { return strcmp(a,b)>0 ? a:b; } //specializing for char* wouldn't work!
int main() {
    cout << max(10, 15) << endl;                  //prints 15
    cout << max('k', 's') << endl;                //prints s
    cout << max(10.1, 15.2) << endl;              //prints 15.2
    cout << max("Aladdin", "Jasmine") << endl; //prints "Jasmine"
    return 0;                    //compiler sees the specialized version and
}                                //uses that one, correctly invoking strcmp
```

Template specialization (3/3)

- Ειδικά στα template classes, μπορούμε να έχουμε specialized versions που να έχουν τελείως διαφορετικές συναρτήσεις και δεδομένα από το template version
  - Μπορούμε ακόμα και να ορίσουμε ένα specialization χωρίς να το υλοποιήσουμε, απαγορεύοντας τη χρήση του template με το συγκεκριμένο τύπο (θα είναι compile error)

```
template<class T> class mycontainer {
   T element;
public:
    mycontainer(T arg) { element = arg; }
    T increase () { return ++element; }
};

template<> class mycontainer <char> {
   char element;
   int someOtherMember;
public:
   mycontainer(char arg)
      {element = arg; }
   char uppercase () {
      if ((element>='a')&&(element<='z'))
          element+='A'-'a';
      return element;
   }
};

template<> class mycontainer <int>;
```

Template partial specialization (1/2)

- Στα template classes μπορούμε να μην κάνουμε specialize όλα τα template arguments αλλά μόνο συγκεκριμένα
  - Δε μπορούμε να κάνουμε το ίδιο στα template functions
- Αφαιρούμε τους τύπους που θέλουμε να κάνουμε specialize από τα template parameters και τους αντικαθιστούμε με τους συγκεκριμένους στα σημεία χρήσης
- Μπορούμε να έχουμε πολλά partial και/ή full specializations

```
#include <iostream>
using namespace std;
template<class T1, class T2>
struct Pair {
  T1 first; T2 second;
  void print(void)
    { cout << "original template"; }
};
template<class T1> struct Pair<T1, int> {
  T1 first; int second;
  void print(void)
    { cout << "partial specialization"; }
};
template<> struct Pair<double, int> {
  double first; int second;
  void print(void)
    { cout << "full specialization"; }
};
Pair<bool, bool>().print(); //original
Pair<bool, int>().print(); //partial
Pair<double, int>().print(); //full
```

Template partial specialization (2/2)

- Κατά το partial specialization μπορεί ο specialized τύπος να είναι πάλι template parameter

```
#include<iostream>
using namespace std;
template<class T> struct Sizeof;  //do not allow generic use of the class
template<> struct Sizeof<double> { enum { value = 8 }; };
template<> struct Sizeof<char> { enum { value = 1 }; };
template<class C> struct Sizeof<C*> { enum { value = 4 }; };

template<class T1, class T2> struct Pair { T1 first; T2 second; };
template<class T1, class T2> struct Sizeof<Pair<T1, T2>>
 { enum { value = Sizeof<T1>::value + Sizeof<T2>::value}; };

class X {};
cout << Sizeof<double>::value << endl;              //prints 8
cout << Sizeof<char*>::value << endl;               //prints 4
cout << Sizeof<X*>::value << endl;                  //prints 4
cout << Sizeof<Pair<double, char>>::value << endl;  //prints 9
```

Template class static members

- Τα template classes μπορούν να έχουν και static members
  - Που πρέπει να χρεωθούν σε κάποιο source αρχείο
- Δηλώνουμε τη static μεταβλητή για το γενικό τύπο και τις δίνουμε τιμή
- Κάνουμε specialize τη δήλωση του static member για να δώσουμε αρχικές τιμές για συγκεκριμένους τύπους
  - Πρέπει βέβαια να υπάρχει η static μεταβλητή στο αντίστοιχο specialization της κλάσης

```
//X.h
#include<iostream>
using namespace std;

template <class T>
struct X { static T s; };

template<> struct X<bool> {}; //no static
          //member in this specialization
//X.cpp
template<class T> T X<T>::s = 0;
template<> int X<int>::s = 3;
template<> char* X<char*>::s = "Hello";

template<> bool X<bool>::s=true;//error:
//no static member in bool specialization

int main() {
   cout << X<int>::s << endl;
   cout << X<char*>::s << endl;
   return 0;
}
```

Παραδοχές στη χρήση templates

- Ένα template μπορεί να κάνει παραδοχές σχετικά με το τι είδους αντικείμενα μπορεί να δεχτεί ως παραμέτρους

```
#include<iostream>
using std::cout; using std::endl;
//For template max, parameter T must have an overloaded > operator
template<class T> T max(T a, T b) { return a > b ? a : b; }
class X {};
struct Y {
   int val;
   bool operator > (const Y& y) { return this->val > y.val; }
   Y(int val) : val(val) {}
};
cout << max(10, 20) << endl;   //prints 20
cout << max(X(), X()) << endl;//compile error:there is no > operator in X
cout << max(Y(4), Y(5)).val << endl; //prints 5
//For println, there must be an overloaded << operator accepting T
template <class T> void println(T x) { cout << x << endl; }
println(Y(4)); //compile error: no appropriate << operator
println(100);  //ok, cout has an overloaded operator << accepting int
```

Templates και type definitions

- Μια καλή προγραμματιστική τεχνική είναι να χρησιμοποιείτε **typedef** όταν σκοπεύετε να χρησιμοποιήσετε στιγμιότυπα από template κλάσεις
- Έτσι, σε όλο το πρόγραμμά σας θα χρησιμοποιείτε το **typedef**-ed όνομα
- Αυτό έχει δύο πλεονεκτήματα
  - Τα **typedef** είναι πολύ χρήσιμα (λόγω απλότητας) όταν χρησιμοποιούμε templates από templates π.χ.
    - typedef vector<int, allocator<int> > IntVector;
  - Όταν ο ορισμός του template αλλάξει τότε το μόνο που θα χρειαστεί να αλλάξουμε είναι ο ορισμός του typedef. Π.χ. όταν το δεύτερο όρισμα δε μας χρειάζεται πια:
    - typedef vector<int> IntVector;

    και όλο το υπόλοιπο πρόγραμμά μας μένει ανέπαφο

Nested template argument lists

- Στα templates επιτρέπεται σαν όρισμα να περάσουμε ένα template. Όποτε το παρακάτω είναι valid:
  - std::list<std::pair<int, double> > pairList;
- Σε αυτήν την περίπτωση πρέπει να αφήσουμε ένα κενό μεταξύ του '>' στο τέλος της εσωτερικής λίστας και του '>' στο τέλος της εξωτερικής λίστας, αλλιώς θα έχουμε ambiguity μεταξύ του operator >> και των 2 '>'
  - std::map<int, std::list<int>> intMap;
- Προσοχή: στο VS το πρόγραμμα θα γίνει compile κανονικά, ενώ στο gcc υπάρχει compile error