Software Languages.Lab

Structuur van Computerprogramma's 2

dr. Dirk Deridder
Dirk.Deridder@vub.ac.be
http://soft.vub.ac.be/

Vrije Universiteit Brussel - Faculty of Science and Bio-Engineering Sciences - Computer Science Department - Pleinlaan 2 - 1050 Brussels - Belgium

Software Languages.Lab

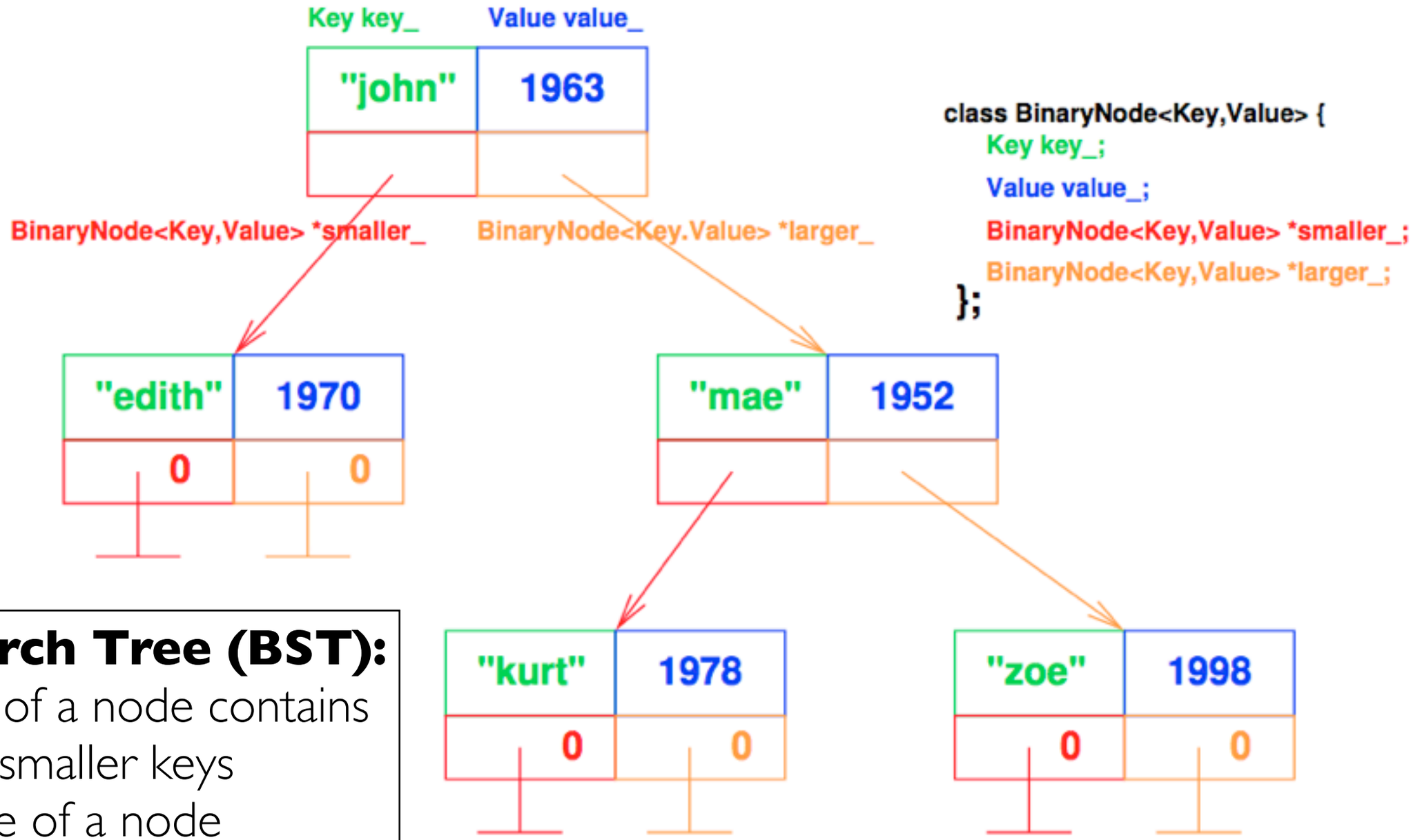
Chapter 5b - User-Defined Type Constructors

Structuur van Computerprogramma's 2

Templates

Binary Search Tree Example

Binary Search Trees




```
class BinaryNode<Key,Value> {
    Key key_;
    Value value_;
    BinaryNode<Key,Value> *smaller_;
    BinaryNode<Key,Value> *larger_;
};
```

Binary Search Tree (BST):

- left subtree of a node contains nodes with smaller keys
- right subtree of a node contains nodes with larger keys
- left and right subtrees are also Binary Search Trees

Example Program using a BST

```
#include <string>
#include <iostream>
#include "BinaryTree.h"

int main(int argc, char* argv[]) {
// a BST that can be parameterised with the type of key and value
    BinTree<std::string, unsigned int> birth_year;

// insert nodes
    birth_year.insert("lisa", 1970);
    birth_year.insert("john", 1936);
    birth_year.insert("mae", 1952);
    birth_year.insert("kurt", 1978);

// alternative way to insert
    birth_year["zoe"] = 1998;
    birth_year["john"] = 1963;

// try to find the birth year of john
    unsigned int year(0);
    if (birth_year.find("john", year))
        std::cout << "birth year of john = " << year << std::endl;
    else
       std::cout << "couldn't find john !" << std::endl;
}
```

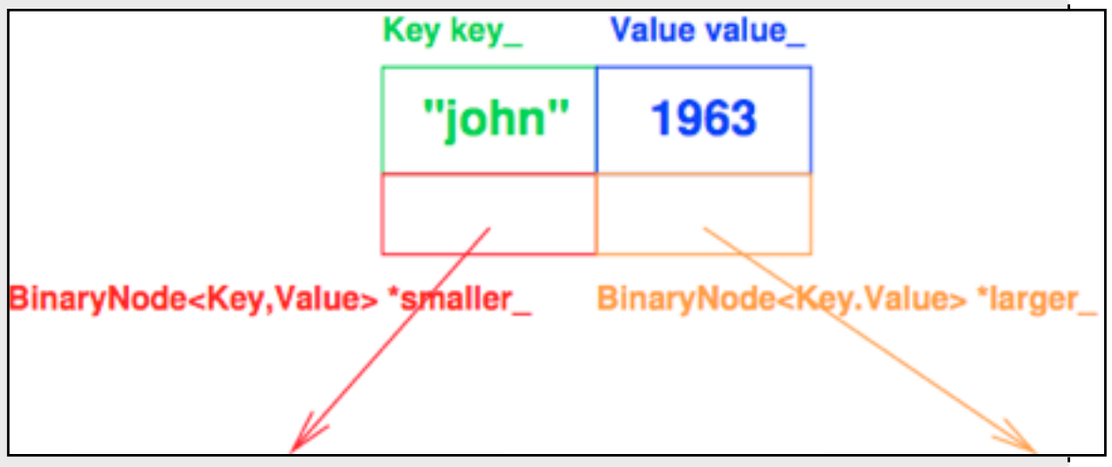
The `BinaryTree` Template Class (1)

```
#ifndef BINTREE_H
#define BINTREE_H

template<typename Key, typename Value>
class BinTree {
private:
// Node represents a node in the binary search tree. There is one Node
// class for every template instantiation of BinTree.
   struct Node { // everything of Node is public, Node itself is private
   // constructor
      Node(Key k, Value v, Node* sml = 0, Node* lrg = 0) :
         key_(k), val_(v), smaller_(sml), larger_(lrg) { }
   // data members
      Key key_;
      Value val_;
      Node* smaller_;
      Node* larger_;
   };

   Node* root_; // the root of the tree

//...
```



The `BinaryTree` Template Class (2)

```
public:
// constructor
   BinTree() :
      root_(0) { }

// destructor
   ~BinTree() {
      zap(root_); // zap deallocates all nodes starting at the argument
      root_ = 0;
   }


// try to find node with key, if found, return
// true and copy its value to val parameter
   bool find(const Key& key, Value& val) const {
      Node* node = find_node(root_, key); // find a node containing key
      if (node) {                         // we found a node, get value
         val = node->val_;
         return true;
      } else
         return false; // no such node, bad luck,
   }

// ...
```

The `BinaryTree` Template Class (3)

```
// insert inserts (key,val) in the tree,
// insert_node should return a node at right position in the tree
   void insert(const Key& key, const Value& val) {
      insert_node(root_, key)->val_ = val;
   }

// t[key] = val is alternative for insert, so operator[] returns
// a non-const reference to a node's value to be able to change it
   Value& operator[](const Key& key) {
      return insert_node(root_, key)->val_;
   }

//...
```

The `BinaryTree` Template Class (4)

```
private:
// forbid copy constructor
   BinTree(const BinTree&);

// forbid assignment
   BinTree& operator=(const BinTree&);

// zap(node) recursively deallocates memory used by
// subtree starting at node
   void zap(Node* node) {
      if (node == 0)
         return;
      zap(node->smaller_);
      zap(node->larger_);
      delete node;
   }

//...
```

The `BinaryTree` Template Class (5)

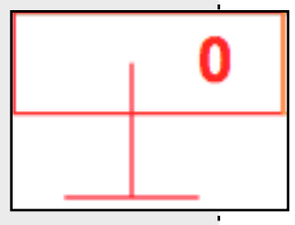
```
// find_node returns a pointer to a node containing k
// or 0, if such a node cannot be found
   Node* find_node(Node* node, const Key& key) const {
      if (node == 0)
         return 0; // key could not be found

      if (node->key_ == key)
         return node; // got it
      else
         if (key < node->key_)
            return find_node(node->smaller_, key); // look left
         else
            return find_node(node->larger_, key);  // look right
   }

// ...
```

The `BinaryTree` Template Class (6)

```
// - insert_node returns the node where key should be
// - if key cannot be found, insert_node returns a
// (pointer to a) fresh node where it should be
// - insert_node will also properly update the tree
// if it needs to create a new node; this is why
// the first parameter is a reference to a pointer
   Node* insert_node(Node*& node, const Key& key) {
      if (node == 0)
         // Don't forget: where there was a 0-pointer,
         // there will now be pointer to a fresh node.
         // Note default ctor for Value.
         return (node = new Node(key, Value()));
      if (key == node->key_) //  node with this key already exists
         return node;        //  return it so its value can be updated
      if (key < node->key_)  //  insert node in the left side
         return insert_node(node->smaller_, key);
      else                   //  insert node in the right side
         return insert_node(node->larger_, key);
   }
};
#endif
```



Example Program using a BST (Recap)

```
#include <string>
#include <iostream>
#include "BinaryTree.h"

int main(int argc, char* argv[]) {
// a BST that can be parameterised with the type of key and value
    BinTree<std::string, unsigned int> birth_year;

// insert nodes
    birth_year.insert("lisa", 1970);
    birth_year.insert("john", 1936);
    birth_year.insert("mae", 1952);
    birth_year.insert("kurt", 1978);

// alternative way to insert
    birth_year["zoe"] = 1998;
    birth_year["john"] = 1963;

// try to find the birth year of john
    unsigned int year(0);
    if (birth_year.find("john", year))
        std::cout << "birth year of john = " << year << std::endl;
    else
       std::cout << "couldn't find john !" << std::endl;
}
```

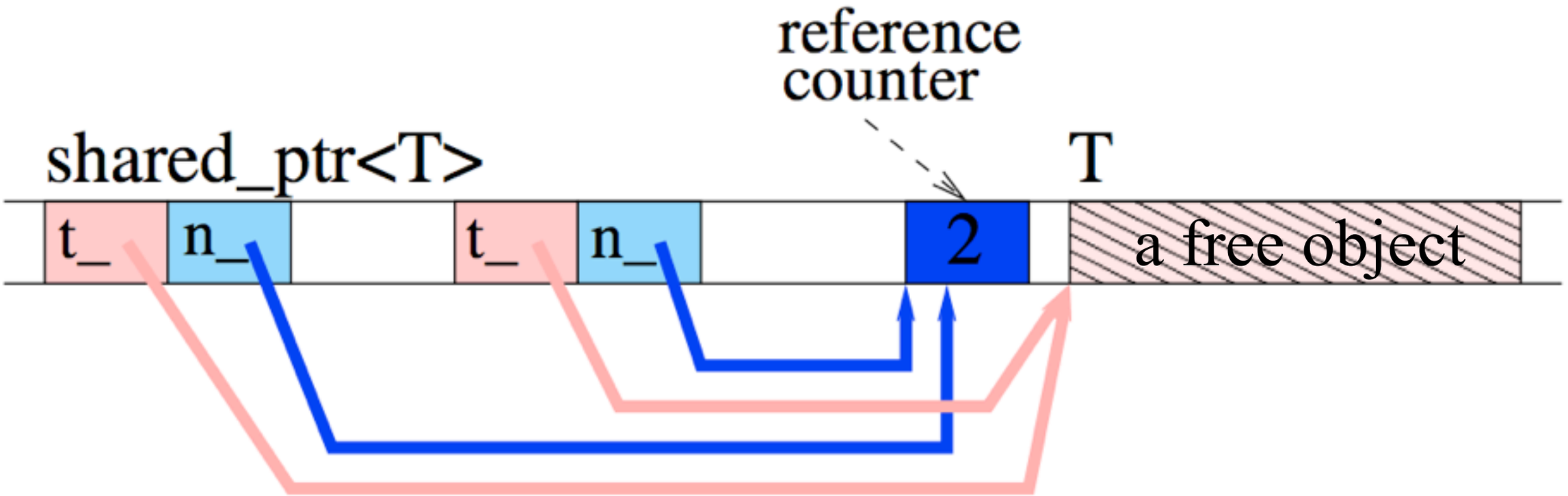
Templates

Reference-Counted Pointers Example

Reference-Counted Pointers

Reference Counting is a naive method of automatic memory management:

- a counter is associated with each free object which is encapsulated in a shared pointer
- each time a new pointer to such an object is created or destroyed, the counter is incremented/decremented
- if the counter reaches `0`, the object can be safely destroyed since no pointer references it
- all the interaction with the free object happens through the shared pointers



Reference-Counted Pointers (1)

```
#ifndef SHARED_PTR
#define SHARED_PTR
template<class T>
class shared_ptr {
public:
// constructor, turns a normal pointer into a shared one
   shared_ptr(T* t = 0) : t_(t), n_(new unsigned int(1)) { }
// copy constructor, should increment the reference count
   shared_ptr(const shared_ptr& r) : t_(r.t_), n_(r.n_) { ++*n_; }
// destructor, inline implementation follows later
   ~shared_ptr();
// conversion to bool, shared_ptr can be used in conditionals
   operator bool() const { return (t_ != 0); }
// make shared pointer act as if you handle the normal pointer
   T& operator*()  const { return *t_; }
   T* operator->() const { return t_; }
// assignment to another shared_ptr, do reference bookkeeping
   shared_ptr& operator=(const shared_ptr& r);
// ...
private:
   T* t_;              // the encapsulated pointer
   unsigned int* n_;   // reference count for *t_
};
```

Reference-Counted Pointers (2)

```
// assignment of a shared_ptr to another one
// bookkeeping needs to be done for shared_ptr1 = shared_ptr2:
//   - shared_ptr1 counter--, shared_ptr2 counter++
//   - delete shared_ptr1 value iff no more references to it
template<typename T>
inline shared_ptr<T>&
shared_ptr<T>::operator=(const shared_ptr& r) {
   if (this == &r) // check for self-assignment
      return *this;
   if (--*n_ == 0) {
      // target is last pointer to *t_: delete it
      // and the reference count
      delete t_;
      delete n_;
   }
   // actual copy of parameter to target
   t_ = r.t_;
   n_ = r.n_;
   ++*n_;
   return *this;
}
```

Reference-Counted Pointers (3)

```
// destructor is called whenever a reference to the shared_ptr
// is destroyed (e.g. by leaving a function scope)
template<typename T>
inline shared_ptr<T>&
shared_ptr<T>::~shared_ptr() {
// destructor decrements shared counter *n_
   if (--*n_) // there are remaining pointers to *t_
      return; // do not delete, return immediately
   delete t_;
   delete n_;
}
#endif
```

Reference-Counted Pointers (4)

```
#include <std::string>
#include "shared_ptr.h"
// You want to share references to Huge objects, but need to do bookkeeping
// of the number of references to it
class Huge {
public:
    // ctor is private; force use of a “factory method”
    static shared_ptr<Huge> create(const char *s) {
        return shared_ptr<Huge> (new Huge(s));
    }
    // ...
private:
    std::string* data_;
    Huge(const char* s) :
        data_(new std::string_(s)) { }
    ~Huge() { delete data_ }
    Huge& operator=(const Huge&); // forbidden
};

int main(int, char **) {
    shared_ptr<Huge> r = Huge::create("c++");
    // next line copies only reference and increments ref. count
    shared_ptr<Huge> p(r);
}
```

Reference-Counted Pointers: Pro/Contra

- (+) Automatic Memory Management like Java
  - Just create, don't worry about deletion
- (+) Low overhead compared to some garbage collection algorithms
- (-) Not for circular structures
  - However these do not often occur in practice

Meta Programs

Partial Template Specialization

```
1  template<int A, int B> // template parameters can be int
2  struct power {
3     enum { result = power<A, B - 1>::result * A };
4  };
5
6  template<int A>          // partial specialization
7  struct power<A, 0> {
8     enum { result = 1 };
9  };
10
11 std::cout << power<2,3>::result; // prints 8
12 // power<2,3>::result
13 // = power<2,2>::result * 2 by line 3
14 // = power<2,1>::result * 2 * 2 by line 3
15 // = power<2,0>::result *2 * 2 * 2 by line 3
16 // = 1 *2 * 2 * 2 = 8 by line 8
```

Metaprograms

- Compute result at compile time
- Compute with types (and (enum) constants) as values

```
template<typename parameter_type_1,
         typename parameter_type_2>
struct Function {
   typedef ... result_type;
   enum { result = ... };
};

... Function<T1, T2>::result ...
... Function<T1, T2>::result_type ...
```

- Use partial specialization to
  - test condition
  - stop recursion
- Computationally complete !

Meta `if ... then ... else`

```
#ifndef IFTHENELSE_H
#define IFTHENELSE_H
// IfThenElse<(1<3),int,double>::result_type => int

// default
template<bool Condition, typename T_true, typename T_false>
struct IfThenElse;

// if Condition = true
template<typename T_true, typename T_false>
struct IfThenElse<true, T_true, T_false> {
   typedef T_true result_type;
};

// if Condition = true
template<typename T_true, typename T_false>
struct IfThenElse<false, T_true, T_false> {
   typedef T_false result_type;
};

#endif
```

Meta Square Root Function

```
#ifndef SQRT_H
#define SQRT_H
// sqrt<N, Lo, Hi>: Lo <= sqrt(N) <= Hi

template<int N, int Lo = 1, int Hi = N>
struct Sqrt {
  enum { mid = (Lo + Hi) / 2 };
  enum { result =
    IfThenElse< (N < mid * mid),
                Sqrt<N, Lo, mid - 1> ,
                Sqrt<N, mid, Hi>
              >::result_type::result };
};

// sqrt<N,L,L>: sqrt(N) = L
template<int N, int L>
struct Sqrt<N, L, L> {
   enum { result = L };
};
#endif
```

Example computation by compiler

```
Sqrt<12>::result
Sqrt<12,1,12>::result
  IfThenElse<(12<6*6),
    Sqrt<12,1,5>,
    Sqrt<12,6,12> >::result_type::result
Sqrt<12,1,5>::result_type::result
  IfThenElse<(12<3*3),
    Sqrt<12,1,2>,
    Sqrt<12,3,5> >::result_type::result
Sqrt<12,3,5>::result_type::result
  IfThenElse<(12<4*4),
    Sqrt<12,3,3>,
    Sqrt<12,4,5> >::result_type::result
Sqrt<12,3,3>::result_type::result
3
```