Part 4: Class definition – inline, friend, operator, templates

Prof. Dr. Ulrik Schroeder

C++ - Einführung ins Programmieren

WS 03/04

http://learntech.rwth-aachen.de/lehre/ws04/c++/folien/index.html

What you know already ...

- separation of class defintion (*.h) & implementation (*.cpp)
- public, protected, private sections
- attributes are encapsulated, only methods can access private members
- Constructors
- object itself (this) is implicit first parameter of each method
- methods can be overloaded

- ... so what more is here
  - inline
  - friend
  - operator overloading
  - symmetric operators
  - conversion
  - generic classes

Destructors

- automatically called when validity of object ends
- has no arguments
- defined like default constructor beginning with ~

```
class X {
public:
    X( ) { /* reserve storage or open files */
        objCounter++; }
    ~X( ) { /* reverse actions of constructor */
        objCounter--;  }
private:
    static int objCounter;
} // class X
```

most important when ressources are dynamically allocated (new / delete or open / close)

```
int X::objCounter=0;
```

must be initialized outside as global variable!

Class attribute shared by all objects or even without!

access to private members (get .. set)

- as with Java get___ and set__ methods
- C convention: overloading method

```
class Linkable{
public:  // read
         inline string data( ) { return _data; }
         inline Linkable* next( ) { return _next; }
         // set
         inline void data( string d ) { _data = d; }
         inline void next( Linkable* n ) { _next = n; }
private:
         string _data;
         Linkable* _next;
}; // class Linkable
```

why define access methods?

hide implementation =>maintainable

controlled access (invariants, assertions, ...)

const objects

- declared as const object
- initialized with definition, must not be changed

  => compiler does not allow method calls

  ... except const methods

```
const Linkable x( "Unchangeable", NULL );
if ( x.next( ) != NULL ) ...
    x.next( x.next( ) );
```

o.k., iff declared as const

void next(Linkable*) not declared as const

```
inline Linkable* next( ) const { return _next; }
inline string data( ) const { return _data; }
```

Canonical Class definitions

- 4 methods are generated automatically (if absent)
  - Default constructor
  - Destructor
  - Copy constructor
  - assignment operator

```
X::X( )
X::~X( )
X::X( X& )
X& X::op=( X& )
```

only called when object is created

called by programmer

```
class X {
private: int dta;
public: X( ) {  };
        ~X( ) {  };
        X( X& x ) { dta = x.dta; };
        X& op=( X& x ) { dta=x.dta;
            return *this; }
} // class X
```

```
int main( ) {
    X  x1,
       x2( x1 ),
       x3 = x2
    x1 = x3;
    int i = f( x1 );
} // main( )
```

default cons

copy cons

copy cons

assignment

copy cons*

destructors

*if not declared as reference param

class Complex revisited

```
class Complex {
public: // Canonical class definition
    Complex( r=0, i=0 ) : real( r ), imag( i ) { }; // X::X( )
     // Complex( Complex& c ) { ... }; X::X( X& ) - autom.
    // ~Complex( ) { }; ~X( ) -- automatic
    Complex& operator =( const Complex& other ) {
        real = other.real; imag = other.imag;
        return *this;
    } // this = other - also automatic, just demo
    ...
private:
    double real, imag;
} // class Complex
```

conversion constructor

default constructor

copy constructor

assignment

calls assignment operators of the classes of the attributes
Here: assignment of float

Using canonical classes

- Declaration, definition, assignment, parameter passing

```
int main( ) {

    Complex  c0,
             c1( 1 ),
             c2( 2, -2 ),
             c3( c2 ),
             c[ 4 ];

    for ( int i = 0; i < 4; i++ )

        c[ i ] = c2;

    Complex* cp = &c0, *cp1;

    cp1 = new Complex( 0, 4 );

    f( c2, c3 );

} // main
```

Default constructor

Conversion constructor

Copy constructor

Default constructor (* 4)

assignment

conversion cons

copy, if not reference param

Destructors

Operator overloading

- arithmetic, boolean, ... operators

```
class Complex { ...
public: // Canonical class definition
    // comparison
    bool operator ==( const Complex& other ) const {
        return real == other.real && imag == other.imag;
    }; // c == other
    bool operator < ( const Complex& other ) const {
        return double(*this) < ( double ) other;
    }; // c  < other
    // !=, <=, >, >= defined per #include <utility>
```

define == with two arguments

method does not change object

conversion to double

Operator overloading(2)

- arithmetic, boolean, ... operators

```
class Complex { ...
public: // Conversion
    operator double( ) const {
        return sqrt( real*real + imag*imag );
    }; // double( ) -- conversion
```

no type!!

no argument!

?

```
int main( ) { ...
    if ( c1 < c2 )
        cout << ( double )c1 << " < " << double( c2 ) << endl;
} // main
```

Operator overloading(3)

- Arithmetic operations:
  - unary: -c, ++c, --c, c'

```
class Complex { ...
public: // Arithmetic
    Complex&    conjunct( ) const;   // c'
    Complex& operator  -( ) const {
        Complex* result = new Complex( -real, -imag );
        return *result;
    } // -c
    Complex& operator ++( ){ real += 1.0; return *this; };// ++c
    Complex& operator --( ){ real -= 1.0; return *this; };// --c
```

```
Complex& operator ++( int i=1 ) {
    Complex* res=new Complex(*this);
    real+=1;
    return *res;
}; // c++
```

why const?

why *result?

why ++c and not c++?

why not const?

how does inline implementation look like?

Operator overloading(4)

```
class Complex { ...
public: // Arithmetic
    Complex& operator +( const Complex& other ) const; // c + other
    Complex& operator -( const Complex& other ) const; // c - other
    Complex& operator *( const Complex& other ) const; // c * other
    Complex& operator /( const Complex& other ) const; // c / other

    Complex& operator +=( const Complex& other ); // c += other
    Complex& operator *=( const Complex& other ); // c *= other
    Complex& operator -=( const Complex& other ); // c -= other
    Complex& operator /=( const Complex& other ); // c /= other
```

Operator overloading(5)

complex.cpp

```
Complex& Complex::operator +( const Complex& other ) const {
    Complex* result   = new Complex( real+other.real, imag+other.imag );
    return *result;
} // c + other
```

binary arithmetic operations combine two objects to create a new one

```
Complex& Complex::operator +=( const Complex& other ) {
    real += other.real;  imag += other.imag;
    return *this;
} // c += other
```

assignment operations change the object itself

at least += can also be defined inline (or both)

Global operators

- if there is conversion, then operators should be symmetric:

```
int main( ) { ...
    c1 = c2 + c3;
    c1 = c2 + 4.3;   // automatic: Complex + double->Complex
    c1 = 4.3 + c2;
} // main
```

actually: only iff double( ) is not defined!!

first operand of + must be Complex!!

```
c1 = ( Complex )4.3 + c2;
c1 = 4.3 + ( double ) c2;
```

with both conversions defined:

ambigious overload: Complex& + double

could either be:

Complex + double->Complex

or

Complex->double + double

Global operators (2)

friend functions can access private members in classes

```
...
    // global operators (friend)
    friend Complex& operator +( const Complex&, const Complex& );

} // class Complex

inline Complex& operator  +( const Complex& c1, const Complex& c2 ) {
    Complex* result=new Complex( c1.real + c2.real, c1.imag + c2.imag );
    return *result;
}; // c + other
...
```

private member in Complex

IO streams for user defined classes

- define stream operators >> and << for Complex: friend

```
    // IO
    friend ostream& operator << ( ostream&, const Complex& );
    friend istream& operator >> ( istream&, Complex& );
} // class Complex
```

```
istream& operator >>( istream& in, Complex& c ) {
    double r, i;
    in >> r;  in >> i;
    c.real=r; c.imag=i;
    return in;
} // set value interactively
```

```
cout << "Enter Complex as real+imaginary: ";
cin >> c1;
```

```
Enter Complex as real+imaginary: 4+3i
```

reads **4** stops at **+** reads next float **3** stops at **i**

IO streams for user defined classes

```
ostream& operator <<( ostream& o, const Complex& c ) {
    if      ( c.imag==0 ) o << c.real;          // as real number
    else if ( c.real==0 )  o << c.imag << 'i';  // pure imaginary
    else {                                      // (a+bi) or (a-bi)
        o << "(" << c.real;
        if ( c.imag >= 0 )
            o << '+';
        o << c.imag << "i)";
    }
    return o;
} // output <<
```

Canonical Class definitions - dynamic

- 4 methods are generated automatically (if absent)
  - Default constructor
  - Destructor
  - Copy constructor
  - assignment operator

```
X::X( )
X::~X( )
X::X( X& )
X& X::op=( X& )
```

only called when object is created

called by programmer

```
class X {
private: char* dta;
public: X( ) {  };
        ~X( ) {  };
        X( X& x ) { dta = x.dta; };
        X& op=( X& x ) { dta=x.dta;
            return *this; }
} // class X
```

?strcpy(dta, x.dta)

```
int main( ) {
    X  x1,
       x2( x1 ),
       x3 = x2
    x1 = x3;
    i = f( x1 );
} // main( )
```

default cons

copy cons

copy cons

assignment

copy cons*

destructor

*if not declared as reference param

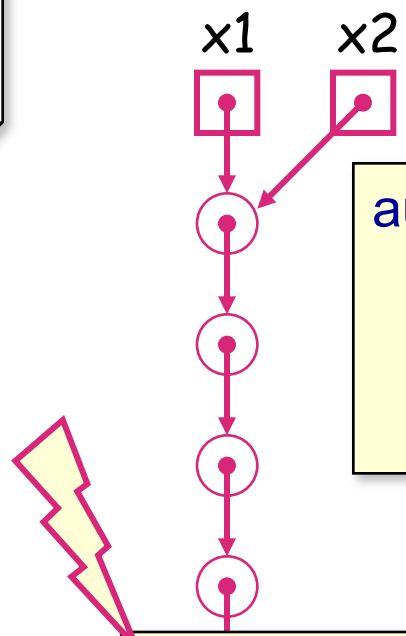
Canonical Class definitions - dynamic

- 4 methods are generated automatically, but you want different semantics, if dynamic structures are involved

```
class X {
public:     ...
private:  Linkable* dta;
} // class X
```

witout user defined X::X(X&) and X&=X&

```
int main( ) {
    X x1, x2;
    x1.fill( ... );
    x2 = x1;
} // main( )
```



automatic semantics:

all elements are assigned to the corresponding of the other object

changes in x2 effect x1 and vice versa!!

Canonical Class definitions - dynamic

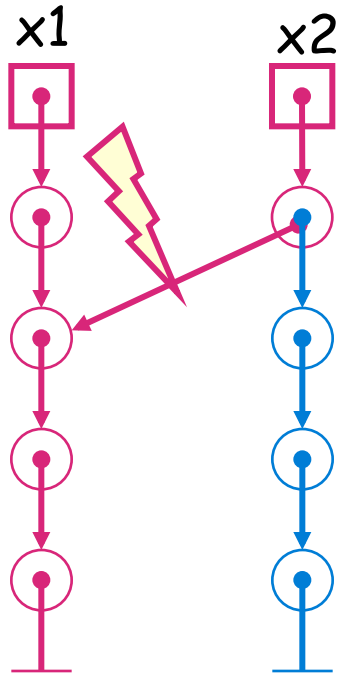
- Copy Constructor and operator= w/out element sharing

```
class X { private: Linkable* dta;

public: X( ) { … };

    X( const X& x ) { if ( this !=&x ) {
        if ( dta ) delete dta ;
        dta = new Linkable( x.dta );  }
    } // X::X(X&)

    ~X( ) { if(dta) delete dta; }

    X& operator=( const X& x ) {
        if ( this != &x ) {
            if ( dta ) delete( dta );
            dta = new Linkable( x.dta );
        }
        return *this;
    } // X::X(X&)
} // class X
```

here we have the first problem

delete for associated new

2. problem ! ?

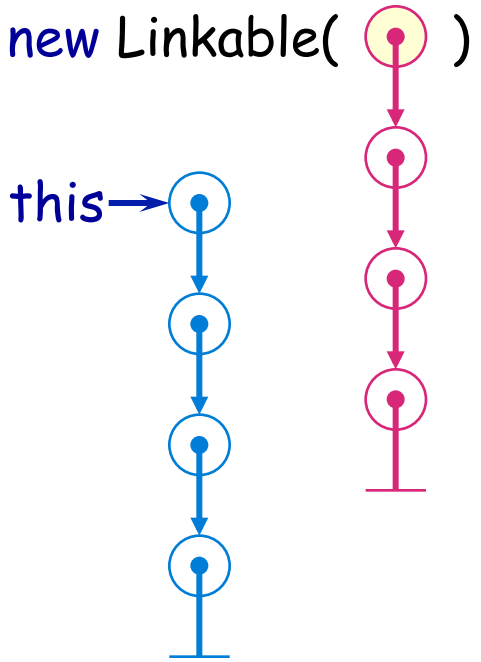


deep_copy must be provided by Linkable

Canonical Class definitions - dynamic

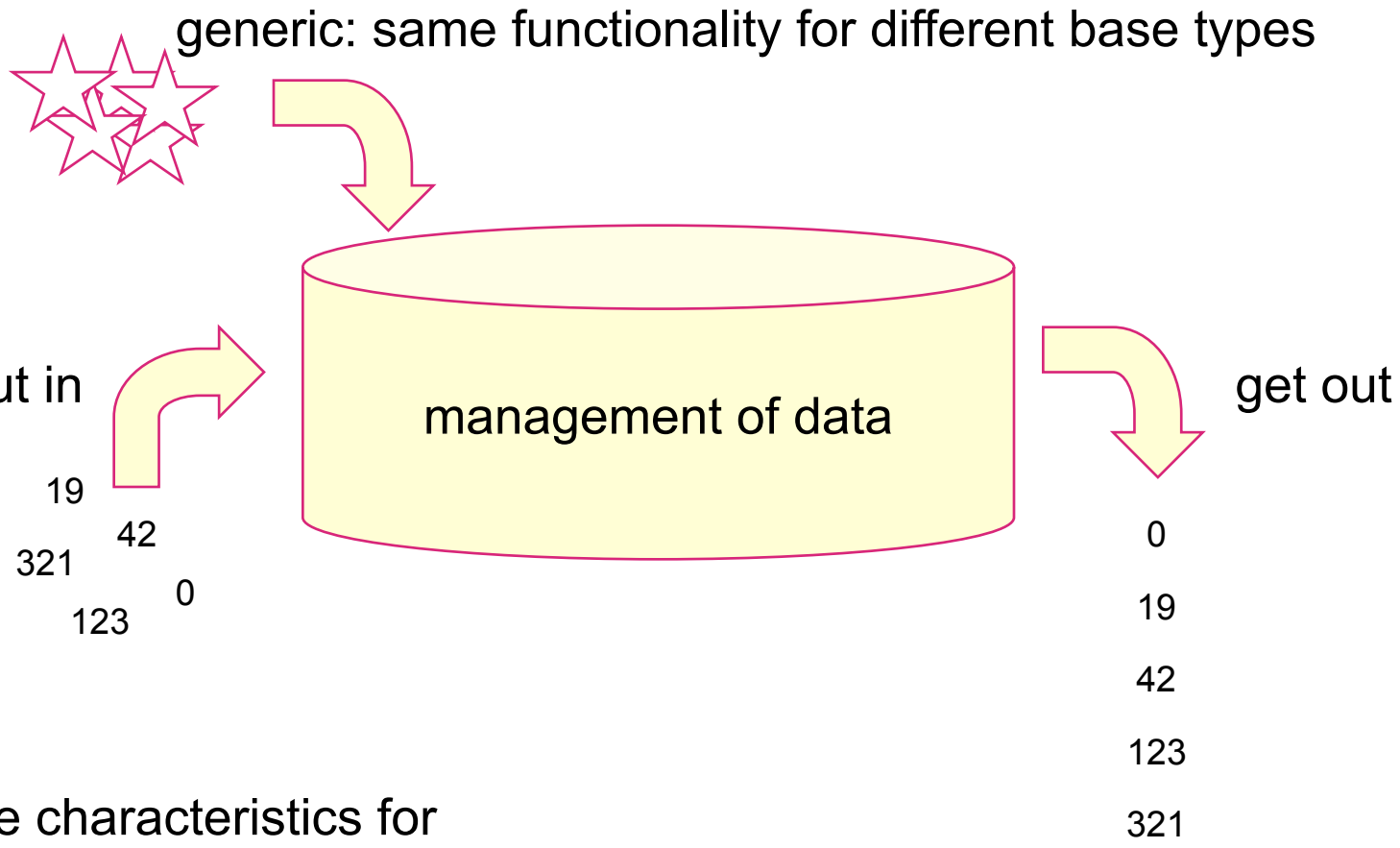
- Copy Constructor and operator= w/out element sharing

```
class Linkable {

private: Linkable* _next; int val;

public: X( ) { … }; …

    Linkable( const Linkable& x ) {
        val = x.val;   // int can't be shared
        if ( x.next() )
            _next = new Linkable( *(x.next()) );

        else
            _next = NULL;
    } // X::X(X&)

    ~Linkable( ) { if(_next) delete _next; }


} // class X
```



automatically calls ~ of _next

Generic Container Classes



diverse characteristics for

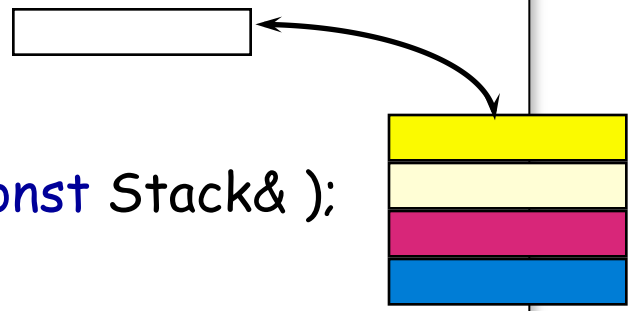
constraints (insertion (cost), access, iteration, deletion, …)

special functionality (sort, search, grow, shrink, …)

example: class Stack

```
class Stack {
private:
      Any  impl[ 250 ];  int pos;
public:
      Stack( ) : pos( -1 ) { } // X::X( ) ... other canonical defs
      void push( Any  x ) { if ( ! isFull( ) ) impl[ pos++ ] = x; } // push
      void pop( ) {  if( ! isEmpty() ) pos--; } // pop
      Any  top( ) { if ( ! isEmpty( ) ) return impl[ pos]; }
      boolean isEmpty( ) { return pos < 0; }
      boolean isFull( ) { return pos >= 249; }
      friend ostream& operator <<( ostream&, const Stack& );
} // class IntStack
inline ostream& operator <<( ostream& o, const Stack& s ) { ...
```

same properties for different types



“Generic Datatypes” in Java

- all object types are conformant with class `Object`

Java root class

```
public class Stack {
        private Object impl[ ] = new Object[ 250 ];
        private int pos = 0;
        public void push( Object o ) {
                if ( isFull( ) )
                        throw new StackIsFullException( “250” );
                impl[ pos++ ] = j; // or copy ??
        } // push( )
        public void pop() {
                if( isEmpty() )
                        throw new StackIsEmptyException( );
                pos--;
        } // pop( )
        public Object top( ) { if (!isEmpty( ) ) return impl[ pos]; }
        boolean isEmpty( ) { return pos == 0; }
        public boolean isFull( ) { return pos == 250; }
} // class Stack
```

client must reconvert all elements back to original

compiler can not check for consistency

Generalization: Stack<T>

- C++: class templates

```
template< class T > class Stack {
public: . . .
    void push( const T& el ) {
        T* copy=new T( el );
        if(!isFull()) impl[++pos]=*copy;
    } // push( )
    T& top( ) { return _impl[ pos ]; } ...
    friend ostream& operator <<<T>( ostr ..
private:
    T _impl[ MAX ];
} // class Stack< T >
template<class T> ostream& operator <<( ...
```

```
Stack( Stack<T>& other ) { ...
```

declares T as a type parameter for class Stack

Typename used paramterized

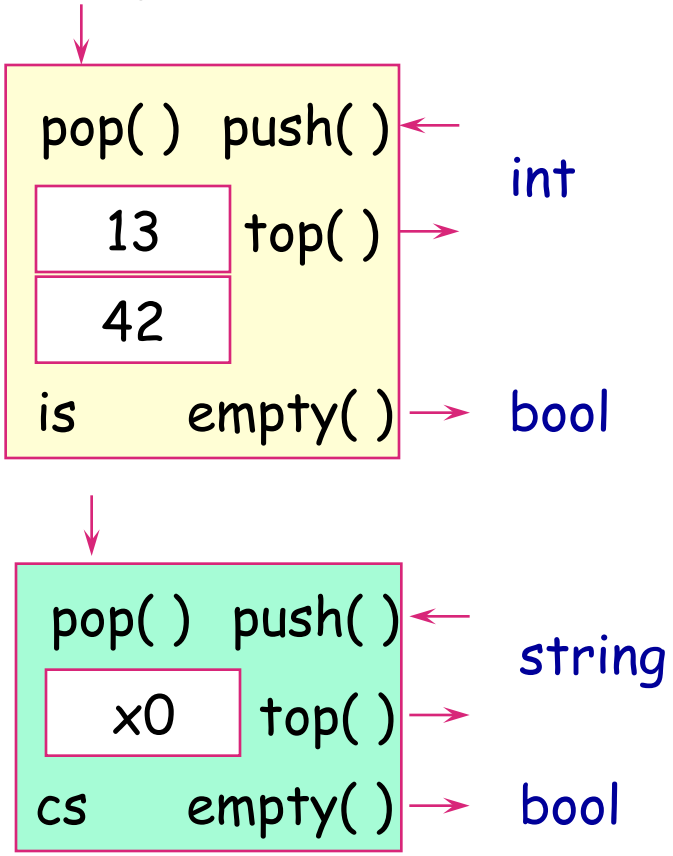
making the connection to instantiate with T

non-inline methods are implemented in the header!!!

Instantiation of template classes

- Stack can be used for different base types

```
typedef Stack<int> iStack;
int main( ) {
        Stack<string> cs;
        iStack is;
        is.push( 42 ); is.push( 13 );
        cs.push( "x0" );
        ...
}
```



Compiler checks validity of each call with every instantiation (compile time)

Code for Stack will only be produced with an instantiation => two classes !!

Why implementing inside the header

- Preprocessor replaces each T with argument of instantiation
- … and then compiles the class => must be provided as source (not object code)

Trick to develop in seperate files

```
template< class T > class Stack
public: . . .
} // class Stack< T >

 #ifndef __STACK_CXX__
 #include "stack.cxx"
 #endif


#endif // __STACK_H__
```

```
#ifndef __STACK_CXX__
#define __STACK_CXX__

#ifndef __STACK_H__
#error Include stack.h first!!!
#endif

template<class T> void Stack<T>::sth( … ) {
    …
}

#endif
```

Generic global functions

- Type parameters can be used on functions
  => function templates

```
template < class T > void swap ( T& x, T& y ) {
    T help( x ); x = y; y = help;
} // swap( ) - exchange two elements
```

can be used to exchange values of any type with copy constructor and assignment, e.g. float values in an array:

```
float a[ 234 ] = ...
...
    if ( a[ i ] < a[ j ] )
        swap( a[ i ], a[ j ] );
...
```

what about this generic function?

```
template < class T > void min ( T& x, T& y ) {
    return ( x < y ) ? x : y;
} // min
```

Template classes & functions

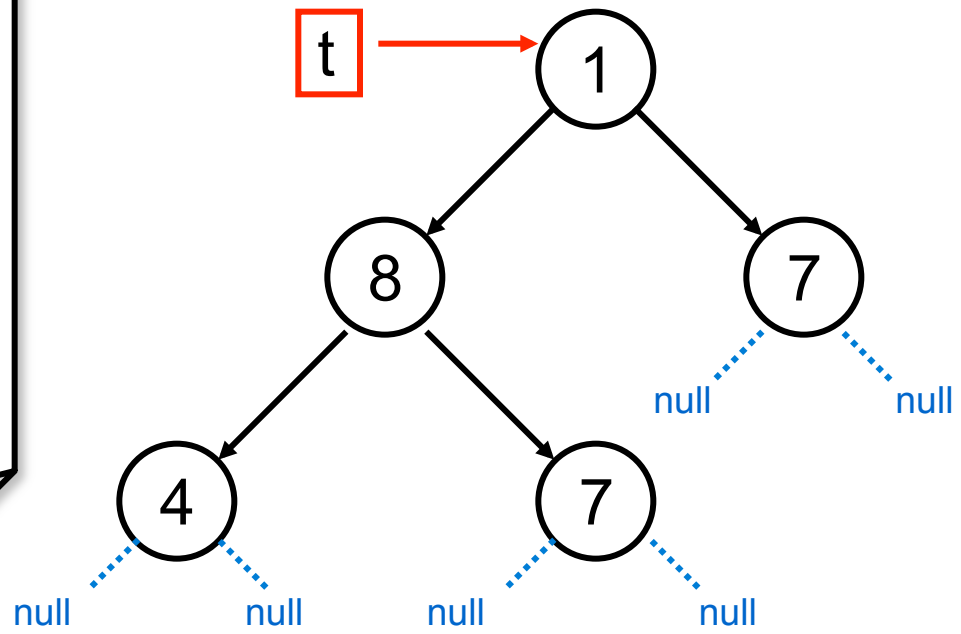
- do not reduce code, if you instantiate many different classes
  - for each instantiation, the compiler produces specific code
- Advantages
  - coding & testing is done once => errors are less probable
  - unary solution for similar problems
- There are also templates with more than one parameter
  - … we skip this part

Recursive methods in recursive structures

- count number of nodes in a binary tree
- sum up the content

} average

```
template<class T> Tree {
public:
        Tree ( Elem e ) { item = e; }
        ...
protected:
        T item;
        Tree *left, *right;
} // Tree
```




```
int size( ) {
        int res = 1; // count oneself
        if ( left )   res += left .size ( );
        if ( right ) res += right. size ( );
        return res;
} // size( )
```

accumulation in a tree

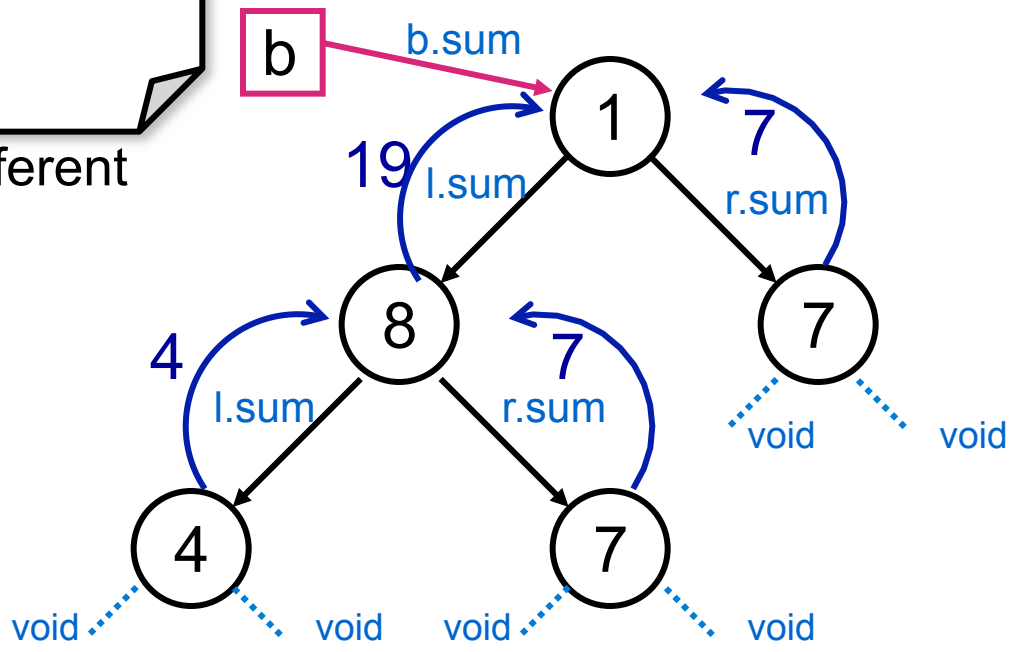
```
class    Tree { ...
    int sum( ) {
        int result = item;
        if ( left )       result += left.sum;
        if ( right )      result += right.sum;
        return result;
    } // sum . . .
```

these are different objects!!

Active Objects:

b.sum

- 1 + l.sum
  - 8 + l.sum
    - 4
    - 12 + r.sum
      - 7
  - 19 + r.sum
    - 7



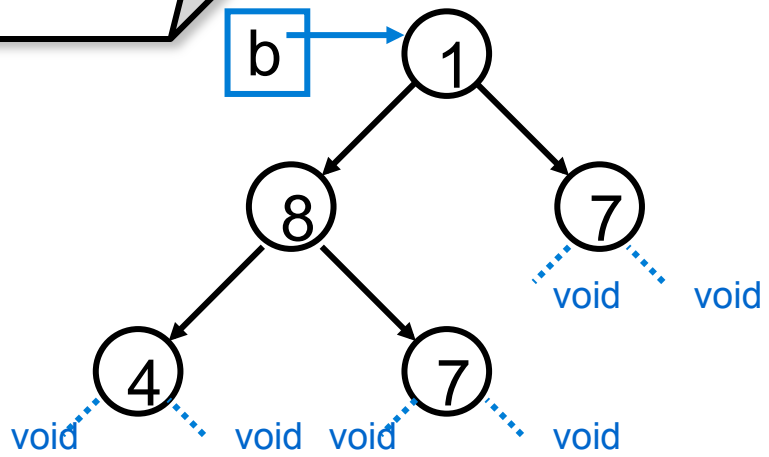
Recursive methods

Task: Print a tree as indented structure

```
void pr( int indent ) {
    // indention as leading dots:
    for ( i = indent; i > 0;  i-- ) cout<<". ";
    cout<<value;
    if ( left )  left.pr( indent + 4 );
    if ( right )    right.pr( indent + 4 );
} // pr
```

this is "Preorder". how to print "Inorder" or "Postorder" ?

```
1
. . . . 8
. . . . . . . . 4
. . . . . . . . 7
. . . . 7
```



Class definitions

- Canonical classes
  - X( ), ~X( ), X(X&), X&=X&
  - copy and assign redefinition for dynamic structures
- operator overloading, inline
  - esp. == and !=, some: <, ... (utility)
  - artithmetic: +, +=, ...
- friend
  - operators of other classes: ostream << and istream >>
  - global operators for Symmetry
- static members
- class & function templates