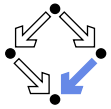
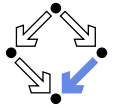
Templates

Wolfgang Schreiner
Wolfgang.Schreiner@risc.jku.at

Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria
http://www.risc.jku.at



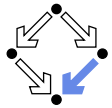
Templates



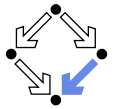
In C++, types can also serve as parameters.

- Parametric polymorphism: genericity based on types as parameters.
  - An extension to the object-oriented paradigm of type polymorphism (genericity based on inheritance).
- Templates: functions or classes parameterized over types.
  - By instantiating the template parameters with concrete types, concrete functions and classes are constructed.
  - Instantiation is resolved at compile time (no runtime overhead).
- Template metaprogramming: let compiler execute programs.
  - Templates allow recursion, selection, and computation.
    - Template programs are executed at compile-time.
  - Selection of different variants of code, modification of the behavior of generated code, set policies for runtime code execution.

The C++ standard library makes heavy use of templates.



Function Templates

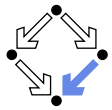


```
template<typename T> T min(T a, T b) {
  T c = a;
  if (b < a) c = b;
  return c;
}

int x = min<int>(10, 20);         // explicit instantiation of min<int>
float y = min<float>(10.4, 20.3); // explicit instantiation of min<float>
int z = min(10, 20);              // automatic instantiation of min<int>
```

- `template<typename` *T*`>`: a template with type parameter *T*.
  - Also `class` *T* denotes type parameters.
    Typically used, if *T* shall denote a class.
- *fun*`<`*A*`>(...)`: instantiation of parameter *T* by concrete type *A*.
  - Instantiated function declaration is generated and compiled.
  - Type checking occurrs for each instantiation separately.
- *fun*`(...)`: automatically deduced instantiation.
  - In most cases, the compiler is able to automatically deduce the appropriate instantiation.

Each instantiation of a function template denotes a separate function.

Operator Templates

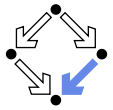


Like functions, also operators may be defined as templates.

```
template<typename T> int operator+(T x, int y)
{
  return x.getIntValue() + y;
}

class Int {
  int x;
  public:
    Int(int x) { this->x = x; }
    int getIntValue() { return x; }
};

Int i(5);
int j = i+1;                    // automatic: operator+<Int>
int k = operator+<Int>(i, 1); // explicit instantiation
```

Also for operators, argument types may be automatically derived.

Template Source



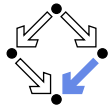
Where to put the source of a template?

```
// min.h
template<typename T> T min(T a, T b) { ... }

// file1.cpp                    // file2.cpp
#include "min.h"                #include "min.h"
... min<int>(...) ...           ... min<int>(...) ...
```

- Short answer: place it in a header file.
  - Each source file using the template includes the file.
  - Each object file contains set of generated template instantiations.
  - Linker merges duplicate instantiations in finally produced executable.
- Disadvantage: compilation overhead.
  - Same instantiation repeatedly generated, type-checked, and compiled.
  - Long compilation times, large object files.

A more comprehensive answer will be given later.

Example: A Generic Sorting Function (V1)

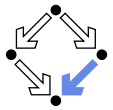


```
template<typename T> void sort(T a[], int n) {
  for (int i=0; i<n-1; i++) {
    for (int j=n-1; i<j; j--) {
      if (a[j] < a[j-1]) {
        T b = a[j];
        a[j] = a[j-1];
        a[j-1] = b;
      }
    }
  }
}

int a[100];
sort(a, 100); // automatic: sort<int>
```

- Function sorts arrays of any base type.
  - Type must provide comparison operator <.
- What if operator < is not appropriate?
  - We might wish another sorting criterium.

We have "hard-wired" the comparison operation in the sorter.

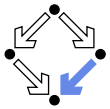
Example: A Generic Sorting Function (V2)



```
class IntComparator { public:
  static bool isSorted(int a, int b) { return a <= b ; }
};
class StringComparator { public:
  static bool isSorted(char *a, char *b) { return strcmp(a, b) <= 0; }
};

template<typename T, typename C> void sort(T a[], int n) {
  for (int i=0; i<n-1; i++) {
    for (int j=n-1; i<j; j--) {
      if (!C::isSorted(a[j-1],a[j])) {
        T b = a[j]; a[j] = a[j-1]; a[j-1] = b;
      }
    }
  }
}

int a[100]; sort<int, IntComparator>(a, 100);
char* b[100]; sort<char*, StringComparator>(b, 100);
```

The sorter is instantiated with a class holding the comparison operation.

Value Parameters



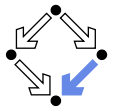
Template parameters may be also values.

```
template<int N, typename T> void fill(T a[N], T x)
{
  for (int i=0; i<N; i++) a[i] = x;
}

int a[3];
fill<3,int>(a, 7); // explicit instantiation
fill<3>(a,7);      // automatic: fill<3,int>
```

- `template<`*T v*`>`: a template with value parameter *v* of type *T*.
  - *T* must be an integral, enumeration, or pointer/reference type.
- *fun*`<`*a*`>(...)`:
  - Argument *a* must be a compile-time constant (a constant expression or the address of an object with external linkage).
  - Must be given by explicit instantiation (subsequent type parameters may be automatically instantiated).

Value parameters in templates may be also used in type declarations.

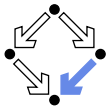
Example: A Generic Sorting Function (V3)



```
template<int N, typename T> void sort(T a[N]) {
  for (int i=0; i<N-1; i++) {
    for (int j=N-1; i<j; j--) {
      if (a[j] < a[j-1]) {
        T b = a[j];
        a[j] = a[j-1];
        a[j-1] = b;
      }
    }
  }
}

int a[100];
sort<100>(a); // automatic: sort<100, int>
```

The sorter is instantiated with the array length.

Example: A Generic Sorting Function (V4)

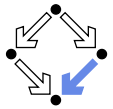


```
template<typename T, bool isSorted(T, T)> void sort(T a[], int n) {
  for (int i=0; i<n-1; i++) {
    for (int j=n-1; i<j; j--) {
      if (!isSorted(a[j-1], a[j])){
        T b = a[j];
        a[j] = a[j-1];
        a[j-1] = b;
      }
    }
  }
}

inline bool lessEqual(int x, int y) { return x <= y; }
int a[100];
sort<int, lessEqual>(a, 100);
```

- Comparison function becomes value parameter of template.
  - Function has generic signature `bool isSorted(T, T)`.
  - Type parameter *T* must appear in template signature before.
  - If defined in current file, comparison function may be inlined.

The sorter is instantiated with the comparison operation.

Specializing Function Templates

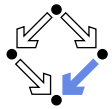


For special instantiations, alternative function definitions may be given.

```
double pow(double x, double y); // exponentiation
int isqrt(int x);               // integer square root

template<typename T> T root(T x, T y) {
  return T(pow(x, 1.0/y));      // conversion to T
}

template<> int root<int>(int x, int y) {
  if (y == 2)
    return isqrt(x);            // special implementation
  else
    return int(pow(x, 1.0/y));
}

int    r1 = root(4, 2);     // root<int> -> isqrt(4)
double r2 = root(4.5, 2.0); // root<double> -> pow(4.5, 0.5)
```

For achieving higher performance, a generic function implementation may be augmented by special implementations.

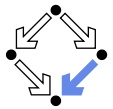
Example: A Generic Sorting Function (V5)



```
// a generic sorting function (V1)
template<typename T> void sort(T a[], int n) { ... }

// provide sorting function applicable to strings
template<> void sort<char*>(char* a[], int n) {
  for (int i=0; i<n-1; i++) {
    for (int j=n-1; i<j; j--) {
      if (strcmp(a[j], a[j-1]) < 0) {
        T b = a[j];
        a[j] = a[j-1];
        a[j-1] = b;
      }
    }
  }
}

char* a[100];
sort(a, 100); // automatic: sort<char*>
```

By specialization, a sorting function with another comparison operation (or another specialized sorting algorithm) may be used.

Overloading Function Templates

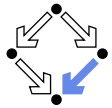


```
template<typename T> void func(T x, T y) { ... }
template<typename T> T func(T x) { ... }
template<typename T, typename U> U func(T x) { ... }

func(y, y);                    // automatic: func<float>(y, y)
int x = func<int>(1);          // explicit instantiation needed
float y = func<int, float>(x); // explicit instantiation needed
```

- Function templates may be overloaded.
  - Same function name but different template/function parameters.
  - If appropriate instantiation can be derived from the types of the function arguments, no explicit instantiation is needed.
  - If different function templates have same names and function parameters, explicit instantiation is needed.

Explicit instantiation may be required in case of disambiguities.

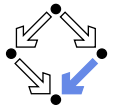
Changing Parameter Types



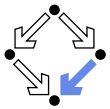
To change some parameter types, overload the function template.

```
double pow(double x, double y);  // exponentiation
double iroot(double x, int y);   // root computation

template<typename T> T root(T x, T y) {
  return T(pow(x, 1.0/y));      // conversion to T
}

template<typename T, typename U> T root(T x, U y) {
  return T(iroot(x, y));        // conversion to T
}

double r2 = root(4.5, 2.0); // root<double> -> pow(4.5, 0.5)
double r2 = root(4.5, 2);   // root<double,int> -> iroot(4.5, 2)
```

Specialization and overloading gives large flexibility in selecting the most efficient implementations for arbitrary combinations of types.



Class Templates

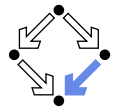


```
template<typename T> class Point {
  T x; T y;
public:
  Point(T a, T b): x(a), y(b) { }
  T getX() { return x; }
  Point flip() { return Point(-x,-y); }
  static Point copy(Point p) { return Point(p.x, p.y); }
};

Point<int> p(5, 3);
int x = p.getX();
Point<int> q = p.flip();
Point<int> r = Point<int>::copy(q);
```

- `template<typename T>`: also possible for declaration of class C.
  - Within the declaration, C is a synonym for instantiation $C<T>$.
  - Instantiation by any other type A must be explicitly stated as $C<A>$.
- $C<A>$: instantiation of parameter T by concrete type A.
  - Outside declaration, class template must be explicitly instantiated.

Each instantiation $C<A>$ represents a separate class.

Class Templates



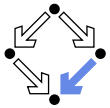
```
// Point.h
template<typename T> class Point { ... } // no member definitions inside

// member definitions outside of class declaration
template<typename T> Point<T>::Point(T a, T b): x(a), y(b) { }
template<typename T> T Point<T>::getX() { return x; }
template<typename T> Point<T> Point<T>::flip() { return Point(-x,-y); }
template<typename T> Point<T> Point<T>::copy(Point p)
{ return Point(p.x, p.y); }
```

- Member definition $C<T>::member$ outside class declaration.
  - Inside definition, C is a synonym for $C<T>$.

    Unfortunately not for types of data members and return types of member functions.

Member functions of class templates are defined in the template header.

Default Arguments

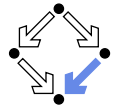


A class template may provide default arguments for its parameters.

```
template<typename T=int, typename U=int>
class Tuple {
  T x; U y;
public:
  Tuple(T a, U b): x(a), y(b) { }
  Tuple() { }
  Tuple(Tuple t): x(t.x), y(t.y) { }
};

Tuple<double, double> t(3.14, 2.71);
Tuple<double> u(3.14, 2);              // Tuple<double, int>
Tuple<> v(3, 2);                       // Tuple<int, int>
```

If the instantiation of a class template provides too few arguments, the default arguments are used instead.

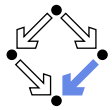
Value Parameters



```
template<int N=100, typename T=int>
class Array {
protected:
  T a[N];
public:
  Array(T x = 0) { for (int i=0; i<N; i++) a[i] = x; }
  virtual ~Array() { } // support subclasses with virtual functions
  int length() { return N; }
  T& operator[](int i) {
    if (i < 0 || i >= N) exit(-1);
    return a[i];
  }
};

Array<50> a(-1);   // Array<50,int>
a[1] = a[1] + 1;
Array<> b(-1);     // Array<100,int>
b[1] = b[1]+a[1];
```

A class template may also have value parameters which may also receive default values; concrete arguments must be compile-time constants.

Specializing Class Templates

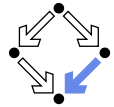


For special instantiations, alternative class definitions may be given.

```
template<typename T> class Container {
  T* value;
public:
  Container(T x) { value = new T; *value = x; }
};

template<> class Container<int> {
  int value;
public:
  Container(int x) { value = x; }
};

Tuple<int, int> t(3, 0);
Container<Tuple<int, int> > c(t);
Container<int> d(5);
```

A generic class may be augmented by special representations.

Partial Specialization

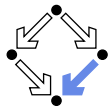


Only some the template parameters may be specialized.

```
template<int N, typename T> class Array { ... }

template<typename T> class Array<1, T> {
protected:
  T a; // no array but scalar variable
public:
  Array(T x = 0): a(x) { }
  virtual ~Array() { }
  int length() { return 1; }
  T& operator[](int i) {
    if (i != 0) exit(-1);
    return a;
  }
};

Array<1, int> a(1);
a[0] = a[0]+1;
```

From a template, more special templates may be derived.

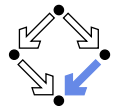
Example: A Generic Sorting Function (V6)



```
template<typename T> class Comparator { public:
  static bool isSorted(T a, T b) { return a <= b ; }
};
template<> class Comparator<char*> { public:
  static bool isSorted(char *a, char *b) { return strcmp(a, b) <= 0; }
};

template<int N, typename T> void sort(Array<N, T>& a) {
  for (int i=0; i<N-1; i++) {
    for (int j=N-1; i<j; j--) {
      if (!Comparator<T>::isSorted(a[j-1],a[j])) {
        T b = a[j]; a[j] = a[j-1]; a[j-1] = b;
      }
    }
  }
}

Array<100, int> a; sort(a);
Array<100, char*> b; sort(b);
```

Generic sorting with the comparison operation attached to the base type.

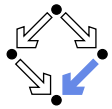
Example: A Generic Sorting Function (V7)



```
template<int N, typename T> class SortableArray: public Array<N, T> { public:
  static bool isSorted(T a, T b) { return a <= b ; }
};
template<int N> class SortableArray<N, char *>: public Array<N,char*> { public:
  static bool isSorted(char *a, char *b) { return strcmp(a, b) <= 0; }
};

template<int N, typename T> void sort(SortableArray<N, T>& a) {
  for (int i=0; i<N-1; i++) {
    for (int j=N-1; i<j; j--) {
      if (!SortableArray<N, T>::isSorted(a[j-1],a[j])) {
        T b = a[j]; a[j] = a[j-1]; a[j-1] = b;
      }
    }
  }
}

Array<100, int> a; sort(a);
Array<100, char*> b; sort(b);
```

Generic sorting with the comparison operation attached to the array type.

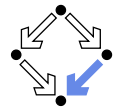
Example: A Generic Sorting Function (V8)



```
template<int N, typename T> class SortableArray2: public Array<N, T> { public:
  virtual bool isSorted(int i, int j) { return this->a[i] <= this->a[j]; }
};
template<int N> class SortableArray2<N, char *>: public Array<N, char*> { public:
  virtual bool isSorted(int i, int j) {return strcmp(this->a[i],this->a[j])<=0;}
};

template<int N, typename T> void sort(SortableArray2<N, T>& a) {
  for (int i=0; i<N-1; i++) {
    for (int j=N-1; i<j; j--) {
      if (!a.isSorted(j-1,j)) {
        T b = a[j]; a[j] = a[j-1]; a[j-1] = b;
      }
    }
  }
}

SortableArray2<100, int> a; sort(a);
SortableArray2<100, char*> b; sort(b);
```

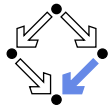
Generic sorting with the comparison operation attached to the array object (i.e. operation is looked up at runtime, no inlining is possible).

Example: A Generic Sorting Function (V8)

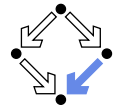


```
template<int N, typename T> class SortableArray2: public Array<N, T> { ... }
template<int N, typename T> void sort(SortableArray2<N, T> a) { ... }

template<int N, typename T>
class ReverseSortedArray: public SortableArray2<N, T> {
public:
  virtual bool isSorted(int i, int j)
  { return !SortableArray2::isSorted(i, j); }
};

ReverseSortedArray<100, int> a; sort(a);
ReverseSortedArray<100, char*> b; sort(b);
```

By inheritance and overriding, arrays may be constructed with different comparison operations attached to them.



Member Templates

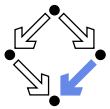


A template may be a member of a class.

```
template<typename T> class Variable {
  T value;
public:
  Variable(T x): value(x) { }
  void set(T x) { value = x; }
  template<typename U> void set(U x) { value = static_cast<T>(x); }
};

Variable<int> var(0);
var.set(1);           // non-template function set
var.set<double>(1.5); // explicit: template function set<double>
var.set(2.5);         // automatic: template function set<double>
```

By overloading, functions and function templates may have same names; compiler gives precedence to non-template functions.

Templates as Parameters



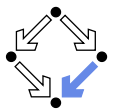
A template may receive class templates as parameters.

```
template<template<int, typename> class C, int N, typename T>
void fill(C<N, T> a, int start, int end, T x) {
  for (int i=start; i<end; i++) a[i] = x;
}

Array<100, int> a;
fill(a, 0, 50, 1);   // automatic: fill<Array, 100, int>
fill(a, 50, 100, 2); // automatic: fill<Array, 100, int>

Array<100, double> b;
fill(b, 0, 100, 1.5); // automatic: fill<Array, 100, double>
```

- `template<template<...> class C>`
  - A template with a class template C as parameter.
  - C must be instantiated as specified by its parameter list <...>.

For function templates, the appropriate instantiations of the template parameters can be typically determined automatically.

Templates and Friends

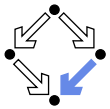


```
class C { };
template<typename T> class U { friend class C; };
class D { friend class U<int>; };
class E { template<typename T> friend class U; };
template<typename T> class V { friend class U<T>; };
template<typename T> class W { template<typename T2> friend class U<T2>; };
```

- Template instances can grant friendship/be granted friendship.
  - Class `C` is friend of every instance of template class `U`.
  - Class `U<int>` is friend of class `D` (but no other instance of `U` is).
  - Every instance of template class `U` is friend of class `E`.
  - For every type T, class `U<`T`>` is friend of class `V<`T`>`.
  - Every instance of template class `U` is friend of every instance of template class `W`.

It is the individual instances that grant/receive friendship status.

Templates and Type Names



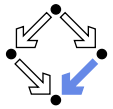
Classes may also hold type definitions.

```
template<int N, typename T> class Array {
  ...
  typedef T value_type;
}

template<typename C>
typename C::value_type get(C a, int i) {
  return a[i];
}

Array<100,int> arr0;
int val = get(arr0, 0); // automatic: get<Array<100,int> >
```

- `typename C::member`
  - Indicates that `C::member` denotes a type (not a value).
  - Compiler needs to know this for correctly parsing the template.

Type names as class members are frequently used for "traits" (see later).

Templates and Name Lookup

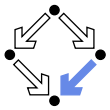


```
template<int N, typename T> class Array { ... protected: T a[N]; ... }

template<int N, typename T> class SortableArray2: public Array<N, T> {
public:
  virtual bool isSorted(int i, int j)
  { return this->a[i] <= this->a[j]; } // wrong: "return a[i] < a[j]"
}
```

- Nondependent names: looked up in context of template declaration.
  - `i` and `j` are non-dependent (do not depend on template parameters).
  - `return a[i] <= a[j]`: variable `a` is looked up in template context (i.e. typically reported as "undeclared" by compiler).
- Dependent names: looked up in context of instantiated template.
  - Name `this` is always dependent.
  - `return this->a[i] <= this->a[j]`: variable `a` is looked up in the context of the instantiated class.

In templates, better refer to class members as `this->`*`member`* (this is mandatory when referring to members of base classes).

Template Metaprogramming

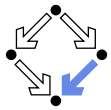


With templates, programs can be written that are executed by compiler.

```
template<int x, unsigned y>
struct ipower {
  static const int value = x*ipower<x,y-1>::value;
};
template<int x>
struct ipower<x, 0> {
  static const int value = 1;
};

int x = ipower<2,5>::value; // 2^5=32, computed by compiler
```

Attention: template metaprograms may take a long time and need not even terminate (compiler might loop forever).

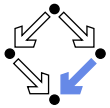
Template Metaprogramming



```
struct empty {};
template<typename H, typename T> struct node {
  typedef H head; typedef T tail; };

template<typename T, typename U> struct same {
  static const bool result = false; };
template<typename T> struct same<T, T> {
  static const bool result = true; };

template<typename T, typename L> struct member {
  static const bool result =
    same<T, typename L::head>::result ||
    member<T, typename L::tail>::result; };
template<typename T> struct member<T, empty> {
  static const bool result = false; };

typedef node<int, node<bool, node<char, empty> > > typelist;
bool haschar = member<char, typelist>::result; // true, computed by compiler
```

Lists of types that are processed at compile time.

Compiling Templates

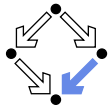


How do compilers handle the compilation of a template `t`?

- Approach 1: Both declaration and definition in header file `t.h`.
  - Some compilers create separate instances for each compiled source; linker ensures that final executable does not contain duplicates.
  - Some compilers maintain a repository of generated instances; this repository is looked up before generating a new instance.
- Approach 2: Only declaration in `t.h`, definition in source file `t.cpp`.
  - Some compilers locate the definition automatically from the name of the template and create new instances from there (if required).
- C++ standard: keyword `export`.
  - If template declarations and definitions are prefixed by `export`, approach 2 is chosen (approach 1, otherwise).
  - Only few compilers support `export`.

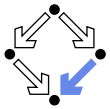
How to organize source code such that it works for all approaches?

Compiling Templates: Configurations



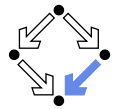
```
// config.h
#ifndef CONFIG_H
#define CONFIG_H

...
// settings for gcc
#ifdef __GNUC__
  #undef HAS_EXPORT
  #define NEED_TEMPLATE_DEFINITIONS
#endif
...

#ifdef HAS_EXPORT
  #define EXPORT export
#else
  #define EXPORT
#endif

#endif CONFIG_H
```

Compiling Templates: Declarations

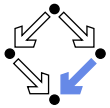


```
// point.h
#ifndef POINT_H
#define POINT_H

#include "config.h"

// template declaration, definitions only for inlining
EXPORT template<typename T> class Point {
  T x; T y;
public:
  Point(T a, T b): x(a), y(b) { }
  T getX() { return x; }
  Point flip();
  static Point copy(Point p);
};

#ifdef NEED_TEMPLATE_DEFINITIONS
#include "point.cpp";
#endif

#endif POINT_H
```

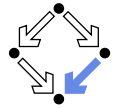

Compiling Templates: Definitions



```
// point.cpp
#include "point.h"

// template definitions
EXPORT template<typename T> Point<T> Point<T>::flip() {
  return Point(-x,-y);
}
EXPORT template<typename T> Point<T> Point<T>::copy(Point p) {
  return Point(p.x, p.y);
}

// main.h
#include "point.h"
int main()
{
  Point<int> p(10, 20);
  ...
}
```

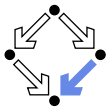
Example: Generic Lists



```
// empty list is created: ()
List<int> l; cout << l;

// 3 elements are inserted at positions 0,1,1: (1, 3, 2)
l.insert(0, 1).insert(1, 2).insert(1, 3); cout << l;

// element is removed and new one is inserted: (1, 4, 2)
l. remove(1).insert(1, 4); cout << l;

// element at position 1 is updated: (1, 2, 2)
l[1] = 2; cout << l;

// a copy of the list is created: (1, 2, 2)
List<int> r = l; cout << r;

// copy is updated, original is unchanged: (1, 3, 2) (1, 2, 2)
r[1] = r[1]+1; cout << r; cout << l;

// list is replaced: (1, 3, 2)
l = r; cout << l;
```

A list of values with flexible access and modification operations.

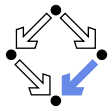
Example: Generic Lists



```
// forward declarations
template<typename T> class List;
template<typename T> ostream& operator<<(ostream& out, List<T>& l);

// Node gives friend status to List and operator<<
template<typename T> class Node
{
  T value;
  Node* next;
public:
  Node(T& v, Node* n): value(v), next(n) { }
  friend class List<T>;
  friend ostream& operator<< <T>(ostream& out, List <T> &l);
};
```

Class for representation of list nodes.

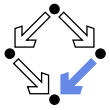
Example: Generic Lists



```
template<typename T> class List
{
  Node<T> *head;        // NULL or pointer to first node
  int len;              // number of nodes
  void reset();         // set list to empty
  void copy(List &l);   // copy content of l to this list
public:
  List();                    // empty list
  ~List();                   // discard all values of list
  List(List& l);             // copy values of l to this list
  List& operator=(List& l);  // assign values of l to this list
  int length();              // number of values in list
  T& operator[](int i);      // reference to value at position i
  List<T>& insert(int i, T value); // insert value at position i
  List<T>& remove(int i);          // remove value from position i

  // give friend status to operator <<
  friend ostream& operator<< <T>(ostream& out, List <T>& l);
};
```

The nodes of a list are never exposed and not shared with any other list.

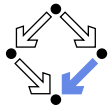
Example: Generic Lists



```
template<typename T> ostream& operator<<(ostream& out, List<T>& l)
{
  Node<T> *node = l.head;
  out << "(";
  while (node != NULL)
  {
    out << node->value;
    node = node->next;
    if (node != NULL) out << ", ";
  }
  out << ")";
  return out;
}
```

Printing requires access to list head and node fields.

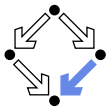
Example: Generic Lists



```
template<typename T> List<T>::List(): head(NULL), len(0) { }
template<typename T> List<T>::~List() { reset(); }
template<typename T> List<T>::List(List& l) { copy(l); }
template<typename T> List<T>& List<T>::operator=(List& l) {
  reset(); copy(l); return l;
}

template<typename T> int List<T>::length() { return len; }

template<typename T> T& List<T>::operator[](int i) {
  Node<T> *node = head;
  for (int j=0; j<i; j++)
    node = node->next;
  return node->value;
}
```

The most fundamental list operations.

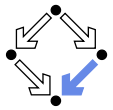
Example: Generic Lists



```
template<typename T> List<T>& List<T>::insert(int i, T value)
{
   Node<T> *prev = NULL;
   Node<T> *next = head;
   for (int j=0; j<i; j++)
   {
     prev = next;
     next = next->next;
   }
   Node<T> *node = new Node<T>(value, next);
   if (prev == NULL)
     head = node;
   else
     prev->next = node;
   len = len+1;
   return *this;
}
```

Creating a new node and inserting it at the denoted position.

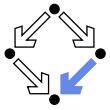
Example: Generic Lists



```
template<typename T> List<T>& List<T>::remove(int i)
{
   Node<T> *prev = NULL;
   Node<T> *next = head;
   for (int j=0; j<i; j++)
   {
     prev = next;
     next = next->next;
   }
   if (prev == NULL)
     head = next->next;
   else
     prev->next = next->next;
   len = len-1;
   delete next;
   return *this;
}
```

Removing a node from the denoted position (deleting it from memory).

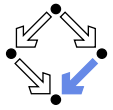
Example: Generic Lists



```
template<typename T> void List<T>::reset()
{
  Node<T> *node = head;
  while (node != NULL)
  {
    Node<T> *prev = node;
    node = node->next;
    delete prev;
  }
  head = NULL;
  len = 0;
}
```

Resetting a list to the original status (deleting all nodes).

Example: Generic Lists



```
template<typename T> void List<T>::copy(List& l)
{
  Node<T> *prev = NULL;
  int n = l.length();
  Node<T> *node = l.head;
  for (int i=0; i<n; i++)
  {
    Node<T> *node0 = new Node<T>(node->value, NULL);
    if (prev == NULL)
      head = node0;
    else
      prev->next = node0;
    prev = node0;
    node = node->next;
  }
  len = n;
}
```

Copying the nodes from another list (assuming this list is empty).