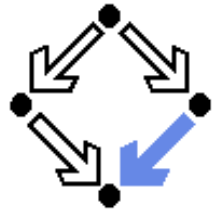
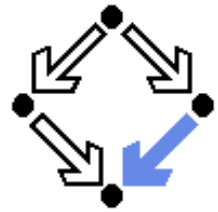
The Standard Library

Wolfgang Schreiner
Wolfgang.Schreiner@risc.uni-linz.ac.at

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



The Standard Library



- Set of headers with declarations.

  `#include <`*name*`>`

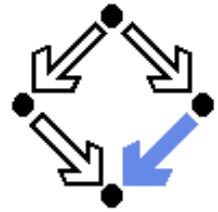
  - Headers need not be physical files (do not use `<`*name*`.h>`).
- Almost all names are in namespace `std`.

  ```
  using namespace std;
  ```

  - Only exceptions are global operators `new` and `delete` (header `<new>`).
- Provides lot of basic functionality.
  - Numerics.
  - Input/Output.
  - Containers, iterators, algorithms.

For effective programming, it is important to know not only a programming language but also the associated basic libraries.

C Library Wrappers



For backward compatibility, the entire C standard library is included.

C++ Header	C Header
<cstdio>	<stdio.h>
<cstdlib>	<stdlib.h>
<cstring>	<string.h>
<cmath>	<math.h>
...	...

- Use of C++ header (places name in namespace std)

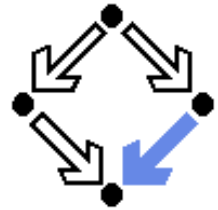
```
#include <cstdio>
int main() { std::printf("Hello, world"); }
```

- Use of C header (places name in global namespace)

```
#include <stdio.h>
int main() { printf("Hello, world"); }
```

The C++ library provides better alternatives for writing new applications.

Traits and Policies



The standard library makes heavy use of traits and policies.

- Trait: a class that provides information about a type.
  - By type definitions and/or static member data in the trait.
- Policy: a trait that also defines an operational interface for the type.
  - By static member functions in the policy.
- Often implemented as specializations of dummy templates.

```
template <type T> class Trait { };   // dummy trait template
template<> class Trait<int> { ... }; // trait for type "int"
```

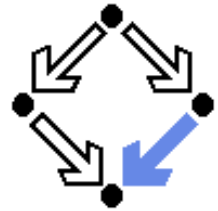
  - Thus the trait for a type can be deduced from the name of a type.
- Mainly used as template arguments.

```
template<class C, class T = Trait<C> >
class Lib { ... C ... T::member ... };
```

  - Template thus receives required information about type parameter.
  - Since trait holds information, atomic type can be template argument.

Many standard types are instantiations of templates with traits/policies.

Example: Class `string`



C++ strings are actually parameterized over the character type.

```
// header <string>
template<typename charT> struct char_traits;
template<> struct char_traits<char> { ... }

template<class charT, class traits = char_traits<charT>, ... >
class basic_string { ... }
typedef basic_string<char> string;
```

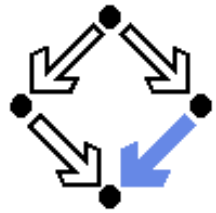
- Wide character type: `wchar_t`
  - Narrow character type `char` is only one byte large.
  - `wchar_t` is typically 32 bit large and may hold any Unicode character.

    ```
    wchar_t pi = '\u03c0'; // greek character "pi"
    ```
- Wide strings: another string type provided by the library.

  ```
  template<> struct char_traits<wchar_t> {...}
  typedef basic_string<wchar_t> wstring;
  ```

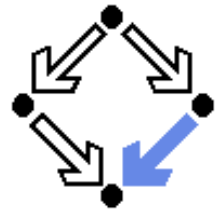
The whole library (also I/O) works with any character type.

Example: Strings that Ignore Cases



```
template<typename T> struct ci_char_traits { };
template<> struct ci_char_traits<char> {
  typedef char char_type; typedef int int_type;
  typedef std::streamoff off_type; typedef std::streampos pos_type;
  typedef std::mbstate_t state_type;
  static void assign(char_type& dst, const char_type src) { dst = src; }
  static char_type* assign(char* dst, std::size_t n, char c)
  { return static_cast<char_type*>(std::memset(dst, n, c)); }
  static bool eq(const char_type& c1, const char_type& c2)
  { return lower(c1) == lower(c2); }
  static bool lt(const char_type& c1, const char_type& c2)
  { return lower(c1) < lower(c2); }
  static int compare(const char_type* s1, const char_type* s2, std::size_t n) {
    for (size_t i = 0; i < n; i++) {
      char_type lc1 = lower(s1[i]); char_type lc2 = lower(s2[i]);
      if (lc1 < lc2) return -1; if (lc1 > lc2) return +1;
    }
    return 0;
  }
  static int_type lower(char_type c) { return std::tolower(to_int_type(c)); }
  ...
};
```

Example (Contd)

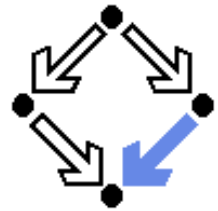


```
typedef std::basic_string<char, ci_char_traits<char> > ci_string;

int main()
{
  ci_string s1 = "Hello, World";
  ci_string s2 = "hello, world";
  std::cout << (s1 == s2); // "true";
}
```

Ray Lischner "C++ in a Nutshell".

Allocators



The standard library is also generic with respect to memory management.

- Allocator: a policy for managing dynamic memory.
  - Use of `new` and `dispose` is not hard-wired in the standard library.
- The library provides a standard allocator

```
// header <memory>
template <class T> class allocator { ...}
```

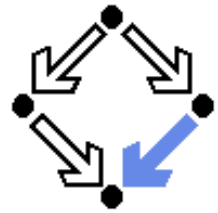
- Standard library classes use this allocator by default

```
// header <string>
template<class charT, class traits = char_traits<charT>,
    class Alloc = allocator<charT> >
class basic_string { ...}
```

- Other allocation schemes are possible

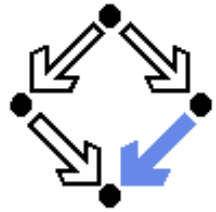
```
template<> class allocator<int> {...} // globally used
class MyCharAllocator {...}           // selectively used
typedef basic_string<char, char_traits<char>,
    MyCharAllocator> mystring;
```

Example



```
template<typename T> class myallocator {
public:
  typedef std::size_t size_type; typedef std::ptrdiff_t difference_type;
  typedef T* pointer; typedef const T* const_pointer;
  typedef T& reference; typedef const T& const_reference;
  typedef T value_type;
  template <class U> struct rebind { typedef myallocator<U> other; };
  myallocator() throw() {}
  myallocator(const myallocator&) throw()        {}
  template <class U> myallocator(const myallocator<U>&) throw() {}
  ~myallocator() throw() {}
  pointer address(reference x) const {return &x;}
  const_pointer address(const_reference x) const {return &x;}
  pointer allocate(size_type n, void* hint = 0)
  { return static_cast<T*>(::operator new (n * sizeof(T)) ); }
  void deallocate(pointer p, size_type n)
  { ::operator delete(static_cast<void*>(p)); }
  size_type max_size() const throw()
  { return std::numeric_limits<size_type>::max() / sizeof(T); }
  void construct(pointer p, const T& val) { new(static_cast<void*>(p)) T(val); }
  void destroy(pointer p) { p->~T(); }
};
```

Example (Cntd)



```
template<> class myallocator<void> {
public:
  typedef void* pointer; typedef const void* const_pointer;
  typedef void value_type;
  template <class U> struct rebind { typedef myallocator<U> other; };
};

template<typename T>
bool operator==(const myallocator<T>&, const myallocator<T>&) { return true; }

template<typename T>
bool operator!=(const myallocator<T>&, const myallocator<T>&) { return false; }

int main() {
  std::list<int, myallocator<int> > data;
  data.push_back(10);
  data.push_back(20);
  return data.size();
}
```

Ray Lischner "C++ in a Nutshell".