

Programmieren 3 Zusammenfassung

Samuel Mueller

11. Januar 2014

Contents

1 Terminology	6
1.1 Expression	6
1.2 Statement	6
1.3 Declaration	6
1.4 Definition	6
1.5 Parameter and Argument	7
1.6 Predicate	7
2 Basic Concepts	8
2.1 Functions	8
2.1.1 A good function	8
2.1.2 return statement	8
2.1.3 function as argument	8
2.2 Exceptions	9
2.2.1 Throwing Anything	9
2.2.2 Throwing Standard Exceptions	10
2.2.3 Function Followed By Try-Catch	10
2.3 Operator Overloading	11
2.3.1 Basic Example	11
2.3.2 less than comparable	12
2.3.3 Increment Postfix vs. Prefix	13
2.4 Enums	14
2.4.1 Enums Are Integers	14
2.4.2 Indices Can Be Defined	15
2.4.3 Casting	16
2.4.4 Operator Overloading	17
2.4.5 Example: Toggle Button	18
2.5 Lambdas	20
2.5.1 Basic Examples	20
2.5.2 Function Objects	20
2.5.3 Capturing	21
3 Classes	23
3.1 Inheritance	23
3.1.1 constructor calls	23
3.1.2 calling super constructors	24
3.1.3 pure virtual function	25
3.1.4 object slicing	26
3.1.5 object slicing with reference and virtual	27
3.1.6 member hiding problem	28
4 Immutability	29
4.1 Const Function Parameters	29
4.2 Const Return Values	30
4.3 Const Objects	31
4.4 Const Values And Pointers	32

5	Streams	33
5.1	Handling Invalid Input	33
5.1.1	Escape After Fail	33
5.1.2	Continue After Fail	34
5.2	Manipulators	35
5.2.1	boolalpha	35
5.2.2	skipws	35
5.2.3	uppercase	36
5.2.4	oct, hex, dec	36
5.2.5	setw	37
5.2.6	left, right, internal	38
5.2.7	setprecision, scientific, fixed	39
6	Iterators	40
6.1	Insert Iterators	40
6.1.1	Insert iterator	40
6.1.2	Back insert iterator	40
6.1.3	Front insert iterator	41
6.2	Stream Iterators	42
6.2.1	Iterate over strings	42
6.2.2	Iterate over ints	42
6.2.3	Iterate over your own types	43
6.2.4	Count Chars	44
6.2.5	Count Specific Chars	44
6.2.6	Count Words	45
6.3	Custom Iterators	46
6.3.1	Line Iterator	46
6.3.2	My Ostream Iterator	48
6.3.3	Square Iterator	49
6.3.4	Prime Iterator	51
6.4	Iterator Semantics	52
6.4.1	Iterator	52
6.4.2	Reverse Iterator	53
7	Containers	54
7.1	Vector	54
7.1.1	Initialization	54
7.1.2	Accessing Elements	54
7.1.3	Assigning Values	55
7.1.4	Appending Values	55
7.1.5	Iterating Elements	56
8	Algorithms	57
8.1	Modifying	57
8.1.1	Copy	57
8.1.2	Move	58
8.1.3	Transform	59
8.2	Partitioning	60
8.2.1	Partitioning	60
8.3	Numeric	61

8.3.1	Accumulate	61
8.3.2	Adjacent Difference	61
8.3.3	Inner Product	62
8.3.4	Partial Sum	62
8.4	Misc	63
8.4.1	Fill Vector With Squares	63
8.4.2	Is Palindrome	64
8.4.3	Multiplication Table	65
8.4.4	Sum Numbers	66
8.4.5	Word List	67
9	Bind	68
9.1	Examples	68
10	Templates	69
10.1	Function Templates	69
10.1.1	median	69
10.1.2	rotate 3 arguments	69
10.1.3	read line	70
10.1.4	read line 2	71
10.1.5	calling unimplemented functions	72
10.1.6	int and double dilemma	72
10.1.7	string fallacy	73
10.2	Class Templates	74
10.2.1	class template specialization	74
10.2.2	prohibited construction	75
10.2.3	vector delegator	76
11	References And Pointers	77
11.1	Basic Examples	77
11.1.1	Reference on int	77
11.1.2	Reference on cat	78
11.2	Reference vs. Pointer	79
11.3	C++ vs. Java	80
11.3.1	Reference in C++	80
11.3.2	Reference in Java	80
11.4	Dynamic Heap Memory Management	81
11.4.1	Unique Pointer	81
11.4.2	Shared Pointer	82
12	Compile Time Calculation	83
12.1	constexpr	83
12.2	static_assert	83
12.3	User Defined Literals	84
12.4	Ring5	85

13 Good To Know	86
13.1 Default floating point type	86
13.2 Assigning floating point to int	86
13.3 Bool is an integer	86
13.4 Initializing variables	87
13.5 Floating points cant be unsigned	87
13.6 Literals	88
13.7 Weird string syntax	88
13.8 Arithmetics with int and double	89
13.9 Unspecified invocation order	89
13.10Factory function	90
13.11Args	90
14 Appendix	91
14.1 Random Code	91
14.1.1 PIMPL idiom	91
14.1.2 Word	95
14.1.3 Ring	97
14.1.4 Sack	98
14.1.5 Highlander	100
14.1.6 dynArray	101

1 Terminology

1.1 Expression

```
1 int foo{1}, bar{2};
2
3 int r = foo + bar;
4 // 'foo + bar' is an expression
5 // 'foo' itself is also an expression
```

1.2 Statement

```
1 void if_statement() {
2     if (true) {} // valid statement
3 }
4
5 void adding_two_ints() {
6     int three = 1 + 2; // '1 + 2' is a statement
7 }
8
9 void calling_a_function() {
10     adding_two_ints(); // calling the function adding_two_ints() is a statement
11 }
12
13 void executable_part_of_a_function() {
14     // the entire code between { } is a statement
15 }
```

1.3 Declaration

```
1 // a declaration gives the compiler information about the signature of a function.
2 // that way, it will know the return type, the name and the parameters of the
3 // function.
4 // however it does not matter to the compiler what the implementation looks like.
5 int this_is_a_declaration(int a, char b);
```

1.4 Definition

```
1 // this is the definition of the previously declared function.
2 // it implements the logic of the function in between {}
3 // a definition is always also a declaration. in this example, the previous
4 // declaration is actually not necessary.
5 int this_is_a_declaration(int a, char b) {
6     // implementation logic goes here
7 }
```

1.5 Parameter and Argument

```
1 // parameter
2 void call_me(int a) { } // 'a' is a parameter
3
4 // argument
5 void call_it() {
6     call_me(1); // '1' is the argument for the call to call_me()
7 }
```

1.6 Predicate

```
1 #include <iostream>
2 #include <vector>
3 #include <algorithm>
4 #include <iterator>
5
6 // a predicate is something that delivers a boolean value on a certain input.
7 // we differ between unary predicates, which take one argument, and binary
8 // predicates, which take two arguments.
9 // in C++, we can represent predicates with the following constructs:
10
11 // a struct overriding the call-operator
12 struct odd_s {
13     bool operator() (int n) {
14         return (n % 2);
15     }
16 };
17
18 // a function
19 bool odd_f(int n) {
20     return (n % 2);
21 }
22
23 // a lambda
24 auto odd_l = [](int n){ return (n % 2); };
25
26 void predicate() {
27     using namespace std;
28
29     vector<int> src{1, 2, 3, 4, 5, 6};
30
31     copy_if(src.begin(), src.end(), ostream_iterator<int>{cout}, odd_f);
32     copy_if(src.begin(), src.end(), ostream_iterator<int>{cout}, odd_l);
33     copy_if(src.begin(), src.end(), ostream_iterator<int>{cout}, odd_s{});
34
35     // output
36     // 135135135
37 }
```

2 Basic Concepts

2.1 Functions

2.1.1 A good function ...

- does exactly one thing (also known as high cohesion)
- has a name that describes its behaviour
- has only few parameters (up to 3 is preferable, no more than 5)
- consists of only a few lines of code
- does not contain deeply nested constructs (if/else, loops etc.)
- guarantees a clear result (also known as contract)

2.1.2 return statement

```
1  #include <iostream>
2
3  int wild_function() {
4      // the compiler automatically returns 1 if we dont define a return statement
5  }
6
7  void return_statement() {
8      std::cout << "wild_function(): " << wild_function() << std::endl;
9
10     // output
11     // wild_function(): 1
12 }
```

2.1.3 function as argument

```
1  double maybe(double d) {
2  }
3
4  // we can define functions as parameters with the following syntax:
5  void call_me(double f(double)) {
6      f(1);
7  }
8
9  void function_as_argument() {
10     call_me(maybe);
11
12     // we can also pass in a lambda with the same signature:
13     auto for_sure = [](double d) {return .0;};
14     call_me(for_sure);
15 }
```

2.2 Exceptions

- Exceptions in C++ have very small overhead. They don't provide additional information such as a stack trace or source code location
- Everything that is copyable can be thrown
- All exceptions are unchecked

2.2.1 Throwing Anything

```
1  #include <string>
2  #include <sstream>
3
4  void throwing_anything() {
5      try {
6          throw 1;
7      } catch (int i) {}
8
9      try {
10         throw "char";
11     } catch (char const * i) {}
12
13
14     try {
15         throw std::string{"string"};
16     } catch (std::string str) {}
17
18     // we can not throw a stream because streams are not copyable
19     // throw std::istringstream{"stream"};
20     // ^ error (...)
21 }
```

2.2.2 Throwing Standard Exceptions

```
1  #include <stdexcept>
2
3  void throwing_std_exceptions() {
4      try {
5          throw std::logic_error{"something weird happend"};
6      } catch (std::logic_error e) {}
7  }
8
9  // other exceptions include
10 //
11 // std::logic_error
12 // std::domain_error
13 // std::invalid_argument
14 // std::length_error
15 // std::out_of_range
16 // std::runtime_error
17 // std::range_error
18 // std::overflow_error
19 // std::underflow_error
```

2.2.3 Function Followed By Try-Catch

```
1  // a functions definition can directly be implemented with a try-catch block
2  void function_followed_by_try_catch()
3  try {
4      // useful implementation that may cause exception goes here
5  } catch(int i) {
6
7  }
```

2.3 Operator Overloading

2.3.1 Basic Example

```

1  #include <iostream>
2
3  struct Age {
4      Age(int age) : data{age} {}
5
6      bool operator<(Age const & comparable) const {
7          //          ^---          ^--- not a requirement, but good practice
8          //          for operators which dont change the state
9          return data < comparable.data;
10     }
11
12     int data;
13 };
14
15 // an operator can also be defined outside of the types context.
16 // note that the member 'data' must be public in order to access it.
17 bool operator<=(Age& a, Age& b) {
18     return a.data < b.data;
19 }
20
21 void basic_example() {
22     Age a1{20};
23     Age a2{30};
24
25     std::cout << std::boolalpha << "20 < 30: " << (a1 < a2) << std::endl;
26     std::cout << std::boolalpha << "20 <= 30: " << (a1 <= a2) << std::endl;
27     // using the operator with a function call:
28     std::cout << std::boolalpha << "20 < 30: " << (a1.operator<(a2)) << std::endl;
29     // output
30     // 20 < 30: true
31     // 20 <= 30: true
32     // 20 < 30: true
33
34     // greater than is unimplemented, using it causes a compiler error:
35     // std::cout << std::boolalpha << "20 > 30: " << (a1 > a2) << std::endl;
36     //          ^ error: (...)
37 }

```

2.3.2 less than comparable

```

1  #include <vector>
2  #include <string>
3  #include <iostream>
4
5  #include <boost/operators.hpp>
6
7  // we can inherit from boosts less_than_comparable and implement the less-than
8  // operator to have all other operators ready for our type:
9  struct Number : private boost::less_than_comparable<Number> {
10     Number(int n) : data{n} {}
11
12     bool operator<(Number const& comparable) const {
13         data < comparable.data;
14     }
15
16     private: int data;
17 };
18
19 void boost_less_than_comparable() {
20     Number n1{0};
21     Number n2{1};
22
23     // we can now call all the comparison operators:
24     n1 < n2;
25     n1 > n2;
26
27     // heres an outline of how boost can simulate each operators behaviour with the
28     // less-than operator:
29     //
30     // operator    simulation with <
31     // -----
32     // a > b        (b < a)
33     // a >= b       !(a < b)
34     // a <= b       !(b < a)
35     // a == b       !(a < b) && !(b < a)
36     // a != b       !(a == b)
37 }

```

2.3.3 Increment Postfix vs. Prefix

```
1  #include <iostream>
2
3  struct Inc {
4      void operator++() {
5          std::cout << "this is the prefix increment" << std::endl;
6      }
7
8      // we must use a dummy parameter of type int so the compiler can differ between the
9      // pre- and postfix increment operators. the dummy parameter doesnt have any
10     // specific value:
11     void operator++(int dummy) {
12         std::cout << "this is the postfix increment" << std::endl;
13     }
14 };
15
16 void increment_postfix_vs_prefix() {
17     Inc i{};
18     ++i;
19     i++;
20
21     // output
22     // this is the prefix increment
23     // this is the postfix increment
24 }
```

2.4 Enums

2.4.1 Enums Are Integers

```
1  #include <iostream>
2
3  void enums_are_integers() {
4      enum season { Spring, Summer, Autumn, Winter };
5          //           0         1         2         3
6
7      int summer = Summer;
8      std::cout << "summer: " << summer << std::endl;
9      std::cout << "Summer + Winter: " << Summer + Winter << std::endl;
10
11     // output
12     // summer: 1
13     // Summer + Winter: 4
14
15     // increment or decrement does not work however:
16     //
17     // Spring++;
18     //           ^ error: no operator++(int) declared for postfix ++ [-fpermissive]
19 }
```

2.4.2 Indices Can Be Defined

```

1  #include <iostream>
2
3  void indices_can_be_defined() {
4      enum season { Spring = 10, Summer, Autumn, Winter };
5      //          10          11          12          13
6      std::cout << "Spring: " << Spring << std::endl;
7      std::cout << "Winter: " << Winter << std::endl;
8      // output
9      // Spring: 10
10     // Winter: 13
11
12     enum brand { Feldschloesschen = 1, Calanda, Quoellfrisch,
13     //          1          2          3
14     Budweiser = 10, Heineken, SanMiguel };
15     //          10          11          12
16
17     std::cout << "Feldschloesschen: " << Feldschloesschen << std::endl;
18     std::cout << "Calanda: " << Calanda << std::endl;
19     std::cout << "Quoellfrisch: " << Quoellfrisch << std::endl;
20     std::cout << "Budweiser: " << Budweiser << std::endl;
21     std::cout << "Heineken: " << Heineken << std::endl;
22     std::cout << "SanMiguel: " << SanMiguel << std::endl;
23     // output
24     // Feldschloesschen: 1
25     // Calanda:          2
26     // Quoellfrisch:     3
27     // Budweiser:       10
28     // Heineken:        11
29     // SanMiguel:       12
30 }
31
32 // enum class d_o_w
33 //   Was ist der Unterschied? Bezeichner sind nicht in einem bestimmten Scope drin
34 //   und global verwendbar.

```

2.4.3 Casting

```
1 void casting() {
2     enum season { Spring, Summer, Autumn, Winter };
3     //           0         1         2         3
4
5     int summer = Summer;
6
7     season favourite_season{Winter};
8     std::cout << "favourite_season: " << favourite_season << std::endl;
9     // output
10    // favourite_season: 3
11
12    // favourite_season = 1;
13    //                   ^ error: invalid conversion from int to casting()::season
14
15    favourite_season = static_cast<season>(1);
16    std::cout << "favourite_season: " << favourite_season << std::endl;
17
18    // output
19    // favourite_season: 1
20 }
```

2.4.4 Operator Overloading

```
1  #include <iostream>
2
3  enum Belt { Yellow, Orange, Green, Blue, Brown, Black };
4  //           0         1         2         3         4         5
5
6  Belt operator++(Belt &belt) {
7      int tmp = (belt + 1) % (Black + 1);
8      belt = static_cast<Belt>(tmp);
9      return belt;
10 }
11
12 void operator_overloading() {
13     Belt belt{Brown};
14
15     std::cout << "belt: " << belt << std::endl;
16     ++belt;
17     std::cout << "belt: " << belt << std::endl;
18     ++belt;
19     std::cout << "belt: " << belt << std::endl;
20
21     // output
22     // belt: 4
23     // belt: 5
24     // belt: 0
25
26     // calling this directly does not work
27     // ++Green;
28     // ^ error: no match for operator++ in ++(Belt)2u
29 }
```

2.4.5 Example: Toggle Button

```
1  #include <ostream>
2  #include <stdexcept>
3
4  struct ToggleButton {
5
6      void hit_button() {
7          state = static_cast<State>((state + 1) % 3);
8      }
9
10     void print(std::ostream &out) const {
11         if (state == State::OFF) {
12             out << "OFF";
13         } else if (state == State::ON) {
14             out << "ON";
15         } else if (state == State::BLINK) {
16             out << "Blink";
17         } else {
18             throw std::invalid_argument {"Unkown state"};
19         }
20     }
21
22     private:
23
24     enum State : short {
25         OFF, ON, BLINK
26     };
27
28     State state { State::OFF };
29
30 };
31
32 std::ostream& operator<<(std::ostream &out, ToggleButton const &button) {
33     button.print(out);
34     return out;
35 }
```

```
1  #include "toggle_button.h"
2
3  void toggle_button_demo() {
4      ToggleButton button{};
5
6      std::cout << "button: " << button << std::endl;
7
8      button.hit_button();
9
10     std::cout << "button: " << button << std::endl;
11
12     button.hit_button();
13
14     std::cout << "button: " << button << std::endl;
15
16     button.hit_button();
17
18     std::cout << "button: " << button << std::endl;
19
20     // output
21     // button: OFF
22     // button: ON
23     // button: Blink
24     // button: OFF
25 }
```

2.5 Lambdas

2.5.1 Basic Examples

```

1  #include <iostream>
2  #include <string>
3
4  void nothing() {
5      auto gapin_void = [ ]( ) { };
6      //           ^ ----- captures
7      //           ^ ----- parameters
8      //           ^ ----- implementation
9
10     gapin_void(); // calling a lambda is syntactically similar to calling a function
11 }
12
13 void say_hi() {
14     auto hello_thing = [](std::string thing) {
15         std::cout << "hello " << thing << "!" << std::endl;
16         return true;
17     };
18
19     bool result = hello_thing("world");
20
21     std::cout << "result: " << result << std::endl;
22
23     // output
24     // hello world!
25     // result: 1
26 }

```

2.5.2 Function Objects

```

1  #include <iostream>
2  #include <functional>
3
4  void auto_vs_function() {
5      // in the previous example we saw how the lambda was assigned to a variable of
6      // type auto. we can also use std::function<signature>:
7      std::function<void(double)> p1 = [](double a) { };
8      std::function<double(double)> p2 = [](double a) { return .0; };
9
10     // checking if something was assigned to a function instance:
11     if (p1) {
12         std::cout << "p1 does something" << std::endl;
13     }
14
15     // output
16     // p1 does something
17 }

```

2.5.3 Capturing

```
1 void capture_for_internal_use() {
2     int magic_number = 8616;
3
4     // if we want to use previously declared variables inside a lambda, we have to
5     // capture them. otherwise, a compiler error will occur:
6     auto capturer = [magic_number]() {
7         //             ^--- capturing magic_number
8
9         int copy = magic_number;
10
11         // trying to change magic_number causes a compiler error
12         // magic_number = 9548;
13         //             ^ error: assignment of read-only variable magic_number
14     };
15     capturer();
16 }
```

```
1 #include <string>
2 #include <iostream>
3
4 void capture_with_write_access() {
5     std::string name{"batman"};
6
7     std::cout << name << std::endl;
8
9     auto change = [&name]() {
10         name = "spiderman";
11     };
12
13     change();
14
15     std::cout << name << std::endl;
16
17     // output
18     // batman
19     // spiderman
20 }
```

```
1  #include <string>
2
3  std::string PIE{"omnom"};
4
5  void capture_everything_by_reference() {
6      double PI = 3.141;
7
8      auto wicked_lambda = [&]() {
9          PIE = "mjam";
10         PI = 3.142;
11     };
12
13     wicked_lambda();
14 }
15
16 void capture_everything_by_value() {
17     double PI = 3.141;
18
19     auto wicked_lambda = [=]() {
20         PIE = "mjam";
21         // PI = 3.142;
22         // ^ error: assignment of read-only variable PI
23     };
24
25     wicked_lambda();
26 }
27
28
29 void mutable_demo() {
30     int n = 1;
31
32     // if we wanted to change a captured variable inside a lambda, we have to define
33     // the lambda mutable. note that this is different to using the ampersand syntax,
34     // since the changes wont be reflected to the outside.
35     auto lambda = [n]() mutable {
36         n = 2;
37
38         // if we didnt specify mutable, the following compiler error would occur:
39         // ^ error: assignment of read-only variable n
40     };
41
42     // n has only been changed inside the lambda
43     std::cout << "n: " << n << std::endl;
44
45     // output
46     // n: 1
47 }
```

3 Classes

3.1 Inheritance

3.1.1 constructor calls

```
1  #include <iostream>
2
3  struct Furniture {
4      Furniture() { std::cout << "furniture created" << std::endl; }
5      ~Furniture() { std::cout << "furniture destroyed" << std::endl; }
6  };
7
8  struct Table : Furniture {
9      Table() { std::cout << "table created" << std::endl; }
10     ~Table() { std::cout << "table destroyed" << std::endl; }
11 };
12
13 struct WoodenTable : Table {
14     WoodenTable() { std::cout << "wooden table created" << std::endl; }
15     ~WoodenTable() { std::cout << "wooden table destroyed" << std::endl; }
16 };
17
18 void constructor_calls() {
19     // the constructor of the base class is called first.
20     // the constructor of the derived class is called last.
21     // this order is reversed at the destruction of the object.
22     WoodenTable wooden_table{};
23
24     std::cout << "---" << std::endl;
25
26     // output
27     // furniture created
28     // table created
29     // wooden table created
30     // ---
31     // wooden table destroyed
32     // table destroyed
33     // furniture destroyed
34 }
```

3.1.2 calling super constructors

```

1  struct Furniture {
2      Furniture() { std::cout << "furniture created" << std::endl; }
3  };
4
5  struct Table : Furniture {
6      Table() { std::cout << "table created" << std::endl; }
7  };
8
9  struct WoodenTable : Table {
10
11      // we can call a constructor of the direct base class
12      WoodenTable() : Table{} {
13          std::cout << "wooden table created" << std::endl;
14      }
15
16      // if we try to call a constructor from the base type Furniture, we get a
17      // compiler error:
18      // WoodenTable() : Furniture{} {
19      //      ^ error: type Furniture is not a direct base of WoodenTable
20      // }
21
22      // we can also delegate an other constructor of the same type. the delegation is
23      // called first:
24      WoodenTable(int i) : WoodenTable{(double) i} {
25          std::cout << "wooden table created with " << i << std::endl;
26      }
27
28      WoodenTable(double i) {
29          std::cout << "doing more " << i << std::endl;
30      }
31
32  };
33
34  void calling_super_constructors() {
35      // calling the constructor which calls the constructor of the direct base:
36      WoodenTable table1{};
37      // furniture created
38      // table created
39      // wooden table created
40
41      // calling the constructor which delegates to a constructor in the same type.
42      // note how the delegation is called first:
43      WoodenTable table2{0};
44      // furniture created
45      // table created
46      // doing more 0
47      // wooden table created with 0
48  }

```

3.1.3 pure virtual function

```
1  struct Drink {
2      // a pure virtual function can be compared to an abstract method in Java.
3      // as soon as a class contains this kind of function, the type becomes abstract
4      // and can not be instantiated.
5      // we can declare pure virtual functions with the following syntax:
6      virtual void prepare() = 0;
7  };
8
9  struct Beer : Drink {
10     void prepare() {}
11 };
12
13 struct Rivella : Drink {
14     // we dont have to implement prepare() here.
15     // however, the type Rivella will remain abstract.
16 };
17
18 void pure_virtual_function() {
19     // trying to instantiate an object of abstract type Drink causes an error:
20     // Drink drink{};
21     //           ^ error: cannot allocate an object of abstract type 'Drink'
22
23     // we cant instantiate an object of type Rivella because it is abstract:
24     // Rivella rivella{};
25     //           ^ error: cannot allocate an object of abstract type 'Rivella'
26
27     Beer beer{};
28 }
```

3.1.4 object slicing

```
1  #include <iostream>
2
3  namespace example_1 {
4      struct Creature {
5          void walk() {}
6      };
7
8      struct Ork : Creature {
9          void grunt() {}
10     };
11
12     void object_slicing() {
13         // creatures can walk
14         Creature creature{};
15         creature.walk();
16
17         // orks can walk and grunt
18         Ork ork{};
19         ork.walk();
20         ork.grunt();
21
22         // here we make a copy of the ork instance.
23         // since we declare copy_of_ork to be of type Creature, only the part of that
24         // type is copied. copy_of_ork therefore can not grunt. this is called object
25         // slicing:
26         Creature copy_of_ork = ork;
27         copy_of_ork.walk();
28         // copy_of_ork.grunt();
29         // ^ error: 'struct Creature' has no member named 'grunt'
30     }
31 }
```

3.1.5 object slicing with reference and virtual

```

1 namespace example_2 {
2     struct Creature {
3         void walk() { std::cout << "creature is walking" << std::endl; }
4         virtual void grunt() { std::cout << "creature is grunting" << std::endl; }
5     };
6
7     struct Ork : Creature {
8         void walk() { std::cout << "ork is walking" << std::endl; }
9         void grunt() { std::cout << "ork is grunting" << std::endl; }
10    };
11
12    void object_slicing() {
13        Ork ork{};
14        Creature ork_c = ork; // a copy of ork.
15                               // contains the implementations of Creature only.
16        Creature& ork_r = ork; // a reference to an Ork object.
17                               // delegates member calls to the implementations of
18                               // Creature unless the member is declared virtual.
19        Creature* ork_p = &ork; // a pointer to an Ork object. behaves the same way
20                               // as the reference.
21
22        // the following gets called on type Ork, as expected
23        ork.walk(); // ork is walking
24        ork.grunt(); // ork is grunting
25
26        // the following gets called on type Creature due to object slicing
27        ork_c.walk(); // creature is walking
28        ork_c.grunt(); // creature is grunting
29
30        // the following calls depend on whether the declarations are virtual or not.
31        // if not declared virtual, the behaviour is the same as with object slicing.
32        // if declared virtual, the calls are delegated to the original type Ork.
33        ork_r.walk(); // creature is walking
34        ork_r.grunt(); // ork is grunting
35
36        ork_p->walk(); // creature is walking
37        ork_p->grunt(); // ork is grunting
38    }
39 }

```

3.1.6 member hiding problem

```
1  #include <iostream>
2
3  struct Vehicle {
4      void accelerate(int amount) {}
5      void repair() {}
6      void refuel() const { std::cout << "refueling vehicle" << std::endl; }
7  };
8
9  struct Toeff : Vehicle {
10     // because we overload the accelerate function in a derived class, we have to
11     // make the functions from the base class available with the following statement:
12     using Vehicle::accelerate;
13
14     void accelerate() {}
15     void refuel() { std::cout << "refueling toeff" << std::endl; }
16 };
17
18 void member_hiding_problem() {
19     Toeff t{};
20
21     // calling a function from base type Vehicle
22     t.repair();
23
24     // calling accelerate defined in type Toeff
25     t.accelerate();
26
27     // calling accelerate defined in type Vehicle
28     // this is only possible because using Vehicle::accelerate;
29     t.accelerate(1);
30
31     // if the using statement is not defined, the following compiler error is raised:
32     // t.accelerate(1);
33     //      ^ error: no matching function for call to Toeff::accelerate(int)
34
35     // the const keyword wont hide overloaded functions. the function of the derived
36     // class is called:
37     t.refuel(); // output: refueling toeff
38
39 }
```

4 Immutability

4.1 Const Function Parameters

```
1 // The parameter a is passed to this function as a reference and can be modified
2 // inside it.
3 void i_can_change_you(int& a) {
4     a++;
5 }
6
7 // This function declares its parameter as a reference and const. Const means that
8 // this parameter can not be modified inside the function.
9 // However, it is possible to pass in a non-const variable.
10 void i_cannot_change_you(const int& a) {
11     // a++;
12     // ^ error: increment of read-only reference 'a'
13 }
14
15 int test_i_cannot_change_you() {
16     // passing a non-const object to a function that declares the parameter const is
17     // possible:
18     int a = 1;
19     i_cannot_change_you(a);
20 }
21
22 // This function declares its parameter not as a reference and const. Const still
23 // means that the object can not be modified, however this makes less sense than in
24 // the previous example since this parameter is passed by value and therefore is a
25 // copy anyways.
26 void i_cannot_change_you_inside(const int a) {
27     // a ++;
28     // ^ error: increment of read-only parameter 'a'
29 }
30
31 // The previous function declaration is the same as the following, which would be
32 // ambiguous:
33 // void i_cannot_change_you_inside(int a) {
34 //     ^ error: redefinition of 'void i_cannot_change_you_inside(int)'
35 // }
```

4.2 Const Return Values

```
1 struct Dog {
2     int age = 0;
3
4     const int& getAge() {
5         return age;
6     }
7
8     int& getMutableAge() {
9         return age;
10    }
11 };
12
13 void const_return_values() {
14     Dog dog{};
15
16     // variables declared const can not be modified
17     const int& age = dog.getAge();
18     // not possible:
19     // age = 6;
20     //           ^ error: assignment of read-only reference 'age'
21
22     // not possible:
23     // int& age = dog.getAge();
24     //           ^ error: invalid initialization of reference of type
25     //           'int&' from expression of type 'const int'
26
27     int& mutable_age = dog.getMutableAge();
28
29     // possible:
30     mutable_age = 6;
31 }
```

4.3 Const Objects

```

1 void init_demo() {
2     // if we define a variable const, it must be initialized:
3     // const int number;
4     //           ^ error: uninitialized const number [-fpermissive]
5
6     const int number{0};
7 }
8
9 struct Cat {
10
11     int age;
12
13     void increaseAge() {
14         age++;
15     }
16
17     // Because this function is declared const, it is not possible to modify members
18     // of Cat.
19     // Also, it is not possible to call member functions that are NOT declared const.
20     void growUp() const {
21         // age++;
22         // ^ error: increment of member 'Cat::age' in read-only object
23
24         // increaseAge();
25         // ^ error: no matching function for call to 'Cat::increaseAge() const'
26     }
27
28     // Functions can be overloaded with the const keyword:
29     // The non-const version will be called if the object itself is not const.
30     // The const version will be called if the object itself is const.
31     void notAmbiguous() {}
32
33     void notAmbiguous() const {}
34
35 };
36
37 void const_functions() {
38     Cat cat{};
39     const Cat const_cat{};
40
41     // calling a const function on a non const object is possible:
42     cat.growUp();
43
44     // calling a non-const function on a const object is not possible:
45     // const_cat.increaseAge();
46     //           ^ error: no matching function for call to Cat::increaseAge() const
47
48     // calling functions overloaded with the const keyword depends on whether the
49     // object is const itself:
50     cat.notAmbiguous();
51     const_cat.notAmbiguous();
52 }

```

4.4 Const Values And Pointers

```

1 void value_is_const() {
2     const int value = 5;
3
4     // This pointer points to a variable of type const int. The pointer itself is not
5     // const and can be modified:
6     const int * value_p = &value;
7     value_p++;
8
9     // Changing the value which is pointed by the pointer is not possible:
10    // *value_p = 6;
11    //      ^ error: assignment of read-only location '* value_p'
12
13    // We can define a pointer of type const int eventhough it does not point to a
14    // const variable:
15    int not_const = 5;
16    const int * not_const_p = &not_const;
17
18    // *not_const_p = 6;
19    //      ^ error: assignment of read-only location * not_const_p
20 }
21
22 void pointer_is_const() {
23     int value = 5;
24
25     // This pointer is const and can not be modified:
26     int* const value_p = &value;
27     // value_p++;
28     //      ^ error: increment of read-only variable 'value_p'
29 }
30
31 void pointer_and_value_are_const() {
32     const int value = 5;
33
34     // Here we define a pointer which both points to a const variable and is const
35     // itself:
36     const int* const value_p = &value;
37     // *value_p = 6;
38     //      ^ error: assignment of read-only location '*(const int*)value_p'
39     // value_p++;
40     //      ^ error: increment of read-only variable 'value_p'
41
42     // Also note that putting the asterisk at the right place is critical:
43     // const int const *value_p = &value;
44     // ^ error: duplicate 'const'
45 }

```

5 Streams

5.1 Handling Invalid Input

5.1.1 Escape After Fail

```
1  #include <iostream>
2  #include <sstream>
3
4  int get_age(std::istream& in) {
5      int age{-1};
6
7      while (!in.eof()) {
8          if (in >> age) {
9              return age;
10         }
11
12         std::cout << "in.good(): " << in.good() << std::endl; // in.good(): 0
13         std::cout << "in.fail(): " << in.fail() << std::endl; // in.fail(): 1
14
15         // at this point we have to read the remaining content of in so eof file is
16         // reached and the while loop is escaped:
17         in.clear();
18         std::string line{};
19         std::getline(in, line);
20     }
21
22     return -1;
23 }
24
25 void reading_integers() {
26     std::istringstream in{"24 a 25"};
27
28     int age;
29
30     age = get_age(in);
31     std::cout << "age: " << age << std::endl; // age: 24
32
33     age = get_age(in);
34     std::cout << "age: " << age << std::endl; // age: -1
35
36     std::cout << "in.eof(): " << in.eof() << std::endl; // in.eof(): 1
37 }
```

5.1.2 Continue After Fail

```

1  #include <iostream>
2  #include <sstream>
3
4  int get_age_2(std::istream& in) {
5      int age{-1};
6
7      while (!in.eof()) {
8          if (in >> age) {
9              return age;
10         }
11
12         // read the invalid sequence then continue
13         in.clear();
14         std::string invalid_sequence{};
15         in >> invalid_sequence;
16     }
17
18     return -1;
19 }
20
21 void reading_integers_2() {
22     std::istringstream in{"24 a 25    "};
23
24     int age;
25
26     age = get_age_2(in);
27     std::cout << "age: " << age << std::endl; // age: 24
28
29     age = get_age_2(in);
30     std::cout << "age: " << age << std::endl; // age: 25
31
32     age = get_age_2(in);
33     std::cout << "age: " << age << std::endl; // age: -1
34
35     std::cout << "in.eof(): " << in.eof() << std::endl; // in.eof(): 1
36 }

```

5.2 Manipulators

5.2.1 boolalpha

```
1  #include <iostream>
2
3  void boolalpha_demo() {
4      std::cout << true << std::endl;
5      std::cout << std::boolalpha << true << std::endl;
6      std::cout << true << std::endl;
7      std::cout << 1 << std::endl;
8      std::cout << 0 << std::endl;
9      std::cout << std::noboolalpha << true << std::endl;
10
11     // Output:
12     // 1
13     // true
14     // true
15     // 1
16     // 0
17     // 1
18 }
```

5.2.2 skipws

```
1  #include <iostream>
2  #include <sstream>
3
4  void skipws_demo() {
5      // Only has effect on istream
6      char a, b, c;
7
8      std::istringstream in{" 123"};
9      in >> std::skipws >> a >> b >> c;
10     std::cout << a << b << c << std::endl;
11     // output: 123
12
13     in.seekg(0); // reset stream to read from beginning
14     in >> std::noskipws >> a >> b >> c;
15     std::cout << a << b << c << std::endl;
16     // output: 1
17
18     std::cout << std::skipws << "  abc  def" << std::endl; // no effect
19 }
```

5.2.3 uppercase

```
1  #include <iostream>
2
3  void uppercase_demo() {
4      // makes hexadecimal representations uppercase.
5      // attention: non numeric types will NOT be uppercase!
6      // the following example shows no effect:
7      std::cout << std::uppercase << "abc" << std::endl << std::nouppercase;
8      // Output:
9      // abc
10
11     // Output integers as hex values:
12     std::cout << std::showbase << std::hex;
13     std::cout << std::uppercase << 77 << std::endl;
14     std::cout << std::nouppercase << 77 << std::endl;
15
16     // Output:
17     // 0X4D
18     // 0x4d
19 }
```

5.2.4 oct, hex, dec

```
1  #include <iostream> // std::cout, std::dec, std::hex, std::oct
2
3  void oct_hex_dec() {
4      int n = 29;
5      std::cout << std::dec << n << std::endl;
6      // the setting persists for subsequent calls
7      std::cout << std::hex << n << std::endl;
8      std::cout << n << std::endl;
9      std::cout << std::oct << n << std::endl;
10     std::cout << n << std::endl;
11
12     // output
13     // 29
14     // 1d
15     // 1d
16     // 35
17     // 35
18 }
```

5.2.5 setw

```
1  #include <iostream>
2  #include <iomanip> // std::setw
3
4  void std_cout_width() {
5      // setting the width on an output stream determines the minimum count of characters
6      // the output shall have:
7      std::cout.width(4);
8      std::cout << "ab" << std::endl;
9
10     // this setting will be consumed by the first output to the stream. the next
11     // output will not have the width set anymore:
12     std::cout << "ab" << std::endl;
13
14     //output
15     //  ab
16     // ab
17 }
18
19 void setw() {
20     // we can use setw(n) as a shortcut to width(n):
21     std::cout << std::setw(4) << "ab" << std::endl;
22     std::cout << "ab" << std::endl;
23
24     //output
25     //  ab
26     // ab
27 }
```

5.2.6 left, right, internal

```
1  #include <iostream>
2
3  void left_right_internal() {
4      int n = -1;
5
6      // using std::left or std::right will align the output to the left or right if a
7      // width has been specified:
8      std::cout.width(6);
9      std::cout << std::left << n << std::endl;
10
11     std::cout.width(6);
12     std::cout << std::right << n << std::endl;
13
14     // std::internal can be used for negative numbers.
15     // for non-numerical values it is equivalent to right:
16     std::cout.width(6);
17     std::cout << std::internal << n << std::endl;
18
19     // output
20     // -1
21     //      -1
22     // -      1
23 }
```

5.2.7 setprecision, scientific, fixed

```
1  #include <iostream> // setprecision, scientific, fixed
2
3  double PI = 3.14159;
4
5  void setprecision() {
6
7      // setprecision sets the maximum digits to display.
8      // the number is automatically rounded.
9      std::cout << std::setprecision(5) << PI << std::endl;
10     // setprecision will persist for subsequent outputs to the stream
11     std::cout << PI << std::endl;
12
13     // output
14     // 3.1416
15     // 3.1416
16 }
17
18 void scientific() {
19     std::cout << std::scientific << 10000000.0 << std::endl;
20
21     std::cout << std::setprecision(0);
22
23     std::cout << std::scientific << 10000000.0 << std::endl;
24
25     // output
26     // 1.000000e+07
27     // 1e+07
28 }
29
30 void fixed() {
31     // expands the output to a minimum of digits.
32     std::cout << std::fixed << 1.1 << std::endl;
33     // 1.100000
34
35     std::cout << std::fixed << std::setprecision(3) << 1.0 << std::endl;
36     std::cout << std::fixed << PI << std::endl;
37     // 1.000
38     // 3.142
39 }
```

6 Iterators

6.1 Insert Iterators

6.1.1 Insert iterator

```
1  #include <iterator>
2  #include <vector>
3  #include <algorithm>
4  #include "util.h"
5
6  void insert_iterator() {
7      std::vector<int> incomplete{1, 4, 5};
8      print("incomplete", incomplete);
9
10     std::vector<int>::iterator it = incomplete.begin();
11     it++; // we have to insert after the first entry, so lets increment by one
12     std::insert_iterator<std::vector<int>> insert_iterator{incomplete, it};
13
14     std::vector<int> addition{2, 3}; // this is the content we want to insert
15     std::copy(addition.begin(), addition.end(), insert_iterator);
16
17     print("complete", incomplete);
18
19     // output
20     // incomplete:    {1, 4, 5}
21     // complete:      {1, 2, 3, 4, 5}
22 }
```

6.1.2 Back insert iterator

```
1  #include <iterator>
2  #include <vector>
3  #include <algorithm>
4  #include "util.h"
5
6  void back_insert_iterator() {
7      std::vector<int> incomplete{5, 4};
8      print("incomplete", incomplete);
9
10     std::vector<int> addition{3, 2, 1};
11     std::copy(addition.begin(), addition.end(), std::back_inserter(incomplete));
12
13     print("complete", incomplete);
14
15     // output
16     // incomplete:    {5, 4}
17     // complete:      {5, 4, 3, 2, 1}
18 }
```

6.1.3 Front insert iterator

```
1  #include <iterator>
2  #include <list>
3  #include <algorithm>
4  #include "util.h"
5
6  void front_insert_iterator() {
7      std::list<int> incomplete{4, 5};
8      print("incomplete", incomplete);
9
10     std::vector<int> addition{3, 2, 1};
11     std::copy(addition.begin(), addition.end(), std::front_inserter(incomplete));
12
13     print("complete", incomplete);
14
15     // output
16     // incomplete:  {4, 5}
17     // complete:    {1, 2, 3, 4, 5}
18 }
```

6.2 Stream Iterators

6.2.1 Iterate over strings

```
1  #include <sstream>
2  #include <iterator>
3  #include <iostream>
4
5  void iterate_over_strings() {
6      using namespace std;
7
8      istringstream in{"ill iterate until youre dizzy"};
9
10     copy(istream_iterator<string>{in}, istream_iterator<string>{},
11          ostream_iterator<string>{cout, "-"});
12
13     // output
14     // ill-iterate-until-youre-dizzy
15 }
```

6.2.2 Iterate over ints

```
1  #include <sstream>
2  #include <iterator>
3  #include <iostream>
4
5  void iterate_over_ints() {
6      using namespace std;
7
8      istringstream in{"1 2 3"};
9
10     copy(istream_iterator<int>{in}, istream_iterator<int>{},
11          ostream_iterator<int>{cout, ""});
12
13     // output
14     // 123
15 }
```

6.2.3 Iterate over your own types

```
1  #include <string>
2  #include <algorithm>
3
4  struct Word {
5
6      void read(std::istream& in) {
7          in >> data;
8          std::transform(data.begin(), data.end(), data.begin(), toupper);
9      }
10
11     void write(std::ostream& os) const {
12         os << data;
13     };
14
15     private:
16
17         std::string data;
18
19     };
20
21     std::istream& operator>>(std::istream& in, Word& r) {
22         r.read(in);
23         return in;
24     }
25
26     std::ostream& operator<<(std::ostream& out, Word const& r) {
27         r.write(out);
28         return out;
29     }
30
31
32     void iterate_over_your_own_types() {
33         using namespace std;
34
35         istringstream in{"ill iterate until youre dizzy"};
36
37         copy(istream_iterator<Word>(in), istream_iterator<Word>{},
38             ostream_iterator<Word>(cout, " "));
39
40         // output
41         // ILL ITERATE UNTIL YOURE DIZZY
42     }
```

6.2.4 Count Chars

```

1  #include <iostream>
2  #include <sstream> // istringstream
3  #include <iterator>
4  #include <algorithm>
5
6  void count_all_chars_skip_whitespace() {
7      std::istringstream in{"ab c d !?  &*"};
8
9      // istream_iterator<char> will iterator through every char in the stream, except
10     // white spaces.
11     using Iterator = std::istream_iterator<char>;
12     Iterator begin{in};
13     Iterator end{};
14
15     // we could also use the count_if algorithm and use a lamda that always returns
16     // true. distance is more elegant, however a bit abstract:
17     int count = std::distance(begin, end);
18
19     std::cout << "count: " << count << std::endl; // count: 8
20 }
21
22 void count_all_chars() {
23     std::istringstream in{"ab c d !?  &*"};
24
25     // using a istreambuf_iterator avoids skipping white spaces.
26     using Iterator = std::istreambuf_iterator<char>;
27     Iterator begin{in};
28     Iterator end{};
29
30     int count = std::distance(begin, end);
31
32     std::cout << "count: " << count << std::endl; // count: 13
33 }

```

6.2.5 Count Specific Chars

```

1  void count_specific_chars() {
2      std::istringstream in{"oppa gangnam style"};
3
4      using Iterator = std::istream_iterator<char>;
5      Iterator begin{in};
6      Iterator end{};
7
8      int count = std::count(begin, end, 'a');
9
10     std::cout << "count: " << count << std::endl; // count: 3
11 }

```

6.2.6 Count Words

```
1 void count_words() {  
2     std::istringstream in{"oppa gangnam style :D"};  
3  
4     // this iterator will read char sequences that are separated by white spaces:  
5     using Iterator = std::istream_iterator<std::string>;  
6     Iterator begin{in};  
7     Iterator end{};  
8  
9     int count = std::distance(begin, end);  
10  
11     std::cout << "count: " << count << std::endl; // count: 4  
12 }
```

6.3 Custom Iterators

6.3.1 Line Iterator

```
1  #include <iostream>
2  #include <iterator>
3  #include <algorithm>
4  #include <sstream>
5  #include <boost/operators.hpp>
6
7  namespace {
8      std::istringstream empty{};
9  }
10
11  struct LineIterator : boost::input_iterator_helper<LineIterator, std::string> {
12
13      LineIterator() : in(empty) {
14          in.clear(std::ios_base::eofbit);
15      }
16
17      explicit LineIterator(std::istream &in) : in(in) {
18          read_next_line();
19      }
20
21      bool operator==(LineIterator const &r) const {
22          return !in.good() && !r.in.good();
23      }
24
25      value_type operator*() const {
26          return current_line;
27      }
28
29      LineIterator &operator++() {
30          read_next_line();
31          return *this;
32      }
33
34  private:
35
36      void read_next_line() {
37          getline(in, current_line);
38      }
39      std::istream &in;
40      std::string current_line;
41  };
```

```
1  #include "line_iterator.h"
2
3  void line_iterator_demo() {
4      std::istringstream in{"first line\nsecond line"};
5      LineIterator li{in};
6
7      std::cout << "*li: " << *li << std::endl;
8      li++;
9      std::cout << "*li: " << *li << std::endl;
10
11     // output
12     // *li: first line
13     // *li: second line
14 }
15
16 int main() {
17     line_iterator_demo();
18 }
```

6.3.2 My Ostream Iterator

```
1  #include <iostream>
2  #include <iterator>
3  #include <algorithm>
4
5  struct MyOstreamIterator
6  : std::iterator<std::output_iterator_tag, int> {
7
8      explicit MyOstreamIterator(std::ostream& out) : out{&out} {}
9
10     MyOstreamIterator& operator =(const int value) {
11         *out << value;
12         return *this;
13     }
14
15     MyOstreamIterator& operator *() { return *this; }
16     MyOstreamIterator& operator ++() { return *this; }
17     MyOstreamIterator& operator ++(int) { return *this; }
18
19 private:
20
21     std::ostream* out;
22
23 };
24
25 void my_ostream_iterator_demo() {
26     MyOstreamIterator moi{std::cout};
27
28     moi = 1;
29     moi = 20;
30     moi = 300;
31
32     // output
33     // 120300
34 }
```

6.3.3 Square Iterator

```
1  #include <iterator>
2
3  struct SquareIterator : std::iterator<std::input_iterator_tag, int>
4  {
5      explicit SquareIterator(int start=0) : value{start} {}
6
7      bool operator==(SquareIterator const &r) const {
8          return value == r.value;
9      }
10
11     bool operator!=(SquareIterator const &r) const {
12         return !(*this == r);
13     }
14
15     value_type operator*() const {
16         return value * value;
17     }
18
19     SquareIterator& operator++() {
20         ++value;
21         return *this;
22     }
23
24     SquareIterator operator++(int) {
25         auto old = *this;
26         ++(*this);
27         return old;
28     }
29
30 private:
31     int value;
32
33 };
34
```

```
1  #include "square_iterator.h"
2
3  #include <iostream>
4  #include <algorithm>
5
6  void square_iterator_demo() {
7
8      SquareIterator begin{2};
9      SquareIterator end{6};
10
11     std::for_each(begin, end, [](int sqr) {
12         std::cout << "sqr: " << sqr << std::endl;
13     });
14
15     // output
16     // sqr: 4
17     // sqr: 9
18     // sqr: 16
19     // sqr: 25
20 }
21
22 int main() {
23     square_iterator_demo();
24 }
```

6.3.4 Prime Iterator

```
1  #include <iostream>
2  #include <vector>
3  #include <array>
4  #include <algorithm>
5  #include <boost/iterator/counting_iterator.hpp>
6  #include <boost/iterator/filter_iterator.hpp>
7
8  bool is_divisible(int x, int divisor) {
9      return !(x % divisor);
10 }
11
12 bool is_prime(unsigned x) {
13     if (x % 2 == 0 || x < 3) return false;
14     return std::none_of(boost::make_counting_iterator(2u),
15                         boost::make_counting_iterator(x),
16                         [x](unsigned divisor) { return is_divisible(x, divisor); });
17 }
18
19 void prime_iterator() {
20     auto counting = boost::make_counting_iterator(1);
21     auto countingEnd = boost::make_counting_iterator(40);
22
23     auto prime_iterator = boost::make_filter_iterator(is_prime, counting);
24     auto prime_iterator_end = boost::make_filter_iterator(is_prime, countingEnd);
25
26     std::copy(prime_iterator, prime_iterator_end,
27              std::ostream_iterator<int>(std::cout, " "));
28
29     // output
30     // 3 5 7 11 13 17 19 23 29 31 37
31 }
```

6.4 Iterator Semantics

6.4.1 Iterator

```
1  #include <iostream>
2  #include <iterator>
3  #include <vector>
4
5  void iterator() {
6      std::vector<int> v{0, 1, 2, 3, 4};
7
8      // 0 1 2 3 4
9      // ^ v.begin()
10
11     // 0 1 2 3 4
12     //           ^ v.end()
13
14     // 0 1 2 3 4
15     //           ^ --v.end()
16
17     std::cout << "*v.begin(): " << *v.begin() << std::endl;
18     std::cout << "*v.end(): " << *v.end() << std::endl;
19     std::cout << "*(--v.end()): " << *(--v.end()) << std::endl;
20
21     // output
22     // *v.begin(): 0
23     // *v.end(): (undefined behaviour)
24     // *(--v.end()): 4
25 }
```

6.4.2 Reverse Iterator

```

1  void reverse_iterator() {
2      std::vector<int> v{10, 11, 12, 13, 14};
3
4      // 10 11 12 13 14
5      //           ^ v.rbegin()
6
7      // 10 11 12 13 14
8      //           ^ v.rbegin() + 1
9
10     // 10 11 12 13 14
11     // ^ v.rend()
12
13     // 10 11 12 13 14
14     // ^ v.rend() + 1
15
16     // 10 11 12 13 14
17     // ^ v.rend() - 1
18
19     std::vector<int>::reverse_iterator rbegin = v.rbegin();
20     std::vector<int>::reverse_iterator rend = v.rend();
21
22     std::cout << "*rbegin:      " << *rbegin      << std::endl;
23     std::cout << "*(rbegin + 1): " << *(rbegin + 1) << std::endl;
24     std::cout << "*rend:      " << *rend      << std::endl;
25     std::cout << "*(rend + 1):  " << *(rend + 1)  << std::endl;
26     std::cout << "*(rend - 1):  " << *(rend - 1)  << std::endl;
27
28     // output
29     // *rbegin:      14
30     // *(rbegin + 1): 13
31     // *rend:      (undefined behaviour)
32     // *(rend + 1):  (undefined behaviour)
33     // *(rend - 1):  10
34 }

```

7 Containers

7.1 Vector

7.1.1 Initialization

```
1  #include <vector>
2
3  void initialization() {
4      // with initializer list
5      std::vector<int> ve1{1, 2, 3};
6
7      // with capacity
8      // elements are zero-initialized
9      std::vector<int> ve2(3);
10 }
```

7.1.2 Accessing Elements

```
1  #include <vector>
2
3  void accessing_elements() {
4      std::vector<int> ve1(3);
5
6      // we can use at() or the []-operator to access elements in a vector
7      ve1.at(1);
8      ve1[1];
9
10     // the difference lies in the behaviour of when trying to access an element with
11     // an index that is out of range:
12
13     // the following code would throw 'std::out_of_range'
14     // ve1.at(3);
15
16     // the following code is undefined behaviour and does not throw an exception
17     // ve1[3];
18 }
```

7.1.3 Assigning Values

```
1  #include <iostream>
2  #include <vector>
3
4  void assigning_values() {
5      std::vector<int> vel{3, 2, 1};
6
7      vel[1] = 10; // OK
8
9      // the following line tries to access an invalid index and is undefined behaviour
10     // vel[3] = 30;
11 }
12
13 void using_front() {
14     std::vector<int> vel{3, 2, 1};
15
16     // because front() returns a reference, we can also assign values to it
17     std::cout << vel.front() << std::endl;
18     vel.front() = 4;
19     std::cout << vel.front() << std::endl;
20     // Output:
21     // 3
22     // 4
23
24     // there is also a const version of front();
25     const int f = vel.front();
26
27     // the opposite to front() is back()
28 }
```

7.1.4 Appending Values

```
1  #include <vector>
2
3  void appending_values() {
4      std::vector<int> vel(3);
5      vel.push_back(0);
6      vel.push_back(1);
7      vel.push_back(2);
8
9      // At this point, the vector will automatically resize.
10     vel.push_back(3);
11 }
```

7.1.5 Iterating Elements

```
1  #include <vector>
2
3  void iteration_with_read_only_access() {
4      std::vector<int> v{1, 2, 3};
5      for (auto const i : v) {
6          std::cout << i << " ";
7
8          // i++;
9          // ^ error: increment of read-only variable i
10     }
11     std::cout << '\n';
12
13     // Output:
14     // 1 2 3
15 }
16
17 void iteration_with_reference() {
18     std::vector<int> v{1, 2, 3};
19     for (auto &i : v) {
20         i++;
21         std::cout << i << " ";
22     }
23     std::cout << '\n';
24
25     // Output
26     // 2 3 4
27 }
```

8 Algorithms

8.1 Modifying

8.1.1 Copy

```
1  #include "util.h"
2
3  #include <algorithm>
4  #include <vector>
5
6  void copy_directly() {
7      std::vector<int> src{1, 2, 3, 4};
8
9      // note that the size of dest must be at least the size of src to avoid
10     // undefined behaviour:
11     std::vector<int> dest(4);
12
13     std::copy(src.begin(), src.end(), dest.begin());
14
15     print("src", src);
16     print("dest", dest);
17
18     // output
19     // src:  1 2 3 4
20     // dest: 1 2 3 4
21 }
22
23 void copy_with_back_inserter() {
24     std::vector<int> src{5, 6, 7, 8};
25     std::vector<int> dest{};
26
27     // using a back_inserter here makes sure that dest is being resized if required:
28     std::copy(src.begin(), src.end(), std::back_inserter(dest));
29
30     print("src", src);
31     print("dest", dest);
32
33     // output
34     // src:  5 6 7 8
35     // dest: 5 6 7 8
36 }
```

8.1.2 Move

```
1  #include "util.h"
2
3  #include <algorithm>
4  #include <vector>
5  #include <string>
6
7  void move_integers() {
8      std::vector<int> numbers{1, 2, 3, 4};
9      std::vector<int> copied_numbers;
10
11      std::move(numbers.begin(), numbers.end(), std::back_inserter(copied_numbers));
12
13      print("numbers", numbers);
14      print("copied_numbers", copied_numbers);
15
16      // moving the elements depends on the move semantics of the given type.
17      // in this case, we are moving ints. the move semantics of ints keeps the original
18      // values in the vector:
19      //
20      // numbers:          1 2 3 4
21      // copied_numbers:    1 2 3 4
22  }
23
24  void move_strings() {
25      std::vector<std::string> strings{"a", "b", "c"};
26      std::vector<std::string> copied_strings;
27
28      std::move(strings.begin(), strings.end(), std::back_inserter(copied_strings));
29
30      print("strings", strings);
31      print("copied_strings", copied_strings);
32
33      // move semantics of string actually moves the elements from one vector to the
34      // other:
35      //
36      // strings:           "" "" ""
37      // copied_strings:    "a" "b" "c"
38  }
```

8.1.3 Transform

```
1  #include "util.h"
2
3  #include <algorithm>
4  #include <vector>
5
6  void unary_transform() {
7      std::vector<int> numbers{1, 2, 3, 4};
8
9      std::transform(numbers.begin(), numbers.end(),
10                     numbers.begin(),
11                     [](int n) { return n + 10; });
12
13     print("numbers", numbers);
14
15     // output
16     // 11 12 13 14
17 }
18
19 void binary_transform() {
20     std::vector<int> numbers1{1, 2, 3, 4};
21     std::vector<int> numbers2{4, 3, 2, 1};
22
23     std::transform(numbers1.begin(), numbers1.end(),
24                    numbers2.begin(), numbers2.begin(),
25                    [](int n1, int n2) { return n1 + n2; });
26
27     print("numbers2", numbers2);
28
29     // output
30     // 5 5 5 5
31 }
```

8.2 Partitioning

8.2.1 Partitioning

```

1  #include "util.h"
2
3  #include <algorithm>
4  #include <vector>
5
6  void partition() {
7      std::vector<int> v{1, 2, 3, 4, 5, 6, 7, 8};
8
9      print("before", v);
10
11     // puts all the even numbers into the first half of the vector and all the odd
12     // numbers into the second half
13     std::partition(v.begin(), v.end(), [](int n) { return !(n % 2); });
14
15     print("after", v);
16
17     // output
18     // before: 1 2 3 4 5 6 7 8
19     // after:  8 2 6 4 5 3 7 1
20 }
21
22 // makes sure the relative order of the elements is preserved
23 void stable_partition() {
24     std::vector<int> v{1, 2, 3, 4, 5, 6, 7, 8};
25
26     print("before", v);
27
28     std::stable_partition(v.begin(), v.end(), [](int n) { return !(n % 2); });
29
30     print("after", v);
31
32     // output
33     // before: 1 2 3 4 5 6 7 8
34     // after:  2 4 6 8 1 3 5 7
35 }
36
37 // besides these there are the following algorithms handling partitioning:
38 //
39 // is_partitioned(v.begin(), v.end(), unary_predicate)
40 //
41 // partition_copy(v.begin(), v.end(), dest1.begin(), dest2.begin(), unary_predicate)
42 //
43 // forward_iterator = partition_point(v.begin(), v.end(), unary_predicate)

```

8.3 Numeric

8.3.1 Accumulate

```

1  void accumulate_demo() {
2      std::vector<int> v{1, 2, 3, 4, 5};
3
4      int sum = std::accumulate(v.begin(), v.end(), 10);
5      std::cout << "sum: " << sum << std::endl;
6
7      int fac = std::accumulate(v.begin(), v.end(), 1, std::multiplies<int>());
8      //                                     ^ multiplication with 0 = 0!
9      std::cout << "fac: " << fac << std::endl;
10
11     // output
12     // sum: 25
13     // fac: 120
14 }

```

8.3.2 Adjacent Difference

```

1  void adjacent_difference_demo() {
2      std::vector<int> v{1, 2, 3, 4, 5, 6, 7, 8, 9};
3
4      std::vector<int> d1{};
5      std::vector<int> d2{};
6
7      std::adjacent_difference(v.begin(), v.end(), std::back_inserter(d1));
8      std::adjacent_difference(v.begin(), v.end(), std::back_inserter(d2),
9                               std::plus<int>());
10
11     std::copy(d1.begin(), d1.end(), std::ostream_iterator<int>(std::cout, " "));
12     std::cout << std::endl;
13     std::copy(d2.begin(), d2.end(), std::ostream_iterator<int>(std::cout, " "));
14
15     // output
16     // 1 1 1 1 1 1 1 1 1
17     // 1 3 5 7 9 11 13 15 17
18 }

```

8.3.3 Inner Product

```
1 void inner_product_demo() {
2     std::vector<int> v1{1, 2, 3};
3     //           *   *   *
4     std::vector<int> v2{4, 5, 6};
5     //           4 + 10 + 18 = 32
6
7     int res = std::inner_product(v1.begin(), v1.end(), v2.begin(), 0);
8
9     std::cout << "res: " << res << std::endl;
10
11     // output
12     // res: 32
13 }
```

8.3.4 Partial Sum

```
1 void partial_sum_demo() {
2     // 1 = 1
3     // 1 + 2 = 3
4     // 1 + 2 + 3 = 6
5     // 1 + 2 + 3 + 4 = 10
6     // 1 + 2 + 3 + 4 + 5 = 15
7
8     std::vector<int> v{1, 2, 3, 4, 5};
9     std::vector<int> r(5);
10
11     std::partial_sum(v.begin(), v.end(), r.begin());
12
13     std::copy(r.begin(), r.end(), std::ostream_iterator<int>(std::cout, " "));
14
15     // output
16     // 1 3 6 10 15
17 }
```

8.4 Misc

8.4.1 Fill Vector With Squares

```
1  #include <iostream>
2  #include <iterator>
3  #include <vector>
4  #include <algorithm>
5  #include <numeric>
6
7  void fill_vector_with_squares() {
8      // create vector containing only ones
9      std::vector<int> v(10, 1);
10
11     // this will make the vector contain the values from 1 to 10
12     std::partial_sum(v.begin(), v.end(), v.begin());
13
14     // square every item in v
15     std::transform(v.begin(), v.end(), v.begin(), [](int i) { return i * i; });
16
17     // print it
18     std::copy(v.begin(), v.end(), std::ostream_iterator<int>(std::cout, " "));
19
20     // output
21     // 1 4 9 16 25 36 49 64 81 100
22 }
```

8.4.2 Is Palindrome

```
1  #include <iostream>
2  #include <cctype>
3  #include <iterator>
4  #include <string>
5  #include <algorithm>
6
7  bool are_equal(char a, char b) {
8      return tolower(a) == tolower(b);
9  }
10
11 bool is_palindrome(std::string word) {
12     return std::equal(word.cbegin(),
13                       word.cbegin() + word.length() / 2,
14                       word.crbegin(),
15                       are_equal);
16 }
17
18 void is_palindrome_demo() {
19     std::cout << std::boolalpha;
20     std::cout << is_palindrome("Rihanna") << std::endl;
21     std::cout << is_palindrome("Anna") << std::endl;
22     std::cout << is_palindrome("abcba") << std::endl;
23     std::cout << is_palindrome("trugtimeinesohellehoseniemitgurt") << std::endl;
24
25     // output
26     // false
27     // true
28     // true
29     // true
30 }
```

8.4.3 Multiplication Table

```

1  #include <iomanip> // std::setw
2
3  void multiplication_table() {
4      std::vector<int> numbers(20, 1);
5
6      // generate a list containing the values from 1 to 20.
7      // could also be done with iota which would be much easier lol.
8      std::transform(numbers.begin(), numbers.end()-1, numbers.begin()+1, numbers.begin()+1,
9                      [](int a, int b) { return a + b; });
10
11     std::for_each(numbers.begin(), numbers.end(), [&numbers](int i) {
12         std::for_each(numbers.begin(), numbers.end(), [&i](int j) {
13             std::cout << std::setw(4) << i * j;
14         });
15         std::cout << '\n';
16     });
17
18     // output
19     //
20     //  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
21     //  2  4  6  8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40
22     //  3  6  9 12 15 18 21 24 27 30 33 36 39 42 45 48 51 54 57 60
23     //  4  8 12 16 20 24 28 32 36 40 44 48 52 56 60 64 68 72 76 80
24     //  5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95 100
25     //  6 12 18 24 30 36 42 48 54 60 66 72 78 84 90 96 102 108 114 120
26     //  7 14 21 28 35 42 49 56 63 70 77 84 91 98 105 112 119 126 133 140
27     //  8 16 24 32 40 48 56 64 72 80 88 96 104 112 120 128 136 144 152 160
28     //  9 18 27 36 45 54 63 72 81 90 99 108 117 126 135 144 153 162 171 180
29     // 10 20 30 40 50 60 70 80 90 100 110 120 130 140 150 160 170 180 190 200
30     // 11 22 33 44 55 66 77 88 99 110 121 132 143 154 165 176 187 198 209 220
31     // 12 24 36 48 60 72 84 96 108 120 132 144 156 168 180 192 204 216 228 240
32     // 13 26 39 52 65 78 91 104 117 130 143 156 169 182 195 208 221 234 247 260
33     // 14 28 42 56 70 84 98 112 126 140 154 168 182 196 210 224 238 252 266 280
34     // 15 30 45 60 75 90 105 120 135 150 165 180 195 210 225 240 255 270 285 300
35     // 16 32 48 64 80 96 112 128 144 160 176 192 208 224 240 256 272 288 304 320
36     // 17 34 51 68 85 102 119 136 153 170 187 204 221 238 255 272 289 306 323 340
37     // 18 36 54 72 90 108 126 144 162 180 198 216 234 252 270 288 306 324 342 360
38     // 19 38 57 76 95 114 133 152 171 190 209 228 247 266 285 304 323 342 361 380
39     // 20 40 60 80 100 120 140 160 180 200 220 240 260 280 300 320 340 360 380 400
40 }

```

8.4.4 Sum Numbers

```
1  #include <numeric>    // accumulate
2  #include <sstream>
3
4  void sum_integers() {
5      std::istringstream in{"1 2 3"};
6
7      using Iterator = std::istream_iterator<int>;
8      Iterator begin{in};
9      Iterator end{};
10
11     int sum = 0;
12
13     sum = std::accumulate(begin, end, 0);
14     std::cout << "sum: " << sum << std::endl; // sum: 6
15 }
16
17 void sum_floats() {
18     std::istringstream in{"1.1 2.2 3.3"};
19
20     using Iterator = std::istream_iterator<double>;
21     Iterator begin{in};
22     Iterator end{};
23
24     double sum = 0;
25
26     sum = std::accumulate(begin, end, .0);
27     //                                     ^--- we have to pass in a double, otherwise result
28     //                                     will be integer!
29     std::cout << "sum: " << sum << std::endl; // sum: 6.6
30 }
```

8.4.5 Word List

```
1  #include <algorithm>
2  #include <iostream>
3  #include <sstream>
4  #include <iterator>
5  #include <set>
6
7  struct Comparator {
8      bool operator() (const std::string& a, const std::string& b) const {
9          return std::lexicographical_compare(a.begin(), a.end(), b.begin(), b.end(),
10             [](char x, char y) {return tolower(x) < tolower(y);});
11      }
12 };
13
14 void word_list() {
15     using Iterator = std::istream_iterator<std::string>;
16
17     std::istringstream in{"this is a test this is A TEST THIS IS"};
18
19     std::set<std::string, Comparator> list(Iterator{in}, Iterator{});
20
21     std::copy(list.begin(), list.end(), std::ostream_iterator<std::string>(std::cout, "\n"));
22
23     // output (note how set automatically sorts its content)
24     // a
25     // is
26     // test
27     // this
28 }
```

9 Bind

9.1 Examples

```
1  #include <functional> // std::bind, std::placeholders
2  #include <cmath>      // sqrt
3  #include <iostream>
4
5  using namespace std::placeholders;
6
7  // x + y
8  void example_1() {
9      auto e = std::bind(std::plus<double>(), _1, _2);
10
11     std::cout << "1 + 2 = " << e(1, 2) << std::endl;
12
13     // output
14     // 1 + 2 = 3
15 }
16
17 // (2 * x) - (y / 3)
18 void example_2() {
19     auto e = std::bind(std::minus<double>(),
20                       std::bind(std::multiplies<double>(), 2, _1),
21                       std::bind(std::divides<double>(), _2, 3));
22
23     std::cout << "(2 * 2) - (6 / 3) = " << e(2, 6) << std::endl;
24
25     // output
26     // (2 * 2) - (6 / 3) = 2
27 }
28
29 // (x * x) % y
30 void example_3() {
31     auto e = std::bind(std::modulus<int>(),
32                       std::bind(std::multiplies<double>(), _1, _1), _2);
33
34     std::cout << "(4 * 4) % 5 = " << e(4, 5) << std::endl;
35
36     // output
37     // (4 * 4) % 5 = 1
38 }
39
40 // sqrt(x * x)
41 void example_4() {
42     auto e = std::bind(sqrt, std::bind(std::multiplies<double>(), _1, _1));
43
44     std::cout << "sqrt(4 * 4) = " << e(4) << std::endl;
45
46     // output
47     // sqrt(4 * 4) = 4
48 }
```

10 Templates

10.1 Function Templates

10.1.1 median

```
1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4
5  template<typename T>
6  T median(T a, T b, T c) {
7      std::vector<T> list{a, b, c};
8      std::sort(list.begin(), list.end());
9      return list.at(1);
10 }
11
12 void median_demo() {
13     std::cout << "median(2, 3, 1): " << median(2, 3, 1) << std::endl;
14
15     // output
16     // median(2, 3, 1): 2
17 }
```

10.1.2 rotate 3 arguments

```
1  template<typename T>
2  void rotate_3_arguments(T& a, T& b, T& c) {
3      T tmp = a;
4      a = b;
5      b = c;
6      c = tmp;
7  }
8
9  void rotate_3_arguments_demo() {
10     int a{0}, b{1}, c{2};
11     rotate_3_arguments(a, b, c);
12     std::cout << "a: " << a << std::endl;
13     std::cout << "b: " << b << std::endl;
14     std::cout << "c: " << c << std::endl;
15
16     // output
17     // a: 1
18     // b: 2
19     // c: 0
20 }
```

10.1.3 read line

```
1  #include <string>
2  #include <sstream>
3
4  void read_line(std::istream& in) {
5      std::cout << "doing absolutely nothing!" << std::endl;
6  }
7
8  template<typename HEAD, typename...ARGS>
9  void read_line(std::istream& in, HEAD& head, ARGS &...args) {
10     in >> head;
11     read_line(in, args...);
12 }
13
14 void read_line_demo() {
15     int a, b, c;
16     std::string input{"1 2 3"};
17     std::istringstream in{input};
18
19     read_line(in, a, b, c);
20
21     std::cout << "a: " << a << std::endl;
22     std::cout << "b: " << b << std::endl;
23     std::cout << "c: " << c << std::endl;
24
25     // output
26     // a: 1
27     // b: 2
28     // c: 3
29 }
30
31 // if (sizeof...(args)) {
32 //     read_line(in, args...);
33 // }
```

10.1.4 read line 2

```
1  #include <iostream>
2  #include <sstream>
3
4  void read_line_2(std::istream& in, std::string& str) {
5      getline(in, str);
6  }
7
8  template<typename HEAD, typename...ARGS>
9  void read_line_2(std::istream& in, HEAD& head, ARGS &...args) {
10     in >> head;
11     read_line_2(in, args...);
12 }
13
14 void read_line_2_demo() {
15     std::istringstream in{"1 some rest 123 \n 2 the rest\n"};
16     std::string rest{};
17     int first{};
18
19     read_line_2(in, first, rest);
20
21     std::cout << first << std::endl;
22     std::cout << rest << std::endl;
23
24     read_line_2(in, first, rest);
25
26     std::cout << first << std::endl;
27     std::cout << rest << std::endl;
28
29     // output
30     // 1
31     //  some rest 123
32     // 2
33     //  the rest
34 }
```

10.1.5 calling unimplemented functions

```
1 struct Lea {
2     void call() {
3     }
4 };
5
6 struct Julia {
7 };
8
9 template<typename T>
10 void call_me(T girl) {
11     girl.call();
12 }
13
14 void calling_unimplemented_functions() {
15     Lea lea{};
16     Julia julia{};
17
18     // will call call() on lea
19     call_me(lea);
20
21     // calling call_me() with julia will cause a compiler error:
22     //
23     // call_me(julia);
24     //           ^ error: struct Julia has no member named call
25 }
```

10.1.6 int and double dilemma

```
1 template<typename T>
2 void take_two(T one, T two) {
3 }
4
5 void int_and_double_dilemma() {
6     // the argument types to the following call to take_two() will be recognized by the
7     // compiler as an integer followed by a double:
8     //
9     // take_two(1, 2.2);
10    //           ^ error: no matching function for call to take_two(int, double)
11
12    // here are two solutions how to avoid this:
13
14    // with static_cast
15    take_two(static_cast<double>(1), 2.2);
16
17    // by specifying template argument
18    take_two<double>(1, 2.2);
19 }
```

10.1.7 string fallacy

```
1  #include <string>
2
3  template<typename T>
4  void compare(T const& one, T const& two) {
5  }
6
7  void string_fallacy() {
8      // the following does not compile. at first it looks like we are passing strings
9      // to the function. however, the compiler will recognize the arguments as char
10     // arrays:
11     //
12     // compare("shorter", "longer");
13     //           ^ error: no matching function for call to
14     //           compare(const char [8], const char [7])
15
16     // this works because both arguments have the same size:
17     compare("one", "two");
18
19     // if we want the arguments to be passed as strings, we can do the following:
20     compare<std::string>("shorter", "longer");
21 }
```

10.2 Class Templates

10.2.1 class template specialization

```

1  // this example is currently incomplete. remove it?
2
3  #include <iostream>
4  #include <string>
5
6  template<typename T>
7  struct Versatile;
8
9  // we can specialize a templated class with the following syntax:
10 template<>
11 struct Versatile<std::string> {
12     void print() {
13         // std::cout << "string: " << data << std::endl;
14         //                                     ^ error: data was not declared in this scope %bamprog%
15     }
16 };
17
18 template<typename T>
19 struct Versatile {
20
21     Versatile(T t) : data{t} {}
22
23     void print() {
24         std::cout << "number: " << data << std::endl;
25     }
26
27     protected: T data;
28
29 };
30
31 void class_template_specialization() {
32     Versatile<int> vint{1};
33     vint.print();
34
35     Versatile<double> vdou{1.1};
36     vdou.print();
37
38     Versatile<std::string> vstr{};
39     vstr.print();
40
41     // output
42     // number: 1
43     // number: 1.1
44     // string: str
45 }

```

10.2.2 prohibited construction

```

1  #include <string>
2
3  // we can prohibit the creation of templated types with partial specializations by
4  // deleting the destructors in the specialized type. The compiler prohibits the
5  // creation of objects of types which dont have a destructor.
6  // the following two specialiations will prohibit creating instances of
7  // Restrictive<char> and Restrictive<T*>.
8
9  template<typename T>
10 struct Restrictive {
11 };
12
13 // prohibit Restrictive<char>
14 template<>
15 struct Restrictive<char> {
16     ~Restrictive() = delete;
17 };
18
19 // prohibit Restrictive<T*>
20 template<typename T>
21 struct Restrictive<T*> {
22     ~Restrictive() = delete;
23 };
24
25 void prohibited_construction() {
26     Restrictive<int> ints{};
27     Restrictive<std::string> strings{};
28
29     // Restrictive<char> chars{};
30     //                               ^ error: use of deleted function
31
32     // Restrictive<int*> pints{};
33     //                               ^ error: use of deleted function
34
35     // Restrictive<char*> pchars{};
36     //                               ^ error: use of deleted function
37 }

```

10.2.3 vector delegator

```
1  #include <vector>
2  #include <string>
3  #include <algorithm>
4  #include <iostream>
5  #include <iterator>
6
7  template<typename T>
8  struct VectorDelegator {
9
10     template<typename ITER>
11     VectorDelegator(ITER a, ITER b) : data(a, b) {}
12
13 private:
14
15     std::vector<int> data{};
16
17 };
18
19 void vector_delegator() {
20     std::vector<int> v{1,2,3};
21
22     VectorDelegator<int> vd{v.begin(), v.end()};
23 }
```

11 References And Pointers

11.1 Basic Examples

11.1.1 Reference on int

```
1  #include <iostream>
2
3  void reference_on_int() {
4      int j = 5;
5      int& r = j;
6
7      std::cout << "j: " << j << std::endl;
8      std::cout << "r: " << r << std::endl;
9
10     j = 6;
11
12     std::cout << "j: " << j << std::endl;
13     std::cout << "r: " << r << std::endl;
14
15     r = 7;
16
17     std::cout << "j: " << j << std::endl;
18     std::cout << "r: " << r << std::endl;
19
20     // output
21     // j: 5
22     // r: 5
23     // j: 6
24     // r: 6
25     // j: 7
26     // r: 7
27
28 }
```

11.1.2 Reference on cat

```
1  #include <iostream>
2  #include <string>
3
4  class Cat {
5      std::string name;
6
7  public:
8      Cat(std::string name) : name{name} {}
9
10     void setName(std::string n) {
11         name = n;
12     }
13
14     std::string getName() {
15         return name;
16     }
17 };
18
19 void reference_on_cat() {
20     Cat cat{"Hector-Pascal"};
21     Cat& same_cat = cat;
22     Cat other_cat = cat;
23
24     std::cout << "cat.getName(): " << cat.getName() << std::endl;
25     std::cout << "same_cat.getName(): " << same_cat.getName() << std::endl;
26     std::cout << "other_cat.getName(): " << other_cat.getName() << std::endl;
27
28     same_cat.setName("Luftdruck");
29
30     std::cout << "cat.getName(): " << cat.getName() << std::endl;
31     std::cout << "same_cat.getName(): " << same_cat.getName() << std::endl;
32     std::cout << "other_cat.getName(): " << other_cat.getName() << std::endl;
33
34     // output
35     // cat.getName():      Hector-Pascal
36     // same_cat.getName():  Hector-Pascal
37     // other_cat.getName(): Hector-Pascal
38     // cat.getName():      Luftdruck
39     // same_cat.getName():  Luftdruck
40     // other_cat.getName(): Hector-Pascal
41 }
```

11.2 Reference vs. Pointer

```
1  #include <iostream>
2  #include <sstream>
3  #include <string>
4
5  struct Exp1 {
6      Exp1(std::ostream& out) : out(out) {}
7
8      void write(std::string something) {
9          out << something << std::endl;
10     }
11
12     std::ostream& out;
13 };
14
15 struct Exp2 {
16     Exp2(std::ostream& out) : out(&out) {}
17     //                                     ^--- address of out
18     //
19     //                                     ^--- reference to out
20
21     void write(std::string something) {
22         *out << something << std::endl;
23     }
24
25     std::ostream* out;
26 };
27
28 void saving_ref_or_p_as_member() {
29     Exp1 exp_1{std::cout};
30     Exp2 exp_2{std::cout};
31
32     exp_1.write("this comes from exp_1");
33     exp_2.write("this comes from exp_2");
34
35     // output
36     // this comes from exp_1
37     // this comes from exp_2
38 }
```

11.3 C++ vs. Java

11.3.1 Reference in C++

```
1  #include <iostream>
2  #include <string>
3
4  void change(std::string& something) {
5      something = "another string";
6  }
7
8  void reference_in_cpp() {
9      std::string str = "a string";
10
11     std::cout << "str: " << str << std::endl;
12
13     change(str);
14
15     std::cout << "str: " << str << std::endl;
16
17     // output
18     // str: a string
19     // str: another string
20 }
```

11.3.2 Reference in Java

```
1  public class ReferenceInJava {
2
3      public static void change(String str) {
4          str = "another string";
5      }
6
7      public static void main(String[] args) {
8          String str = "a string";
9
10         System.out.println(str);
11
12         change(str);
13
14         System.out.println(str);
15
16         // output
17         // a string
18         // a string
19     }
20
21 }
```

11.4 Dynamic Heap Memory Management

11.4.1 Unique Pointer

```
1  #include <iostream>
2  #include <memory>
3
4  std::unique_ptr<int> create_on_heap(int i) {
5      return std::unique_ptr<int>{new int{i}};
6  }
7
8  void unique_ptr_demo() {
9      // a unique pointer can only exist once. we can not copy it. we can only move the
10     // unique pointer to another variable, making the original variable invalid.
11
12     auto p = create_on_heap(10);
13
14     std::cout << std::boolalpha;
15     std::cout << "is p valid? " << static_cast<bool>(p) << std::endl;
16
17     auto j = std::move(p);
18     // not possible:
19     // auto j = p;
20
21     std::cout << "is p valid? " << static_cast<bool>(p) << std::endl;
22
23     // output:
24     // is p valid? true
25     // is p valid? false
26 }
```

11.4.2 Shared Pointer

```
1  #include <iostream>
2  #include <memory>
3
4  struct A {
5      A(int n) : n{n} {}
6      int n;
7  };
8
9  void shared_ptr_demo() {
10     // shared pointers can be copied:
11
12     auto i = std::make_shared<A>(123);
13
14     auto j = i;
15
16     std::cout << "i->n: " << i->n << std::endl;
17     std::cout << "(*j).n: " << (*j).n << std::endl;
18
19     // output
20     // i->n: 123
21     // (*j).n: 123
22 }
```

12 Compile Time Calculation

12.1 constexpr

```
1  #include <vector>
2  #include <iostream>
3
4  constexpr int add_at_compile_time(int a, int b) {
5      return a + b;
6  }
7
8  void constexpr_demo() {
9      // to c will be assigned the value 3 at compile time, because the called function
10     // add_at_compile_time() is a constexpr.
11     // writing 'int c = 1 + 2;' would be equivalent.
12     int c = add_at_compile_time(1, 2);
13 }
```

12.2 static_assert

```
1  void static_assert_demo() {
2      // with static assert we can do assertions at compile time:
3      const int i = 3;
4      static_assert(i >= 3, "nooope");
5
6      // if the assertion fails a compilation error is raised:
7      // static_assert(i < 3, "nooope");
8      // ^ error: static assertion failed: nooope
9
10     // static_assert only works with constant conditions. working with the non-const
11     // variable j will fail:
12     int j = 3;
13     // static_assert(j, "");
14     // ^ error: non-constant condition for static assertion
15     // ^ error: the value of j is not usable in a constant expression
16 }
```

12.3 User Defined Literals

```
1 constexpr double operator"" _cm(long double x) {
2     return x / 100.;
3 }
4
5 constexpr double operator"" _cm(unsigned long long x) {
6     return x / 100.;
7 }
8
9 constexpr double operator"" _m(long double x) {
10    return x;
11 }
12
13 constexpr double operator"" _m(unsigned long long x) {
14    return x;
15 }
16
17 constexpr double operator"" _km(long double x) {
18    return x * 1000;
19 }
20
21 constexpr double operator"" _km(unsigned long long x) {
22    return x * 1000;
23 }
24
25 void user_defined_literals() {
26     std::cout << "1_cm: " << 1_cm << " meters" << std::endl;
27     std::cout << "1_km: " << 1_km << " meters" << std::endl;
28
29     // output
30     // 1_cm: 0.01 meters
31     // 1_km: 1000 meters
32 }
```

12.4 Ring5

```

1  struct Ring5 {
2      explicit constexpr
3      Ring5(unsigned x=0u) : val{ x % 5 } {}
4
5      constexpr unsigned value() const { return val; }
6
7      constexpr operator unsigned() const { return val; }
8
9      constexpr bool operator==(Ring5 const &r) const {
10         return val == r.val;
11     }
12
13     constexpr bool operator!=(Ring5 const &r) const {
14         return !(*this == r);
15     }
16
17     // this function can not be constexpr because it changes internal state.
18     Ring5 operator+=(Ring5 const &r) {
19         val = (val + r.value())%5;
20         // this error would be raised if we tried to use constexpr:
21         // ^ error: assignment of member val in read-only object
22         return *this;
23     }
24
25     Ring5 operator*=(Ring5 const&r) {
26         val = (val * r.value())%5;
27         return *this;
28     }
29
30     constexpr Ring5 operator+(Ring5 const &r) const {
31         return Ring5{val+r.val};
32     }
33
34     constexpr Ring5 operator*(Ring5 const &r) const {
35         return Ring5{val*r.val};
36     }
37
38 private:
39     unsigned val;
40 };

```

13 Good To Know

13.1 Default floating point type

```
1 // the default floating point type is double, not float
2 auto this_is_a_double = 3.141;
```

13.2 Assigning floating point to int

```
1 #include <iostream>
2
3 void assigning_floating_point_to_int() {
4     // if we assign a floating point to a variable of type integer, it will
5     // automatically be casted.
6     int autoconverted_to_int = 7.5;
7
8     std::cout << autoconverted_to_int << std::endl;
9
10    // output:
11    // 7
12 }
```

13.3 Bool is an integer

```
1 #include <iostream>
2
3 void bool_is_an_integer() {
4     std::cout << "5 + true:  " << 5 + true << std::endl;
5     std::cout << "5 + 1:      " << 5 + 1 << std::endl;
6     std::cout << "6 + false: " << 6 + false << std::endl;
7     std::cout << "6 - 0:      " << 6 - 0 << std::endl;
8
9     // output
10    // 5 + true:  6
11    // 5 + 1:      6
12    // 6 + false: 6
13    // 6 - 0:      6
14 }
```

13.4 Initializing variables

```
1 void initializing_variables() {
2     int a{1};           // initialized with 1
3     int b{};           // default initialization (in case of int zero)
4     int c;              // undefined behaviour
5     static int d;       // zero initialized
6
7     std::cout << "a: " << a << std::endl;
8     std::cout << "b: " << b << std::endl;
9     std::cout << "c: " << c << std::endl;
10    std::cout << "d: " << d << std::endl;
11
12    // output
13    // a: 1
14    // b: 0
15    // c: 7
16    // d: 0
17 }
```

13.5 Floating points cant be unsigned

```
1 // unsigned double ud = 1.1;
2 //           ^ error: signed or unsigned invalid for ud
3 // unsigned float  uf = 1.1;
4 //           ^ error: signed or unsigned invalid for uf
5
6 unsigned int      ui = 1;
```

13.6 Literals

```

1 void chars() {
2     char a;
3     a = 'a';
4     a = '\n';
5     a = '\x0a';
6 }
7
8 void integers() {
9     int i = 1;
10    long l = 1L;
11    long long ll = 1LL;
12 }
13
14 void unsigned_integers() {
15     unsigned int ui = 1u;
16     unsigned long ul = 1ul;
17     unsigned long long ull = 1ull;
18 }
19
20 void octal_hex_full() {
21     int octal = 020;
22     int hex = 0x1f;
23     long long full = 0XFULL;
24 }
25
26 void floating_points() {
27     float f = 0.1f;
28     double d1 = .33;
29     double d2 = 1e9;
30     long double d3 = 42.E-12L;
31     long double d4 = .31;
32 }
33
34 void char_array() {
35     char a[] = "hello"; // char[6] {'h', 'e', 'l', 'l', 'o', '\0'}
36                       // '\0' terminates the string
37 }

```

13.7 Weird string syntax

```

1 void weird_string_syntax() {
2     std::string s1 = "line1"
3                     "line2"
4                     "line3";
5
6     std::cout << s1 << std::endl;
7
8     // output
9     // line1line2line3
10 }

```

13.8 Arithmetics with int and double

```

1 void arithmetics_with_int_and_double() {
2     // doing divisions only with integers will return an integer
3     std::cout << "7 / 2:  \t\t" << 7 / 2 << std::endl;
4     // as soon as a floating point is involved, the result is a floating point too
5     std::cout << "7 / 2.0: \t\t" << 7 / 2.0 << std::endl;
6     std::cout << "7.0 / 2:  \t\t" << 7.0 / 2 << std::endl;
7
8     // output
9     // 7 / 2:      3
10    // 7 / 2.0:    3.5
11    // 7.0 / 2:    3.5
12
13    // assigning the division of two integers to a double variable results in an int
14    double x = 7 / 2;
15    std::cout << "x: " << x << std::endl;
16    // output
17    // x: 3
18 }
19
20 void division_by_zero_demo() {
21     // division by zero is undefined behaviour. the following code compiles and does
22     // not cause an exception at runtime:
23     // std::cout << "1 / 0: \t\t" << 1 / 0 << std::endl;
24 }

```

13.9 Unspecified invocation order

```

1 int get_a() {
2     std::cout << "get_a()" << std::endl;
3     return 0;
4 }
5
6 int get_b() {
7     std::cout << "get_b()" << std::endl;
8     return 0;
9 }
10
11 void random_function_name(int a, int b) { }
12
13 void unspecified_invocation_order() {
14     // if we call functions to pass arguments to another function, we don't know in
15     // which order the functions are called.
16     random_function_name(get_a(), get_b());
17
18     // output could for example be
19     // get_b()
20     // get_a()
21 }

```

13.10 Factory function

```
1  #include <stdexcept>
2
3  namespace Galaxy {
4      struct Planet {
5          Planet() = default;
6
7          Planet(int distance) {
8              if (distance < 0) throw std::invalid_argument("");
9          }
10
11         int distance;
12     };
13
14     Planet make_planet(int distance)
15     try {
16         return Planet{distance};
17     } catch(std::invalid_argument e) {
18         return Planet{};
19     }
20 }
```

13.11 Args

```
1  #include <iostream>
2  #include <iterator>
3  #include <algorithm>
4  #include <vector>
5
6  int main(int argc, char* argv[]) {
7      std::vector<std::string> params{};
8      std::copy(argv + 1, argv + argc, std::back_inserter(params));
9
10     std::copy(params.begin(), params.end(),
11               std::ostream_iterator<std::string>(cout, "\n"));
12 }
```

14 Appendix

14.1 Random Code

14.1.1 PIMPL idiom

```
1  #ifndef PERSON_H_
2  #define PERSON_H_
3  #include <memory>
4  #include <string>
5  #include <vector>
6  #include <iosfwd>
7
8  class Person {
9      std::shared_ptr<class PersonImpl> person;
10
11      Person(std::shared_ptr<class PersonImpl> person) :
12          person { person } {}
13
14  public:
15      Person(std::string name);
16      Person(std::string name, Person father, Person mother);
17      ~Person();
18      void addChild(Person child);
19      std::string getName() const;
20      Person findChild(std::string name) const;
21      void killChild(Person child);
22      void killMe();
23      operator bool() const;
24      void print(std::ostream &out) const;
25  };
26
27 #endif /* PERSON_H_ */
```

```

1  #include "Person.h"
2  #include <iostream>
3  #include <algorithm>
4  #include <functional>
5
6  using PersonPtr=std::shared_ptr<class PersonImpl>;
7  using WeakPersonPtr=std::weak_ptr<class PersonImpl>;
8
9  class PersonImpl : public std::enable_shared_from_this<PersonImpl> {
10     std::string name;
11     WeakPersonPtr father; // don't lock parent objects
12     WeakPersonPtr mother;
13     std::vector<PersonPtr> children;
14
15     PersonPtr myLock() {
16         try {
17             auto me=shared_from_this(); // throws when called from dtor!
18             return me;
19         } catch(std::bad_weak_ptr const &ex){}
20         std::cout << "+++already dead? " << name<< '\n';
21         return PersonPtr{}; // already dead
22     }
23
24     public:
25     PersonImpl(std::string name,PersonPtr father,PersonPtr mother)
26     :name{name},father{father},mother{mother}{
27         // can not do shared_from_this here!
28         //no if(father) father->addChild(shared_from_this());
29     }
30
31     ~PersonImpl() {
32         std::cout << "killing me: " << name << '\n';
33         //killMe(); // can not call shared_from_this() in dtor!
34     }
35
36     void addChild(PersonPtr child){
37         children.push_back(child);
38     }
39
40     std::string getName() const {
41         return name;
42     }
43

```

```

44  PersonPtr findChild(std::string name) const {
45      using namespace std::placeholders;
46      auto finder=[name](PersonPtr const &person){
47          return person->getName() == name;
48      };
49      auto it=find_if(children.begin(),children.end(),finder);
50      if (it != children.end()) return *it;
51      return nullptr;
52  }
53
54  void killChild(PersonPtr child) {
55      if (child){
56          children.erase(find(children.begin(),children.end(),child));
57          //if (child->father == ) ?
58      }
59  }
60
61  void killMe() {
62      // here shared_from_this is possible
63      auto me=myLock();
64      if (!me) return; // already dead
65      auto realfather=father.lock();
66      if (realfather) realfather->killChild(me);
67      auto realmother=mother.lock();
68      if (realmother) realmother->killChild(me);
69      children.clear();
70  }
71
72  void print(std::ostream &out) const {
73      out << "Person: " << name ;
74      auto realfather=father.lock();
75      out << "    " << (realfather?realfather->getName():"orphan");
76      auto realmother=mother.lock();
77      out << "    " << (realmother?realmother->getName():"orphan");
78      out << "\n    ";
79      for(auto const &child:children){
80          out << child->name << ", ";
81      }
82      out << '\n';
83  }
84

```

```

85     static PersonPtr makePerson(std::string name,
86                               PersonPtr father={},
87                               PersonPtr mother={}) {
88         auto res = std::make_shared<PersonImpl>(name, father, mother);
89         if (father) father->addChild(res);
90         if (mother) mother->addChild(res);
91         return res;
92     }
93 };
94
95 Person::Person(std::string name) :
96     person { PersonImpl::makePerson(name) } {
97 }
98
99 Person::Person(std::string name, Person father, Person mother) :
100     person { PersonImpl::makePerson(name, father.person, mother.person) } {
101 }
102
103 Person::~Person() {}
104
105 void Person::addChild(Person child) { person->addChild(child.person); }
106 std::string Person::getName() const { return person->getName(); }
107 Person Person::findChild(std::string name) const {
108     auto result = person->findChild(name);
109     return Person { result };
110 }
111
112 void Person::killChild(Person child) {
113     person->killChild(child.person);
114 }
115 void Person::killMe() {
116     person->killMe();
117 }
118
119 void Person::print(std::ostream &out) const {
120     person->print(out);
121 }
122
123 Person::operator bool() const {
124     return person.get();
125 }

```

14.1.2 Word

```
1  #ifndef WORD_H_
2  #define WORD_H_
3
4  #include <iosfwd>
5  #include <string>
6
7  #include <boost/operators.hpp>
8
9  struct Word : boost::less_than_comparable<Word>, boost::equality_comparable<Word> {
10     Word():data{} {}
11     Word(std::string);
12
13     bool isValid() { return !data.empty(); }
14     void read(std::istream&);
15     void write(std::ostream& os) const { os << data; };
16
17     bool operator<(Word const& r) const { return toLower() < r.toLower(); }
18     bool operator==(Word const& r) const { return toLower() == r.toLower(); }
19     std::string operator+(std::string const& r) const { return data + r; }
20 private:
21     std::string data;
22     std::string toLower() const;
23 };
24
25 std::istream& operator>>(std::istream&, Word&);
26 std::ostream& operator<<(std::ostream&, Word const&);
27
28 #endif
```

```
1  #include "word.h"
2
3  #include <istream>
4  #include <ostream>
5  #include <string>
6  #include <algorithm>
7  #include <sstream>
8
9  Word::Word(std::string word):data{} {
10     std::istringstream in{word};
11     read(in);
12 }
13
14 void Word::read(std::istream& in) {
15     data.clear();
16
17     char c{};
18     while(in.get(c)) {
19         if(std::isspace(c) && !isValid()) continue;
20
21         if(std::isalpha(c)) {
22             data.push_back(c);
23         } else {
24             break;
25         }
26     }
27 }
28
29 std::string Word::toLower() const {
30     std::string lowered{data};
31     std::transform(lowered.begin(), lowered.end(), lowered.begin(), tolower);
32     return lowered;
33 }
34
35 std::istream& operator>>(std::istream& l, Word& r) {
36     r.read(l);
37     return l;
38 }
39
40 std::ostream& operator<<(std::ostream& l, Word const& r) {
41     r.write(l);
42     return l;
43 }
```

14.1.3 Ring

```

1  struct Ring5 {
2
3      explicit constexpr Ring5(unsigned x=0u) : val{ x % 5 } {}
4
5      constexpr unsigned value() const {
6          return val;
7      }
8
9      constexpr operator unsigned() const {
10         return val;
11     }
12
13     constexpr bool operator==(Ring5 const &r) const {
14         return val == r.val;
15     }
16
17     constexpr bool operator!=(Ring5 const &r) const {
18         return !(*this == r);
19     }
20
21     Ring5 operator+=(Ring5 const &r) {
22         val = (val + r.value()) % 5;
23         return *this;
24     }
25
26     Ring5 operator*=(Ring5 const&r) {
27         val = (val * r.value()) % 5;
28         return *this;
29     }
30
31     constexpr Ring5 operator+(Ring5 const &r) const {
32         return Ring5{val+r.val};
33     }
34
35     constexpr Ring5 operator*(Ring5 const &r) const {
36         return Ring5{val*r.val};
37     }
38
39 private:
40
41     unsigned val;
42
43 };

```

14.1.4 Sack

```

1  #ifndef SACK_H_
2  #define SACK_H_
3
4  #include<vector>
5  #include<map>
6  #include <iterator>
7
8  template <typename T, template<typename...> class C=std::vector>
9  class Sack
10 {
11     using SackType=C<T>;
12     using size_type=typename SackType::size_type;
13     SackType theSack{};
14
15 public:
16     Sack(std::initializer_list<T> const &items) :
17         theSack(items) {
18     }
19
20     bool empty() const {
21         return theSack.empty();
22     }
23
24     size_type size() const {
25         return theSack.size();
26     }
27
28     void putInto(T const &item) {
29         theSack.push_back(item);
30     }
31
32     T getOut() {
33         if (empty()) {
34             throw std::logic_error{"empty Sack"};
35         }
36
37         auto index = static_cast<size_type>(rand() % size());
38
39         T retval { theSack.at(index) };
40         theSack.erase( theSack.begin() + index);
41         return retval;
42     }
43 };
44 #endif /* SACK_H_ */

```

```
1  template <typename T>
2  class Sack<T, std::map> {
3      using SackType = std::map<T, unsigned>;
4      using size_type=typename SackType::size_type;
5      SackType theSack{};
6
7  public:
8      bool empty() {
9          return theSack.empty();
10     }
11
12     size_type size() {
13         size_type total = 0;
14         for (std::pair<T, unsigned> pair : theSack) {
15             total += pair.second;
16         }
17
18         return total;
19     }
20
21     void putInto(T const &item) {
22         theSack[item]++;
23     }
24
25     T getOut() {
26         if (empty()) {
27             throw std::logic_error{"empty Sack"};
28         }
29
30         auto index = static_cast<size_type>(rand() % size());
31
32         std::pair<T, unsigned> element = *std::next(theSack.begin(), index);
33
34         if (element.second == 1) {
35             theSack.erase(element.first);
36         } else {
37             theSack[element.first] = element.second - 1;
38         }
39         return element.first;
40     }
41 };
```

14.1.5 Highlander

```

1  #ifndef LIMITNUMBEROFINSTANCES_H_
2  #define LIMITNUMBEROFINSTANCES_H_
3
4  #include <stdexcept>
5
6  template <typename TOBELIMITED, unsigned int maxNumberOfInstances>
7  class LimitNofInstances {
8
9      static unsigned int counter;
10
11  protected:
12
13      void checkNofInstances() {
14          if(counter == maxNumberOfInstances) throw std::logic_error("too many instances");
15      }
16      LimitNofInstances() {
17          checkNofInstances();
18          ++counter;
19      }
20      ~LimitNofInstances() {
21          --counter;
22      }
23      LimitNofInstances(const LimitNofInstances &other){
24          checkNofInstances();
25          ++counter;
26      }
27  };
28
29  template <typename TOBELIMITED, unsigned int maxNumberOfInstances>
30  unsigned int
31  LimitNofInstances<TOBELIMITED,maxNumberOfInstances>::counter(0);
32
33  #endif /* LIMITNUMBEROFINSTANCES_H_ */
34
35  // using it:
36  class One : LimitNofInstances<One, 1>{ /*...*/};

```

14.1.6 dynArray

```

1  #ifndef DYNARRAY_H_
2  #define DYNARRAY_H_
3
4  #include <vector>
5
6  template<typename T>
7  struct dynArray {
8
9      using DynArrayType = std::vector<T>;
10     using size_type = typename DynArrayType::size_type;
11     using value_type = typename DynArrayType::value_type;
12     using iterator = typename DynArrayType::iterator;
13     using const_iterator = typename DynArrayType::const_iterator;
14     using allocator_type = typename DynArrayType::allocator_type;
15
16     // Constructors
17
18     explicit dynArray(const allocator_type& alloc = allocator_type()):container{alloc} {}
19     explicit dynArray(size_type n):container{n} {}
20     explicit dynArray(size_type n, value_type const& val, allocator_type const& alloc = allocator_type):container{n, val, alloc} {}
21
22     template<class InputIterator>
23     dynArray(InputIterator first, InputIterator last, allocator_type const& alloc = allocator_type()):container{first, last, alloc} {}
24
25     dynArray(dynArray const& x):container{x.container} {}
26     dynArray(dynArray const& x, allocator_type const& alloc):container{x.container, alloc} {}
27     dynArray(std::initializer_list<value_type> il, allocator_type const& alloc = allocator_type()):container{il, alloc} {}
28
29     // Iterators
30
31     iterator begin() {
32         return container.begin();
33     }
34
35     const_iterator begin() const {
36         return container.begin();
37     }
38
39     iterator end() {
40         return container.end();
41     }
42
43     const_iterator end() const {
44         return container.end();
45     }
46

```

```
47     iterator rbegin() {
48         return container.rbegin();
49     }
50
51     const_iterator rbegin() const {
52         return container.rbegin();
53     }
54
55     iterator rend() {
56         return container.rend();
57     }
58
59     const_iterator rend() const {
60         return container.rend();
61     }
62
63     const_iterator cbegin() const {
64         return container.cbegin();
65     }
66
67     const_iterator cend() const {
68         return container.cend();
69     }
70
71     const_iterator crbegin() const {
72         return container.crbegin();
73     }
74
75     const_iterator crend() const {
76         return container.crend();
77     }
78
79     // Capacity
80
81     size_type size() const {
82         return container.size();
83     }
84
```

```
85  void resize(size_type n) {
86      container.resize(n);
87  }
88
89  void resize(size_type n, const value_type& val) {
90      container.resize(n, val);
91  }
92
93  size_type capacity() const {
94      return container.capacity();
95  }
96
97  bool empty() const {
98      return container.empty();
99  }
100
101  // Element access
102
103  value_type& operator[](int n) {
104      return container[normalize_index(n)];
105  }
106
107  value_type const& operator[](int n) const {
108      return container[normalize_index(n)];
109  }
110
111  value_type& at(int n) {
112      return container.at(normalize_index(n));
113  }
114
115  value_type const& at(int n) const {
116      return container.at(normalize_index(n));
117  }
118
119  value_type& front() {
120      return container.front();
121  }
122
123  value_type const& front() const {
124      return container.front();
125  }
126
```

```
127     value_type& back() {
128         return container.back();
129     }
130
131     value_type const& back() const {
132         return container.back();
133     }
134
135     // Modifiers
136
137     void push_back(value_type const& val) {
138         container.push_back(val);
139     }
140
141     void push_back(value_type&& val) {
142         container.push_back(val);
143     }
144
145     void pop_back() {
146         container.pop_back();
147     }
148
149     iterator erase(iterator position) {
150         return container.erase(position);
151     }
152
153     iterator erase(iterator first, iterator last) {
154         return container.erase(first, last);
155     }
156
157     void clear() {
158         container.clear();
159     }
160
161     // Allocator
162
163     allocator_type get_allocator() const {
164         return container.get_allocator();
165     }
166
```

```
167 private:
168
169     DynArrayType container;
170
171     int normalize_index(int n) const {
172         return (n < 0) ? size()+n : n;
173     }
174
175 };
176
177 // Factories
178
179 template <typename T>
180 dynArray<T> makeDynArray(std::initializer_list<T> list) {
181     return dynArray<T>{list};
182 }
183
184 #endif /* DYNARRAY_H_ */
```
