C++ Code Snippets

Samuel Mueller

11. October 2014

Contents

1 Terminology

1.1 Expression

```
int foo{1}, bar{2};

int r = foo + bar;
// 'foo + bar' is an expression
// 'foo' itself is also an expression
```

1.2 Statement

```
void if_statement() {
  if (true) {} // valid statement
}

void adding_two_ints() {
  int three = 1 + 2; // '1 + 2' is a statement
}

void calling_a_function() {
  adding_two_ints(); // calling the function adding_two_ints() is a statement
}

void executable_part_of_a_function() {
  // the entire code between { } is a statement
}
```

1.3 Declaration

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

1.4 Definition

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

1.5 Parameter and Argument

```
// parameter
void call_me(int a) { } // 'a' is a parameter

// argument
void call_it() {
  call_me(1); // '1' is the argument for the call to call_me()
}
```

1.6 Predicate

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>

// a predicate is something that delivers a boolean value on a certain input.
// we differ between unary predicates, which take one argument, and binary
// predicates, which take two arguments.
// in C++, we can represent predicates with the following constructs:

// a struct overriding the call-operator
struct odd_s {
  bool operator() (int n) {
    return (n % 2);
  }
};

// a function
bool odd_f(int n) {
  return (n % 2);
}

// a lambda
auto odd_l = [](int n){ return (n % 2); };

void predicate() {
  using namespace std;

  vector<int> src{1, 2, 3, 4, 5, 6};

  copy_if(src.begin(), src.end(), ostream_iterator<int>{cout}, odd_f);
  copy_if(src.begin(), src.end(), ostream_iterator<int>{cout}, odd_l);
  copy_if(src.begin(), src.end(), ostream_iterator<int>{cout}, odd_s{});

  // output
  // 135135135
}
```

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

```cpp
#include <iostream>

int wild_function() {
  // the compiler automatically returns 1 if we dont define a return statement
}

void return_statement() {
  std::cout << "wild_function(): " << wild_function() << std::endl;

  // output
  // wild_function(): 1
}
```

2.1.3 function as argument

```cpp
double maybe(double d) {
}

// we can define functions as parameters with the following syntax:
void call_me(double f(double)) {
  f(1);
}

void function_as_argument() {
  call_me(maybe);

  // we can also pass in a lambda with the same signature:
  auto for_sure = [](double d) {return .0;};
  call_me(for_sure);
}
```

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

```
#include <string>
#include <sstream>

void throwing_anything() {
  try {
    throw 1;
  } catch (int i) {}

  try {
    throw "char";
  } catch (char const * i) {}


  try {
    throw std::string{"string"};
  } catch (std::string str) {}

  // we can not throw a stream because streams are not copyable
  // throw std::istringstream{"stream"};
  //                                ^  error (...)
}
```

2.2.2 Throwing Standard Exceptions

```cpp
#include <stdexcept>

void throwing_std_exceptions() {
  try {
    throw std::logic_error{"something weird happend"};
  } catch (std::logic_error e) {}
}

// other exceptions include
//
// std::logic_error
// std::domain_error
// std::invalid_argument
// std::length_error
// std::out_of_range
// std::runtime_error
// std::range_error
// std::overflow_error
// std::underflow_error
```

2.2.3 Function Followed By Try-Catch

```cpp
// a functions definition can directly be implemented with a try-catch block
void function_followed_by_try_catch()
try {
  // useful implementation that may cause exception goes here
} catch(int i) {

}
```

2.3 Operator Overloading

2.3.1 Basic Example

```cpp
#include <iostream>

struct Age {
  Age(int age) : data{age} {}

  bool operator<(Age const & comparable) const {
    //               ^---                ^--- not a requirement, but good practice
    //                                       for operators which dont change the state
    return data < comparable.data;
  }

  int data;
};

// an operator can also be defined outside of the types context.
// note that the member 'data' must be public in order to access it.
bool operator<=(Age& a, Age& b) {
  return a.data < b.data;
}

void basic_example() {
  Age a1{20};
  Age a2{30};

  std::cout << std::boolalpha << "20 < 30: " << (a1 < a2) << std::endl;
  std::cout << std::boolalpha << "20 <= 30: " << (a1 <= a2) << std::endl;
  // using the operator with a function call:
  std::cout << std::boolalpha << "20 < 30: " << (a1.operator<(a2)) << std::endl;
  // output
  // 20 < 30: true
  // 20 <= 30: true
  // 20 < 30: true

  // greater than is unimplemented, using it causes a compiler error:
  // std::cout << std::boolalpha << "20 > 30: " << (a1 > a2) << std::endl;
  //                                                ^  error: (...)
}
```

2.3.2 less than comparable

```cpp
#include <vector>
#include <string>
#include <iostream>

#include <boost/operators.hpp>

// we can inherit from boosts less_than_comparable and implement the less-than
// operator to have all other operators ready for our type:
struct Number : private boost::less_than_comparable<Number> {
  Number(int n) : data{n} {}

  bool operator<(Number const& comparable) const {
    data < comparable.data;
  }

  private: int data;
};

void boost_less_than_comparable() {
  Number n1{0};
  Number n2{1};

  // we can now call all the comparation operators:
  n1 < n2;
  n1 > n2;

  // heres an outline of how boost can simulate each operators behaviour with the
  // less-than operator:
  //
  // operator   simulation with <
  // -------------------------------
  // a > b       (b < a)
  // a >= b     !(a < b)
  // a <= b     !(b < a)
  // a == b     !(a < b) && !(b < a)
  // a != b     !(a == b)
}
```

2.3.3 Increment Postfix vs. Prefix

```cpp
#include <iostream>

struct Inc {
  void operator++() {
    std::cout << "this is the prefix increment" << std::endl;
  }

  // we must use a dummy parameter of type int so the compiler can differ between the
  // pre- and postfix increment operators. the dummy parameter doesnt have any
  // specific value:
  void operator++(int dummy) {
    std::cout << "this is the postfix increment" << std::endl;
  }
};

void increment_postfix_vs_prefix() {
  Inc i{};
  ++i;
  i++;

  // output
  // this is the prefix increment
  // this is the postfix increment
}
```

2.4 Enums

2.4.1 Enums Are Integers

```cpp
#include <iostream>

void enums_are_integers() {
  enum season { Spring, Summer, Autumn, Winter };
  //            0       1       2       3

  int summer = Summer;
  std::cout << "summer: " << summer << std::endl;
  std::cout << "Summer + Winter: " << Summer + Winter << std::endl;

  // output
  // summer: 1
  // Summer + Winter: 4

  // increment or decrement does not work however:
  //
  // Spring++;
  //        ^  error: no operator++(int) declared for postfix ++ [-fpermissive]
}
```

2.4.2 Indices Can Be Defined

```cpp
#include <iostream>

void indices_can_be_defined() {
  enum season { Spring = 10, Summer, Autumn, Winter };
  //            10           11      12      13
  std::cout << "Spring: " << Spring << std::endl;
  std::cout << "Winter: " << Winter << std::endl;
  // output
  // Spring: 10
  // Winter: 13

  enum brand { Feldschloesschen = 1, Calanda, Quoellfrisch,
  //           1                     2        3
               Budweiser = 10, Heineken, SanMiguel };
  //           10              11        12

  std::cout << "Feldschloesschen: " << Feldschloesschen << std::endl;
  std::cout << "Calanda: " << Calanda << std::endl;
  std::cout << "Quoellfrisch: " << Quoellfrisch << std::endl;
  std::cout << "Budweiser: " << Budweiser << std::endl;
  std::cout << "Heineken: " << Heineken << std::endl;
  std::cout << "SanMiguel: " << SanMiguel << std::endl;
  // output
  // Feldschloesschen: 1
  // Calanda:          2
  // Quoellfrisch:     3
  // Budweiser:       10
  // Heineken:        11
  // SanMiguel:       12
}

// enum class d_o_w
//   Was ist der Unterschied? Bezeichner sind nicht in einem bestimmten Scope drin
//   und global verwendbar.
```

2.4.3 Casting

```
void casting() {
  enum season { Spring, Summer, Autumn, Winter };
  //            0       1       2       3

  int summer = Summer;

  season favourite_season{Winter};
  std::cout << "favourite_season: " << favourite_season << std::endl;
  // output
  // favourite_season: 3

  // favourite_season = 1;
  //                    ^  error: invalid conversion from int to casting()::season

  favourite_season = static_cast<season>(1);
  std::cout << "favourite_season: " << favourite_season << std::endl;

  // output
  // favourite_season: 1
}
```

2.4.4 Operator Overloading

```cpp
#include <iostream>

enum Belt { Yellow, Orange, Green, Blue, Brown, Black };
//          0       1       2      3     4      5

Belt operator++(Belt &belt) {
  int tmp = (belt + 1) % (Black + 1);
  belt = static_cast<Belt>(tmp);
  return belt;
}

void operator_overloading() {
  Belt belt{Brown};

  std::cout << "belt: " << belt << std::endl;
  ++belt;
  std::cout << "belt: " << belt << std::endl;
  ++belt;
  std::cout << "belt: " << belt << std::endl;

  // output
  // belt: 4
  // belt: 5
  // belt: 0

  // calling this directly does not work
  // ++Green;
  // ^  error: no match for operator++ in ++(Belt)2u
}
```

2.4.5 Example: Toggle Button

```
#include <ostream>
#include <stdexcept>

struct ToggleButton {

  void hit_button() {
    state = static_cast<State>((state + 1) % 3);
  }

  void print(std::ostream &out) const {
    if (state == State::OFF) {
      out << "OFF";
    } else if (state == State::ON) {
      out << "ON";
    } else if (state == State::BLINK) {
      out << "Blink";
    } else {
      throw std::invalid_argument {"Unkown state"};
    }
  }

private:

  enum State : short {
    OFF, ON, BLINK
  };

  State state { State::OFF };

};

std::ostream& operator<<(std::ostream &out, ToggleButton const &button) {
  button.print(out);
  return out;
}
```

```cpp
#include "toggle_button.h"

void toggle_button_demo() {
  ToggleButton button{};

  std::cout << "button: " << button << std::endl;

  button.hit_button();

  std::cout << "button: " << button << std::endl;

  button.hit_button();

  std::cout << "button: " << button << std::endl;

  button.hit_button();

  std::cout << "button: " << button << std::endl;

  // output
  // button: OFF
  // button: ON
  // button: Blink
  // button: OFF
}
```

2.5 Lambdas

2.5.1 Basic Examples

```cpp
#include <iostream>
#include <string>

void nothing() {
  auto gapin_void = [ ]( ){ };
  //                  ^ ------------- captures
  //                    ^ ---------- parameters
  //                       ^ ------- implementation

  gapin_void(); // calling a lambda is syntactically similar to calling a function
}

void say_hi() {
  auto hello_thing = [](std::string thing) {
    std::cout << "hello " << thing << "!" << std::endl;
    return true;
  };

  bool result = hello_thing("world");

  std::cout << "result: " << result << std::endl;

  // output
  // hello world!
  // result: 1
}
```

2.5.2 Function Objects

```cpp
#include <iostream>
#include <functional>

void auto_vs_function() {
  // in the previous example we saw how the lambda was assigned to a variable of
  // type auto. we can also use std::function<signature>:
  std::function<void(double)> p1 = [](double a){  };
  std::function<double(double)> p2 = [](double a){ return .0; };

  // checking if something was assigned to a function instance:
  if (p1) {
    std::cout << "p1 does something" << std::endl;
  }

  // output
  // p1 does something
}
```

2.5.3 Capturing

```cpp
void capture_for_internal_use() {
  int magic_number = 8616;

  // if we want to use previously declared variables inside a lambda, we have to
  // capture them. otherwise, a compiler error will occur:
  auto capturer = [magic_number]() {
    //                          ^--- capturing magic_number

    int copy = magic_number;

    // trying to change magic_number causes a compiler error
    // magic_number = 9548;
    //                ^  error: assignment of read-only variable magic_number
  };
  capturer();
}
```

```cpp
#include <string>
#include <iostream>

void capture_with_write_access() {
  std::string name{"batman"};

  std::cout << name << std::endl;

  auto change = [&name]() {
    name = "spiderman";
  };

  change();

  std::cout << name << std::endl;

  // output
  // batman
  // spiderman
}
```

```cpp
#include <string>

std::string PIE{"omnom"};

void capture_everything_by_reference() {
  double PI = 3.141;

  auto wicked_lambda = [&]() {
    PIE = "mjam";
    PI = 3.142;
  };

  wicked_lambda();
}

void capture_everything_by_value() {
  double PI = 3.141;

  auto wicked_lambda = [=]() {
    PIE = "mjam";
    // PI = 3.142;
    //      ^  error: assignment of read-only variable PI
  };

  wicked_lambda();
}
```

```cpp
void mutable_demo() {
  int n = 1;

  // if we wanted to change a captured variable inside a lambda, we have to define
  // the lambda mutable. note that this is different to using the ampersand syntax,
  // since the changes wont be reflected to the outside.
  auto lambda = [n]() mutable {
    n = 2;

    // if we didnt specify mutable, the following compiler error would occur:
    //  ^  error: assignment of read-only variable n
  };

  // n has only been changed inside the lambda
  std::cout << "n: " << n << std::endl;

  // output
  // n: 1
}
```

3 Classes

3.1 Inheritance

3.1.1 constructor calls

```cpp
#include <iostream>

struct Furniture {
  Furniture() { std::cout << "furniture created" << std::endl; }
  ~Furniture() { std::cout << "furniture destroyed" << std::endl; }
};

struct Table : Furniture {
  Table() { std::cout << "table created" << std::endl; }
  ~Table() { std::cout << "table destroyed" << std::endl; }
};

struct WoodenTable : Table {
  WoodenTable() { std::cout << "wooden table created" << std::endl; }
  ~WoodenTable() { std::cout << "wooden table destroyed" << std::endl; }
};

void constructor_calls() {
  // the constructor of the base class is called first.
  // the constructor of the derived class is called last.
  // this order is reversed at the destruction of the object.
  WoodenTable wooden_table{};

  std::cout << "---" << std::endl;

  // output
  // furniture created
  // table created
  // wooden table created
  // ---
  // wooden table destroyed
  // table destroyed
  // furniture destroyed
}
```

3.1.2 calling super constructors

```
struct Furniture {
  Furniture() { std::cout << "furniture created" << std::endl; }
};

struct Table : Furniture {
  Table() { std::cout << "table created" << std::endl; }
};

struct WoodenTable : Table {

  // we can call a constructor of the direct base class
  WoodenTable() : Table{} {
    std::cout << "wooden table created" << std::endl;
  }

  // if we try to call a constructor from the base type Furniture, we get a
  // compiler error:
  // WoodenTable() : Furniture{} {
  //                 ^  error: type Furniture is not a direct base of WoodenTable
  // }

  // we can also delegate an other constructor of the same type. the delegation is
  // called first:
  WoodenTable(int i) : WoodenTable{(double) i} {
    std::cout << "wooden table created with " << i << std::endl;
  }

  WoodenTable(double i) {
    std::cout << "doing more " << i << std::endl;
  }

};

void calling_super_constructors() {
  // calling the constructor which calls the constructor of the direct base:
  WoodenTable table1{};
  // furniture created
  // table created
  // wooden table created

  // calling the constructor which delegates to a constructor in the same type.
  // note how the delegation is called first:
  WoodenTable table2{0};
  // furniture created
  // table created
  // doing more 0
  // wooden table created with 0
}
```

3.1.3 pure virtual function

```
struct Drink {
  // a pure virtual function can be compared to an abstract method in Java.
  // as soon as a class contains this kind of function, the type becomes abstract
  // and can not be instantiated.
  // we can declare pure virtual functions with the following syntax:
  virtual void prepare() = 0;
};

struct Beer : Drink {
  void prepare() {}
};

struct Rivella : Drink {
  // we dont have to implement prepare() here.
  // however, the type Rivella will remain abstract.
};

void pure_virtual_function() {
  // trying to instantiate an object of abstract type Drink causes an error:
  // Drink drink{};
  //             ^  error: cannot allocate an object of abstract type 'Drink'

  // we cant instantiate an object of type Rivella because it is abstract:
  // Rivella rivella{};
  //                 ^  error: cannot allocate an object of abstract type 'Rivella'

  Beer beer{};
}
```

3.1.4 object slicing

```
#include <iostream>

namespace example_1 {
  struct Creature {
    void walk() {}
  };

  struct Ork : Creature {
    void grunt() {}
  };

  void object_slicing() {
    // creatures can walk
    Creature creature{};
    creature.walk();

    // orks can walk and grunt
    Ork ork{};
    ork.walk();
    ork.grunt();

    // here we make a copy of the ork instance.
    // since we declare copy_of_ork to be of type Creature, only the part of that
    // type is copied. copy_of_ork therefore can not grunt. this is called object
    // slicing:
    Creature copy_of_ork = ork;
    copy_of_ork.walk();
    // copy_of_ork.grunt();
    //              ^  error: 'struct Creature' has no member named 'grunt'
  }
}
```

3.1.5 object slicing with reference and virtual

```
namespace example_2 {
  struct Creature {
    void walk() { std::cout << "creature is walking" << std::endl; }
    virtual void grunt() { std::cout << "creature is grunting" << std::endl;}
  };

  struct Ork : Creature {
    void walk() { std::cout << "ork is walking" << std::endl; }
    void grunt() { std::cout << "ork is grunting" << std::endl; }
  };

  void object_slicing() {
    Ork ork{};
    Creature  ork_c = ork;  // a copy of ork.
                            // contains the implementations of Creature only.
    Creature& ork_r = ork;  // a reference to an Ork object.
                            // delegates member calls to the implementations of
                            // Creature unless the member is declared virtual.
    Creature* ork_p = &ork; // a pointer to an Ork object. behaves the same way
                            // as the reference.

    // the following gets called on type Ork, as expected
    ork.walk();     // ork is walking
    ork.grunt();    // ork is grunting

    // the following gets called on type Creature due to object slicing
    ork_c.walk();   // creature is walking
    ork_c.grunt();  // creature is grunting

    // the following calls depend on whether the declarations are virtual or not.
    // if not declared virtual, the behaviour is the same as with object slicing.
    // if declared virtual, the calls are delegated to the original type Ork.
    ork_r.walk();   // creature is walking
    ork_r.grunt();  // ork is grunting

    ork_p->walk();   // creature is walking
    ork_p->grunt();  // ork is grunting
  }
}
```

3.1.6 member hiding problem

```cpp
#include <iostream>

struct Vehicle {
  void accelarate(int amount) {}
  void repair() {}
  void refuel() const { std::cout << "refueling vehicle" << std::endl; }
};

struct Toeff : Vehicle {
  // because we overload the accelarate function in a derived class, we have to
  // make the functions from the base class available with the following statement:
  using Vehicle::accelarate;

  void accelarate() {}
  void refuel() { std::cout << "refueling toeff" << std::endl; }
};

void member_hiding_problem() {
  Toeff t{};

  // calling a function from base type Vehicle
  t.repair();

  // calling accelarate defined in type Toeff
  t.accelarate();

  // calling accelarate defined in type Vehicle
  // this is only possible because using Vehicle::accelarate;
  t.accelarate(1);

  // if the using statement is not defined, the following compiler error is raised:
  // t.accelarate(1);
  //               ^  error: no matching function for call to Toeff::accelarate(int)

  // the const keyword wont hide overloaded functions. the function of the derived
  // class is called:
  t.refuel(); // output: refueling toeff

}
```

4 Immutability

4.1 Const Function Parameters

```
// The parameter a is passed to this function as a reference and can be modified
// inside it.
void i_can_change_you(int& a) {
  a++;
}

// This function declares its parameter as a reference and const. Const means that
// this parameter can not be modified inside the function.
// However, it is possible to pass in a non-const variable.
void i_cannot_change_you(const int& a) {
  // a++;
  // ^  error: increment of read-only reference 'a'
}

int test_i_cannot_change_you() {
  // passing a non-const object to a function that declares the parameter const is
  // possible:
  int a = 1;
  i_cannot_change_you(a);
}

// This function declares its parameter not as a reference and const. Const still
// means that the object can not be modified, however this makes less sense than in
// the previous example since this parameter is passed by value and therefore is a
// copy anyways.
void i_cannot_change_you_inside(const int a) {
  // a ++;
  // ^  error: increment of read-only parameter 'a'
}

// The previous function declaration is the same as the following, which would be
// ambiguous:
// void i_cannot_change_you_inside(int a) {
//      ^  error: redefinition of 'void i_cannot_change_you_inside(int)'
// }
```

4.2 Const Return Values

```
struct Dog {
  int age = 0;

  const int& getAge() {
    return age;
  }

  int& getMutableAge() {
    return age;
  }
};

void const_return_values() {
  Dog dog{};

  // variables declared const can not be modified
  const int& age = dog.getAge();
  // not possible:
  // age = 6;
  //       ^  error: assignment of read-only reference 'age'

  // not possible:
  // int& age = dog.getAge();
  //                        ^  error: invalid initialization of reference of type
  //                           'int&' from expression of type 'const int'

  int& muatable_age = dog.getMutableAge();

  // possible:
  muatable_age = 6;
}
```

4.3 Const Objects

```
void init_demo() {
  // if we define a variable const, it must be initialized:
  // const int number;
  //           ^  error: uninitialized const number [-fpermissive]

  const int number{0};
}

struct Cat {

  int age;

  void increaseAge() {
    age++;
  }

  // Because this function is declared const, it is not possible to modify members
  // of Cat.
  // Also, it is not possible to call member functions that are NOT declared const.
  void growUp() const {
    // age++;
    // ^  error: increment of member 'Cat::age' in read-only object

    // increaseAge();
    // ^  error: no matching function for call to 'Cat::increaseAge() const'
  }

  // Functions can be overloaded with the const keyword:
  // The non-const version will be called if the object itself is not const.
  // The const version will be called if the object itself is const.
  void notAmbiguous() {}

  void notAmbiguous() const {}

};

void const_functions() {
  Cat cat{};
  const Cat const_cat{};

  // calling a const function on a non const object is possible:
  cat.growUp();

  // calling a non-const function on a const object is not possible:
  // const_cat.increaseAge();
  //           ^  error: no matching function for call to Cat::increaseAge() const

  // calling functions overloaded with the const keyword depends on whether the
  // object is const itself:
  cat.notAmbiguous();
  const_cat.notAmbiguous();
}
```

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

```cpp
#include <iostream>
#include <sstream>

int get_age(std::istream& in) {
  int age{-1};

  while (!in.eof()) {
    if (in >> age) {
      return age;
    }

    std::cout << "in.good(): " << in.good() << std::endl; // in.good(): 0
    std::cout << "in.fail(): " << in.fail() << std::endl; // in.fail(): 1

    // at this point we have to read the remaining content of in so eof file is
    // reached and the while loop is escaped:
    in.clear();
    std::string line{};
    std::getline(in, line);
  }

  return -1;
}

void reading_integers() {
  std::istringstream in{"24 a 25"};

  int age;

  age = get_age(in);
  std::cout << "age: " << age << std::endl; // age: 24

  age = get_age(in);
  std::cout << "age: " << age << std::endl; // age: -1

  std::cout << "in.eof(): " << in.eof() << std::endl; // in.eof(): 1
}
```

5.1.2 Continue After Fail

```cpp
#include <iostream>
#include <sstream>

int get_age_2(std::istream& in) {
  int age{-1};

  while (!in.eof()) {
    if (in >> age) {
      return age;
    }

    // read the invalid sequence then continue
    in.clear();
    std::string invalid_sequence{};
    in >> invalid_sequence;
  }

  return -1;
}

void reading_integers_2() {
  std::istringstream in{"24 a 25    "};

  int age;

  age = get_age_2(in);
  std::cout << "age: " << age << std::endl; // age: 24

  age = get_age_2(in);
  std::cout << "age: " << age << std::endl; // age: 25

  age = get_age_2(in);
  std::cout << "age: " << age << std::endl; // age: -1

  std::cout << "in.eof(): " << in.eof() << std::endl; // in.eof(): 1
}
```

5.2 Manipulators

5.2.1 boolalpha

```cpp
#include <iostream>

void boolalpha_demo() {
  std::cout << true << std::endl;
  std::cout << std::boolalpha << true << std::endl;
  std::cout << true << std::endl;
  std::cout << 1 << std::endl;
  std::cout << 0 << std::endl;
  std::cout << std::noboolalpha << true << std::endl;

  // Output:
  // 1
  // true
  // true
  // 1
  // 0
  // 1
}
```

5.2.2 skipws

```cpp
#include <iostream>
#include <sstream>

void skipws_demo() {
  // Only has effect on istream
  char a, b, c;

  std::istringstream in{"  123"};
  in >> std::skipws >> a >> b >> c;
  std::cout << a << b << c << std::endl;
  // output: 123

  in.seekg(0); // reset stream to read from beginning
  in >> std::noskipws >> a >> b >> c;
  std::cout << a << b << c << std::endl;
  // output:   1

  std::cout << std::skipws << "   abc   def" << std::endl; // no effect
}
```

5.2.3 uppercase

```cpp
#include <iostream>

void uppercase_demo() {
  // makes hexadecimal representations uppercase.
  // attention: non numeric types will NOT be uppercase!
  // the following example shows no effect:
  std::cout << std::uppercase << "abc" << std::endl << std::nouppercase;
  // Output:
  // abc

  // Output integers as hex values:
  std::cout << std::showbase << std::hex;
  std::cout << std::uppercase << 77 << std::endl;
  std::cout << std::nouppercase << 77 << std::endl;

  // Output:
  // 0X4D
  // 0x4d
}
```

5.2.4 oct, hex, dec

```cpp
#include <iostream> // std::cout, std::dec, std::hex, std::oct

void oct_hex_dec() {
  int n = 29;
  std::cout << std::dec << n << std::endl;
  // the setting persists for subsequent calls
  std::cout << std::hex << n << std::endl;
  std::cout << n << std::endl;
  std::cout << std::oct << n << std::endl;
  std::cout << n << std::endl;

  // output
  // 29
  // 1d
  // 1d
  // 35
  // 35
}
```

5.2.5 setw

```cpp
#include <iostream>
#include <iomanip> // std::setw

void std_cout_width() {
  // setting the width on an output stream determines the minimum count of characters
  // the output shall have:
  std::cout.width(4);
  std::cout << "ab" << std::endl;

  // this setting will be consumed by the first output to the stream. the next
  // output will not have the width set anymore:
  std::cout << "ab" << std::endl;

  //output
  //   ab
  // ab
}

void setw() {
  // we can use setw(n) as a shortcut to width(n):
  std::cout << std::setw(4) << "ab" << std::endl;
  std::cout << "ab" << std::endl;

  //output
  //   ab
  // ab
}
```

5.2.6 left, right, internal

```cpp
#include <iostream>

void left_right_internal() {
  int n = -1;

  // using std::left or std::right will align the output to the left or right if a
  // width has been specified:
  std::cout.width(6);
  std::cout << std::left << n << std::endl;

  std::cout.width(6);
  std::cout << std::right << n << std::endl;

  // std::internal can be used for negative numbers.
  // for non-numerical values it is equivalent to right:
  std::cout.width(6);
  std::cout << std::internal << n << std::endl;

  // output
  // -1
  //     -1
  // -    1
}
```

5.2.7 setprecision, scientific, fixed

```cpp
#include <iostream> // setprecision, scientific, fixed

double PI = 3.14159;

void setprecision() {

  // setprecision sets the maximum digits to display.
  // the number is automatically rounded.
  std::cout << std::setprecision(5) << PI << std::endl;
  // setprecision will persist for subsequent outputs to the stream
  std::cout << PI << std::endl;

  // output
  // 3.1416
  // 3.1416
}

void scientific() {
  std::cout << std::scientific << 10000000.0 << std::endl;

  std::cout << std::setprecision(0);

  std::cout << std::scientific << 10000000.0 << std::endl;

  // output
  // 1.000000e+07
  // 1e+07
}

void fixed() {
  // expands the output to a minimum of digits.
  std::cout << std::fixed << 1.1 << std::endl;
  // 1.100000

  std::cout << std::fixed << std::setprecision(3) << 1.0 << std::endl;
  std::cout << std::fixed << PI << std::endl;
  // 1.000
  // 3.142
}
```

6 Iterators

6.1 Insert Iterators

6.1.1 Insert iterator

```cpp
#include <iterator>
#include <vector>
#include <algorithm>
#include "util.h"

void insert_iterator() {
  std::vector<int> incomplete{1, 4, 5};
  print("incomplete", incomplete);

  std::vector<int>::iterator it = incomplete.begin();
  it++; // we have to insert after the first entry, so lets increment by one
  std::insert_iterator<std::vector<int>> insert_iterator{incomplete, it};

  std::vector<int> addition{2, 3}; // this is the content we want to insert
  std::copy(addition.begin(), addition.end(), insert_iterator);

  print("complete", incomplete);

  // output
  // incomplete:   {1, 4, 5}
  // complete:     {1, 2, 3, 4, 5}
}
```

6.1.2 Back insert iterator

```cpp
#include <iterator>
#include <vector>
#include <algorithm>
#include "util.h"

void back_insert_iterator() {
  std::vector<int> incomplete{5, 4};
  print("incomplete", incomplete);

  std::vector<int> addition{3, 2, 1};
  std::copy(addition.begin(), addition.end(), std::back_inserter(incomplete));

  print("complete", incomplete);

  // output
  // incomplete:   {5, 4}
  // complete:     {5, 4, 3, 2, 1}
}
```

6.1.3 Front insert iterator

```cpp
#include <iterator>
#include <list>
#include <algorithm>
#include "util.h"

void front_insert_iterator() {
  std::list<int> incomplete{4, 5};
  print("incomplete", incomplete);

  std::vector<int> addition{3, 2, 1};
  std::copy(addition.begin(), addition.end(), std::front_inserter(incomplete));

  print("complete", incomplete);

  // output
  // incomplete:   {4, 5}
  // complete:     {1, 2, 3, 4, 5}
}
```

6.2 Stream Iterators

6.2.1 Iterate over strings

```cpp
#include <sstream>
#include <iterator>
#include <iostream>

void iterate_over_strings() {
  using namespace std;

  istringstream in{"ill iterate until youre dizzy"};

  copy(istream_iterator<string>{in}, istream_iterator<string>{},
       ostream_iterator<string>{cout, "-"});

  // output
  // ill-iterate-until-youre-dizzy
}
```

6.2.2 Iterate over ints

```cpp
#include <sstream>
#include <iterator>
#include <iostream>

void iterate_over_ints() {
  using namespace std;

  istringstream in{"1 2 3"};

  copy(istream_iterator<int>{in}, istream_iterator<int>{},
       ostream_iterator<int>{cout, ""});

  // output
  // 123
}
```

6.2.3 Iterate over your own types

```
#include <string>
#include <algorithm>

struct Word {

  void read(std::istream& in) {
    in >> data;
    std::transform(data.begin(), data.end(), data.begin(), toupper);
  }

  void write(std::ostream& os) const {
    os << data;
  };

private:

  std::string data;

};

std::istream& operator>>(std::istream& in, Word& r) {
  r.read(in);
  return in;
}

std::ostream& operator<<(std::ostream& out, Word const& r) {
  r.write(out);
  return out;
}


void iterate_over_your_own_types() {
  using namespace std;

  istringstream in{"ill iterate until youre dizzy"};

  copy(istream_iterator<Word>{in}, istream_iterator<Word>{},
       ostream_iterator<Word>{cout, " "});

  // output
  // ILL ITERATE UNTIL YOURE DIZZY
}
```

6.2.4 Count Chars

```cpp
#include <iostream>
#include <sstream> // istringstream
#include <iterator>
#include <algorithm>

void count_all_chars_skip_whitespaces() {
  std::istringstream in{"ab c d !?  &*"};

  // istream_iterator<char> will iterator through every char in the stream, except
  // white spaces.
  using Iterator = std::istream_iterator<char>;
  Iterator begin{in};
  Iterator end{};

  // we could also use the count_if algorithm and use a lamda that always returns
  // true. distance is more elegant, however a bit abstract:
  int count = std::distance(begin, end);

  std::cout << "count: " << count << std::endl; // count: 8
}

void count_all_chars() {
  std::istringstream in{"ab c d !?  &*"};

  // using a istreambuf_iterator avoids skipping white spaces.
  using Iterator = std::istreambuf_iterator<char>;
  Iterator begin{in};
  Iterator end{};

  int count = std::distance(begin, end);

  std::cout << "count: " << count << std::endl; // count: 13
}
```

6.2.5 Count Specific Chars

```cpp
void count_specific_chars() {
  std::istringstream in{"oppa gangnam style"};

  using Iterator = std::istream_iterator<char>;
  Iterator begin{in};
  Iterator end{};

  int count = std::count(begin, end, 'a');

  std::cout << "count: " << count << std::endl; // count: 3
}
```

6.2.6 Count Words

```
void count_words() {
  std::istringstream in{"oppa gangnam style :D"};

  // this iterator will read char sequences that are separated by white spaces:
  using Iterator = std::istream_iterator<std::string>;
  Iterator begin{in};
  Iterator end{};

  int count = std::distance(begin, end);

  std::cout << "count: " << count << std::endl; // count: 4
}
```

6.3 Custom Iterators

6.3.1 Line Iterator

```cpp
#include <iostream>
#include <iterator>
#include <algorithm>
#include <sstream>
#include <boost/operators.hpp>

namespace {
  std::istringstream empty{};
}

struct LineIterator : boost::input_iterator_helper<LineIterator, std::string> {

  LineIterator() : in(empty) {
    in.clear(std::ios_base::eofbit);
  }

  explicit LineIterator(std::istream &in) : in(in) {
    read_next_line();
  }

  bool operator==(LineIterator const &r) const {
    return !in.good() && !r.in.good();
  }

  value_type operator*() const {
    return current_line;
  }

  LineIterator &operator ++() {
    read_next_line();
    return *this;
  }

private:

  void read_next_line() {
    getline(in, current_line);
  }
  std::istream &in;
  std::string current_line;
};
```

```cpp
#include "line_iterator.h"

void line_iterator_demo() {
  std::istringstream in{"first line\nsecond line"};
  LineIterator li{in};

  std::cout << "*li: " << *li << std::endl;
  li++;
  std::cout << "*li: " << *li << std::endl;

  // output
  // *li: first line
  // *li: second line
}

int main() {
  line_iterator_demo();
}
```

6.3.2 My Ostream Iterator

```cpp
#include <iostream>
#include <iterator>
#include <algorithm>

struct MyOstreamIterator
: std::iterator<std::output_iterator_tag, int>  {

  explicit MyOstreamIterator(std::ostream& out) : out{&out} {}

  MyOstreamIterator& operator =(const int value) {
    *out << value;
    return *this;
  }

  MyOstreamIterator& operator *() { return *this; }
  MyOstreamIterator& operator ++() { return *this; }
  MyOstreamIterator& operator ++(int) { return *this; }

private:

  std::ostream* out;

};

void my_ostream_iterator_demo() {
  MyOstreamIterator moi{std::cout};

  moi = 1;
  moi = 20;
  moi = 300;

  // output
  // 120300
}
```

6.3.3 Square Iterator

```
#include <iterator>

struct SquareIterator : std::iterator<std::input_iterator_tag, int>
{
  explicit SquareIterator(int start=0) : value{start} {}

  bool operator ==(SquareIterator const &r) const {
    return value == r.value;
  }

  bool operator !=(SquareIterator const &r) const {
    return !(*this == r);
  }

  value_type operator *() const {
    return value * value;
  }

  SquareIterator& operator ++() {
    ++value;
    return *this;
  }

  SquareIterator operator ++(int) {
    auto old = *this;
    ++(*this);
    return old;
  }

private:

  int value;

};
```

```cpp
#include "square_iterator.h"

#include <iostream>
#include <algorithm>

void square_iterator_demo() {

  SquareIterator begin{2};
  SquareIterator end{6};

  std::for_each(begin, end, [](int sqr) {
    std::cout << "sqr: " << sqr << std::endl;
  });

  // output
  // sqr: 4
  // sqr: 9
  // sqr: 16
  // sqr: 25
}

int main() {
  square_iterator_demo();
}
```

6.3.4 Prime Iterator

```cpp
#include <iostream>
#include <vector>
#include <array>
#include <algorithm>
#include <boost/iterator/counting_iterator.hpp>
#include <boost/iterator/filter_iterator.hpp>

bool is_divisable(int x, int divisor) {
  return !(x % divisor);
}

bool is_prime(unsigned x) {
  if (x % 2 == 0 || x < 3) return false;
  return std::none_of(boost::make_counting_iterator(2u),
                      boost::make_counting_iterator(x),
                      [x](unsigned divisor) { return is_divisable(x, divisor); });
}

void prime_iterator() {
  auto counting = boost::make_counting_iterator(1);
  auto countingEnd = boost::make_counting_iterator(40);

  auto prime_iterator = boost::make_filter_iterator(is_prime, counting);
  auto prime_iterator_end = boost::make_filter_iterator(is_prime, countingEnd);

  std::copy(prime_iterator, prime_iterator_end,
            std::ostream_iterator<int>{std::cout, " "});

  // output
  // 3 5 7 11 13 17 19 23 29 31 37
}
```

6.4 Iterator Semantics

6.4.1 Iterator

```cpp
#include <iostream>
#include <iterator>
#include <vector>

void iterator() {
  std::vector<int> v{0, 1, 2, 3, 4};

  // 0 1 2 3 4
  // ^ v.begin()

  // 0 1 2 3 4
  //           ^ v.end()

  // 0 1 2 3 4
  //         ^ --v.end()

  std::cout << "*v.begin():   " << *v.begin()   << std::endl;
  std::cout << "*v.end():     " << *v.end()     << std::endl;
  std::cout << "*(--v.end()): " << *(--v.end()) << std::endl;

  // output
  // *v.begin():   0
  // *v.end():     (undefined behaviour)
  // *(--v.end()): 4
}
```

6.4.2 Reverse Iterator

```cpp
void reverse_iterator() {
  std::vector<int> v{10, 11, 12, 13, 14};

  // 10 11 12 13 14
  //             ^ v.rbegin()

  // 10 11 12 13 14
  //          ^ v.rbegin() + 1

  //   10 11 12 13 14
  // ^ v.rend()

  //     10 11 12 13 14
  // ^ v.rend() + 1

  // 10 11 12 13 14
  // ^ v.rend() - 1

  std::vector<int>::reverse_iterator rbegin = v.rbegin();
  std::vector<int>::reverse_iterator rend = v.rend();

  std::cout << "*rbegin:       " << *rbegin       << std::endl;
  std::cout << "*(rbegin + 1): " << *(rbegin + 1) << std::endl;
  std::cout << "*rend:         " << *rend         << std::endl;
  std::cout << "*(rend + 1):   " << *(rend + 1)   << std::endl;
  std::cout << "*(rend - 1):   " << *(rend - 1)   << std::endl;

  // output
  // *rbegin:       14
  // *(rbegin + 1): 13
  // *rend:         (undefined behaviour)
  // *(rend + 1):   (undefined behaviour)
  // *(rend - 1):   10
}
```

7 Containers

7.1 Vector

7.1.1 Initialization

```cpp
#include <vector>

void initialization() {
  // with initializer list
  std::vector<int> ve1{1, 2, 3};

  // with capacity
  // elements are zero-initialized
  std::vector<int> ve2(3);
}
```

7.1.2 Accessing Elements

```cpp
#include <vector>

void accessing_elements() {
  std::vector<int> ve1(3);

  // we can use at() or the []-operator to access elements in a vector
  ve1.at(1);
  ve1[1];

  // the difference lies in the behaviour of when trying to access an element with
  // an index that is out of range:

  // the following code would throw 'std::out_of_range'
  // ve1.at(3);

  // the following code is undefined behaviour and does not throw an exception
  // ve1[3];
}
```

7.1.3 Assigning Values

```cpp
#include <iostream>
#include <vector>

void assigning_values() {
  std::vector<int> ve1{3, 2, 1};

  ve1[1] = 10; // OK

  // the following line tries to access an invalid index and is undefined behaviour
  // ve1[3] = 30;
}

void using_front() {
  std::vector<int> ve1{3, 2, 1};

  // because front() returns a reference, we can also assign values to it
  std::cout << ve1.front() << std::endl;
  ve1.front() = 4;
  std::cout << ve1.front() << std::endl;
  // Output:
  // 3
  // 4

  // there is also a const version of front();
  const int f = ve1.front();

  // the opposite to front() is back()
}
```

7.1.4 Appending Values

```cpp
#include <vector>

void appending_values() {
  std::vector<int> ve1(3);
  ve1.push_back(0);
  ve1.push_back(1);
  ve1.push_back(2);

  // At this point, the vector will automatically resize.
  ve1.push_back(3);
}
```

7.1.5 Iterating Elements

```cpp
#include <vector>

void iteration_with_read_only_access() {
  std::vector<int> v{1, 2, 3};
  for (auto const i : v) {
    std::cout << i << " ";

    // i++;
    // ^  error: increment of read-only variable i
  }
  std::cout << '\n';

  // Output:
  // 1 2 3
}

void iteration_with_reference() {
  std::vector<int> v{1, 2, 3};
  for (auto &i : v) {
    i++;
    std::cout << i << " ";
  }
  std::cout << '\n';

  // Output
  // 2 3 4
}
```

8 Algorithms

8.1 Modifying

8.1.1 Copy

```cpp
#include "util.h"

#include <algorithm>
#include <vector>

void copy_directly() {
  std::vector<int> src{1, 2, 3, 4};

  // note that the size of dest must be at least the size of src to avoid
  // undefined behaviour:
  std::vector<int> dest(4);

  std::copy(src.begin(), src.end(), dest.begin());

  print("src", src);
  print("dest", dest);

  // output
  // src:  1 2 3 4
  // dest: 1 2 3 4
}

void copy_with_back_inserter() {
  std::vector<int> src{5, 6, 7, 8};
  std::vector<int> dest{};

  // using a back_inserter here makes sure that dest is being resized if required:
  std::copy(src.begin(), src.end(), std::back_inserter(dest));

  print("src", src);
  print("dest", dest);

  // output
  // src:  5 6 7 8
  // dest: 5 6 7 8
}
```

8.1.2 Move

```
#include "util.h"

#include <algorithm>
#include <vector>
#include <string>

void move_integers() {
  std::vector<int> numbers{1, 2, 3, 4};
  std::vector<int> copied_numbers;

  std::move(numbers.begin(), numbers.end(), std::back_inserter(copied_numbers));

  print("numbers", numbers);
  print("copied_numbers", copied_numbers);

  // moving the elements depends on the move semantics of the given type.
  // in this case, we are moving ints. the move semantics of ints keeps the original
  // values in the vector:
  //
  // numbers:          1 2 3 4
  // copied_numbers:   1 2 3 4
}

void move_strings() {
  std::vector<std::string> strings{"a", "b", "c"};
  std::vector<std::string> copied_strings;

  std::move(strings.begin(), strings.end(), std::back_inserter(copied_strings));

  print("strings", strings);
  print("copied_strings", copied_strings);

  // move semantics of string actually moves the elements from one vector to the
  // other:
  //
  // strings:          "" "" ""
  // copied_strings:   "a" "b" "c"
}
```

8.1.3 Transform

```cpp
#include "util.h"

#include <algorithm>
#include <vector>

void unary_transform() {
  std::vector<int> numbers{1, 2, 3, 4};

  std::transform(numbers.begin(), numbers.end(),
                 numbers.begin(),
                 [](int n) { return n + 10; });

  print("numbers", numbers);

  // output
  // 11 12 13 14
}

void binary_transform() {
  std::vector<int> numbers1{1, 2, 3, 4};
  std::vector<int> numbers2{4, 3, 2, 1};

  std::transform(numbers1.begin(), numbers1.end(),
                 numbers2.begin(), numbers2.begin(),
                 [](int n1, int n2) { return n1 + n2; });

  print("numbers2", numbers2);

  // output
  // 5 5 5 5
}
```

8.2 Partitioning

8.2.1 Partitioning

```cpp
#include "util.h"

#include <algorithm>
#include <vector>

void partition() {
  std::vector<int> v{1, 2, 3, 4, 5, 6, 7, 8};

  print("before", v);

  // puts all the even numbers into the first half of the vector and all the odd
  // numbers into the second half
  std::partition(v.begin(), v.end(), [](int n) { return !(n % 2); });

  print("after", v);

  // output
  // before: 1 2 3 4 5 6 7 8
  // after:  8 2 6 4 5 3 7 1
}

// makes sure the relative order of the elements is preserved
void stable_partition() {
  std::vector<int> v{1, 2, 3, 4, 5, 6, 7, 8};

  print("before", v);

  std::stable_partition(v.begin(), v.end(), [](int n) { return !(n % 2); });

  print("after", v);

  // output
  // before: 1 2 3 4 5 6 7 8
  // after:  2 4 6 8 1 3 5 7
}

// besides these there are the following algorithms handling partitioning:
//
// is_partitioned(v.begin(), v.end(), unary_predicate)
//
// partition_copy(v.begin(), v.end(), dest1.begin(), dest2.begin(), unary_predicate)
//
// forward_iterator = partition_point(v.begin(), v.end(), unary_predicate)
```

8.3 Numeric

8.3.1 Accumulate

```cpp
void accumulate_demo() {
  std::vector<int> v{1, 2, 3, 4, 5};

  int sum = std::accumulate(v.begin(), v.end(), 10);
  std::cout << "sum: " << sum << std::endl;

  int fac = std::accumulate(v.begin(), v.end(), 1, std::multiplies<int>());
  //                                               ^ multiplication with 0 = 0!
  std::cout << "fac: " << fac << std::endl;

  // output
  // sum: 25
  // fac: 120
}
```

8.3.2 Adjacent Difference

```cpp
void adjacent_difference_demo() {
  std::vector<int> v{1, 2, 3, 4, 5, 6, 7, 8, 9};

  std::vector<int> d1{};
  std::vector<int> d2{};

  std::adjacent_difference(v.begin(), v.end(), std::back_inserter(d1));
  std::adjacent_difference(v.begin(), v.end(), std::back_inserter(d2),
                           std::plus<int>());

  std::copy(d1.begin(), d1.end(), std::ostream_iterator<int>{std::cout, " "});
  std::cout << std::endl;
  std::copy(d2.begin(), d2.end(), std::ostream_iterator<int>{std::cout, " "});

  // output
  // 1 1 1 1 1 1 1 1 1
  // 1 3 5 7 9 11 13 15 17
}
```

8.3.3 Inner Product

```cpp
void inner_product_demo() {
  std::vector<int> v1{1,   2,   3};
  //                  *    *    *
  std::vector<int> v2{4,   5,   6};
  //                  4 + 10 + 18   = 32

  int res = std::inner_product(v1.begin(), v1.end(), v2.begin(), 0);

  std::cout << "res: " << res << std::endl;

  // output
  // res: 32
}
```

8.3.4 Partial Sum

```cpp
void partial_sum_demo() {
  // 1 =                  1
  // 1 + 2 =              3
  // 1 + 2 + 3 =          6
  // 1 + 2 + 3 + 4 =     10
  // 1 + 2 + 3 + 4 + 5 = 15

  std::vector<int> v{1, 2, 3, 4, 5};
  std::vector<int> r(5);

  std::partial_sum(v.begin(), v.end(), r.begin());

  std::copy(r.begin(), r.end(), std::ostream_iterator<int>(std::cout, " "));

  // output
  // 1 3 6 10 15
}
```

8.4 Misc

8.4.1 Fill Vector With Squares

```cpp
#include <iostream>
#include <iterator>
#include <vector>
#include <algorithm>
#include <numeric>

void fill_vector_with_squares() {
  // create vector containing only ones
  std::vector<int> v(10, 1);

  // this will make the vector contain the values from 1 to 10
  std::partial_sum(v.begin(), v.end(), v.begin());

  // square every item in v
  std::transform(v.begin(), v.end(), v.begin(), [](int i) { return i * i; });

  // print it
  std::copy(v.begin(), v.end(), std::ostream_iterator<int>{std::cout, " "});

  // output
  // 1 4 9 16 25 36 49 64 81 100
}
```

8.4.2 Is Palindrome

```cpp
#include <iostream>
#include <cctype>
#include <iterator>
#include <string>
#include <algorithm>

bool are_equal(char a, char b) {
  return tolower(a) == tolower(b);
}

bool is_palindrome(std::string word) {
  return std::equal(word.cbegin(),
                    word.cbegin() + word.length() / 2,
                    word.crbegin(),
                    are_equal);
}

void is_palindrome_demo() {
  std::cout << std::boolalpha;
  std::cout << is_palindrome("Rihanna") << std::endl;
  std::cout << is_palindrome("Anna") << std::endl;
  std::cout << is_palindrome("abcba") << std::endl;
  std::cout << is_palindrome("trugtimeinesohellehoseniemitgurt") << std::endl;

  // output
  // false
  // true
  // true
  // true
}
```

8.4.3 Multiplication Table

```cpp
#include <iomanip> // std::setw

void multiplication_table() {
  std::vector<int> numbers(20, 1);

  // generate a list containing the values from 1 to 20.
  // could also be done with iota which would be much easier lol.
  std::transform(numbers.begin(), numbers.end()-1, numbers.begin()+1, numbers.begin()+1,
                 [](int a, int b) { return a + b; });

  std::for_each(numbers.begin(), numbers.end(), [&numbers](int i) {
    std::for_each(numbers.begin(), numbers.end(), [&i](int j) {
      std::cout << std::setw(4) << i * j;
    });
    std::cout << '\n';
  });

  // output
  //
  //   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19  20
  //   2   4   6   8  10  12  14  16  18  20  22  24  26  28  30  32  34  36  38  40
  //   3   6   9  12  15  18  21  24  27  30  33  36  39  42  45  48  51  54  57  60
  //   4   8  12  16  20  24  28  32  36  40  44  48  52  56  60  64  68  72  76  80
  //   5  10  15  20  25  30  35  40  45  50  55  60  65  70  75  80  85  90  95 100
  //   6  12  18  24  30  36  42  48  54  60  66  72  78  84  90  96 102 108 114 120
  //   7  14  21  28  35  42  49  56  63  70  77  84  91  98 105 112 119 126 133 140
  //   8  16  24  32  40  48  56  64  72  80  88  96 104 112 120 128 136 144 152 160
  //   9  18  27  36  45  54  63  72  81  90  99 108 117 126 135 144 153 162 171 180
  //  10  20  30  40  50  60  70  80  90 100 110 120 130 140 150 160 170 180 190 200
  //  11  22  33  44  55  66  77  88  99 110 121 132 143 154 165 176 187 198 209 220
  //  12  24  36  48  60  72  84  96 108 120 132 144 156 168 180 192 204 216 228 240
  //  13  26  39  52  65  78  91 104 117 130 143 156 169 182 195 208 221 234 247 260
  //  14  28  42  56  70  84  98 112 126 140 154 168 182 196 210 224 238 252 266 280
  //  15  30  45  60  75  90 105 120 135 150 165 180 195 210 225 240 255 270 285 300
  //  16  32  48  64  80  96 112 128 144 160 176 192 208 224 240 256 272 288 304 320
  //  17  34  51  68  85 102 119 136 153 170 187 204 221 238 255 272 289 306 323 340
  //  18  36  54  72  90 108 126 144 162 180 198 216 234 252 270 288 306 324 342 360
  //  19  38  57  76  95 114 133 152 171 190 209 228 247 266 285 304 323 342 361 380
  //  20  40  60  80 100 120 140 160 180 200 220 240 260 280 300 320 340 360 380 400
}
```

8.4.4 Sum Numbers

```cpp
#include <numeric>   // accumulate
#include <sstream>

void sum_integers() {
  std::istringstream in{"1 2 3"};

  using Iterator = std::istream_iterator<int>;
  Iterator begin{in};
  Iterator end{};

  int sum = 0;

  sum = std::accumulate(begin, end, 0);
  std::cout << "sum: " << sum << std::endl; // sum: 6
}

void sum_floats() {
  std::istringstream in{"1.1 2.2 3.3"};

  using Iterator = std::istream_iterator<double>;
  Iterator begin{in};
  Iterator end{};

  double sum = 0;

  sum = std::accumulate(begin, end, .0);
  //                                ^--- we have to pass in a double, otherwise result
  //                                     will be integer!
  std::cout << "sum: " << sum << std::endl; // sum: 6.6
}
```

8.4.5 Word List

```cpp
#include <algorithm>
#include <iostream>
#include <sstream>
#include <iterator>
#include <set>

struct Comparator {
  bool operator() (const std::string& a, const std::string& b) const {
    return std::lexicographical_compare(a.begin(), a.end(), b.begin(), b.end(),
                        [](char x, char y) {return tolower(x) < tolower(y);});
  }
};

void word_list() {
  using Iterator = std::istream_iterator<std::string>;

  std::istringstream in{"this is a test this is A TEST THIS IS"};

  std::set<std::string, Comparator> list(Iterator{in}, Iterator{});

  std::copy(list.begin(), list.end(), std::ostream_iterator<std::string>{std::cout, "\n"});

  // output (note how set automatically sorts its content)
  // a
  // is
  // test
  // this
}
```

9 Bind

9.1 Examples

```cpp
#include <functional> // std::bind, std::placeholders
#include <cmath>      // sqrt
#include <iostream>

using namespace std::placeholders;

// x + y
void example_1() {
  auto e = std::bind(std::plus<double>(), _1, _2);

  std::cout << "1 + 2 = " << e(1, 2) << std::endl;

  // output
  // 1 + 2 = 3
}

// (2 * x) - (y / 3)
void example_2() {
  auto e = std::bind(std::minus<double>(),
                     std::bind(std::multiplies<double>(), 2, _1),
                     std::bind(std::divides<double>(), _2, 3));

  std::cout << "(2 * 2) - (6 / 3) = " << e(2, 6) << std::endl;

  // output
  // (2 * 2) - (6 / 3) = 2
}

// (x * x) % y
void example_3() {
  auto e = std::bind(std::modulus<int>(),
                     std::bind(std::multiplies<double>(), _1, _1), _2);

  std::cout << "(4 * 4) % 5 = " << e(4, 5) << std::endl;

  // output
  // (4 * 4) % 5 = 1
}

// sqrt(x * x)
void example_4() {
  auto e = std::bind(sqrt, std::bind(std::multiplies<double>(), _1, _1));

  std::cout << "sqrt(4 * 4) = " << e(4) << std::endl;

  // output
  // sqrt(4 * 4) = 4
}
```

10 Templates

10.1 Function Templates

10.1.1 median

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

template<typename T>
T median(T a, T b, T c) {
  std::vector<T> list{a, b, c};
  std::sort(list.begin(), list.end());
  return list.at(1);
}

void median_demo() {
  std::cout << "median(2, 3, 1): " << median(2, 3, 1) << std::endl;

  // output
  // median(2, 3, 1): 2
}
```

10.1.2 rotate 3 arguments

```cpp
template<typename T>
void rotate_3_arguments(T& a, T& b, T& c) {
  T tmp = a;
  a = b;
  b = c;
  c = tmp;
}

void rotate_3_arguments_demo() {
  int a{0}, b{1}, c{2};
  rotate_3_arguments(a, b, c);
  std::cout << "a: " << a << std::endl;
  std::cout << "b: " << b << std::endl;
  std::cout << "c: " << c << std::endl;

  // output
  // a: 1
  // b: 2
  // c: 0
}
```

10.1.3 read line

```cpp
#include <string>
#include <sstream>

void read_line(std::istream& in) {
  std::cout << "doing absolutely nothing!" << std::endl;
}

template<typename HEAD, typename...ARGS>
void read_line(std::istream& in, HEAD& head, ARGS &...args) {
  in >> head;
  read_line(in, args...);
}

void read_line_demo() {
  int a, b, c;
  std::string input{"1 2 3"};
  std::istringstream in{input};

  read_line(in, a, b, c);

  std::cout << "a: " << a << std::endl;
  std::cout << "b: " << b << std::endl;
  std::cout << "c: " << c << std::endl;

  // output
  // a: 1
  // b: 2
  // c: 3
}

  // if (sizeof...(args)) {
  //   read_line(in, args...);
  // }
```

10.1.4 read line 2

```cpp
#include <iostream>
#include <sstream>

void read_line_2(std::istream& in, std::string& str) {
  getline(in, str);
}

template<typename HEAD, typename...ARGS>
void read_line_2(std::istream& in, HEAD& head, ARGS &...args) {
  in >> head;
  read_line_2(in, args...);
}

void read_line_2_demo() {
  std::istringstream in{"1 some rest 123 \n 2 the rest\n"};
  std::string rest{};
  int first{};

  read_line_2(in,first,rest);

  std::cout << first << std::endl;
  std::cout << rest << std::endl;

  read_line_2(in,first,rest);

  std::cout << first << std::endl;
  std::cout << rest << std::endl;

  // output
  // 1
  //  some rest 123
  // 2
  //  the rest
}
```

10.1.5 calling unimplemented functions

```
struct Lea {
  void call() {
  }
};

struct Julia {
};

template<typename T>
void call_me(T girl) {
  girl.call();
}

void calling_unimplemented_functions() {
  Lea lea{};
  Julia julia{};

  // will call call() on lea
  call_me(lea);

  // calling call_me() with julia will cause a compiler error:
  //
  // call_me(julia);
  //         ^  error: struct Julia has no member named call
}
```

10.1.6 int and double dilemma

```
template<typename T>
void take_two(T one, T two) {
}

void int_and_double_dilemma() {
  // the argument types to the following call to take_two() will be recoginzed by the
  // compiler as an integer followed by a double:
  //
  // take_two(1, 2.2);
  //                ^  error: no matching function for call to take_two(int, double)

  // here are two solutions how to avoid this:

  // with static_cast
  take_two(static_cast<double>(1), 2.2);

  // by specifying template argument
  take_two<double>(1, 2.2);
}
```

10.1.7 string fallacy

```
#include <string>

template<typename T>
void compare(T const& one, T const& two) {
}

void string_fallacy() {
  // the following does not compile. at first it looks like we are passing strings
  // to the function. however, the compiler will recognize the arguments as char
  // arrays:
  //
  // compare("shorter", "longer");
  //                            ^  error: no matching function for call to
  //                               compare(const char [8], const char [7])

  // this works because both arguments have the same size:
  compare("one", "two");

  // if we want the arguments to be passed as strings, we can do the following:
  compare<std::string>("shorter", "longer");
}
```

10.2 Class Templates

10.2.1 class template specialization

```cpp
// this example is currently incomplete. remove it?

#include <iostream>
#include <string>

template<typename T>
struct Versatile;

// we can specialize a templated class with the following syntax:
template<>
struct Versatile<std::string> {
  void print() {
    // std::cout << "string: " << data << std::endl;
//                               ^  error: data was not declared in this scope %bamprog%
  }
};

template<typename T>
struct Versatile {

  Versatile(T t) : data{t} {}

  void print() {
    std::cout << "number: " << data << std::endl;
  }

protected: T data;

};

void class_template_specialization() {
  Versatile<int> vint{1};
  vint.print();

  Versatile<double> vdou{1.1};
  vdou.print();

  Versatile<std::string> vstr{};
  vstr.print();

  // output
  // number: 1
  // number: 1.1
  // string: str
}
```

10.2.2 prohibited construction

```cpp
#include <string>

// we can prohobit the creation of templated types with partial specializations by
// deleting the destructors in the specialized type. The compiler prohibits the
// creation of objects of types which dont have a destructor.
// the following two specialiations will prohibit creating instances of
// Restrictive<char> and Restrictive<T*>.

template<typename T>
struct Restrictive {
};

// prohibit Restrictive<char>
template<>
struct Restrictive<char> {
  ~Restrictive() = delete;
};

// prohibit Restrictive<T*>
template<typename T>
struct Restrictive<T*> {
  ~Restrictive() = delete;
};

void prohibited_construction() {
  Restrictive<int> ints{};
  Restrictive<std::string> strings{};

  // Restrictive<char> chars{};
  //                   ^  error: use of deleted function

  // Restrictive<int*> pints{};
  //                   ^  error: use of deleted function

  // Restrictive<char*> pchars{};
  //                    ^  error: use of deleted function
}
```

10.2.3 vector delegator

```
#include <vector>
#include <string>
#include <algorithm>
#include <iostream>
#include <iterator>

template<typename T>
struct VectorDelegator {

  template<typename ITER>
  VectorDelegator(ITER a, ITER b) : data(a, b) {}

private:

  std::vector<int> data{};

};

void vector_delegator() {
  std::vector<int> v{1,2,3};

  VectorDelegator<int> vd{v.begin(), v.end()};
}
```

11 References And Pointers

11.1 Basic Examples

11.1.1 Reference on int

```cpp
#include <iostream>

void reference_on_int() {
  int j = 5;
  int& r = j;

  std::cout << "j: " << j << std::endl;
  std::cout << "r: " << r << std::endl;

  j = 6;

  std::cout << "j: " << j << std::endl;
  std::cout << "r: " << r << std::endl;

  r = 7;

  std::cout << "j: " << j << std::endl;
  std::cout << "r: " << r << std::endl;

  // output
  // j: 5
  // r: 5
  // j: 6
  // r: 6
  // j: 7
  // r: 7

}
```

11.1.2 Reference on cat

```cpp
#include <iostream>
#include <string>

class Cat {
  std::string name;

public:
  Cat(std::string name) : name{name} {}

  void setName(std::string n) {
    name = n;
  }

  std::string getName() {
    return name;
  }
};

void reference_on_cat() {
  Cat cat{"Hector-Pascal"};
  Cat& same_cat = cat;
  Cat other_cat = cat;

  std::cout << "cat.getName(): " << cat.getName() << std::endl;
  std::cout << "same_cat.getName(): " << same_cat.getName() << std::endl;
  std::cout << "other_cat.getName(): " << other_cat.getName() << std::endl;

  same_cat.setName("Luftdruck");

  std::cout << "cat.getName(): " << cat.getName() << std::endl;
  std::cout << "same_cat.getName(): " << same_cat.getName() << std::endl;
  std::cout << "other_cat.getName(): " << other_cat.getName() << std::endl;

  // output
  // cat.getName():       Hector-Pascal
  // same_cat.getName():  Hector-Pascal
  // other_cat.getName(): Hector-Pascal
  // cat.getName():       Luftdruck
  // same_cat.getName():  Luftdruck
  // other_cat.getName(): Hector-Pascal
}
```

11.2 Reference vs. Pointer

```cpp
#include <iostream>
#include <sstream>
#include <string>

struct Exp1 {
  Exp1(std::ostream& out) : out(out) {}

  void write(std::string something) {
    out << something << std::endl;
  }

  std::ostream& out;
};

struct Exp2 {
  Exp2(std::ostream& out) : out(&out) {}
  //                            ^--- address of out
  //
  //               ^--- reference to out

  void write(std::string something) {
    *out << something << std::endl;
  }

  std::ostream* out;
};

void saving_ref_or_p_as_member() {
  Exp1 exp_1{std::cout};
  Exp2 exp_2{std::cout};

  exp_1.write("this comes from exp_1");
  exp_2.write("this comes from exp_2");

  // output
  // this comes from exp_1
  // this comes from exp_2
}
```

11.3 C++ vs. Java

11.3.1 Reference in C++

```cpp
#include <iostream>
#include <string>

void change(std::string& something) {
  something = "another string";
}

void reference_in_cpp() {
  std::string str = "a string";

  std::cout << "str: " << str << std::endl;

  change(str);

  std::cout << "str: " << str << std::endl;

  // output
  // str: a string
  // str: another string
}
```

11.3.2 Reference in Java

```java
public class ReferenceInJava {

  public static void change(String str) {
    str = "another string";
  }

  public static void main(String[] args) {
    String str = "a string";

    System.out.println(str);

    change(str);

    System.out.println(str);

    // output
    // a string
    // a string
  }

}
```

11.4 Dynamic Heap Memory Management

11.4.1 Unique Pointer

```cpp
#include <iostream>
#include <memory>

std::unique_ptr<int> create_on_heap(int i) {
  return std::unique_ptr<int>{new int{i}};
}

void unique_ptr_demo() {
  // a unique pointer can only exist once. we can not copy it. we can only move the
  // unique pointer to another variable, making the original variable invalid.

  auto p = create_on_heap(10);

  std::cout << std::boolalpha;
  std::cout << "is p valid? " << static_cast<bool>(p) << std::endl;

  auto j = std::move(p);
  // not possible:
  // auto j = p;

  std::cout << "is p valid? " << static_cast<bool>(p) << std::endl;

  // output:
  // is p valid? true
  // is p valid? false
}
```

11.4.2 Shared Pointer

```cpp
#include <iostream>
#include <memory>

struct A {
  A(int n) : n{n} {}
  int n;
};

void shared_ptr_demo() {
  // shared pointers can be copied:

  auto i = std::make_shared<A>(123);

  auto j = i;

  std::cout << "i->n: " << i->n << std::endl;
  std::cout << "(*j).n: " << (*j).n << std::endl;

  // output
  // i->n: 123
  // (*j).n: 123
}
```

12 Compile Time Calculation

12.1 constexpr

```cpp
#include <vector>
#include <iostream>

constexpr int add_at_compile_time(int a, int b) {
  return a + b;
}

void constexpr_demo() {
  // to c will be assigned the value 3 at compile time, because the called function
  // add_at_compile_time() is a constexpr.
  // writing 'int c = 1 + 2;' would be equivalent.
  int c = add_at_compile_time(1, 2);
}
```

12.2 static_assert

```cpp
void static_assert_demo() {
  // with static assert we can do assertions at compile time:
  const int i = 3;
  static_assert(i >= 3, "nooope");

  // if the assertion fails a compilation error is raised:
  // static_assert(i < 3, "nooope");
  // ^  error: static assertion failed: nooope

  // static_assert only works with constant conditions. working with the non-const
  // variable j will fail:
  int j = 3;
  // static_assert(j, "");
  // ^  error: non-constant condition for static assertion
  // ^  error: the value of j is not usable in a constant expression
}
```

12.3 User Defined Literals

```
constexpr double operator"" _cm(long double x){
  return x / 100.;
}

constexpr double operator"" _cm(unsigned long long x){
  return x / 100.;
}

constexpr double operator"" _m(long double x){
  return x;
}

constexpr double operator"" _m(unsigned long long x){
  return x;
}

constexpr double operator"" _km(long double x){
  return x * 1000;
}

constexpr double operator"" _km(unsigned long long x){
  return x * 1000;
}

void user_defined_literals() {
  std::cout << "1_cm: " << 1_cm << " meters" << std::endl;
  std::cout << "1_km: " << 1_km << " meters" << std::endl;

  // output
  // 1_cm: 0.01 meters
  // 1_km: 1000 meters
}
```

12.4 Ring5

```
struct Ring5 {
  explicit constexpr
  Ring5(unsigned x=0u) : val{ x % 5 } {}

  constexpr unsigned value() const { return val; }

  constexpr operator unsigned() const { return val; }

  constexpr bool operator==(Ring5 const &r) const {
    return val == r.val;
  }

  constexpr bool operator!=(Ring5 const &r) const {
    return !(*this == r);
  }

  // this function can not be constexpr because it changes internal state.
  Ring5 operator+=(Ring5 const &r) {
    val = (val + r.value())%5;
    // this error would be raised if we tried to use constexpr:
    //                      ^  error: assignment of member val in read-only object
    return *this;
  }

  Ring5 operator*=(Ring5 const&r) {
    val = (val * r.value())%5;
    return *this;
  }

  constexpr Ring5 operator+(Ring5 const &r) const {
    return Ring5{val+r.val};
  }

  constexpr Ring5 operator*(Ring5 const &r) const {
    return Ring5{val*r.val};
  }

private:
  unsigned val;
};
```

13 Good To Know

13.1 Default floating point type

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

13.2 Assigning floating point to int

```cpp
#include <iostream>

void assigning_floating_point_to_int() {
  // if we assign a floating point to a variable of type integer, it will
  // automatically be casted.
  int autoconverted_to_int = 7.5;

  std::cout << autoconverted_to_int << std::endl;

  // output:
  // 7
}
```

13.3 Bool is an integer

```cpp
#include <iostream>

void bool_is_an_integer() {
  std::cout << "5 + true:  " << 5 + true  << std::endl;
  std::cout << "5 + 1:     " << 5 + 1     << std::endl;
  std::cout << "6 + false: " << 6 + false << std::endl;
  std::cout << "6 - 0:     " << 6 - 0     << std::endl;

  // output
  // 5 + true:  6
  // 5 + 1:     6
  // 6 + false: 6
  // 6 - 0:     6
}
```

13.4 Initializing variables

```cpp
void initializing_variables() {
  int a{1};      // initialized with 1
  int b{};       // default initialization (in case of int zero)
  int c;         // undefined behaviour
  static int d; // zero initialized

  std::cout << "a: " << a << std::endl;
  std::cout << "b: " << b << std::endl;
  std::cout << "c: " << c << std::endl;
  std::cout << "d: " << d << std::endl;

  // output
  // a: 1
  // b: 0
  // c: 7
  // d: 0
}
```

13.5 Floating points cant be unsigned

```cpp
// unsigned double ud = 1.1;
//                 ^  error: signed or unsigned invalid for ud
// unsigned float  uf = 1.1;
//                 ^  error: signed or unsigned invalid for uf

unsigned int    ui = 1;
```

13.6 Literals

```
void chars() {
  char a;
  a = 'a';
  a = '\n';
  a = '\x0a';
}

void integers() {
  int i =         1;
  long l =        1L;
  long long ll = 1LL;
}

void unsigned_integers() {
  unsigned int ui =         1u;
  unsigned long ul =        1ul;
  unsigned long long ull = 1ull;
}

void octal_hex_full() {
  int octal =       020;
  int hex =         0x1f;
  long long full =  0XFULL;
}

void floating_points() {
  float f =         0.1f;
  double d1 =       .33;
  double d2 =       1e9;
  long double d3 = 42.E-12L;
  long double d4 = .3l;
}

void char_array() {
  char a[] = "hello"; // char[6] {'h', 'e', 'l', 'l', 'o', '\0'}
                      // '\0' terminates the string
}
```

13.7 Weird string syntax

```
void weird_string_syntax() {
  std::string  s1 = "line1"
                    "line2"
                    "line3";

  std::cout << s1 << std::endl;

  // output
  // line1line2line3
}
```

13.8 Arithmetics with int and double

```
void arithmetics_with_int_and_double() {
  // doing divisions only with integers will return an integer
  std::cout << "7   / 2:    \t\t" << 7 / 2   << std::endl;
  // as soon as a floating point is involved, the result is a floating point too
  std::cout << "7   / 2.0:  \t\t" << 7 / 2.0 << std::endl;
  std::cout << "7.0 / 2:    \t\t" << 7.0 / 2 << std::endl;

  // output
  // 7   / 2:        3
  // 7   / 2.0:      3.5
  // 7.0 / 2:        3.5

  // assigning the division of two integers to a double variable results in an int
  double x = 7 / 2;
  std::cout << "x: " << x << std::endl;
  // output
  // x: 3
}

void division_by_zero_demo() {
  // division by zero is undefined behaviour. the following code compiles and does
  // not cause an exception at runtime:
  // std::cout << "1 / 0: \t\t" << 1 / 0 << std::endl;
}
```

13.9 Unspecified invocation order

```
int get_a() {
  std::cout << "get_a()" << std::endl;
  return 0;
}

int get_b() {
  std::cout << "get_b()" << std::endl;
  return 0;
}

void random_function_name(int a, int b) { }

void unspecified_invocation_order() {
  // if we call functions to pass arguments to another function, we don't know in
  // which order the functions are called.
  random_function_name(get_a(), get_b());

  // output could for example be
  // get_b()
  // get_a()
}
```

13.10 Factory function

```cpp
#include <stdexcept>

namespace Galaxy {
  struct Planet {
    Planet() = default;

    Planet(int distance) {
      if (distance < 0) throw std::invalid_argument("");
    }

    int distance;
  };

  Planet make_planet(int distance)
  try {
    return Planet{distance};
  } catch(std::invalid_argument e) {
    return Planet{};
  }
}
```

13.11 Args

```cpp
#include <iostream>
#include <iterator>
#include <algorithm>
#include <vector>

int main(int argc, char* argv[]) {
  std::vector<std::string> params{};
  std::copy(argv + 1, argv + argc, std::back_inserter(params));

  std::copy(params.begin(), params.end(),
            std::ostream_iterator<std::string>{cout, "\n"});
}
```

14 Appendix

14.1 Random Code

14.1.1 PIMPL idiom

```
#ifndef PERSON_H_
#define PERSON_H_
#include <memory>
#include <string>
#include <vector>
#include <iosfwd>

class Person {
  std::shared_ptr<class PersonImpl> person;

  Person(std::shared_ptr<class PersonImpl> person) :
    person { person } {}

public:
  Person(std::string name);
  Person(std::string name, Person father, Person mother);
  ~Person();
  void addChild(Person child);
  std::string getName() const;
  Person findChild(std::string name) const;
  void killChild(Person child);
  void killMe();
  operator bool() const;
  void print(std::ostream &out) const;
};

#endif /* PERSON_H_ */
```

```
#include "Person.h"
#include <iostream>
#include <algorithm>
#include <functional>

using PersonPtr=std::shared_ptr<class PersonImpl>;
using WeakPersonPtr=std::weak_ptr<class PersonImpl>;

class PersonImpl : public std::enable_shared_from_this<PersonImpl> {
  std::string name;
  WeakPersonPtr father; // don't lock parent objects
  WeakPersonPtr mother;
  std::vector<PersonPtr> children;

  PersonPtr myLock() {
    try {
      auto me=shared_from_this(); // throws when called from dtor!
      return me;
    } catch(std::bad_weak_ptr const &ex){}
    std::cout << "++++already dead? " << name<< '\n';
    return PersonPtr{}; // already dead
  }

public:
  PersonImpl(std::string name,PersonPtr father,PersonPtr mother)
  :name{name},father{father},mother{mother}{
    // can not do shared_from_this here!
    //no if(father) father->addChild(shared_from_this());
  }

  ~PersonImpl() {
    std::cout << "killing me: "<< name << '\n';
    //killMe(); // can not call shared_from_this() in dtor!
  }

  void addChild(PersonPtr child){
    children.push_back(child);
  }

  std::string getName() const {
    return name;
  }

```

```
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

```
#ifndef WORD_H_
#define WORD_H_

#include <iosfwd>
#include <string>

#include <boost/operators.hpp>

struct Word : boost::less_than_comparable<Word>, boost::equality_comparable<Word> {
  Word():data{} {}
  Word(std::string);

  bool isValid() { return !data.empty(); }
  void read(std::istream&);
  void write(std::ostream& os) const { os << data; };

  bool operator<(Word const& r) const { return toLower() < r.toLower(); }
  bool operator==(Word const& r) const { return toLower() == r.toLower(); }
  std::string operator+(std::string const& r) const { return data + r; }
private:
  std::string data;
  std::string toLower() const;
};

std::istream& operator>>(std::istream&, Word&);
std::ostream& operator<<(std::ostream&, Word const&);

#endif
```

```cpp
#include "word.h"

#include <istream>
#include <ostream>
#include <string>
#include <algorithm>
#include <sstream>

Word::Word(std::string word):data{} {
  std::istringstream in{word};
  read(in);
}

void Word::read(std::istream& in) {
  data.clear();

  char c{};
  while(in.get(c)) {
    if(std::isspace(c) && !isValid()) continue;

    if(std::isalpha(c)) {
      data.push_back(c);
    } else {
      break;
    }
  }
}

std::string Word::toLower() const {
  std::string lowered{data};
  std::transform(lowered.begin(), lowered.end(), lowered.begin(), tolower);
  return lowered;
}

std::istream& operator>>(std::istream& l, Word& r) {
  r.read(l);
  return l;
}

std::ostream& operator<<(std::ostream& l, Word const& r) {
  r.write(l);
  return l;
}
```

14.1.3 Ring

```
struct Ring5 {

  explicit constexpr Ring5(unsigned x=0u) : val{ x % 5 } {}

  constexpr unsigned value() const {
    return val;
  }

  constexpr operator unsigned() const {
    return val;
  }

  constexpr bool operator==(Ring5 const &r) const {
    return val == r.val;
  }

  constexpr bool operator!=(Ring5 const &r) const {
    return !(*this == r);
  }

  Ring5 operator+=(Ring5 const &r) {
    val = (val + r.value()) % 5;
    return *this;
  }

  Ring5 operator*=(Ring5 const&r) {
    val = (val * r.value()) % 5;
    return *this;
  }

  constexpr Ring5 operator+(Ring5 const &r) const {
    return Ring5{val+r.val};
  }

  constexpr Ring5 operator*(Ring5 const &r) const {
    return Ring5{val*r.val};
  }

private:

  unsigned val;

};
```

14.1.4 Sack

```
#ifndef SACK_H_
#define SACK_H_

#include<vector>
#include<map>
#include <iterator>

template <typename T, template<typename...> class C=std::vector>
class Sack
{
   using SackType=C<T>;
   using size_type=typename SackType::size_type;
   SackType theSack{};

public:
   Sack(std::initializer_list<T> const &items) :
     theSack(items) {
   }

   bool empty() const {
     return theSack.empty();
   }

   size_type size() const {
     return theSack.size();
   }

   void putInto(T const &item) {
     theSack.push_back(item);
   }

   T getOut() {
     if (empty()) {
       throw std::logic_error{"empty Sack"};
     }

     auto index = static_cast<size_type>(rand() % size());

     T retval { theSack.at(index) };
     theSack.erase( theSack.begin() + index);
     return retval;
   }
};
#endif /* SACK_H_ */
```

```
template <typename T>
class Sack<T, std::map> {
  using SackType = std::map<T, unsigned>;
  using size_type=typename SackType::size_type;
  SackType theSack{};

public:
  bool empty() {
    return theSack.empty();
  }

  size_type size() {
    size_type total = 0;
    for (std::pair<T, unsigned> pair : theSack) {
      total += pair.second;
    }

    return total;
  }

  void putInto(T const &item) {
    theSack[item]++;
  }

  T getOut() {
     if (empty()) {
       throw std::logic_error{"empty Sack"};
     }

     auto index = static_cast<size_type>(rand() % size());

     std::pair<T, unsigned> element = *std::next(theSack.begin(), index);

     if (element.second == 1) {
       theSack.erase(element.first);
     } else {
       theSack[element.first] = element.second -1;
     }
     return element.first;
  }
};
```

14.1.5 Highlander

```
#ifndef LIMITNUMBEROFINSTANCES_H_
#define LIMITNUMBEROFINSTANCES_H_

#include <stdexcept>

template <typename TOBELIMITED, unsigned int maxNumberOfInstances>
class LimitNofInstances {

  static unsigned int counter;

protected:

  void checkNofInstances() {
    if(counter == maxNumberOfInstances) throw std::logic_error("too many instances");
  }
  LimitNofInstances() {
    checkNofInstances();
    ++counter;
  }
  ~LimitNofInstances() {
    --counter;
  }
  LimitNofInstances(const LimitNofInstances &other){
    checkNofInstances();
    ++counter;
  }
};

template <typename TOBELIMITED, unsigned int maxNumberOfInstances>
unsigned int
LimitNofInstances<TOBELIMITED,maxNumberOfInstances>::counter(0);

#endif /* LIMITNUMBEROFINSTANCES_H_ */

// using it:
class One : LimitNofInstances<One, 1>{ /*...*/};
```

14.1.6 dynArray

```
#ifndef DYNARRAY_H_
#define DYNARRAY_H_

#include <vector>

template<typename T>
struct dynArray {

  using DynArrayType = std::vector<T>;
  using size_type = typename DynArrayType::size_type;
  using value_type = typename DynArrayType::value_type;
  using iterator = typename DynArrayType::iterator;
  using const_iterator = typename DynArrayType::const_iterator;
  using allocator_type = typename DynArrayType::allocator_type;

  // Constructors

  explicit dynArray(const allocator_type& alloc = allocator_type()):container{alloc} {}
  explicit dynArray(size_type n):container{n} {}
  explicit dynArray(size_type n, value_type const& val, allocator_type const& alloc = allocator_type

  template<class InputIterator>
  dynArray(InputIterator first, InputIterator last, allocator_type const& alloc = allocator_type()):

  dynArray(dynArray const& x):container{x.container} {}
  dynArray(dynArray const& x, allocator_type const& alloc):container{x.container, alloc} {}
  dynArray(std::initializer_list<value_type> il, allocator_type const& alloc = allocator_type()):cor

  // Iterators

  iterator begin() {
    return container.begin();
  }

  const_iterator begin() const {
    return container.begin();
  }

  iterator end() {
    return container.end();
  }

  const_iterator end() const {
    return container.end();
  }

```

```
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_ */
```