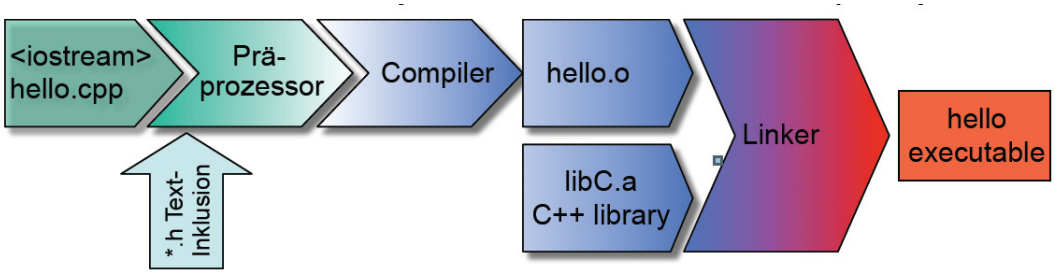

CUTE

```
#include "cute.h"
#include "ide_listener.h"
#include "cute_runner.h"

namespace {
  void testExceptionThrow() {
    throw std::runtime_error(”error”);
  }
  void thisIsATest() {
    ASSERTM("start writing tests", false);
    ASSERT_EQUAL(1, 1);
  }
  void thisThrowsAnException() {
    ASSERT_THROWS(testExceptionThrow(),std::runtime_error);
  }
  void runSuite(){
    cute::suite s;
    //TODO add your test here
    s.push_back(CUTE(thisIsATest));
    s.push_back(CUTE(thisThrowsAnException));
    cute::ide_listener lis;
    cute::makeRunner(lis)(s, "The Suite");
  }
}
```

boost::assign

```
#include <vector>
#include <map>
#include <string>
#include "boost/assign/std/vector.hpp" // for += operator
#include "boost/assign.hpp"

using namespace std;
using namespace boost::assign;

int main(void)
{
  vector<int> v;
  v += 0,4,2,6,89,3; // boost::assign

  map<string,int> m;
  m["Frodo"]=21;
  insert(m)("Bilbo",123)("Gandalf",502)("Harry Potter",-42); // boost::assign
}
```

ostream_iterator

```
#include <iostream> /* use <iosfwd> if you don't need the stream objects */
#include <algorithm>
#include <iterator>
#include <vector>

using namespace std;

int main(void)
{
  vector<int> v;
  // abfüllen...
  // zeilenweise ausgeben:
  copy(v.begin(),v.end(), ostream_iterator<int>(cout,"\n"));
  return 0;
}
```

istream_iterator

```
#include <iostream> /* use <iosfwd> if you don't need the stream objects */
#include <algorithm>
#include <iterator>
#include <vector>

using namespace std;

int main(void) // int's in vector einlesen und wieder ausgeben
{
      vector<int> v;
      typedef istream_iterator<int> input;
      copy(input(cin),input(),back_inserter(v));
      copy(v.begin(),v.end(),ostream_iterator<int>(cout,"\n"));
}
```

Vector

```
#include <iostream> /* use <iosfwd> if you don't need the stream objects */
#include <algorithm>
#include <iterator>
#include <vector>
#include <string>

using namespace std;

vector<char>::iterator::distance_type charcount()
{
  vector<char> v;
  char input;
  while(cin) {
    cin >> input;
    v.push_back(input);
  }
  return v.size() - 1;
}

int main(void)
{
  vector<char>::iterator::distance_type count = charcount();
  cout << "a): " << count << endl;
}
```

Class / Copy Constructor / Separated .h & .cpp / <<, >>, <, = Operators

```
// Wort.h
#ifndef WORT_H_
#define WORT_H_

#include <iostream> /* use <iosfwd> if you don't need the stream objects */
#include <string>
#include <vector>

class Wort
{
public:
   Wort(std::string wort="");
   Wort(const Wort &copy);

   std::string getValue() const;
   int getCount() const;
   void setCount(int value);
   Wort& operator=(const Wort &rValue);
   std::ostream& print(std::ostream &out) const;
   std::istream& read(std::istream &in);
   static bool compare(char c1, char c2);
private:
   std::string _word;
   int _count;
};

bool operator<(Wort const &left, Wort const &right);
std::ostream& operator<<(std::ostream &out, const Wort &w);
std::istream& operator>>(std::istream &in, Wort &w);
#endif /*WORT_H_*/


// Wort.cpp
#include "Wort.h"
#include "boost/bind.hpp"
#include <string>
#include <iterator>
#include <iostream> /* use <iosfwd> if you don't need the stream objects */
#include <complex>
#include <cctype>

using namespace std;

Wort::Wort(string wort) : _word(wort), _count(1) { }

Wort::Wort(const Wort &copy) : _word(copy._word), _count(copy._count) { }

std::string Wort::getValue() const {
   return _word;
}

int Wort::getCount() const {
   return _count;
}
```

```
Wort& Wort::operator=(const Wort &rValue) {
   _word = rValue._word;
   _count = rValue._count;
   return *this;
}
void Wort::setCount(int value) {
   _count = value;
}

bool Wort::compare(char c1, char c2) {
   return tolower(c1) < tolower(c2); // a case-insensitive comparison function
}

std::ostream& Wort::print(std::ostream &out) const {
   out << _word;
   return out;
}

std::istream& Wort::read(std::istream &in) {
   if (in)
   {
      in >> _word; // read 1 word
   }
   return in;
}


bool operator<(Wort const &left, Wort const &right) {
   const char * leftValues = left.getValue().data();
   const char * rightValues = right.getValue().data();

   return lexicographical_compare(
      leftValues,
      leftValues + left.getValue().length(),
      rightValues,
      rightValues + right.getValue().length(),
      boost::bind(&Wort::compare, _1, _2));
}


ostream& operator<<(ostream &out, const Wort &w) {
   return w.print(out);
}

std::istream& operator>>(std::istream &in, Wort &w) {
   return w.read(in);
}
```

Private Implementation (PIMPL) Ideom *(nicht vollständig)*

```
// Car.h
#ifndef CAR_H
#define CAR_H

#include "Thing.h"

#include <string>
#include <vector>
#include <iostream> /* use <iosfwd> if you don't need the stream objects */
#include <boost/shared_ptr.hpp>

namespace Things
{
   typedef boost::shared_ptr<class CarImpl> CarImplPtr;

   class Car : public Thing {
   public:
      Car(std::string brand = "[]", int seatsCount = 1, ThingColor color = none);
      virtual ~Car();

      virtual std::string getMoveString() const;

   private:
      CarImplPtr _pImpl;
   };

   std::ostream& operator<<(std::ostream &out, const Car& p);
}
#endif


// Car.cpp
#include "Thing.h"
#include "Car.h"
#include <vector>
#include <iostream> /* use <iosfwd> if you don't need the stream objects */
#include <boost/shared_ptr.hpp>

using namespace std;

namespace Things
{
   class CarImpl {
   public:
      static CarImplPtr make() {
         return CarImplPtr(new CarImpl());
      }
      static void dispose(CarImplPtr toDispose) {
         toDispose->disposeCar();
      }


      CarImpl() {
      }

      ~CarImpl() {
      }
```

```
    std::string getMoveString() const {
      return "[drive]";
    }

  private:
    CarImpl(const CarImpl& copy);
    CarImpl& operator=(CarImpl& rValue);

    void disposeCar() {
    }

  };


  /*************************************************************************
   * Public class definition
   ************************************************************************/

  Car::Car(std::string brand, int seatsCount, ThingColor color)
    : Thing(brand, seatsCount, color) {
    _pImpl = CarImpl::make();
  }

  Car::~Car() {
  }

  std::string Car::getMoveString() const {
    return _pImpl->getMoveString();
  }


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

Template Function (in *.h file)

```
template<class Iter, class FnCheck, class FnExec>
inline void do_multiple(Iter begin, Iter end, FnCheck check, FnExec exec)
{
  do { exec(begin, end); }
  while (check(begin, end));
}
```

Algorithm / Palindrom Parsing

```
#include <iostream> /* use <iosfwd> if you don't need the stream objects */
#include <algorithm>
#include <string>
#include <ctype.h>

bool isPalindrom(std::string input) { // creates implicitly a copy of input
      std::transform(input.begin(), input.end(), input.begin(), tolower);
      return std::equal(input.begin(), input.end(), input.rbegin());
}

int main()
{
      std::cout << isPalindrom("ReLiefpFeiLeR") << std::endl;
      system("pause");
}
```

Exception Handling / Template Class

```
#include <algorithm>
#include <iostream> /* use <iosfwd> if you don't need the stream objects */
#include <string>
#include <vector>
#include <exception> /* for bad_alloc, bad_cast, bad_typeid, runtime_error, …*/

using namespace std;

class VectorException : public exception
{
  virtual const char* what() const throw()
  {
    return "Out Of Range Exception Occured";
  }
};


template<typename T>
  class Vector
{
public:
  typedef typename vector<T>::size_type size_type;
  typedef typename vector<T>::value_type value_type;
  typedef typename vector<T>::const_iterator const_iterator;
  typedef typename vector<T>::reverse_iterator reverse_iterator;
  typedef typename vector<T>::iterator iterator;

  Vector() { }

  Vector<T>(iterator start, iterator end)
    : _storage(start, end) { }

  T& operator[](size_type index) {
    return at(_storage.size() - index - 1);
  }

  T& at(size_type idx) {
    if (idx < 0 || idx > _storage.size() - 1)
    {
      throw VectorException();
    }
    return _storage[idx];
  }

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

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

  void push_back(T value) {
    _storage.push_back(value);
  }

private:
  vector<T> _storage;
};
```

Functor *(nicht vollständig)*

```
class FillFunctor : unary_function<string, void> /* häufig auch struct */
{
public:
   FillFunctor(vector<string>& input, set<vector<string>>& combinations)
      : _combinations(combinations), _input(input) { }

   void operator()(string& input) {
      vector<string> entries;

      rotate_copy(
         _input.begin(),
         find(_input.begin(), _input.end(), input),
         _input.end(),
         back_inserter(entries));

      _combinations.insert(entries);
   }
private:
   set<vector<string>>& _combinations;
   vector<string>& _input;
};
```

typeid()

```
#include <iostream> /* use <iosfwd> if you don't need the stream objects */

template <typename T>
void printIdRef(T const & t) {
   std::cout << "printIdRef(T const &t): " << typeid(t).name() << std::endl;
}

template <typename T>
void printIdVal(T t) {
   std::cout << "printIdVal(T const &t): " << typeid(t).name() << std::endl;
}

int main(){
   printIdRef("hello");
   printIdVal("hello");
   system("pause");
}
```

ostream_iterator / istream_iterator combined

```
#include <iostream> /* use <iosfwd> if you don't need the stream objects */
#include <algorithm>
#include <string>
#include <sstream>

using namespace std;

// lese 2. Arg. elementweise ein, gebe ihn linienweise aus
int main(int argc, char* argv[]) {
   stringstream ss; //stringstream ss("das ist\nein\ntest");
   ss << argv[2];
   istream_iterator<string> in(ss);
   istream_iterator<string> inEnd;
   ostream_iterator<string> out(cout, "\n");
   copy(in, inEnd, out);
   system("pause");
}
```

Iterator / advanced boost::bind

```
#include <iostream> /* use <iosfwd> if you don't need the stream objects */
#include <algorithm>
#include <iterator>
#include <functional>

using namespace std;

template<typename T = unsigned int>
class counting_iterator : public iterator<input_iterator_tag, T>
{
public:
   typedef typename T value_type; /* for iterator_traits */
   typedef typename T* pointer; /* for iterator_traits */
   typedef typename T& reference; /* for iterator_traits */

   counting_iterator(T number)
      : _number(number) { }

   counting_iterator& operator=(const counting_iterator& other) {
      _number = other._number;
      return(*this);
   }

   bool operator==(const counting_iterator& other) const {
      return (_number == other._number);
   }

   bool operator!=(const counting_iterator& other) const {
      return !(*this == other);
   }

   counting_iterator& operator++() {
      ++_number;
      return *this;
   }

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

   T& operator*() {
      return _number;
   }

   T* operator->() {
      return &(*this);
   }

private:
   T _number;
}
```

```
template<typename T = unsigned int>
class is_prime
   : public unary_function<T, bool>
{
public:
   is_prime() { }
   bool operator()(T number)
   {
      if (number == 1) return true;
      if (number == 2) return false; // little optimization

      counting_iterator<T> numStart(2);
      counting_iterator<T> numEnd((number / 2) + 1);
      return (find_if(
                numStart,
                numEnd,
                boost::bind(&is_prime::is_factor, this, _1, number)) == numEnd);
   }
private:
   bool is_factor(int factor, int number) { return (number % factor == 0); }
};


// pseudocode:
int main1(){
   counting_iterator<> anfang(1);
   counting_iterator<> ende(100);
   ostream_iterator<unsigned int> output(cout,", ");
   remove_copy_if(
      anfang,
      ende,
      output,
      boost::bind(std::logical_not<bool>(), boost::bind(is_prime<>(), _1)));

   system("pause");
   return 0;
}
```

boost::function

```
#include "boost/bind.hpp"
#include "boost/function.hpp"
#include <iostream> /* use <iosfwd> if you don't need the stream objects */
#include <algorithm> // find_if
#include <functional> // logical_not
#include <vector>

int main() // suche in v nach der ersten durch 17 teilbaren zahl
{
  std::vector<int> v;
  v.push_back(2);
  v.push_back(345);
  v.push_back(456);
  v.push_back(1105); // gesucht
  v.push_back(6);
  v.push_back(17);
  v.push_back(1);
  v.push_back(9);
  v.push_back(8);

  boost::function<bool(int)> div17 = boost::bind(
                                     std::logical_not<bool>(),
                                     boost::bind(std::modulus<int>(), _1, 17));

  std::vector<int>::const_iterator iter = find_if(v.begin(), v.end(), div17);
  // std:: vor find_if nicht benötigt
  std::cout << *iter << std::endl;
  system("PAUSE");
}
```

boost::ref *(nicht vollständig)*

```
#include <boost/bind.hpp>
#include <boost/ref.hpp>
#include <algorithm>

void Person::printKinder(std::ostream& os) {
  for_each (
    kinder.begin(),
    kinder.end(),
    boost::bind(&Person::print, _1, boost::ref(os)));
}
```

array & algorithm

```
#include <iostream> /* use <iosfwd> if you don't need the stream objects */
#include <iterator>

int main(int argc, char* argv[])
{
  // argv[0] ist der Programmname (char[])
  // argv[1] erstes Argument, usw. (char[])

  using namespace std;
  char const s[] = "Hello World"; // besser string, oder boost::array<char,12> s;

  ostream_iterator<char> outc(cout, ".");
  copy(s, s+sizeof(s), outc);

  cout<< endl;cout<< s << endl;
  cout<< endl;
}
```

RAII (Resource Acquisition Is Initialization) – Invariante sicherstellen mit boost::shared_ptr

```
#include <fstream>
#include <iostream> /* use <iosfwd> if you don't need the stream objects */
#include <boost/shared_ptr.hpp>

typedef boost::shared_ptr<std::ostream> FstrPtr;

FstrPtr createFile() {
  FstrPtr file(new std::ofstream("hallo.txt", std::ios::app));

  if(!*file) throw "file unwritable";
  return file;
}

int main() {
  try {
    FstrPtr file = createFile();
    (*file) << "Hello world\n";
  }
  catch(char const* & e) {
    std::cerr << e << std::endl;
  }
}
```

File Stream

```
#include <fstream>
#include <iostream> /* use <iosfwd> if you don't need the stream objects */

int main()
{
  std::ofstream theOutfile("out.txt");

  if (!theOutfile) {
    std::cerr << "Dateifehler" << std::endl;
  }
  else {
    // gib irgend etwas aus
    theOutfile << "Kuck mal wer da spricht…" << std::endl;
  }
}
```

copy_if

```
#ifndef COPY_IF_H_
#define COPY_IF_H_

template<class InT, class OutT, class PredT>
OutT copy_if(InT start, InT end, OutT result, PredT predicate) {
  while ( start != end ) {
    if ( predicate(*start) ) {
      *result = *start;
      ++result;
    }
    ++start;
  }
  return result; // nächster freier Platz im Zielbereich
}
#endif /*COPY_IF_H_*/
```