Informatik–8 Pages

Variables

Type	Description
`void`	no value
`bool`	true/false
`char`	single character
`int`	integer number
`unsigned int`	natural number
`float`	32bit floating point number
`double`	64bit floating point number

Without specifying double is assumed over float. Similarly, numbers without decimal point are assumed to be int.

Mixed expression order of number types: bool=char < **int** < unsigned int < float < **double**

Note that if a mixed expression contains bool and char, they are converted to int first.

To avoid long type names, `using` can be used:

```
using uint = unsigned int;
```

Control Structures

If-else statements:

```
if (condition) {// do something
} else if{// do something else
} else { // do another thing
}
```

Loops:

```
for (int i = 0; i < 10; i++) {
    // do something 10 times
}
while (condition) {
    // do something while condition is true
}
do {
    // do something at least once
} while (condition);
```

Pay attention to the semicolon after the `do-while` loop!

In order to break out of loops early, use `break;`. To skip the current iteration, use `continue;` which still performs the increment in a for loop.

Semantics for the for loop are:

1. initialization
2. evaluate condition, if false exit loop
3. execute loop body
4. execute update expression
5. go to step 2

Semantics for the while loop are:

1. evaluate condition, if false exit loop
2. execute loop body
3. go to step 1

The switch statement can be used for multiple branches:

```
switch (variable) {
    case value1:
        // do something
        break;
    case value2:
        // do something else
        break;
    default:
        // do another thing
        break;
}
```

Break statements are nessecairy to avoid fallthrough (which can be intended sometimes).

Looping over a container:

```
for (auto it = v.begin(); it != v.end(); ++it) {
    // use *it
}
for (auto& elem : v) {
    // use elem with reference
}
for (auto elem : v) {
    // use elem
}
```

Functions

Functions are defined with a return type, a name and parameters:

```
T fname(T1 param1, T2 param2, ...) {
    // function body
    return value; // of type T
}
fname(expr1, expr2, ...); // function call
```

It is important to make sure that there always is a return statement, except when using `void` as return type.

Function parameters are local variables. Each call declares a new set of parameters.

Functions can be forward declared using `T fname(T1, T2, ...);` before the definition of it.

Libraries and Namespaces

Functions can be exported into a file **`library.cpp`** and declared in a header file **`library.h`**. The header file can then be included using `#include "library.h"`.

To avoid name clashes, namespaces can be used:

```
namespace mynamespace {
    // some functions
}
mynamespace::fname(...); // call from namespace
    // or
using namespace mynamespace;
```

Vectors

Vectors store multiple values of the same type. They are included using one of the following lines:

```
std::vector<T> name; // empty vector
std::vector<T> name(size); // default initialized
std::vector<T> name(size, val); // size elements = val
std::vector<T> name = {val1, val2, ...};
```

To access elements use `name[index]` or the `name.at(index)`. This is an l-value. Later prevents out of bounds acces. Basic operations:

- `v.size()` returns unsigned int
- `std::ssize(v)` returns int
- `v.push_back(val);`
- `v.begin()`, `v.end()` return iterators

A vector of vectors can be used to create 2D arrays. For short notation, use the auto keyword or `using`.

Sets

Requires `#include <set>`. Sets store unique elements in no particular order.

```
b = some_vec.begin(), some_vec.end();
std::set<int> my_set(b, e); // from range
std::set<int> my_set; // empty set
my_set.insert(val); // insert value
my_set.erase(val); // erase value
bool exists = my_set.count(val); // check existence
```

Arrays

Arrays store multiple values of the same type in a fixed size.

```
int arr[10]; // array of 10 ints
int arr2[5] = {1,2,3,4,5}; // initialized array
```

Stuff has to be done element-wise. Also length has to be known at compile time.

Debugging

`#include <cassert>` can be used to include assertions in the code using the command `assert(condition);`.

Characters

Often times its more useful to use character arithmetic:

```
for (char c = 'a'; c <= 'z'; c++) {}
```

Strings

Requires `#include <string>`. Strings store sequences of characters.

```
std::string s1; // empty string
std::string s2 = "hello"; // initialized string
s1.length(); // length of string
s1 += s2; // concatenate
s1[i]; // access character at index i
```

Integers

Computers use binary representation to store data. An integer is of the form:

$$b_n 2^n + b_{n-1} 2^{n-1} + \dots + b_1 2^1 + b_0 2^0.$$

where each b_i is either 0 or 1. The normal `int` type uses 32 bits (4 bytes) to store values from -2^{31} to $2^{31} - 1$.

Unsigned integers go from 0 to $2^{32} - 1$.

To use binary numbers in code, prefix them with `0b` or for hexadecimal with `0x`.

```
int a = 0b1101; // binary for 13
int b = 0x1A;   // hexadecimal for 26
```

Floating Point Numbers

A floating point number system is given by

$$F(\beta, p, e_{\min}, e_{\max}).$$

Where β is the base, p the precision, $e_{\min}$ the minimum exponent and $e_{\max}$ the maximum exponent. A floating point number is represented as

$$\pm d_0.d_1 d_2 \ldots d_{p-1} \times \beta^e.$$

A *normalized* floating point number has $d_0 \neq 0$. The largest representable number is

$$\max = (1 - \beta^{-p}) \times \beta^{e_{\max}+1}.$$

The smallest normalized representable number is

$$\min_{\text{norm}} = \beta^{e_{\min}}.$$

For calculation, the numbers are denormalized, added exactly and rounded to the nearest representable number.

The total amount of different floating point numbers is

$$2 \times (\beta - 1) \times \beta^{p-1} \times (e_{\max} - e_{\min} + 1).$$

Powers of 2

2^5	32	2^{-5}	0.03125
2^4	16	2^{-4}	0.0625
2^3	8	2^{-3}	0.125
2^2	4	2^{-2}	0.25
2^1	2	2^{-1}	0.5
2^0	1	2^0	1

Base Conversions

To convert a number x from base 10 to base b, repeatedly divide x by b and keep track of the remainders. The remainders form the digits of the new base number, starting from the last remainder to the first.

```
void convertToBase(int x, int b) {
  std::vector<int> digits;
  while (x > 0) {
    digits.push_back(x % b);
    x /= b;
  }}
```

Reference Types

A reference is an alias for another variable. It is declared with `&type name = var;` and must be initialized at declaration. It can be used like the original variable.

In functions, references can be used to avoid copying large data structures (pass by reference) or even by constant reference to avoid modification:

```
void func(std::vector<int> &v) {
  // use v without copying
}
void func2(const std::vector<int> &v) {
    // use v without copying or modifying
}
```

Functions can also return references:

```
int& getElement(std::vector<int> &v, int index) {
    return v[index];
}
```

Warning: Returning a reference to a local variable leads to undefined behavior.

Pointers

A pointer stores the memory address of a variable. It is declared with `type* name = &var;`. The address of a variable is obtained with `&var` and the value at the address with `*ptr`. Pointers can be null using `nullptr`.

Furthermore, pointer arithmetic can be used to compute different things

```
int* ptr = new int[6];
ptr + 3; // Temporary shifts
ptr - 3;
++ptr; // Modifies ptr
--ptr;
ptr+=3;
ptr1-ptr2; //Calculates distance in elements
ptr1 < ptr2; //Comparison
```

Const

The `const` keyword can be used to declare variables that cannot be modified after initialization:

```
const int x = 42; // x cannot be modified
```

Const references mean that the variable cannot be modified through the reference:

```
double a = 3.0;
double& b = a; // b can modify a
const double& c = a; // c cannot modify a
c = 4.0; // error
a = 4.0; b = 5.0; // ok
double& d = c; // error
```

For pointers, `const` can be used in different ways:

```
int value = 42;
int* ptr1 = &value; // pointer to int
const int* ptr2 = &value; // pointer to const int
int const* ptr2b = &value; // same as above
int* const ptr3 = &value; // const pointer to int
const int* const ptr4 = &value; // const pointer to const int
int const* const ptr4b = &value; // same as above
```

Pointers to const int cannot be used to modify the int value. Const pointers cannot be changed to point to another address.

For iterators, `const_iterator` can be used to prevent modification of the elements. Do NOT use `const auto it` as this makes the iterator itself const, not the elements it points to.

Const member functions cannot modify the object they belong to. They are declared with `T fname(...) const {...}`.

Iterators

Iterators are objects that point to elements in a container. They can be used to traverse the container. Common operations are:

- `it++` to move to the next element
- `*it` to access the element
- `it != end` to check if the end is reached

Dynamic Memory

Memory can be allocated at runtime using `new` and *must* be freed using `delete` to avoid memory leaks.

```
int* p = new int(42); // allocate int
// use *p
delete p; // free memory
```

`new` returns a pointer to the allocated memory. after deleting a pointer, all pointers to that memory should be set to `nullptr` to avoid dangling pointers.

For arrays, use `new type[size]` and `delete[] ptr;`

Structs

Structs are user-defined data types that group multiple variables together.

```
struct Point {
    int x;
    int y;
};
Point p; // declare variable of type Point
p.x = 10; p.y = 20; // access members
Point q = {30, 40}; // initialize
```

Per default, all members are public.

Classes

Classes are similar to structs but have private members by default.

Member Functions

Member functions are functions that belong to a class. Every member function has access to the `*this` pointer, which points to the object the function was called on. It does not have to be specified explicitly.

```
class MyClass {
public:
    void myFunction(int param) {
        // use param and this
    }
};
```

The Constructor is a special member function that initializes an object of the class. It has no return type and the same name as the class.

```
class MyClass {
public:
    MyClass(int value) : member(value) {} // constructor
private:
    int member;
};
```

Rule of Three

If any of the following are defined, all three should be defined:

The destructor is called when an object goes out of scope or is deleted.

```
~MyClass() {
    // cleanup code
}
```

The copy constructor is called when an object is initialized from another object of the same class.

```
MyClass(const MyClass& other) {
    // copy code
}
```

The assignment operator is called when an object is assigned from another object of the same class.

```
MyClass& operator=(const MyClass& other) {
    // assignment code
    return *this;
}
```

As a rule of thumb, `operator=` usually does the tasks of the copy constructor, but first cleans up the current object.

Input and Output overloading

Input and output operators can be overloaded for custom classes.

```
std::ostream& operator<<(std::ostream& out, const MyClass& obj) {
    // output code
    return out;
}
std::istream& operator>>(std::istream& in, MyClass& obj) {
    // input code
    return in;
}
```

Positive Modulo

```
int pos_mod(int a, int b){
  return (a%b + b) % b;
}
```

GCD and LCM

```
int gcd(int a, int b){
    while (b != 0){
        int temp = b;
        b = a % b;
        a = temp;
    }
    return a;
}
int lcm(int a, int b){
    int max = (a > b) ? a : b;
    while (true){
        if (max % a == 0 && max % b == 0){
            return max;
        }
        max++;
    }
}
```

Equivalent Resistance

```
// R1, R2 in series, R3, R4 in series, then both in parallel
const int R12 = R1 + R2;
const int R34 = R3 + R4;
const int R = (R12 * R34 + (R12 + R34) / 2) / (R12 + R34);
// Pure int calculation with rounding
```

Max in Vector

```
int max_in_vector(const std::vector<int>& v){
    if (vec.size() == 0) return 0;
    int max = v[0];
    for (size_t i = 1 ; i < v.size(); ++i){
        if (v[i] > max){
            max = v[i];
        }
    }
    return max;
}
```

Vecotr Sorting

```
void sort_vector(std::vector<int>& v){
  int n = v.size();
  for (int i = 0 ; i < n-1 ; i++){
    for (int j = 0 ; j < n-i-1 ; j++){
      if (v[j] > v[j+1]){
        int temp = v[j];
        v[j] = v[j+1];
        v[j+1] = temp;
      }
    }
  }
}
```

Wrap At

```
// PRE: data is non-empty
int& wrap_at(std::vector<int>& data, unsigned int index) {
  return data[index % std::ssize(data)];
}

int main() {
  std::vector<int> data = {0, 1, 2, 3};

  std::cout << wrap_at(data, 2) << '\n';
  std::cout << wrap_at(data, 6) << '\n';

  wrap_at(data, 6) = -77;
  std::cout << wrap_at(data, 6) << '\n';

  return 0;
}
```

Largest Independent Set

```
int maxIndependentSet(const std::vector<std::vector<int>>& graph){
  int n = graph.size();
  std::vector<int> dp(n+1,0);

  for (int i = 2 ; i <= n ; i++){
    dp[i] = std::max(dp[i-1], dp[i-2] + graph[i-1][i-2]);
  }

  return dp[n];
}
```

Bitstrings of Length n

```
void all_bitstrings1(const int n, std::string prefix) {
  if (std::ssize(prefix) == n) {
    std::cout << prefix << "\n";
  } else {
    all_bitstrings1(n, prefix + "0");
    all_bitstrings1(n, prefix + "1");
  }
}

void all_bitstrings2(std::string& bitstring, int i) {
  if (i == std::ssize(bitstring)) {
    std::cout << bitstring << "\n";
  } else {
    bitstring[i] = '0';
    all_bitstrings2(bitstring, i+1);
    bitstring[i] = '1';
    all_bitstrings2(bitstring, i+1);
  }
}
```

Bitstrings up to Length n

```
std::vector<std::string> all_bitstrings_up_to_n(int n) {
  // Base Case
  if (n==0){
    return {""};
  }

  std::vector<std::string> output = all_bitstrings_up_to_n(n-1);
  int length = output.size();

  // For each substring of length n-1 prepend the character 1 and 0
  // Prepend 0
  for (int i = 0 ; i < length ; i++){
    if (output[i].size() == unsigned(n-1))
      output.push_back("0"+output[i]);
  }

  // Preprend 1
  for (int i = 0 ; i < length ; i++){
    if (output[i].size() == unsigned(n-1))
      output.push_back("1"+output[i]);
  }
  return output;
}
```

Rounding

```
// PRE: x is roundable to a number in the value range of type int
// POST: return value is the integer nearest to x, or the one further
//       away from 0 if x lies right in between two integers.
int round_number(double x) {
  int out = x;
  double rest = x-out;
  // We differ cases where x is positive negative as there the rounding works
  // different.
  if (x > 0){
    if (rest >= 0.5) return ++out;
    return out;
  }
  else{
    if (rest <= -0.5) return --out;
    return out;
  }
}
```

Number Reverter

```
// PRE: n >= 0
// POST: print the digits of the input in reverse order to std::cout
void reverse(int n)
{
  // base condition to end recursive calls
  if (n < 10) {
    std::cout << n;
  } else {
    std::cout << n % 10;
    reverse(n / 10);
  }
}
```

Number Base Converter

```
// PRE: A number n >= 0 in base 10, and a base 1 < b < 10
// POST: Return the number in base b, i.e. return the number whose digits
//       are the digits of the representation of n in base b
// Examples:
//    * 10 2 ~~> 1010, * 14 9 ~~> 15
int convert_base(int n, int b) {
  int result = 0;
  int basetenposition = 1;

  // Reduce Number with integer division until it is 0
  while (n != 0){
    result += basetenposition*(n%b); // n%b gives last digit of n base b
    n /= b;
    basetenposition *= 10;
  }
  return result;
}
```

Binary Expansion

```
// PRE:  none
// POST: Returns 1 if x >= 1, otherwise 0.
int calculate_b(double x) {
  int b;
  if (x >= 1) {
    b = 1;
  } else {
    b = 0;
  }
  return b;
}

int main() {
  std::cerr << "Please enter decimal number x in the range 0<=x<2:\n";

  double x;
  std::cin >> x;

  if (x < 0.0 || x >= 2.0) {
    std::cout << "Input out of bounds\n";
    return 1;
  }

  int b = calculate_b(x);
  std::cout << b << ".";

  for (int i = 1; i < 16; ++i) {
    x -= b;
    x *= 2;
    b = calculate_b(x);
    std::cout << b;
  }

  std::cout << "\n";
  return 0;
}
```

DFS

```
int valid_paths(const std::vector<std::vector<bool>>& mat, int n,int m){
  // Base Case
  if (n==1 && m==1) return 1;

  int total = 0;
  // If Cell to Left or Above is Valid, then count paths to that node
  if (n > 1 && mat[n-2][m-1] == false) total += valid_paths(mat, n-1,m);
  if (m > 1 && mat[n-1][m-2] == false) total += valid_paths(mat, n,m-1);
  return total;
}
```

All Permutations

```
#include "solution.h"

StringSet all_permutations_1(const std::string& s) {
  int n = std::ssize(s);

  // base case: empty string
  if (n == 0) {
    return StringSet("");
  }

  // init empty result
  StringSet result;

  // loop over all element in string
  for (int i = 0; i < n; ++i) {
    // use current character as prefix
    std::string prefix = s.substr(i, 1);

    // get all permutations of remaining string if current character is
    ↪ removed
    std::string remaining_string = s.substr(0, i) + s.substr(i + 1);
    StringSet all_remaining_permutations =
    ↪ all_permutations_1(remaining_string);

    // add all strings to result which start with the current character
    ↪ and end
    // with any permutation of the remaining characters
    for (int j = 0; j < all_remaining_permutations.size(); ++j) {
      result.insert(prefix + all_remaining_permutations[j]);
    }
  }
  return result;
}

StringSet all_permutations_2(const std::string& s) {
  int n = std::ssize(s);

  // base case: empty string
  if (n == 0) {
    return StringSet("");
  }

  // init empty result
  StringSet result;

  // split string into first character and remaining characters
  std::string first_character = s.substr(0, 1);
  std::string remaining_string = s.substr(1);
```

```
  // generate all permutations of remaining string
  StringSet all_remaining_permutations =
↪   all_permutations_2(remaining_string);

  // insert first character at every location in every permutation
  for (int j = 0; j < all_remaining_permutations.size(); ++j) {
    for (int i = 0; i < n; ++i) {
      result.insert(all_remaining_permutations[j].substr(0, i) +
                    first_character +
                    all_remaining_permutations[j].substr(i));
    }
  }
  return result;
}
```

Set Product

```
#include "set.h"
#include "solution.h"

#include <string>
#include <vector>

StringSet set_product(const std::vector<CharSet>& sets) {
    // Base Case
    if (sets.size() == 0){
        return StringSet("");
    }

    // Split Vector into two
    CharSet first_set = sets[0];
    std::vector<CharSet> rest(sets.begin()+1, sets.end());
    StringSet old_sets = set_product(rest);

    StringSet out;

    // Build Cartesian Product of first vector with the rest
    for (int i = 0; i < first_set.size(); ++i) {
        for (int j = 0 ; j < old_sets.size(); ++j){
            out.insert(first_set[i]+old_sets[j]);
        }
    }

    return out;
}
//-----------------------------------------------------------------------
// Musti Sol
StringSet set_product(const std::vector<CharSet>& sets) {
  // base case: product of 0 sets
  if (sets.size() == 0) {
    return StringSet("");
  }
  // we have at least 1 set => split into first and remaining sets
  CharSet first_set = sets[0];
  std::vector<CharSet> remaining_sets;
  for (int i = 1; i < std::ssize(sets); ++i) {
    remaining_sets.push_back(sets[i]);
  }

  // get set product for remaining sets
  StringSet remaining_sets_product = set_product(remaining_sets);

  // compute product of first set with product of remaining sets
  StringSet product;
  for (int i = 0; i < first_set.size(); ++i) {
    for (int j = 0; j < remaining_sets_product.size(); ++j) {
      product.insert(first_set[i] + remaining_sets_product[j]);
    }
  }
  return product;
}
```

Rational Class

```
// pre: a, b >= 0
// post: return greatest common divisor of a and b
int gcd(int a, int b){
  if (b != 0){
    return gcd(b, a % b);
  }
  return a;
}

class rational{
public:
  rational (int nom, int den): n(nom), d(den) {
    assert(d != 0);
  }
  // conversion from int
  rational (int n): rational(n,1) {}
  rational (): rational(1,1) {}
  rational (const rational& r): rational(r.n,r.d){ // copy constructor
    // called upon initialization rational l = r
  }
  rational& operator=(const rational& r){
    n = r.n; d = r.d;
    return *this;
  }

  std::ostream& output(std::ostream& out) const {
    return out << n << ":" << d;
  }

  std::istream& input(std::istream& in){
    char ignore;
    in >> n >> ignore >> d;
    assert(ignore == ':'); assert(d != 0);
    return in;
  }

  rational& operator+=(const rational& r){
    n = n*r.d + d * r.n; d *= r.d;
    return reduce();
  }

  bool operator==(const rational&r) const{
    return d * r.n == r.d * n;
  }

  // prefix ++operator
  rational& operator++(){
    n += d; return reduce();
  }

  // postfix operator++
  rational operator++(int){
    rational temp = *this;
    ++(*this);
    return temp;
  }

  // conversion to double
  operator double(){ return 1.0*n/d; }

private:

  // post: normalize (r.d > 0) and reduce r
  rational& reduce(){
    if (d < 0) {n = -n;d = -d;}
    bool negative = n<0;
    if (negative) n = -n;
    int k = gcd(n,d);
    n /= k; d /= k;
    if (negative) n = -n;
    return *this;
  }

  int n; int d; // INV d != 0;
};

rational operator+(rational l, const rational& r){ return l += r; }

std::istream& operator>>(std::istream& in, rational& l){ return
↪   l.input(in); }

std::ostream& operator<<(std::ostream& out, const rational& r){ return
↪   r.output(out); }

int main(){
  rational r = rational(); rational s = rational();
  std::cin >> r >> s;
// more use cases
  rational t = r;
  // t.n = 10; // error
  t += rational(1,2);
  std::cout << t << std::endl;
  std::cout << ++t << std::endl;

  t = 2; // same as t = rational(2);
  t += 3; // even this works via conversion

  // various ways to initialize
  rational sa = rational(1,2);
  //rational sb = {1,2};
  //rational sc = {1,2};
  double x = sa;
  std::cout << sa << " = " << x << std::endl;
}
```

Complex Numbers

```
#include "complex.h"

bool read_input(std::istream& in, Complex& a){
  char c;
  // Reads one character from input stream and if it does not succeed
↪   (e.g.,
  // there are no characters left in stream) then return false.
  // If the character does not match the expected format, then return
↪   false.
  if(   !(in >> c) || c != '['
     || !(in >> a.real)
     || !(in >> c) || c != ','
     || !(in >> a.imag)
     || !(in >> c) || c != ']')
    return false;
  else
    return true;
}

std::istream& operator>>(std::istream& in, Complex& a) {
    if(!read_input(in, a)) {
        in.setstate(std::ios::failbit);
    }
    return in;
}

std::ostream& operator<<(std::ostream& out, const Complex& a) {
  return out << "[" << a.real << "," << a.imag << "]";
}

Complex operator+(const Complex& a, const Complex& b) {
  Complex res;
  res.real = a.real + b.real;
  res.imag = a.imag + b.imag;
  return res;
}

Complex operator-(const Complex& a, const Complex& b) {
  Complex res;
  res.real = a.real - b.real;
  res.imag = a.imag - b.imag;
  return res;
}

Complex operator*(const Complex& a, const Complex& b) {
  Complex result;
  result.real = a.real * b.real - a.imag * b.imag;
  result.imag = a.real * b.imag + a.imag * b.real;
  return result;
}

Complex operator/(const Complex& a, const Complex& b) {
  Complex result;
  double divisor = b.real * b.real + b.imag * b.imag;
  result.real = (a.real * b.real + a.imag * b.imag) / divisor;
  result.imag = (a.imag * b.real - a.real * b.imag) / divisor;
  return result;
}

Complex operator-(const Complex& a) {
  Complex res;
  res.real = -a.real;
  res.imag = -a.imag;
  return res;
}

bool operator==(const Complex& a, const Complex& b) {
  double diff_real = a.real - b.real;
  if (diff_real < 0) diff_real = -diff_real;
  double diff_imag = a.imag - b.imag;
  if (diff_imag < 0) diff_imag = -diff_imag;
  return diff_real < abs_err && diff_imag < abs_err;
}

bool operator!=(const Complex& a, const Complex& b) {
  return !(a == b);
}
```

Our List

```
void our_list::print() const {
  for (lnode* n = head; n != nullptr; n = n->next){
    std::cout << n->value << " ";
  }
  std::cout << std::endl;}
int& our_list::operator[](int i) {
  lnode* n = head;
  for (int j = 0; j != i; ++j){
    n = n->next;
  }
  return n->value;
}

int our_list::size() const {
  int sz = 0;
  for (lnode* n = head; n != nullptr; n = n->next){
    ++sz;
  }
  return sz;
}

void our_list::push_front(int e){
  head = new lnode(e,head);
}

our_list::our_list(int size): head(nullptr) {
  for (int i = 0; i != size; ++i){push_front(0);}
}

void our_list::push_back(int e){
  lnode* n = head;
  if (n == nullptr){head = new lnode(e);}
  else {
    while (n->next != nullptr){n = n->next;}
    n->next = new lnode(e);
  }
}

our_list::iterator::iterator(lnode* n): node(n) {}
our_list::iterator& our_list::iterator::operator++(){
    node = node->next; return *this;
}
int& our_list::iterator::operator*(){
  return node->value;
}
bool our_list::iterator::operator!=(const iterator& other) const {
  return other.node != node;
}
```

Dynamic Queue

```
#include "queue.h"

#include <cassert>
#include <iostream>

Queue::Queue(){
  first = nullptr;
  last = nullptr;
}

void Queue::enqueue(int value) {
  Node* nodeptr = new Node(value, nullptr);

  // Check if queue empty
  if (last == nullptr){
    first = nodeptr; last = nodeptr;
    return;
  }

  //Adjust pointers of last (node)
  last -> next = nodeptr;
  last = nodeptr;
}

int Queue::dequeue() {
  assert(first != nullptr);

  int value = first -> value;

  // Check if only one element
  if (first->next == nullptr){
    delete first; //Free memory
    first = nullptr;
    last = nullptr;
  }
  else {
    // Normal swap out algorithm of a pointer
    Node* temp = first->next;
    delete first;
    first = temp;
  }

    // TODO
    return value;
}

bool Queue::is_empty() const {
    return first==nullptr;
}

// PRE: -
// Post: prints the sequence of integers in nodes rachable from n,
//        in reverse order with a space before each integer.
void print_reverse_node(Node* n){
    if (n->next != nullptr) {
        print_reverse_node(n->next);
        std::cout << " " << n->value;
    }
}

void Queue::print_reverse() const {
  std::cout << "[";

  if (first != nullptr){
    std::cout << last->value;
    print_reverse_node(first);
  }
  std::cout << "]";
}


// PRE: -
```

```
// POST: print the sequence of integers in nodes reachable from n,
//       in order, with one space before each integer.
void print_node(Node* n) {
    if (n != nullptr) {
        std::cout << " " << n->value;
        print_node(n->next);
    }
}

void Queue::print() const {
    std::cout << "[";
    if (first != nullptr) {
        std::cout << first->value;
        print_node(first->next);
    }
    std::cout << "]";
}
```

Stack

```
class stack {
  struct lnode {
    int value;
    lnode* next;

    lnode(int v, lnode* n) : value (v), next (n) {}
  };
  lnode* topn;
public:
  stack() : topn(nullptr) {} // default constructor
  // POST: *this is initialized with a copy of s
  stack(const stack& s) : topn(nullptr) {
    if (s.topn == nullptr) return;

    topn = new lnode(s.topn->value, nullptr);

    lnode* prev = topn;
    for (lnode* n = s.topn->next; n != nullptr; n = n->next) {
      lnode* copy = new lnode(n->value, nullptr);
      prev->next = copy; prev = copy;
    }
  }
  // POST: *this (left operand) becomes a copy of s (right operand)
  stack& operator=(const stack& s) {
    if (this != &s) { // no self-assignment
      stack copy = s; // Copy Construction
      std::swap(topn, copy.topn);
      // now *this has the copied data and copy has the garbage, and
    } // copy is cleaned up automatically
    return *this;
  }

  // post: deconstruct (clean-up) stack
  ~stack() {
    while (topn != nullptr) {
      lnode* t = topn; topn = t->next; delete t;
    }
  }
  // post: Push an element onto the stack
  void push(int value) {
      topn = new lnode(value, topn);
  }
  // pre: non-empty stack
  // post: Delete top most element from the stack
  void pop() {
    assert(!empty());
    lnode* p = topn; topn = topn->next; delete p;
  }
  // pre: non-empty stack
  // post: return value of top most element
  int top() const {
    assert(!empty()); return topn->value;
  }
  // post: return if stack is empty
  bool empty() const {return topn == nullptr;}
  // post: print out the stack
  void print(std::ostream& out) const {
    for(const lnode* p = topn; p != nullptr; p = p->next)
      out << p->value << " ";
  }
};

std::ostream& operator<<(std::ostream& out, const stack& s) {
  s.print(out); return out;
}
int main() {
  stack s1; s1.push(1);
  std::cout << "[A] s1: " << s1 << '\n';
  stack s2 = s1; // copy constructor call
  //// Alternative to copy constructor call
  // stack s2;
  // s2 = s1; // assignment
}
```

Array Based Vector

```
our_vector::our_vector(const our_vector& vec) /* TODO */ {
  count = vec.size();
  elements = new tracked[count]();
  for (int i = 0 ; i < count ; i++){ // Copy vector data
    elements[i] = vec.elements[i];
  }
}

our_vector& our_vector::operator=(const our_vector& t) {
  if (this != &t){ // Create copy and swap elements step by step
    our_vector copy = t;
    std::swap(count, copy.count);
    std::swap(elements, copy.elements);
  } // Automatically deconstruct Copy
  return(*this);
}

our_vector::~our_vector() {
  delete[] elements;
}
```

Smart Pointers

```
Smart::Smart(){
  count = nullptr; ptr = nullptr;
}

Smart::Smart(tracked* t){
  if (t == nullptr){
    count = nullptr; ptr = nullptr; return;
  }

  // Init Smartptr
  count = new int; *count = 1; ptr = t;
}

Smart::Smart(const Smart& src){ // Copy Constructor
  this->ptr = src.ptr; this->count = src.count;

  if (count != nullptr) (*count)++; // Check that nullptr doesnt get
  ↪ increased
}

Smart::~Smart() {
  if (count == nullptr) return;

  *count = *count - 1;
  if (*count == 0){ //Special Treatment if no pointers exist anymore
    delete ptr; delete count;
    count = nullptr; ptr = nullptr;
  }
}

Smart& Smart::operator=(const Smart& src){ // Assignment Operator
  // Handle old ptr
  if (count != nullptr){
    (*count)--;
    if (*count == 0){delete ptr; delete count;}
  }
  // Update new ptr
  count = src.count; ptr = src.ptr;

  if (count != nullptr) (*count)++;
  return(*this);
}

tracked& Smart::operator*() { // Dereference Operator
  return *ptr;
}
```

Circular Linked List

```
// PRE: value > 0
// POST: insert a new Node containing the value at the beginning of the
↪ list
void CircularLinkedList::insertAtBegin(int value) {
  assert(value > 0);
  Node* node = new Node(value);
  node->next = sentinel->next;
  sentinel->next = node;
}

// PRE: value > 0
// POST: insert a new Node containing the value at the end of the list
void CircularLinkedList::insertAtEnd(int value) {
  assert(value > 0);
  sentinel->value = value;
  Node* newSentinel = new Node(0);
  newSentinel->next = sentinel->next;
  sentinel->next = newSentinel;
  sentinel = newSentinel;
}

// PRE: value > 0
// POST: removes *all* the nodes with the provided value from the list
//       and deallocates the memory of the deleted nodes.
void CircularLinkedList::removeValues(int value) {
  assert(value > 0);
  Node* current = sentinel;
  while (current->next != sentinel) {
    if (current->next->value == value) {
      // Remove current->next
      Node* tmp = current->next;
      current->next = current->next->next;
      delete tmp;
    }
    else {
      // Simply advance
      current = current->next;
    }
  }
}

// POST: Deconstructs the whole list, which includes deallocating
//       all its nodes and the sentinel
CircularLinkedList::~CircularLinkedList() {
  while (sentinel->next != sentinel) {
    Node* tmp = sentinel->next;
    sentinel->next = tmp->next;
    delete tmp;
  }
  delete sentinel;
}
```

Sorted Linked List

```
// post: add a new node with value to the sorted linked list
//       several nodes with the same value are possible
void sorted_list::add(int value){
  // this creates a new node in dynamic memory, wrapped in a shared
  ↪ pointer
  // analogously to new node(value) for normal pointers
  node_ptr newNode = std::make_shared<node>(value);

  node_ptr prev = nullptr;
  node_ptr n = first;
  while (n != nullptr && n->value < value){
    prev = n;
    n = n->next;
  }

  if (prev == nullptr){
    // Standard switch in
    node_ptr temp = first;
    first = newNode;
    newNode->next = temp;
  } else {
    prev->next = newNode;
    newNode->next = n;
  }
}

// post: remove the first node which holds 'value', if any.
//       if there is no such node, return false
//       otherwise return true
bool sorted_list::remove(int value){
  node_ptr prev = nullptr;
  node_ptr n = first;

  // Iterate until we are at candidate
  while (n != nullptr && n->value < value){
    prev = n;
    n = n->next;
  }

  // Check if candidate is correct
  if (n->value != value) return false;

  if (prev != nullptr){
    prev->next = n->next;
    return true;
  }

  // At the beginning of list
  if (n != nullptr){
    first = n->next;
    return true;
  }

  return false;
}
```

Minimax

```
// POST: Return the best possible value for the maximizing player from
↪ this
//       position in the tree. This function does NOT modify the tree.
int minimax(Node* position, bool colour) {
  if (!(position->left) && !(position->right)) {
    // It's a leaf
    return position->value;
  }

  // Evaluate the next level
  int leftValue = 0;
  int rightValue = 0;
  if (position->left) {
    leftValue = minimax(position->left, !colour);
  }
  if (position->right) {
    rightValue = minimax(position->right, !colour);
  }

  if (!(position->right)) { // Only one child (left)
    return leftValue;
  }
  if (!(position->left)) { // Only one child (right)
    return rightValue;
  }

  if (colour) {
    return std::max(leftValue, rightValue); // Purple player
  } else {
    return std::min(leftValue, rightValue); // Black player
  }
}
```

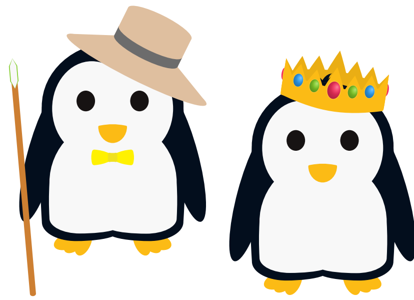
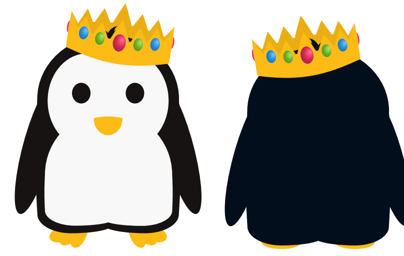
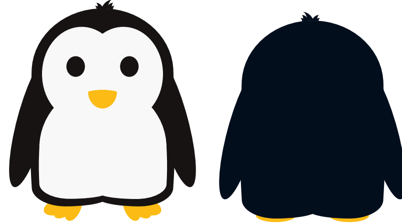
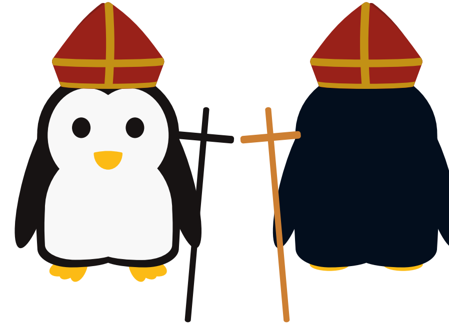
Alpha Beta Pruning

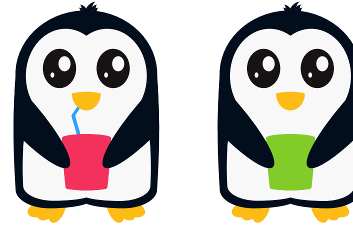
```
// PRE: The (sub-)tree under consideration is complete, i.e. all of its
↪ nodes
//       have either 0 or 2 children
// POST: Performs Alpha Beta pruning to remove the branches that are not
//       worth considering. The function only performs pruning and does
↪ NOT
//       modify the values of the nodes.
int alpha_beta_pruning(Node* position, int alpha, int beta, bool colour) {
  if (!(position->left) || !(position->right)) {
    assert(position->left == position->right); // We assume that a node
    ↪ has either no or two children

    // It's a leaf
    return position->value;
  }

  // Evaluate left child
  int leftValue = alpha_beta_pruning(position->left, alpha, beta,
  ↪ !colour);

  if (colour) {
    alpha = std::max(leftValue, alpha); // Purple player
  } else {
    beta = std::min(leftValue, beta); // Black player
  }

  if (beta <= alpha) {
    // Delete the subtree without memory leaks
    delete position->right;
    position->right = nullptr;
    return leftValue;
  }
  else {
    // Evaluate right child
    int rightValue = alpha_beta_pruning(position->right, alpha, beta,
    ↪ !colour);

    // Return the best evaluation
    if (colour) {
      return std::max(leftValue, rightValue); // Purple player
    } else {
      return std::min(leftValue, rightValue); // Black player
    }
  }
}
```

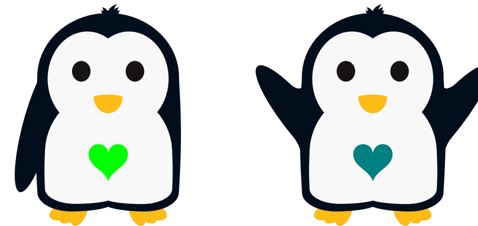
Trapezoid Printer

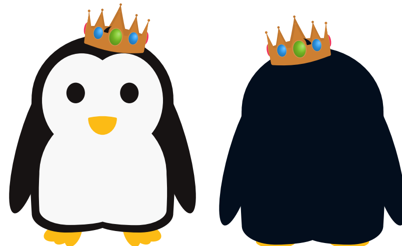
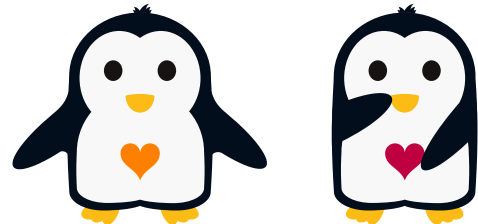
```
void print_row(int n_bricks, int offset_left, int height) {
  // print spaces for offset
  for (int i = 0; i < offset_left; ++i) {
    std::cout << " ";
  }

  // print bricks
  for (int i = 0; i < n_bricks; ++i) {
    if (i != 0) {
      std::cout << " ";
```

```
    }
    std::cout << "O";
  }

  if (height != 0) {
    std::cout << std::endl;
  }
}

void print_trapezoid(int base_width, int height,
                     int offset_left, bool upsidedown) {
  assert(base_width >= height);

  if (upsidedown){
    if (height == 0) return;
    print_row(base_width, offset_left, height);
    print_trapezoid(base_width-1, height-1, offset_left+1, upsidedown);

  }
  else{
    if (height == 0){
      return;
    }
    print_trapezoid(base_width-1, height-1, offset_left+1, upsidedown);
    print_row(base_width, offset_left, height);
  }
}
```





You can do it!

Good Luck with your Exam!

Number Table

Dec	Bin	Hex	Dec	Bin	Hex	Dec	Bin	Hex	Dec	Bin	Hex	Dec	Bin	Hex
0	0000	0	20	00010100	14	40	00101000	28	60	00111100	3c	80	01010000	50
1	0001	1	21	00010101	15	41	00101001	29	61	00111101	3d	81	01010001	51
2	0010	2	22	00010110	16	42	00101010	2a	62	00111110	3e	82	01010010	52
3	0011	3	23	00010111	17	43	00101011	2b	63	00111111	3f	83	01010011	53
4	0100	4	24	00011000	18	44	00101100	2c	64	01000000	40	84	01010100	54
5	0101	5	25	00011001	19	45	00101101	2d	65	01000001	41	85	01010101	55
6	0110	6	26	00011010	1a	46	00101110	2e	66	01000010	42	86	01010110	56
7	0111	7	27	00011011	1b	47	00101111	2f	67	01000011	43	87	01010111	57
8	1000	8	28	00011100	1c	48	00110000	30	68	01000100	44	88	01011000	58
9	1001	9	29	00011101	1d	49	00110001	31	69	01000101	45	89	01011001	59
10	1010	a	30	00011110	1e	50	00110010	32	70	01000110	46	90	01011010	5a
11	1011	b	31	00011111	1f	51	00110011	33	71	01000111	47	91	01011011	5b
12	1100	c	32	00100000	20	52	00110100	34	72	01001000	48	92	01011100	5c
13	1101	d	33	00100001	21	53	00110101	35	73	01001001	49	93	01011101	5d
14	1110	e	34	00100010	22	54	00110110	36	74	01001010	4a	94	01011110	5e
15	1111	f	35	00100011	23	55	00110111	37	75	01001011	4b	95	01011111	5f
16	00010000	10	36	00100100	24	56	00111000	38	76	01001100	4c	96	01100000	60
17	00010001	11	37	00100101	25	57	00111001	39	77	01001101	4d	97	01100001	61
18	00010010	12	38	00100110	26	58	00111010	3a	78	01001110	4e	98	01100010	62
19	00010011	13	39	00100111	27	59	00111011	3b	79	01001111	4f	99	01100011	63

ASCII Table

Dec	Ch	Dec	C	Dec	Ch	Dec	Ch	Dec	Ch	Dec	Ch	Dec	Ch	Dec	Ch
32	' '	44	,	56	8	68	D	80	P	92	\	104	h	116	t
33	!	45	-	57	9	69	E	81	Q	93	]	105	i	117	u
34	"	46	.	58	:	70	F	82	R	94	^	106	j	118	v
35	#	47	/	59	;	71	G	83	S	95	_	107	k	119	w
36	$	48	0	60	<	72	H	84	T	96	`	108	l	120	x
37	%	49	1	61	=	73	I	85	U	97	a	109	m	121	y
38	&	50	2	62	>	74	J	86	V	98	b	110	n	122	z
39	'	51	3	63	?	75	K	87	W	99	c	111	o	123	{
40	(	52	4	64	@	76	L	88	X	100	d	112	p	124	|
41	)	53	5	65	A	77	M	89	Y	101	e	113	q	125	}
42	*	54	6	66	B	78	N	90	Z	102	f	114	r	126	
43	+	55	7	67	C	79	O	91	[	103	g	115	s	127	DEL

Precedence

Prec	Operator	In	Out
2	`a++`, `a--`	L	R
2	`[]`, `.at()`	L	L
2	`.`, `->`	L	L
3	`++a`, `--a`	L	L
3	`*a`	L	L
3	`&a`	L	R
5	`*`, `/`, `%`	R	R
6	`+`, `-`	R	R
9	`<`, `<=`, `>`, `>=`	R&R	R
10	`==`, `!=`	R&R	R
10	`>>`	L&L	L
10	`<<`	L&R	L
14	`&&`	R&R	R
15	`||`	R&R	R
16	`=`, `+=`, `-=`	L&R	L
16	`*=`, `/=`, `%=`	L&R	L

Error Types

U = Undefined behaviour
C = Compiler error
R = Runtime error

Variables	
Reading undeclared variable	C
Reading uninitialized variable	U
Redeclaration, Conflicting Declaration	C
Unassignable left-hand side	C
Incompatible types	C
Assignment to const variable	C
Expressions	
Result depends on evaluation order	U
References	
Using reference to non-existing object	U
declaring uninitialized reference	C
initializing T& reference with variable of type other than T	C
initializing non-const reference with a const-reference	C
Data Types and Data Structures	
int Overflow / Underflow	U
Out of bound access vectors with [ ]	U
Out of bound access vectors with .at	R
Functions	
End of **non**-void function without return	U
Stack Overflow (Recursion) (**usually**, gives a segmentation fault)	U
Giving const object as non-const reference to a function	C
Function Overloading	
exactly same signature except for the **names** of the parameters	C
same signature except for return type	C
Classes and Structs	
Accessing private member (Reading AND writing)	C
No matching constructor available	C
const object calling non-const member function	C
Using deleted (copy) constructor, assignment operator	C
struct wagon{int id; wagon next;};	C
The same with wagon&: references cannot be uninitialised	R
Containers	
std :: vector<T>::iterator it=x.begin() with x a const-container	C
Pointers	
T* not pointing to object of type T	C
dereferencing nullptr	U
accessing uninitialized pointer	U
not using new to create object but pointer to object outside of the scope	U
Pointer Arithmetic	
Adding number beyond array size	U
pointer difference of pointers pointing to different arrays	U
p1 − p2 with p1 having a smaller address than p2	U
Dynamic Memory	
Dereferencing a dangling pointer	U
delete an object more than once	U

Templates

Loops:

```
for (int i = 0; i < n; ++i) { }
for (size_t i = 0; i < v.size(); ++i) { }

for (auto& x : v) { }        // modify
for (const auto& x : v) { }  // read-only
```

2D - Loop:

```
for (int i = 0; i < n; ++i){
  for (int j = 0; j < m; ++j){
    // use i and j
  }
}
```

Vector Input/Output:

```
for (int& x : v) std::cin >> x;
for (const int& x : v) std::cout << x << " ";
```

Loop through vector with index:

```
for (int i = 0; i < std::ssize(v); ++i)
```

Pass by Reference/Value:

```
void f(std::vector<int>& v);         // modify
void f(const std::vector<int>& v);   // read-only
void f(std::vector<int> v);          // copy
```

Recursion Template:

```
T solve(args) {
  if (base_case) return base_value;
  // recursive step
  return combine(solve(smaller));
}
```

Always implement a base case!

Standard Operators:

```
// Copy constructor
T::T(const T& other) {
  // copy data from other to this
}
// Assignment operator
T& T::operator=(const T& other) {
  if (this != &other) {
    T copy = other;           // copy constructor
    std::swap(data, copy.data); // swap data members
  } // copy is destructed here
  return *this;
}
// Call Copy constructor
T t2 = t1;
// Call Assignment operator
t2 = t1;
```

Pitfalls

- Assignment operator returns an l-value
- `++i` is an l-value, `i++` is an r-value
- Scopes of variables
- Pay attention to integer overflow
- Loop overflows für `i<=int_max`
- Do not test floats for equality
- Do not add floats of very different magnitudes
- Do not subtract nearly equal floats
- R-Values dont have a memory address
- No break in switch statements causes fallthrough
- `v.size()` returns `size_t` (unsigned)
- Integer Division truncates towards zero