L-Value: Hat Speicherplatz und kann geändert werden. z.B. Variablen **Operatoren:** L=R, L+=R, ++L, L>>L, L<<R…
R-Value: Kein Speicherplatz und ist nicht veränderbar. z.B. Literal **Operatoren:** R*R, !R, L++

Operatoren-Präzedenzen (Leserichtung - farblos: links->rechts, rot: rechts->links) nur relevante Operatoren!

1	2	3	4	5	6	9	10	14	16
::	a++ a– type() f( ) v[ ] . **oder** ->	++a !b *a &a new delete	.* ->*	a*b a/b a%b	a+b a-b	< <= > >=	== !=	&& ||	= += -= *= /= **oder** %=

Type-Präzedenzen (bei Type-conversion) von links->rechts gehen keine Daten verloren, andere Richtung schon.

bool - char - short int - **int** - **unsigned int** - long - unsigned - longlong - **float** - **double** - long double

bool -> int **false->0, true->1**
int -> bool **0->false, alle anderen->true**
bool -> int **false->0, true->1**
char -> int **gemäss ASCII Tabelle**
int -> char **gemäss ASCII Tabelle**
int -> u int **positiv bleiben, negativ werden von max value(u int)=2^32 - 1 rückwärts gezählt**

Binär (oben: Binär, mitte: int, unten: unsigned int) Tipp für int: Falls erste zahl 1 -> negativ

0000 0 0u	**0001 1 1u**	**0010 2 2u**	**0011 3 3u**	**0100 4 4u**	**0101 5 5u**	**0110 6 6u**	**0111 7 7u**
1000 -8 8u	**1001 -7 9u**	**1010 -6 10u**	**1011 -5 11u**	**1100 -4 12u**	**1101 -3 13u**	**1110 -2 14u**	**1111 -1 15u**

11	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	**-1**	**-2**	**-3**
2048	**1024**	**512**	**256**	**128**	**64**	**32**	**16**	**8**	**4**	**2**	**1**	**0.5**	**0.25**	**0.125**

max value (u int) = 2^B - 1 **min value (u int) = 0** **für int/uint in C++ gilt B=32**
max value (int) = 2^(B-1) -1 **min value (int) = -2^(B-1)** **for code use INT_MAX, UINT_MAX**

Hexadezimal

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**

Fliesskommazahlen F*(**β**, p, e_min, e_max)
β ist Basis d.h. **β**=2 -> Binär etc.
p ist Präzision, d.h. Anzahl Ziffern (inklusive führende 1 und 0 am Ende)
Zahl wird dargestellt in **β-Basis mit p Ziffern und dem Faktor β^e**, wobei e zwischen e_min und e_max is
Vorzeichen wird explizit geschrieben und es wird per default auf die nächstmögliche Darstellung gerundet.

WATCH OUT FOR SCOPES! BOOLIAN OPERATOR: Is it even relevant?

Overloading examples - CLL insert at beginning/end, remove given values, destructor

```
// 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) {
      Node* tmp = current->next;// Remove current->next
      current->next = current->next->next;
      delete tmp;}
    else current = current->next;// Simply advance
  }
}
```

```
// 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 LL - add new element, remove given value, print 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){
    first = newNode;
    newNode->next = n;
  } 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;
  while (n != nullptr and n -> value != value) {
    prev = n;
    n = n -> next;
  }
  if (n == nullptr) return false;
  if (prev == nullptr) first = n->next;
  else prev->next = n->next;
  return true;
}
```

```
// post: output the linked list of nodes
//       in the form 1->2->3->null
void print(std::ostream& out){
  node_ptr n = first;
  while(n != nullptr){
    out << n->value << "->";
    n = n->next;
  }
  out << "null";
}
};
```

Queue - enqueue, dequeue

```
//Post: adds Node with given int Value to the end of the queue
void Queue::enqueue(int nvalue) {
  Node* addendum = new Node{nvalue, nullptr};
  if (first != nullptr){
    last -> next = addendum;
    last = addendum;
  }
  else {
    first = addendum;
    last = first;
  }
}
```

```
//Post: First node of the queue gets dequeued and it´s value returned.
int Queue::dequeue() {
  int first_value = (first -> value);
  first = first -> next;
  if (first == nullptr){
    last = nullptr;
  }
  return first_value;
}
```

our_vector (rule of three) - copy constructor, assignment operator, destruct

```
//Assignment operator
our_vector& our_vector::operator=(const our_vector& t) {
  if (elements != t.elements) {
    our_vector copy = our_vector(t);
    std::swap(copy.elements, elements);
    std::swap(copy.count, count);
  }
  return(*this);
}
```

```
//destructor
our_vector::~our_vector() {
  delete[] elements;
  elements = nullptr;//optional
}
//Copy constructor
our_vector::our_vector(const our_vector& vec) : count(vec.count) {
  elements = new tracked[count]();
  for (int i = 0; i != count; ++i) {
    elements[i] = vec.elements[i];
  }
}
```

```
type a = 5; //constructor
type c; //default constructor
type b = a; //copy constructor
c = a; //assignment operator
```

Tree - is Leaf, maximum value, height of tree, number of nodes

```
// Function to check if a node is a leaf
bool is_leaf(TreeNode* node) {
  if (!node) return false; // Null nodes are not leaves
  return (!node->left && !node->right);
}

//Post: find maximal value of tree
int findMax(TreeNode* node) {
  if (!node) return INT_MIN; //return INT_MIN if tree is empty;
  int leftMax = findMax(node->left);
  int rightMax = findMax(node->right);
  return std::max(node->value, std::max(leftMax, rightMax));
}
```

```
//Post: returns number of levels of the tree
int getTreeHeight(TreeNode* node) {
    if (!node) return 0;
    int leftHeight = getTreeHeight(node->left);
    int rightHeight = getTreeHeight(node->right);
    return 1 + std::max(leftHeight, rightHeight);
}

//Post: returns number of nodes in tree
int countNodes(TreeNode* node) {
    if (!node) return 0;
    return 1 + countNodes(node->left) + countNodes(node->right);
}
```

Tree - delete (sub)tree, left rotate, search tree, equality of (sub)trees, number of non full nodes

```
//Post: delets subtree if value of parent node is equal to the given value
void delete_subtree(Node*& node, int value){
  if (node == nullptr) return;
  else if (node->value == value){
    Node* n = node;
    delete n;
    node = nullptr;
  }
  else {
    delete_subtree(node->left, value);
    delete_subtree(node->right, value);
  }
}
//Post: returns true only if key was found in any of the values in (sub)tree
bool search(TreeNode* root, int key) {
    if (root == nullptr) return false;
    if (root->value == key) return true;
    if (key < root->value)
        return search(root->left, key);
    else
        return search(root->right, key);
}
//Post: returns number of non full nodes in (sub)tree
unsigned int nonFullNodes(Node* root){
  if (root==nullptr) return 0;
  unsigned int result = nonFullNodes(root->left) + nonFullNodes(root->right);
  return result + ((root->left ==nullptr) != (root->right ==nullptr));
}
```

```
//Post: right-child becomes new root, left child of that new root is old root
//real world applications: f.e. to balance trees and cut computing time
Node* left_rotate(Node* subroot){
  Node* right_child = subroot->right;
  Node* right_left_child = right_child->left;
  right_child->left = subroot;
  subroot->right = right_left_child;
  return right_child;
}
```

```
//Post: Check if Two Trees are Equal
bool isEqual(Node* root1, Node* root2) {
    if (!root1 && !root2) return true;
    if (!root1 || !root2) return false;
    return (root1->value == root2->value) &&
           isEqual(root1->left, root2->left) &&
           isEqual(root1->right, root2->right);
}
```

BST - is BST, LCA in BST

```
//Post: returns true if Tree is BST - all values in left subtree are smaller
//than root value, all values in right subtree bigger
bool isValidBST(Node* root, Node* minNode = nullptr, Node* maxNode = nullptr){
    if (!root) return true;
    if ((minNode && root->value <= minNode->value) ||
        (maxNode && root->value >= maxNode->value))
        return false;
    return isValidBST(root->left, minNode, root) &&
           isValidBST(root->right, root, maxNode);
}
```

```
//Post: returns Lowest Common Ancestor in a BST
Node* lowestCommonAncestor(Node* root, Node* p, Node* q) {
    if (!root) return nullptr;
    if (p->value < root->value && q->value < root->value)
        return lowestCommonAncestor(root->left, p, q);
    if (p->value > root->value && q->value > root->value)
        return lowestCommonAncestor(root->right, p, q);
    return root; // This is the LCA
}
```

GCD and LCM, prime-factors as vector, is_prime

```
//Post: return gcd of two numbers
unsigned int gcd(unsigned int a, unsigned int b){
  if(a==0 or b==0) return 1;
  for (unsigned int i = a; i<=a; i--){
    if (a%i==0 and b%i==0) return i;
  }
}
//Post: returns lcm of two numbers
unsigned int lcm(unsigned int a, unsigned int b){
  if(a==0 or b==0) return 0;
  return (a/gcd(a,b) *b);
}
//Post: returns true if n is prime
bool is_prime(int n) {
  if (n <= 1) return false;
  if (n <= 3) return true;
  if (n % 2 == 0 || n % 3 == 0) return false;
  for (int i = 5; i * i <= n; i += 6) {
    if (n % i == 0 || n % (i + 2) == 0) return false;
  }
  return true; // n is prime if no divisors were found
}
```

```
//Post: return prime factors as vector
std::vector<int> primeFactorization(int n) {
  std::vector<int> factors;
  // Handle smallest prime number (2)
  while (n % 2 == 0) {
    factors.push_back(2);
    n /= 2;}
  // Check for odd factors from 3 to sqrt(n)
  for (int i = 3; i * i <= n; i += 2) {
    while (n % i == 0) {
      factors.push_back(i);
      n /= i;}}
  // If n is still greater than 2, it must be a prime number
  if (n > 2) {
    factors.push_back(n);
  }
  return factors;
```

Basic Code Examples - recursion, swap, in-/output, loops, conditionals, arrays, classes, dynamic memory

```cpp
#include <iostream>

class Rectangle {
private:
  int length, width;
public:
  Rectangle(int l, int w) : length(l), width(w) {} // Constructor
  int area() const { return length * width; }     // Calculate area
};
// Function for factorial using recursion
int factorial(int n) {
    return (n <= 1) ? 1 : n * factorial(n - 1);
}
// Function to swap two numbers using pointers
void swap(int* a, int* b) {
  int temp = *a;
  *a = *b;
  *b = temp;
}

int main() {
  // 1. Input/Output
  int a, b;
  std::cout << "Enter two numbers: ";
  std::cin >> a >> b;
  std::cout << "Sum: " << a + b << "\n";
  // 2. Loops: Print numbers 1-10
  std::cout << "Numbers from 1 to 10: ";
  for (int i = 1; i <= 10; i++) std::cout << i << " ";
  std::cout << "\n";
  // 3. Conditionals: Even or Odd
  std::cout << a << " is " << ((a % 2 == 0) ? "even" : "odd") << "\n";
  // 4. Arrays: Find the largest element
  int arr[] = {10, 20, 5, 30, 25};
  int arraySize = sizeof(arr) / sizeof(arr[0]);
  int largest = arr[0];
  for (int i = 1; i < arraySize; i++)
    if (arr[i] > largest) largest = arr[i];
  std::cout << "Largest element in the array: " << largest << "\n";
  // 5. Factorial function
  int num;
  std::cout << "Enter a number to find factorial: ";
  std::cin >> num;
  std::cout << "Factorial of " << num << " is " << factorial(num) << "\n";
  // 6. Pointers: Swap two numbers
  std::cout << "Before swap: a = " << a << ", b = " << b << "\n";
  swap(&a, &b);
  std::cout << "After swap: a = " << a << ", b = " << b << "\n";
  // 7. Classes: Rectangle area
  Rectangle rect(10, 5);
  std::cout << "Area of rectangle: " << rect.area() << "\n";
  // 8. Dynamic Memory Allocation
  int* ptr = new int(42); // Allocate memory dynamically
  std::cout << "Dynamically allocated value: " << *ptr << "\n";
  delete ptr; // Free allocated memory

  return 0;
}
```

Iterating over a vector/string/array with simple loops or iterators

```cpp
std::vector<int> vec = {1, 2, 3, 4, 5};
//iterate over a vector using iterators
for (auto it = vec.begin(); it != vec.end(); ++it) {
  std::cout << *it << " ";}
//iterates over a vector using reverse iterators
for (auto it = vec.rbegin(); it != vec.rend(); ++it) {
  std::cout << *it << " ";}
//iterates over a vector using simple for loop
for (int i = 0; i != vec.size(); ++i) {
  std::cout << vec[i] << " ";}
//iterates over a vector in reverse using simple for loop
for (int i = vec.size() - 1; i != -1; --i) {
  std::cout << vec[i] << " ";}
return 0;
```

```cpp
std::string str = "Hello, World!";
//Iterate over string using iterator
for (auto it = str.begin(); it != str.end(); it++){
  std::cout << *it;}
//Iterate over string using reversed iterator
for (auto it = str.rbegin(); it != str.rend(); it++){
  std::cout << *it;}
//Iterate over string using simple loop
for (int i = 0; i != str.size(); i++){
  std::cout << str[i];}
//Iterate over string reversed using simple loop
for (int i = str.size(); i != -1; i--){
  std::cout << str[i];}
```

```cpp
std::array<int, 5> arr = {1, 2, 3, 4, 5};
//iterate over array using iterator
for (auto it = arr.begin(); it != arr.end(); it++){
  std::cout << *it;}
//iterate over array using reversed iterator
for (auto it = arr.rbegin(); it != arr.rend(); it++){
  std::cout << *it;}
//iterate over array using simple loop
for (int i = 0; i != arr.size(); i++){
  std::cout << arr[i];}
//iterate over array using reversed loop
for (int i = arr.size() - 1; i != - 1; i--){
  std::cout << arr[i];}
```

Working with inputs - add inputs while of type double, using switch

```cpp
//Post: all consecutivly typed in doubles are stored in the numbers vector
double num;
std::vector<double> numbers; // Vector to store the doubles
// Read doubles until there's no more valid input
while (std::cin >> num) {
  numbers.push_back(num); // Add the input number to the vector
}
```

```cpp
int choice;
std::cin >> choice;
switch (choice) {
  case 1:
    std::cout << "You selected 1.\n";
    break; // Stops execution here if case 1 matches
  case 2:
    std::cout << "You selected 2.\n";
  case 3:// If choice is 2 falls through here and also prints 3
    std::cout << "You selected 3.\n";
    break;
  default:// If choice doesn't match default case
    std::cout << "Invalid selection.\n";
    break;
  }
```

simplifying data types - using

```cpp
// Creating type aliases
using IntVector = std::vector<int>;
IntVector vec; //instead of std::vector<int> vec;
```

Overloading Operators - input/output

```cpp
// Overload the input operator >>
friend std::istream& operator>>(std::istream& is, Person& person) {
  std::cout << "Enter name: ";
  is >> person.name;
  std::cout << "Enter age: ";
  is >> person.age;
  return is;
}

// Overload the output operator <<
friend std::ostream& operator<<(std::ostream& os, const Person& person) {
  os << "Name: " << person.name << ", Age: " << person.age;
  return os;
}
```

```cpp
// Overload the unary minus operator
Point operator-() const {
  return Point(-x, -y); // Negates both x and y
}
// Overload the binary + operator
Point operator+(const Point& other) const {
  return Point(x + other.x, y + other.y);
}
// Overload the * operator for scalar multiplication
Point operator*(int scalar) const {
  return Point(x * scalar, y * scalar);
}
```