

## Aufgabe 1: Typen und Werte (Basistypen) (12P)

Geben Sie für jeden der Ausdrücke auf der nächsten Seite jeweils C++ Typ und Wert an. Wenn der Wert nicht bestimmt werden kann, schreiben Sie "undefiniert".

In manchen Teilaufgaben finden Sie den Ausdruck unter einer dünnen Linie. Darüber finden Sie eine oder mehrere Anweisungen, welche direkt vor dem Ausdruck ausgeführt wurden und welche relevant sind zur Bestimmung des Typs und Wertes des gefragten Ausdrucks.

*For each of the expressions on the next page provide the C++ type and value. If a value cannot be determined, write "undefined".*

*In some tasks you find the expression below a thin line. Above the line you find one or more statements that have been executed immediately before the expression. They are relevant to determine the type and value of the expression in question.*

(a) `double c = 20;`

/2P

`(9/5) * c + 32.0`

Typ / *Type*

`double`

Wert / *Value*

`52.0`

(b) `bool a = false; int b = 6;`

/2P

`a++ + ++b`

Typ / *Type*

`int`

Wert / *Value*

`7`

(c) `int d = 0;`

/2P

`(10 % 5) + ++d > 0`

Typ / *Type*

`bool`

Wert / *Value*

`true`

(d) `int g = 5;`

/2P

`g -= g`

Typ / *Type*

`int`

Wert / *Value*

`0`

(e) `bool h = true; bool i = 4 > 4;`

/2P

`h || !i && !h || i`

Typ / *Type*

`bool`

Wert / *Value*

`true`

(f) `std::vector<int> v = { 1, 2, 3, 4 };  
int f = v[1];`

/2P

`f-- * v[4]`

Typ / *Type*

`int`

Wert / *Value*

`undefined`

## Aufgabe 2: Konstrukte (15P)

Geben Sie zu folgenden Codestücken jeweils die erzeugte Ausgabe an.

*Provide the output for each of the following pieces of code.*

```
std::vector<int> a {5, 6, 7};
std::cout << a[0] << " ";
int* x = &a[0];
std::cout << x[1] << " ";
*x *= a[1];
std::cout << a[0]+a[2];
```

/5P (a)

Ausgabe / *Output*: 5 6 37

```
std::vector<int> a {1, 5, 2, 0, 3, 4};
for (int* i = &a[0]; *i != 3; ++i) {
    std::cout << a[*i];
}
```

/5P (b)

Ausgabe / *Output*: 5421

```
char my_function(char* p, int x) {
    if (x == 0) return **p;
    else return **p;
}
```

/5P (c)

```
std::string c1 = "ab";
std::string c2 = "ab";
char p1, p2;
p1 = my_function(&c1[0], 0);
p2 = my_function(&c2[0], 1);
std::cout << p1 << p2 << c1[0] << c2[0];
```

Ausgabe / *Output*: bbab

## Aufgabe 3: Zahlendarstellungen (11P)

- (a) Die Zahl in der linken Spalte der nachfolgenden Tabelle ist jeweils als Literal der Sprache C++ zu verstehen. Berechnen Sie, was jeweils verlangt ist.  
Hinweis: Eine Hexadezimalzahl (Prefix 0x) besteht aus Zahlen mit Basis 16 und einem Alphabet {0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F}.

*The number displayed to the left in the following table shall be considered a number literal in C++ language. Compute what is requested.  
Hint: A hexadecimal number (prefix 0x) consists of numbers with base 16 and an alphabet {0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F}.*

/3P

0x2A	als Dezimalzahl / to decimal number	=	42
31	als binäre Zahl (5 Bits) / to binary number (5 Bits)	=	11111
111111	die binäre Zahl als Dezimalzahl / from binary number to decimal number	=	63

- (b) Markieren Sie die Werte, die durch den Typ unsigned int auf einem 7-Bit System dargestellt und ausgegeben werden können.

*Mark the values that can be represented and outputted by the type unsigned int on a 7-bit system.*

/4P

☒ 0   ☒ 127   ☐ -64   ☐ 128

- (c) Geben Sie die grösste Zahl und die kleinste positive Zahl, die das normalisierte Flieskommasystem  $F^*$  repräsentieren kann, an. Beide Antworten sind in dezimaler Darstellung anzugeben. Hinweis:  $p$  zählt auch die führende Ziffer.

*Provide the largest number and the smallest positive number representable by the normalized floating point system  $F^*$ . Provide both answers in decimal representation. Hint:  $p$  does also count the leading digit.*

/4P

$$F^*(\beta, p, e_{min}, e_{max}) = F^*(2, 5, -2, 3)$$

grösste Zahl / largest number

15.5

kleinste positive Zahl / smallest positive number

0.25

## Aufgabe 4: Rekursion: Collatz-Problem (10P)

In dieser Aufgabe berechnen Sie die Zahlenfolge die sich durch die wiederholte Anwendung der Collatz-Funktion  $C(x)$  ergibt.

*In this task you compute the sequence of numbers built by repeatedly applying the Collatz function  $C(x)$ .*

$$C(x) = \begin{cases} \frac{x}{2} & \text{falls } x \text{ gerade / if } x \text{ even} \\ 3x + 1 & \text{andernfalls / otherwise} \end{cases}$$

Es wird angenommen, dass egal mit welcher Zahl die Folge gestartet wird, sie irgendwann bei der Zahl 1 ankommt. Da 1 ungerade ist folgt auf sie die 4, auf diese dann die 2 und schließlich wieder die 1; die Folge wiederholt sich also unendlich.

Für den (positiven) Zahlenbereich der von `int` abgedeckt wird ist diese Vermutung bewiesen worden.

*Presumably the sequence always reaches 1 after some steps, regardless of the number the sequence was started with. Since 1 is odd, the next number is 4, followed by 2 and 1 again: The sequence thus starts to repeat forever.*

*For the range of (positive) numbers representable by `int` the above assumption has been verified.*

- /2P (a) Implementieren Sie die Collatz Funktion!

*Implement the Collatz function!*

```
int collatz(int num) {
```

```
    if (num % 2 == 0) return num / 2;
    else return 3 * num + 1;
```

```
}
```

- /4P (b) Implementieren Sie die Funktion `count_collatz_rec` als **rekursive** Funktion. Sie gibt zurück wie oft die Collatz-Funktion auf eine Zahl angewandt werden muss um als Ergebnis 1 zu erhalten.

*Implement the function `count_collatz_rec` using **recursion**. The function shall return how often the Collatz function needs to be applied to the given number until the result is 1.*

```
unsigned int count_collatz_rec(int num) {
```

```
    if (num == 1) return 0;
    return 1 + count_collatz_rec(collatz(num));
```

```
}
```

- (c) Implementieren Sie die Funktion `count_collatz_it` als **iterative** Funktion (also mit einer Schleife). Sie gibt zurück wie oft die Collatz-Funktion auf eine Zahl angewandt werden muss um als Ergebnis 1 zu erhalten.

*Implement the function `count_collatz_it` using **iteration** (e.g. a loop). The function shall return how often the Collatz function needs to be applied to the given number until the result is 1.*

/4P

```
unsigned int count_collatz_it(int num) {  
    unsigned int count = 0;  
  
    while (num != 1) {  
        num = collatz(num);  
        ++count;  
    }  
  
    return count;  
}
```

## Aufgabe 5: EBNF (13P)

Die folgende EBNF definiert eine einfache Grammatik zum Beschreiben von nicht leeren Dateien. Vervollständigen Sie die Funktionen in den Teilaufgaben.

**Anmerkung:** Leerschläge sind im Rahmen dieser EBNF bedeutungslos.

*The following EBNF defines a simple grammar for the language of non-empty data files. Complete the functions in the subparts of this task.*

**Remark:** Whitespaces are irrelevant in the context of this EBNF.

```
Datafile = Record { Record }.
Record  = String { ";" String } ".".
String  = '"' Letter { Letter } '"'.
Letter  = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j".
```

Beachten Sie, dass String von einfachen Anführungszeichen umgeben ist.

*Note that String is surrounded by single quotes.*

Sie können davon ausgehen, dass die folgenden Funktionen bereits implementiert sind.

*You can assume that the following functions are already implemented.*

```
bool Letter(std::istream& is); // Consumes next Letter or returns false.

// POST: if the next available non-whitespace character equals c,
// it is consumed and returns true, otherwise returns false.
bool consume(std::istream& is, char c);
```

- /5P (a) Vervollständigen Sie die Funktion `String` gemäss der gegebenen EBNF. *Complete the function `String` to implement the corresponding EBNF.*

```
// Consumes next String = '"' Letter { Letter } '"'.
bool String(std::istream& is) {
```

```
    if (!consume(is, '"')) return false;
    if (!Letter(is)) return false;
    while (Letter(is));
    return consume(is, '"');
```

```
}
```

- (b) Vervollständigen Sie die Funktion Record gemäss der gegebenen EBNF.

*Complete the function Record to implement the corresponding EBNF.*

/5P

```
// Consumes next Record = String { ";" String } ".".
bool Record(std::istream& is) {
    if (String(is)) {
        while(consume(is, ';')) {
            if (!String(is)) return false;
        }
        return consume(is, '.');
    }
    return false;
}
```

- (c) Vervollständigen Sie die Funktion Datafile gemäss der gegebenen EBNF.

*Complete the function Datafile to implement the corresponding EBNF.*

/3P

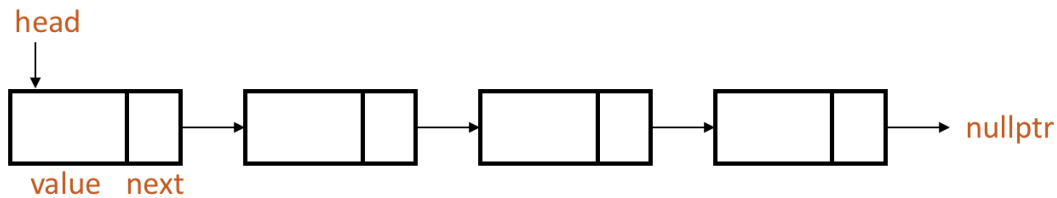
```
// Consumes next Datafile = Record { Record }.
bool Datafile(std::istream& is){
    if (Record(is)){
        while(Record(is));
        return true;
    }
    return false;
}
```



## Aufgabe 6: Sortierte Liste (13P)

Sie implementieren ein Programm, das, ausgehend von einer aufsteigend sortierten Liste, einen neuen Knoten hinzufügt, während die Liste sortiert bleibt.

*You are to implement a program which, given a linked list whose values are sorted in ascending order, adds a new node while keeping the list sorted.*



/13P (a) Vervollständigen Sie Part a) - c) in der Funktion addNode.

*Complete part a) - c) in the function addNode.*

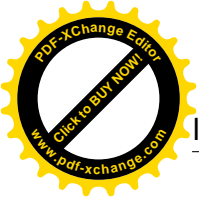
```

struct Node {
    int value;
    Node* next;
    Node(int _value) : value(_value), next(nullptr) {}
};

class SortedList {
private:
    Node* head;
public:
    SortedList() : head(nullptr) {}

    void addNode(Node* newNode) {
        // Part a): If the list is empty (represented by head being nullptr),
        // the new node should be the head.
        if (head == nullptr) {
            head = newNode;
        } else {
            // Part b): Find a point in the list such that the value of
            // newNode is greater than the one of left Node and
            // less or equal to one of right node.
            Node* left = nullptr;
            Node* right = head;

            while (right != nullptr) {
                // right should always point to the next node of left.
                if (left != nullptr)
                    assert(left->next == right);
            }
        }
    }
};
  
```



```
// if the value of newNode is less or equal to the
// one of right Node, insert newNode between left and right nodes
if ( newNode->value <= right->value ) {
    // place newNode to the right side of left Node,
    // unless newNode is the smallest (that is, first)
    if (left == nullptr)
        head = newNode;
    else
        left->next = newNode;

    // then place it to the left side of right Node.
    newNode->next = right;
    break;
}

// move the left and right node to the right
left = right;
right = right->next;
}


// Part c): If the value of newNode is greater than any
// value in the list, append newNode to the tail.
if ( right == nullptr ) {
    left->next = newNode;
    newNode->next = nullptr;
}
}
};


int main() {
    SortedList slist;
    Node zero(0), one(1), two(2), three(3);
    slist.addNode(&one);
    slist.addNode(&three);
    slist.addNode(&two);
    slist.addNode(&zero);
    // result: 1 / 1->3 / 1->2->3 / 0->1->2->3
}
```

## Aufgabe 7: Binärer Suchbaum (16P)

Ein binärer Suchbaum hat die folgenden Eigenschaften: 1) Der linke Teilbaum eines Knotens enthält nur Knoten mit Schlüsseln, die kleiner als der Schlüssel des Knotens sind. 2) Der rechte Teilbaum eines Knotens enthält nur Knoten mit Schlüsseln, die grösser als der Schlüssel des Knotens sind. 3) Der linke und der rechte Teilbaum müssen jeweils auch ein binärer Suchbaum sein. 4) Es dürfen keine doppelten Schlüssel vorhanden sein.

Implementieren Sie die Funktion `delete`.

-  /4P (a) Vervollständigen Sie die Definition von `Node` und die Hilfsfunktion `minValueNode`, die den Knoten findet, der den kleinsten Wert in der Baumstruktur enthält. Hinweis: Dieser Knoten ist der Knoten ganz links im Baum.

-  /12P (b) Vervollständigen Sie die Funktion `deleteNode` indem Sie drei Fälle in Betracht ziehen: 1) Der zu löschende Knoten ist ein Blatt: Einfach aus dem Baum entfernen. 2) Der zu löschende Knoten hat nur einen untergeordneten Knoten: Ersetzen Sie den zu löschende Knoten durch seinen untergeordneten Knoten. 3) Der zu löschende Knoten hat zwei untergeordnete Knoten: Suchen Sie den Knoten mit dem kleinsten Wert im rechten Teilbaum. Kopieren Sie den Inhalt dieses Blattknotens in den aktuellen Knoten und löschen Sie den Blattknoten. Hinweis: Sie können `minValueNode` verwenden.

*A binary search tree has the following properties: 1) The left subtree of a node contains only nodes with values lesser than the node's value. 2) The right subtree of a node contains only nodes with values greater than the node's value. 3) The left and right subtree each must also be a binary search tree. 4) There must be no duplicate keys.*

*Implement the `delete` function.*

*Complete the definition of `Node` and the helper function `minValueNode`, which finds the node containing the smallest value in the tree. Hint: this node is the left-most node of the tree.*

*Complete the function `deleteNode` by considering three cases: 1) Node to be deleted is leaf: Simply remove from the tree. 2) Node to be deleted has only one child: replace the node to be deleted with its child node. 3) Node to be deleted has two children: Find the node containing the smallest value in the right subtree. Copy contents of this leaf node to the current node and delete the leaf node. Hint: you can use `minValueNode`.*

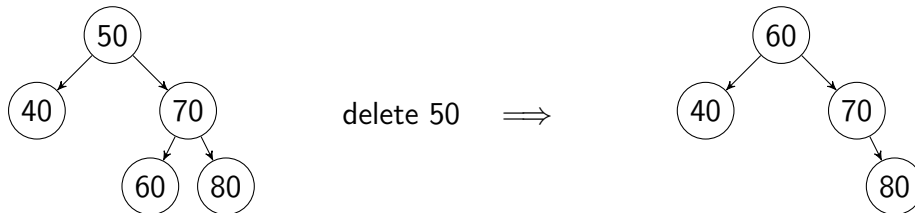
Beispiel / *Example*



Fall 1: Der zu löschende Knoten ist ein Blatt / *Case 1: Node to be deleted is leaf*



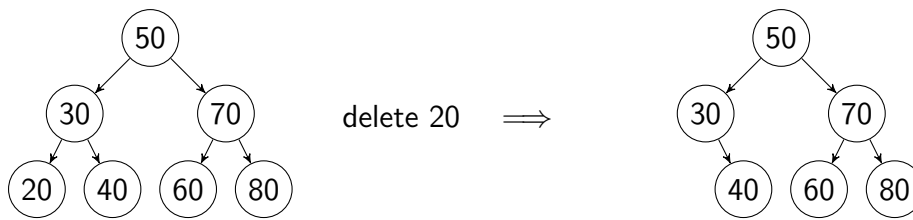
Fall 2: Der zu löschende Knoten hat nur ein Kind / *Case 2: Node to be deleted has only one child*



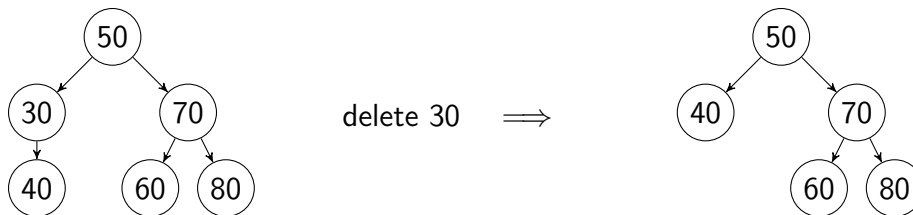
Fall 3: Der zu löschende Knoten hat zwei Kinder / *Case 3: Node to be deleted has two children*

Bitte umblättern, um die Aufgabe zu lösen! / *Please turn the page to solve this task!*

Das gleiche Beispiel noch einmal / *The same example again*



Fall 1: Der zu löschende Knoten ist ein Blatt / *Case 1: Node to be deleted is leaf*



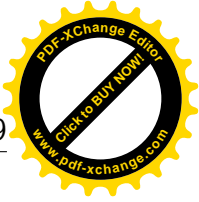
Fall 2: Der zu löschende Knoten hat nur ein Kind / *Case 2: Node to be deleted has only one child*



Fall 3: Der zu löschende Knoten hat zwei Kinder / *Case 3: Node to be deleted has two children*

```
// Define Node structure.
// A node stores its value, a pointer to its left and right children.
struct Node {
    int value;
    Node* left;
    Node* right;
};

// PRE: the root of a non-empty binary tree
// POST: returns the node with the minimum value inside the tree.
Node* minValueNode(Node* root) {
    while (root->left != nullptr)
        root = root->left;
    return root;
}
```



```
// PRE: root of a non-empty binary tree and the value to be deleted.
// POST: deletes the value and returns the new root of the tree.
Node* deleteNode(Node* root, const int value) {
    // If the value to be deleted is less than the root's value,
    // recurse in the left subtree
    if (value < root->value)
        root->left = deleteNode(root->left, value);
    // If the value to be deleted is greater than the root's value,
    // recurse in the right subtree
    else if (value > root->value)
        root->right = deleteNode(root->right, value);
    // the value of root equals the value to be deleted.
    else
    {
        // Case 1: root is a leaf.
        if (root->left == nullptr && root->right == nullptr) {
            // delete this node
            delete root;
            return nullptr;
        }
        // Case 2: root is a node with only one child.
        if (root->left == nullptr || root->right == nullptr) {
            Node* temp;
            if (root->left == nullptr)
                temp = root->right;
            else
                temp = root->left;
            delete root;
            return temp;
        }
        // Case 3: root is a node with two children.
        // Find the inorder successor of this node (the node with the
        // next greater value).
        struct Node* temp = minValueNode(root->right);
        // Copy the inorder successor's content to this node
        root->value = temp->value;
        // Delete the inorder successor
        root->right = deleteNode(root->right, temp->value);
    }
    return root;
}
```