

Zusammenfassung INF2 – Abteilung C

Autor: chrisvin

1 Dateien

Dateien read/write (<stdio.h>)

- fopen → file öffnen
- fprintf / fscanf → eingabe ausgabe aus file (formatiert)
- fputc/ fgetc → zeichen eingeben / holen
- fputs / fgets → string schreiben / lesen
- fflush → von memory in den langzeitspeicher schreiben
- fclose → aktuelle Datei schliessen
- remove → aktuelle Datei löschen

Beispiel Code:

```
FILE *fopen (const char *filename, const char *mode)
// FILE -> struct der Datei
// *filename -> absoluter oder relativer pfad zur zu öffnenden Datei
// *mode -> zugriffsart (r(read), w (write), rw (readwrite), a(append),
// rb(read binary), wb(write binary))
// bei binary Zugriff wird definiert wie das new-line gehandhabt wird
```

Fileopen(<stdio.h>)

```
#include <stdio.h>
int main (void) {
    FILE *fp; // filepointer *fp auf struct FILE
    char filename[] = "test.txt";
    fp = fopen(filename, "w"); // pointer fp soll test.txt im modus write
    // öffnen
    if (fp == NULL) {
        printf("Error opening file %s\n", filename); // errorhandling
        return 1;
    }
    fprintf(fp, "Hello World\n"); // schreibe in das file
    fclose(fp); // schliesse das file. Wichtig: Geschriebene Daten wurden evtl.
    // nur gebuffert und werden erst mit f.close() oder f.flush() mit
    // Sicherheit ins File gespeichert
}
```

Stringtokenizer (<string.h>)

token → ein einzelnes, nicht weiter zerlegbares Bauelement einer Programmiersprache

```
#include <stdio.h>
#include <string.h>

int main (void) {
    char str[] = "one,two,three"; // Initialisierung des Strings (muss ein
    // veränderbares Array sein)
    char *token = strtok(str, ",");
    // Erster Aufruf: Findet das erste Wort und ersetzt das ',' durch '\0'
    while (token != NULL) { // Die Schleife läuft, solange strtok noch ein
    // Teilstück findet
        printf("token: %s\n", token); // Gibt das aktuelle Teilstück aus
        token = strtok(NULL, ","); // NULL signalisiert strtok, beim alten
        // String weiterzumachen
    }
    return 0;
}
```

Standard Streams

- stdin → Eingabekanal
- stdout → Ausgabekanal
- stderr → Fehlerkanal

Beispiel mit zwei unterschiedlichen Codes mit gleichem Resultat:

```
printf("Hello World\n");
fprintf(stdout, "Hello World\n");
```

Shell mapping

```
# 1. stdout in eine Datei umleiten (überschreiben)
echo "Hallo Welt" > output.txt

# 2. stdout an eine Datei anhängen (append)
echo "Nächste Zeile" >> output.txt

# 3. stderr in eine Datei umleiten
ls /nonexistent 2> error.log

# 4. stdout UND stderr in eine Datei umleiten
ls /nonexistent /tmp 1> combined.log 2>&1
# Oder kürzer (Bash ≥4):
ls /nonexistent /tmp &> combined.log

# 5. stdin aus einer Datei lesen
grep "Welt" < output.txt

# 6. stdout von einem Befehl als stdin für den nächsten verwenden (Pipe)
cat output.txt | grep "Hallo"
```

2 Präprozessor und Modulkonzept

Kopfzeilenanweisungen (Direktiven)

- #include → inkludiere eine Headerfile (<...> → im allgemeinen lib ordner des Compilers; '...' → im Verzeichnis des Programms) kopiert anschliessend diese Files in das Programm
- #define → definiere ein Makro ersetzt das Makrowort mit dem definierten text/code/zahl

Für Headerfiles:

- #ifdef → ist es schon definiert?
- #ifndef → wenn nicht definiert
- #endif → beenden des Defineblockes

Beispiel Headerfile mit Includeguard (test.h)

```
#ifndef TEST_H // wenn TEST_H nicht definiert ist
#define TEST_H // definiere TEST_H;
// sollte TEST_H schon definiert worden sein springt von ifndef direkt zu
// endif -> verhindert mehrfaches definieren des gleichen Headers
... // define Space
#endif
```

Modularisierung

- Aufspaltung eines grossen Projekts in kleinere Unterprogramme die losgelöst sind vom grossen Monolith
- Einfachere Wartung durch Mitprogrammierende
- Sichtbarkeit von Variablen und Funktionen steuern

3 Dynamische Speicherverwaltung

Speicherorganisation

- **Programmcode**
- **Variables** → Globale Variablen
- **Heap** → Dynamischer Speicher
- **Stack** → Statischer Speicher (Lokale Variablen in Funktionen)
- **static** → Variablen nur sichtbar im gleichen Modul (wenn in Funktion: Variable bleibt nach dem verlassen der Funktion bestehen)
- **extern** → greife auf globale Variablen zu (auch Modulübergreifend)

Dynamischer Speicher (<stdlib.h>)

- **malloc** → alloziere neuen Speicherplatz
- **realloc** → verschiebe und reserviere neuen Speicherplatz
- **calloc** → alloziere neuen Speicherplatz und initialisiere den auf 0
- **free** → gibt reservierten Speicherplatz frei (kann danach wieder von anderen Prozessen beschrieben werden)

Beispielcode für Dyn. Speicher (<stdlib.h>)

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *zahlen = malloc(2 * sizeof(int)); // 1. MALLOC: Reserviere Speicher für 2 Integer
    if (zahlen == NULL) return 1; // Sicherheitshalber prüfen

    zahlen[0] = 10; zahlen[1] = 20;
    printf("Nach malloc: %d, %d\n", zahlen[0], zahlen[1]);

    int *temp = realloc(zahlen, 3 * sizeof(int)); // 2. REALLOC: Speicherplatz erweitern
    if (temp != NULL) {
        zahlen = temp; // Temp-Peinter nutzen gegen Datenverlust
        zahlen[2] = 30; // Dritte Zahl speichern
    }

    printf("Nach realloc: %d, %d, %d\n", zahlen[0], zahlen[1], zahlen[2]);
    free(zahlen); // 3. FREE: Speicherblock wird freigegeben
    // ACHTUNG: zahlen und temp sind jetzt Dangling Pointers -> wird zu Laufzeitproblemen führen
    zahlen = NULL; // Best Practice
    temp = NULL; // Wichtig, um Dangling Pointer zu vermeiden
    return 0;
}
```

4 Verkettete Listen

Structs die in sich einen Pointer haben der auf das nächste Struct zeigt ergeben eine Liste.

- **einfach Verkettet** → kann nur vom Anfang her durchiteriert werden
- **doppelt Verkettet** → kann von zwei Seiten durchiteriert werden
- **Binär Baum** → hat zwei Zeiger auf zwei weitere Items (gut für Grosse Datenbanken)

Beispiel Struct:

```
typedef struct myListedStruct {
    int rank; // testweise mit int rank fuer das Verstaendnis
    struct myListedStruct *next; // so ist es eine einfach Verkettete Liste
    struct myListedStruct *last; // so ist es eine doppelt verkettete Liste
    // aber im Bsp nicht nötig
} myListedStruct;
```

Einfügen sortiert nach Rang (Verkettete Liste):

```
myListedStruct* insertSorted(myListedStruct *head, int newRank) {
    myListedStruct *newNode = malloc(sizeof(myListedStruct));
    newNode->rank = newRank; // setzen den neuen Rang
    newNode->next = NULL;

    if (head == NULL || newRank < head->rank) { // Fall: Liste leer oder neues Element wird Kopf
        newNode->next = head;
        return newNode; // Der neue Knoten ist jetzt der Start
    }

    myListedStruct *current = head; // Wandern bis zur richtigen Stelle
    while (current->next != NULL && current->next->rank < newRank) {
        current = current->next;
    }

    newNode->next = current->next; // umhaengen der next pointer
    current->next = newNode;

    return head; // Urspruenglicher Kopf bleibt Start
}
```

Beispiel für Binärbaum

```
#include <stdio.h>
#include <stdlib.h>

// Definition eines Knotens im Binärbaum
typedef struct Node {
    int data;
    struct Node* left;
    struct Node* right;
} Node;

// Funktion zur Erstellung eines neuen Knotens
Node* createNode(int value) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    if (newNode == NULL) {
        printf("Speicherfehler!\n");
        exit(1);
    }
    newNode->data = value;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}

// Inorder-Traversierung (gibt den Baum sortiert aus, wenn es ein Suchbaum ist)
void printInorder(Node* root) {
    if (root == NULL) return;

    printInorder(root->left); // Gehe nach links
    printf("%d ", root->data); // Verarbeite aktuellen Knoten
    printInorder(root->right); // Gehe nach rechts
}

int main() {
    // 1. Wurzel erstellen
    Node* root = createNode(10);

    // 2. Kinder hinzufügen (Beispiel für einen einfachen Baum)
    root->left = createNode(5);
    root->right = createNode(15);
    root->left->left = createNode(2);
    root->left->right = createNode(7);

    // Baum ausgeben
    printf("Inorder-Ausgabe des Baums: ");
    printInorder(root);
    printf("\n");

    // Hinweis: Im echten Einsatz sollte hier noch der Speicher mit free() freigegeben werden!
    return 0;
}
```

Zusammenfassung INF2 – Abteilung Cpp

Autor: chrisvin

5 C vs Cpp

- Umstieg von prozeduraler Sprache in OOP (Object Oriented Programming)
- **Namespace:** Zusammenfassen von Funktionen/Methoden unter einem Namen (z.B. using namespace std statt std::string)
- **Referenzen:** Im Hintergrund wie Pointer mit automatischer Dereferenzierung. Sie wissen also immer über das Objekt Bescheid, auf das sie zeigen. Sind für die Programmierung einen Alias (anderen Namen für das Objekt)
- **Kapselung:** Zugriffsberechtigungen modifizieren mit Attributen (public, protected, private)
- **Datentypen:** Neue native Typen wie bool, std::string etc.

Beispielcode (Namespace & Kapselung):

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

namespace MeinNamespace {
    class Kapselung {
    public:
        Kapselung(const std::string& wert) : geheimnis(wert) {}
        const std::string& getGeheimnis() const { return geheimnis; }
    private:
        std::string geheimnis; // Kapselung: privat
    };
}

int main() {
    MeinNamespace::Kapselung obj("Hallo Welt"); // Namespace-Aufruf mit "::"
    const std::string& ref = obj.getGeheimnis(); // Referenz auf gekapseltes Datum
    std::cout << ref << std::endl; // Ausgabe: Hallo Welt
    return 0;
}
```

Konstruktoren-Typen

- **Default-Konstruktor:** Parameterlos. Wird vom Compiler generiert, wenn *kein* anderer Konstruktor definiert ist.
- **Copy-Konstruktor:** Initialisiert Objekt mit bestehendem Objekt gleichen Typs (T(const T&)).
- **Explicit:** Das Schlüsselwort explicit verhindert ungewollte, implizite Typkonvertierungen durch den Compiler bei Single-Argument-Konstruktoren.

Syntax Copy & Explicit:

```
class ExplicitBeispiel {
public:
    // explicit verhindert: ExplicitBeispiel e = 42;
    explicit ExplicitBeispiel(int x) : data(x) {}

    // Copy-Konstruktor (per constant-reference!)
    ExplicitBeispiel(const ExplicitBeispiel& other) : data(other.data) {}
private:
    int data;
};
```

6 Kapselung & Klassen

- **struct:** Kann in C++ neu auch Funktionen/Methoden enthalten
- **class:** Ähnlich wie struct, aber standardmässig sind alle Members private
- **public:** Members können von überall her aufgerufen werden
- **protected:** Members können nur von abgeleiteten Klassen aufgerufen werden
- **private:** Members können nur innerhalb der eigenen Klasse aufgerufen werden
- **static:** static-Methoden können ohne Objekt aufgerufen werden, besitzen keinen this Pointer und dürfen nur auf statische Member zugreifen (siehe auch Kap. 15)

Beispiel Struktur einer Klasse:

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

class myClass {
public:
    // Konstruktor mit Elementinitialisierungsliste (vor dem Rumpf)
    myClass(const std::string& name) : name_(name) {
        std::cout << "Der Konstruktor baut auf\n";
    }
    ~myClass() { // Destruktor
        std::cout << "Der Destruktor räumt auf\n";
    }
private:
    const std::string name_;
};
```

7 I/O Handling

- **<iostream>:** I/O auf der Konsole (std::cout, std::cin)
- **<fstream>:** I/O in Dateien

- **std::endl:** Zeilenumbruch + Buffer-Flush (== "\n")
- **cout.fill('*');** Füllt leere Felder mit Argument auf
- **cout.width(10) / setw(10):** Setzt minimale Anzeigelänge
- **Datei-Modi:** ios::in (Lesen, analog zu "r"), ios::out (Schreiben, analog zu "w")

Beispiel I/O Ströme & Manipulatoren:

```
#include <iostream>
#include <iomanip>
#include <string>

int main() {
    std::string myString = "Einschub";
    std::cout << "Text " << myString << " Textende" << std::endl;

    float myFloat = 530.55;
    // Nutzen von iomanip Manipulatoren (setfill, setw)
    std::cout << std::setfill('*') << std::setw(10) << myFloat << std::endl;
    // Output: *****530.55
}
```

Const-Correctness & Referenzen

- Eine Referenz (&) ist ein unveränderlicher Alias. Sie kann nach der Initialisierung nie mehr auf ein anderes Objekt zeigen.
- **Regel:** Eine non-const Referenz kann **nicht** mit einer const-Variablen initialisiert werden (Schreibschutz-Verletzung).
- **Ausnahme:** Eine const-Referenz darf alles! Sie kann sowohl const-Variablen, non-const-Variablen als auch temporäre R-Values (z.B. Literale wie 42) binden.

Referenzen-Erlaubnisse:

```
int x = 10;
const int cx = 20;

int& ref1 = x; // Erlaubt
//ref1 = cx; // Erlaubt! Ändert nicht die Referenz, sondern den Wert von x!
// int& ref2 = cx; // COMPILER-ERROR! (Konstanz verletzt)
const int& ref3 = x; // Erlaubt (Lesezugriff eingeschränkt)
const int& ref4 = cx; // Erlaubt
const int& ref5 = 5; // Erlaubt (Temporäres Literal gebunden)
```

8 Überladen von Operatoren

Erlaubt die Neuzuweisung bzw. Erweiterung der Logik von Operatoren für eigene Datentypen.

Beispiel für den Stream-Output-Operator <<:

```
#include <iostream>

class Person {
public:
    std::string name = "John";
};

// Definition des globalen ueberladenen Operators
std::ostream& operator<<(std::ostream& lhs, const Person& rhs) {
    lhs << rhs.name; // Beschreibung der Funktionalitaet
    return lhs; // Erlaubt das Chaining von <<
}
```

9 Container und Templates

Arbeiten im Hintergrund mit Schablonen (Templates) zur typunabhängigen Speicherverwaltung im Namespace std.

- **vector<T>**: Dynamisches, kontinuierliches Array
- **array<T, N>**: Statisches Array mit fester Grösse *N*
- **list<T>**: Doppelt verkettete Liste
- **set<T>**: Geordnete Kollektion strikt einzigartiger Elemente
- **unordered_set<T>**: Ungeordnete Kollektion einzigartiger Elemente
- **auto**: Compiler bestimmt Datentyp zur Compilezeit selbst

Iteratoren & STL-Algorithmen

- **Iteratoren** verhalten sich wie verallgemeinerte Pointer und erlauben das Durchwandern von Containern mittels **begin()** und **end()**.
- **end()** zeigt immer auf das Element **nach** dem letzten gültigen Eintrag (exklusives Ende: [begin, end)).
- **<algorithm>**: Bietet optimierte Standard-Operationen wie **std::sort** oder **std::find**.

STL-Algorithmen anwenden:

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

int main() {
    std::vector<int> v = {3, 1, 4};

    // Sortieren des gesamten Vektors
    std::sort(v.begin(), v.end());

    // Suchen nach der Zahl 4
    auto it = std::find(v.begin(), v.end(), 4);
    if (it != v.end()) {
        std::cout << "Gefunden: " << *it << "\n"; // Dereferenzierung
    }
}
```

10 Speicherverwaltung Cpp

- **Rule of 3**: Explizite Definition von Destruktor, Copy-Konstruktor und Copy-Zuweisungsoperator nötig, falls eine Klasse Ressourcen händisch verwaltet.
- **new / delete**: Alloziert Speicher für ein Einzelobjekt auf dem Heap und ruft Konstruktor / Destruktor auf.
- **new[] / delete[]**: Erstellt / löscht Arrays auf dem Heap dynamisch.

Beispiel new / delete:

```
#include <iostream>

class Point {
public:
    Point(double x, double y) : x_(x), y_(y) { std::cout << "Point(x,y)\n"; }
    ~Point() { std::cout << "~Point()\n"; }
private:
    double x_{0.0}, y_{0.0};
};

int main() {
    auto *point = new Point{3.0, 4.0}; // Allokation & Konstruktion
    delete point; // Destruktion & Freigabe
    point = nullptr; // Gegen Dangling-Pointer
}
```

Smart Pointer

std::unique_ptr übernimmt das Resource-Management automatisch über das RAII-Prinzip (beim Verlassen des Scopes wird **delete** gefeuert). Destruktoren in umgekehrter Reihenfolge wie Konstruktoren (zuletzt erstellte Objekte werden als erstes gelöscht).

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

class Point {
public:
    Point(double x, double y) : x_(x), y_(y) { std::cout << "Point(x,y)\n"; }
    ~Point() { std::cout << "~Point()\n"; }
    double getX() { return x_; }
private:
    double x_{0.0}, y_{0.0};
};

void f() {
    std::unique_ptr<Point> point{new Point{3.0, 4.0}};
    if (point->getX() > 0.0) { /* ... */ }
    // Automatischer Destruktoraufruf hier am Scope-Ende!
}
```

NO_COPY_NO_MOVE

- Verhindert **Slicing** (Verlust von abgeleiteten Informationen bei Wertzuweisung an eine Basisklasse).
- Löscht gezielt Kopier- und Verschiebefunktionalitäten aus verwaltenden Klassen oder Basisklassen.

Makro für Rule of 5 (Deaktivierung):

```
#define NO_COPY_NO_MOVE(CLASS) \
CLASS(const CLASS &) = delete; \
CLASS &operator=(const CLASS &) = delete; \
CLASS(CLASS &&) = delete; \
CLASS &operator=(const CLASS &&) = delete

// Entfernt systematisch: Kopierkonstruktor, Kopier-Zuweisung,
// Move-Konstruktor und Move-Zuweisungsoperator
```

11 Vererbung

Ermöglicht das Übernehmen (Kopieren) von Basisklassen-Members in eine abgeleitete (derived) Klasse.

Beispiel Statische Objekthierarchie:

```
#include <iostream>

class Base {
public:
    void printB() const { std::cout << "B\n"; } // Base HAS A void printB() (composition)
};

class Derived : public Base { // Derived IS A Base (inheritance)
public:
    void printD() const { std::cout << "D\n"; }
};

int main() {
    Derived d(); // Auf dem Stack (statischer Speicher)
    Base& r = d; // Referenz (Kein Slicing!)
    Base *p = &d; // Pointer (Kein Slicing!)
    Base bObj = d; // ACHTUNG: SLICING! Schneidet Derived-Inhalte ab.
    d.printD(); // Eigene Methode
    d.printB(); // geerbte Methode
    r.printB(); // Hinweis: r.printD() geht nicht, da der statische Typ von 'r' nur 'Base' kennt!
    p->printB(); // Geerbte Methode über alle Wege
}
```

12 Polymorphismus

- **virtual**: Methoden der Basisklasse mit diesem Präfix werden zur Laufzeit dynamisch durch gleichnamige Implementierungen der Kindklasse überschrieben (override).
- **pure virtual**: Erzwingt das Überschreiben in abgeleiteten Klassen (**virtual void f() = 0;**). Macht eine Klasse abstrakt (keine Direktinstanzen erlaubt).
- **ABC**: Abstract Base Class (Klasse, die mindestens eine rein virtuelle Methode enthält).

Beispiel Virtuelle Methoden & ABC:

```
#include <iostream>

class abstractBaseClass {
public:
    abstractBaseClass() {}
    virtual ~abstractBaseClass() {} // Struktur-Pflicht bei Polymorphie!
    virtual void doIt() = 0; // Pure virtual Methode
};

class baseClass : public abstractBaseClass {
public:
    void doIt() override { std::cout << "Base erledigt.\n"; } // override ist
    // defensive Programmierung die Verhindert das keine nicht
    // überschreibbare Methoden überschrieben werden
    virtual void PrintvB() { std::cout << "vB (Basis)\n"; }
    void PrintB() { std::cout << "B (Basis statisch)\n"; }
};

class derivedClass : public baseClass {
public:
    void doIt() override { std::cout << "Derived erledigt.\n"; }
    void PrintD() { std::cout << "D (Nur in Derived)\n"; }
    void PrintvB() override { std::cout << "DvB (Dynamisch aus Derived!)\n"; }
};

int main() {
    derivedClass d{};
    baseClass& rb = d; // Basis-Referenz auf Kindobjekt

    // 1. DYNAMISCHES BINDEN (Polymorphie):
    rb.PrintvB(); // Ausgabe: DvB (Dynamisch aus Derived!)
    rb.doIt(); // Ausgabe: Derived erledigt.

    // 2. STATISCHES BINDEN:
    rb.PrintB(); // Ausgabe: B (Basis statisch)

    // 3. DIREKTER AUFRUF DURCH SCOPE
    rb.baseClass::doIt(); // Ausgabe: Base erledigt.

    // 4. SLICING BEISPIEL (kann verhindert werden durch NCNM):
    baseClass bc = d;
    bc.doIt(); // Ausgabe: Base erledigt.
    // Begründung: bc kopiert NUR den baseClass-Anteil von d auf den Stack.
    // Der derivedClass-Teil wird abgeschnitten (Slicing).
    // Da baseClass nicht mehr abstrakt ist, ist bc voll funktionsfähig.
    return 0;
}
```

13 Errorhandling (Exceptions)

- **try-catch:** Funktionen signalisieren Ausnahmezustände mittels throw. Diese werden in einem umschließenden try-Block abgefangen.
- (...): Universeller Catch-Block (fängt jeden Fehler-typ ab).
- `std::logic_error`: `invalid_argument`, `out_of_range`.
- `std::runtime_error`: `overflow_error`, `range_error`.
- Weitere Standard-Exceptions: `bad_alloc`, `bad_cast`.

Beispiel Exception-Szenario:

```
#include <iostream>
#include <stdexcept>

double dividiere(double a, double b) {
    if (b == 0) {
        throw std::runtime_error("Division by 0!"); // Fehler werfen
    }
    return a / b;
}

int main() {
    try {
        double resultat = dividiere(10, 0);
    } catch (const std::exception& e) { // Fangen per Referenz (Best Practice)
        std::cout << "Error: " << e.what() << std::endl;
    }
    return 0;
}
```

14 STL (Standard Template Library)

Die 4 Säulen der STL (Hauptkomponenten)

- **Container:** Datenhaltung (z.B. `vector` (Array auf Stack/Heap), `map` (Key-Value-Baum)).
- **Iteratoren:** Zeiger-ähnliche Objekte zum Navigieren durch Container (`++it`, `*it`).
- **Algorithmen:** Generische Funktionen wie `std::sort()` oder `std::find()`.
- **Funktions-Objekte:** Callables/Funktoen zur Konfiguration von Algorithmen.

STL Iteratoren (Wichtig für Schleifen)

Iteratoren imitieren Pointer-Verhalten. Bereiche sind immer **inklusive begin**, **exklusive end** ($begin \leq x < end$).

- Idiomatisches C++: Immer `++it` statt `it++` verwenden (Performance!).
- **Const-Iterator:** Nur lesend zugreifen via `c.begin()` / `c.cend()`.

```
std::vector<int> v = {1, 2, 3};

// Klassische Iterator-Schleife (Lese- und Schreibzugriff):
for (auto it = v.begin(); it != v.end(); ++it) {
    *it = *it * 2; // Dereferenzierung wie bei Pointern
}

// Moderne Range-based For-Schleife (Sicher & Idiomatisch):
for (const auto& element : v) {
    std::cout << element << " ";
}
```

Assoziative Container (`std::map`)

Speichert Key-Value-Paare sortiert nach Key. **Achtung Fangfrage:** Der Operator `dict[key]` erzeugt automatisch

einen neuen Standardeintrag (Value = 0), falls der Key noch nicht existiert!

15 Weiteres

Beispielcode für Casting

```
double d = 1.2;
int i = (int)d; // C-Style-Cast: riskant, da er "alles" erzwingt
int j = static_cast<int>(d); // C++-Cast: sicher, wird zur Compilezeit
// geprüft
```

Beispiel `static`-Klassenmember und Funktor

```
class MyClass {
public:
    MyClass(int factor) : factor_(factor) {} // Konstruktor
    int operator()(int x) const {
        return x * factor_;
    }
    // hier überladen wir den "()" Operator, also den Funktionsaufrufoperator
    // und machen die Klasse somit zu einem Funktor

    int multiply_by(int x){
        // standard member methode (funktion nur auf Instanzen der Klasse
        // verfügbar)
        return x * factor_;
    }

    static int double_the_value(int x){
        // Statische Methode welche OHNE Instanziierung verfügbar ist
        return two_times * x;
    }

    static const int two_times = 2;
    // muss public (damit extern sichtbar), static (damit ohne Instanziierung
    // verfügbar) und const (weil ISO C++ so verlangt) sein, damit dieser
    // Wert von double_the_value() benutzt werden kann

private:
    int factor_; // standard private member variable
};

int main(){
    MyClass three_times(3); // Instanz von MyClass mit factor_ = 3
    int res0 = three_times(5); // res0=15 "Instanz wird direkt aufgerufen"
    // --> geht weil operator() definiert (Funktor)

    int res1 = three_times.multiply_by(6); // res=18 Klassischer Weg mit
    // Aufruf einer Member Funktion einer instanziierten Klasse

    int res2 = MyClass::double_the_value(4); // res2 = 8 Statische Methode
    // kann auch ohne Instanz aufgerufen werden. Vgl. namespaces (von der
    // Syntax her)
}
```

Beispiel für default-Präfix

Der default Präfix erzwingt einen Leeren Konstruktor welcher mit allen 'default Konstrukoren' erzeugt wird (z.B Move-Konstruktor)

```
class MyClass{
    MyClass(int var): secret(var){} // hier definieren wir einen neuen
    nicht-Standard Konstruktor und somit wird der bestehende default
    Konstruktor überschrieben. Die Klasse kann nicht mehr mit
    MyClass() (ohne argument var) erstellt werden ausser
    MyClass() = default; // zwingt den Compiler, dass defaultkonstruktor
    immernoch gültig ist
    MyClass(){}; // gleich wie MyClass() =default ohne den move konstruktor.
    beachte dass nur eine Variante möglich ist (MyClass::MyClass()
    kann nicht zweimal exakt gleich überladen werden)

private:
    int secret;
}

int main(){
    myinst = MyClass(); // geht sofern MyClass(){} oder MyClass() = default
    definiert ist
    myonst = MyClass(55);
}
```