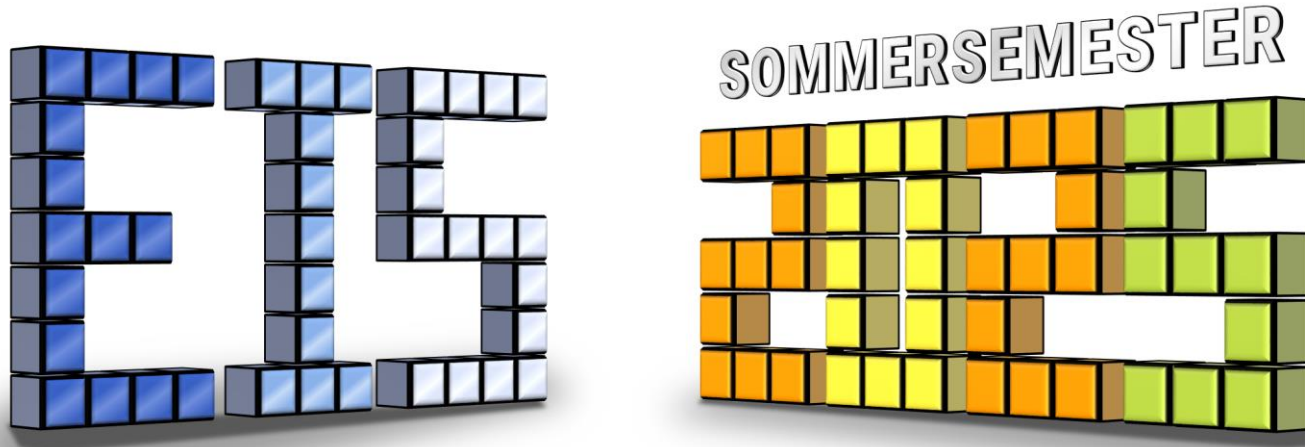


EINFÜHRUNG IN DIE SOFTWAREENTWICKLUNG

Sommersemester 2025



Foliensatz #10

Hardwarenahe Programmierung

Michael Wand
Institut für Informatik
Michael.Wand@uni-mainz.de



Inhaltsverzeichnis

Inhalt EIS Sommer 2024

- Programmiersprachen
- Programmentwurf / Muster
 - Prozedural
 - Objekt-orientiert
 - Funktional
- Effiziente Programmierung
 - Hardwarenahe Abstraktionen
- Nebenläufige Programmierung & Architekturen
- Fortgeschrittene Ideen & Architekturen
 - Meta-Programmierung
 - Fortgeschrittene Muster & Architekturen

Low-Level Programmierung

Hardwarenahe Programme

Verschiedene Aspekte

- Performance
 - Hardware „besser ausreizen“
- Direkter Zugriff
 - Betriebssystemtreiber und Komponenten
- Abstraktionen
 - Schnell + Komfortabel/Sicher?

Unser Fokus: 1+3

- Effizienter Code am Beispiel C++
- Abstraktionstechniken

Lernziele

Überblick hardwarenahe Programmierung

■ Vorlesung:

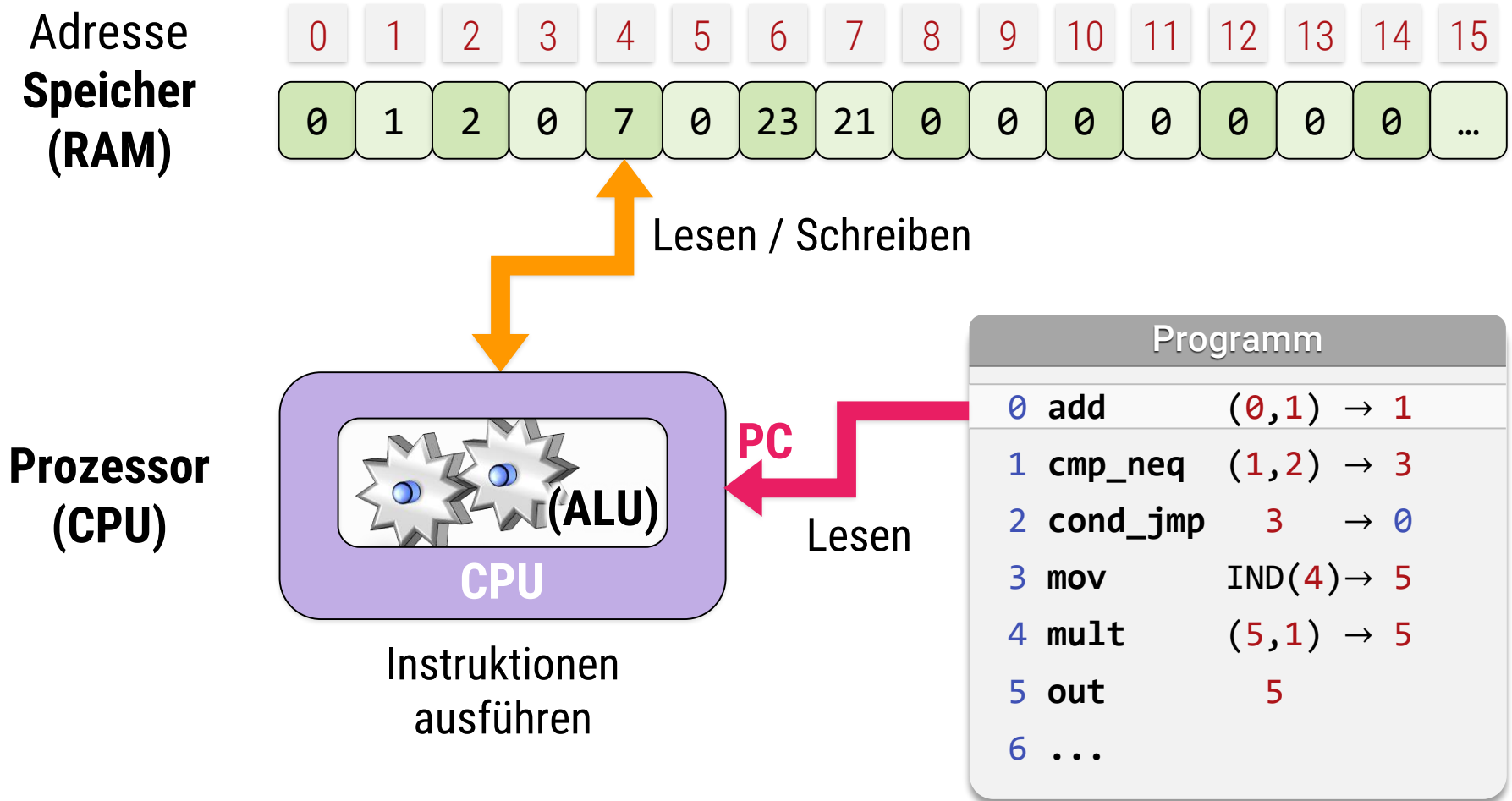
- Programmiertechniken
- Abstraktionsmechanismen
- Übersichtsartig, Details nicht wesentlich

■ Übungen:

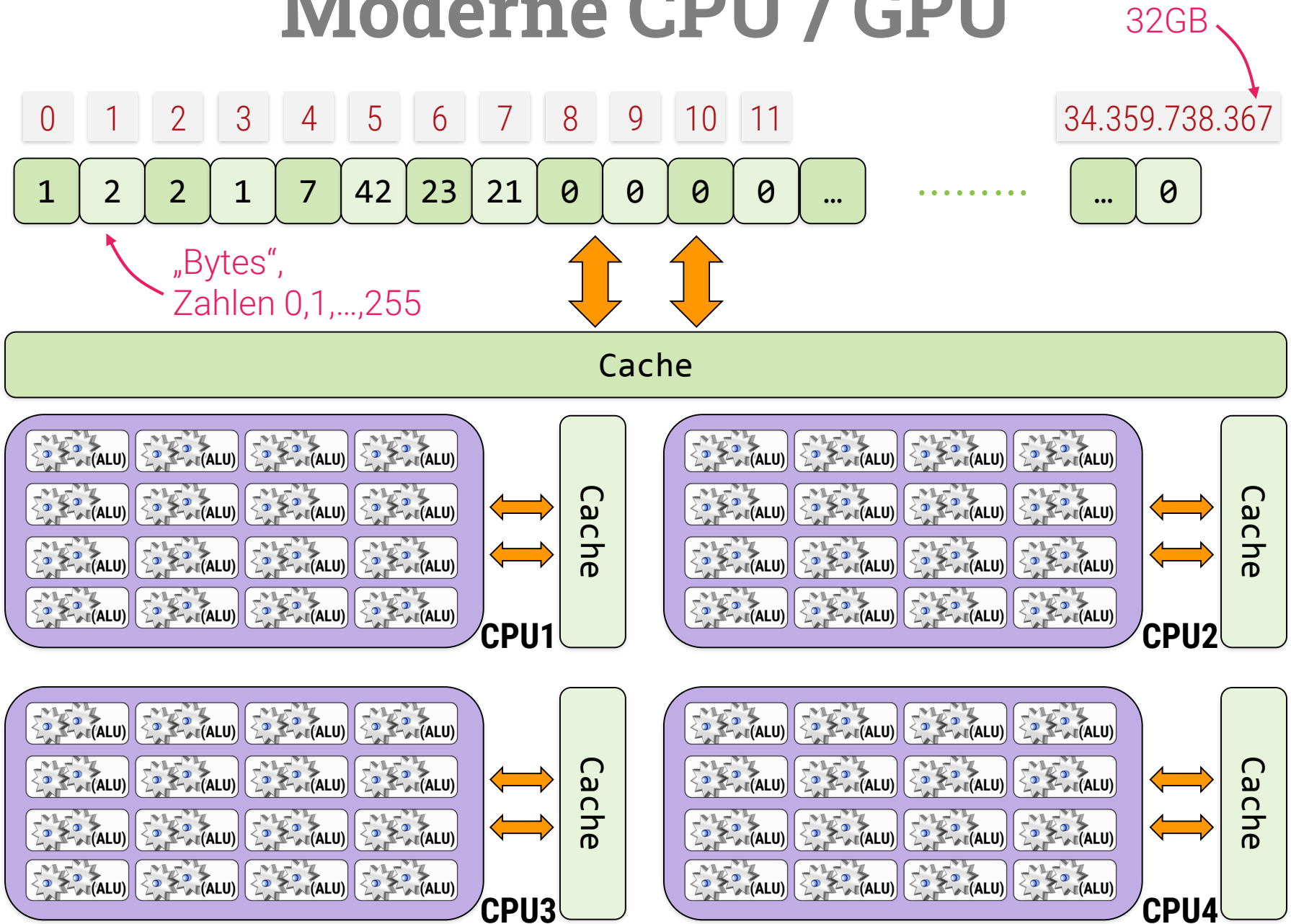
- Programmieren einfacher, schneller
Datenverarbeitung in C++
- Optional: Integration C++ & Python (nützlich)

Computer: Hardware

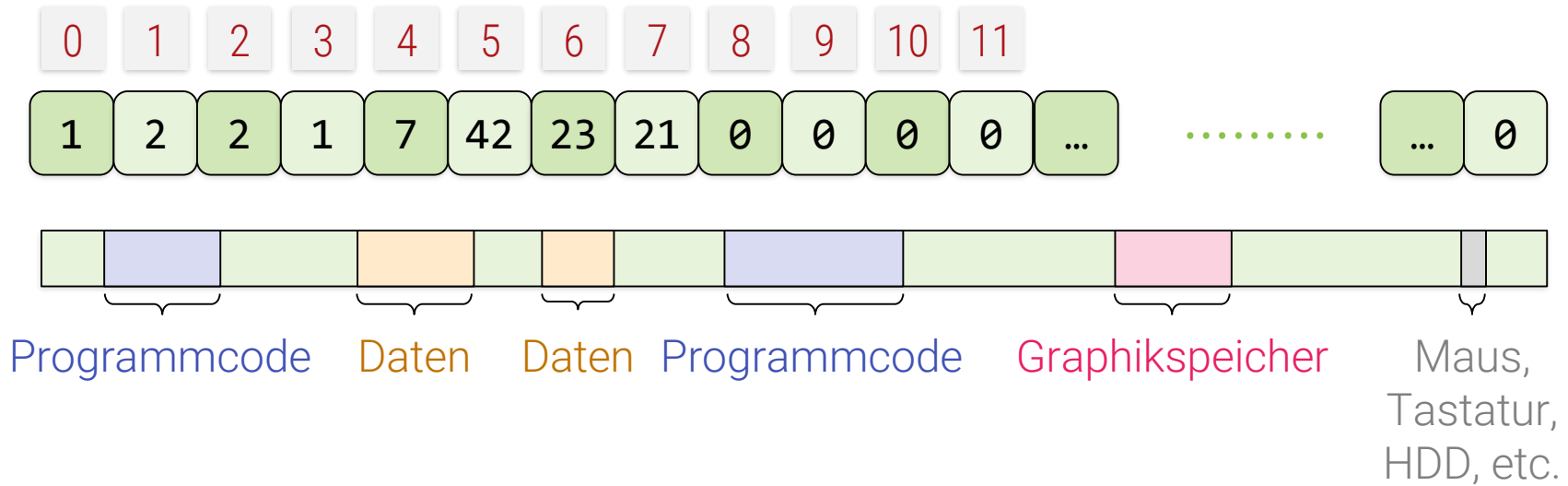
Ein abstrakter Computer



Moderne CPU / GPU



Rechnerorganisation



Von-Neumann Rechner

- Programme im selben Speicher wie Daten

Memory-Mapped I/O

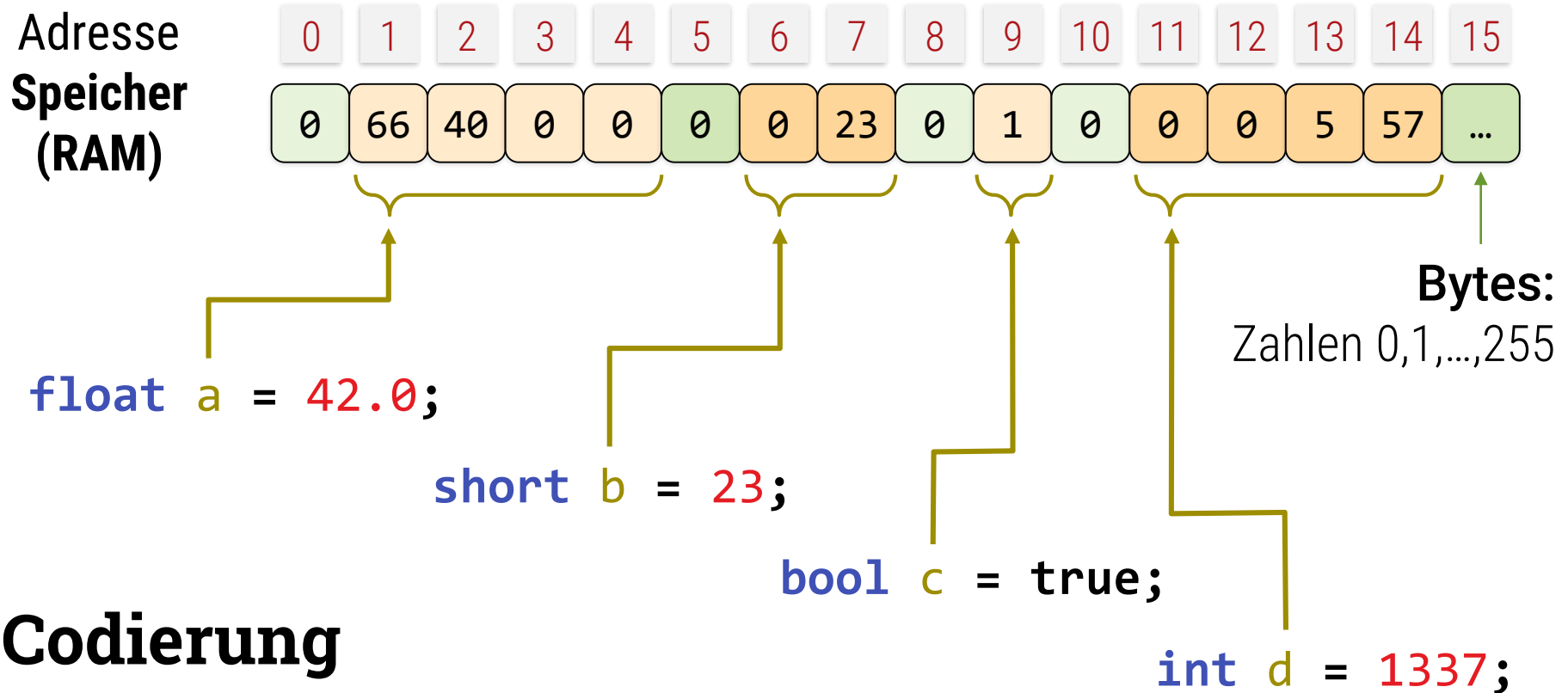
- Pseudo-Speicherzellen für Geräte und Bildschirm

Interrupts:

- Code wird angesprungen bei Ereignissen (z.B.: Maus bewegt)

Zur Erinnerung: Zeiger

Maschinenrepräsentation



Codierung

- Aufeinanderfolgende Bytes
- Variable = Zeiger auf erstes Byte (Länge implizit)

Zeiger in C++

Zeiger:

- Integraler Bestandteil der Sprache
- Wichtig effizientes und maschinennahes Programmieren
- Wichtig für OOP („Identity Objects“ wie in Python)

Drei Regeln

- **&<var>**: Adresse (Zeiger auf) <var>
- ***<var>**: Inhalt von Zeiger <var>
- **<typ>***: Zeigertyp auf <typ>



nicht verwechseln
mit **Referenzen**
<Typ> &

Zeiger in C++

Zeigertypen

- Verschiedene Zeigertypen!
 - `int*`, `float*`, `short unsigned int*` etc.
- „Bezugstyp“ – C++ weiß, worauf der Zeiger zeigt!
 - Konvertierung ist möglich (explizit via cast)
- Arrays sind Zeiger
 - Bemerkung: `std::vector` ist komplexer


Optimieren mit Zeigern

Beispiel: Bildverarbeitung

```
struct Pixel {float r; float g; float b;}  
  
struct Image {  
    Pixel *buffer; int width; int height;  
  
    Image(int w, int h) {  
        with = w; height = h;  
        buffer = new Pixel[w*h];  
    }  
  
    inline Pixel &operator[](int x, int y) {  
        if (x<0) x=0; if (x>=w) x=w-1;  
        if (y<0) y=0; if (y>=h) y=h-1;  
        return buffer[x + w*h];  
    }  
  
    ~Image() {  
        delete[] buffer;  
    }  
  
};
```

Beispiel: Bildverarbeitung

```
void brightness_contrast(Image &img,  
                           float bright, float contr) {  
    for (int y=0; y<img.height; y++) {  
        for (int x=0; x<img.width; x++) {  
            img[x, y].r = img[x, y].r * contr + bright;  
            img[x, y].g = img[x, y].g * contr + bright;  
            img[x, y].b = img[x, y].b * contr + bright;  
        }  
    }  
}
```



*Code kann noch recht „langsam“ sein.
(je nachdem, wie schlau der Compiler ist)*

Beispiel: Bildverarbeitung

```
struct Pixel {float r; float g; float b;}
struct Image {Pixel *buffer; int width; int height; ... }

void brightness_contrast_opt(Image &img,
                             float bright, float contr) {

    float *ptr = (float*)img.buffer; // densely packed floats
    float *end = ptr + 3 * img.width * img->height;

    for (; ptr<end; ptr++) {
        *ptr = *ptr * contr + bright;
    }
}
```

Geht immer noch schneller...

```
struct Pixel {float r; float g; float b;}  
struct Image {Pixel *buffer; int width; int height; ... }  
  
void brightness_contrast_AVX2(Image &img,  
                               float bright, float contr) {  
  
    float *ptr = (float*)img.buffer; // densely packed floats  
    float *end = ptr + 3 * img.width * img->height;  
    float *end8 = ptr + (3 * img.width * img->height) / 8 * 8;  
  
    for (; ptr<end8; ptr+=8) { // vectorized! (multiple of 8)  
        __m256 *vec_ptr = (__m256*)ptr; // pseudo-code (impl. dependent!)  
        *vec_ptr = *vec_ptr * contr + bright; // pseudo code (vectorized)  
    }  
  
    for (; ptr<end; ptr++) { // now the rest (unvectorized)  
        *ptr = *ptr * contr + bright;  
    }  
}
```

Diskussion Beispiel

C++ erlaubt

- Eigenes Speicherlayout anlegen
 - Hier die „Image“ Klasse
 - Ordnet Elemente selbst im Speicher an
- Maschinenbefehle für Zugriff
 - Mit Compiler/CPU-abhängigen Erweiterungen auch Spezialitäten wie MMX, SSE, AVX, NEON, u.ä.
- Low-Level Optimierungen
 - Zeiger „casten“ um CPU am besten zu „füttern“
- Es kann sein, dass alle Code-Beispiele gleich schnell laufen (C++-Compiler optimieren aggressiv)

Geht immer noch schneller...

```
void brightness_contrast_AVX2(Image &img,
                               float bright, float contr) {

    float *ptr = (float*)img.buffer; // densely packed floats
    float *end = ptr + 3 * img.width * img->height;
    float *end8 = ptr + (3 * img.width * img->height) / 8 * 8;

    #pragma omp parallel for // Alle Cores (via „OpenMP“)
    for (; ptr < end8; ptr += 8) { // vectorized! (multiple of 8)
        __m256 *vec_ptr = (__m256*)ptr; // pseudo-code (impl. dependent!)
        *vec_ptr = *vec_ptr * contr + bright; // pseudo code (vectorized)
    }

    for (; ptr < end; ptr++) { // now the rest (unvectorized)
        *ptr = *ptr * contr + bright;
    }
}
```

**) OMP macht der compiler nicht automatisch :-)*

Hardwarenahe Programmierung

Hardware ansprechen

Treiber / BS Programmierung

- Zugriff auf feste Adressen
- z.B. Kontrollregister (Memory-Mapped I/O)
- Möglich über konstante Pointer
 - Cast von `(long) int` auf Pointer-Typ

Hardware ansprechen

```
// Beispiel: Hintergrundfarbe auf C64  
// (Memory-mapped „MOS/VIC 6569“ Videocontroller)  
// Programm läuft nur auf einem Comodore C64
```

```
uint8_t *border      = (uint8_t*)53280; // Rahmenfarbe  
uint8_t *background  = (uint8_t*)53281; // Hintergrundfarbe  
  
*border = (uint8_t*)14; // Hellblau  
*background = (uint8_t*)6; // Blau
```



[C64 Boot Screen]

```
// Beispiel: VGA Mode 0x13 320x200  
// Erfordert IBM-PC kompatiblen 32bit Rechner mit VGA-Graphik
```

```
uint8_t *videoMem = (uint8_t*)0xA0000; // Hexadezimal  $\cong$  655.360 dezimal  
  
for (int y=0; y<200; y++) {  
    for (int x=0; x<320; x++) {  
        videoMem[x+320*y] = 0; // Bildschirm löschen  
    }  
}
```



[FutureCrew/2nd Reality, 1993]

Beispiel: Container

Ziel: (Effiziente) Abstraktion

Kapselung

- Low-level Programmierung
 - Zeiger
 - Manuelle Speicherverwaltung mit `new` & `delete`
 - ggf. Hardwarenahe Optimierung
- LL-P ist aufwendig und fehleranfällig
- Daher Kapseln in Komponenten

Beispiel: Wie man das in C++ macht

- Ausschnittsartig präsentiert

Klasse für Listen (Arrays)

Containerklasse: Listen

```
struct IntList {  
    int *memory;  
    unsigned int size;  
    // Konstruktor  
    IntList(unsigned int initialSize = 0) {  
        memory = new int[initialSize]; // Speicher reservieren (0 erlaubt)  
        size = initialSize;  
    }  
    // Neu: Destruktor (bei Löschen des Objektes)  
    ~IntList(unsigned int initialSize = 0) {  
        delete[] memory; // Speicher freigeben  
    }  
    // Zugriff auf Elemente: Überladener Operator  
    int &operator[](unsigned int index) {  
        if (index >= size) {throw std::out_of_range("out of bounds");}  
        return memory[index]; // ↑↑ Neu: Exceptions – ähnlich wie in Python  
    }  
};
```

Klasse für Listen (Arrays)

Containerklasse: Listen

```
struct IntList {  
    ...  
    // copy-Konstruktor  
    IntList(const IntList &other) {  
        memory = new int[other.size]; // Speicher reservieren  
        size = other.size;             // Daten kopieren  
        for (int i=0; i<size; i++) {   // Daten kopieren  
            memory[i] = other.memory[i];  
        }  
    }  
    // Zuweisungsoperator  
    void operator=(const IntList &other) {  
        delete[] memory;  
        memory = new int[other.size]; // Speicher reservieren  
        size = other.size;             // Daten kopieren  
        for (int i=0; i<size; i++) {   // Daten kopieren  
            memory[i] = other.memory[i];  
        }  
    }  
}
```

Benutzung


```
IntList *doSomething(int num) {  
    IntList l1(3);  
    for (int i=0; i<42; i++) {  
        IntList l2(3); // Konstrukturaufruf l2! (42 mal!)  
        l2[0] = 23;  
        l2[1] = 42;  
        l2[2] = 1337;  
    } // Destrukturaufruf l2! (42 mal!)  
    IntList *l3 = new IntList(3); // Konstrukturaufruf l3  
    return l3; // l3 überlebt return, da auf Heap angelegt  
} // Destrukturaufruf l1! (einmal pro Aufruf von doSomething)  
  
void main() {  
    IntList *my_l = doSomething();  
    my_l[2] = 1337;  
    delete my_l;  
}
```

RAII – Halbautomatischer Speicher

```
{  
    IntList l2(3); // Konstruktor  
    l2[0] = 23;  
    l2[1] = 42;  
    l2[2] = 1337;  
} // Destruktor (automatisch)  
// „Scope“ {} verlassen!
```

„RAII“-Stil

```
{  
    IntList *l2 = new IntList(3);  
    l2[0] = 23;  
    l2[1] = 42;  
    l2[2] = 1337;  
}  
...  
delete l2;
```



Kein „RAII“-Stil

(üblich in JAVA/SCALA, dort ohne delete)

RAII

- „Resource Acquisition Is Initialization“
- Lokale Variablen benutzen, um Ressource (z.B. Speicher) automatisch freizugeben
- Macht Speicherverwaltung einfach (geht nicht immer)

Speicherverwaltungsstrategien

Wir brauchen ein Architekturmodell

- RAII
 - Einfach und sicher, aber Lebensdauer beschränkt auf Blöcke
 - Keine „Persistenz“ (z.B. Dokument einer Anwendung)
- „Ownership“-Modell
 - Baumstrukturierter Objektgraph
 - Eltern-Objekt ist „Owner“ der Kinder (verantwortlich für delete)
 - Querverweise (non-owned) möglich, kein delete
- Referenz-Counting
 - Für azyklisch-gerichtete Graphen (mehrere Owner)
- Allgemeine Graphen: Sonderlösungen oder GC

Spezielle C++ Features

Effiziente Abstraktionen

- Konstruktoren und Destruktoren
 - Speichermanagement automatisieren
- Operatoren überladen
 - „+“, „*“, „+=“, etc.: Einfachere Schreibweise
 - „.“ oder „->“: Smart-Pointer
- Spezialoperatoren (Zuweisung, Kopieren, etc.)
 - „Compiler-interne“ Operationen definieren
 - Sehr viel fein-Tuning möglich
 - (z.B. move vs. copy seit C++11)
- „**inline**“-Funktionen (Unterprogramme ohne Kosten)

Spezielle C++ Features

Idee

- Komplexität hinter Fassade verstecken
 - Trotzdem schnell
 - „Zero-cost abstractions“
- Virtuelle Methoden sind z.B. nicht richtig „zero-cost“
 - JVM versucht „just-in-time“ Optimierung
 - In C++: templates & generische Programmierung
 - Zusätzlich zu den üblichen „Tricks“ verfügbar
 - z.B. innere Schleifen in Methoden reinziehen

Generische Container

Container für verschiedene Typen

Containerklasse: Listen

```
template <typename T>
struct List {
    T *memory;
    unsigned int size;
    // Konstruktor
    List(unsigned int initialSize = 0) {
        memory = new T[initialSize]; // Speicher reservieren
        size = initialSize;
    }
    // Destruktor (bei Löschen des Objektes)
    ~List(unsigned int initialSize = 0) {...}
    // Zugriff auf Elemente
    T &operator[](unsigned int index) {...}
    // Copy-Constructor
    List(const List<T> &other) {...}
    // Zuweisungsoperator
    void operator=(const List<T> &other) {...}
};
```

Templates

```
List<int> l1(3);  
l1[0] = 23;  
l1[1] = 42;  
l1[2] = 1337;
```

```
List<double> l2(3);  
l2[0] = 23.0;  
l2[1] = 42.0;  
l2[2] = 1337.0;
```

```
List<Bruch> l3(1);  
l3[0].zaehler = 23.0;  
l3[0].nenner = 42.0;
```

Statische Typisierung

- Erfordert genaue Typspezifikation
- Viele doppelte Tipparbeit

Lösung: „templates“ (generische Programmierung)

- Typparamter für Klassen/Structs oder Funktionen
- Benutzung mit spitzen Klammern (s.o.)

Generische Programmierung

(„parametrische Polymorphie“)

Templates in C++

„Generische Programmierung“

- Eine Form von Polymorphie
- Der selbe Code kann unterschiedliche Dinge tun
- In C++ motiviert & optimiert von/für Effizienz

Werkzeug in C++: Templates

- Präambel „**template** <...>“ erzeugt generischen
 - Typ (struct/class)
 - Member-Funktion (von structs/classes)
 - Funktion (alleinstehend)
- Benutzung mit **Typ**<...> bzw. **Funktion**<...>

Beispiele

// Generischer Typ

```
template <typename T>
struct List {
    T *memory;
    ...
};
```

// Generische Funktion

```
template <typename T>
T square(T val) {
    return val*val;
}
```

// Member-Template

```
struct IntList {
    int *memory;
    template <typename T>
    void setElementAs(unsigned i, const T val)
        {memory[i] = (int)val;}
};
```

// Benutzung generischer Klassen

```
List<int> a;
b = List<int>();
```

// Benutzung generischer Funkt.

```
float a = square(2.0f); // implizit
double b = square<double>(2); // explizit
```

// Benutzung generischer Memberf.

```
IntList a;
a.setElementAs<double>(2.0);
a.setElementAs(2.0f); // implizit
```

Explizite Spezialisierung


// Beispiel für Member-Templates; funktioniert mit allen Templates

```
struct IntList {  
    int *memory;  
  
    ...  
  
    template <typename T>  
    void setElementAs(unsigned i, const T val) {  
        static_assert(false, "Type not permitted."); // requires C++11 or later  
    }  
  
    template <>  
    void setElementAs<int>(unsigned int i, const int val) {  
        memory[i] = val;  
    }  
  
    template <>  
    void setElementAs<double>(unsigned int i, const double val) {  
        memory[i] = (int)val;  
        cout << "Warning: had to round value.";  
    }  
};
```

Template Parameter

```
// Generisches Dictionary (langsam mit Liste)
template <typename KeyType, typename ValueType>
struct Dictionary {
    struct Entry { // lokale Definition erlaubt (Name außerhalb ist Dictionary::Entry)
        ValueType val;
        KeyType key;
    };
    List<Entry> allEntries; // Speichert alle Einträge
    const ValueType &operator[](const KeyType &key) const {
        for (int i=0; i<allEntries.size(); i++) {
            if (allEntries[i].key == key)
                return allEntries[i].value;
        }
        throw SomeException(...);
    }
};
```

// read-only (Instanz nicht beschreiben)
// kann überladen werden! (r/w extra)



// Benutzung
Dictionary<string, int> a;
...
cout << a["Hello"];

Integer-Parameter

Containerklasse: Listen

```
template <typename T, unsigned int size>
struct FixedList { // Speicher reservieren
    T memory[size];

    // Konstruktor nicht nötig
    // Speicher ist bereits reserviert!
    FixedList() {}
    // Destruktor auch nicht nötig
    ~FixedList() {}

    // Zugriff auf Elemente
    T &operator[](unsigned int index) {...}

    // Copy-Constructor
    FixedList(const FixedList<T,size> &other) {...}

    // Zuweisungsoperator
    void operator=(const FixedList<T,size> &other) {...}
}
```

```
// Benutzung
FixedList<double, 3> threeDVector;

// Man kann auch Namen vergeben
// (geht mit allen Typen!)
typedef FixedList<float,3> Vector3f;
typedef FixedList<double,3> Vector3d;

Vector3f v = Vector3f();
```

Python & Scala?

Gibt es „templates“ in Python?

- Eigentlich nicht notwendig...
- Jedes Unterprogramm ist generisch!

```
def square(a):  
    return a*a
```

 - Dynamisch typisiert
- What about MyPy?
 - Ja, Typparameter werden unterstützt
 - Mit Type-Bounds: Anforderungen an Typen via Subtyping
 - In Scala ähnlich
 - In C++ via „Concepts“ (seit C++20, eher kompliziert)
- Details in Foliensatz 12