# 1 Typumwandlung

- **Implicit Casting**: Automatische Typumwandlung durch den Compiler.
  - `int a = 5.4;` $\Longrightarrow$ a wird zu einem int (5)
  - `float b = 7/2;` $\Longrightarrow$ Ganzzahlige Division, Ergebnis 3 wird zu double (3.0)
  - `float c = 7/2.0;` $\Longrightarrow$ Einer der Werte ist float, Ergebnis 3.5
  - `double d = 'A'- 12;` $\Longrightarrow$ char wird zu int (65), dann -12 (53), dann zu double (53.0)
  - `int e = true + 3;` $\Longrightarrow$ bool wird zu int (1) + 3 (4), dann zu int (4)
  - Allgemein: *Der kleinere Typ wird in den größeren umgewandelt*
- **Explicit Casting**: Manuelle Typumwandlung durch den Programmierer.
  - `int x = (int)3.7;` $\Longrightarrow$ Klassischer Cast: Ergebnis ist 3
  - `int y = static_cast<int>(3.7);` $\Longrightarrow$ Moderner Cast mit `static_cast`: Ergebnis ist ebenfalls 3

# 2 Hierarchie von Operatoren

| Priorität | Operator | Beschreibung |
|---|---|---|
| **Hoch** | `! * &` | **Unär:** Log. NICHT, Deref., Adresse |
| ↓ | `* /` | **Binär:** Multiplikation, Division |
| ↓ | `+ -` | **Binär:** Addition, Subtraktion |
| ↓ | `<< >>` | **Binär:** Bit-Shift Links/Rechts |
| ↓ | `&` | **Binär:** Bitweises UND |
| ↓ | `\|` | **Binär:** Bitweises ODER |
| ↓ | `&&` | **Binär:** Logisches UND |
| **Niedrig** | `\|\|` | **Binär:** Logisches ODER |

# 3 Wertebereiche von Datentypen

| Datentyp | Bytes | Wertebereich |
|---|---|---|
| `bool` | 1 | `true` oder `false` |
| `char` | 1 | -128 bis 127 |
| `unsigned char` | 1 | 0 bis 255 |
| `short` | 2 | -32.768 bis 32.767 |
| `unsigned short` | 2 | 0 bis 65.535 |
| `int` | 4 | -2.147.483.648 bis 2.147.483.647 |
| `unsigned int` | 4 | 0 bis 4.294.967.295 |
| `long long` | 8 | ca. -9,2 x $10^{18}$ bis 9,2 x $10^{18}$ |
| `float` | 4 | ca. ±3.4 x $10^{38}$ (7 Dezimalstellen) |
| `double` | 8 | ca. ±1.8 x $10^{308}$ (15 Dezimalstellen) |

# 4 Overflow von Zahlen

Overflow = Zugewiesene oder berechnete Zahl liegt außerhalb des darstellbaren Bereichs eines Datentyps.

- **Ganzzahlen**: Undefiniertes Verhalten. z.B. zu hohe Bits werden abgeschnitten oder es wird auf den Minimalwert zurückgesetzt.
- **Gleitkommazahlen**: Im IEEE 754 Standard wird bei Overflow der Wert `inf` (unendlich) zugewiesen.

# 5 Definition und Deklaration

- **Definition**: Reserviert Speicherplatz für eine Variable oder Funktion und kann optional initialisiert werden.
  - Beispiel Variable: `int x = 5;`
- **Deklaration**: Informiert den Compiler über den Typ und Namen einer Variable oder Funktion, reserviert aber keinen Speicherplatz.
  - Beispiel Variable: `int x;`
  - Beispiel Funktion: `void foo();`
- **Prototyp**: Funktionsdeklaration ohne Funktionskörper.
  - Beispiel: `int map(double[], int, int (*)(double));`
- **Wichtig**: *Jede Definition ist auch eine Deklaration!*

# 6 String und Vector API

| Typ | Methode | Beschreibung |
|---|---|---|
| `string/vector` | `.size()` / `.length()` | Gibt die Anzahl der Elemente bzw. die Länge zurück |
| `string/vector` | `.empty()` | Prüft, ob leer |
| `string/vector` | `.clear()` | Löscht den Inhalt |
| `string` | `.append(str)` | Fügt `str` am Ende an (auch `+=` möglich) |
| `vector` | `.push_back(val)` | Fügt `val` am Ende hinzu |
| `vector` | `.pop_back()` | Entfernt das letzte Element |
| `string/vector` | `.at(idx)` | Gibt Element/Zeichen an Position `idx` zurück |
| `string/vector` | `.front()/ .back()` | Erstes/letztes Element zurückgeben |
| `string/vector` | `.begin()/ .end()` | Iteratoren auf Anfang/Ende |
| `string` | `.substr(start, len)` | Teilstring ab `start` mit Länge `len` |
| `string` | `.find(str)` | Sucht nach `str` und gibt Startposition zurück |

# 7 Nützliche std:: Funktionen

Benötigt `#include <algorithm>` und `#include <functional>`

| Methode | Beschreibung |
|---|---|
| `std::sort(b, e)` | `void` Sortiert einen Bereich |
| `std::find(b, e, v)` | `Iterator` Sucht einen Wert im Bereich |
| `std::reverse(b, e)` | `void` Dreht die Reihenfolge im Bereich um |
| `std::max(a, b)` | `T` Gibt das größere von zwei Werten zurück |
| `std::find_if(b, e, p)` | `Iterator` Sucht das erste Element, das das Prädikat erfüllt |
| `std::count_if(b, e, p)` | `int` Zählt Elemente, die das Prädikat erfüllen |
| `std::all_of(b, e, p)` | `bool` Prüft, ob alle Elemente das Prädikat erfüllen |
| `std::any_of(b, e, p)` | `bool` Prüft, ob mindestens ein Element das Prädikat erfüllt |
| `std::transform(b, e, d, f)` | `void` Wendet Funktion `f` auf alle Elemente an und speichert sie in `dest` |
| `std::max_element(b, e)` | `Iterator` Gibt den Iterator auf das größte Element im Bereich zurück |
| `std::min_element(b, e)` | `Iterator` Gibt den Iterator auf das kleinste Element im Bereich zurück |

- `b` = `begin()`, `e` = `end()`
- `p` = Prädikat (Funktion, die bool zurückgibt) z.B. `[](int x){return x>5;}`
- `v` = Wert, der gesucht wird
- `d` = Zieliterator (z.B. Anfang eines anderen Containers)
- `f` = Funktion, die auf jedes Element angewendet wird (z.B. `[](int x){return x*2;}`)

# 8 Konventionen

- **Zugriffsmodifikatoren**: Reihenfolge: `public:`, `protected:`, `private:`
- **Konstruktoren**: Immer Explicit angeben
- **Destruktoren**: Immer Virtual angeben, wenn die Klasse vererbt wird
- Membervariablen: Immer mit `m_` oder `_m` kennzeichnen. Keine gleichen Namen wie Parameter im Konstruktor verwenden.
- **Funktionen / Methoden**: Nicht komplett inline definieren: `int add(int a, int b){return a+b}`
- **Void als Parameter**: Nie `void` als Parameter verwenden: `int foo(void);`

# 9 Objektorientierung

- **Konstruktor / Destruktor**: Konstruktoren werden in verschachtelten Klassen von der innersten zur äußersten Klasse aufgerufen. Destruktoren in umgekehrter Reihenfolge.
- **Virtual / Override**: Virtuelle Funktionen werden in der Basisklasse mit `virtual` deklariert und in der abgeleiteten Klasse mit `override` überschrieben.
  - Wenn eine Methode als `virtual` deklariert ist, wird zur Laufzeit die passende Methode der abgeleiteten Klasse aufgerufen, auch wenn der Zeiger oder die Referenz den Typ der Basisklasse hat.
  - Wenn eine Methode nicht als `virtual` deklariert ist, wird die Methode abhängig vom Typ des Zeigers oder der Referenz aufgerufen (statischer Bindung).
- **Final**: Mit `final` kann verhindert werden, dass eine Klasse weiter vererbt wird oder eine Methode überschrieben wird.

# 10 Smart Pointer

Smart Pointer sind Klassen, die die Verwaltung von dynamisch allozierten Objekten übernehmen und automatisch den Speicher freigeben, wenn der Pointer nicht mehr benötigt wird.

- `std::unique_ptr<T>`: Besitzt ein Objekt exklusiv. Kann nicht kopiert, nur verschoben werden. Nutzt `std::move()` zum Übertragen des Besitzes.
- `std::shared_ptr<T>`: Teilt den Besitz eines Objekts mit anderen `shared_ptr`s. Verwendet Referenzzählung, um zu wissen, wann das Objekt gelöscht werden kann.

## make_shared / make_unique

Empfohlene Methode zur Erstellung von Smart Pointern, da sie effizienter und sicherer ist als die direkte Verwendung von `new`.

- `auto ptr = std::make_unique<T>();`: Erstellt einen `unique_ptr` zu einem neuen Objekt vom Typ `T`.
- `auto ptr = std::make_shared<T>();`: Erstellt einen `shared_ptr` zu einem neuen Objekt vom Typ `T`.

## std::move

std::move ist ein Cast, der ein Objekt als "bewegbar" markiert; dies erlaubt dem Compiler, statt einer teuren Kopie eine schnelle Ressourcen-Übernahme durchzuführen, wobei das Originalobjekt in einem gültigen, aber unbestimmten Zustand zurückbleibt und sicher am Ende seines Gültigkeitsbereichs zerstört wird (Wird oft bei `unique_ptr` verwendet).

# 11 Functional und Lambda

Benötigt `#include <functional>`

`std::function<T>` ist nützlich um Funktionen als Objekt zu deklarieren, speichern und übergeben zu können. **Beispiel**:

```
std::function<int(int,int)> sum = [](int a, int b){
return a + b; };
```

## Lambda Funktionen

Lambda Funktionen sind anonyme Funktionen, die direkt im Code definiert werden können. Sie haben die folgende Syntax:

```
[capture](parameters)-> return_type { body }
```

- **Capture**: Bestimmt, welche Variablen aus dem umgebenden Kontext verwendet werden können.
  - `[]`: Keine Variablen werden erfasst.
  - `[=]`: Alle Variablen werden per Wert erfasst.
  - `[&]`: Alle Variablen werden per Referenz erfasst.
  - `[x, &y]`: Variable `x` wird per Wert und `y` per Referenz erfasst.
- **Parameters**: Die Parameter der Lambda Funktion, ähnlich wie bei normalen Funktionen.
- **Return Type**: Der Rückgabetyp der Funktion. Kann oft weggelassen werden, da der Compiler ihn ableiten kann.

- **Body**: Der eigentliche Code der Funktion, eingeschlossen in geschweifte Klammern.